



Lecture 1: ISA I

CS10014 Computer Organization

Tsung Tai Yeh

Department of Computer Science
National Yang Ming Chiao University



Acknowledgements and Disclaimer

- Slides were developed in the reference with
 - CS 61C at UC Berkeley
 - <https://inst.eecs.berkeley.edu/~cs61c/sp23/>
 - CS 252 at UC Berkeley
 - <https://people.eecs.berkeley.edu/~culler/courses/cs252-s05/>
 - E85 at HMC
 - <https://pages.hmc.edu/harris/class/e85/old/fall21/>



Outline

- Introduction
 - Computer Architecture
 - Instruction Set Architecture (ISA)
- RISC-V Assembly Language
 - Instructions
 - Register Set
 - Memory
 - Programming constructs



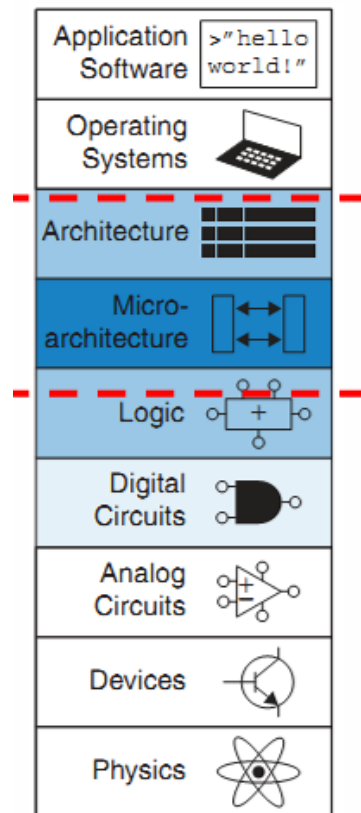
What You Will Learn

- How programs are translated into the machine language
 - How the hardware execute them
- The hardware/software interface
- What determines program performance
- How hardware designers improve performance
- What is parallel processing



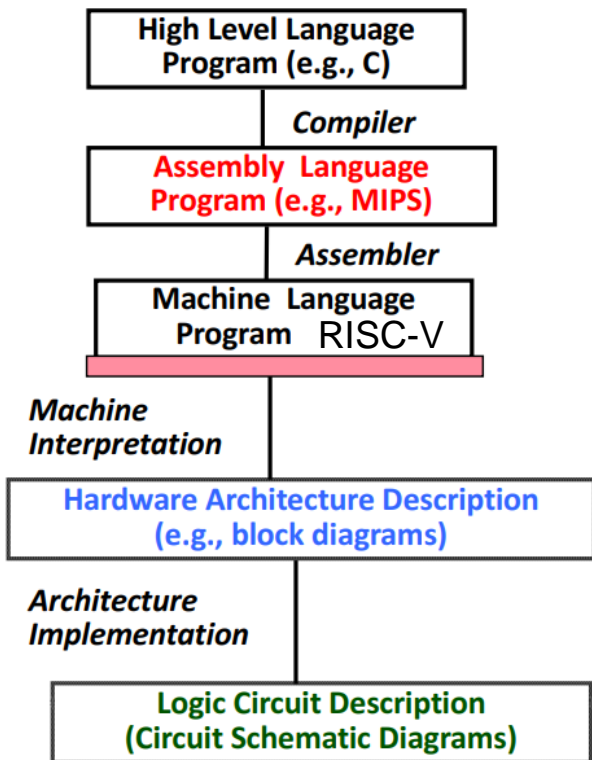
What is “Computer Architecture” ?

- Covers three aspects of computer design
 - **Instruction set architecture**
 - Software and hardware interfaces
 - **Organization or microarchitecture**
 - CPU, memory, cache architecture
 - **Hardware**
 - Computer systems, e.g., I/O devices





Abstractions of Computer Architecture

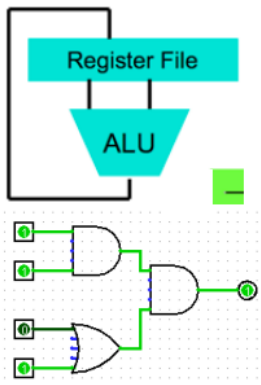


```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

```
lw $t0, 0($2)  
lw $t1, 4($2)  
sw $t1, 0($2)  
sw $t0, 4($2)
```

Anything can be represented
as a number,
i.e., data or instructions

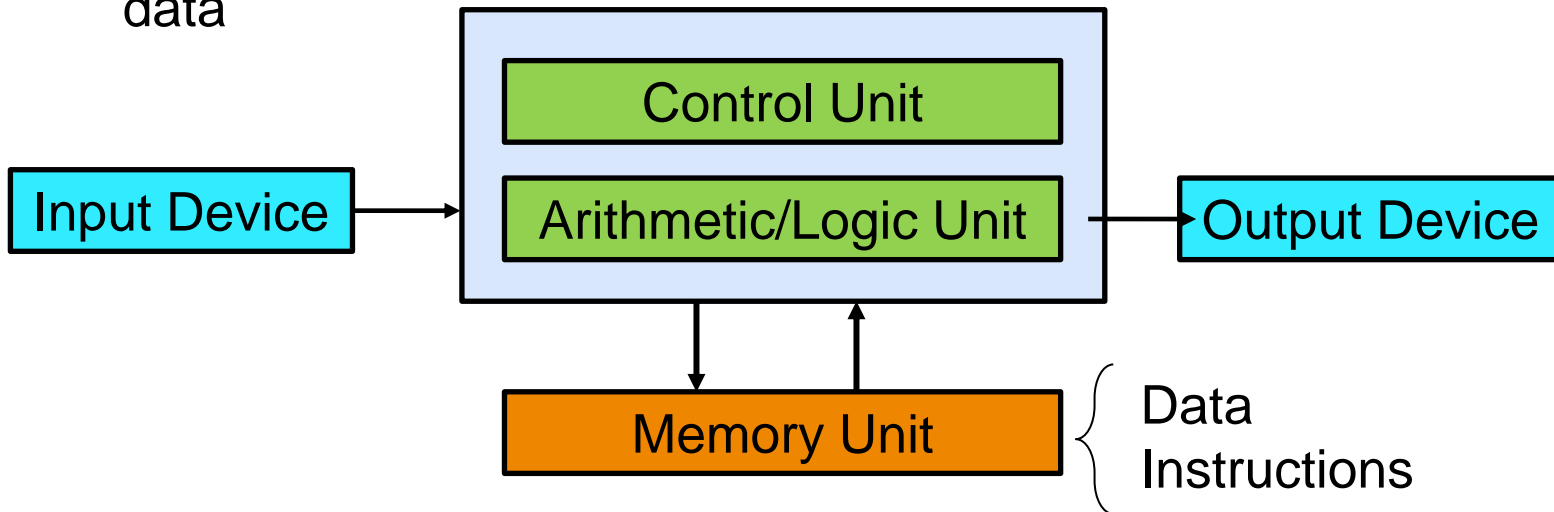
```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```





Von Neumann Machine

- **Stored-program computer** (Five key components)
 - Instructions and data are stored in the same memory
 - Instructions are fetched over the same data path used to fetch data





Power of the Stored-Program Computer

- 32-bit instructions & data stored in memory
- Sequence of instructions: only difference between two applications
- To run a new program:
 - No rewiring required
 - Simply store new program in memory
- Program Execution:
 - Processor *fetches* (reads) instructions from memory in sequence
 - Processor performs the specified operation



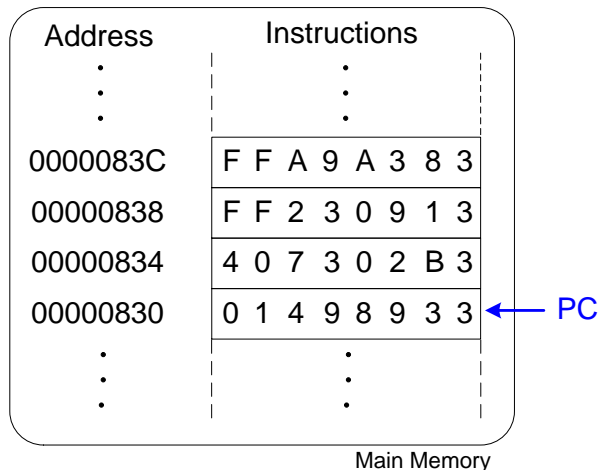
The Stored Program

Assembly Code

Machine Code

add s2, s3, s4	0x01498933
sub t0, t1, t2	0x407302B3
addi s2, t1, -14	0xFF230913
lw t2, -6(s3)	0xFFA9A383

Stored Program



Program Counter (PC): a special register that keeps track of the **memory address** of the next instruction to be executed in a program



Instruction Set Architecture (ISA)

- The instruction set
 - The repertoire of instructions of a computer
 - Different computers have different instruction sets
 - Early computers had very simple instructions
 - RISC vs CISC

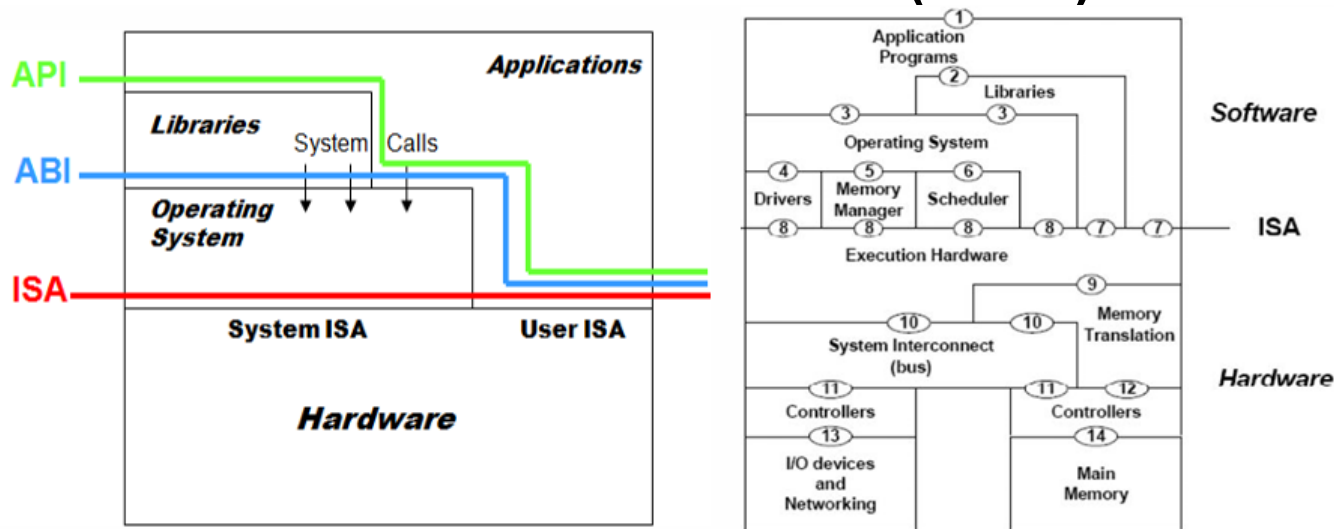


RISC vs CISC

- **Make the common case fast**
 - RISC-V includes only simple, commonly used instructions
 - Hardware to decode and execute instructions can be simple, small, and fast
 - More complex instructions performed using multiple simple instructions
 - RISC-V is a **reduced instruction set computer (RISC)** with a small number of simple instructions.
 - Other architectures, such as Intel's x86, are **complex instruction set computers (CISC)**



Instruction Set Architecture (ISA)



- **API:** application programming interface
- **ABI:** application binary interface = system call + user ISA
- **ISA:** formal specification of a system's interface and the logical behavior of its visible resources



RISC-V Instruction Sets

- **Instructions:**
 - Commands in a computer's language
 - Assembly language: human-readable format of instructions
 - Machine language: computer-readable format (1's and 0's)
- **RISC-V architecture:**
 - First open-source computer architecture
 - RISC-V ISA manual
 - <https://riscv.org/specifications/isa-spec-pdf/>



RISC-V 32-bit ISA

- **RISC-V Base ISAs:**

Name	Description
RV32I	32-bit integer instruction set
RV32E	32-bit integer instruction set for embedded microprocessors
RV64I	64-bit integer instruction set
RV128I	128-bit integer instruction set



RISC-V 32-bit ISA

- **RISC-V Extensions:**

Suffix	Description
M	Standard extension for integer multiplication and division
A	Standard extension for atomic instruction
F	Standard extension for single-precision Floating Point
D	Standard extension for double-precision Floating Point
C	Standard extension for compressed instructions
B	Standard extension for bit manipulation
P/V	Standard extension for packed-SIMD/vector instructions



RISC-V 32-bit ISA

- **RISC-V 32-bit ISAs:**
 - It supports 32-bit address spaces
 - It contains thirty-three 32-bit registers
 - It represents signed integer values in two's complement
 - It contains integer computational/loads/stores instructions, and control-flow instructions
 - It contains instructions to multiply and divide values held in the integer registers (M extension)



Arithmetic Operations of the Computer Hardware



Instructions: Addition

C Code

```
a = b + c;
```

RISC-V assembly code

```
add a, b, c
```

- **RISC-V assembly code**
 - **add**: mnemonic indicates operation to perform (**opcode**)
 - **b, c**: source operands (on which the operation is performed) (**rs1,2**)
 - **a**: destination operand (to which the result is written) (**rd**)
 - All arithmetic operations have this form



Multiple Instructions

C Code

```
a = b + c - d;
```

RISC-V assembly code

```
add t, b, c # t = b + c  
sub a, t, d # a = t - d
```

- More complex code is handled by multiple RISC-V instructions



Signed and Unsigned Number



Sign Magnitude Representation

- The Most Significant bit (MSB) of the number is a sign bit
- The remaining bit represents the magnitude of the number in a binary form

MSB **Magnitude**
0 0 1 0 0 0 1 0

- Example: 8-bit sign-magnitude form

+34 = 0 0 1 0 0 0 1 0

-34 = 1 0 1 0 0 0 1 0

Using n-bits, the range of numbers is from $-(2^{n-1})$ to $(2^{n-1} - 1)$



1's Complement Representation

- The representation of the negative number is different from the positive number representation
- Example: The represent -34 in 1's complement form

$$\begin{array}{rcccccccc} +34 = & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ -34 = & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \end{array}$$

Invert all 1s in that number by 0s and 0s by 1s



2's Complement Representation

- Translate negative number from 1's complement to 2's complement form
 - Write the number corresponding to +34
 - Find 1's complement of +34
 - Add 1 to the 1's complement number

$$\begin{array}{r} +34 = 00100010 \\ \quad \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \\ \quad 11011101 \quad \text{(1's complement of +34)} \\ \quad + \quad \quad \quad 1 \\ \hline -34 = 11011110 \quad \text{(2's complement of +34)} \end{array}$$



Takeaway Questions

- What is the correct binary representation of “ -2_{ten} ” using the 2’s complement method?
 - (A) 00000000 00000000 00000000 00000010_{two}
 - (B) 11111111 11111111 11111111 11111101_{two}
 - (C) 11111111 11111111 11111111 11111110_{two}



Understanding of overflow

- Carry indicates overflow

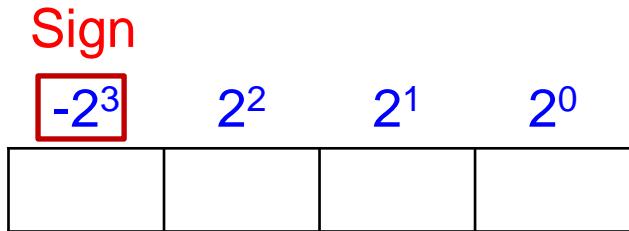
$$\begin{array}{rcccccl} & 1 & 0 & 1 & 1 & 7(\text{DEC}) \\ + & 0 & 1 & 1 & 1 & 11(\text{DEC}) \\ \hline \boxed{1} & 0 & 0 & 1 & 0 & 19(\text{DEC}) \end{array}$$

Overflow -> 19 is out of the range of the 4-bit value representation (0-15)

A	B	C	D	Unsigned
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15



Overflow in Signed numbers (2's Complement)



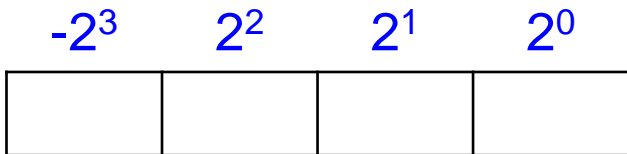
The range of 4-bits signed number
 $-2^{n-1} \leftrightarrow (2^{n-1} - 1) \implies -8 \leftrightarrow 7$

	1	0	0	1	-7(DEC)	
+	1	1	0	1	-3(DEC)	
<hr/>						
	1	0	1	1	0	-10(DEC)

Overflow !



Overflow in Signed numbers (2's Complement)



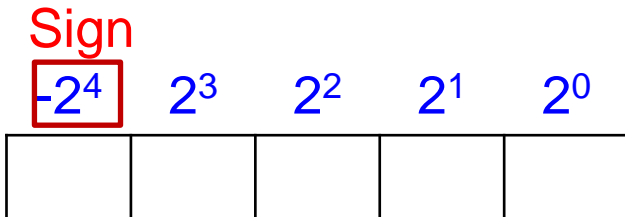
The range of 4-bits signed number
 $-2^{n-1} \leftrightarrow (2^{n-1} - 1) \implies -8 \leftrightarrow 7$

	0	1	1	1	7(DEC)
+	0	0	0	1	1(DEC)
	<hr/>				
	1	0	0	0	-8(DEC)

Overflow ! How to fix this problem?



Overflow in Signed numbers (2's Complement)



The range of 5-bits signed number
 $-2^{n-1} \leftrightarrow (2^{n-1} - 1) \implies -16 \leftrightarrow 15$

	0	0	1	1	1	7(DEC)
+	0	0	0	0	1	1(DEC)
	<hr/>					
	0	1	0	0	0	8(DEC)

Extend 4-bit value to 5 bits to hold
the correct result



What is sign extension?

- Sign-extension
 - Represent a number using more bits
 - Replicate the sign bit to the left
 - Unsigned values: extend with 0s
 - Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110
 - In RISC-V instruction set
 - 1b: sign-extend loaded byte
 - 1bu: zero-extend loaded byte



What is sign extension?

- 8-bit encoding of decimal **signed number** -56 can be sign-extended as follows:

Sign

000111000 <- 8-bit value of 56

11000111 <- 8-bit value of -56 (1's complement)

11001000 <- 8-bit value of -56 (2's complement)

11111111 11001000 <- extended to 16-bit value

11111111 11111111 11111111 11001000 <- extended 32-bit value



Operands of the Computer Hardware



Operands

- **Operand location: physical location in computer**
 - Registers
 - Memory
 - Constants (also called immediates)



Operands: Registers

- RISC-V has 32 32-bit registers
- Registers are faster than memory
- RISC-V called “32-bit architecture” because it operates on 32-bit data
 - RISC-V also has 64-bit spec.
- RISC-V includes only a small number register
 - Smaller is faster



RISC-V Registers

x0 (Zero Register), **a0** (Argument Register 0),
while a0 is a variable

Name	Register Number	Usage
zero	x0	Constant value 0
<u>ra</u>	x1	Return address
<u>sp</u>	x2	Stack pointer
<u>gp</u>	x3	Global pointer
<u>tp</u>	x4	Thread pointer
t0-2	x5-7	Temporaries
<u>s0</u>/<u>fp</u>	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries



Operands: Registers

- **Registers:**
 - Can use either name (i.e., ra, zero) or x0, x1, etc.
 - Using name is preferred
- Registers used for **specific purposes:**
 - Zero always holds the **constant value 0**
 - The remaining registers (x1-x31) are general purpose register and can be used interchangeably
 - The **saved registers**, s0-s11, used to hold variables
 - The **temporary registers**, t0-t6, used to hold intermediate values during a larger computation



Operands: Memory

- RV32I native datatypes and their respective sizes in bytes

RV32I native data type name	Size in bytes
Byte	1
Unsigned byte	1
Halfword	2
Unsigned halfword	2
Word	4
Unsigned word	4



Memory

- RISC-V is a Load/Store architecture
 - Requires values to be loaded/stored explicitly from/to memory before operating on them
 - Requires the data to be first retrieved from memory into a register by executing a load instruction
- First, we'll discuss **word-addressable memory**
- Then, we will discuss **byte-addressable memory**

RISC-V is **byte-addressable**

```
lw    a5, 0(a0)
add   a6, a5, a5
sw    a6, 0(a0)
```



Word-Addressable Memory

- Each 32-bit data word has a unique address
 - 1 word = 4 bytes

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

1 byte (8 bits)



Reading Word-Addressable Memory

- Memory read called **load**
- Mnemonic: load word (lw)
- Format: `lw <rd>, <offset>(<base register>)`
`lw t1, 5(s0)`
- Address calculation
 - Add offset (5) to the base address (s0)
 - Address = (s0 + 5)
- Destination register (rd)
 - t1 holds the value at address (s0 + 5)
 - Any register may be used as based address



Reading Word-Addressable Memory

- Example:
 - read a word of data at memory address 1 into s3
 - Address = $(0 + 1) = 1$
 - S3 = 0xF2F1AC07 after load

Assembly code

```
lw s3, 1(zero) # read memory word 1 into s3
```

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0



Writing Word-Addressable Memory

- Memory write are called **store**
- Mnemonic: store word (**sw**)
- Format similar to load

```
sw <src1>, <offset>(<base register>)
```



Writing Word-Addressable Memory

- Example:
 - Write (store) the value in t4 into memory address 7
 - t4 = 0x12345678
 - Add the base address (zero) to the offset (0x7)
 - Address: $(0 + 0x7) = 7$
 - Offset can be written in decimal (default) or hexadecimal

Assembly code

```
sw t4, 0x7(zero) # write the value in t4  
# to memory word 7
```

Word Address	Data	
⋮	12345678	Word 7
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

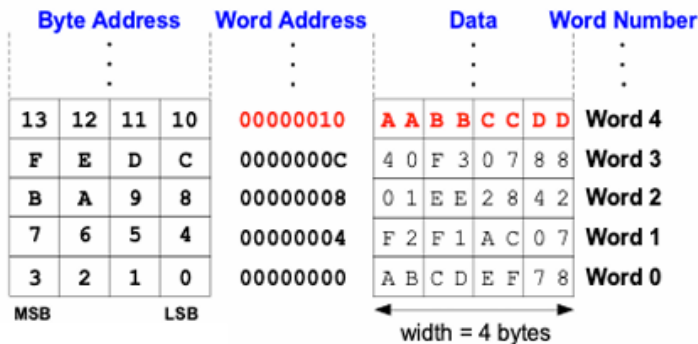


Writing Byte-Addressable Memory

- Example
 - Store the value held in t7 into memory address 0x10 (16)
 - If t7 holds the value 0xAABBCDD, then after the sw completes, word 4 (at address 0x10) in memory will contain that value

RISC-V assembly code

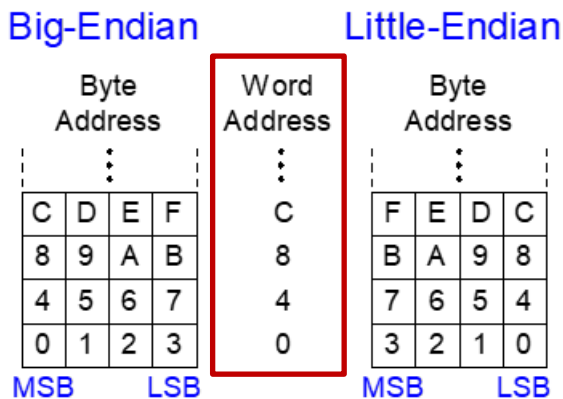
```
SW t7, 0x10(zero) # write t7 into address 16
```





Big-Endian & Little-Endian

- **Little-endian**
 - Byte numbers start at the little (least significant) end
- **Big-endian**
 - Byte numbers start at the big (most significant) end
 - Word address is the same for big- or little-endian





Big-Endian & Little-Endian Example

- Suppose t0 initially contains 0x23456789
 - After following code runs on **big-endian system**, what value is s0?
 - In a **little-endian system**?

```
sw t0, 0(zero)
```

```
lb s0, 1(zero)
```



Big-Endian & Little-Endian Example

- Big-endian

- s0 = 0x00000045

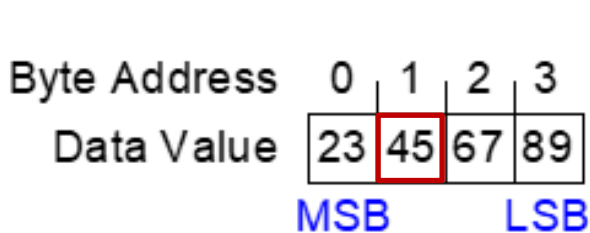
- Little-endian (**RISC-V**)

- s0 = 0x00000067

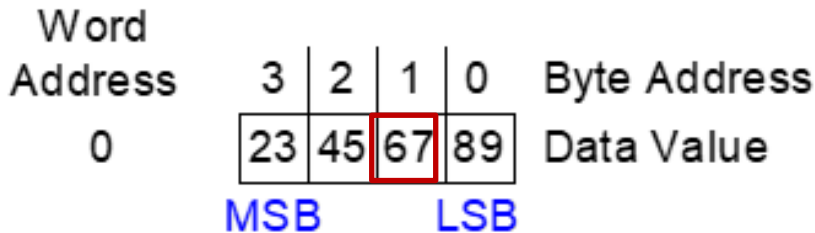
```
sw t0, 0(zero)
```

```
lb s0, 1(zero)
```

Big-Endian



Little-Endian





Loading and Storing Bytes

- RISC-V has **byte** data transfers:

- Load byte: **lb**
- Store byte: **sb**

- For example

- `addi x11, x0, 0x3f5`
`sw x11, 0(x5)`
`lb x12, 1(x5)`
- What is the value in x12?
 - Note that 0x3f5 (HEX) =
0011 1111 0101 (BIN)
3 **f** **5**
0x3f5 = 1013 (DEC)

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



Takeaway Questions

- What is the value in x12?
 - (A) 0x8
 - (B) 0xf8
 - (C) 0xffffffff8

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



Takeaway Questions

- What is the value in x12?
 - (A) 0x8
 - (B) 0xf8
 - (C) 0xffffffff8

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

Sign

0x8f5 \Leftrightarrow 1000 1111 0101 (2' complement) \Leftrightarrow -779(DEC)

1000 1111 0101 (2' complement) \rightarrow -779

1000 1111 0100 (1' complement)

0111 0000 1011 (unsigned 779)



Takeaway Questions

- What is the value in x12?
 - (A) 0x8
 - (B) 0xf8
 - (C) 0xffffffff8

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

Sign

0x8f5 \Leftrightarrow 1000 1111 0101 (2' complement) \Leftrightarrow -779(DEC)

1111 1111 1111 1111 1111 1000 1111 0101 (Signed extend
0x8f5 to 32-bits) \Rightarrow 0xffffffff8f5



Takeaway Questions

- What is the value in x12?
 - (A) 0x8
 - (B) 0xf8
 - (C) 0xffffffff8

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

- **addi x11, x0, 0x8f5**
- The immediate got sign extended, x11 is 0xffff8f5 because x11 is signed 32-bit register
- **sw x11, 0(x5)**
- the value of x11 is copied to x5 = 0xffff8f5



Takeaway Questions

- What is the value in x12?
 - (A) 0x8
 - (B) 0xf8
 - (C) 0xffffffff8

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

- **lb x12, 1(x5)**
- Load byte sign extend to the register
- $0(x5) = 0xf5$
- **$1(x5) = 0xffffffff8$**



Logical Operations



Logical Instructions

- **and, or, xor**
 - **and**: useful for **masking** bits
 - Masking all but the least significant byte of a value:
 $0xF234012F \text{ AND } 0x000000FF = 0x0000002F$
 - **or**: useful for combining bit fields
 - Combine $0xF2340000$ with $0x000012BC$
 $0xF2340000 \text{ OR } 0x000012BC = 0xF23412BC$
 - **xor**: useful for inventing bits:
 - $A \text{ xor } -1 = \text{NOT } A$ (remember that $-1 = 0xFFFFFFFF$)

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0



Logical Instructions Example

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Source Registers

s1	0100 0110	1010 0001	1111 0001	1011 0111
s2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly Code

```
and s3, s1, s2  
or  s4, s1, s2  
xor s5, s1, s2
```

Result

s3	0100 0110	1010 0001	0000 0000	0000 0000
s4	1111 1111	1111 1111	1111 0001	1011 0111
s5	1011 1001	0101 1110	1111 0001	1011 0111



Logical Instructions Example

Source Values

t3	0011	1010	0111	0101	0000	1101	0110	1111
imm	1111	1111	1111	1111	1111	1010	0011	0100

← sign-extended →

Assembly Code

```
andi s5, t3, -1484
ori s6, t3, -1484
xori s7, t3, -1484
```

Result

s5	0011	1010	0111	0101	0000	1000	0010	0100
s6	1111	1111	1111	1111	1111	1111	0111	1111
s7	1100	0101	1000	1010	1111	0111	0101	1011

1484 =
010111001100
-1484 =
1's complement
101000110011
2's complement
101000110100

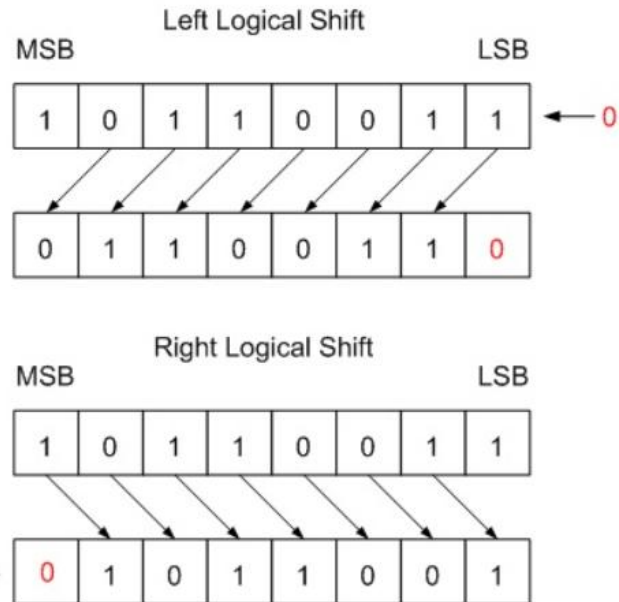
-1484 = **0xA34** in 12-bit 2's complement representation.



Shift Instructions

- **Logical shift**
 - Correspond to (left-shift) **multiplication by 2**, (right-shift) **integer division by 2**.
 - **sll: shift left logical**
 - Example: `sll t0, t1, t2 # t0 = t1 << t2`
 - **srl: shift right logical**
 - Example: `srl t0, t1, t2 # t0 = t1 >> t2`

```
and    a0, a2, a6    # a0 := a2 & a6
slli   a1, s3, 2     # a1 := a3 << 2
sub    a4, a5, a6    # a4 := a5 - a6
```





Shift Instructions

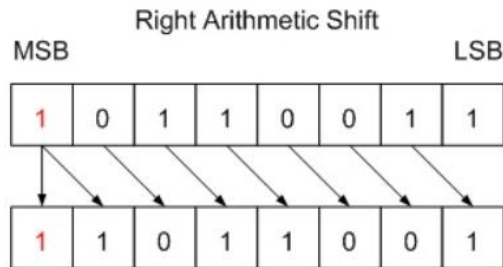
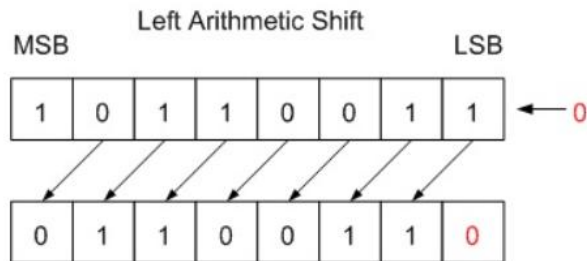
- **Arithmetic shift**

- The sign is the leftmost bit, then arithmetic shift preserves the sign (this is called sign extension).

- **sra: shift right arithmetic**

- Example: `sra t0, t1, t2 # t0 = t1 >>> t2`

Arithmetic shift	
Left shift	Pad 0
Right shift	Keep sign





Immediate Shift Instructions

- Shift amount is an immediate between 0 to 31
- **slli: shift left logical immediate**
 - Example: `slli t0, t1, 23 # t0 = t1 << 23`
- **srlr: shift right logical immediate**
 - Example: `srlr t0, t1, 18 # t0 = t1 >> 18`
- **srai: shift right arithmetic immediate**
 - Example: `srai t0, t1, 5 # t0 = t1 >>> 5`



Instructions for Making Decisions



Control-flow Instructions

- **if** statements
- **if/else** statements
- **while** loops
- **for** loops



Branching

- **Conditional branches**
 - branch if equal (beq)
 - branch if not equal (bne)
 - branch if less than (blt/bltu)
 - branch if greater than or equal to (bge/bgeu)
- **Unconditional branches**
 - jump (j)
 - jump register (jr)
 - jump and link (jal)
 - jump and link register (jalr)



Conditional Branching (beq)

RISC-V assembly

```
addi s0, zero, 4      # s0 = 0 + 4 = 4
addi s1, zero, 1      # s1 = 0 + 1 = 1
slli s1, s1, 2        # s1 = 1 << 2 = 4
beq  s0, s1, target  # branch is taken
addi  s1, s1, 1      # not executed
sub    s1, s1, s0      # not executed

target:                # label
    add  s1, s1, s0    # s1 = 4 + 4 = 8
```

- Label indicates instruction location. They can't be reserved words and must be followed by colon (:)



The Branch Not Taken (bne)

RISC-V assembly

```
addi s0, zero, 4           # s0 = 0 + 4 = 4
addi s1, zero, 1           # s1 = 0 + 1 = 1
slli s1, s1, 2             # s1 = 1 << 2 = 4
bne  s0, s1, target       # branch not taken
addi s1, s1, 1             # s1 = 4 + 1 = 5
sub    s1, s1, s0           # s1 = 5 - 4 = 1
```

target:

```
add    s1, s1, s0           # s1 = 1 + 4 = 5
```



Other Conditional Branches

- Branch if less than (blt/bltu)

```
blt s0, s1, target    # branches if s0 < s1 (signed)
bltu s0, s1, target  # same as blt but interprets
                       # s0 and s1 as unsigned
```

- Branch if great than (bge/bgeu)

```
bge s0, s1, target    # branches if s0 > s1 (signed)
bgeu s0, s1, target  # branches if s0 > s1 (unsigned)
```



Unconditional Branching (j)

RISC-V assembly

```
j          target          # jump to target
          srai      s1, s1, 2  # not executed
          addi     s1, s1, 1  # not executed
          sub      s1, s1, s0  # not executed
```

target:

```
add      s1, s1, s0      # s1 = 1 + 4 = 5
```



If Statement

C Code

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

```
f = f - i;
```

RISC-V assembly code

```
# s0 = f, s1 = g, s2 = h
```

```
# s3 = i, s4 = j
```

```
    bne s3, s4, L1
```

```
    add s0, s1, s2
```

```
L1:
```

```
    sub s0, s0, s3
```

Assembly tests opposite case (i != j) of high-level code (i == j)



If/Else Statement

C Code

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

```
else
    f = f - i;
```

RISC-V assembly code

```
# s0 = f, s1 = g, s2 = h
```

```
# s3 = i, s4 = j
```

```
    bne s3, s4, L1
```

```
    add s0, s1, s2
```

```
    j   done
```

```
L1:
```

```
    sub s0, s0, s3
```

```
done:
```



While loops

C Code

```
// determines the power
// of x such that 2x = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

RISC-V assembly code

```
# s0 = pow, s1 = x

    addi s0, zero, 1
    add  s1, zero, zero
    addi t0, zero, 128

while:
    beq  s0, t0, done
    slli s0, s0, 1
    addi s1, s1, 1
    j    while
done:
```

Assembly tests for the opposite case (pow == 128) of the C code (pow != 128).



For loops

- **for (initialization; condition; loop operation)**
 - **Statement**
- **Initialization:** executes before the loop begins
- **Condition:** is tested at the beginning of each iteration
- **Loop operation:** Executes at the end of each iteration
- **Statement:** executes each time the condition is met



For loops

C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

RISC-V assembly code

```
# s0 = i, s1 = sum
    addi s1, zero, 0
    add  s0, zero, zero
    addi t0, zero, 10

for:
    beq  s0, t0, done
    add  s1, s1, s0
    addi s0, s0, 1
    j    for

done:
```



Less Than Comparison

C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
    sum = sum + i;
}
```

RISC-V assembly code

```
# s0 = i, s1 = sum
    addi s1, zero, 0
    addi s0, zero, 1
    addi t0, zero, 101

loop:
    bge s0, t0, done
    add s1, s1, s0
    slli s0, s0, 1
    j loop

done:
```



Unconditional Branching (j)

- j (jump)
 - Perform an unconditional jump to a specified memory address.
 - Used for implementing loops, conditional statements

C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
    sum = sum + i;
}
```

RISC-V assembly code

```
# s0 = i, s1 = sum
    addi s1, zero, 0
    addi s0, zero, 1
    addi t0, zero, 101

loop:
    bge  s0, t0, done
    add  s1, s1, s0
    slli s0, s0, 1
    j   loop

done:
```



Unconditional Branching (jal)

- jal (jump and link)
 - performs an unconditional jump like "j," but it additionally stores the return address in a register
 - commonly used for implementing **procedure calls** and **function returns**
 - allows a subroutine to jump to a target address and then return back to the original caller by using the stored return address.



Unconditional Branching (jr)

- jr (jump register)
 - Used to perform an unconditional jump to an address specified in a register
 - used for implementing function returns when the return address is stored in a register.

RISC-V assembly

```
0x00000200      addi s0, zero, 0x210
0x00000204      jr s0
0x00000208      addi s1, zero, 1    # not executed
0x0000020C      sra  s1, s1, 2     # not executed
0x00000210      lw   s3, 44(s1)
```



Unconditional Branching (jalr)

- jalr (jump and link register)
 - combines the functionalities of "jal" and "jr."
 - performs an unconditional jump to an address specified in a register and stores the return address in another register.
 - used for function returns when the return address is stored in a register



Inequalities in RISC-V

- General programs need to test “<” and “>” as well
- Create a RISC-V Inequality instruction
 - Set on Less Than
 - Syntax: `slt reg1, reg2, reg3`
 - Meaning: `reg1 = (reg2 < reg3)`

```
if (reg2 < reg3)
    reg1 = 1;
else reg1 = 0;
```



Inequalities in RISC-V

- For example
 - `if (g < h) goto Less;` #g:\$s0, h:\$s1
- RISC-V code

```
slt $t0,$s0,$s1 # $t0 = 1 if g<h
bne $t0,$0,Less # goto Less
                # if $t0!=0
                # (if (g<h)) Less:
```

- Branch if \$t0 != 0 -> (g < h)
- Register \$0/\$x0 always contains the value 0, so “bne” and “beq” often use it for comparison after an “slt” instruction
- A slt -> bne pair means if (... < ...) goto...



Takeaway Questions

C Code:

```
if(i<j) {  
    a = b /* then */  
} else {  
    a = -b /* else */  
}
```

In English:

- If TRUE, execute the THEN block
- If FALSE, execute the ELSE block

RISCV (???):

```
# i→s0, j→s1  
# a→s2, b→s3
```

```
slt t0 s0 s1
```

```
??? t0,??? else
```

What is ???

```
then:
```

```
add s2, s3, x0
```

```
j end
```

```
else:
```

```
sub s2, x0, s3
```

```
end.
```



Takeaway Questions

- What C code properly fills in the following blank?
 - (A) `j >= 2 && j < i`
 - (B) `j < 2 || j < i`
 - (C) `j < 2 && j >= i`

```
do {i--; } while((z = _____));
```

Loop:

```
addi    s0, s0, -1
slti    t0, s1, 2
bne     t0, x0, Loop
slt     t0, s1, s0
bne     t0, x0, Loop
```



Takeaway Questions

- What C code properly fills in the following blank?
 - (A) `j >= 2 && j < i`
 - (B) `j < 2 || j < i`
 - (C) `j < 2 && j >= i`

```
do {i--; } while((z = _____));
```

```
Loop:          # i→s0, j→s1
addi   s0, s0, -1    # i = i - 1
slti   t0, s1, 2     # t0 = (j < 2)
bne    t0, x0, Loop  # goto Loop if t0!=0
slt    t0, s1, s0    # t1 = (j < i)
bne    t0, x0, Loop  # goto Loop if t0!=0
```



Takeaway Questions

- Final compiled RISC-V assembly code:

```
Loop: sll $t1, $s3, 2    # $t1 = 4*i
      add $t1, $t1, $s5  # $t1 = addr A
      lw  $t1, 0($t1)   # $t1 = A[i]
      add $s1, $s1, $t1  # g = g + A[i]
      add $s3, $s3, $s4  # i = i + j
      bne $s3, $s2, Loop # goto Loop
                          # if i != h
```

- What is Original C codes of above assembly codes?



Takeaway Questions

- Final compiled RISC-V assembly code:

```
Loop: sll  $t1, $s3, 2      # $t1 = 4*i
      add  $t1, $t1, $s5   # $t1 = addr A
      lw   $t1, 0($t1)    # $t1 = A[i]
      add  $s1, $s1, $t1  # g = g + A[i]
      add  $s3, $s3, $s4  # i = i + j
      bne  $s3, $s2, Loop # goto Loop
                          # if i != h
```

- What is Original C codes of above assembly codes?

```
Loop: g = g + A[i];
      i = i + j;
      if (i != h) goto Loop;
```



Conclusion

- Introduction
 - Instruction Set Architecture (ISA)
- RISC-V Assembly Language
 - Instructions
 - Register Set
 - Memory
 - Programming constructs