

CS 141 Final Review

◆ Last Time

- Input/Output Interconnects
- Input/Output Structures (polling, interrupts)

◆ This Time

- Whirlwind Review of Course Material

◆ Reminders/Announcements

- Final Exam 3/20, 3-6pm, Peterson 110
- TA Review Session, Saturday 3/18? (ask the TA's, check in the newsgroup)

CS 141 Final Exam

- ◆ **3 hours**
- ◆ **Targetted a 2-hour exam, but this doesn't mean you shouldn't be able to “retrieve” information rapidly**
- ◆ **A mixture of short and long questions (as before)**
- ◆ **Calculators NOT ALLOWED**
- ◆ **Coverage: all course material**
 - approximately 33% on material covered by midterm
 - approximately 67% on material since the midterm

Performance

- ◆ **How to Measure it? (metrics, interchangeability of computing power)**
- ◆ **Metrics (MIPS, CPI, Mflops, exec time) and what's good/bad about them**
- ◆ **Relative performance**
- ◆ **How to Summarize performance (and how Not to)**

Assembly/Machine Language

- ◆ **Instruction Types**
- ◆ **Add class, load/store, control flow, procedure call (JAL, JR)**
- ◆ **Instruction formats: R, I, J-format and how they relate to instruction types**
- ◆ **Assembler directives: Labels, .text, .data ...**
- ◆ **Alignment (data sizes) and addressing (byte addressable)**
- ◆ **Basic control flow, and branching**

Procedure Call/Return

- ◆ Linkage (JAL, JR, \$RA)
- ◆ Labels -- to glue together
- ◆ Stacks - push, pop and optimized arithmetic
- ◆ Recursive procedures
- ◆ Calling/Register usage conventions
 - Argument Registers
 - Caller Saves, Callee Saves
 - Other regs
- ◆ Where and when state must be saved and restored to give “C” compatible interfaces
- ◆ Stack state, based on the call sequence

Basics of Instruction Execution

◆ Elements of Instruction Execution

- Instruction Fetch
- Instruction Decode
- Read Registers
- Execute operation in the ALU
- Write the registers
- Increment the Program counter
- Repeat the cycle

◆ **Computer: a machine designed to do this repeatedly.**

◆ **Differs from other machines because it is programmable -- many instruction seqs possible**

Basic Computer Implementation

◆ Single Cycle Implementation

- **Instruction Memory, PC, Register File, ALU, Data Memory**
- **Single set of control settings, data flows through the datapath, settles and end of clock period transitions to the next instruction**
- **R, I, and J class instructions**
- **Control logic is pure combinational (implementations)**
- **Computer = FSM with PC = state**
- **Advantages: simple control**
- **Disadvantages: slow (same clock period for all instructions), large hardware requirements (separate memories), low hardware utilization**

Basic Computer Implementation (cont)

◆ Multiple Cycle Implementation

- **Idea: Break execution into clock periods**
- **Several sets of control in sequence for the datapath**
- **Control = Finite State Machine**
- **Instruction: execution path through a sequence of states**
- **Advantages: hardware reuse (ALU, Memory), variable instruction execution times (less waste, still some)**
- **Disadvantages: More complex control, still low hardware utilization (a little better)**
- **=> Generally a necessity for Complex Instruction Sets (CISC's)**

Summary (through Midterm)

◆ Performance

- Metrics
- Precise performance comparisons and summaries

◆ Basics of Instruction sets

- Assembly Programming
- C / Assembly correspondence

◆ Basics of Instruction set implementation

- Datapath
- Control

More Aggressive Implementations

◆ Pipelining Concepts

- Pipelining space and time shares the hardware
- Divides the circuitry into units for separate operations
- Throughput = rate of operation completion / initiation
- Latency = time from initiation to completion of a single operation
- Pipelining INCREASES throughput
- Pipelining INCREASES latency

◆ How much benefit is possible?

- Maximum increase in throughput is proportional to number stages
- Latency generally increases due to latching and “roundup” to one clock period

Pipelining Instruction Execution

- ◆ **Basic Stages: IF, ID, EX, MEM, WR**
- ◆ **Same schedule for R and I class instructions**
- ◆ **Special schedule for J class instructions**
- ◆ **No dependences -> things are easy**
- ◆ **Control for Pipelining?**
 - **Add registers parallelling the datapath**
 - **Generate control as before**
 - **Shift it along with the data values, use it just in time**
 - **Only complication is with write-back to register file**
 - **PIPELINE the register write address along with the other control**

Complications of Pipelining

- ◆ **Why doesn't pipelining always give a benefit?**
- ◆ **Instructions with different schedules/latency**
 - Increased latency
 - Must pass through same functional units
- ◆ **Dependences or Hazards**
 - Instructions that share values: producer/consumer
 - Forwarding (minimizing the losses)
 - Data dependence delays EX stage, loss of performance
 - Control dependence delays IF stage, loss of performance

Advanced Pipelining

- ◆ **Instruction Level Parallelism as a General Instruction Throughput Technique**
- ◆ **Superscalar (hardware extracts dependences)**
- ◆ **Superpipelined (marketing term only)**
- ◆ **Very Long Instruction Word (VLIW), (compiler extracts dependences)**
- ◆ **Intel IA-64 (EPIC), a hybrid VLIW**

Memory Hierarchies

- ◆ **Memory sizes and speeds**
- ◆ **Larger => slower**
- ◆ **Locality -- structure in the reference patterns**
 - **Spatial: clustering of references in the address space**
 - **Temporal: clustering of accesses to same address in time**
- ◆ **Memory hierarchies exploit this to provide the illusion of a large fast memory (which is physically unrealizable).**
- ◆ **Caches: implicitly (automatically) managed memory hierarchies**
- ◆ **Registers, local memories: explicitly managed memory hierarchies**

Memory Hierarchies (cont.)

◆ Caches: Basic Idea

- Spatial locality in lines (blocks)
- Temporal locality in replacement policy

◆ Cache organization

- Find stuff fast, cost of flexibility in placement
- Hits times and miss penalties
- Direct mapped caches, 2-way, n-way associative caches
- Tagging and indexing issues, how they work
- Implications for tagging overhead, access time

◆ Average memory access time (AMAT)

- Definition and what it means, how to calculate it
- Hit and Miss rates, not ALWAYS a win, but mostly

Busses

- ◆ **Electrical structure**
- ◆ **Advantages: cost, interoperability**
- ◆ **Limitations: scalability, speed**
- ◆ **Arbitration -- why it's needed**
 - **Correct Data**
 - **Avoid blowing up the hardware!**
 - **Fairness and latency issues**
- ◆ **Daisy-chain arbitration**
 - **How it works...**
- ◆ **Multi-level bus structures: pros and cons**

Input/Output

- ◆ **Notion of Input/Output**
- ◆ **I/O devices, range of characteristics**
- ◆ **This is what makes computers interesting!
Interactive!**
- ◆ **System Structure: P-M bus, I/O bus, Bus adaptors**
- ◆ **Detecting I/O events *and Achieving I/O transfers***
- ◆ **Detection: Polling vs. Interrupts**
 - Polling simpler, steady cost, dependent on latency tolerable
 - Interrupts more efficient, better program modularity, only do work as needed

Input/Output

- ◆ **Increasing bottleneck in both sequential and parallel systems (I/O bottleneck!)**
- ◆ **Critical for many new applications of computers**
 - Voice (audio)
 - Images (multimedia)
 - Graphics
 - Virtual reality
 - 100's of Gigabytes, terabytes (10^{12}), and petabytes (10^{15}) of storage
- ◆ **Parallel disks, parallel tapes**
- ◆ **Video servers, WWW servers, Database servers**
- ◆ ***This is where the action is in computer applications!***

Summary

- ◆ **Software-hardware interface: Machine Language**
 - ◆ **Inside the processor: Implementation and performance issues**
 - ◆ **Outside: the Memory hierarchy and Input/Output**
- That's essentially the entire machine. The rest is software layering atop this structure!**