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

Figure 1.2 Basic Instruction Cycle
• 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
- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- Number references: $0, \; \$1, \; \$2, \; \ldots \; \$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)

  Later will explain other 16 register names

• 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
    • e.g. 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),
  • It 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
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: \texttt{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: \texttt{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

```c
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)
• How do we do this?

\[ f = (g + h) - (i + j); \]

• Use intermediate temporary register

```assembly
add $t0,$s1,$s2  # temp = g + h
add $t1,$s3,$s4  # temp = i + j
sub $s0,$t0,$t1  # f=(g+h)-(i+j)
```
Immediates

• Immediates are numerical constants.

• They appear often in code, so there are special instructions for them.

• Add Immediate:

  addi  $s0,$s1,10  (in MIPS)

  \( f = g + 10 \) (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 = 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 $f, g$
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
  \[ \text{add } $s0,$s1,$zero \] (in MIPS)
  \[ f = g \] (in C)
  where MIPS registers $s0, $s1 are associated with C variables \( f, g \)
- Defined in hardware, so an instruction
  \[ \text{add } $zero,$zero,$s0 \]
  will not do anything!
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

- **LI** $s5, 10 # Load immediate
  Loads the value 10 into $s5
# My First MIPS program

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

.text
.globl main

main:
    # load constants and add values
    lw      $s0, One   # first operand
    lw      $s1, Two  # second operand
    add     $s2, $s0, $s1 # add

    # print value
    move    $a0, $s2   # pass result to syscall
    li       $v0, 1    # syscall is a write integer
    syscall  # execute syscall

    # all done....
    li       $v0, 10   # adios....
    syscall
Summary

• In MIPS Assembly Language:
  • Registers replace C and Java variables
  • One Instruction (simple operation) per line
  • Simpler is Better
  • Smaller is Faster

• New Instructions:
  lw, sw, add, addi, sub

• New Registers:
  Persistent Variables: $s0 - $s7
  Temporary Variables: $t0 - $t7
  Zero: $zero