Overview
- C operators, operands
- Variables in Assembly: Registers
- Comments in Assembly
- Addition and Subtraction in Assembly
- Memory Access in Assembly

Review C Operators/Operands (1/2)
- Operators: +, -, *, /, % (mod);
  \( \frac{7}{4} = 1, 7 \% 4 = 3 \)
- Operands:
  - Variables: lower, upper, fahr, celsius
  - Constants: 0, 1000, -17, 15.4
- Assignment Statement:
  - Variable = expression
  - Examples:
    - celsius = 5*(fahr-32)/9;
    - a = b+c+d-e;

Assembly Design: Key Concepts
- Keep it simple!
  - Limit what can be a variable and what can’t
  - 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

Assembly Variables: Registers (1/4)
- Unlike HLL, 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
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 - $22 = $a0 - $a7
    - (correspond to C variables)
  - $8 - $15 = $t0 - $t7
    - (correspond to temporary variables)
- In general, use names to make your code more readable

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

Addition and Subtraction (1/4)

- Syntax of Instructions:
  1 2,3,4
  - where:
  1) operation by 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 and Subtraction (2/4)

° Addition in Assembly
  - Example: add $s0,$s1,$s2 (in MIPS)
  - Equivalent to: \( a = b + c \) (in C)
  - where registers \( s0,s1,s2 \) are associated with variables \( a, b, c \)

° Subtraction in Assembly
  - Example: sub $s3,$s4,$s5 (in MIPS)
  - Equivalent to: \( d = e - f \) (in C)
  - where registers \( s3,s4,s5 \) are associated with variables \( d, e, f \)

Addition and Subtraction (3/4)

° How do the following C statement?
  \( a = b + c + d - e; \)

° Break into multiple instructions
  - add $s0, $s1, $s2 # a = b + c
  - add $s0, $s0, $s3 # a = a + d
  - sub $s0, $s0, $s4 # a = a - 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 (4/4)

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

° Use intermediate temporary register
  - add $s0,$s1,$s2 # f = g + h
  - add $t0,$s3,$s4 # t0 = i + j
  - need to save i+j, but can’t use \# f, so use t0
  - sub $s0,$s0,$t0 # f = (g+h) - (i+j)

Administrivia

° Project 1 due Midnight
° Lab 3: Your first MIPS program!
° HW 2 (due Mon 9/11) and HW3 (9/18) online and available
° Reading assignment:
  - P&H 3.1-3.3, 3.5, 3.8 (page 145)

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 registers \( s0,s1 \) are associated with variables \( f, g \)
° Syntax similar to add instruction, except that last argument is a number instead of a register.

Register Zero

° One particular immediate, the number zero (0), appears very often in code.
° So we define register zero \( (s0 \ or \ $zero) \) to always have the value 0.
° This is defined in hardware, so an instruction like
  - addi $0,$0,5
    - will not do anything.
° Use this register, it’s very handy!
Assembly Operands: Memory

- C variables map 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
- Register to memory

Data Transfer: Memory to Reg (1/4)

- To transfer a word of data, we need to specify two things:
  - Register: specify this by number (0 - 31)
  - Memory address: more difficult
    - Think of memory as a single one-dimensional array, so we can address it simply by supplying a pointer to a memory address.
    - Other times, we want to be able to offset from this pointer.

Data Transfer: Memory to Reg (2/4)

- To specify a memory address to copy from, specify two things:
  - A register which contains a pointer to memory
  - A numerical offset (in bytes)
- The desired memory address is the sum of these two values.

Example: \( 8(\$t0) \)
- specifies the memory address pointed to by the value in \( \$t0 \), plus 8 bytes

Data Transfer: Memory to Reg (3/4)

- Load Instruction Syntax:
  - \( lw \) \( \$1,3(4) \)
  - where
    - 1) operation name
    - 2) register that will receive value
    - 3) numerical offset in bytes
    - 4) register containing pointer to memory

Instruction Name:
- \( lw \) (meaning Load Word, so 32 bits or one word are loaded at a time)

Data Transfer: Memory to Reg (4/4)

- Example: \( lw \) \( \$t0,12(\$s0) \)
  - This instruction will take the pointer in \( \$s0 \), add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register \( \$t0 \)

Notes:
- \( \$s0 \) is called the base register
- 12 is called the offset
- Offset is generally used in accessing elements of array: base reg points to beginning of array

Data Transfer: Reg to Memory (1/2)

- Also want to store value from a register into memory
- Store instruction syntax is identical to Load instruction syntax

Instruction Name:
- \( sw \) (meaning Store Word, so 32 bits or one word are loaded at a time)
Data Transfer: Reg to Memory (2/2)

° Example: \( \text{sw} \ S0,12(Ss0) \)

This instruction will take the pointer in \( Ss0 \), add 12 bytes to it, and then store the value from register \( S0 \) into the memory address pointed to by the calculated sum.

Pointers v. Values

° Key Concept: A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory address), etc.
° If you write \( \text{add} \ S2, S1, S0 \)
  then \( S0 \) and \( S1 \) better contain values
° If you write \( \text{lw} \ S2,0(S0) \)
  then \( S0 \) better contain a pointer
° Don’t mix these up!

Addressing: Byte vs. word

° Every word in memory has an address, similar to an index in an array.
° Early computers numbered words like C numbers elements of an array:
  \( \text{Memory}[0], \text{Memory}[1], \text{Memory}[2], \ldots \)
  \text{Called the \textit{address} of a word}
° Computers needed to access 8-bit bytes as well as words (4 bytes/word)
° Today machines address memory as bytes, hence word addresses differ by 4:
  \( \text{Memory}[0], \text{Memory}[4], \text{Memory}[8], \ldots \)

Compilation with Memory

° What offset in \( \text{lw} \) to select \( A[8] \) in C?
° \( 4 \times 8=32 \) to select \( A[8] \): byte v. word
° Compile by hand using registers:
  \( g = h + A[8] \);
  • \( g: S1, h: S2, S3: \text{base address of } A \)
° 1st transfer from memory to register:
  \( \text{lw} \ S0, 32(S3) \) # \( S0 \) gets \( A[8] \)
  • Add \( 32 \) to \( S3 \) to select \( A[8] \), put into \( S0 \)
° Next add it to \( h \) and place in \( g \)
  \( \text{add} \ S1, S2, S0 \) # \( S1 = h+A[8] \)

Notes about Memory

° Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.
  • Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes.
  • So remember that for both \( \text{lw} \) and \( \text{sw} \), the sum of the base address and the offset must be a multiple of 4 (to be \textit{word aligned})

More Notes about Memory: Alignment

° MIPS requires that all words start at addresses that are multiples of 4 bytes

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

° Called Alignment: objects must fall on address that is multiple of their size.
Role of Registers vs. Memory
° What if more variables than registers?
  • Compiler tries to keep most frequently used variable in registers
  • Writing less common to memory: spilling
° Why not keep all variables in memory?
  • Smaller is faster: registers are faster than memory
  • Registers more versatile:
    - MIPS arithmetic instructions can read 2, operate on them, and write 1 per instruction
    - MIPS data transfer only read or write 1 operand per instruction, and no operation

“And in Conclusion…” (1/2)
° In MIPS Assembly Language:
  • Registers replace C variables
  • One Instruction (simple operation) per line
  • Simpler is Better
  • Smaller is Faster
° Memory is byte-addressable, but lw and sw access one word at a time.
° A pointer (used by lw and sw) is just a memory address, so we can add to it or subtract from it (using offset).

“And in Conclusion…” (2/2)
° New Instructions:
  add, addi,
  sub
  lw, sw
° New Registers:
  C Variables: $s0 - $s7
  Temporary Variables: $t0 - $t9
  Zero: $zero