CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/fa10

Agenda
• Review
• Real Numbers and Instructions as Numbers
• Assembly Language to Machine Language
• Administrivia
• Technology Break
• More on C and Pointers
• Summary

Review from Last Lecture
• Integer and floating point operations can lead to results too big to store within their representations: overflow/underflow
• Floating point is an approximation of reals
• Everything is a (binary) number in a computer
  – Instructions and data; stored program concept
• MIPS ISA guided by 4 design principles:
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

Goals for Floating Point
• Standard arithmetic for reals for all computers
  – Like two’s complement
• Keep as much precision as possible in formats
• Help programmer with errors in real arithmetic
  – ↔, →, Not-A-Number (NaN), exponent overflow, exponent underflow
• Keep encoding that is somewhat compatible with two’s complement
  – E.g., 0 in Fl. Pt. is 0 in two’s complement
  – Make it possible to sort without needing doing floating point comparison

More Floating Point
• Zero: Bit pattern all 0s means 0.000
  ⇒ But 0 in exponent should mean most negative exponent (want 0 to be next to smallest real)
  ⇒ Can’t use two’s complement (1000 0000<sub>two</sub>)
• Bias notation: subtract bias from exponent
  – Single precision uses bias of 127; DP uses 1023
• 0 uses 0000 0000<sub>two</sub> ⇒ 0-127 = -127;
  ∞, NaN uses 1111 1111<sub>two</sub> ⇒ 255-127 = +128
  – Smallest SP real can represent: 1.00...00 x 2<sup>-126</sup>
  – Largest SP real can represent: 1.11...11 x 2<sup>127</sup>

MIPS Floating Point Instructions
• C, Java has single precision (float) and double precision (double) types
• MIPS instructions: s for single, d for double
  – Fl. Pt. Addition single precision: add.s
  – Fl. Pt. Addition double precision: add.d
  – Fl. Pt. Subtraction single precision: sub.s
  – Fl. Pt. Subtraction double precision: sub.d
  – Fl. Pt. Multiplication single precision: mul.s
  – Fl. Pt. Divide single precision: div.s
  – Fl. Pt. Divide double precision: div.d
MIPS Floating Point Instructions

- C, Java has single precision (float) and double precision (double) types
- MIPS instructions: .s for single, .d for double
  - FI. Pt. Comparison single precision:
    - FI. Pt. branch:
  - Since rarely mix integers and FI. Pt., MIPS has separate registers for floating-point operations: $f0, f1, ..., f31$
  - Double precision uses adjacent even-odd pairs of registers:
    - $f0$ and $f2$, $f3$ and $f5$, $f4$ and $f6$, ..., $f30$ and $f31$
- Need data transfer instructions for these new registers
  - lw1 (load word), sw1 (store word)
- Double precision uses two lw1 instructions, two sw1 instructions

Converting C to MIPS Machine code

```c
&A=*$t0 (reg 8), $t1 (reg 9), h=$s2 (reg 18)
A[300] = h + A[300];
```

Encoding of MIPS Instructions: Must Be Unique!

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>0</td>
<td>33sbus</td>
</tr>
<tr>
<td>subu R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>35sbus</td>
<td>n.n...</td>
</tr>
<tr>
<td>sll R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>reg constant</td>
<td>0</td>
<td>n.n...</td>
</tr>
<tr>
<td>add unsigned i</td>
<td>9sbus</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>reg constant</td>
<td>constant</td>
<td>address</td>
</tr>
<tr>
<td>lw (load word)</td>
<td>35sbus</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>n.n...</td>
<td>n.n...</td>
<td>address</td>
</tr>
<tr>
<td>sw (store word)</td>
<td>43sbus</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>n.n...</td>
<td>n.n...</td>
<td>address</td>
</tr>
<tr>
<td>beq</td>
<td>4sbus</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>n.n...</td>
<td>n.n...</td>
<td>address</td>
</tr>
<tr>
<td>bne</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>n.n...</td>
<td>n.n...</td>
<td>address</td>
</tr>
<tr>
<td>j (jump)</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>2sbus</td>
<td>n.n...</td>
<td>address</td>
</tr>
<tr>
<td>jal</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>3sbus</td>
<td>n.n...</td>
<td>address</td>
</tr>
<tr>
<td>jr (jump reg)</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>8sbus</td>
</tr>
</tbody>
</table>

Addressing in Branches

- Programs much bigger than 2^16 bytes, but branch address must fit in 16-bit field
  - Must specify a register for branch addresses for big programs: PC = Register + Branch address
  - Which register?
- Conditional branching for IF-statement, loops
  - Tend to be near branches; ½ within 16 instructions
- Idea: PC-relative branching

Addressing in Jumps

- Same trick for Jumps, Jump and Link
  - PC = Jump address * 4
- Since PC = 32 bits, and Jump address * 4 = 28 bits, what about other 4 bits?
  - Jump and Jump and Link only changes bottom 28 bits of PC
Assembly and Pseudo-instructions

- Turning textual MIPS instructions into machine code called **assembly**, program called **assembler**
  - Calculates addresses, maps register names to numbers, produces binary machine language
  - Textual language called assembly language
- Can also accept instructions convenient for programmer but not in hardware
  - **Load immediate (li)** allows 32-bit constants, assembler turns into lui or ori (if needed)
  - **Load double (fd)** uses two lw/h instructions to load a pair of 32-bit floating point registers
- Called **Pseudo-Instructions**

### Agenda

- Review
- Real Numbers and Instructions as Numbers
- Administrivia
- Technology Break
- More on C and Pointers
- Summary

### Pointers in C

- A pointer is just another kind of value
  - A basic type in C

```
int *ptr;
```

The variable “ptr” is a pointer to an “int.”

### Pointers in C

- If \( T \) is a type, \( T *p \) declares \( p \) a pointer to that type
- You can use \( p \) as a pointer to a \( T \)
- You can use \( *p \) as a \( T \)
- \( p++ \) increments \( p \) by the size of a \( T \)
  - Important because of the way arrays are treated
- You can make a pointer to any variable
  - If \( x \) is any variable, then \( &x \) is its address

### 32 bit constants in MIPS

- Can create a 32-bit constant from two 32-bit MIPS instructions
- **Load Upper Immediate (lui or “Louie”)** puts 16 bits into upper 16 bits of destination register
- MIPS to load 32-bit constant into register $s0?
  
  \[
  \begin{array}{cccc}
  \text{lui} & \text{addr} & \text{op} & \text{constant} \\
  \text{ori} & \text{addr} & \text{op} & \text{constant}
  \end{array}
  \]

<table>
<thead>
<tr>
<th>Address</th>
<th>800</th>
<th>804</th>
<th>808</th>
<th>812</th>
<th>816</th>
<th>820</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{lui} ) $s0$, 61 # 61 ( = 0000 \ 0000 \ 0011 \ 1101 )</td>
<td>( \text{ori} ) $s0$, $s0$, 2304 # 2304 ( = 0000 \ 1001 \ 0000 \ 0000 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- MIPS to load 32-bit constant into register $s0?

---

Many of Slides 18 to 32 come from “Arrays and Pointers in C” by Alan Cox and T S. Ng, Rice University,

www.cs.berkeley.edu/~alanmi/compsci216/notes/03 Arrays Pointers.ppt

---

Converting to MIPS Machine code

### Loop:

- `sll $t1, $s3, 2`
- `addu $t1, $t1, $s6`
- `lw $t0, 0($t1)`
- `bne $t0, $s5, Exit`
- `addiu $s3, $s3, 1`
- `j Loop`

### Exit:

### Agenda

- Review
- Real Numbers and Instructions as Numbers
- Administrivia
- Technology Break
- More on C and Pointers
- Summary
Arrays in C

- Array indexing is syntactic sugar for pointers
- \( a[i] \) is treated as \(*\( a+i \)\)
- To zero out an array:
  - for \((i=0; i < \text{size}; i++)\) \( a[i] = 0; \)
  - for \((i=0; i < \text{size}; i++)\) \(*\( a+i \)\) = 0;
  - for \((p=a; p < a+\text{size}; p++)\) \(*p = 0; \)

Pointer Operations in C

- Creation
  \& variable Returns variable's memory address
- Dereference
  \* pointer Returns contents stored at address
- Indirect assignment
  \* pointer = val Stores value at address
- Of course, still have...Assignment
  pointer = ptr Stores pointer in another variable

Using Pointers

```c
int i1;
int i2;
int *ptr1;
int *ptr2;
i1 = 1;
i2 = 2;
ptr1 = &i1;
ptr2 = ptr1;
*ptr1 = 3;
i2 = *ptr2;
```

Using Pointers (cont.)

```c
int int1 = 1036; /* some data to point to */
int int2 = 8;
int *int_ptr1 = &int1; /* get addresses of data */
int *int_ptr2 = &int2;
*int_ptr1 = int_ptr2;
*int_ptr1 = int2;
```

What happens?

Type check warning: \*int_ptr2 is not an int
int1 becomes 8

Using Pointers (cont.)

```c
int int1 = 1036; /* some data to point to */
int int2 = 8;
int *int_ptr1 = &int1; /* get addresses of data */
int *int_ptr2 = &int2;
int_ptr1 = int_ptr2;
int_ptr1 = int_ptr2;
```

What happens?

Type check warning: \*int_ptr2 is not an int

Changes int_ptr1 - doesn't change int1

Pointer Arithmetic

\( \text{pointer} + \text{number} \quad \text{pointer} - \text{number} \)

E.g., \( \text{pointer} + 1 \) adds 1 something to a pointer

```c
char *p;
char a;
char b;
p = a;
p += 1; /* In each, p now points to b (Assuming compiler doesn't reorder variables in memory) */
```

Adds 1*sizeof(char) to the memory address

Pointer arithmetic should be used cautiously
Is Pass by Reference Really by Reference?

- In C, the default passing strategy is pass by copy
- To pass by reference, we use pass by copy – because in C, everything is pass by copy
- So, the value that we have to pass by copy is the address of the actual argument, which we achieve using the address operator &
- In other words, in C pass by reference is actually pass by copy – because you copy the address
**Summary**

- Everything is a (binary) number in a computer
  - Instructions and data; stored program concept
- Assemblers can enhance machine instruction set to help assembly-language programmer
- Unlike Java pointers, C pointers you must know
  - when to use a pointer
  - when to dereference the pointer
  - when to pass an address to a variable rather than the variable itself
  - when to use pointer arithmetic to change the pointer
  - how to use pointers without making your programs unreadable