CS 61C: Great Ideas in Computer Architecture

(Brief) Review Lecture

Instructor: Alan Christopher

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:** nybble (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

31	30 … 23	22 … 0
S	**Exponent** (8)	**Mantissa** (23)

$$(-1)^{S} \times (1\,.\,\textbf{Mantissa}) \times 2^{(\textbf{Exponent}-127)}$$

  - 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:**

Exponent	Significand	Meaning
0	0	± 0
0	non-zero	± Denorm Num
1-254	anything	± Norm Num
255	0	± ∞
255	non-zero	NaN

Exponent of 2^{-126}

- 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}:

Y (Digit)	Value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512

+

X (Digit)	Value
0	--
1	Kibi (Ki)
2	Mebi (Mi)
3	Gibi (Gi)
4	Tebi (Ti)
5	Pebi (Pi)
6	Exbi (Ei)
7	Zebi (Zi)
8	Yobi (Yi)

Things

C Topics (1/4)

- **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*)

C Topics (2/4)

- **Array:** sequential collection of objects of the same data type
  - Must be initialized with a size, cannot be changed

`type array[SIZE];` or

Size implicit in initialization

`type array[] = {d1,…,dn};`

  - Bounds checking done manually (pass size)
  - Access array: `array[i]` ↔ `*(array + i)`

C Topics (3/4)

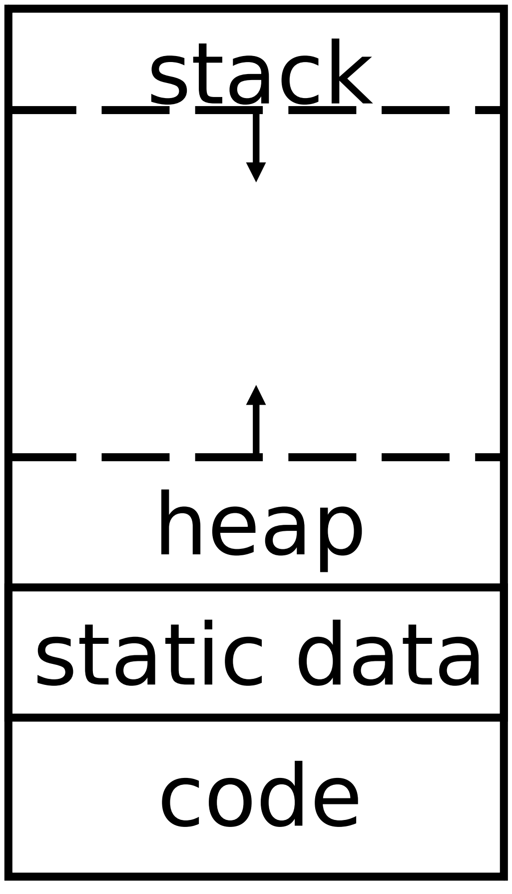
- **Strings**
  - Array of characters; null terminated (`'\0'=0`)
  - Don't need to pass length because can look for null terminator (`strlen` does not count it)
  - For *n* characters, need space for *n*+1 bytes (`sizeof(char)=1`)

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 nameorig namenew;`

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



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 <u>before</u> freeing struct itself
  - Avoid memory leaks!

Memory Management (4/4)

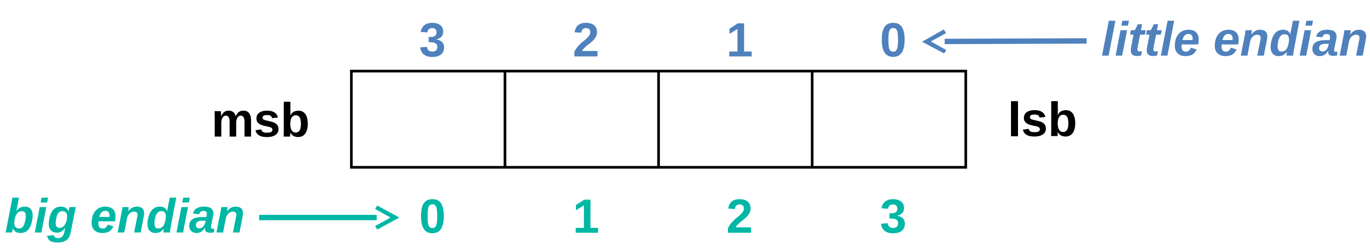
- Memory management
  - Want fast with minimal memory overhead
  - Avoid *fragmentation*
- Basic **allocation strategies**
  - *Best-fit:* Choose smallest block that fits request
  - *First-fit***:** Choose first block that is large enough (always starts from beginning)
  - *Next-fit***:** Like first-fit, 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
- Remember 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:*



MIPS Topics (3/6)

- **Instruction Formats**

R:	`opcode`	`rs`	`rt`	`rd`	`shamt`	`funct`
I:	`opcode`	`rs`	`rt`	`immediate`		
J:	`opcode`	`target address`				

  - 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) + (immediate*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`, `$ra`
  Volatile registers: `$t0-$t9`, `$v0-$v1`, `$a0-$a3`
  - CalleR saves volatile registers it is using before making a procedure call
  - CalleE saves saved registers it intends to use

C.A.L.L.

- **Compiler** converts a single HLL file into a single assembly file `.c --> .s`
- **Assembler** removes pseudo-instructions, converts what it can to machine language, and creates symbol and relocation tables `.s --> .o`
  - Resolves addresses by making 2 passes (for internal forward references)
- **Linker** combines several object files and resolves absolute addresses `.o --> .out`
  - Enable separate compilation and use of libraries
- **Loader** loads executable into memory and begins execution

Performance

- Performance measured in *latency* or *bandwidth*

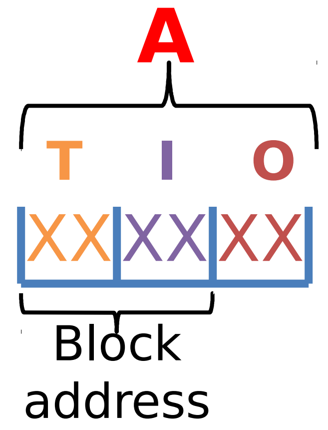
$$\mathrm{Perf}_X = \frac{1}{\text{Program Execution Time}_X}$$

- Latency measurement:
  - $\text{CPU Time} = \text{Instructions} \times \text{CPI} \times \text{Clock Cycle Time}$
  - 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 $2^{\mathbf{A}}$ bytes ↔ **A** address bits
  - Block size **K** bytes ↔ **O** = $\log_2$(**K**) offset bits
  - Associativity **N** ↔ N slots/set
  - Cache size **C** bytes ↔ **C**/**K**/**N** sets ↔ **I** = $\log_2$(**C**/**K**/**N**) index bits
  - $2^{\mathbf{T}}$ blocks map to set ↔ **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 *slot*

per *set*

Caches (4/4)

- 3 C's of Cache Misses
  - Compulsory, Capacity, Conflict
- Performance
  - $\text{AMAT} = \text{HT} + \text{MR} \times \text{MP}$
  - $\text{AMAT} = \text{HT}_1 + \text{MR}_1 \times (\text{HT}_2 + \text{MR}_2 \times \text{MP}_2)$
  - $\text{CPI}_{\text{stall}} = \text{CPI}_{\text{base}} + \frac{\text{Accesses}}{\text{Instr}} \times \text{MR} \times \text{MP}$
  - $\text{CPI}_{\text{stall}} = \text{CPI}_{\text{base}} + \frac{\text{Accesses}}{\text{Instr}} (\text{MR}_1 \times \text{HT}_2 + \text{MR}_1 \times \text{MR}_2 \times \text{MP}_2)$
    - Extra terms if L1\$ split among I\$ and D\$
  - $\text{MR}_{\text{global}}$ = product of all MR_i

Technology Break

Other Questions?