Hate EMACS? Love EMACS?

Richard M. Stallman, a famous proponent of open-source software, the founder of the GNU Project, and the author of emacs and gcc, will be giving a speech. We're working on securing some type of food for the meeting, but we have secured a raffle prize valued at $100. The raffle will be open to all those who attend, so be sure to come and bring your friends!

Brought to you by CalLUG (UC Berkeley GNU/Linux User Group).
Tuesday, September 20, 6-8 PM in 100 GPB.

Our website with more information can be found at http://linux.berkeley.edu/
There is one handout today at the front and back of the room!

Lecturer PSOE, new dad Dan Garcia

www.cs.berkeley.edu/~ddgarcia

Stolen laptop found! ⇒
Back in March, a laptop with the sensitive info of 98,000 students was stolen from Sproul. It was sold to a man in SF who sold it on Ebay, and was recovered in SC.

idalert.berkeley.edu/update914.html
Review

• Several techniques for managing heap via malloc and free: best-, first-, next-fit
  • 2 types of memory fragmentation: internal & external; all suffer from some kind of frag.
  • K&R, Slab allocator, Buddy system (adaptive)

• Automatic memory management relieves programmer from managing memory.
  • All require help from language and compiler
  • Reference Count: not for circular structures
  • Mark and Sweep: complicated and slow, works
  • Copying: Divides memory to copy good stuff

• In MIPS Assembly Language:
  • One Instruction (simple operation) per line
  • Simpler is better, smaller is faster
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
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 (cannot mix and match int and char variables).

• In Assembly Language, the registers have no type; operation determines how register contents are treated.
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

• Ok, enough already…gimme my MIPS!
MIPS 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 of Integers (2/4)

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

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

\[
\begin{align*}
\text{add } & \quad \text{\$t0, \$s1, \$s2} \quad \# \text{ temp = b + c} \\
\text{add } & \quad \text{\$t0, \$t0, \$s3} \quad \# \text{ temp = temp + d} \\
\text{sub } & \quad \text{\$s0, \$t0, \$s4} \quad \# \text{ a = temp - e}
\end{align*}
\]

• 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

```mips
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 $\texttt{zero}$) 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:

  \( \text{addi} \ $s0,$s1,10 \) (in MIPS)

  \( f = g + 10 \) (in C)

  where MIPS registers \$s0,\$s1 are associated with C variables \( f, g \)

• Syntax similar to \texttt{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 $f, g$
A. Types are associated with *declaration* in C (normally), but *are associated with instruction* (operator) in MIPS.

B. Since there are only 8 local ($s$) and 8 temp ($t$) variables, we can’t write MIPS for C exprs that contain > 16 vars.

C. If $p$ (stored in $s0$) were a pointer to an array of ints, then $p++$; would be

\[
\text{addi } s0 \ s0 \ 1
\]
**Administrivia**

- Project 1 deadline extended until Monday!
  - The Autograder is up!
- `gcc -o foo foo.c`
  - We shouldn’t see any `a.out` files anymore now that you’ve learned this!
- You should be able to finish labs within the allotted time.
  - If you can’t, get checked off for what you have, finish @ home, check off next week
  - If this becomes a pattern, think about working on labs @ home
- HW2 frozen! (1 week regrades start now)
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
Anatomy: 5 components of any Computer

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 to memory when done.

These are “data transfer” instructions…
Data Transfer: Memory to Reg (1/4)

• To transfer a word of data, we need to specify two things:
  • **Register**: specify this by # ($0 - $31) or symbolic name ($s0,…, $t0, …)
  • **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.

Remember: “Load FROM memory”
Data Transfer: Memory to Reg (2/4)

• To specify a memory address to copy from, specify two things:
  • A register containing 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:
  1  2,3(4)
  • where
    1) operation name
    2) register that will receive value
    3) numerical offset in bytes
    4) register containing pointer to memory

• MIPS 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: \texttt{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 or structure: base reg points to beginning of array or structure
Data Transfer: Reg to Memory

- Also want to store from register into memory
  - Store instruction syntax is identical to Load’s

- MIPS Instruction Name:
  
  `sw` (meaning Store Word, so 32 bits or one word are loaded at a time)

- Example: `sw $t0, 12($s0)`

  This instruction will take the pointer in $s0, add 12 bytes to it, and then store the value from register $t0 into that memory address

- Remember: “Store INTO memory”
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), and so on.

• If you write `add $t2,$t1,$t0` then `$t0` and `$t1` better contain values.

• If you write `lw $t2,0($t0)` then `$t0` 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:
  - `Memory[0]`, `Memory[1]`, `Memory[2]`, ...

  Called the “address” of a word

• Computers needed to access 8-bit bytes as well as words (4 bytes/word)

• Today machines address memory as bytes, (i.e., “Byte Addressed”) hence 32-bit (4 byte) word addresses differ by 4
  - `Memory[0]`, `Memory[4]`, `Memory[8]`, ...
Compilation with Memory

• What offset in `lw` to select `A[5]` in C?
• 4x5=20 to select `A[5]`: byte v. word

• Compile by hand using registers:
  \[ g = h + A[5]; \]
  • `g`: $s1, `h`: $s2, `$s3`: base address of `A`

• 1st transfer from memory to register:
  \[ lw \ $t0,20($s3) \quad \# \ \$t0 \ gets \ A[5] \]
  • Add 20 to `$s3` to select `A[5]`, put into `$t0`

• Next add it to `h` and place in `g`
  \[ \text{add } \$s1,$s2,$t0 \quad \# \ \$s1 = h+A[5] \]
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 `lw` and `sw`, the sum of the base address and the offset must be a multiple of 4 (to be word aligned)
More Notes about Memory: Alignment

- MIPS requires that all words start at byte addresses that are multiples of 4 bytes.

<table>
<thead>
<tr>
<th>Aligned</th>
<th>Not Aligned</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Last hex digit of address is:
- 0, 4, 8, or $C_{\text{hex}}$
- 1, 5, 9, or $D_{\text{hex}}$
- 2, 6, A, or $E_{\text{hex}}$
- 3, 7, B, or $F_{\text{hex}}$

- 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
  • Less common in 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
Loading, Storing bytes 1/2

- In addition to word data transfers (lw, sw), MIPS has byte data transfers:
  - load byte: lb
  - store byte: sb
  - same format as lw, sw
Loading, Storing bytes 2/2

• What do with other 24 bits in the 32 bit register?
  • `lb`: sign extends to fill upper 24 bits

```
xxxxx xxxxx xxxxx xxxxx xxxxx xxxxx xzzzz zzzzz
```

...is copied to “sign-extend”

• Normally don't want to sign extend chars

• MIPS instruction that doesn’t sign extend when loading bytes:

  load byte unsigned: `lbu`
“And in conclusion…”

- 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.
  - One can store & load (signed and unsigned) bytes too

- A pointer (used by `lw` & `sw`) is just a mem address, so we can add to it or subtract from it (via offset).

- New Instructions:
  - `add`, `addi`, `sub`, `lw`, `sw`, `lb`, `sb`, `lbu`

- New Registers:
  - C Variables: `$s0 - $s7`
  - Temporary Variables: `$t0 - $t9`
  - Zero: `$zero`