National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 3: ISA I

CS10014 Computer Organization

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

Acknowledgements and Disclaimer

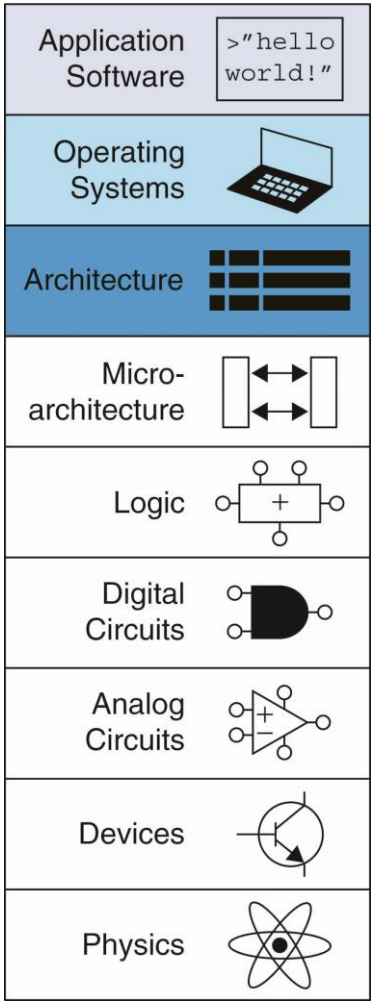
- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS 252 at UC Berkeley
    - https://people.eecs.berkeley.edu/~culler/courses/cs252-s05/
  - E85 at HMC
    - https://pages.hmc.edu/harris/class/e85/old/fall21/

Outline

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

Introduction

- Architecture:
  - Programmer's view of computer
  - Defined by instructions & operand locations
- Microarchitecture:
  - How to implement an architecture in hardware (See single-cycle CPU lecture)



Assembly Language

- 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)

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
  - b, c: source operands (on which the operation is performed)
  - a: destination operand (to which the result is written)

Instructions: Subtraction

C Code

```
a = b - c;
```

RISC-V assembly code

```
sub a, b, c
```

- RISC-V assembly code
  - sub: mnemonic
  - b, c: source operands
  - a: destination operand

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

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)

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 Register Set

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	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

Instructions with Registers

- Revisit add instruction

# indicates a single-line comment

C Code

```
a = b + c;

a = b + 6;
```

RISC-V assembly code

```
# s0 = a, s1 = b, s2 = c
add s0, s1, s2

# s0 = a, s1 = b
addi s0, s1, 6
```

Name	Register Number	Usage
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
s2-11	x18-27	Saved registers

Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables kept in registers

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

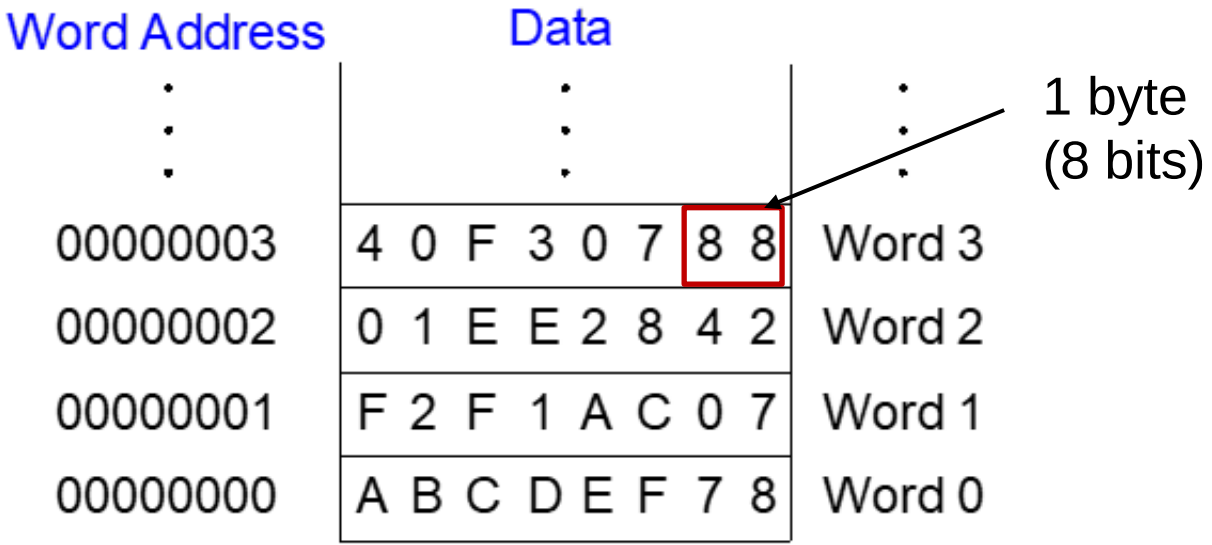
- RISC-V is a Load/Store architecture
  - Requires values to be loaded/stored explicitly from/to memory before operating on them
  - Requires the data to be first retrieved from memory into a register by executing a load instruction
- First, we'll discuss word-addressable memory
- Then, we will discuss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



Reading Word-Addressable Memory

- Memory read called load
- Mnemonic: load word (lw)
- Format:

```
lw <rd>, <offset>(<base register>)
lw t1, 5(s0)
```

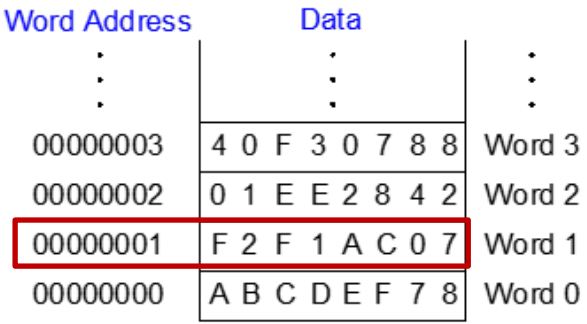
- Address calculation
  - Add offset (5) to the base address (s0)
  - Address = (s0 + 5)
- Destination register (rd)
  - t1 holds the value at address (s0 + 5)
  - Any register may be used as based address

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
```



Writing Word-Addressable Memory

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

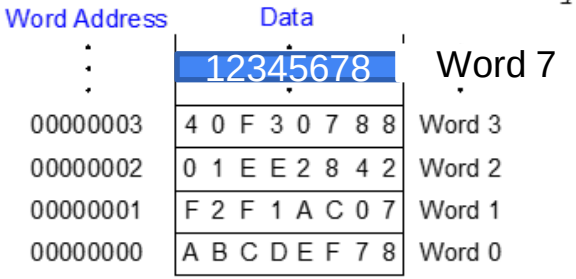
```
sw <src1>, <offset>(<base register>)
```

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

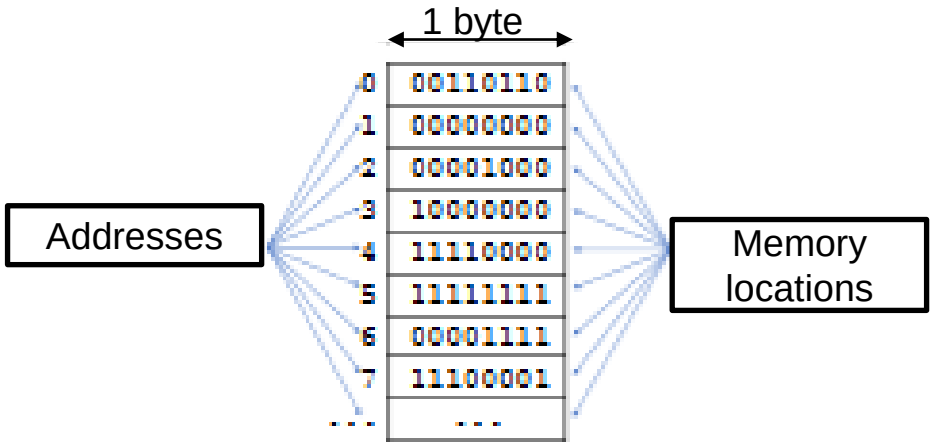
Assembly code

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



Byte-Addressable Memory

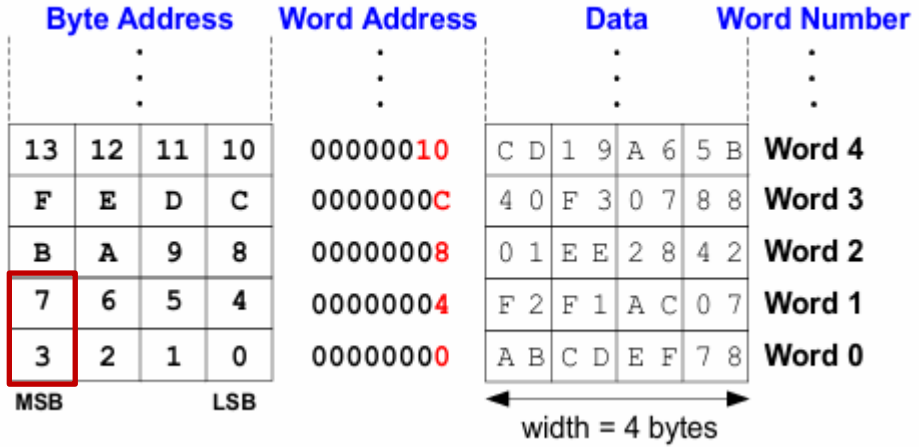
- Byte addressable memory
  - Each memory location stores a single byte and is associated with a unique address



Byte-Addressable Memory

- Each data byte has unique address
  - Load/store words or single bytes:
  - load byte (**lb**) and store byte (**sb**)
  - 32-bit word = 4 bytes, so word address increments by 4

MSB has an address of base + 3 = 7

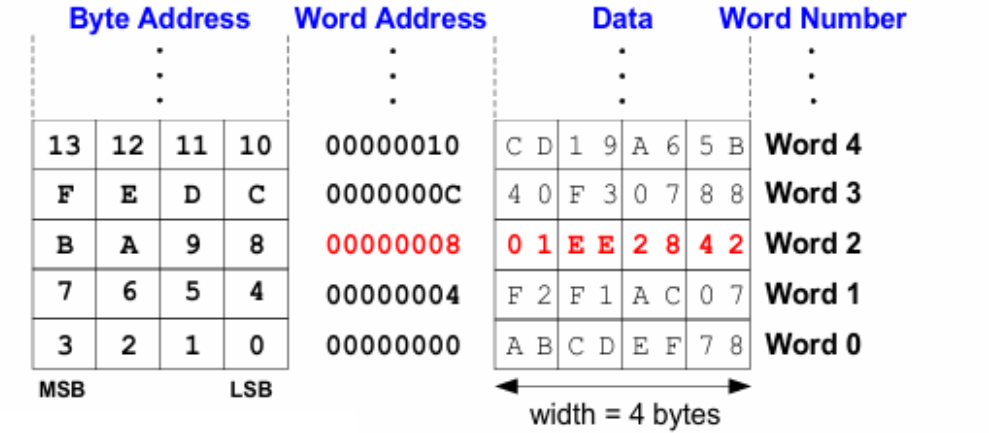


Reading Byte-Addressable Memory

- The address of a memory word must be multiplied by 4
  - Load a word of data at memory address 8 into s3
  - s3 holds the value 0x1EE2842 after load

RISC-V assembly code

```
lw s3, 8(zero)  # read word at address 8 into s3
```

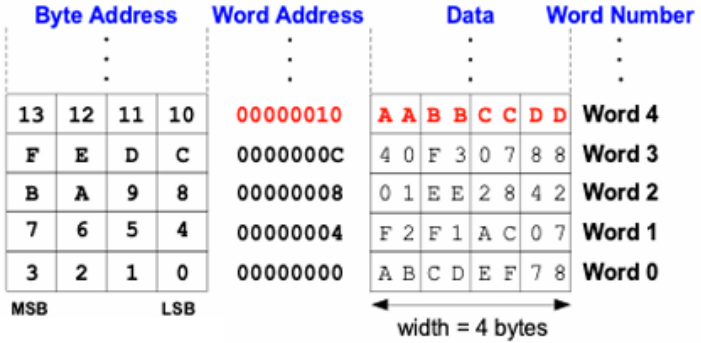


Writing Byte-Addressable Memory

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

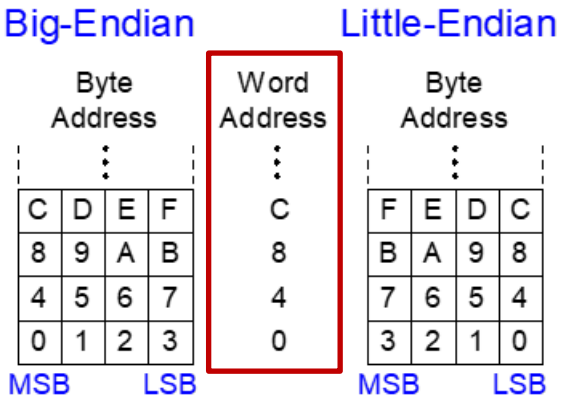
RISC-V assembly code

```
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

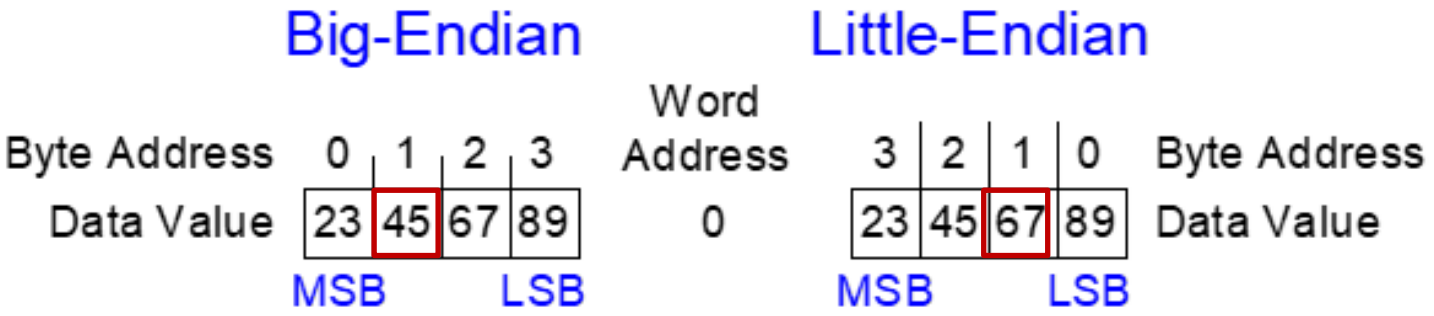
- Suppose t0 initially contains 0x23456789
  - After following code runs on **big-endian syst**em, what value is s0?
  - In a **little-endian system**?

```
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)
```



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 of RISC-V Register 1 (x1 = x0 + x0)?
  - (A) 1
  - (B) 0
  - (C) 2
- What are advantages of the RISC instructions?
  - (A) Reducing the complexity of the processor
  - (B) Decreasing the number of executed instructions
  - (C) Simplify the compiler design

Takeaway Questions

- What is the value in x12?
  - (A) 0x8
  - (B) 0xf8
  - (C) 0xffffff8

```
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) 0xfffffff8

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

Sign

0x8f5 <=> 1000 1111 0101 (2' complement) <=> -779(DEC)

1000 1111 0101 (2'complement) -> -779
1000 1111 0100 (1' complement)
0111 0000 1011 (unsigned 779)

Takeaway Questions

- What is the value in x12?
  - (A) 0x8
  - (B) 0xf8
  - (C) 0xffffff8

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

Sign

0x8f5 <=> 1000 1111 0101 (2' complement) <=> -779(DEC)

1111 1111 1111 1111 1111 1000 1111 0101 (Signed extend 0x8f5 to 32-bits) => 0xfffff8f5

Takeaway Questions

- What is the value in x12?
  - (A) 0x8
  - (B) 0xf8
  - (C) 0xffffff8

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

- **addi x11, x0, 0x8f5**
- The immediate got sign extended, x11 is 0xfffff8f5 because x11 is signed 32-bit register
- **sw x11, 0(x5)**
- the value of x11 is copied to x5 = 0xfffff8f5

Takeaway Questions

- What is the value in x12?
  - (A) 0x8
  - (B) 0xf8
  - (C) 0xffffff8

```
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) = 0xffffff8

Programming

- High-level constructs
  - Loops, conditional statements
- First, introduce:
  - Logical operations
  - Shifty instructions
  - Generating constants
  - Multiplication

Logical Instructions

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

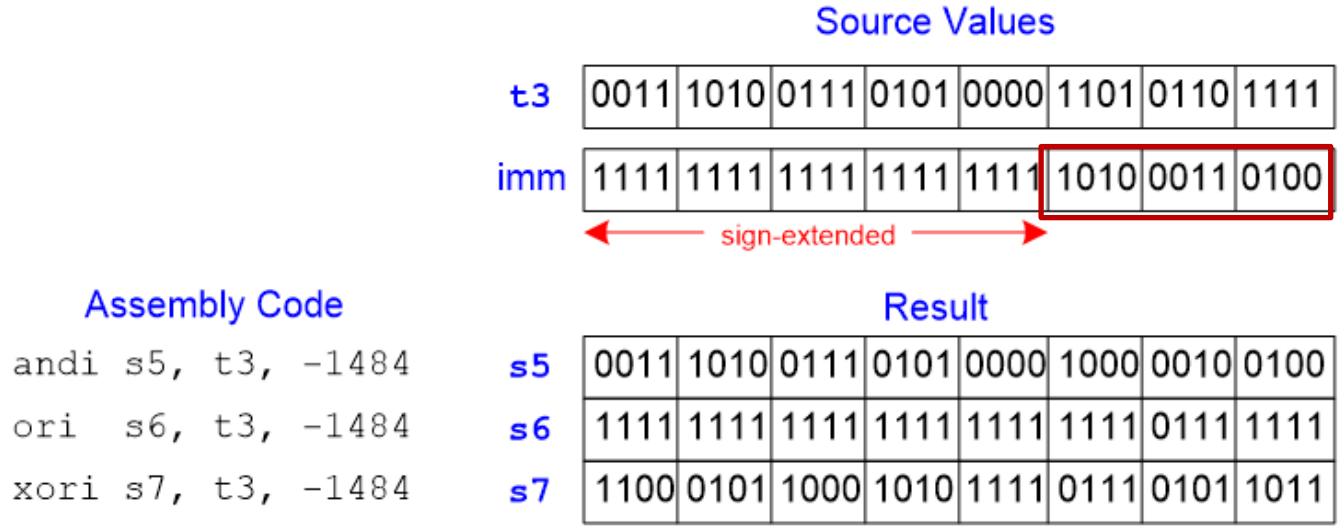
Logical Instructions Example

Source Registers

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

Assembly Code	Result				
`and s3, s1, s2`	s3	0100 0110	1010 0001	0000 0000	0000 0000
`or  s4, s1, s2`	s4	1111 1111	1111 1111	1111 0001	1011 0111
`xor s5, s1, s2`	s5	1011 1001	0101 1110	1111 0001	1011 0111

Logical Instructions Example

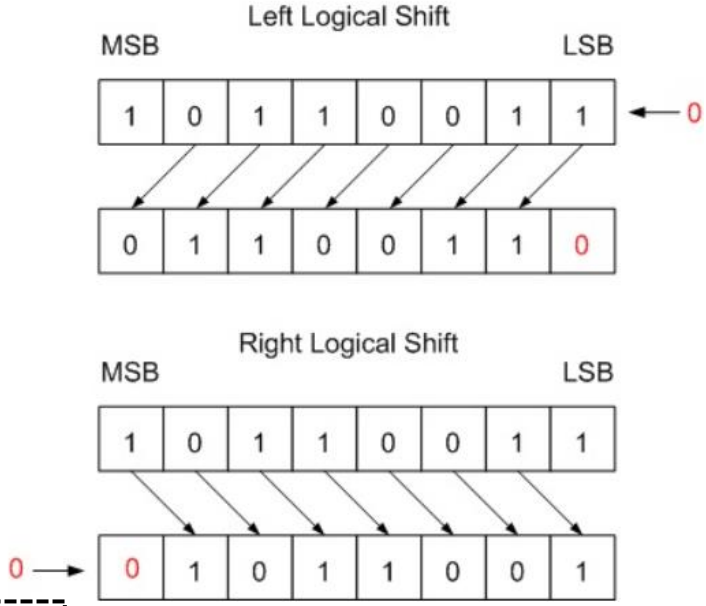


1484 =
010111001100
-1484 =
1's complement
101000110011
2's complement
101000110100

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

Shift Instructions

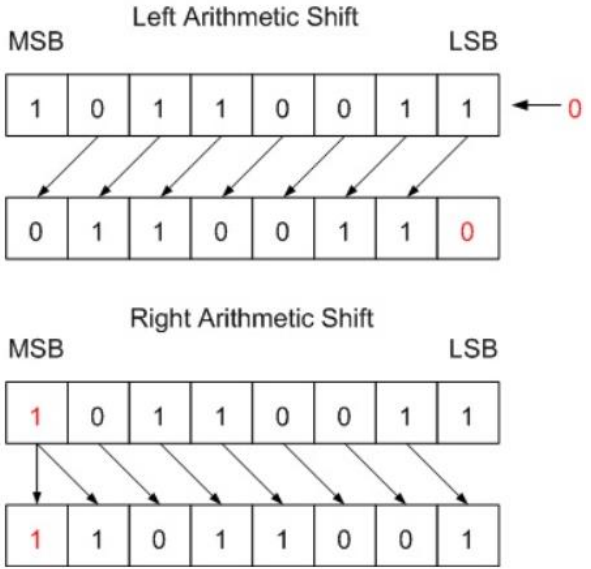
- **Logical shift**
  - Correspond to (left-shift) multiplication by 2, (right-shift) integer division by 2.
- sll: shift left logical
  - Example: sll t0, t1, t2 # t0 = t1 << t2
- srl: shift right logical
  - Example: srl t0, t1, t2 # t0 = t1 >> t2




```
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



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
- srli: shift right logical immediate
  - Example: srli t0, t1, 18 # t0 = t1 >> 18
- srai: shift right arithmetic immediate
  - Example: srai t0, t1, 5 # t0 = t1 >>> 5

Generating Constants

- 12-bit signed constants using addi:

C Code

```
// int is a 32-bit signed word
int a = -372;
```

RISC-V assembly code

```
# s0 = a
addi s0, 0, -372
```

- Any immediate that needs more than 12 bits cannot use this method

Generating 32-bit Constants

- Use load upper immediate (**lui**) and **addi**:
  - **lui**: puts an immediate in the upper 20 bits of destination register, 0's in lower 12 bits

C Code

```
int a = 0xFEDC8765;
```

RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC8
addi s0, s0, 0x765
```

  - Remember that addi sign-extends its 12-bit immediate

Generating 32-bit Constants

- If bit 11 of 32-bit constant is 1, increment upper 20 bits by 1 in lui
  - if the MSB of the 12-bit constant (i.e. bit 11) is a 1, the constant is then sign extended.

C Code

```
int a = 0xFEDC8EAB;
```

Note: -341 = 0xEAB

RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC9     # s0 = 0xFEDC9000
addi s0, s0, -341    # s0 = 0xFEDC9000 + 0xFFFFFEAB
                     #    = 0xFEDC8EAB
```

0xFFFFF = -1

Signed extension

-341 = 0xEAB =
1110 1010 1011

bit 11 of 32-bit

RISC-V: Pseudo-instruction

- Load immediate 32-bit word is tedious
- Pseudo-instruction
  - Assembler program translate “Load immediate” instruction “li” to two real RISC-V instructions: “lui” and “addi’

C Code

```
int a = 0xFEDC8EAB;
```

Note: -341 = 0xEAB

RISC-V pseudoinstructions

```
# s0 = a
li   s0, 0xFEDC8EAB
```

RISC-V real instructions

```
# s0 = a
lui  s0, 0xFEDC9
addi s0, s0, 0xEAB
```

RISC-V: Pseudo-instruction

- There is no instruction to load a register with a constant value
  - To load s0 with the small constant 6, we use the instruction
    - addi s0, zero, 6
  - To load s0 with a large constant 0xFEDC8EAB
    - lui s0, 0xFEDC9
    - addi s0, s0, 0xEAB
  - To load a register with a constant of any size constant (up to 32 bits)
    - li s0, 6
    - li s0, 0xFEDC9

RISC-V: Addressing Modes

- How do we address the operands?
  - Register only
  - Immediate
  - Base addressing
  - PC-relative

Register Only

- Operands found in registers
  - **Example:** add s0, t2, t3
  - **Example:** sub t6, s1, 0

Immediate

- 12-bit signed immediate used as an operand
  - **Example:** addi s4, t5, -73
  - **Example:** ori t3, t7, 0xFF

RISC-V: Base + Offset Addressing

- Base Addressing
  - Loads and Stores
  - Base address + immediate
    - lw s4, 72(zero) # address = 0 + 72
    - sw t2, -25(t1) #address = t1 - 25
  - In C++, we call this a pointer – it points to the place where the operand is stored
  - An offset value must be a 12-bit 2's complement immediate constant

RISC-V: PC-relative Addressing

- PC-Relative Addressing: branch and jal

Example:

Address		Instruction
0x354	L1:	addi s1, s1, 1
0x358		sub t0, t1, s7
...		...
0xEB0		bne s8, s9, L1

  - The operand is derived from the Program Counter (PC) value by adding a 13-bit 2's complement offset
  - The label is (0xEB0-0x354) = 0xB5C (2908) instructions before bne
  - This type of addressing is ONLY used by the branch and jump instructions

Multiplication

- 32 x 32 multiplication, 64-bit result
- mul s0, s1, s2
  - s0 = lower 32 bits of result
- mulh s0, s1, s2
  - s0 = upper 32 bits of result, treats operands as signed
- mulhu s0, s1, s2
  - s0 = upper 32 bits of result, treats operand as unsigned
- mulhsu s0, s1, s2
  - s0 = upper 32 bits of result, treats s1 as signed, s2 as unsigned

Division

- 32 x 32 division, 32-bit quotient, remainder
  - div s1, s2, s3 # s1 = s2/s3
  - divu s1, s2, s3 # unsigned division

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 2^x = 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

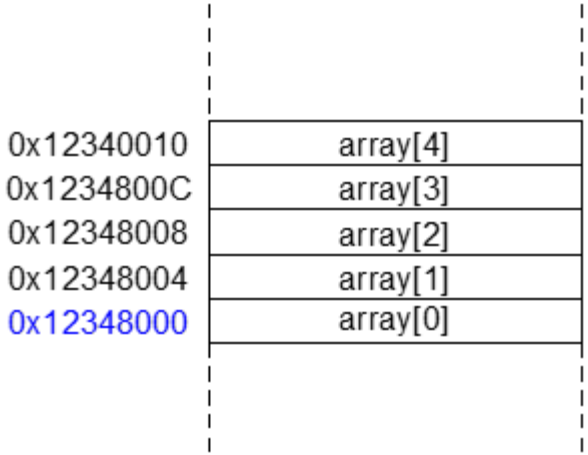
```
# s0 = i, s1 = sum
      addi  s1, zero, 0
      addi  s0, zero, 1
      addi  t0, zero, 101
loop:
      bge   s0, t0, done
      add   s1, s1, s0
      slli  s0, s0, 1
      j     loop
done:
```

Arrays

- Access large amounts of similar data
- Index: access each element
- Size: number of elements

Arrays

- 5-element array
  - Base address = 0x12348000 (address of first element, array[0])
  - First step in access an array: load base address into a register



Accessing Arrays

```
// C Code
  int array[5];
  array[0] = array[0] * 2;
  array[1] = array[1] * 2;

# RISC-V assembly code
# s0 = array base address
  lui   s0, 0x12348                 # 0x12348 in upper 20 bits of s0

  lw    t1, 0(s0)           # t1 = array[0]
  slli  t1, t1, 1           # t1 = t1 * 2
  sw    t1, 0(s0)           # array[0] = t1

  lw    t1, 4(s0)           # t1 = array[1]
  slli  t1, t1, 1           # t1 = t1 * 2
  sw    t1, 4(s0)           # array[1] = t1
```

Accessing Using For Loops

```
// C Code
   int array[1000];
   int i;

   for (i=0; i < 1000; i = i + 1)
          array[i] = array[i] * 8;

# RISC-V assembly code
# s0 = array base address, s1 = i
```

Accessing Using For Loops

```
# RISC-V assembly code
# s0 = array base address, s1 = i
# initialization code
  lui  s0, 0x23B8F          # s0 = 0x23B8F000
  ori  s0, s0, 0x400        # s0 = 0x23B8F400
  addi s1, zero, 0          # i = 0
  addi t2, zero, 1000       # t2 = 1000

loop:
  bge  s1, t2, done         # if not then done
  slli t0, s1, 2            # t0 = i * 4 (byte offset)
  add  t0, t0, s0           # address of array[i]
  lw   t1, 0(t0)            # t1 = array[i]
  slli t1, t1, 3            # t1 = array[i] * 8
  sw   t1, 0(t0)            # array[i] = array[i] * 8
  addi s1, s1, 1            # i = i + 1
  j    loop                 # repeat
done:
```

ori:
Description:
Performs bitwise OR on register rs1 and the sign-extended 12-bit immediate and place the result in rd
Implementation:
x[rd] = x[rs1] | sext(immediate)

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…

RISC-V Assembly Code Example

- Search for the max value on an array

```
1  /* Global array */
2  int numbers[10];
3
4  /* Returns the largest value from array numbers. */
5  int get_largest_number()
6  {
7      int largest = numbers[0];
8      for (int i=1; i<10; i++) {
9          if (numbers[i] > largest)
10             largest = numbers[i];
11     }
12     return largest;
13 }
```

RISC-V Assembly Code Example

```
.data
# Allocate the numbers array (10 integers * 40 bytes)
numbers: .skip 40

.text
get_largest_number:
    la a5, numbers          # a5 := &numbers
    lw a0, (a5)             # a0 (largest) := numbers[0]
    li a1, 1                # a1 (i) := 1
    li t4, 10
    for:
        bge  a1, t4, end    # if i >= 10, then exit the loop (end label)
        slli t1, a1, 2      # t1 := i * 4
        add  t2, a5, t1     # t2 := &numbers + i*4
        lw   t3, (t2)       # t3 := numbers[i]
        blt  t3, a0, skip # if numbers[i] < largest, then skip
        mv   a0, t3         # update largest
        skip:
        addi a1, a1, 1      # i := i + 1
        j    for
    end:
    ret                     # return
```

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
then:
add s2, s3, x0
j   end
else:
sub s2, x0, s3
end:
```

What is ???

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