MIPS Assembly

Digital Design and Computer Architecture
Mohammad Sadrosadati
Frank K. Gürkaynak

`http://safari.ethz.ch/ddca`

In This Lecture

- **Assembly Language**
- **Architecture Design Principles**
  - Simplicity favors regularity
  - Make the common case fast
  - Smaller is faster
  - Good design demands good compromises
- **Where to store data (memory/register)**
- **Main types of MIPS instructions**
  - (R)egister type
  - (I)mmediate type
  - (J)ump type

Introduction

- **Jumping up a few levels of abstraction**
- ***Architecture:*** **the programmer's view of the computer**
  - Defined by instructions (operations) and operand locations
- ***Microarchitecture:*** **Implementation of an architecture (Chapter 7)**

Abstraction Levels	Examples
Application Software	Programs
Operating Systems	Device drivers
Architecture	Instructions, Registers
Micro architecture	Datapath, Controllers
Logic	Adders, Memories
Digital Circuits	AND gates, NOT gates
Analog Circuits	Amplifiers
Devices	Transistors, Diodes
Physics	Electrons

Assembly Language

- **To command a computer, you must understand its language**
  - ***Instructions:*** words in a computer’s language
  - ***Instruction set:*** the vocabulary of a computer’s language
- **Instructions indicate the operation to perform and the operands to use**
  - ***Assembly language:*** human-readable format of instructions
  - ***Machine language:*** computer-readable format (1’s and 0’s)
- **MIPS architecture:**
  - Developed by John Hennessy and colleagues at Stanford in the 1980’s
  - Used in many commercial systems (Silicon Graphics, Nintendo, Cisco)
- **Once you’ve learned one architecture, it’s easy to learn others**

John Hennessy

- President of Stanford University
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Co-invented the Reduced Instruction Set Computer (RISC)
- Developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors have been sold

Architecture Design Principles

- **Underlying design principles, as articulated by Hennessy and Patterson:**
  - Simplicity favors regularity
  - Make the common case fast
  - Smaller is faster
  - Good design demands good compromises

MIPS Instructions: Addition

High-level code

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

MIPS assembly

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

- **add:** mnemonic indicates what operation to perform
- **b, c:** source operands on which the operation is performed
- **a:** destination operand to which the result is written

MIPS Instructions: Subtraction

High-level code

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

MIPS assembly

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

- Subtraction is similar to addition, only mnemonic changes
- sub: mnemonic indicates what operation to perform
- b, c: source operands on which the operation is performed
- a: destination operand to which the result is written

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

Instructions: More Complex Code

High-level code

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

MIPS assembly code

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

- **More complex code is handled by multiple MIPS instructions.**

Design Principle 2

Make the common case fast

- **MIPS includes only simple, commonly used instructions**
- **Hardware to decode and execute the instruction can be simple, small, and fast**
- **More complex instructions (that are less common) can be performed using multiple simple instructions**

RISC and CISC

- **Reduced instruction set computer (RISC)**
  - means: small number of simple instructions
  - example: MIPS
- **Complex instruction set computers (CISC)**
  - means: large number of instructions
  - example: Intel’s x86

Operands

- **A computer needs a physical location from which to retrieve binary operands**

- **A computer retrieves operands from:**
  - Registers
  - Memory
  - Constants (also called immediates)

Operands: Registers

- **Main Memory is slow**
- **Most architectures have a small set of (fast) registers**
  - MIPS has thirty-two 32-bit registers
- **MIPS is called a 32-bit architecture because it operates on 32-bit data**
  - A 64-bit version of MIPS also exists, but we will consider only the 32-bit version

Design Principle 3

Smaller is Faster

- **MIPS includes only a small number of registers**
- **Just as retrieving data from a few books on your table is faster than sorting through 1000 books, retrieving data from 32 registers is faster than retrieving it from 1000 registers or a large memory.**

The MIPS Register Set

Name	Register Number	Usage
`$0`	0	the constant value 0
`$at`	1	assembler temporary
`$v0-$v1`	2-3	procedure return values
`$a0-$a3`	4-7	procedure arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved variables
`$t8-$t9`	24-25	more temporaries
`$k0-$k1`	26-27	OS temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	procedure return address

Operands: Registers

- **Written with a dollar sign ($) before their name**
  - For example, register 0 is written “$0”, pronounced “register zero” or “dollar zero”

- **Certain registers used for specific purposes:**
  - $0 always holds the constant value 0
  - the **saved registers**, $s0-$s7, are used to hold variables
  - the **temporary registers**, $t0 - $t9, are used to hold intermediate values during a larger computation

- **For now, we only use the temporary registers ($t0 - $t9) and the saved registers ($s0 - $s7)**

- **We will use the other registers in later slides**

Instructions with registers

High-level code

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

MIPS assembly

```
# $s0 = a
# $s1 = b
# $s2 = c

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

- **Revisit add instruction**
  - The source and destination operands are now in registers

Operands: Memory

- **Too much data to fit in only 32 registers**
- **Store more data in memory**
  - Memory is large, so it can hold a lot of data
  - But it’s also slow
- **Commonly used variables kept in registers**
- **Using a combination of registers and memory, a program can access a large amount of data fairly quickly**

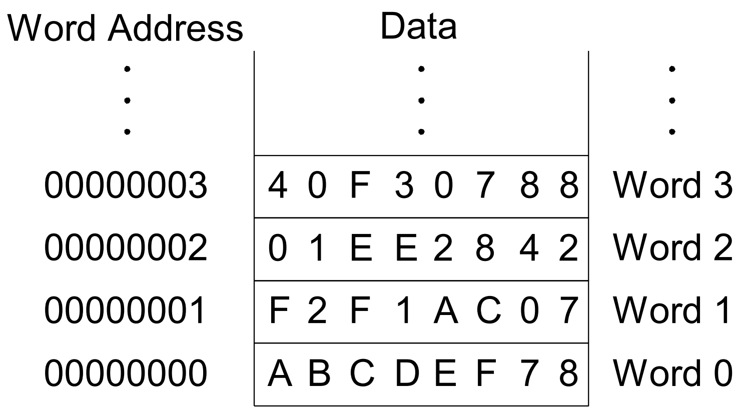
Word-Addressable Memory

- **Each 32-bit data word has a unique address**

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

Reading Word-Addressable Memory

- **Memory reads are called loads**
- **Mnemonic: load word (lw)**
- **Example: read a word of data at memory address 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

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

Reading Word-Addressable Memory

- **Example: read a word of data at memory address 1 into $s3**
- **Memory address calculation:**
  - add the base address ($0) to the offset (1)
  - address = ($0 + 1) = 1
  - `$s3` holds the value **`0xF2F1AC07`** after the instruction completes
- **Any register may be used to store the base address**

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

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

Writing Word-Addressable Memory

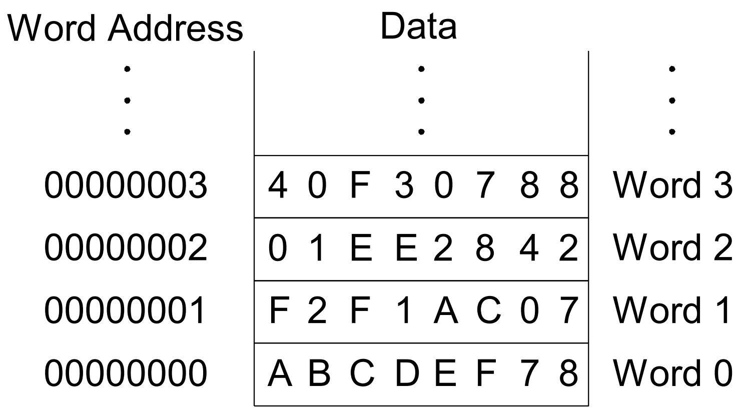
- **Memory writes are called stores**
- **Mnemonic: store word (sw)**
- **Example: Write (store) the value held in `$t4` into memory address 7**

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

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

Writing Word-Addressable Memory

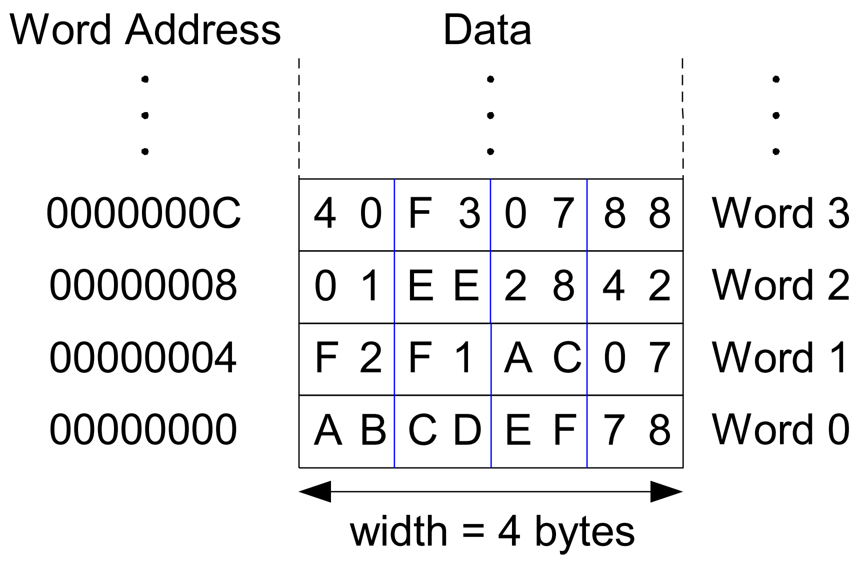
- **Example: Write (store) the value held in `$t4` into memory address 7**
- **Memory address calculation:**
  - add the base address ($0) to the offset (7)
  - address = ($0 + 7) = 7
  - Offset can be written in decimal (default) or hexadecimal
- **Any register may be used to store the base address**



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

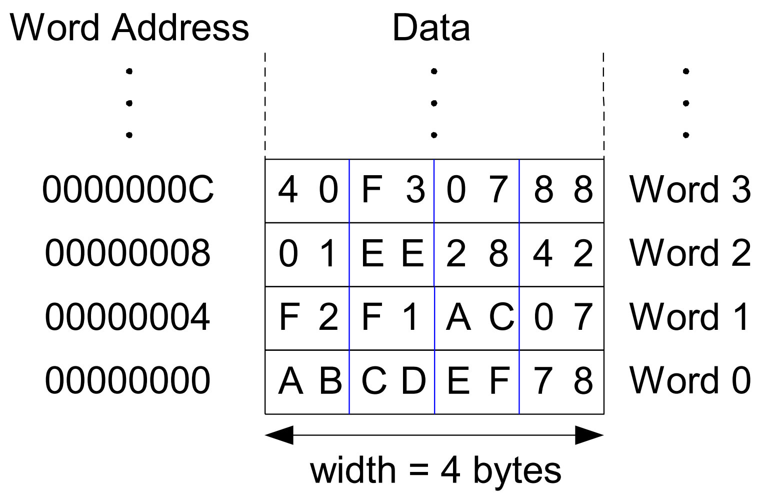
Byte-Addressable Memory

- **Each data byte has a unique address**
- **Load/store words or single bytes: load byte (lb) and store byte (sb)**
- **Each 32-bit words has 4 bytes, so the word address increments by 4. MIPS uses byte addressable memory**



Reading Byte-Addressable Memory

- **Load a word of data at memory address 4 into $s3.**
- **Memory address calculation:**
  - add the base address ($0) to the offset (4)
  - address = ($0 + 4) = 4
- **$s3 holds the value `0xF2F1AC07` after the instruction completes**



```
lw $s3, 4($0)  # read word at address 4 into $s3
```

Writing Byte-Addressable Memory

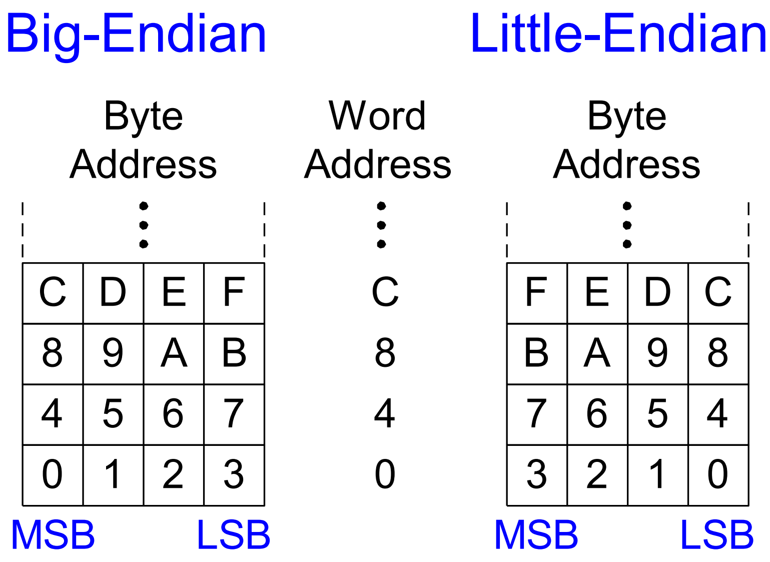
- **Example: store the value held in $t7 into the eleventh 32-bit memory location.**

- **Memory address calculation:**
  - Byte addressable address for word eleven
    $11x4 = 44_{10} = 0x2C_{16}$
  - add the base address ($0) to the offset (0x2c)
  - address = ($0 + 44) = 44

```
sw $t7, 44($0)  # write $t7 into address 44
```

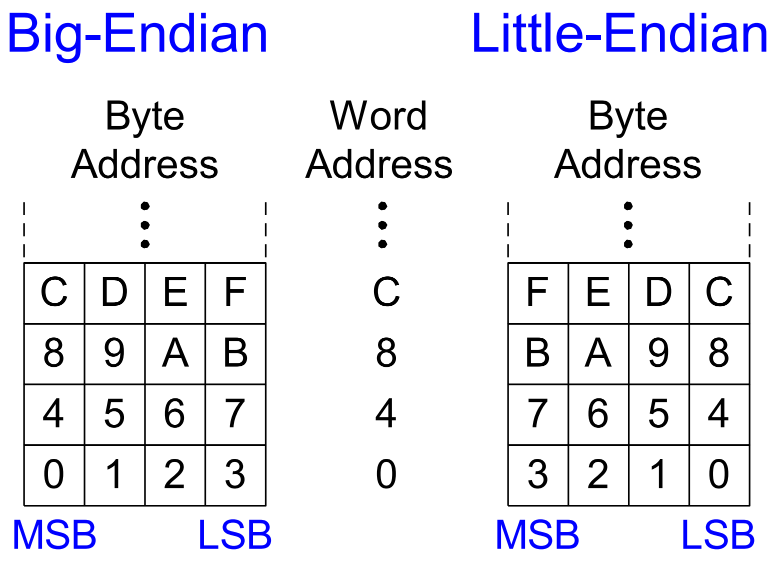
Big-Endian and Little-Endian Memory

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



Big-Endian and Little-Endian Memory

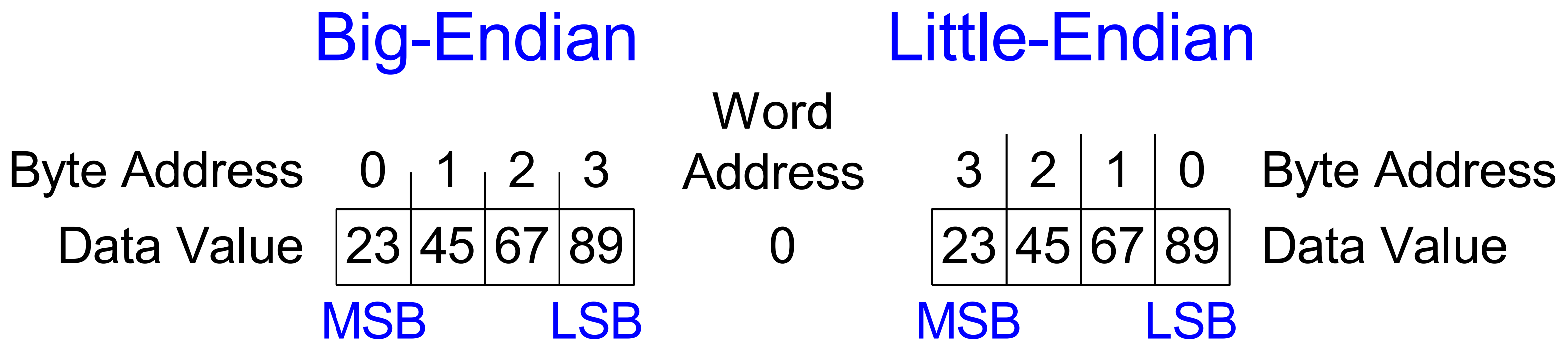
- **From Jonathan Swift's Gulliver's Travels where the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end.**
  - As indicated by the farcical name, it doesn't really matter which addressing type is used – except when the two systems need to share data!



Big- and Little-Endian Example

- **Suppose $t0 initially contains 0x23456789. After the following program is run on a big-endian system, what value does $s0 contain? In a little-endian system?**

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

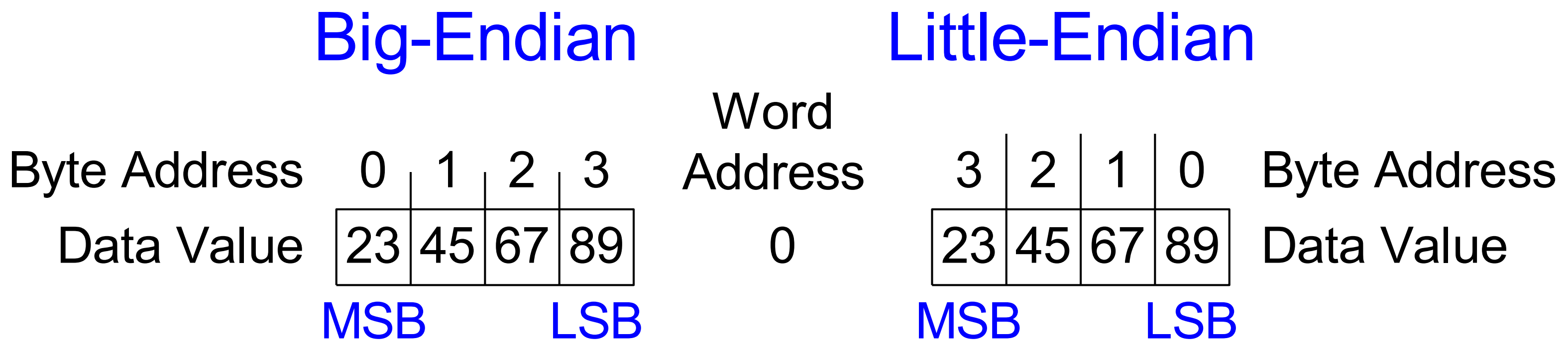


Big- and Little-Endian Example

- **Suppose $t0 initially contains 0x23456789. After the following program is run on a big-endian system, what value does $s0 contain? In a little-endian system?**

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

- **Big-endian:** `0x00000045`
  Little-endian: `0x00000067`



Design Principle 4

Good design demands good compromises

- **Multiple instruction formats allow flexibility**
  - **`add, sub`**: use 3 register operands
  - **`lw, sw`**: use 2 register operands and a constant
- **Number of instruction formats kept small**
  - to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster)

Operands: Constants/Immediates

High-level code

```
a = a + 4;
b = a – 12;
```

MIPS assembly code

```
# $s0 = a, $s1 = b
addi $s0, $s0, 4
addi $s1, $s0, -12
```

- **`lw` and `sw` illustrate the use of constants or immediates**
  - Called immediates because they are directly available
  - Immediates don’t require a register or memory access
- **The add immediate (`addi`) instruction adds an immediate to a variable (held in a register)**
  - An immediate is a 16-bit two’s complement number
- **Is subtract immediate (subi) necessary?**

Machine Language

- **Computers only understand 1's and 0's**
- **Machine language: binary representation of instructions**
- **32-bit instructions**
  - Again, simplicity favors regularity: 32-bit data, 32-bit instructions, and possibly also 32-bit addresses
- **Three instruction formats:**
  - **R-Type**: register operands
  - **I-Type**: immediate operand
  - **J-Type**: for jumping (we'll discuss later)

R-Type

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- **Register-type, 3 register operands:**
  - **rs**, **rt**: source registers
  - **rd**: destination register
- **Other fields:**
  - **op**: the operation code or opcode (0 for R-type instructions)
  - **funct**: the function together, the opcode and function tell the computer what operation to perform
  - **shamt**: the shift amount for shift instructions, otherwise it's 0

R-Type Examples

Assembly Code

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

Field Values

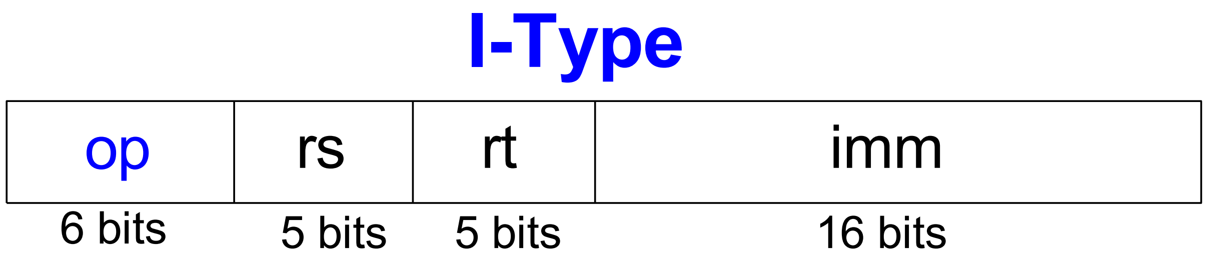
op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Note the order of registers in the assembly code: `add rd, rs, rt`

I-Type



- **Immediate-type, has 3 operands:**
  - **rs**, **rt**: register operands
  - **imm**: 16-bit two’s complement immediate
- **Other fields:**
  - **op**: the opcode
- **Simplicity favors regularity: all instructions have opcode**
- **Operation is completely determined by the opcode**

I-Type Examples

Assembly Code / Field Values

Assembly Code	op	rs	rt	imm
`addi $s0, $s1, 5`	8	17	16	5
`addi $t0, $s3, -12`	8	19	8	-12
`lw $t2, 32($0)`	35	0	10	32
`sw $s1, 4($t1)`	43	9	17	4
	6 bits	5 bits	5 bits	16 bits

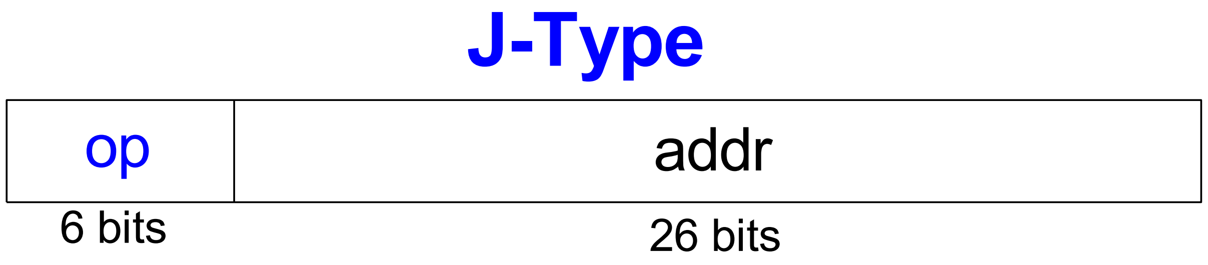
Machine Code

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

Note the differing order of registers in the assembly and machine codes:

```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```

Machine Language: J-Type



- **Jump-type**
- **26-bit address operand (addr)**
- **Used for jump instructions (`j`)**

Review: Instruction Formats

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

J-Type

op	addr
6 bits	26 bits

The Power of the Stored Program

- **32-bit instructions and data stored in memory**
- **Sequence of instructions: only difference between two applications (for example, a text editor and a video game)**
- **To run a new program:**
  - No rewiring required
  - Simply store new program in memory
- **The processor hardware executes the program:**
  - fetches (reads) the instructions from memory in sequence
  - performs the specified operation

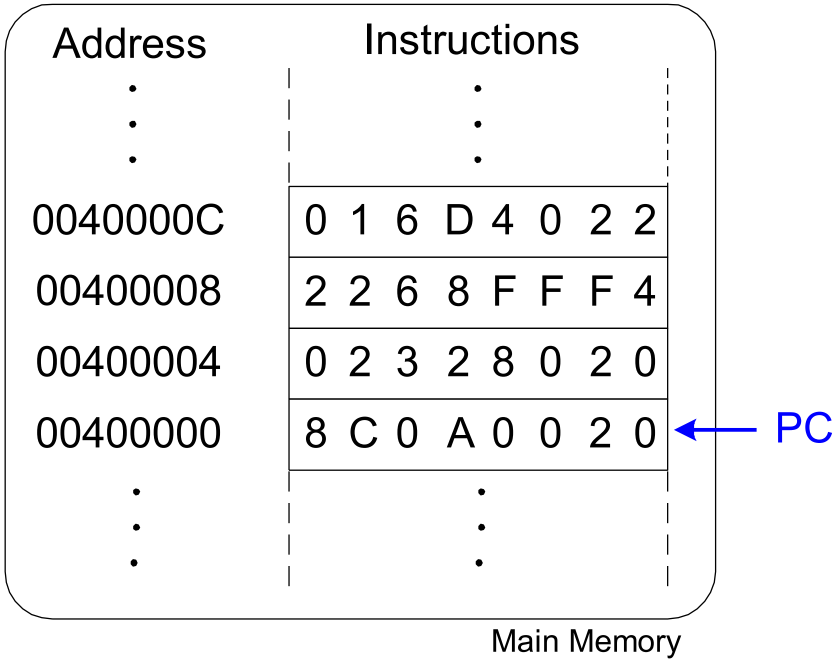
Program counter

- **The processor hardware executes the program:**
  - fetches (reads) the instructions from memory in sequence
  - performs the specified operation
  - continues with the next instruction
- **The program counter (PC) keeps track of the current instruction**
  - In MIPS, programs typically start at memory address `0x00400000`

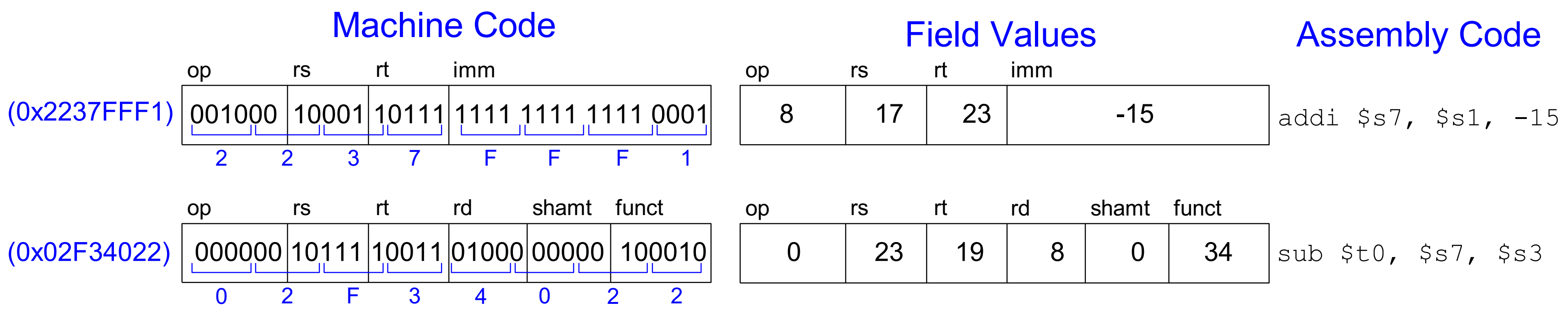
The Stored Program

Assembly Code	Machine Code
`lw   $t2, 32($0)`	`0x8C0A0020`
`add  $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub  $t0, $t3, $t5`	`0x016D4022`

Stored Program



Interpreting Machine Language Code



Machine Code

	op	rs	rt	imm			
(0x2237FFF1)	001000	10001	10111	1111 1111 1111 0001			
	2 2 3 7 F F F 1						
	op	rs	rt	rd	shamt	funct	
(0x02F34022)	000000	10111	10011	01000	00000	100010	
	0 2 F 3 4 0 2 2						

Field Values

op	rs	rt	imm		
8	17	23	-15		
op	**rs**	**rt**	**rd**	**shamt**	**funct**
0	23	19	8	0	34

Assembly Code

```
addi $s7, $s1, -15
sub $t0, $s7, $s3
```

- **Start with opcode**
  - Opcode tells how to parse the remaining bits
- **If opcode is all 0's**
  - R-type instruction
  - Function bits tell what instruction it is
- **Otherwise**
  - opcode tells what instruction it is

What did we learn?

- **Addressing types**
  - Byte addressable (MIPS is byte addressable)
  - Word addressable
- **Three different types of MIPS instructions**
  - R-type: operates on registers
  - I-type: immediate type, using constants
  - J-type: for jump instructions, 24 bit address