## Questions:

1. m/c (high-level concepts like VM, cache, locality, etc.)
2. 1-word answers (high-level concepts: SIMULTANEOUS MULTITHREADING)
3. memory-level optimization
4. MIPS
5. gdb dump question
6. Buffer Overflow
7. Parallel Programming
8. Machine-Independent Optimization

## 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!

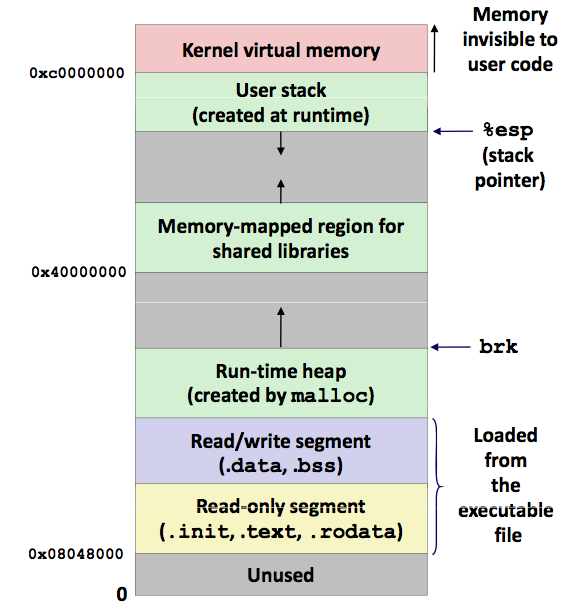
Processors advance in two ways:

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)

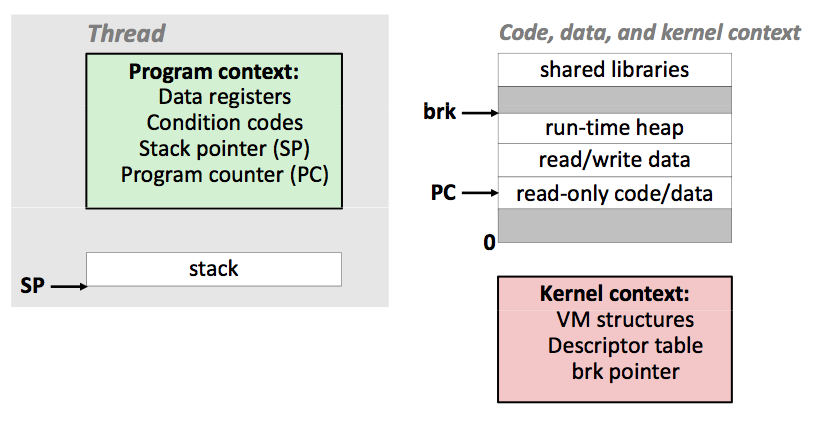
### OpenMP

* **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)

## 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

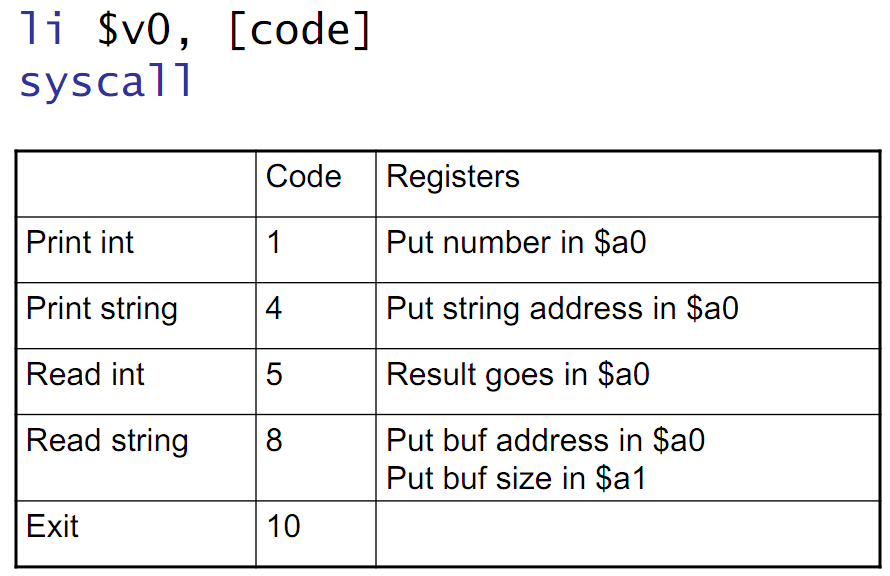
(x86: operator op, oper/dest :: xxx -> yyy )

(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) |

* 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:



(read int goes into $v0)