

ECE-425: MIPS Programming examples & The end of Chapter 2

Prof: Mohamed El-Hadedy

Email: mealy@cpp.edu

Office: 909-869-2594

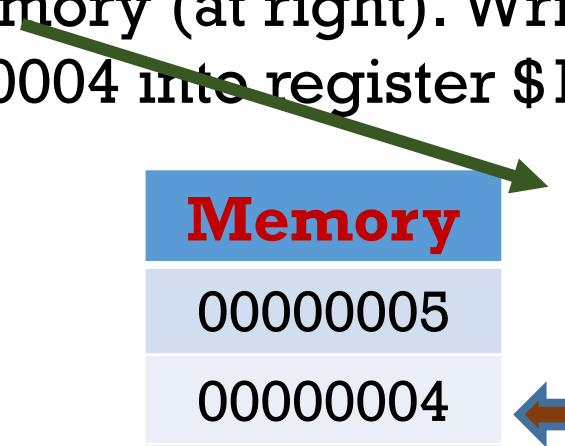
Example:

Look at registers \$12 and \$13 and memory (at right). Write the instruction that puts the value 0x00000004 into register \$12

- Register \$12 contains 0x135FCBA9
- Register \$13 contains 0x00040000

Answer:

lw \$12, 0x10(\$13)



Memory	Addresses
00000005	00040014
00000004	00040010
00000003	0004000C
00000002	00040008
00000001	00040004
00000000	00040000

The original value in \$12 is irrelevant; it is replaced with a value from memory (memory remains unchanged)

Store Word Instruction

The store word instruction (sw) → copies data from a register to memory. The register is not changed. The memory address is specified using a base/register pair.

```
sw  t, off(b)  # word at memory address (b+ff) ← $t
               # b is a register. Off is 16-bit two's complement.
```

As with the lw instruction, the memory address must be word aligned (a multiple of four) (the reason of multiple of four comes from the size of the memory location is 1-byte and the register wise is 4-bytes).

Example:

The store word instruction (sw) → copies data from a register to memory. The register is not changed. The memory address is specified using a base/register pair.

```
sw  t, off(b)  # word at memory address (b+ff) ← $t
               # b is a register. Off is 16-bit two's complement.
```

As with the lw instruction, the memory address must be word aligned (a multiple of four) (the reason of multiple of four comes from the size of the memory location is 1-byte and the register wise is 4-bytes).

Example:

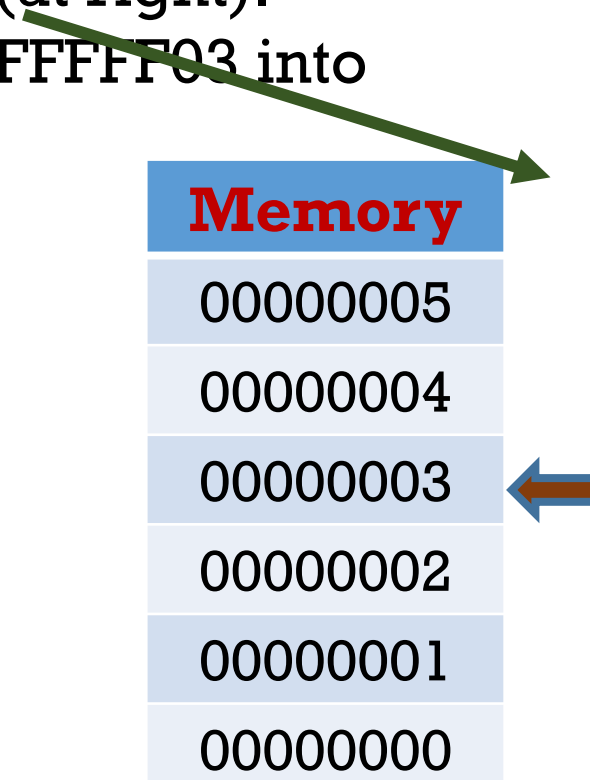
Look at registers \$12 and \$13 and memory (at right).
Write the instruction that puts the word 0xFFFFF03 into
memory location 0x0004000C.

- Register \$12 contains 0xFFFFF03
- Register \$13 contains 0x00040014

Solution:

Sw \$12, 0xFFF8(\$13)

(0x0004000C-0x00040014)



Memory	Addresses
00000005	00040014
00000004	00040010
00000003	0004000C
00000002	00040008
00000001	00040004
00000000	00040014

Setting up the Base Register

The first instruction of the answer expresses minus eight using 16-bit two's complement. This is the bit pattern that is actually contained in the machine instruction. This is awkward to read and to calculate.

The second instruction uses signed decimal notation to specify minus eight. The assembler translates this assembler instruction into exactly the same machine instruction as the first assembler instruction.

By using a 32-bit base register and an offset a 32-bit **lw** or **sw** instruction can reference all of memory.

But how does the base address get into the base register?

This is where the **lui** (load upper immediate) instruction is useful. It copies its 16-bit immediate operand to the *upper two bytes* of the designated register.

lui d,const # upper two bytes of \$d ← two byte const
lower two bytes of \$d ← 0x0000

Example:

lui \$13, **Unknown**(????)

lw \$12, 0x10 (\$13)

The base register contains the address 0x00040000, so complete the lui instruction above

Solution:

Unknown location is **0x0004**

After the **lui** instruction \$13 contains 0x00040000. To get the address we need to use an offset of 0x10

Memory	Addresses
00000005	00040014
00000004	00040010
00000003	0004000C
00000002	00040008
00000001	00040004
00000000	00040000



Filling in the bottom half

By using the **lui** instruction, the base register can be loaded with multiples of 0x00010000. But often you want a more specific address in the base register. Use the **ori** instruction to fill the bottom 16 bits. Recall that:

Ori d,s, imm # zero-extends imm to 32 bits then does a bitwise OR of that with the contents of register \$s. The result goes into register \$d.

Example:

Say that memory is as above. The lw instruction (below) loads the word at 0x0060500C into \$12.

```
lui $13, 0x-----①  
ori $13, $13, 0x-----②  
lw $12, 0xc($13)
```

Complete the instruction sequence so that the base
The address 0x00605000, which will be used with
A displacement of 0x0c.

Solution:

- ① 0x0060
- ② 0x5000

Memory		Addresses
00000005		00040014
00000004	←	00040010
00000003		0004000C
00000002		00040008
00000001		00040004
00000000		00040000

Alternate Sequence

The above **ori** instruction "fills in" the lower 16 bits of register **\$13** by doing the following:

\$13 after lui	:	0000 0000 0110 0000 0000 0000 0000 0000
zero-extended imm. op.	:	0000 0000 0000 0000 0101 0000 0000 0000 (2 in previous slide)
result of bitwise OR	:	0000 0000 0110 0000 0101 0000 0000 0000

Other sequences of instructions also will work: Because the "upper half" of an address is 16 bits and the offset of the **lw** instruction is 16-bits, the two in combination can address any byte of memory.

Lui \$13, 0x0060

lw \$12, 0x500c(\$13)

The problem was to load **\$12** with the word at address 0x0060500C. Here is another way to do it: Split the address into halves: 0x0060 and 0x500C. Load the top half into **\$13** and use the bottom half as the offset.

Choose whichever method works best in the context of the rest of the program

Trick 😊

Since the ori instruction is used often with destination register as one of the operands, there is a shorthand instruction in assembly language.

ori \$d, const ⇔ **ori \$d, \$d, const**

Example:

Do the following two assembly instructions assemble to the same machine instruction?

ori \$10, \$10, 0x00c4

ori \$10, \$0 , 0x00c4

Solution:

No 😊. The first keeps the upper half of the \$10 the same and Ors in some bits in the lower half. For the second instruction, it replaces all 32-bits of \$10 with the zero-extended immediate operand.

Assembly Arrays

An array of int is implemented as a sequence of words in successive word-aligned memory locations. For example, the diagram shows a possible implementation of:

```
int data[] = {0, 1, 2, 3, 4, 5};
```

The above is expressed using the language C, but you don't have to know any C to understand the example. The declaration creates an array of six elements and initializes them to the integers zero through five.

Question:

What is the most sensible address to have in the base register for processing this array?

Answer:

The address of data[0]: **0x00605000**. In fact, in ANSI C, the identifier for an array (in this case *data*) stands for the address of its first element. At run time this address will likely be in a base register.

Memory		Addresses
00000005		00605014
00000004	←	00605010
00000003		0060500C
00000002		00605008
00000001		00605004
00000000		00605000

Example Program

Write using MIPS assembly a program to evaluate the polynomial $5x^2 - 12x + 97$. The value x is located in memory. Store the result at location `poly` in memory.

Question:

- 1) How many `lw` instructions will be needed?
- 2) How many `sw` instructions will be needed?

Answer:

- 1) One, near the start of the program to load x into a register.
- 2) One, near the end of the program to save the result in `poly`.

Symbolic Address

Memory locations are **x** and **poly**. Addresses are 32-bit patterns.

```
## poly.asm
##
## evaluate  $5x^2 - 12x + 97$ 
##
```

```
    .text
    .globl main
main:
```

```
    . . . . many instructions
    .data
```

```
    # In SPIM, the data section
    # starts at address 0x10000000
```

```
x:    .word 0x11 # The base register points here.
```

```
poly:    .word 0x00
## End of file
```

.data means: *here is the start of the data section of memory*

.word means: *put a 32-bit two's complement integer here. The integer is specified using base 10 (by default).*

calls for a 32-bit two's complement representation of an integer that in base 10 is "17", which is 0x11 in hex

Program (cont)

The assembler in SPIM automatically assembles the .data section starting at address 0x10000000

- 1) What address corresponds to the symbolic address **x**?
- 2) What address corresponds to the symbolic address **poly**?

Answer:

- 1) 0x10000000 → SPIM automatically assembles code starting at address 0x10000000
- 2) **0x10000004**

Program (cont)

```
## poly.asm -- complete program
##
## evaluate 5x^2 -12x + 97
##
## Register Use:
##
## $10 base register, address of x
## $11 x
## $12 value of the polynomial
## $13 temporary

        .text
        .globl main

main:
    lui    $10,0x1000    # Init base register
    lw     $11,0($10)    # Load x

    ori    $12,$0,97     # Initialize the accumulator
                        # during the "load delay slot"

    ori    $13,$0,12     # evaluate second term
    mult   $11,$13        # 12x
    mflo   $13            # assume 32 bit result
    subu   $12,$12,$13    # accumulator = -12x +97

    mult   $11,$11        # evaluate third term
                        # x^2
    mflo   $11            # assume 32 bit result
    ori    $13,$0,5      # 5
    mult   $11,$13        # 5x^2
    mflo   $13            #
    addu   $12,$12,$13    # accumulator = 5x^2-12x+97

    sw     $12,4($10)     # Store result in poly

        .data
x:       .word    17      # Edit this line to change the value of x
poly:    .word    0       # Result is placed here.

## End of file
```


The Intel x86 ISA

- Evolution with backward compatibility
 - 8080 (1974): 8-bit microprocessor
 - Accumulator, plus 3 index-register pairs
 - 8086 (1978): 16-bit extension to 8080
 - Complex instruction set (CISC)
 - 8087 (1980): floating-point coprocessor
 - Adds FP instructions and register stack
 - 80286 (1982): 24-bit addresses, MMU
 - Segmented memory mapping and protection
 - 80386 (1985): 32-bit extension (now IA-32)
 - Additional addressing modes and operations
 - Paged memory mapping as well as segments

The Intel x86 ISA

- Further evolution...
 - i486 (1989): pipelined, on-chip caches and FPU
 - Compatible competitors: AMD, Cyrix, ...
 - Pentium (1993): superscalar, 64-bit datapath
 - Later versions added MMX (Multi-Media eXtension) instructions
 - The infamous FDIV bug
 - Pentium Pro (1995), Pentium II (1997)
 - New microarchitecture (see Colwell, *The Pentium Chronicles*)
 - Pentium III (1999)
 - Added SSE (Streaming SIMD Extensions) and associated registers
 - Pentium 4 (2001)
 - New microarchitecture
 - Added SSE2 instructions

The Intel x86 ISA

- And further...
 - [AMD64 \(2003\): extended architecture to 64 bits](#)
 - EM64T – Extended Memory 64 Technology (2004)
 - AMD64 adopted by Intel (with refinements)
 - Added SSE3 instructions
 - Intel Core (2006)
 - Added SSE4 instructions, virtual machine support
 - [AMD64 \(announced 2007\): SSE5 instructions](#)
 - [Intel declined to follow, instead...](#)
 - Advanced Vector Extension (announced 2008)
 - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
 - Technical elegance \neq market success

Basic x86 Registers

Name	31	0	Use
EAX			GPR 0
ECX			GPR 1
EDX			GPR 2
EBX			GPR 3
ESP			GPR 4
EBP			GPR 5
ESI			GPR 6
EDI			GPR 7
	CS		Code segment pointer
	SS		Stack segment pointer (top of stack)
	DS		Data segment pointer 0
	ES		Data segment pointer 1
	FS		Data segment pointer 2
	GS		Data segment pointer 3
EIP			Instruction pointer (PC)
EFLAGS			Condition codes

Basic x86 Addressing Modes

- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
 - Address in register
 - $\text{Address} = R_{\text{base}} + \text{displacement}$
 - $\text{Address} = R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}}$ (scale = 0, 1, 2, or 3)
 - $\text{Address} = R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}} + \text{displacement}$

x86 Instruction Encoding

a. JE EIP + displacement

4	4	8
JE	Condition	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

- Variable length encoding
 - Postfix bytes specify addressing mode
 - Prefix bytes modify operation
 - Operand length, repetition, locking, ...

Implementing IA-32

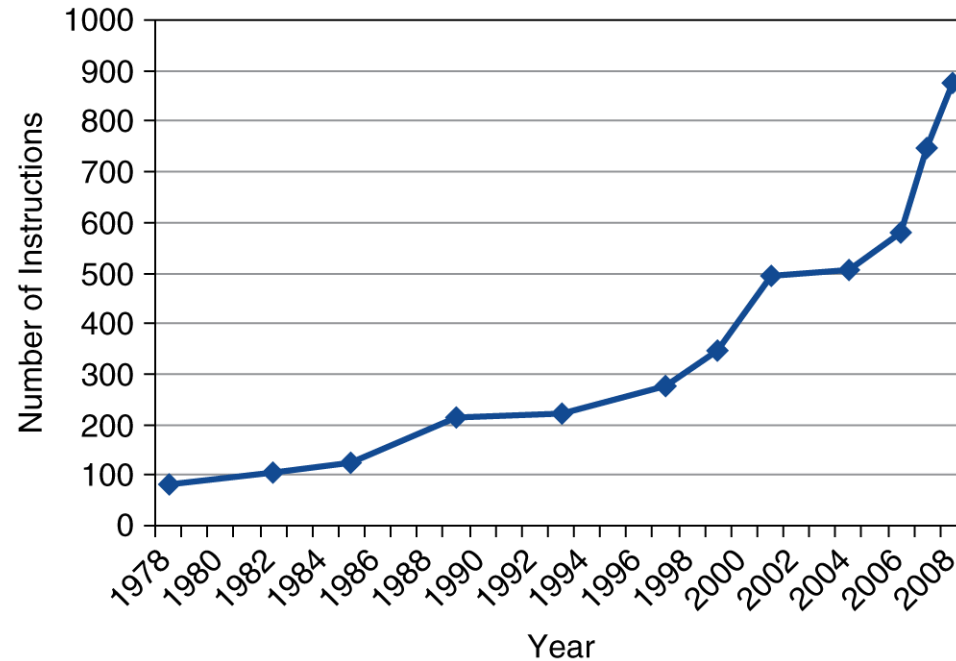
- Complex instruction set makes implementation difficult
 - Hardware translates instructions to simpler microoperations
 - Simple instructions: 1–1
 - Complex instructions: 1–many
 - Microengine similar to RISC
 - Market share makes this economically viable
- Comparable performance to RISC
 - Compilers avoid complex instructions

Fallacies

- Powerful instruction \Rightarrow higher performance
 - Fewer instructions required
 - But complex instructions are hard to implement
 - May slow down all instructions, including simple ones
 - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
 - But modern compilers are better at dealing with modern processors
 - More lines of code \Rightarrow more errors and less productivity

Fallacies

- Backward compatibility \Rightarrow instruction set doesn't change
 - But they do accrete more instructions



x86 instruction set

Concluding Remarks

- Design principles
 1. Simplicity favors regularity
 2. Smaller is faster
 3. Make the common case fast
 4. Good design demands good compromises
- Layers of software/hardware
 - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
 - c.f. x86

Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
 - Consider making the common case fast
 - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	add, sub, addi	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	12%	4%
Cond. Branch	beq, bne, slt, slti, sltiu	34%	8%
Jump	j, jr, jal	2%	0%

Alternative Architectures

- Design alternative:
 - provide more powerful operations
 - goal is to reduce number of instructions executed
 - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as “RISC vs. CISC”

CISC Machine

- 32-bit machine
- Words can be 1, 8, 16, 32 & 64
- >70 distinct instructions (long, byte,..)
- Memory-mapped I/O
- Fixed point instruction set
- Micro programmed control unit

CISC Machine (Cont.)

- Different types of operands
- Different addressing modes
 - direct
 - indirect
 - Immediate
- Instructions can access memory directly

Instruction Formats

- Instruction: operation & operands
- Smaller instruction size – less program storage space
- Different sizes and formats (CISC vs. RISC)
- CISC:
 - Instructions are M to R, M to M or R to R
 - Various addressing modes

Instruction Formats (Cont.)

- RISC Vs. CISC:
 - PCU of CISC is more complicated
 - RISC instructions are simpler, fixed length
 - Memory addressing in RISC is limited to load and store
 - Single cycle execution is easier in RISC
- Most RISC inst. are R to R

Numbers of Addresses

- CISC: different inst. length with different number of addresses
- Fewer addresses – longer programs
- Large instructions – complex decoding and processing circuits
- Most instructions requires 3 addresses
 - One address: ADD x
 - Two addresses: ADD x,y
 - Zero addressing: push down stack

RISC Vs. CISC

- Complex instruction Vs. multi-instructions
- RISC
 - Fewer instructions
 - Fixed and easily decoded instruction formats
 - Single cycle execution
 - Hardwired rather than microprogrammed
 - Limited memory access
 - Compiler Optimization

Characteristics of Classic CISC and Pure RISC Architecture

Characteristics	Classic CISC Architecture	Pure RISC Architecture
Instruction format	Variable formats: 16 to 32 and 64 b	Fixed 32-b instructions
Clock rate*	100-266 MHz	180-500 MHz
Register files	8-24 general-purpose registers (GPRs)	32-192 GPRs, separate integer and floating-point files
Instruction set size and types	Around 300, with over four dozen instruction types	Around 100, most are register-based except <i>load/store</i>
Addressing modes	Around one dozen, including indirect/indexed addressing	Limited to 3 to 5, only <i>read/ store</i> addressing the memory
Cache design	Earlier model used unified cache, some use split caches	Most use split data cache and instruction cache
CPI, and average CPI	1 to 20 cycles, average 4 cycles	1 cycle for simple operations, average around 1.5 cycles
CPU control	Most microcoded, some use hardwired control	Most hardwired control without control memory
Representative commodity processors	Intel x86, VAX 8600, IBM 390, MC 68040, Intel Pentium, AMD 486, and Cyrix 686	Sun UltraSparc, MIPS R10000, PowerPC 604, HP PA-8000, Digital 21164

Reference:

- **Book:** D. A. Patterson and J. L. Hennessy, **Computer Organization and Design: The Hardware/ Software Interface**, 5th Edition, San Mateo, CA: Morgan and Kaufmann. ISBN: 1-55860-604-1
- <https://www.mips.com/>
- <https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html>
- **Simulator:** <http://spimsimulator.sourceforge.net/>
- <https://www.usna.edu/Users/cs/lmcdowel/courses/si232/spim/PCSPIM-HowTo.htm>
- <http://chortle.ccsu.edu/assemblytutorial/index.html#part4>
- Professor El-Naga ECE-425 notes