CS33

Bits/Bytes

Intel is *little endian*, so the least significant bytes come first (at the lowest address)
1 byte = 8 bits

$(2^{20} = 1MB,\ 2^{30} = 1GB,\ 2^{31} = 4GB)$

Left Shift: logical/arithmetic same
Right Shift: logical: pad with 0's. Arithmetic: pad with most significant bit
XOR Swap:

```
X := X XOR Y
Y := X XOR Y
X := X XOR Y
```

Integers

Unsigned

Represented like a normal 4-byte number
In operations, Unsigned casting will always take precedence
CASTING = JUST INTERPRETING THE BITS DIFFERENTLY

Two's complement

Most significant digit is 1
This makes the negative range 1 larger than the positive range

```
~x+1=-x
```

Floating Point

$(-1)^s M 2^E$

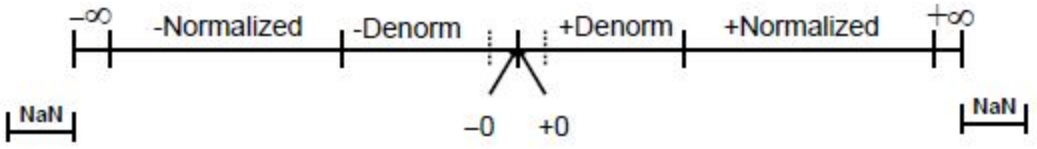
Bits in the mantissa are $\frac{1}{2}, \frac{1}{4}, \frac{1}{2^{-n}}$

Only exact representation of $x/2^k$

Bias is $2^{e-1} - 1$ (e is the number of exponent bits)

Values:

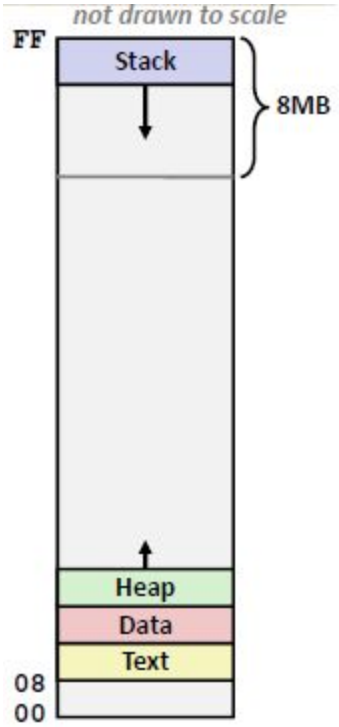
- Exp NOT 000...000 and NOT 111...111 : NORMALIZED
    - **E = exp - bias**
    - Has implied leading 1 to the frac
        - **M = 1.xxxx** (where xxxx is frac)
- Exp 000...000 and Frac NOT 000...000: DENORMALIZED
    - **E = -Bias + 1** (exponent value)
    - No implied leading 1 onto frac
        - **M = 0.xxxx** (wher xxxx is frac)
- Exp 000...000 and Frac 000...000 : ZERO
    - can be positive or negative zero
- Exp 111....111 and Frac 000...000 : INFINITY
    - Positive or negative infinity (for overflow/underflow)
- Exp 111...111 and Frac NOT 000...000 : NaN
    - Not a Number
    - Always returns false in any comparison



Sizes:

	Sign	**Exponent**	**Fraction**	**Bias**
Float (4bytes)	1	8	23	127
Double (8bytes)	1	11	52	1023

Assembly



Stack grows downward!!
Important Instructions (size-dependent! see below):

- push [A] pushes [A] onto the stack and decrements %esp by 4
- pop [A] pops the topmost thing off the stack into [A]. Increments %esp by 4
- mov [A][B] copies [A] and puts it into [B]
- leal[A][B] [A] uses an address mode expression. lea calculates the address and puts it in [B]
- add, sub, imul (signed), mul (unsigned), sal, sar, shr, shl, xor [A][B] Calculate A (operator) B and store in B
- inc (++), dec (--), neg (-), not (~) [A]

Return value: %rax/%eax

Instruction Sizes:

- B : 8 bits
- W : 16 bits
- L : 32 bits
- Q : 64 bits

Addressing Methods:

- %rax : the value in %rax

- (%rax) : the value in the memory address contained in %rax (dereferencing a pointer)
- 4(%rax) : the value 4 bytes after the address in %rax
- [N](%rax,%rbx,8) : the address of ((%rbx * 8) + %rax + N)
    - mov will get what is at this address; lea will get the address (number)

lea vs mov: lea does not go into memory, while mov gets what is in memory at the specified address. (lea is used for addition)

Flags (EFLAGS):

- cmp [A],[B]
    - set flags based on the result of B-A
- test [A],[B]
    - set flags based on result of B&A

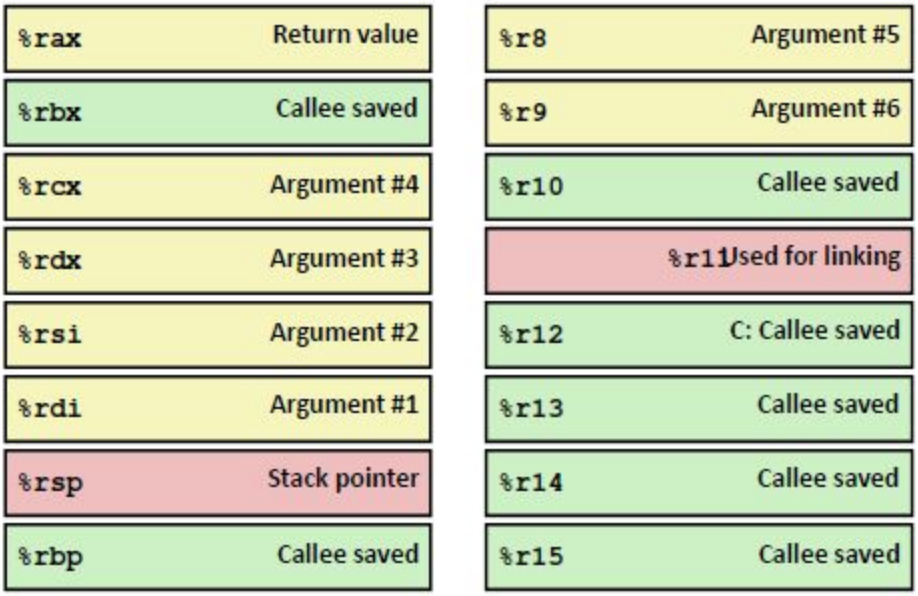
Actual flags:

CF (carry)
ZF (zero)
SF (sign)
OF (overflow)

Flags always set by arithmetic operations (except lea)

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

x86-64



Function Arguments (first-last)

1. %rdi
2. %rsi
3. %rdx
4. %rcx
5. %r8
6. %r9
7. Other arguments go onto the stack (like IA-32)

Callee-Saved Registers

1. %rbx
2. %rbp

Red Zone: 128 bytes longer than the stack pointer. Guaranteed not to be overwritten

Procedures

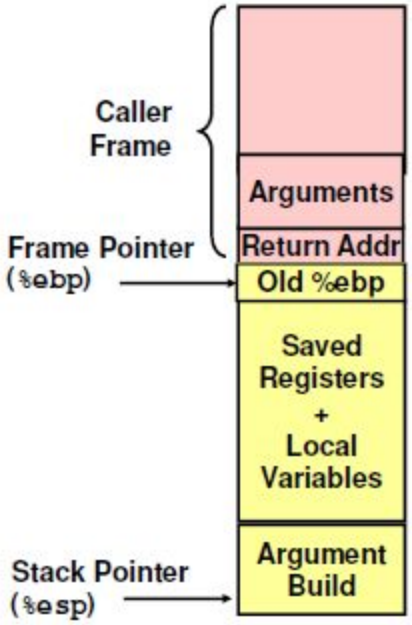
Jmp vs Call

Jmp just changes %eip

Call changes %eip and pushes a return address to the stack

(code must still `mov %esp, %ebp`)

Ret: pops the top of the stack into %eip (return address must be top of the stack!)



Data Structures

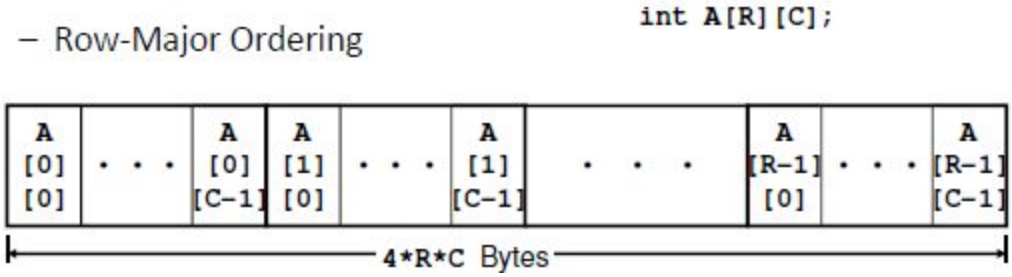
Alignment

All multi-byte values are stored at an address that is a multiple of their size.

x86-64

- shorts : multiple of 2.
- ints : multiple of 4.
- floats : multiple of 4.
- doubles : multiple of 8.
- pointers (8bytes) : multiple of 8.

Arrays



Structs

Contiguously allocated in memory
Entire struct must conform to alignment requirements of largest value
Offset computed by compiler

Unions

Overlay data (same memory)
Start every set of data from smallest memory address (but little endian prints largest byte first!)

Optimizations

Blockers:

- Memory **Aliasing**
- Procedure Call Side-effects
- Analysis only within a procedure

Methods:

- **Code Motion** (move expensive code out of loops)
- **Strength Reduction** (make low-powered instructions)
- Use Registers & take advantage of **blocking**

```
for (y = 0; y < dim; y += BLKSZ)
for (x = 0; x < dim; x += BLKSZ)
      for (i = y; i < y+BLKSZ; i++)
      for (j = x; j < x+BLKSZ; j++)
```

Memory Hierarchy:

- Main Memory/**DRAM**
    - highest-capacity
    - slowest & destructive
    - holds “pages” that the OS manages
- Caches/**SRAM**
    - expensive and fast & non-destructive
    - Hold “**cache blocks**” (managed by machine)
    - **Miss rate** (how often misses), **hit time** (cycles to get memory from hit), **miss cost** (cycles to get memory from miss)
    - Takes advantage of **locality** (temporal and spatial)

Parallel Computing

- **Domain Decomposition** (taking apart the data)
- **Functional Decomposition** (parallelizing different functions)
- **Pipelining**

Amdhal’s Law: Total Execution time = Parallel + Series :: series matters!
Processors advance in two ways:

1. Clock/Cycle Time
    - Feature Size (how big the transistors are)
    - Dynamic Thread Management (manages temperature of CPU dynamically by adjusting clock)
2. Microarchitectural Innovation
    - In-Order vs. Out-Of-Order cores:: take advantage of pipelining to put faster instructions first (instruction-level parallelism)
    - Simultaneous MultiThreading (SMT): Fill in the “blanks” of the pipeline by stuffing other threads in... sometimes bad threads worsen performance overall
    - Heterogenous processors: a gpu and cpu on same chip (slow parallel/simple and fast complex/sequential)

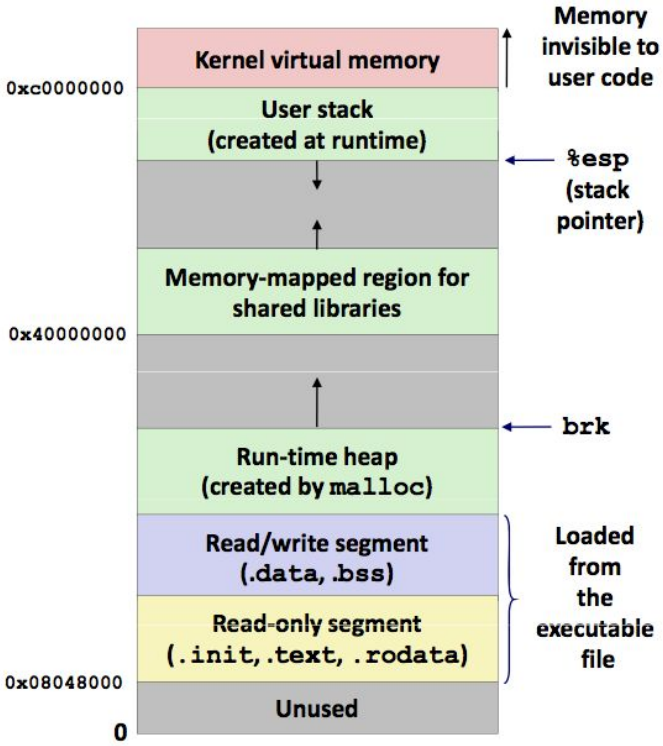
OpenMP

- **Fork/Join** model (awaken other threads)
- NOT concurrency, but true parallelism (no OS)
- Pragmas:
    - **`parallel for`** divides a for loop into tasks for each processor. The total number of iterations needs to be known at entry to the loop
    - **`int omp_get_num_procs()`** returns the number of physical/usable cores
    - **`void omp_set_num_threads(int t)`** set the number of threads that omp will use
    - … **`private(var1, var2)`** sets these variables private to each processor (uninitialized on entry/exit!)
    - … **`firstprivate (var)`** sets the variable for each processor to be initialized to what it was previously in the code
    - … **`lastprivate (var)`** sets the variable at the end of the loop to be the natural end of the loop
    - **`parallel`** next statement (or block) is parallel (duplicated on each processor); can use #pragma omp for inside this
    - **`nowait`** don't wait for all the processors to finish before continuing after the statement; this eliminates a "barrier synchronization" at the end
    - **`single`** only a single thread executes the following statement (or master)
    - **`reduction(op:var)`** does the operation (op) to var at the end; *needs* to be associative!!
    - **`critical`** marks a section that can only be entered with one processor at a time
    - **`barrier`** Make sure every thread reaches this point before continuing
- Prevent race conditions
- Prevent deadlock: lock/unlock resources in the same order (rank resources)

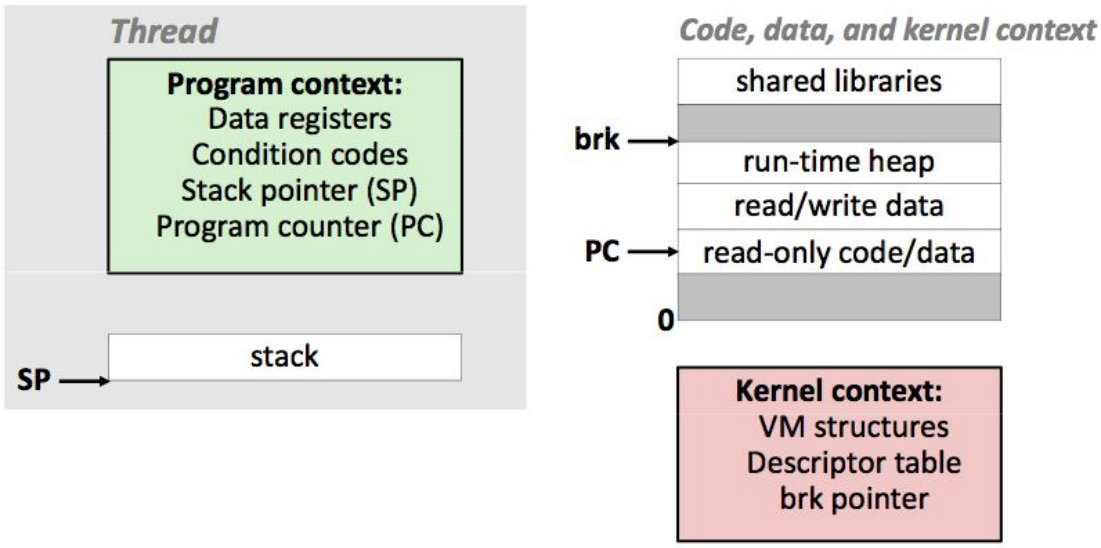
Flynn's Taxonomy

1. SISD: Traditional CPUs
2. SIMD: GPUs (data-level parallelism)
3. MIMD: Cell processor (multiple streams of instruction & data)
4. MISD: Uh...

Operating System



Threads/Processes



- Kernel handles context switching (within each core)
- Threads: only unique threads/stack

Virtual Memory

- Allows programs to “use” all memory available
- OS (or hardware) maps virtual addresses to actual addresses
- Also allows for pages (page tables!)
    - “page fault” if not in DRAM
    - Problem with pages: page thrashing (switching too much)

Exceptions

- Asynchronous Exceptions (“interrupts”)
    - the interrupt pin on the processor is triggered (comes from the interrupt controller on the IO Bus)
    - Different timing than the CPU
- Synchronous Exceptions
    - **Traps** (intentional, recoverable; returns to next instruction): breakpoints, system calls, etc.
    - **Faults** (unintentional, mostly recoverable): page fault, protection fault, divide by 0, etc. Can be handled by applications explicitly
    - **Aborts** (uninteintional, never recoverable): machine check, etc.
- Options to handle exceptions:
    - Return to last instruction
    - Go to next instruction
    - Abort

Linking

- Less re-compilation, readability, modularity, etc. from multiple files (Modularity, Efficiency, Space)
- .o file (relocatable object file): compiled, but not linked to actual addresses, so not executable
- .a file (archive) has a lot of .o files, linked at compilation time (and built into the same binary, **static linking**)
- **Dynamic Linking**
    - link at execution to save memory on binary, allow multiple programs to use same library, cache locality
    - .so or .dll
    - The loader of the OS links when the process starts (loadtime; or loaded at runtime)
    - shared libraries only in memory once :: virtual memory mapping
    - Problems: viruses, replace with malicious code, etc.
- 1) Symbol Resolution [symbol table] 2) Relocation [actual addresses]

MIPS

(x86: operator op, oper/dest :: xxx -> yyy)
(MIPS: operator dest, op, op :: zzz <- xxx yyy)

RISC (reduced instruction set)	CISC (complex instruction set)
• Fixed-length instructions (4B) • embedded/mobile systems • Takes more memory • simpler instructions • easier to pipeline/decode • limited addressing modes (only base and displacement) • More registers (32-ish) • all operations only apply to registers • **load/store machine** • more parallelism with a smaller pipeline! • ARM	• Lots of ops • Storage of instructions is smaller (bad for CPU) • x86 • variable-length instructions: hart to decode, but code size goes down • addressing mode (many! they are complex, but the number of instructions goes down. ex. displacement(base,index,scaling) in x86) • Few registers (8-16) • **Register-Memory Machine** • Intel/AMD (but break instructions into microps :: easier to pipeline)

$0	$zero	Always 0	$16	$s0	
$1	$at	For assembler	$17	$s1	Callee-saved
$2	$v0	Return val	$18	$s2	Registers
$3	$v1	Return val	$19	$s3	
$4	$a0	Arg 0	$20	$s4	
$5	$a1	Arg 1	$21	$s5	
$6	$a2	Arg 2	$22	$s6	
$7	$a3	Arg 3	$23	$s7	
$8	$t0		$24	$t8	
$9	$t1		$25	$t9	
$10	$t2	Caller-saved	$26	$k0	Kernel
$11	$t3	Temporary	$27	$k1	reserved
$12	$t4	registers	$28	$gp	Global pointer
$13	$t5		$29	$sp	Stack pointer
$14	$t6		$30	$fp	Frame pointer (ebp)
$15	$t7		$31	$ra	Return address

- Labels stay intact
- Immediates are 16 bits
- Loads/stores
  - `li $v0,4` (load immediate 4 into register)
  - `la $d0, msg` (place the address of msg into register)
  - `lw $t0, x` (load what is *at* x and put in register)
  - `sw $t0, y` (store what is in $t0 and put at memory location y)
- Mathematical Operations
  - `OPER **DEST**, OPR1, OPR1`
  - `add $t0, $t3, $t4` (place $t3+$t4 into $t0)
  - `sub, mul`, etc. (different instructions for immediates: just add an `i`)
  - Non-destructive!!
- Jumps
  - `j` (jump; can't jump everywhere because can't type a 32-bit address into the 4 bytes with the jump instruction!)
  - `jal` (jump and link; changes $ra; like call)
  - `jr` (jump to register)
- Branches
  - `beq $t0, $t1, label` (if $t0 == $t1, branch to label)

    - `bneq` (branch if not equal), `bltz` (branch if less than zero), etc.
- Push:
    - `addi $sp, $sp, -4`
    - `sw   $s0, ($sp)`
- Pop:
    - `lw $s0, ($sp)`
    - `addi $sp, $sp, 4`

System Calls:

```
li $v0, [code]
syscall
```

	Code	Registers
Print int	1	Put number in $a0
Print string	4	Put string address in $a0
Read int	5	Result goes in $a0
Read string	8	Put buf address in $a0 Put buf size in $a1
Exit	10	

(read int goes into $v0)