Lecture 2: 
Instruction Set Architectures (ISA) and MIPS Assembly language

Assembly Language

• Basic job of a CPU: execute lots of instructions.
• Instructions are the primitive operations that the CPU may execute.
• Von Neumann architecture model of program execution:
  - Stored program model: instructions and data are stored in memory
  - Fetch/execute cycle: instructions are fetched from memory to the CPU and executed by the hardware

Basic Instruction Cycle

Instruction Execution

Instruction Set Architectures

• Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an Instruction Set Architecture (ISA).
  - Examples: Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS, Intel IA64, ...

Instruction Set Architectures - History

• Accumulator ISA
  - One register in the CPU for arithmetic called the accumulator
    LOAD ACC.X
    ADD ACC.Y
    STORE ACC.Z

• Stack ISA
  - A stack is used for arithmetic
    PUSH X
    PUSH Y
    ADD
    POP Z
**Instruction Set Architectures**

- General Purpose Register ISA
- Three types based on where operands for arithmetic operations can come from.
  - Memory-memory
  - Register-memory
  - Register-register (Load/Store)

MIPS has a Load/Store ISA

---

**RISC Architectures**

- Fixed instruction lengths
- Load/store instruction sets
- Limited addressing modes (ways to access variables in memory)
- Limited operations

- Sun SPARC, IBM PowerPC, MIPS

---

**MIPS Architecture**

- MIPS – semiconductor company that built one of the first commercial RISC architectures
- We will study the MIPS architecture in some detail in this class (also used in CIS 429 Computer Architecture - Spring ‘05)
- 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 apps, x86 little used in embedded, and more embedded computers than PCs

---

**Assembly Variables: Registers (1/4)**

- Unlike HLL 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)

---

**Assembly Variables: Registers (2/4)**

- Drawback: Since registers are in hardware, there are a predetermined number of them
  - 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 called a **word** in MIPS
Assembly Variables: Registers (3/4)

- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- Number references:
  \$0, \$1, \$2, ... \$30, \$31

Assembly Variables: Registers (4/4)

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

C, Java variables vs. registers

- In C (and most High Level Languages) variables 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

MIPS data in memory vs. registers

- In MIPS, you can declare memory variables using .data
- Each item is one word
- Give it a symbolic name
- Give it an initial value

  .data
  One: .word 1 # first value, initialized to 1
  Two: .word 2 # second value, initialized to 2

Comments in Assembly

- Another way to make your code more readable: comments!
- Hash (#) is used for MIPS comments
  - anything from hash mark to end of line is a comment and will be ignored
- Note: Different from C.
  - C comments have format */ comment */
  - so they can span many lines

Assembly Instructions

- In assembly language, each statement (called an instruction), executes exactly one of a short list of simple commands
- Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, *, /) in C or Java
Assembly Instructions for memory access

- **LW** $s3, X         # Load Word
  Loads one word of data from memory location X into register $S3

- **SW** $s4, Y         # Store Word
  Stores one word of data from register $S4 into memory location Y

MIPS Addition and Subtraction (1/4)

- Syntax of Instructions:
  - **OP** Dest,Src1,Src2
  - where:
    - **OP** operation by name
    - Dest) operand getting result (“destination”)
    - Src1) 1st operand for operation (“source1”)
    - Src2) 2nd operand for operation (“source2”)

- Syntax is rigid:
  - 1 operator, 3 operands
  - Why? Keep Hardware simple via regularity

Addition and Subtraction of Integers (2/4)

- **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 $d, e, f$

Addition and Subtraction of Integers (3/4)

- **How do the following C statement?**
  - $a = b + c + d - e$

  Break into multiple instructions
  - **add** $t0, $s1,$s2 # temp = b + c
  - **add** $t0, $t0,$s3 # temp = temp + d
  - **sub** $s0, $t0,$s4 # $a = temp - e$

- Notice: A single line of C may break up into several lines of MIPS.

- Notice: Everything after the hash mark on each line is ignored (comments)

Addition and Subtraction of Integers (4/4)

- **How do we do this?**
  - $f = (g + h) - (i + j)$;

- Use intermediate temporary register
  - **add** $t0,$s1,$s2 # temp = g + h
  - **add** $t1,$s3,$s4 # temp = i + j
  - **sub** $s0,$t0,$t1 # $f=(g+h)-(i+j)$

Register Zero

- 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; eg
  - **add** $s0,$s1,$zero (in MIPS)
  - $f = g$ (in C)
  - where MIPS registers $s0, $s1 are associated with C variables $f, g$

- defined in hardware, so an instruction
  - **add** $zero,$zero,$s0
  - will not do anything!
Immediates

• Immediates are numerical constants.
• They appear often in code, so there are special instructions for them.
• Add Immediate:
  \[
  \text{addi } s0, s1, 10 \quad \text{(in MIPS)}
  \]
  \[f = g + 10 \quad \text{(in C)}\]
  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
  • \text{addi ..., -X = subi ..., X = so no subi}
  • \text{addi } s0, s1, -10 \quad \text{(in MIPS)}
  \[
  f = g - 10 \quad \text{(in C)}
  \]
  where MIPS registers \(s0, s1\) are associated with C variables \(f, g\)

Summary

• Instruction set architecture (ISA)
  • The design of the basic machine level instructions understood by a computer
  • Assembly lang. v. machine language
• RISC (Reduced Instruction Set Computers)
  • Simple, efficient design for the basic building blocks
  • Push complexity up a level to SW and compiler optimizations

Summary (cont.)

• In MIPS Assembly Language:
  • Registers replace C and Java variables
  • One Instruction (simple operation) per line
  • Simpler is Better
  • Smaller is Faster
• New Instructions:
  \(\text{lw, sw, add, addi, sub}\)
• New Registers:
  Persistent Variables: \(s0 - s7\)
  Temporary Variables: \(t0 - t9\)
  Zero: \(\$0\)

I. Instruction Execution

Basic Instruction Cycle

\[\text{Figure 1.1. Computer Components, Real-Level View}\]

\[\text{Figure 1.2. Basic Instruction Cycle}\]
Program Execution

• My First MIPS program
• Simple example of a SPIM program -- compute the polynomial
  \[ y = ax^2 + bx + c \]

Program Execution (Text)

• 4. EXECUTE: The contents of the AC and the contents of location 941 are added and stored back into the AC. AC now has the value "$005$

• 5. FETCH: The PC is incremented to 302 and the next instruction is fetched.

• Comment: The first four bits ("2") give the opcode for "STORE." The remaining twelve bits ("941") give the address of where to store the result.

• 6. EXECUTE: The contents of the AC are stored at address 941.

```mips
lw $s0,One       # first operand
lw $s1,Two       # second operand
add $s2,$s0,$s1   # add
# move $s0,$s2    # pass result to syscall
li $v0, 1        # specify syscall is a write integer
syscall          # execute syscall
# all done....
li $v0, 10       # adios....
syscall
```

Program Execution

```mips
lw $s0,One       # first operand
lw $s1,Two       # second operand
add $s2,$s0,$s1   # add
move $s0,$s2     # pass result to syscall
li $v0, 1        # specify syscall is a write integer
syscall          # execute syscall
# all done....
li $v0, 10       # adios....
syscall
```
# compute polynomial
mul $t0, $s0, $s0   # t0 = x^2
mul $t0, $t0, $s1   # t0 = a*x^2
mul $t1, $s0, $s2   # t1 = b*x
add $t0, $t0, $t1   # t0 = a*x^2 + b*x
add $t0, $t0, $s3   # done -- store final result in y
sw $t0, Y

# print result
move $a0, $t0        # copy $t0 to $a0
li $v0, 1
syscall

# exit program
li $v0, 10         # "syscall"
syscall

# data section
.data
X: .word 7
A: .word 3
B: .word 4
C: .word 5
Y: .word 0