CS61C Midterm 1

Number Representation A bit has 1 of 2 values (0 or 1). A byte is 8 bits.

Radices	Base	Prefix
Decimal	10	None
Binary	2	0b
Hexadecimal	16	0x

Convert to decimal: Sum powers of base multiplied by respective bit

Binary → hex: For each group of 4 bits starting from rightmost, convert to hex digit (0–f). Do reverse to convert to Binary.

- We can also use IEC prefixes; 2^{16} = 64 Ki, 64 Gi = 2^{36}
- Binary addition: 0+0=0; 0+1=1; 1+0=1; 1+1=10; 1+1+1=11

Signed Integers: Two's Complement: Most significant bit is 0 if pos, 1 if neg.

- To negate a two's complement number, flip the bits and add 1.
- Range of n-bit number: -2^{n-1} to $2^{n-1}-1$ (vs. 0 to 2^n-1 for unsigned)
- Arithmetic operations of addition, subtraction, and multiplication are identical to those of signed integers (as long as overflow discarded)
- An n-digit number in base b can represent b^n distinct items

Bitwise Operations: AND (&), OR (|), XOR (^), NOT (~) $\tau_{ceil(\log_b(x))}$

x&1 == x|0 == x^0 = x; x&0 = 0; x^1 = ~x; x|1 = 1

Bit Shifts

Logical left shift (x << n): Effect is multiply x by 2^n

Logical right shift (x >> n): Effect is divide x by 2^n (rounded down)

Note that logical right only works on unsigned integers. Arithmetic right shift fills bits with high order sign bit, but is not equivalent to dividing by 2^n.

C Low-level function-oriented language

Only 0 and NULL are false; everything else true.

- Uses explicit pointers: integers containing memory addresses.
- sizeof() returns size of something in bytes. Usually, char is 1 byte, and ints and pointers are 4 bytes.

<ptr type>* ptr = <address>; &x returns address of x; *ptr dereferences a ptr

Pointer Arithmetic: (automatically scaled by sizeof(ptr type))

Operator	Returns	Effect
*p++	*p	p=p+1
*--p	*(p-1)	p=p-1
++*p	(*p)+1	*p=*p+1
(*p)++	*p	*p=*p+1

Pointers make it possible to simulate "pass by reference". Recall C, Java, Python are all pass by value.

Arrays

- Initialization: int arr[10]; or int arr[] = {1,2}; * Holds garbage until assigned!
- Alternatively, int *arr = {1,2}. Here, arr is a pointer to arr[0] rather than an array variable. Subtly different (sizeof differs)
- Access into arrays by arr[i] or *(arr + i)
- Strings are arrays of chars, terminated by '\0'.
- Structs are custom data types. Not a Java object, but similar purpose
- typedef struct Node {...} Node; often used to define structs
- bar->member == (*bar).member

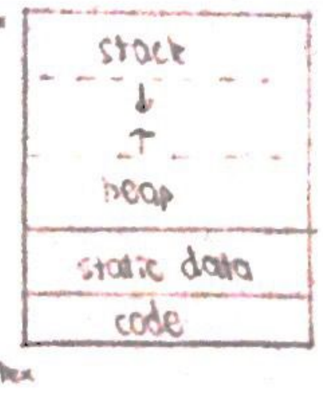
Functions must be declared before use, e.g. char* foo(char* str);

A ptr to a function looks like char *(*f)(char* str);

So one could then do f = &foo;

Note that pointers to pointers are also possible, denoted **ptr.

Memory Management in C



Stack: Function local variables. Freed when function returns. (think GIA frames)

Heap: Dynamic data requested by malloc() and others

Static: Data declared outside functions, i.e., static & global variables, string literals, constants

Code: Machine instructions. Cannot be modified. Includes defines & macros.

malloc(size_t n): Returns void* ptr to block of n bytes

calloc(size_t n): Same as malloc, but initializes everything to 0

realloc(ptr, size_t n): transfers contents to new block, frees ptr, and returns void* ptr to new block.

free(ptr): frees memory pointed to by ptr. Must have been malloced and must not have been modified.

Internally, malloc is implemented with data structures. High level languages have completely automated memory management

Assembly

- A CPU executes instructions. The particular set of instructions a CPU executes is an Instruction Set Architecture (ISA).
- RISC (reduced instruction set computing) philosophy means small and simple instruction set to make hardware easier and faster. Let software do complicated operations.

MIPS is one RISC language. Very simple and elegant.

- Assembly operands are registers built into hardware.
- There are 32 registers, each of 32 bits (a word) in MIPS. See green sheet for full list; refer to them by name for readability
- Immediates are numerical constants
- There are signed and unsigned operations; the only difference is overflow is detected in signed, ignored in unsigned.

Word addresses must be multiples of 4, halfwords 2, and bytes 1.

lw $rt, imm($rs): $rt = *($rs + imm) (imm sign-extended)

sw $rt, imm($rt): *($rs + imm) = $rt

Translating C to MIPS

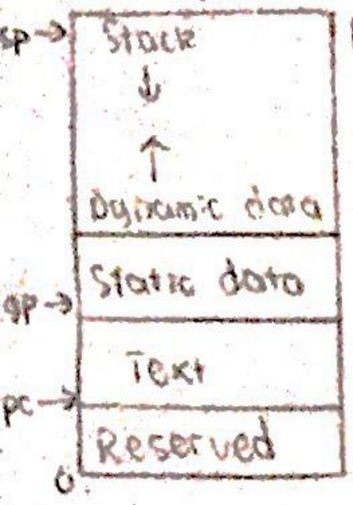
- Arithmetic operations are straightforward, sometimes several instructions
- Control: If → beq or bne. Use slt w/ beq or bne for inequalities
- Loops: Have loop label and exit label. At start of loop, have beq/bne to exit on condition. Follow with loop body that ends with j loop.

Note: Instead of manually setting return address, use jal <fn> which saves PC+4 to $ra and jumps to <fn>. jalr is the same, except for addresses in registers.

Function Calls

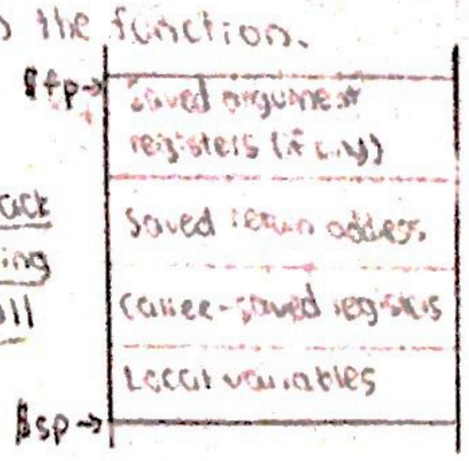
Registers that are saved across function calls are called callee-saved, and not saved called caller-saved.

Use the stack to store local variables that don't fit in registers, and any callee-saved registers used in the function.



MIPs memory allocation

Dynamic data requested by sbrk syscall; stack contracted manually; text contains instructions



Stack during call

Pseudo-instructions are translated into TAL instructions by assemblers. For example, mult and div use special registers hi and lo to get 64 bit results.

li expansion: (load immediate): li $rs, imm

```
if INT16_MIN ≤ imm ≤ INT16_MAX:
    addi $rs, $0, imm
else:
    mask = 0x0000ffff;
    int upper16bits = imm >> 16;
    int lower16bits = (imm << 16) >> 16;
    upper16bits &= mask;    // Strip off upper 16 bits
    lower16bits &= mask;
    lui $at, upper16bits
    ori $rs, $at, lower16bits
```

Example structure of .s file: *Here we implement the factorial fn:

```
.data
n: .word 10
str: .asciiz "Hello"
.text
main:
    lw $a0, n      } load argument, call fn
    jal fact
    li $v0, 10     } exit program
    syscall
fact:
    addi $sp, $sp, -8    } prologue;
    sw $ra, 4($sp)       } decrement stack, sw
    sw $a0, 0($sp)       } $ra and vars
    slti $t0, $a0, 1     } body
    beq $t0, $0, ELSE    }
    add $v0, $0, 1       }
    addi $sp, $sp, 8     } epilogue;
    jr $ra               } restore registers
ELSE:
    addi $a0, $a0, -1
    jal fact
    lw $a0, 0($sp)
    lw $ra, 4($sp)
    addi $sp, $sp, 8
    mul $v0, $a0, $v0
    jr $ra
```

```
int fact(int n) {
  if (n<1) return 1;
  return n * fact(n-1);
}
```

Instructions as Numbers

All MIPs instructions are encoded in 32 bits (words), divided into fields. There are three main instruction formats, R, I, and J.

R-type instructions include arithmetic operations & jumps.
All have opcode 0. Most follow argument order op $rd, $rs, $rt, but see green sheet for specifics. Only format w/ nonzero func.

I-type instructions include operations with immediates and branches. They have a 16-bit immediate field that is sign-extended (except andi, ori, xori) and usually follow argument order op $rt, rs, imm (except branches, where $rs comes first)

J-type instructions (just j and jal) follow format op label.

MIPS Addressing

PC-relative: Calculates address as follows:
PC = PC+4 + (immediate << 2) ← since addresses are word-aligned
Used by I-type branching instructions. Can represent -2^{15} to $2^{15}-1$ addresses from PC+4.

Psuedo-direct: Uses upper 4 bits of PC, 26 bits of addr, and 00 lowest bits: PC = {(PC+4)[31:28], target addr, 00}
Used by J-format instructions. If 6 is the top 4 bits of PC+4, then can represent 0x6000 0000 to 0x6fff ffff.

Base Displacement: Adds immediate to register value. Used by lw, lb, lh, sw, sb, sh.

Register Addressing: Use value in register. Used by jr.

Filling out 32 bit instructions: Identify format, fill out template with register values and other fields, put immediate/jump values appropriately. Convert fields to binary, then hex.

CALL (Compiler, Assembler, Linker, Loader)

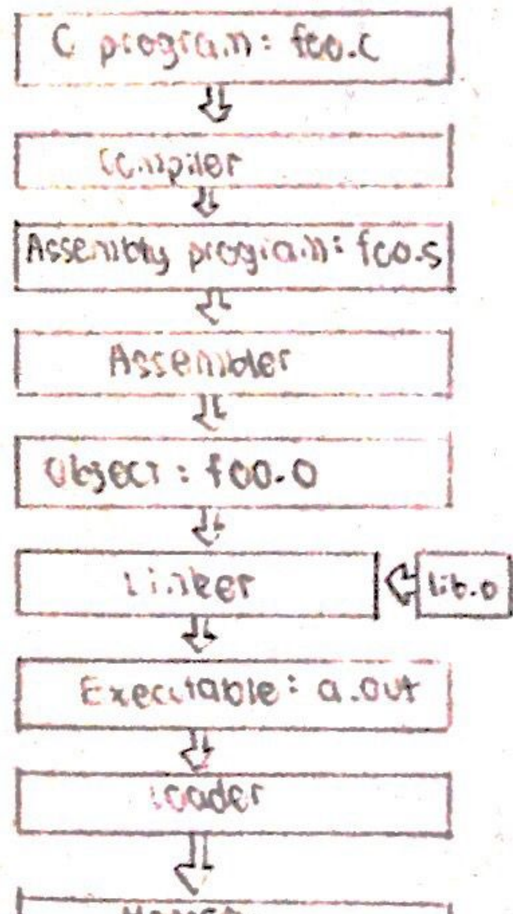
Compiler: Take high level code and produce assembly language code file. This is what we do translating C to MIPS by hand.

Assembler: Expand pseudo-instructions, handles PC-relative addresses. Does 2 passes to resolve addresses from forward reference. Produces .o file with following components:
- Object file header: size & position of pieces in the object file
- text segment: machine code
- data segment: static data from source file
- Relocation table: checklist of items whose address needs filling in (e.g., labels from j or jal, external lib files, data from static section)
- Symbol Table: List of labels, data from .data which can be accessed from other files. Linker uses this to search for references in relocation.

Linker: Combine several object (.o) files into single executable. Concatenates text and data segments, fills in absolute addresses for items in relocation table.

Loader: An OS task; loads executable and arguments into memory, begins execution.

- We've discussed statically-linked libraries, which load entire contents of lib into executable. Dynamically-linked libraries involve linking at run-time to segments needed by program. More complex for OS, but overall huge improvement



CS61C Midterm 2

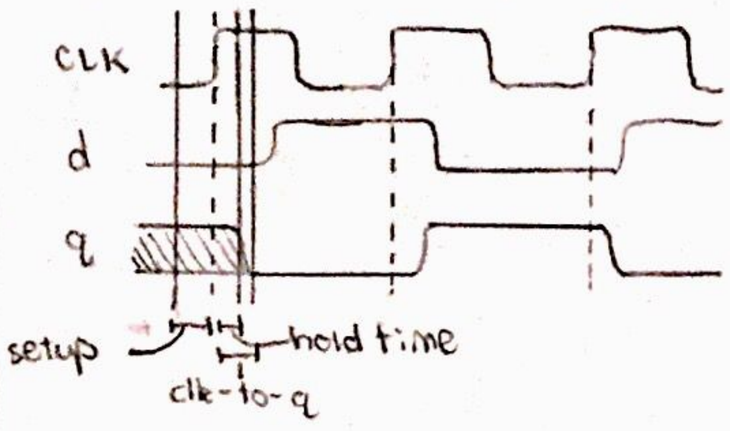
Boolean Algebra: Simplify Expressions!

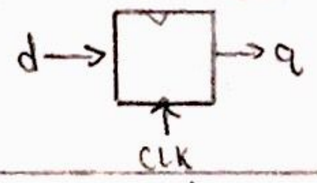
$X\bar{X} = 0$	$X + \bar{X} = 1$
$X0 = 0$	$X + 1 = 1$
$X1 = X$	$X + 0 = X$
$XX = X$	$X + X = X$
$XY = YX$	$X + Y = Y + X$
$(XY)Z = X(YZ)$	$(X+Y)+Z = Z+(Y+Z)$
$X(Y+Z) = XY + XZ$	$X + YZ = (X+Y)(X+Z)$
$XY + X = X$	$(X+Y)X = X$
$\bar{X}Y + X = X + Y$	$(\bar{X}+Y)X = XY$
$\overline{XY} = \bar{X} + \bar{Y}$	$\overline{X+Y} = \bar{X}\,\bar{Y}$

Convert a truth table into boolean expression using sum of products, then laws of boolean algebra to simplify. Then implement logic gates.

Synchronous Digital Logic (SDS) consists of combinational logic circuits, and sequential logic (state elements)

Registers consist of flip flops. Ex of positive-edge-triggered:



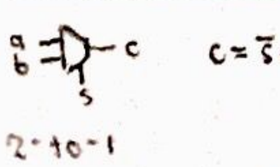


To prevent hold-time violation, must have $t_{hold} \le t_{logic} + t_{cdreg}$ where t_{logic} is shortest path between registers and t_{cdreg} is the clk-to-q delay of the register.

The minimum period (time of critical path) is $T \ge t_{clk\text{-}to\text{-}q} + t_{CL} + t_{setup}$ where t_{CL} is the time of longest path through combinational logic between registers.

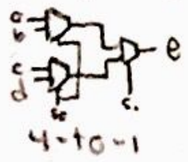
Hence maximum clock frequency is $1/T$ (note 1GHz = 1 cycle/ns)

Simple Functional Units: Multiplexer (mux) used to select output



$c = \bar{s}a + sb$

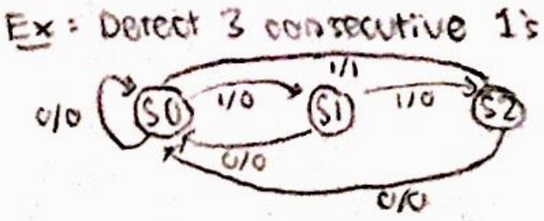
2-to-1



4-to-1

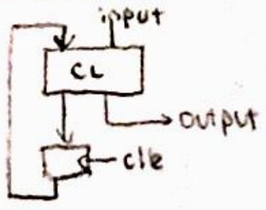
Finite State Machine

Ex: Detect 3 consecutive 1's from input



input/output on edges

In hardware:



Specify CL with truth table mapping every possible state and input to next state and output. Every clock cycle, move to new state and produce output

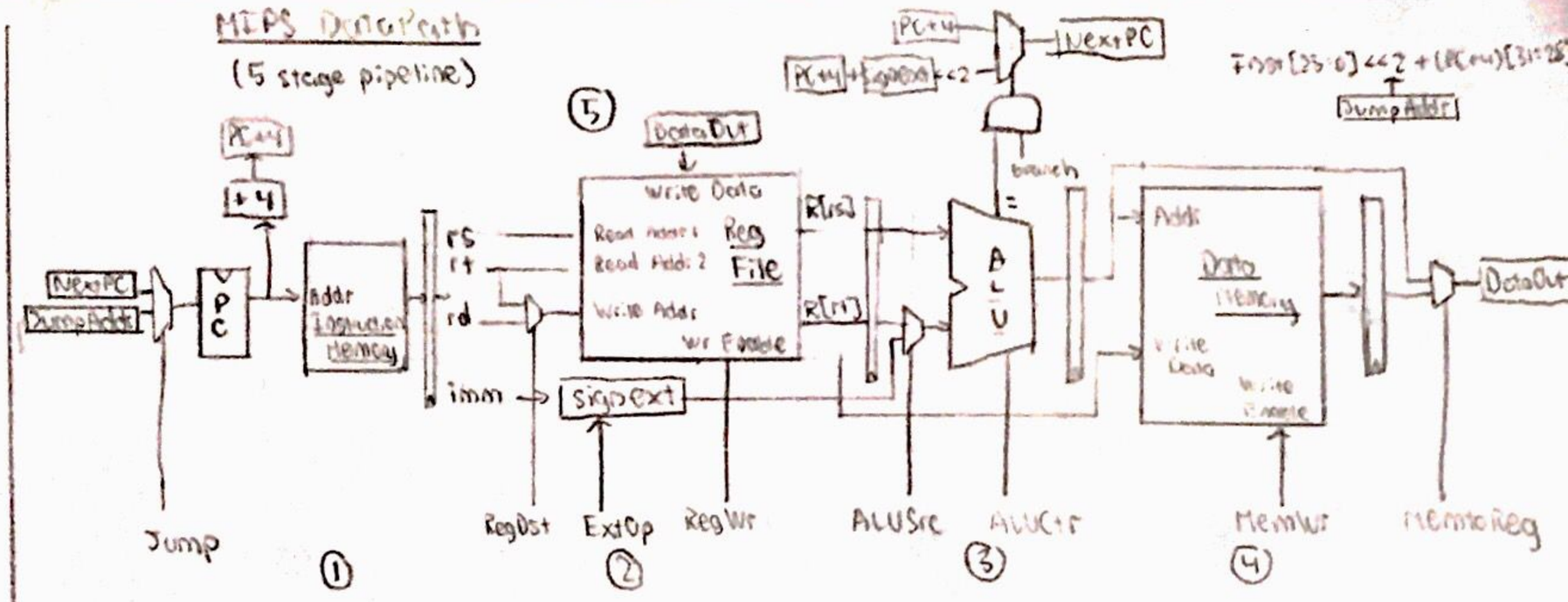
MIPS DataPath Stages

1. Instruction Fetch: load inst from memory, set PC appropriately
2. Instruction Decode: Gather data from fields, set control signals, read appropriate registers, sign-extend the immediate
3. Execute: Perform arithmetic/logical operations
4. Memory: Read/write to memory (only load/store instructions)
5. Register Write: Write result to register (stores, branches, jumps idle)

Note: Only load instructions use all stages. (Jumps use fewest)

Critical path for a load: $t_{clk\text{-}to\text{-}q} + t_{instfetch} + t_{memread} + t_{REFetch} + t_{mux} + t_{ALU} + t_{REFsetup}$

MIPS Datapath (5 stage pipeline)



Control Signal Examples:

	add	ori	lw	sw	beq	j
RegDst	1	0	0	X	X	X
ALUSrc	0	1	1	1	0	X
MemtoReg	0	0	1	X	X	X
RegWr	1	1	1	0	0	0
MemWr	0	0	0	1	0	0
Branch(nPCsel)	0	0	0	0	1	X
Jump	0	0	0	0	0	1
ExtOp	X	0	1	1	X	X
ALUCtr	add	or	add	add	sub	X

Actual control logic implemented with AND logic used on opcode and func, OR logic on the decoded instruction

Latency: time for single instruction to finish

Throughput: Number of operations finished per unit time

Pipelining

Does not help latency of single task, but throughput of entire workload. In MIPS, place registers in between each stage to create simple 5-stage pipeline.

Let T_c denote time between completion of instructions.

Then $T_{c,pipelined} \ge \frac{T_{c,single\text{-}cycle}}{\#\text{ of stages}}$ * unbalanced pipelined stages and setup & clk-to-q delays prevent this from being equal

Speedup is then ratio of $T_{c,pipelined}$ and $T_{c,single\text{-}cycle}$.

Pipelining Hazards arise when instructions have dependencies

Structural Hazard: Required resource needed in multiple stages

Data Hazard: Require data from previous instruction

Control Hazard: Flow of execution depends on earlier instruction

- Solve memory structural hazard by having separate data and instruction caches. Solve register structural hazard by writing/reading in 1st/2nd half of clock cycle or having independent read/write ports.
- Forwarding can solve many data hazards. We use a forwarding unit to send result of execute stage from instruction to future instructions that need it, in their execute stage. In general, an execute stage will need the forwarded output of the first/second/third last instruction's ALU if rt or rs of current instruction is same as past instruction's regDst.

However, forwarding can't solve all data hazards. An instruction following a load that needs the result of it must stall for one cycle before being forwarded. There is a load delay slot after every load; let the compiler (trig) to insert a non-dependent instruction into it.

- Control hazards occur with branches or jumps. Can solve by stalling for two cycles. This is bad, so first step is introduce special branch comparator in decode stage to immediately set PC if branch taken. This eliminates one stall. MIPs uses a branch delay slot (and jump delay slot) which contains an instruction always executed. It is job of assembler to place some instruction preceding the branch in the delay slot as long as it doesn't affect logic; if none, just a nop. This is only for MIPs TAL, which is why jal sets pc=pc+8.

Caches

Real memory access patterns exhibit temporal and spatial locality.

Divisions of a memory address for cache use:

	tag	set index	block offset
Bits:	remaining	$\log_2 S$	$\log_2 B$

where B is the size of the cache blocks (in bytes) and S is the number of sets.

Let K be the number of cache blocks (cache size / block size).

Cache Types

Fully Associative: 1 set with K 'ways' (no index field)

Direct Mapped: K sets with 1 'way'

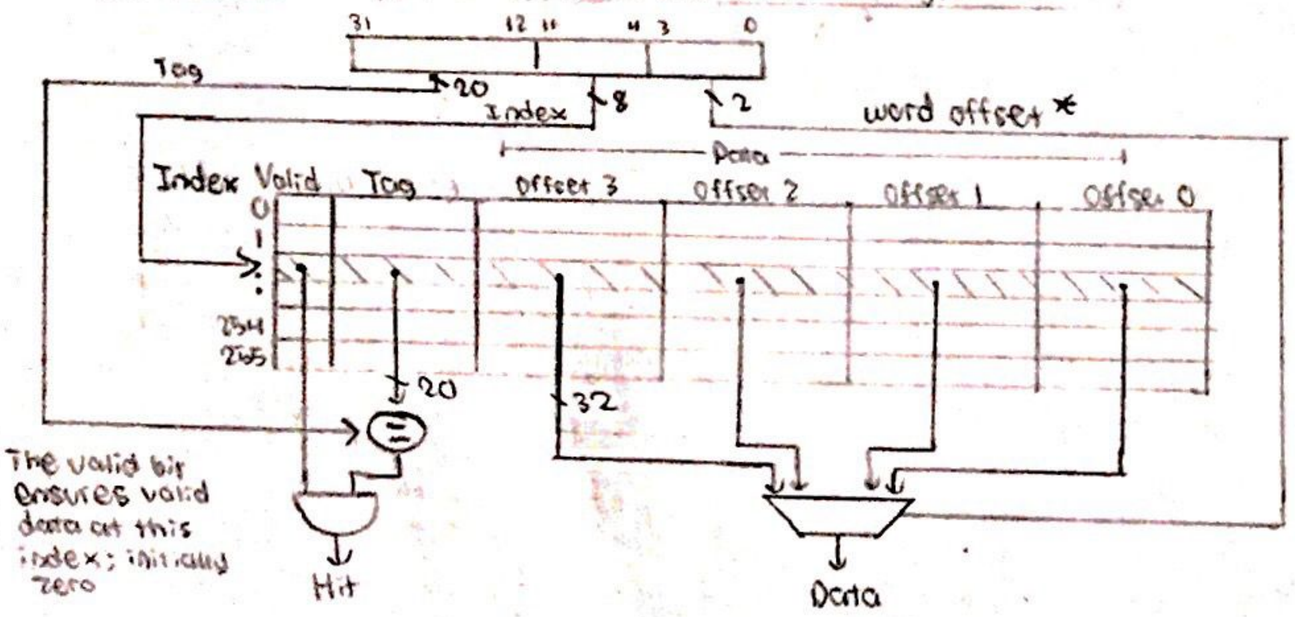
N-way Associative: K/N sets with N 'ways'

→ Total cache capacity = associativity × S × B

so for fixed capacity and block size, increasing associativity increases hit time (muxes) but reduces miss rate.

Remember: on a cache miss, entire cache block is loaded from memory.

Example of Direct-Mapped 4KiB cache with 16 byte blocks:



*Since most ISAs have word-aligned memory, only use upper 2 bits of block offset in address field.

Other cache types are similar, but have N comparators, one for each set. And of course, for a given cache size, increasing associativity means reducing number of blocks or size of blocks.

Writing Policies

Write-through: Write to cache & memory every access (using write buffer)

Write-back: Write to cache, and only to memory when block is getting evicted. As a result, cache must contain "dirty bit" which is set whenever a block is written to.

Writing-miss Policies

Write-allocate: data at write memory is loaded into cache followed by a write-hit operation. Usually paired w/ write-back cache

No-write-allocate: only write directly to memory. Usually paired w/ write-through cache.

Average Memory Access Time (AMAT) = Hit Time + Miss Rate × Miss Penalty

Cache Eviction Policies (for associative caches) Least-Recently-used is most standard choice. Can be implemented with 1 bit for 2-way associativity; otherwise more complicated.

Cache Miss Types (compulsory, conflict, capacity)

- If the data has never been seen before, it is compulsory. If data has been seen and there was space in cache, it is a conflict (could be solved w/ fully associative cache). Otherwise, it is a capacity.

- Instruction & data cache misses are hugely detrimental to performance; pipeline must stall for hundreds of cycles. Prefetching data in advance can help. MIPS supports the implicit prefetch `lw $0 0($t0)` to load Mem[R[t0]] without stalling the pipelining on a miss.

- Miss penalties can be reduced with a memory hierarchy of increasing cache sizes. Most CPUs have L1 cache, L2 cache and L3 cache. The global miss rate, or fraction of reference that miss some level of cache, is much smaller than the local miss rate, or fraction of references to one level that miss. The local miss rate of L2 = L2 misses / L1 misses, for example. The global miss rate is L1 × L2 × L3 miss rate

IEEE 754 Floating Point Standard

Single Precision:

sign	exponent	fraction
1 bit	8 bits	23 bits

Normalized Number: $(-1)^S \times (1.\text{fraction}) \times 2^{\text{exponent}-\text{bias}}$

Let ex, fr be # of bits for exponent and fraction, respectively. The bias is $2^{ex-1}-1$ (so 127 for single precision). The largest possible exponent MAX is $2^{ex}-1$ (so 255 for s.p.).

Range: $\pm 2^{1-\text{bias}}$ to $(2-2^{-fr}) \times 2^{MAX-1-\text{bias}}$

For s.p., this is $\pm 2^{-126}$ to $(2-2^{-23}) \times 2^{127}$.

For a given exponent e, the gap between numbers is 2^{k-fr}. Hence there is a "sliding window" of precision.

Denormalized Number: $(-1)^S \times (0.\text{fraction}) \times 2^{1-\text{bias}}$

Range: $\pm 2^{1-\text{bias}-fr}$ to $(1-2^{-fr}) \times 2^{1-\text{bias}}$

For s.p., this is $\pm 2^{-149}$ to $(1-2^{-23}) \times 2^{-126}$

Summary of fields:

Exponent	Fraction	Object
0	0	±0
0	≠0	±Denorm
1 to MAX-1	any	±Norm
MAX	0	±∞
MAX	≠0	NaN

Ex: Express -38.125 in floating point

1) Put in fixed point binary: 100110.001 (2^{-1}, 2^{-3})

2) $100110.001 = 1.00110001 \times 2^5$

Take S = 1 (for negative), exponent = 5 + bias (in binary) and fraction = 00110001 to populate fields.

Performance

CPU Time is time spent in processor. Iron Law of Performance:

$$\text{CPU Time/Program} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{clock cycle}}{\text{instruction}} \times \frac{\text{seconds}}{\text{clock cycle}}$$

↑ Instruction count; ↑ CPI: clock cycles per instruction; ↑ 1/clock freq

Amdahl's Law Suppose that enhancement E accelerates a fraction F ($0 \le F \le 1$) of the task by a factor S.

Then the speedup is $\frac{1}{(1-F)+F/S}$

Notice that $\lim_{S \to \infty} \text{speedup} = \frac{1}{1-F}$. The idea is that the theoretical speedup is limited by the part of the task that cannot be improved. In the context of parallel computing, this means parallel computing with many processors is only useful for very parallelizable programs.

CS61C Final

Flynn's Taxonomy: A classification of computer architectures.
- SISD: (single instruction/single data stream): standard uniprocessor machine
- SIMD: Multiple data streams against a single instruction stream, like in a GPU.
- MISD: Rare
- MIMD: multicore processors or WSCs

Data Level Parallelism: Executing same operation on multiple data streams. Loop unrolling exploits data parallelism. Streaming SIMD Extensions (SSE) are Intel extensions to x86 architecture that provide C intrinsics that map directly to assembly. Through 128-bit XMM registers, they operate over multiple contiguous array elements simultaneously.

Thread-Level Parallelism: A thread is a sequential flow of instructions, each thread having own PC and processor registers. OS swaps threads in and out of execution.

Parallel multithreading libraries are available; OpenMP is an API for this.

Example:
```
#include <stdio.h>
#include <omp.h>
int main() {
   int thread_num, thread_id;
   /* Fork into threads, each with private var thread_id */
   #pragma omp parallel private(thread_id)
   {
      thread_id = omp_get_thread_num();
      if (thread_id == 0) {      // only master thread does this
         printf("Number of threads: %d\n", omp_get_num_threads());
      }
   } /* All threads join master and terminate */
}
```

Two memory accesses form a data race if different threads attempt to access same location, and at least one is a write, after one another.

Lock Synchronization uses a 'lock' to grant access to a region 'critical selection' so only one thread can operate at a time.

Hardware support is required: In MIPs introduce 2 new instructions:

Load Linked: ll rt, off(rs)
Store conditional: sc rt, off(rs)

// Operate similar to lw/sw, but sc overwrites R[rt] w/ 1 if success (mem has not changed since ll instruction) or 0 otherwise.

Example of 'test and set' in MIPs:
```
Try: addiu $t0, $zero, 1   // $t0 = 1
     ll $t1, 0($s1)        // $s1 is the lock
     bne $t1, zero, Try    // lock is set; try again
     sc $t0, 0($s1)        // Attempt to set lock
     beq $t0, $zero, Try   // Is not set, try again
Locked: #critical selection
Unlock: sw $zero, 0($s1)   // Reset the lock
```

Another synchronization method is semaphore; using increment, decrement ops, allow up to n threads to enter.

Deadlock occurs when each thread is waiting on another. Solve by having one thread 'drop' and try again random time later.

In OpenMP, we can use shorthand #pragma omp parallel for to automatically break up for loop into appropriate # of threads. Also reduction to specify 1 or more variables that are private are subject to reduction after parallel region. Syntax: reduction(operation: var) Example:
```
void main() {
   int i;
   double x, pi, sum = 0.0;
   #pragma omp parallel for private(x) reduction(+:sum)
      for(i=1; i<num_steps; i++) {
         x = (i-0.5)*step; sum = sum + 4.0/(1+x*x);
      }
   pi = sum/num_steps;          // Reduction op must be commutative
}                                  and associative
```

Cache Coherence: Different cores have different caches (L1) while usually sharing L2 or higher. How to account for writes? Idea is to have an interconnection network; on every store instruction, check contents of other caches and invalidate their data if needed. False sharing occurs when different threads are accessing the same memory and invalidating each other's caches. A fourth cache miss is coherence miss, caused by traffic w/ other processes.

MSI is a simple cache coherence protocol. Each cache block is either: Modified: cache entry has been written to and is inconsistent with memory; Shared: block is unmodified; or Invalid: not in cache or has been invalidated by other cache.

When a read request occurs:
If block is 'M' or 'S', supply data. Otherwise, if block is in M state in any other cache, it must write to memory and go to 'S' state. The cache then obtains the data from cache in 'S' state. The cache block turns to 'S' state.

When write request occurs:
If M state, modify data locally. If S state, invalidate all other caches who have the entry in 'S' state, and proceed to M state. Else I state, invalidate all other caches who have the entry in M or S state. Then load from memory, do write, go to 'M' state.

Warehouse Scale Computing

Power Usage Effectiveness (PUE) = Total Building Power / IT Equipment Power. A perfect PUE is 1.0. In reality, cooling, lighting, air movement raise it.

WSC exploits request level parallelism, in that requests are largely independent and read-only.

MapReduce: Programming model for processing large datasets. Provided a map function and reduce function, it parallelizes computation across clusters of machines.
```
map(in_key, in_val):                 reduce(interm_key, list(interm_val)):
  // DO work                           // DO work
  emit(interm_key, interm_val)         emit(out_key, out_val)
```

Apache Hadoop and Apache Spark are MapReduce frameworks. The Hadoop Distributed File System (HDFS) provides a robust distributed file system to handle failures.

WordCount in Spark looks like:
```
file.flatMap(lambda line: line.split())  // func: return a sequence
    .map(lambda word: (word, 1))         // func: return new data list
    .reduceByKey(lambda a, b: a+b)       // func: (v,v) -> v
```

GPA calculator in Spark: # students = list((studentName, courseData))
```
studentsData = sc.parallelize(students)
out = studentsData.map(lambda (k,v): (k, (v.grade, 1)))
    .reduceByKey(lambda v1,v2: (v1[0]+v2[0], v1[1]+v2[1]))
    .map(lambda (k,v): (k, v[0]/v[1]))
```

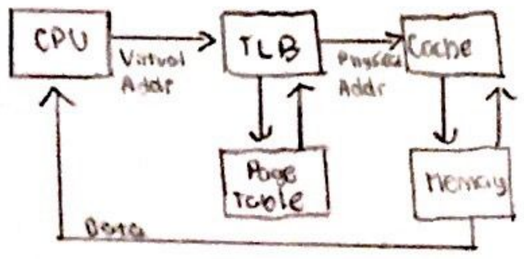
Example of Data Level Parallelism with Intel SSE:
```
static int product(int n, int *a) {   // returns product of entries of array
  int result[4];
  __m128i prod = _mm_load_si128();    // returns 128 bit wide vector
  for (int i=0; i<n/4*4; i+=4) {
      prod = _mm_mul_ps(prod, _mm_loadu_si128((__m128i*) a[i]));
  }                                                  ^ returns vector at a[i]
  _mm_storeu_si128((__m128i*) result, prod);
  for (int i = n/4*4; i<n; i++) {     // tail case
      result[0] *= a[i];
  }
  return result[0]*result[1]*result[2]*result[3];
}
```

Virtual Memory

Henceforth 'memory' means DRAM, in contrast to disk mem.

Virtual memory is the bridge between memory and disk in the memory hierarchy. It simulates a full address space for each process, and enforces protection between processes. By swapping from disk memory, VM also allows programs to address more memory than is available in main memory.

Logical Flow:



Memory is divided into pages.

A page table maps virtual addresses to physical addresses of the base of each page. The translation lookaside buffer (TLB) is a fully-associative cache to store page table data.

Virtual Address: (what program uses)

Virtual Page Number	Page Offset

$\log_2$(page size) bits wide

Physical Address:

Physical Page Number	Page offset

- TLB Reach = # of entries × size of page (in bytes)

Page Table

Index (Virtual Page Number)	Page Valid	Page Dirty	Permission Bits	Physical Page Number
0				
1				
2				
⋮				

sometimes we include LRU bits, $\log_2$(# of entries in TLB)

TLB

Assuming fully-associative with one block being a single page table entry

Valid	Tag = Virtual Page Number	Page Table Entry
		(page valid, dirty, etc...) ...

- The paging supervisor creates and manages page tables for each process. If hardware raises a page fault exception (i.e. page is not in memory), it is brought in from disk, and the page table is updated to reflect physical location of the virtual address. If main memory is full, we must swap out a page to disk, mark the corresponding page table entry as invalid, and then proceed as usual.

- A processor that supports virtual memory must have a page table base register. An OS loads the address of the start of that processes's page table into the PTBR whenever a new process is dispatched (called a context switch). This requires invalidating the TLB entries.

Memory Access Steps: (if access is write, assume we set dirty bit)

1) Obtain VPN and offset from virtual address field
2) Search TLB for VPN

If matching entry and valid: Hit! If protection check passes, return physical page number. Otherwise, protection fault.

If no matching entry: TLB miss!

Do page table walk to row index of VPN. If valid, return physical page number. Update the TLB appropriately. If invalid, page fault; bring in page from disk to next available memory frame, or evict a frame to disk. Then update page table w/ new physical page number.

3) Obtain actual address using PPN with the page offset.

I/O

Polling: Checking a ready bit regularly. It does not require additional hardware and has low overhead. Good for mouse and keyboard. Infeasible for devices with fast transfer rate that are rarely ready (e.g. ethernet card)

Interrupts: Hardware fires exception when ready. CPU changes RPC to execute code in interrupt handler. High overhead but necessary for fast devices (e.g. network card, hard drive)

Data Transfer:

Programmed I/O (PIO): CPU uses instructions that access I/O address space to perform data transfer in memory-mapped I/O.

Direct Memory Access (DMA): The I/O device reads/writes to memory directly while CPU can do other things. High overhead.

Networking

The traditional network is shared, in which all hosts compete for bandwidth. In contrast, a switch network has direct, point-to-point connections between hosts. Most wired connections are switched; wireless shared.

A protocol defines the format of messages sent and received, the order of messages sent and received, and the actions to take upon sending or receiving a message. A protocol family is a set of cooperating protocols that implement the network stack.

Transmission Control Protocol/Internet Protocol (TCP/IP) family is the basis of the internet. IP makes best effort to deliver, but packets can be lost. TCP guarantees reliable, in-order delivery.

TCP and UDP are contrasting internet transfer protocols. TCP is reliable, a two-way relationship good for webpages. UDP is suitable for applications that need fast transmission (video games)

Dependability

Mean Time to Failure (MTTF) Mean Time to Repair (MTTR)

Mean Time Between Failures (MTBF): MTTF + MTTR

Availability: MTTF/(MTTF + MTTR)

Disk failures often follow a 'bathtub' curve: decreases from break-in period, then increases from wear-out.

Hamming Error-Correcting Codes

Each data word maps to a unique code word, formed by inserting parity bits into data word. Define the distance as minimum # of bit positions in which two valid codewords differ. Hamming ECC detects up to 2 bit errors and can correct 1 bit errors.

Suppose we have $2^r - r - 1$ bits. We need r parity bits to form the code word.

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Encoded data		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16
Parity bit coverage	p1	X		X		X		X		X		X		X		X	
	p2		X	X			X	X			X	X			X	X	...
	p4				X	X	X	X					X	X	X	X	
	p8								X	X	X	X	X	X	X	X	
	p16																X

To check for errors: Check all parity bits. If all correct, no error. Otherwise, sum of erroneous bits identifies the erroneous bit position. For example, if parity bits in 1,2,8 indicate error, then bit 1+2+8 = 11 is error, so flip it. If only parity bit is error, then parity bit itself is in error.

RAID (Redundant Arrays of Inexpensive Disks)

Raid 0 - Striping (not used): Data is spread out ("striped") over disks. Good for non-critical storage of data needed to be accessed at high speed.

Raid 1 - Mirroring: Extra copy of each disk. Good availability, read speed, but expensive. Good for mission-critical storage.

Raid 3 - Parity (not used): Byte-level striping with a dedicated parity disk. (that is, a disk containing sum of disks at a stripe (mod 2)).

Raid 4 - Parity (not used): Block-level striping with dedicated parity disk. Not good for small writes.

Raid 5 - Distributed Parity: Recognizing that parity disk is bottleneck on write operations, we distribute parity information over each disk with block level striping. At least 3 disks needed. We can recover from any single disk failure. But write overhead still higher than Raid 1.