E85

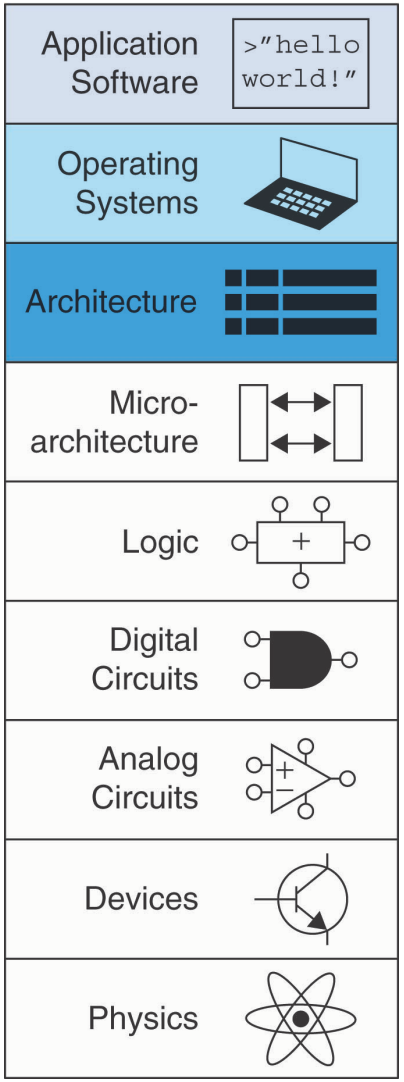
Digital Electronics & Computer Architecture

Lecture 16:

RISC-V Assembly Language

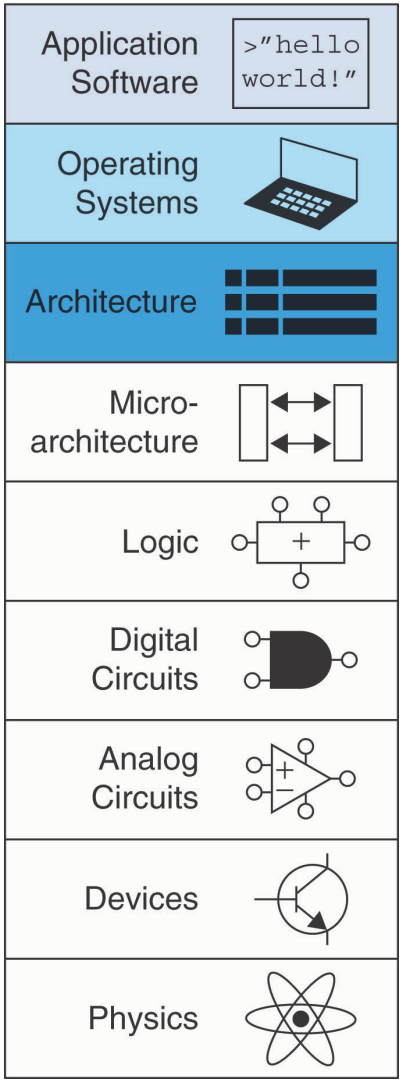
Lecture 16

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



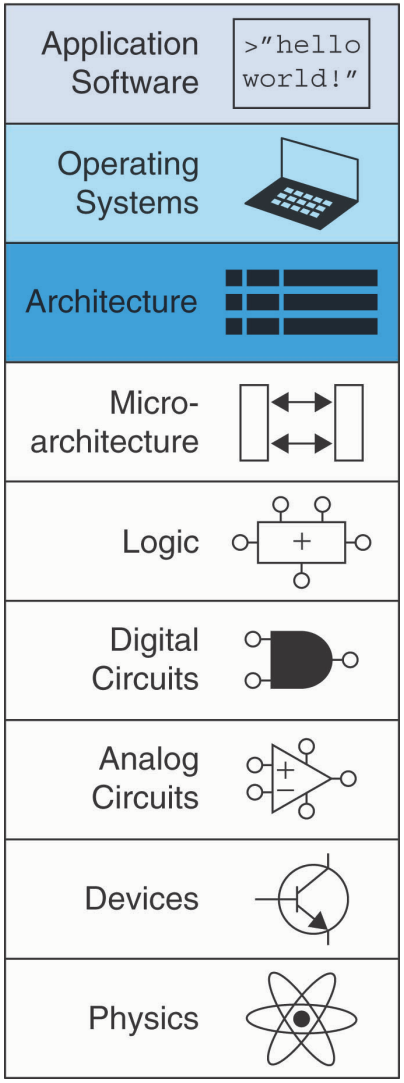
Lecture 16

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



Introduction

- Jumping up a few levels of abstraction
- **Architecture:** programmer's view of computer
  - Defined by _____ & _____
- **Microarchitecture:** how to implement an architecture in hardware (covered in Chapter 7)



Assembly Language

- **Instructions:** commands in a computer's language
  - **Assembly language:** _____ format of instructions
  - **Machine language:** _____ format (1's and 0's)
- **RISC-V** architecture:
  - Developed by Krste Asanovic, David Patterson and their colleagues at UC Berkeley in 2010.
  - First open-source computer architecture

Once you've learned one architecture, it's easier to learn others

Instruction Set Architecture Manual

- RISC-V ISA Manual available at: https://riscv.org/specifications/isa-spec-pdf/
- Details base integer ISA along with optional extensions
- Gives not only the specs but also some helpful rationale and reasoning behind the decisions

The RISC-V Instruction Set Manual
Volume I: Unprivileged ISA
Document Version 20191213

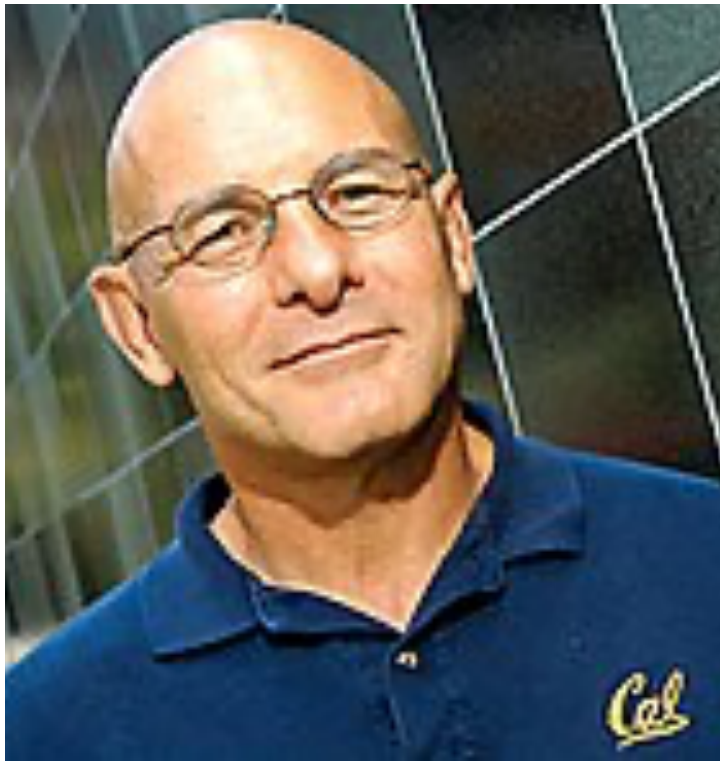
Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu
December 13, 2019

Krste Asanovic

- Professor of Computer Science at the University of California, Berkeley
- Developed RISC-V during one summer
- Chairman of the Board of the RISC-V Foundation
- Co-Founder of SiFive, a company that commercializes and develops supporting tools for RISC-V

David Patterson

- Professor of Computer Science at the University of California, Berkeley since 1976
- Coinvented the Reduced Instruction Set Computer (**RISC**) with John Hennessy in the 1980's
- Developed the **RISC** architecture at Berkeley in 1984, which was later commercialized as SPARC architecture



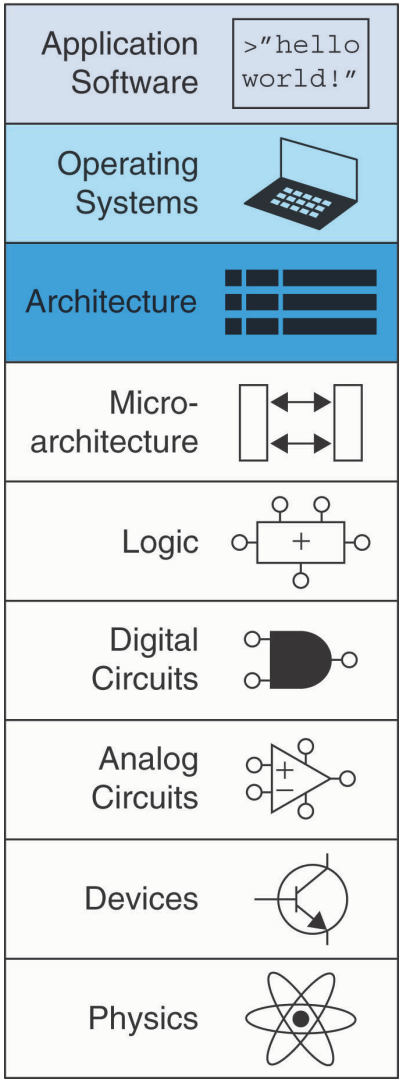
Architecture Design Principles

Underlying design principles, as articulated by Hennessy and Patterson:

1. **Simplicity favors regularity**
2. **Make the common case fast**
3. **Smaller is faster**
4. **Good design demands good compromises**

Lecture 16

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



Instructions: Addition

C Code

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

RISC-V assembly code

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

- **`add:`** _____ indicates operation to perform
- **`b, c:`** _____ (on which the operation is performed)
- **`a:`** _____ (to which the result is written)

Instructions: Subtraction

Similar to addition - only **mnemonic** changes

C Code

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

RISC-V assembly code

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

- **`sub:`** mnemonic
- **`b, c:`** source operands
- **`a:`** destination operand

Design Principle 1

Simplicity favors regularity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Easier to encode and handle in hardware

Multiple Instructions

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

C Code

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

RISC-V assembly code

```
_____   # t = b + c
_____   # a = t - d
```

Design Principle 2

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 (that are less common) performed using multiple simple instructions
- RISC-V is a _____, with a small number of simple instructions
- Other architectures, such as Intel's x86, are _____ _____

Operands

- Operand location: physical location in computer
  - _____
  - _____
  - _____

Operands: Registers

- RISC-V has 32 32-bit registers
- Registers are faster than memory
- RISC-V called "_____" because it operates on 32-bit data

Design Principle 3

Smaller is Faster

- RISC-V includes only a small number of registers

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 _____, s0-s11, used to hold variables
  - the _____, t0-t6, used to hold intermediate values during a larger computation
  - Discuss others later

Instructions with Registers

- Revisit add instruction

C Code

```
a = b + c
```

RISC-V assembly code

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

# indicates a _____

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

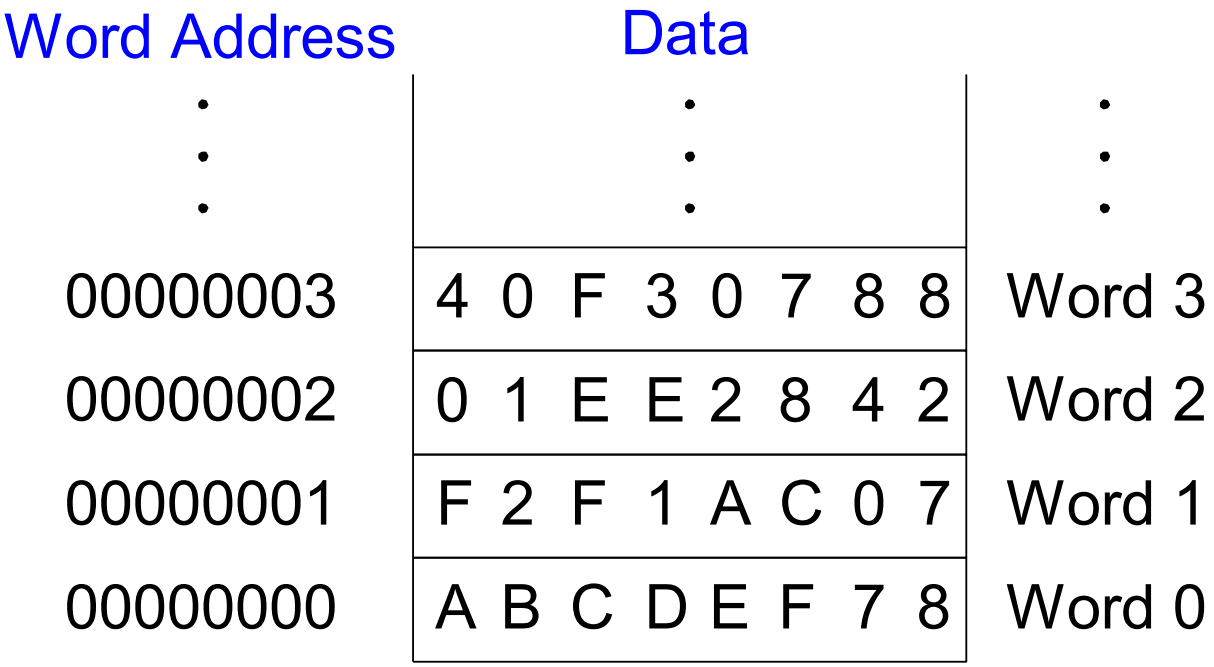
Memory

- First, we'll discuss **word-addressable** memory
- Then we'll discuss **byte-addressable** memory

RISC-V is _____

Word-Addressable Memory

- Each 32-bit data word has a unique address



Note: RISC-V uses **byte-addressable** memory, which we'll talk about next.

Reading Word-Addressable Memory

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

  _____

  _____

  Address calculation:
  - add _____ to the _____
  - address = _____
- Destination register (rd):
  - ___ holds the value at address (_____)

  Any register may be used as base 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

```
                    # 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

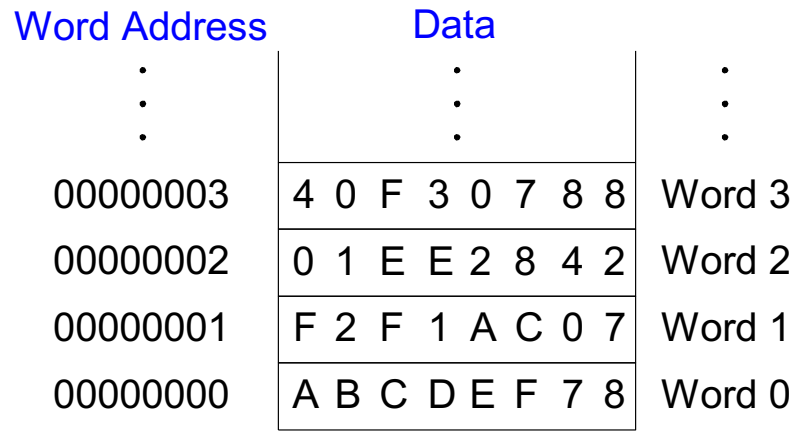
Writing Word-Addressable Memory

- **Example:** Write (store) the value in t4 into memory address 7
  - add the base address (zero) to the offset (0x7)
  - address: (0 + 0x7) = 7

Offset can be written in decimal (default) or hexadecimal

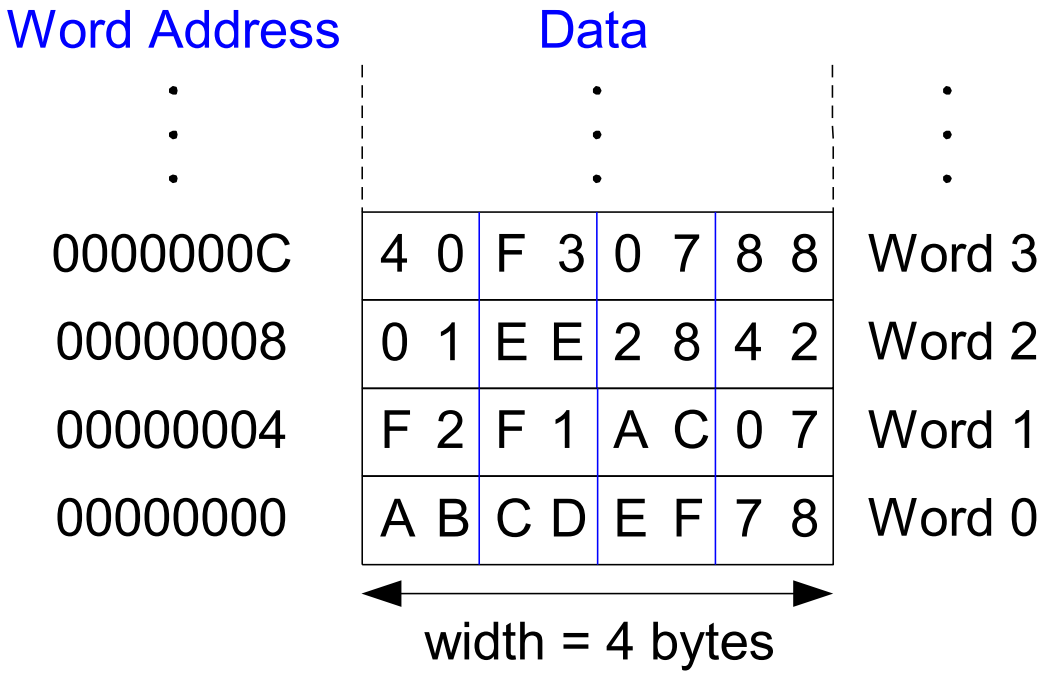
Assembly code

```
                        # write the value in t4
                        # to memory word 7
```



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



Reading Byte-Addressable Memory

- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is
  - the address of memory word 10 is

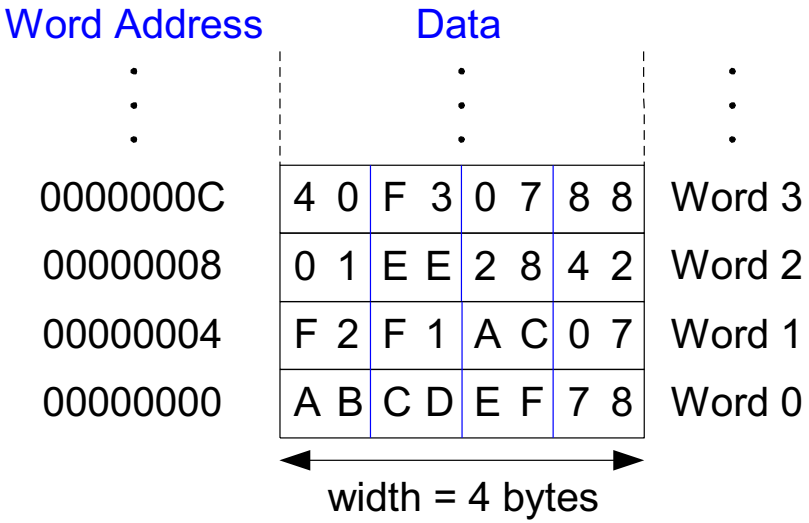
- **RISC-V is _____, not _____**

Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 4 into `s3`.
- `s3` holds the value 0xF2F1AC07 after load

RISC-V assembly code

```
                    # read word at address 4 into s3
```

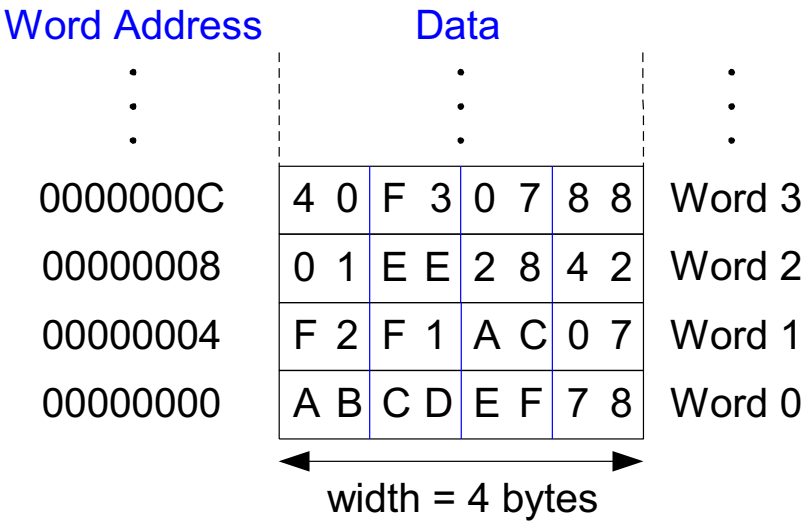


Writing Byte-Addressable Memory

- **Example:** stores the value held in `t7` into memory address 0x2C (44)

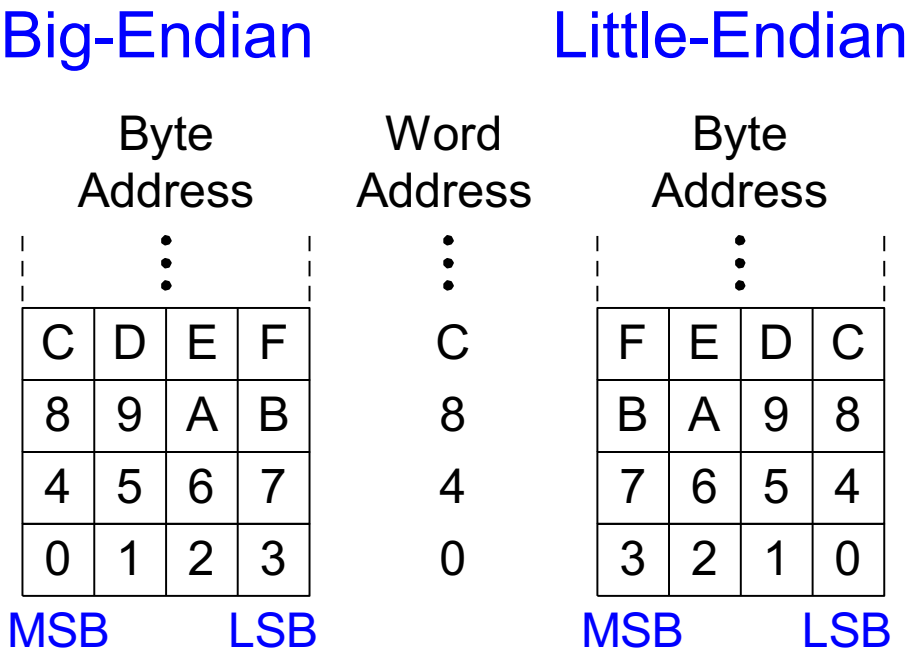
RISC-V assembly code

```
                    # write t7 into address 44
```



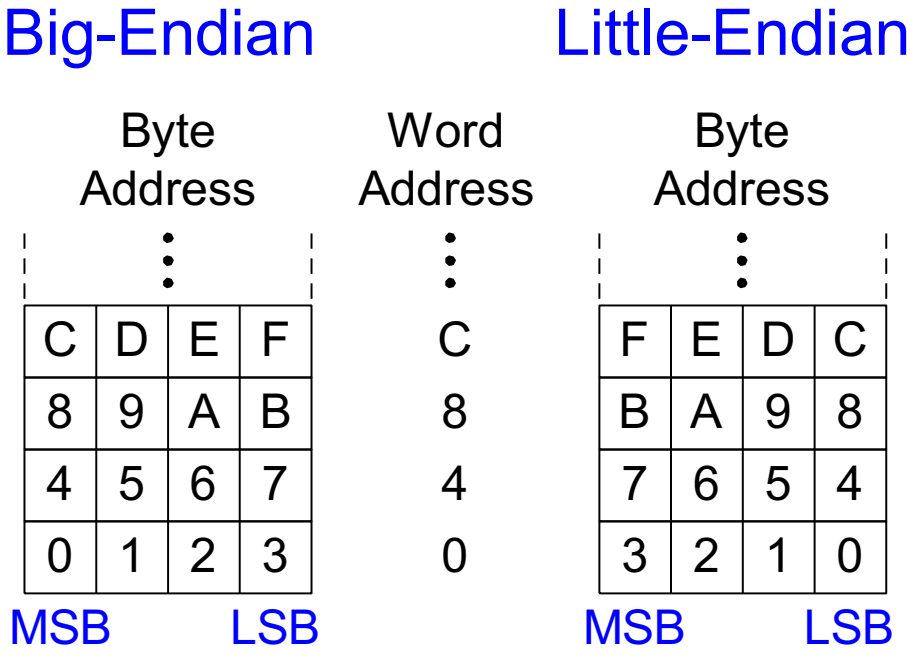
Big-Endian & Little-Endian Memory

- How to number bytes within a word?
- _____ byte numbers start at the little (least significant) end
- _____ byte numbers start at the big (most significant) end
- **Word address** is the **same** for big- or little-endian



Big-Endian & Little-Endian Memory

- Jonathan Swift's *Gulliver's Travels*: the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- It doesn't really matter which addressing type used – except when the two systems need to share data!



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?

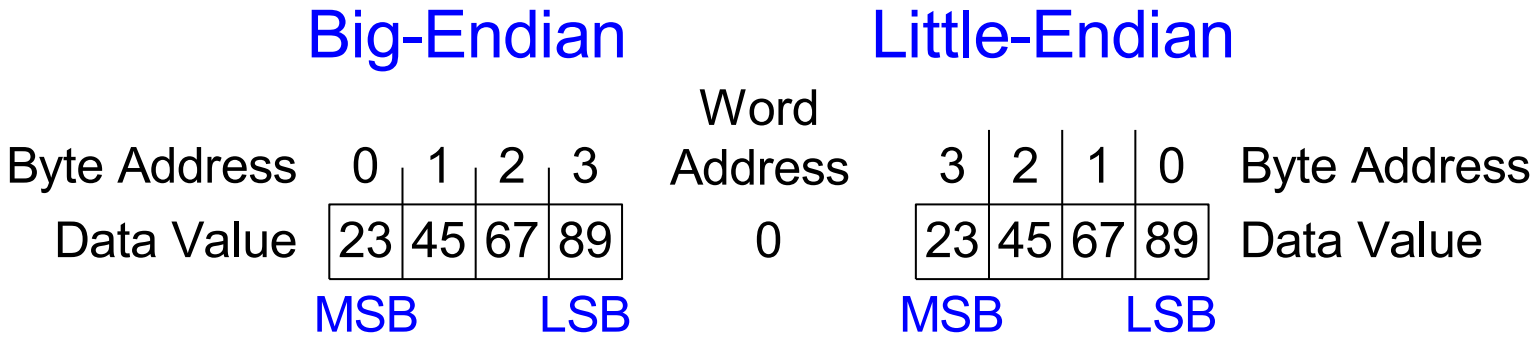
```
sw t0, 0(zero)
lb s0, 1(zero)
```

Big-Endian & Little-Endian Example

- 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:** `s0` = _____
- **Little-endian:** `s0` = _____



Programming

- High-level languages:
  - e.g., C, Java, Python
  - Written at higher level of abstraction
- Common high-level software constructs:
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Ada Lovelace, 1815-1852

- Wrote the first computer program
- Her program calculated the Bernoulli numbers on Charles Babbage's Analytical Engine
- She was the daughter of the poet Lord Byron

Branching

- Execute instructions out of sequence
- Types of branches:
  - **Conditional**
    - 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**
    - jump (`j`)
    - jump register (`jr`)
    - jump and link (`jal`)
    - jump and link register (`jalr`)

Will talk about these when we discuss function calls

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
```

_____ indicate 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 less 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
```

Programming

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

Logical Instructions

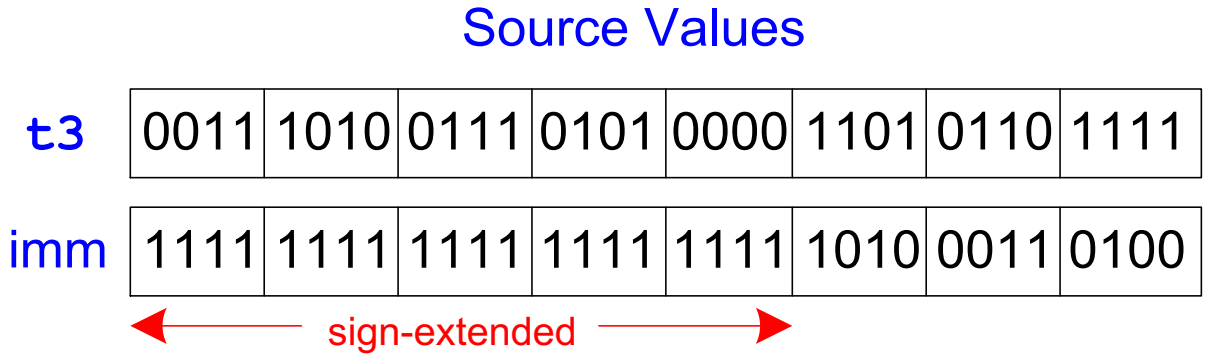
- **`and`, `or`, `xor`**
  - `and`: useful for _____ bits
    - Masking all but the least significant byte of a value:
      0xF234012F AND 0x000000FF = 0x0000002F
  - `or`: useful for _____ bit fields
    - Combine 0xF2340000 with 0x000012BC:
      0xF2340000 OR 0x000012BC = 0xF23412BC
  - `xor`: useful for _____ bits:
    - A xor -1 = ____ A (remember that -1 = 0xFFFFFFFF)

Logical Instructions Example 1

	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 2



Assembly Code

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

Result

s5								
s6								
s7								

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

Shift Instructions

Shift amount is in (lowest 5 bits of) a register

- `sll`: shift left logical
  - Example: _____ `# t0 = t1 << t2`
- `srl`: shift right logical
  - Example: _____ `# t0 = t1 >> t2`
- `sra`: shift right arithmetic
  - Example: _____ `# t0 = t1 >>> t2`

Immediate Shift Instructions

Shift amount is an immediate between 0 to 31

- `slli`: shift left logical immediate
  - Example: _____ `# t0 = t1 << 23`
- `srli`: shift right logical immediate
  - Example: _____ `# t0 = t1 >> 18`
- `srai`: shift right arithmetic imeediate
  - Example: _____ `# 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
_____________________
```

Any immediate that needs **more than ___ 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` _____ 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`

C Code

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

Note: 0xEAB = -341

RISC-V assembly code

```
# s0 = a
_____________________   # s0 = 0xFEDC9000
_____________________   # s0 = 0xFEDC9000 + 0xFFFFFEAB
                        #    = 0xFEDC8EAB
```

Multiplication

- 32 × 32 multiplication, 64-bit result

`____ s0, s1, s2`

`s0` = lower 32 bits of result

`_____ s0, s1, s2`

`s0` = upper 32 bits of result, treats operands as signed

`_____ s0, s1, s2`

`s0` = upper 32 bits of result, treats operands as unsigned

`_____ s0, s1, s2`

`s0` = upper 32 bits of result, treats `s1` as signed, `s3` as unsigned

Multiplication

- 32 × 32 multiplication, 64-bit result
- For full 64-bit result:

```
{s4, s3} = s1 x s2
```

- Could also use `mulhu` or `mulhsu` instead of `mulh`

Division

- 32-bit division, 32-bit quotient, remainder
  - _____ `# s1 = s2/s3`
  - _____ `# unsigned division`

High-Level Code Constructs

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

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

    _____________________
    add s0, s1, s2

L1:
    sub s0, s0, s3
```

Assembly tests opposite case (_____) of high-level code (_____)

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

    ____________________
    add  s0, s1, s2
    ____________________

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:
      _____________________
      slli s0, s0, 1
      addi s1, s1, 1
      j    while
done:
```

Assembly tests for the opposite case (_____) of the C code (_____).

For Loops

```
for (initialization; condition; loop operation)
   statement
```

- **`initialization:`** executes _____ the loop begins
- **`condition:`** is tested _____ of each iteration
- **`loop operation:`** executes at the ___ of each iteration
- **`statement:`** executes _____ 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:
        ____ s0, t0, done
        add  s1, s1, s0
        addi s0, s0, 1
        ____ 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

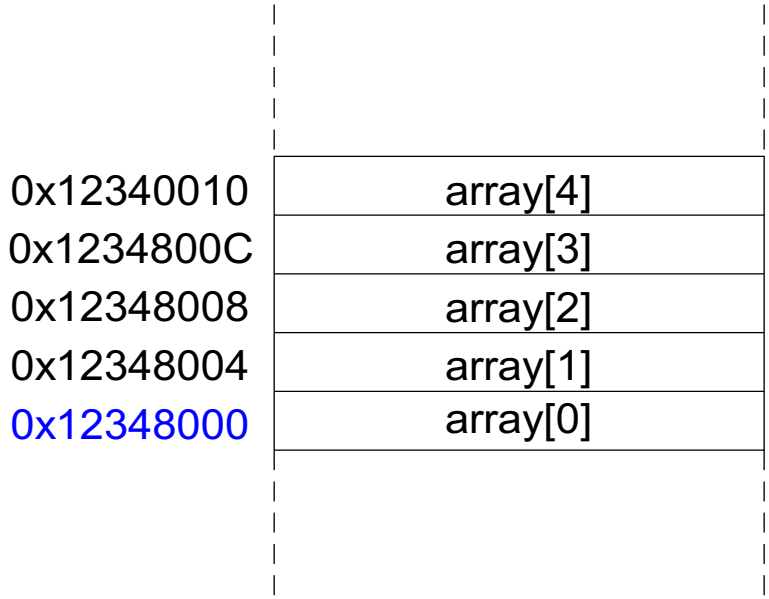
```
# s0 = i, s1 = sum
        addi  s1, zero, 0
        addi  s0, zero, 1
        addi  t0, zero, 101
loop:
        ____  s0, t0, done
        add   s1, s1, s0
        slli  s0, s0, 1
        ____  loop
done:
```

Arrays

- Access large amounts of similar data
- _____ access each element
- _____ number of elements

Arrays

- 5-element array
- _____ = _____ (address of first element, `array[0]`)
- First step in accessing 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
  _____                    # 0x12348 in upper 20 bits of s0

  _____                    # t1 = array[0]
  _____                    # t1 = t1 * 2
  _____                    # array[0] = t1

  _____                    # t1 = array[1]
  _____                    # t1 = t1 * 2
  _____                    # array[1] = t1
```

Arrays 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
```

Arrays 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:
```

ASCII Code

- *American Standard Code for Information Interchange*
- Each text character has unique byte value
  - For example, S = 0x53, a = 0x61, A = 0x41
  - Lower-case and upper-case differ by 0x20 (32)

Cast of Characters

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Unconditional Branching (`jr`)

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

Lecture 16

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