

---

# EECE 321: Computer Organization

Mohammad M. Mansour  
*Dept. of Electrical and Compute Engineering*  
*American University of Beirut*

Lecture 5: Machine Instructions

---

# Announcements

---

- HW2 due on Friday
- Reading assignment
  - Sections 2.1, 2.2, 2.3
- Project group member name due today
- Makeup sessions
  - Thursday or Saturday

# MIPS ISA

---

- MIPS
  - Microprocessor without Interlocked Pipeline Stages
  - A semiconductor company that built one of the first commercial RISC architectures
  - We will study the MIPS architecture in detail in this class
- Why MIPS instead of Intel 80x86?
  - MIPS is simple, elegant. Don't want to get bogged down in gritty details.
  - MIPS widely used in embedded applications (e.g., NEC, Nintendo, Silicon Graphics, Sony)
  - x86 rarely used in embedded computers
  - There more embedded computers than PCs !



Most HP LaserJet workgroup printers are driven by MIPS-based™ 64-bit processors.

# Assembly Variables: Registers

---

- Unlike high-level languages like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Assembly Operands are Registers
  - Limited number of special locations built directly into the hardware
  - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they are very fast (faster than 1 billionth of a second)
- Drawback: Since registers are in hardware, they are limited
  - Solution: MIPS code must be very carefully put together to efficiently use registers
- 32 registers in MIPS
  - Why 32? Smaller is faster
- Each MIPS register is 32 bits wide
  - Groups of 32 bits form a word in MIPS
- Registers are numbered from 0 to 31
  - Each register can be referred to by number or name
  - Number references (convention):  
\$0, \$1, \$2, ... , \$30, \$31

# Assembly Variables: Registers

---

- By convention, each register also has a name to make it easier to code
- For now:
  - \$16 – \$23 → \$s0 – \$s7  
(correspond to **C variables**)
  - \$8 – \$15 → \$t0 – \$t7  
(correspond to **temporary variables**)
- Later will explain the other 16 register names
- In general, use names to make your code more readable

# Assembly Language vs. C/C++, Java

---

- Statements in an assembly language are instructions. They execute exactly one of a short list of simple commands.
- Unlike in C, Java (and most other High Level Languages), each line of assembly code contains at most 1 instruction.
- Comments in Assembly:
  - Hash (#) is used for MIPS comments, anything from hash mark to end of line is a comment and will be ignored
  - Not like C comments which can span multiple line `/* comment */`
- In C, Java (and most HLLs) variables are declared first and given a type
  - Example:  
`int fahr, celsius;`  
`char a, b, c, d, e;`
- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match `int` and `char` variables).
- In Assembly Language, the registers have no type; operation determines how register contents are treated
  - There are no types associated with variables – the types are associated with the instructions.
- Instructions are related to operations (`=, +, -, *, /`) in C/C++ or Java

# MIPS Addition and Subtraction

---

- Syntax of Instructions:

1 2,3,4

where:

1) operation name

2) operand getting result (“destination”)

3) 1st operand for operation (“source1”)

4) 2nd operand for operation (“source2”)

- Syntax is rigid:

- 1 operator, 3 operands

- Why? Keep Hardware simple via regularity

- Addition in Assembly

- Example:           add \$s0, \$s1, \$s2                   # in MIPS

- Equivalent to:    a = b + c                           /\* in C \*/

where MIPS registers \$s0, \$s1, \$s2 are associated with C variables a, b, c

- Subtraction in Assembly

- Example:           sub \$s3, \$s4, \$s5                   # in MIPS

- Equivalent to:    d = e – f                           /\* in C \*/

where MIPS registers \$s3, \$s4, \$s5 are associated with C variables d, e, f

# Compiling C statements into Assembly

---

- Compile the following C statement into MIPS Assembly
  - $a = b + c + d - e;$
- Break into multiple instructions:
  - `add $t0, $s1, $s2`       $\# \text{temp} = b + c$
  - `add $t0, $t0, $s3`       $\# \text{temp} = \text{temp} + d$
  - `sub $s0, $t0, $s4`       $\# a = \text{temp} - e$
- Notice: A single line of C may break up into several lines of MIPS.
- Compile the following C statement into MIPS Assembly
  - $f = -2 * g;$
- Use intermediate temporary registers
  - `add $t0, $s1, $s1`       $\# \text{temp0} = 2 * g$
  - `add $t1, $t0, $t0`       $\# \text{temp1} = 4 * g$
  - `sub $s2, $t0, $t1`       $\# f = 2 * g - 4 * g$

# What About Immediate Operands?

---

- One particular immediate, the number zero (0), appears very often in code.
- So we define register zero (`$0` or `$zero`) to always have the value 0; e.g.
  - `add $s0, $s1, $zero`      *# in MIPS*
  - `f = g`      */\* in C \*/*where MIPS registers `$s0`, `$s1` are associated with C variables `f`, `g`
- `$zero` is defined in hardware, so an instruction
  - `add $zero, $zero, $s0`  
will not do anything if the destination address is the register `$zero`!
- In general, immediates are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:
  - `addi $s0, $s1, 10`      *# in MIPS: add the immediate constant 10 to contents of \$s1*
  - `f = g + 10`      */\* in C \*/*where MIPS registers `$s0`, `$s1` are associated with C variables `f`, `g`
- Syntax similar to add instruction, except that last argument is a number instead of a register.

# Immediates

---

- There is no Subtract Immediate in MIPS: Why?
- Limit types of operations that can be done to absolute minimum
  - if an operation can be decomposed into a simpler operation, don't include it
  - `addi ..., -X` is equivalent to `subi ..., X` => so no `subi`
- `addi $s0, $s1, -10`                      # in MIPS
- `f = g - 10`                                  /\* in C \*/  
where MIPS registers `$s0, $s1` are associated with C variables `f, g`

# Overflow in Arithmetic

---

- Reminder: Overflow occurs when there is a mistake in arithmetic due to the limited precision in computers.

- Example (4-bit unsigned numbers):

+15	1111
+3	0011
-----	-----
+18	10010

- But we don't have room for 5-bit solution, so the solution would be 0010, which is +2, and wrong.
- Some languages detect overflow (Ada), some don't (like C)
- MIPS solution is 2 kinds of arithmetic instructions to recognize 2 choices:
  - add (**add**), add immediate (**addi**) and subtract (**sub**) cause overflow to be detected
  - add unsigned (**addu**), add immediate unsigned (**addiu**) and subtract unsigned (**subu**) do not cause overflow detection
- Compiler selects appropriate arithmetic
- MIPS C compilers produce
  - **addu, addiu, subu**

# Logic Instructions

---

- Logical Shift Left:
  - `sll $s1, $s2, 2`     $\# s1 = s2 \ll 2$
  - Stores in `$s1` the value from `$s2` shifted 2 bits to the left, inserting 0's on right;
  - Equivalent to the "`<<`" operator in C
- Example:
  - `$s2 = 0x0000 0002 = 0000 0000 0000 0000 0000 0000 0000 0010(2)`
  - `sll $s1, $s2, 2`
  - `$s1 = 0x0000 0008 = 0000 0000 0000 0000 0000 0000 0000 1000(2)`
  - Operation equivalent to multiplication by 4
- What arithmetic effect in general does shift left have?
  - Answer: Multiplication by a power of 2
- Shift Right (`srl`):
  - `srl` is opposite shift
  - Equivalent to "`>>`" in C
  - Arithmetic effect: Divide by a power of 2, then take the floor
    - Ex: `$s2 = 0x0000 004C = 0000 0000 0000 0000 0000 0000 0100 1100(2)`
    - `sll $s1, $s2, 2 => $s1 = 0x0000 0013 = 0000 0000 0000 0000 0000 0000 0001 0011(2)`
  - Note if LSBs are not zero, right-shifting is equivalent to dividing by a power of 2 then taking the floor.

# Summary of Instructions So Far

---

- To summarize, in MIPS Assembly Language:
  - Registers replace C variables
  - One Instruction (simple operation) per line
  - Simpler is Better
  - Smaller is Faster
- New Instructions:
  - Arithmetic: `add`, `addi`, `sub`, `addu`, `subu`, `addiu`
  - Logical: `sll`, `srl`
- New Registers:
  - C Variables: `$s0` - `$s7`
  - Temporary Variables: `$t0` - `$t9`
  - Zero: `$zero`

# Assembly Operands: Memory

---

- C variables are mapped onto registers.
  - What about large data structures like arrays?
- 1 of 5 components of a computer: memory contains such data structures
- But MIPS arithmetic instructions only operate on registers, never directly on memory.
- Data transfer instructions transfer data between registers and memory:
  - Memory to register: **LOAD**
  - Register to memory: **STORE**
- Registers are in the datapath of the processor
  - If operands are in memory, we must transfer them to the processor to operate on them, and then transfer back result to memory when done.

