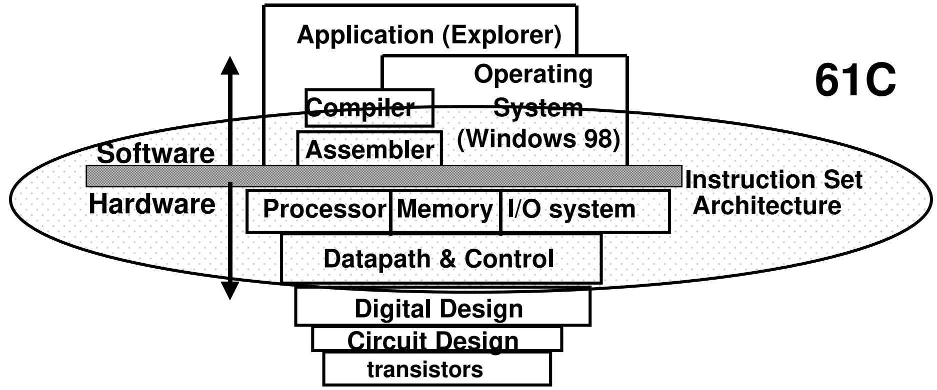
CS61C
C/Assembler Operators and Operands
Lecture 2
January 22, 1999

Dave Patterson
(http.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs61c/schedule.html

"Machine Structures" Review #1/3



- Coordination of many *levels of abstraction*

Review #2/3

- Continued Rapid Improvement in Computing
  - 2X every 1.5 years; processor speed, memory size, disk capacity; Moore's Law as enabler (2X transistors/chip/1.5 yrs)
- 5 classic components of all computers

  1. Control $\left.\right\}$ Processor
  2. Datapath
  3. Memory
  4. Input
  5. Output

Review #3/3

- 15 weeks to learn big ideas in CS&E
  - Principle of abstraction, used to build systems as layers
  - Compilation v. interpretation to move down layers of system
  - Pliable Data: a program determines what it is
  - Stored program concept: instructions are data
  - Principle of Locality, exploited via a memory hierarchy (cache)
  - Greater performance by exploiting parallelism
  - Principles/pitfalls of performance measurement

Overview

- C operators, operands (2 min)
- MIPS Assembly Instructions (5 min)
- Compilation (5 min)
- MIPS Assembly Operands: Registers (5)
- Administrivia, “Computers in the News” (5 min)
- MIPS Assembly Operands: Memory (20)
- C/Assembly Examples (5 min)
- Conclusion (2 min)

C Operators/Operands

- Arithmetic operators: `+`, `-`, `*`, `/`, `%` (mod)
- Assignment: Variable = expression
- Operands:
  - Variables: `lower`, `upper`, `fahr`, `celsius`
  - Constants: e.g., `0`, `1000`, `-17`, `15`
- In C (and most High Level Languages) variables declared 1st and given a type
  - Example:

```
int fahr, celsius;
int a, b, c, d, e;
```


C Operators/Operands Examples

```
a = b + c;

a = b + c + d - e;

celsius = 5 * (fahr - 32) / 9;
```

- Called <u>Assignment statements</u>

Assembly Operators

- Syntax of Assembly Operator
  1) operation by name
  2) operand getting result
  3) 1st operand for operation
  4) 2nd operand for operation
- Ex. add b to c and put the result in a: `add a, b, c`
  - Called an (assembly language) <u>Instruction</u>
- Equivalent assignment statement in C:

```
a = b + c;
```

Assembly Operators/Instructions

- MIPS Assembly Syntax is rigid: 1 operation, 3 variables
  - Why? Keep Hardware simple via regularity
- How do following C statement?

```
a = b + c + d - e;
```

- Break into multiple instructions

```
add a, b, c # a = sum of b & c
add a, a, d # a = sum of b,c,d
sub a, a, e # a = b+c+d-e
```

- To right of sharp (#) is a comment terminated by end of the line
  - C comments have format `/* comment */`, can span many lines

Assembly Operators/Intructions

- Note: Unlike C (and most other HLL), each line of assembly contains at most 1 instruction

Compilation

- How turn notation programmers prefer into notation computer understands?
- Program to translate C statements into Assembly Language instructions; called a compiler
- Example: compile by hand this C code:

```
a = b + c;
d = a - e;
```

- Easy:

```
add a, b, c
sub d, a, e
```

- Big Idea: compiler translates notation from 1 level of abstraction to lower level

Compilation 2

- Example: compile by hand this C code:

```
f = (g + h) - (i + j);
```

- First sum of `g` and `h`. Where put result?

```
add f,g,h    # f contains g+h
```

- Now sum of `i` and `j`. Where put result?
  - Cannot use `f` !
  - Compiler creates temporary variable to hold sum: `t1`

```
add t1,i,j   # t1 contains i+j
```

- Finally produce difference

```
sub f,f,t1   # f=(g+h)-(i+j)
```

Compilation 2 Summary

- C statement (5 operands, 3 operators):

```
f = (g + h) - (i + j);
```

- Becomes 3 assembly instructions (6 unique operands, 3 operators):

```
add f,g,h    # f contains g+h
add t1,i,j   # t1 contains i+j
sub f,f,t1   # f=(g+h)-(i+j)
```

- <u>In general, each line of C produces many assembly instructions</u>
  - One reason why people program in C vs. Assembly; fewer lines of code
  - Other reasons?

Assembly Operands

- Unlike HLL, assembly instructions cannot use variables
  - Why not? Keep Hardware Simple
- Instruction operands are <u>registers</u>: limited number of special locations; 32 registers in MIPS
  - Why 32? Smaller is faster
- Each MIPS register is 32 bits wide
  - Groups of 32 bits called a <u>word</u> in MIPS

Assembly Operands: Registers

- Naming of 32 registers: instead of r0, r1, …, r31, use
  - `$s0, $s1,` … for registers corresponding to C variables
  - `$t0, $t1,` … for registers corresponding to temporary variables
  - Will explain mapping convention later of `$s0, $s1,` … , `$t0, $t1,` … , to r0, r1, …
- Note: whereas C declares its operands, Assembly operands (registers) are fixed and not declared

Compilation using Registers

- Compile by hand using register names:

```
f = (g + h) - (i + j);
```

  - `f` maps onto `$s0`
    `g` maps onto `$s1`
    `h` maps onto `$s2`
    `i` maps onto `$s3`
    `j` maps onto `$s4`
- MIPS Instructions: Try this yourself during Adminstrivia

Administrivia

- Grading: fixed scale, not on a curve
- Class voted cheating penalty:
  - 0 for problem on lab, assignment
  - 0 for project, exam
  - Per university rules, contact Office of Student Conduct to report cheating
- To try to switch sections go to old and new sections to ask for TAs' OK
- Viewing lectures again: 100 Wheeler T/T 5-6 (old tapes in 205 McLaughlin)
- Readings: Ch.3 COD 3.1- 3.3,3.5,3.8

Administrivia

- Read web page: Intro, FAQ, Schedule
  `www-inst.eecs/~cs61c`
  Midterm 3/17 5-8 PM, Final 5/12 5-8 PM
- 1st homework: COD Exercises 1.1-1.16, 1.18, 1.21-1.23, 1.25, 1.27-1.30, 1.34-1.41,1.43-1.44, 1.56; Due Wed. 1/27 7PM (do by yourself); readings all but 1.4,1.5
- 1st project: C spelling checker `philspel` Due Wed. 2/3 7PM (do by yourself)
  `www-inst.EECS/~cs61c/handouts/proj1.pdf`
- 1st lab: Read SPIM (A-39 to A-48); see newsgroup for what to do before lab

"Computers in the News" (see slide #2)

- "Browsers and Borders Are Argued at the Microsoft Trial", N.Y. Times 1/20/99
  - "... the Government spent most of today attacking the company's central defense-- that its Internet browsing software is a seamless feature of its Windows operating system and not a stand-alone product. The Government is trying to prove that the browser and operating system are two products, illegally bundled together mainly to stifle competition."
  - "... even Michael Dertouzos[MIT], ... who was initially on Microsoft's witness list but was later dropped -- had testified in a deposition that he considered the browser an application program, separate from the operating system."

Compilation using Registers

- Compile by hand using registers:

```
f = (g + h) - (i + j);
```

  - `f: $s0, g: $s1, h: $s2, i: $s3, j: $s4`

- MIPS Instructions:

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

Assembly Operands: Memory

- C variables map onto registers; what about data structures like arrays?
- 1 of 5 components of a computer: memory contains such data structures
- But arithmetic instructions only operate on registers?
- Data transfer instructions transfer data between registers and memory
  - Think of memory as a large single dimensioned array, starting at 0

Data Transfer Instruction: Memory to Reg

- Load: moves instruction from memory to register
  - Syntax:

  1) operation name

  2) register to be loaded

  3) constant and register to access memory
- MIPS name, `lw` for load word:
  - Example: `lw $t0, 8($s3)`

Called "offset" Called "base register"

Compilation when Operand is in Memory

- Compile by hand using registers:
  `g = h + A[8];`
  - `g: $s1`, `h: $s2`, `$s3`: starting address (base address) of array `A`
- Since A[8] is in memory, 1st transfer from memory to (temporary) register:

```
lw $t0,8($s3)   # $t0 gets A[8]
```

  - Adds 8 to `$s3` to select `A[8]`, puts into `$t0`
- Next add it to h and place in g

```
add $s1,$s2,$t0    # $s1 = h+A[8]
```


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, hence word addresses differ by 4
  - `Memory[0]`, `Memory[4]`, `Memory[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.**
  - **Another advantage of compilers: they use proper byte address for words**

Compilation with Memory Revisited

- **What offset in `lw` to select `A[8]` in C?**
- **4x8=32 to select `A[8]`: byte v. word**
- **Compile by hand using registers:**
  `g = h + A[8];`
  - **`g`: `$s1`, `h`: `$s2`, `$s3`:base address of `A`**
- **1st transfer from memory to register:**

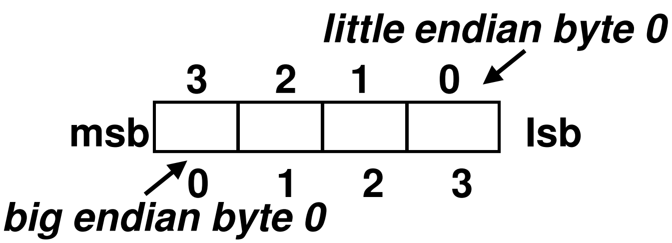
```
lw $t0,32($s3)  # $t0 gets A[8]
```

  - **Add <u>32</u> to `$s3` to select `A[8]`, put into `$t0`**
- **Next add it to h and place in g**

```
add $s1,$s2,$t0    # $s1 = h+A[8]
```


More Notes about Memory

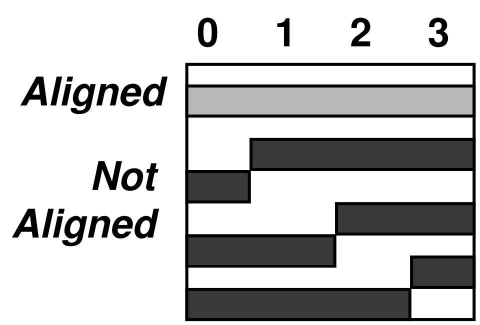
- **How are bytes numbered in a word?**



- **Gulliver's Travels: Which end of egg to open?**
  Cohen, D. "On holy wars and a plea for peace (data transmission)." *Computer*, vol.14, (no.10), Oct. 1981. p.48-54.
  Little Endian address of least significant byte: Intel 80x86, DEC Alpha
  Big Endian address of most significant byte HP PA, IBM/Motorola PowerPC, SGI, Sparc
- **Russian Czar to settle arbitrary decision?**

More Notes about Memory: Alignment

- **MIPS requires that all words start at addresses that are multiples of 4**



- **Called <u>Alignment</u>: objects must fall on address that is multiple of their size.**

(Later we'll see how alignment helps performance)

Data transfer: Register to Memory

- Store: complementary instruction to load
- MIPS: `sw` for Store Word, same syntax as `lw`
- Example: `A[12] = h;`
  - base address of `A: $s3, h: $s2`
- MIPS assembly instructions:

```
sw $s2,48($s3) # h into A[12]
```


Compile with variable index

- What if array index not a constant?
  `g = h + A[i];`
  - `g: $s1, h: $s2, i: $s4, $s3`:base address of `A`
- To load `A[i]` into a register, first turn `i` into a byte address; multiply by 4
- How multiply using adds?
  - i + i = 2i, 2i + 2i = 4i

```
add $t1,$s4,$s4    # $t1 = 2*i
add $t1,$t1,$t1    # $t1 = 4*i
```


Compile with variable index, con't

- Next add to base of A:

```
add $t1,$t1,$s3 #$t1=address of
                #A[i] (4*i+$s3)
```

- Now load `A[i]` into a temporary register:

```
lw $t0,0($t1) # Temp $t0 = A[i]
```

- Finally add to `h` and put sum in `g`:

```
add $s1,$s2,$t0    # g = h + A[i]
```


Compile with variable index: Summary

- C statement:

```
g = h + A[i];
```

- Compiled MIPS assembly instructions:

```
add $t1,$s4,$s4    # $t1 = 2*i
add $t1,$t1,$t1    # $t1 = 4*i
add $t1,$t1,$s3    #$t1=addr A[i]
lw $t0,0($t1)      # $t0 = A[i]
add $s1,$s2,$t0    # g = h + A[i]
```

Role of Registers vs. Memory

- What if more variables than registers?
  - Compiler trys 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 instruction 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

(If time allows) Do it yourself:

- Compile this MIPS code:

```
B[i] = h + A[i];
```

  - `h`: `$s2`, `i`: `$s4`, `$s3`:base address of `A`, `$s5`:base address of `B`

And in Conclusion (2 slides) ...

- Big idea in CS&E; compilation to translate from one level of abstraction to lower level
  - Generally single HLL statement produces many assembly instructions
  - Also hides address calculations (byte vs. word addressing)
- Design of an Assembly Language like MIPS shaped by

  1) Desire to keep hardware simple: e.g., each operation has 3 operands

  2) Smaller is faster: e.g., MIPS has 32 registers

And in Conclusion (last slide) ...

- MIPS assembly language thus far:
  - Instructions: `add, sub, lw, sw`
  - At most one assembly instruction per line
  - Comments start with # to end of line
  - Operands: registers `$s0, $s1, ... ; $t0, $t1, ...`
  - Operands: memory `Memory[0], Memory[4], Memory[8], ,... , Memory[4294967292]`
- Next: difference between computers and calculators: Making decisions