CS 61C: Great Ideas in Computer Architecture

(Brief) Review Lecture

Instructor: Justin Hsia
Topic List So Far (1/2)

• Number Representation
  – Signed/unsigned, Floating Point
  – Powers of 2 (IEC)

• C Topics
  – Pointers, Arrays, Strings, Structs
  – Bitwise operators and bit masking

• Memory Management
  – Memory Layout
  – Memory Allocation Strategies
Topic List So Far (2/2)

- MIPS Topics
  - Instruction Formats
  - Labels, Branches, and Jumps
  - Relative vs. Absolute Addressing
  - Function/Register Conventions
- C.A.L.L.
- Performance
- Caches
Number Representation (1/5)

- Anything can be represented as a number!
  - Different interpretations of the same numeral
  - $n$ digits in base $B$ can represent at most $B^n$ things
- **Bases:** binary (2), decimal (10), hex (16)
- **Bit sizes:** nibble (4), byte (8), word (32)
- **Overflow:** result of operation can’t be properly represented
  - Signed vs. Unsigned
Number Representation (2/5)

• **Integers**
  – Unsigned vs. Signed: sign & magnitude, 1’s complement, 2’s complement, biased
  – Negation procedures vs. representation names
  – Sign Extension vs. Zero Extension

• **Characters**
  – Smallest unit of data (1 byte)
  – ASCII standard maps characters to small numbers (e.g. ‘0’ = 48, ‘a’ = 97)
Number Representation (3/5)

• **Floating Point**
  
  – Binary point encoded in scientific notation
  
  $(-1)^S \times (1 \cdot \text{Mantissa}) \times 2^{(\text{Exponent} - 127)}$
  
  – Exponent uses *biased notation* (bias of $2^7 - 1 = 127$)
  
  – Can only save 23 bits past binary point in Mantissa (rest gets rounded off)
  
  – Double precision uses $|1|11|52|$ split
Number Representation (4/5)

• **Floating Point Special Cases:**

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>± 0</td>
</tr>
<tr>
<td>0</td>
<td>non-zero</td>
<td>± Denorm Num</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>± Norm Num</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>± ∞</td>
</tr>
<tr>
<td>255</td>
<td>non-zero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

• What numbers can we can represent?
  – Watch out for exponent *overflow* and *underflow*
  – Watch out for *rounding* (and loss of associativity)
Number Representation (5/5)

- **Powers of 2 (IEC Prefixes)**
  
  - Convert $2^{XY}$:

<table>
<thead>
<tr>
<th>Y (Digit)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X (Digit)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>--</td>
</tr>
<tr>
<td>1</td>
<td>Kibi (Ki)</td>
</tr>
<tr>
<td>2</td>
<td>Mebi (Mi)</td>
</tr>
<tr>
<td>3</td>
<td>Gibi (Gi)</td>
</tr>
<tr>
<td>4</td>
<td>Tebi (Ti)</td>
</tr>
<tr>
<td>5</td>
<td>Pebi (Pi)</td>
</tr>
<tr>
<td>6</td>
<td>Exbi (Ei)</td>
</tr>
<tr>
<td>7</td>
<td>Zebi (Zi)</td>
</tr>
<tr>
<td>8</td>
<td>Yobi (Yi)</td>
</tr>
</tbody>
</table>

+ Things
• **Pointer**: data type that holds an *address*
  - Visually can draw an arrow to another variable
  - `NULL (0x00000000)` means pointer to nothing
  - `& (address of)` and `* (dereference)` operators
  - *Pointer arithmetic* moves correct number of bytes for data type (e.g. 1 for *char*, 4 for *int*)
  - Trying to access invalid, un-owned, or unaligned addresses causes errors
  - Use pointers to *pass by reference* (C functions naturally *pass by value*)
• **Array**: sequential collection of objects of the same data type
  - Must be initialized with a size, cannot be changed
    
    ```
    type array[SIZE];  
    ```
    
    ```
    type array[] = \{d_1, ..., d_n\};  
    ```
    
  - Bounds checking done manually (pass size)
  - Access array: \( \text{array}[i] \leftrightarrow *(\text{array} + i) \)
C Topics (3/4)

- **Strings**
  - Array of characters; null terminated (`\0` = 0)
  - Don’t need to pass length because can look for null terminator (`strlen` does not count it)
  - For $n$ characters, need space for $n+1$ bytes ($sizeof(char)=1$)
**C Topics (4/4)**

- **Structs**
  - Collection of variables (stored together in mem)
  - Definition: `struct name { ... fields ... };`
  - Variable type is `struct name` (2 words)
  - Access field (.)
  - With pointers: `x->field ↔ (*x).field`

- **Typedef**
  - Rename an existing variable type
  - `typedef name_orig name_new;`
Memory Management (1/4)

- Program’s *address space* contains 4 regions:
  - **Stack**: local variables, grows downward
  - **Heap**: space requested for pointers via *malloc()*; resizes dynamically, grows upward
  - **Static Data**: global and static variables, does not grow or shrink
  - **Code**: loaded when program starts, does not change size

```
+---------------------+    +-------------------+    +-------------------+
| stack              |    | heap              |    | static data       |
|                    +    +-------------------+    +-------------------+
|                    | ~FFFF FFFF_{hex} |    |                    |
|                    +    +-------------------+    +-------------------+
|                    |    | code               |    |                    |
|                    |    +-------------------+    +-------------------+
|                    | ~0_{hex}            |    |                    |
```

Memory Management (2/4)

• **Stack** grows in *frames* (1 per function)
  – Hold local variables (these disappear)
  – Bottom of Stack tracked by $sp$
  – LIFO action (pop/push)

• Stack holds what we can’t fit in registers
  – *Spilling* if more variables than registers
  – Large local variables (arrays, structs)
  – Old values we want to save (saved or volatile registers)
Memory Management (3/4)

• **Heap managed with** `malloc` and `free`
  – Pointers to chunks of memory of requested size (in bytes); use `sizeof` operator
  – Check pointer; **NULL** if allocation failed
  – Don’t lose original address (need for `free`)
  – With structs, free allocated fields **before** freeing struct itself
  – Avoid memory leaks!
Memory Management (4/4)

• Memory management
  – Want fast with minimal memory overhead
  – Avoid *fragmentation*

• Basic **allocation strategies**
  – *Best-fit*: Choose smallest block that fits request
  – *First-fit*: Choose first block that is large enough (always starts from beginning)
  – *Next-fit*: Like first-fit, but resume search from where we last left off
MIPS Topics (1/6)

• MIPS is a **RISC** ISA
  – “Smaller is faster” and “Keep is simple”
  – Goal is to produce faster hardware
  – *Pseudo-instructions* help programmer, but not actually part of ISA (MAL vs. TAL)

• 32 32-bit registers are extremely fast
  – Only operands of instructions
  – Need `lw/sw/lb/sb` for memory access

• Learn how to use your Green Card!
  – It has soooooooooo much info on it
MIPS Topics (2/6)

• **Stored Program Concept**
  – Instructions are data, too!

• **Memory is *byte-addressed***
  – Most data (including instructions) in words and *word-aligned*, so all word addresses are multiples of 4 (end in 0b00)
  – *Endianness*:

```
lsb 0 1 2 3
  ^
```

```
msb 0 1 2 3
  ^
```

---

little endian

big endian
MIPS Topics (3/6)

- **Instruction Formats**

<table>
<thead>
<tr>
<th>R:</th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I:</td>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>J:</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td>target address</td>
<td></td>
</tr>
</tbody>
</table>

- Determine format/instr using `opcode/funct` (6)
- Register fields (5): `rs/rt/rd`
- Shift amount (5): `shamt`
- Constants/addresses: `immediate` (16), `target address` (26)
MIPS Topics (4/6)

• **Relative addressing:**
  - Branch instructions relative to PC
  - $PC = (PC+4) + (\text{immediate} \times 4)$
  - Can count *by instruction* for immediate
  - Max forward: $2^{15}$ instr = $2^{17}$ bytes
  - Max backwards: $-2^{15}+1$ instr = $-2^{17}+4$ bytes
  - Do not need to be relocated

signed

Relative to current instruction
MIPS Topics (5/6)

• **Pseudo-absolute addressing:**
  - Jump instructions try to specify exact address
  - `j/jal`: \( PC = \{ (PC+4)[31..28], \) target address, 00 \}
  - `jr`: \( PC = R[rs] \)
  - Target address field is desired byte address/4
  - `j/jal` can specify \( 2^{26} \) instr = \( 2^{28} \) bytes
  - `jr` can specify \( 2^{32} \) bytes = \( 2^{30} \) instr
  - Always need to be relocated
MIPS Topics (6/6)

- **MIPS functions**
  - `jal` invokes function, `jr $ra` returns
  - `$a0-$a3` for args, `$v0-$v1` for return vals

- **Saved registers:** `$s0-$s7`, `$sp`, `$ra`
  - CalleR saves volatile registers it is using before making a procedure call

- **Volatile registers:** `$t0-$t9`, `$v0-$v1`, `$a0-$a3`
  - CalleE saves saved registers it intends to use

This is the confusing one
C.A.L.L.

• **Compiler** converts a single HLL file into a single assembly file \(.c \rightarrow .s\)

• **Assembler** removes pseudo-instructions, converts what it can to machine language, and creates symbol and relocation tables \(.s \rightarrow .o\)
  – Resolves addresses by making 2 passes (for internal forward references)

• **Linker** combines several object files and resolves absolute addresses \(.o \rightarrow .out\)
  – Enable separate compilation and use of libraries

• **Loader** loads executable into memory and begins execution
Performance

• Performance measured in *latency* or *bandwidth*

\[
\text{Perf}_X = \frac{1}{\text{Program Execution Time}_X}
\]

• Latency measurement:
  – CPU Time = Instructions × CPI × Clock Cycle Time
  – Affected by different components of the computer
Caches (1/4)

• Why cache?
  – Take advantage of temporal/spatial locality to improve memory performance

• Cache Terminology
  – Block: unit of data transfer between $ and Mem
  – Slot: place to hold block of data in $
  – Set: set of slots an address can map into
  – Hit, Miss, Replacement
  – Hit Rate, Miss Rate, Hit:Miss Ratio
Caches (2/4)

• Request is an address:
  – Search by Index, check Valid & Tag(s) for match

• Cache Parameters
  – Address space $2^A$ bytes $\leftrightarrow A$ address bits
  – Block size $K$ bytes $\leftrightarrow O = \log_2(K)$ offset bits
  – Associativity $N$ $\leftrightarrow N$ slots/set
  – Cache size $C$ bytes $\leftrightarrow C/K/N$ sets
    $\leftrightarrow I = \log_2(C/K/N)$ index bits
  – $2^T$ blocks map to set $\leftrightarrow T = A - I - O$
Caches (3/4)

• Policies
  – Write hit: write-back / write-through
  – Write miss: write allocate / no-write allocate
  – Replacement: random / LRU

• Implementation
  – Valid bit
  – Dirty bit (if write-back)
  – Tag (identifier)
  – Block data (8*K bits)
  – LRU management bits per set
  – per slot
Caches (4/4)

• 3 C’s of Cache Misses
  – Compulsory, Capacity, Conflict

• Performance
  – \( \text{AMAT} = \text{HT} + \text{MR} \times \text{MP} \)
  – \( \text{AMAT} = \text{HT}_1 + \text{MR}_1 \times (\text{HT}_2 + \text{MR}_2 \times \text{MP}_2) \)
  – \( \text{CPI}_{\text{stall}} = \text{CPI}_{\text{base}} + \frac{\text{Accesses}}{\text{Instr}} \times \text{MR} \times \text{MP} \)
  – \( \text{CPI}_{\text{stall}} = \text{CPI}_{\text{base}} + \frac{\text{Accesses}}{\text{Instr}} \left( \text{MR}_1 \times \text{MP}_1 + \text{MR}_1 \times \text{MR}_2 \times \text{MP}_2 \right) \)
  • Extra terms if L1$ split among I$ and D$
  – \( \text{MR}_{\text{global}} = \text{product of all MR}_i \)
Get To Know Your Staff

• Category: Food