Topics Before Midterm

- C & Malloc
- Memory Management
- MIPS
- Number Representation
- Floating Point
- CAL
Topics Since Midterm

• Digital Logic
  – Verilog
  – State Machines
• CPU Design
• Pipelining
• Caches
• Virtual Memory
• I/O and Performance
Today’s Focus

• Focus on material from after the midterm. Don’t forget to go back and look over the old review session and your midterm
• More emphasis on material that we covered just after the midterm, less on stuff from the past few weeks.
  – Lots of stuff on digital logic
  – Like one slide on performance
• My intent is to help you “page in” (no replacement policy, please) material from the second half of the course that you may have long since forgotten
Topics Since Midterm

- Digital Logic
  - Verilog
  - State Machines
- CPU Design
- Pipelining
- Caches
- Virtual Memory
- I/O and Performance
Digital Logic (and such)

- Truth Tables
- Boolean Algebra
- Canonical SOP
- Combinational Logic
- State Machines
- Timing Diagrams & Tables
- Programmable Logic Arrays
Digital Logic – Truth Tables

- A table describing the output of a function for every possible input
- Usually done bitwise when dealing with digital logic
- $2^n$ entries (each input can have two possible states)
Digital Logic – Boolean Operators

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
<th>AND</th>
<th>OR</th>
<th>NAND</th>
<th>NOR</th>
<th>XOR</th>
<th>XNOR</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ AB \quad A+B \quad (AB)' \quad (A+B)' \quad A \oplus B \quad A \equiv B \quad A' \]
Digital Logic – Boolean Algebra

• An algebraic expression using Boolean operators
  – e.g. \( AB + A'B + A(B+C') \)

• Similar laws to normal algebra
  – Distributive: \( AB + A'B + AB + AC' \)
  – Idempotent: \( AB + A'B + AC' \)
  – Complementarity: \( B + AC' \)
Minimize the following using Boolean Algebra simplification rules:

\[ s = ((a + ab) + b) + (a + b) \]

What does this function do?
Simplifies to:

\[ s = \overline{ab} + \overline{ab} \]

This is an XOR
Digital Logic – Canonical SOP Form

• Standardized form to describe a truth table uniquely
• For every 1 in the output column of the Truth Table have a term in the SOP
• Put every input variable in every term, with the ones that were 0 for that entry in the TT negated
Digital Logic – Combinational Logic

• Built from Boolean Algebra operators turned into gates
• Output is purely a function of current input
• Therefore, cannot have memory or remember state
• However, this is perfect for describing a truth table as we have seen it so far!
Digital Logic – Sequential Logic & State

• To implement many practical circuits we need some form of “memory”.
• Registers (created from Flip Flops) are our statefull circuit elements.
• Adding state to a circuit introduces a notion of time centered around a clock
  – Complicates Truth Table
  – Necessitates Timing Diagrams
Digital Logic – Timing

Input data must be stable in this period.

"setup" time

"hold" time

"clk-to-q" delay
Digital Logic – Finite State Machines

- An abstraction of any system with a finite number of states and logical transitions between them
- Useful when trying to come up with the truth table for a problem
- States in a FSM are represented by data stored in registers
Digital Logic – Putting It All Together

Exercise

Come up with the FSM, Truth Table (NS/Output), Canonical SOP form, Simplified Boolean Equation, and circuit diagram for this function over a continuous bit stream:

Output 1 if the input was a 0. Output 0 if the input was a 1. Unless it was the 3\textsuperscript{rd} 1 in a row, in which case you output 1 (and start over counting 1s.
FSM Solution – Created At Review
Solution

- \( \text{MSB(NS)} = S_1' \times S_0 \times n \)
- \( \text{LSB(NS)} = S_1' \times S_0' \times n \)
- \( \text{OUT} = S_1' \times n' + S_1 \times S_0' \)
Digital Logic – Verilog

- Hardware Description Language
- Verilog description is a collection of interconnected modules
- All modules work in parallel
- Structural vs. Behavioral
Digital Logic – Verilog

Exercise

Implement the following circuit in both structural and behavioral Verilog:
Digital Logic – Verilog

Behavioral

module pri_enc(in0, in1, in2, e0, e1);
    input in0, in1, in2;
    output e0, e1;
    assign e0 = (~in1 & in0) | in2;
    assign e1 = in1 | in2;
endmodule;
module pri_enc(in0, in1, in2, e0, e1);
    input in0, in1, in2;
    output e0, e1;
    wire notIn1, and01;
    not(in1, notIn1);
    and(notIn1, in0, and01);
    or(and01, in2, e0);
    or(in1, in2, e1);
endmodule;
Digital Logic – Programmable Logic Arrays

- Creating customized hardware is expensive
- We would like to be able to pre-fabricate a circuit and then allow it to be programmed by the developer
- PLAs are the answer!
- Review how to program one on your own
Topics Since Midterm

• Digital Logic
  – Verilog
  – State Machines

• CPU Design
• Pipelining
• Caches
• Virtual Memory
• I/O and Performance
Single Cycle CPU Design

• Overview Picture
• Two Major Issues
  – Datapath
  – Control
• Control is the hard part, but is make easier by the format of MIPS instructions
Single-Cycle CPU Design

**Datapath**

**Instruction Memory**

- Instruction Address
- Next Address
- Clk

**32 32-bit Registers**

- Rw
- Ra
- Rb

**Control Signals**

- Control
- Conditions

**Datapath**

- Data Address
- Data In
- Clk

**Ideal Data Memory**

- Data Out

**Ideal Instruction Memory**

- Instruction
- Rd
- Rs
- Rt

**ALU**

- A
- B
- 32 32-bit ALU

**Next Address**

- Clk

**Ideal Data Memory**

- Data In
- Clk
CPU Design – Steps to Design/Understand a CPU

• 1. Analyze instruction set architecture (ISA) => datapath requirements
• 2. Select set of datapath components and establish clocking methodology
• 3. Assemble datapath meeting requirements
• 4. Analyze implementation of each instruction to determine setting of control points.
• 5. Assemble the control logic
Putting it All Together: A Single Cycle Data
CPU Design – Components of the Datapath

- Memory (MEM)
  - instructions & data
- Registers (R: 32 x 32)
  - read RS
  - read RT
  - Write RT or RD
- PC
- Extender (sign extend)
- ALU (Add and Sub register or extended immediate)
- Add 4 or extended immediate to PC
CPU Design – Control Signals

- **Branch**: 1 for branch, 0 for other
- **ALU control**
  - **MemWrite, MemRead(=MemtoReg)**: 1 if writing to/reading from memory, 0 if not
  - **ALUSrc**: choice of ALU input; 1 for immed, 0 for reg
  - **RegWrite**: 1 if writing a reg, 0 if not
  - **RegDst**: 1 if output reg is specified in bits 15-11 (R-fmt), 0 if output reg is in bits 20-16 (I-fmt)
  - **MemtoReg**: 1 if writing reg from memory, 0 if writing reg from ALU
  - **PCSrc**: 1 for branch address, 0 for PC+4
CPU Design – Instruction Implementation

- **Instructions supported (for our sample processor):**
  - lw, sw
  - beq
  - R-format (add, sub, and, or, slt)
  - corresponding I-format (addi …)

- **You should be able to,**
  - given instructions, write control signals
  - given control signals, write corresponding instructions in MIPS assembly
What Does An ADD Look Like?
Add

\[ R[rd] = R[rs] + R[rt] \]

- **Rd** = R[rs] + R[rt]
- **RegDst = 1**
- **RegWr = 1**
- **ALUctr = Add**
- **MemtoReg = 0**
- **Instruction<31:0>**
- **Rt**
- **Rs**
- **Rd**
- **Imm16**
- **AluSrc = 0**
- **MemWr = 0**
- **Zero**
- **WrEn**
- **Adr**
- **Data Memory**
- **imm16**
- **16**
- **32**

32 32-bit Registers

Instruction Fetch Unit

ALU

Data In

Extender

Mux

MemtoReg = 0

Clk
How About ADDI?
R[rt] = R[rs] + SignExt[Imm16]
One More: lw

Inst Memory Addr

Instruction<31:0>

Rs Rs Rs Rs Rs Rs Rs Rs Rs Rs
Rt Rt Rt Rt Rt Rt Rt Rt Rt Rt
Rd Rd Rd Rd Rd Rd Rd Rd Rd Rd
Imm16 Imm16 Imm16 Imm16 Imm16 Imm16 Imm16 Imm16 Imm16 Imm16

nPC_sel RegDst

Equal ALUctr MemWr MemtoReg

RegWr

Rw Ra Rb
32 32-bit Registers

busA busA busA busA busA busA busA busA busA busA
busW busW busW busW busW busW busW busW busW busW
Clk Clk Clk Clk Clk Clk Clk Clk Clk Clk

imm16 16 imm16 16 imm16 16 imm16 16 imm16 16

Adder Adder Adder Adder Adder Adder Adder Adder Adder Adder

Mux Mux Mux Mux Mux Mux Mux Mux Mux Mux

PC Ext PC Ext PC Ext PC Ext PC Ext PC Ext PC Ext PC Ext PC Ext PC Ext

Data In Data In Data In Data In Data In Data In Data In Data In Data In Data In

WrEnAddr WrEnAddr WrEnAddr WrEnAddr WrEnAddr WrEnAddr WrEnAddr WrEnAddr WrEnAddr WrEnAddr

Data Memory Data Memory Data Memory Data Memory Data Memory Data Memory Data Memory Data Memory Data Memory

- $R[rt] = \text{Data Memory } \{R[rs] + \text{SignExt}[imm16]\}$

---

**Instruction Fetch Unit**

- **op**
- **rs**
- **rt**
- **immediate**

**ALU**

- **ALUctr = Add**
- **ALUSrc = 1**
- **Zero**
- **MemWr = 0**

**Memory**

- **MemtoReg = 1**

**Data Memory**

- **WrEn**
- **Adr**

**Extender**

- **imm16**

**Mux**

- **busW**
- **busA**
- **busB**

**Registers**

- **Rw**, **Ra**, **Rb**
- **32 32-bit Registers**

**Clk**

- **RegDst = 0**
- **RegWr = 1**

---

**Instruction<31:0>**

- $<0:15>$
- $<11:15>$
- $<16:20>$
- $<20:25>$
- $<25:31>$

---

$R[rt] = Data\ Memory \ \{R[rs] + \text{SignExt}[imm16]\}$
<table>
<thead>
<tr>
<th>inst</th>
<th>Register Transfer</th>
<th>ALU src, ALU Ctr, Reg Dst, Reg Wr, nPC Sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>R[rd] &lt;- R[rs] + R[rt]; PC &lt;- PC + 4</td>
<td>ALU src = RegB, ALU Ctr = “add”, Reg Dst = rd, Reg Wr, nPC Sel = “+4”</td>
</tr>
<tr>
<td>SUB</td>
<td>R[rd] &lt;- R[rs] - R[rt]; PC &lt;- PC + 4</td>
<td>ALU src = RegB, ALU Ctr = “sub”, Reg Dst = rd, Reg Wr, nPC Sel = “+4”</td>
</tr>
<tr>
<td>ORi</td>
<td>R[rt] &lt;- R[rs] + zero_ext(Imm16); PC &lt;- PC + 4</td>
<td>ALU src = Im, Ext op = “Z”, ALU Ctr = “or”, Reg Dst = rt, Reg Wr, nPC Sel = “+4”</td>
</tr>
<tr>
<td>LOAD</td>
<td>R[rt] &lt;- MEM[ R[rs] + sign_ext(Imm16)]; PC &lt;- PC + 4</td>
<td>ALU src = Im, Ext op = “Sn”, ALU Ctr = “add”, Mem to Reg, Reg Dst = rt, Reg Wr, nPC Sel = “+4”</td>
</tr>
<tr>
<td>STORE</td>
<td>MEM[ R[rs] + sign_ext(Imm16)] &lt;- R[rs]; PC &lt;- PC + 4</td>
<td>ALU src = Im, Ext op = “Sn”, ALU Ctr = “add”, Mem Wr, nPC Sel = “+4”</td>
</tr>
<tr>
<td>BEQ</td>
<td>if ( R[rs] == R[rt] ) then PC &lt;- PC + sign_ext(Imm16)</td>
<td></td>
</tr>
</tbody>
</table>
## A Summary of the Control Signals (2/2)

### Table

<table>
<thead>
<tr>
<th>func</th>
<th>10 0000</th>
<th>10 0010</th>
<th>We Don’t Care :-)</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>00 0000</td>
<td>00 0000</td>
<td>00 1101</td>
</tr>
<tr>
<td>add</td>
<td>10 0011</td>
<td>10 0111</td>
<td>00 0100</td>
</tr>
<tr>
<td>sub</td>
<td>00 0010</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>ori</td>
<td></td>
<td>10 1011</td>
<td>x</td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>jump</td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

### Columns
- **RegDst**: Destination register
- **ALUSrc**: ALU source
- **MemtoReg**: Memory to register
- **RegWrite**: Register write
- **MemWrite**: Memory write
- **nPCEsel**: Next PC select
- **Jump**: Jump
- **ExtOp**: External operation
- **ALUctr<2:0>**: ALU control

### Binary Operations
- **Add**: 0010000
- **Subtract**: 1010000
- **Or**: 1110001
- **And**: 1100000
- **Xor**: 0100001
- **Shift Left**: 1010100
- **Shift Right**: 0110100
- **Multiply**: 1101000
- **Divide**: 0101000

### Appendix A
- See Appendix A
- We Don’t Care :-)

### Instruction Formats
- **R-type**: `op rs rt rd shamt funct` (add, sub)
- **I-type**: `op rs rt immediate` (ori, lw, sw, beq)
- **J-type**: `op target address` (jump)
Control

Instruction<31:0>

Inst Memory

Adr

Op  Fun  Rt  Rs  Rd  Imm16

PCSsrc  RegWr  RegDst  ALUSrc  ALUctr  MemWr  MemtoReg  Zero

DATA PATH
CPU Design – Designing the Control

- ISA design
  - Instruction formats
  - opcode/funct assignment
- Truth table
- Logic expressions of control signals
- Simplified logic expressions

- Similar instructions producing similar control signals should have similar opcode/funct code
  - $\text{Add}(\text{funct}=32)$ / $\text{Addu}(\text{funct}=33)$
BREAK!
Topics Since Midterm

- Digital Logic
  - Verilog
  - State Machines

- CPU Design
- Pipelining
- Caches
- Virtual Memory
- I/O and Performance
Pipelining

- View the processing of an instruction as a sequence of potentially independent steps
- Use this separation of stages to optimize your CPU by starting to process the next instruction while still working on the previous one
- In the real world, you have to deal with some interference between instructions
Review Datapath

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Write Back
• Sequential laundry takes 8 hours for 4 loads
Pipelined Laundry

- Pipelined laundry takes 3.5 hours for 4 loads!
Pipelining -- Key Points

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- **Multiple** tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Time to “fill” pipeline and time to “drain” it reduces speedup
Pipelining -- Limitations

• Pipeline rate limited by slowest pipeline stage
• Unbalanced lengths of pipe stages also reduces speedup
• Interference between instructions – called a hazard
Pipelining -- Hazards

- **Hazards** prevent next instruction from executing during its designated clock cycle
  - **Structural hazards**: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - **Control hazards**: Pipelining of branches & other instructions stall the pipeline until the hazard; “bubbles” in the pipeline
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)
Pipelining – Structural Hazards

• Avoid memory hazards by having two L1 caches – one for data and one for instructions

• Avoid register conflicts by always writing in the first half of the clock cycle and reading in the second half
  – This is ok because registers are much faster than the critical path
Pipelining – Control Hazards

- Occurs on a branch or jump instruction
- Optimally you would always branch when needed, never execute instructions you shouldn’t have, and always have a full pipeline
  - This generally isn’t possible
- Do the best we can
  - Optimize to 1 problem instruction
  - Stall
  - Branch Delay Slot
Pipelining – Data Hazards

• Occur when one instruction is dependant on the results of an earlier instruction

• Can be solved by forwarding for all cases except a load immediately followed by a dependant instruction
  – In this case we detect the problem and stall (for lack of a better plan)
Pipelining -- Exercise

Assuming one instruction executed per clock cycle, delayed branch, forwarding, interlock on load hazards, and a full pipeline: how many cycles will this code take? Where is there forwarding?

Loop:  lw, $t0, 0($s1)
      addu, $t0, $t0, $s2
      sw, $t0, 0($s1)
      addiu, $s1, $s1, -4
      bne, $s1, $zero, Loop
      addu $s3, $t1, $t1
Topics Since Midterm

- Digital Logic
  - Verilog
  - State Machines
- CPU Design
- Pipelining
- Caches
- Virtual Memory
- I/O and Performance
Caches

• The Problem: Memory is slow compared to the CPU
• The Solution: Create a fast layer in between the CPU and memory that holds a subset of what is stored in memory
• We call this creature a Cache
Caches – Reference Patterns

• Holding an arbitrary subset of memory in a faster layer should not provide any performance increase
  – Therefore we must carefully choose what to put there
• Temporal Locality: When a piece of data is referenced once it is likely to be referenced again soon
• Spatial Locality: When a piece of data is referenced it is likely that the data near it in the address space will be referenced soon
Caches – Format & Mapping

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>(msb)ttttttttttttttttttttttt</td>
<td>iiiiiiiiiiiii</td>
<td>oooooo(lsb)</td>
</tr>
</tbody>
</table>

- **Tag**: Unique identifier for each block in memory that maps to same index in cache
- **Index**: Which “row” in the cache the data block will map to (for direct mapped cache each row is a single cache block)
- **Block Offset**: Byte offset within the block for the particular word or byte you want to access
Caches – Direct Mapped

- Cache Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ...
  - 4 blocks => any memory location that is multiple of 4

- Cache Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ...
  - 4 blocks => any memory location that is multiple of 4
Here's a simple 2-way set associative cache.
Caches -- Block Size Tradeoff

- Miss Penalty
  - Block Size

- Miss Rate
  - Exploits Spatial Locality
  - Fewer blocks: compromises temporal locality

- Average Access Time
  - Increased Miss Penalty & Miss Rate

- Block Size
Caches – Exercise 1

How many bits would be required to implement the following cache?

Size: 1MB  
Associativity: 8-way set associative  
Write policy: Write back  
Block size: 32 bytes  
Replacement policy: Clock LRU (requires one bit per data block)
Caches – Solution 1

Number of blocks = 1MB / 32 bytes = 32 Kblocks ($2^{15}$)

Number of sets = 32Kblocks / 8 ways = 4 Ksets ($2^{12}$) ➔ 12 bits of index

Bits per set = 8 * (8 * 32 + (32 – 12 – 5) + 1 + 1 + 1) ➔ 32bytes + tag bits + valid + LRU + dirty

Bits total = (Bits per set) * (# Sets) = $2^{12} * 2192$ = 8,978,432 bits
Caches – Exercise 2

Given the following cache and access pattern, classify each access as hit, compulsory miss, conflict miss, or capacity miss:

Cache:
Word addressed
2 words/block
8 blocks
2-way set associative
LRU replacement

Access Pattern (word addresses):
3, 10, 15, 0, 5, 1, 9, 4, 10, 16, 0, 3
Caches – Solution 2

3 (comp, s1, w0), 10 (comp, s1, w1),
15 (comp, s3, w0), 0 (comp, s0, w0),
5 (comp, s2, w0), 1 (hit),
9 (comp, s0, w1), 4 (hit),
10 (hit), 16 (conf, s0, w0),
0 (conf, s0, w1), 3 (hit)
Topics Since Midterm

• Digital Logic
  – Verilog
  – State Machines

• CPU Design

• Pipelining

• Caches

• Virtual Memory

• I/O and Performance
Virtual Memory

- Caching works well – why not extend the basic concept to another level?
- We can make the CPU think it has a much larger memory than it actually does by “swapping” things in and out to disk
- While we’re doing this, we might as well go ahead and separate the physical address space and the virtual address space for protection and isolation
VM – Virtual Address

- Address space broken into fixed-sized pages
- Two fields in a virtual address
  - VPN
  - Offset
- Size of offset = $\log_2(\text{size of page})$
VM – Address Translation

- VPN used to index into page table and get Page Table Entry (PTE)
- PTE is located by indexing off of the Page Table Base Register, which is changed on context switches
- PTE contains valid bit, Physical Page Number (PPN), access rights
VM – Translation Look-Aside Buffer (TLB)

- VM provides a lot of nice features, but requires several memory accesses for its indirection – this really kills performance
- The solution? Another level of indirection: the TLB
- Very small fully associative cache containing the most recently used mappings from VPN to PPN
VM – Exercise

Given a processor with the following parameters, how many bytes would be required to hold the entire page table in memory?

- Addresses: 32-bits
- Page Size: 4KB
- Access modes: RO, RW
- Write policy: Write back
- Replacement: Clock LRU (needs 1-bit)
VM – Solution

**Number of bits page offset = 12 bits**

**Number of bits VPN/PPN = 20 bits**

**Number of pages in address space**

= 232 bytes/212 bytes = 220 = 1M pages

**Size of PTE = 20 + 1 + 1 + 1 = 23 bits**

**Size of PT = 220 * 23 bits = 24,117,248 bits**

= 3,014,656 bytes
The Big Picture
CPU – TLB – Cache – Memory – VM

Processor → VA
   ^miss
   |       TLB Lookup
   |       miss
   |       Translation
   |       hit
   |       data
   v
Cache ← Main Memory
   → miss

Big Picture – Exercise

What happens in the following cases?

- TLB Miss
- Page Table Entry Valid Bit 0
- Page Table Entry Valid Bit 1
- TLB Hit
- Cache Miss
- Cache Hit
Big Picture – Solution

• TLB Miss – Go to the page table to fill in the TLB. Retry instruction.
• Page Table Entry Valid Bit 0 / Miss – Page not in memory. Fetch from backing store. (Page Fault)
• Page Table Entry Valid Bit 1 / Hit – Page in memory. Fill in TLB. Retry instruction.
• TLB Hit – Use PPN to check cache.
• Cache Miss – Use PPN to retrieve data from main memory. Fill in cache. Retry instruction.
• Cache Hit – Data successfully retrieved.

• Important thing to note: Data is always retrieved from the cache.
Topics Since Midterm

• Digital Logic
  – Verilog
  – State Machines

• CPU Design

• Pipelining

• Caches

• Virtual Memory

• I/O and Performance
IO and Performance

- IO Devices
- Polling
- Interrupts
- Networks
## I/O Device Examples and Speeds

- **I/O Speed**: bytes transferred per second
  - (from mouse to Gigabit LAN: 100-million-to-1)

<table>
<thead>
<tr>
<th>Device</th>
<th>Behavior</th>
<th>Partner</th>
<th>Data Rate (KBytes/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>Input</td>
<td>Human</td>
<td>0.01</td>
</tr>
<tr>
<td>Mouse</td>
<td>Input</td>
<td>Human</td>
<td>0.02</td>
</tr>
<tr>
<td>Voice output</td>
<td>Output</td>
<td>Human</td>
<td>5.00</td>
</tr>
<tr>
<td>Floppy disk</td>
<td>Storage</td>
<td>Machine</td>
<td>50.00</td>
</tr>
<tr>
<td>Laser Printer</td>
<td>Output</td>
<td>Human</td>
<td>100.00</td>
</tr>
<tr>
<td>Magnetic Disk</td>
<td>Storage</td>
<td>Machine</td>
<td>10,000.00</td>
</tr>
<tr>
<td>Wireless Network</td>
<td>I or O</td>
<td>Machine</td>
<td>10,000.00</td>
</tr>
<tr>
<td>Graphics Display</td>
<td>Output</td>
<td>Human</td>
<td>30,000.00</td>
</tr>
<tr>
<td>Wired LAN Network</td>
<td>I or O</td>
<td>Machine</td>
<td>1,000,000.00</td>
</tr>
</tbody>
</table>

*When discussing transfer rates, use $10^x$*
IO – Problems Created by Device Speeds

- CPU runs far faster than even the fastest of IO devices
- CPU runs many orders of magnitude faster than the slowest of currently used IO devices
- Solved by adhering to well defined conventions
  - Control Registers
IO – Device Communication

- Some processors handle IO via special instructions. This is called Programmed IO.
- MIPS (and many other platforms) use variations on Memory Mapped IO where writing to or reading from certain portions of the address space actually communicates with the IO device.
- SPIM fakes this communication by using 4 special device registers.
SPIM I/O Simulation

- SPIM simulates 1 I/O device: memory-mapped terminal (keyboard + display)
- Read from keyboard (**receiver**); 2 device regs
- Writes to terminal (**transmitter**); 2 device regs

### Receiver Control
- **0xffff0000**
  - Unused (00...00)

### Receiver Data
- **0xffff0004**
  - Unused (00...00)

### Transmitter Control
- **0xffff0008**
  - Unused (00...00)

### Transmitter Data
- **0xffff000c**
  - Unused

---

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffff0000</td>
<td>Receiver Control</td>
</tr>
<tr>
<td>0xffff0004</td>
<td>Receiver Data</td>
</tr>
<tr>
<td>0xffff0008</td>
<td>Transmitter Control</td>
</tr>
<tr>
<td>0xffff000c</td>
<td>Transmitter Data</td>
</tr>
</tbody>
</table>
IO – Polling

- CPU continuously checks the device’s control registers to see if it needs to take any action
- Extremely easy to program
- Extremely inefficient. The CPU spends potentially huge amounts of times polling IO devices.
IO – Interrupts

• Asynchronous notification to the CPU that there is something to be dealt with for an IO device

• Not associated with any particular instruction – this implies that the interrupt must carry some information with it

• Causes the processor to stop what it was doing and execute some other code
IO – Types of Interrupts

- **Exception**: A signal marking that something strange has happened. This is usually the direct result of an instruction (i.e. an overflow exception)

- **Interrupt**: An asynchronous signal that some wants the attention of the CPU, interrupts the CPU in the middle of what it is doing (i.e. a device interrupt)

- **Trap**: A synchronous exception that is explicitly called for by the programmer to force the processor to do something (i.e. a kernel trap)
IO – Interrupts for Device IO

- Device raises a flag in the processor – usually called an interrupt line
  - When this line is asserted the processor jumps to a specific location
- This location is an entry point to the Interrupt Service Routine (ISR)
- ISR uses the information that is available about which device raised the interrupt flag to jump to the proper device routine
- Device routine deals with whatever caused the IO Device to generate the interrupt
Shared vs. Switched Based Networks

- **Shared Media vs. Switched**: In switched, pairs (“point-to-point” connections) communicate at the same time; shared 1 at a time.

- **Aggregate bandwidth (BW)** in switched network is many times shared:
  - point-to-point faster since no arbitration, simpler interface.

![Diagram showing shared and crossbar switch networks](image)
Networks – Props To Kansas City: Sprint Headquarters
Networks – Protocols

- A protocol establishes a logical format and API for communication
- Actual work is done by a layer beneath the protocol, so as to protect the abstraction
- Allows for encapsulation – carry higher level information within lower level “envelope”
- Fragmentation – packets can be broken in to smaller units and later reassembled
Networks – TCP in 20 seconds

• TCP guarantees in-order delivery of complete packets
• Accomplishes this by keeping a lot of extra data around in case it needs to be resent
• Sender holds on to old data and retries the transmission if it does not get an ACK back from receiver within a time limit
• TCP Packets look like this: (kinda)

```
Net ID  Net ID  Len   ACK INFO CMD/ Address /Data
Header  Payload  Trailer
```

Checksum
Networks – Alright, I Know You’re Curious…

TCP Packet Format

<table>
<thead>
<tr>
<th>source port number</th>
<th>destination port number</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence number</td>
<td></td>
</tr>
<tr>
<td>acknowledgement number</td>
<td></td>
</tr>
<tr>
<td>unused</td>
<td>r</td>
</tr>
<tr>
<td>head len</td>
<td>u</td>
</tr>
<tr>
<td>Internet checksum</td>
<td></td>
</tr>
<tr>
<td>ptr to urgent data</td>
<td></td>
</tr>
<tr>
<td>options</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td></td>
</tr>
</tbody>
</table>
Networks – Complications

- Packet headers eat into your total bandwidth
- Software overhead for transmission limits your effective bandwidth significantly
Networks – Exercise

What percentage of your total bandwidth is being used for protocol overhead in this example:

• Application sends 1MB of true data
• TCP has a segment size of 64KB and adds a 20B header to each packet
• IP adds a 20B header to each packet
• Ethernet breaks data into 1500B packets and adds 24B worth of header and trailer
Networks – Solution

1MB / 64K = 16 TCP Packets
16 TCP Packets = 16 IP Packets
64K/1500B = 44 Ethernet packets per TCP Packet
16 TCP Packets * 44 = 704 Ethernet packets
20B overhead per TCP packet + 20B overhead per IP packet + 24B overhead per Ethernet packet =
20B * 16 + 20B * 16 + 24B * 704 = 17,536B of overhead

We send a total of 1,066,112B of data. Of that, 1.64% is protocol overhead.
Magnetic Disks

- **Purpose:**
  - Long-term, nonvolatile, inexpensive storage for files
  - Large, inexpensive, slow level in the memory hierarchy (discuss later)
Disk Device Terminology

- Several platters, with information recorded magnetically on both surfaces (usually)
- Bits recorded in tracks, which in turn divided into sectors (e.g., 512 Bytes)
- Actuator moves head (end of arm) over track (“seek”), wait for sector rotate under head, then read or write
Disk Performance Model /Trends

• Capacity : + 100% / year (2X / 1.0 yrs)
  Over time, grown so fast that # of platters has reduced
  (some even use only 1 now!)

• Transfer rate (BW) : + 40%/yr (2X / 2 yrs)

• Rotation+Seek time : – 8%/yr (1/2 in 10 yrs)

• Areal Density
  • Bits recorded along a track: **Bits/Inch** (BPI)
  • # of tracks per surface: **Tracks/Inch** (TPI)
  • We care about bit density per unit area **Bits/Inch**$^2$
  • Called **Areal Density** = BPI x TPI

• MB/$$: > 100%/year (2X / 1.0 yrs)
  • Fewer chips + areal density
Disks – RAID

• Idea was to use small, relatively inexpensive disks in place of large, very expensive disks to reduce cost and increase Mean Time to Failure
• Some RAID models are more successful than others
• We are not going to go over them today, since we just learned this last week.
• That being said, make sure you know them!
Performance -- Metrics

- Best overall raw computational power
- Least Cost
- Best power / Cost
- Response Time?
- Throughput?

Benchmarks help quantify these ideas.
That Was A LOT of Slides!

• Stick around if you have any more questions
• See you at the final, this Tuesday 12/14 from 12:30 to 3:30 at 230 Hearst Gym (and if that isn’t confusing, I don’t know what is)
• Don’t forget to review stuff from before the midterm!
• STUDY!!!