Lecture 19
Summary

Benjamin Lee
Electrical and Computer Engineering
Duke University

www.duke.edu/~bcl15
www.duke.edu/~bcl15/class/class_ece252fall11.html
7 December 2012

- Please submit project reports to Sakai by midnight

Final Exam

- Wednesday, Dec 12, 2-5pm
- Closed book, closed notes exam
- Cumulative, with emphasis on latter half.

- 6-7 Questions
- 1/3 on earlier material
- 2/3 on later material

- 1/3 extended example or design questions
- 2/3 short answer
Computer architecture defines HW/SW interface

Evaluate architectures quantitatively
Computer architecture is the design of abstraction layers, which allow efficient implementations of computational applications on available technologies.
Abstraction Layers

Domain of early computer architecture ('50s-'80s)

Application
Algorithm
Programming Language
Operating System/Virtual Machines
Instruction Set Architecture (ISA)
Microarchitecture
Gates/Register-Transfer Level (RTL)
Circuits
Devices
Physics

Domain of recent computer architecture (since '90s)
In-order Datapath (built, ECE152)

Chip Multiprocessors (understand, experiment ECE252)
Performance Factors

Latency = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}

Seconds / Cycle
- Technology and architecture
- Transistor scaling
- Processor microarchitecture

Cycles / Instruction (CPI)
- Architecture and systems
- Processor microarchitecture
- System balance (processor, memory, network, storage)

Instructions / Program
- Algorithm and applications
- Compiler transformations, optimizations
- Instruction set architecture
Power and Energy

Definitions

- Energy (Joules) = $a \times C \times V^2$
- Power (Watts) = $a \times C \times V^2 \times f$

Power Factors and Trends

- activity ($a$): function of application resource usage
- capacitance ($C$): function of design; scales with area
- voltage ($V$): constrained by leakage, which increases as $V$ falls
- frequency ($f$): varies with pipelining and transistor speeds
Datapath: CISC versus RISC

Complex Instruction Set Computing
- microprogramming
- motivated by technology (slow instruction fetch)

Reduced Instruction Set Computing
- hard-wired datapath
- motivated by technology (caches, fast memory)
- complex instructions rarely used
CISC Microprograms

instr fetch:

\[
\begin{align*}
&MA \leftarrow PC & \text{# fetch current instr} \\
&A \leftarrow PC & \text{# next PC calculation} \\
&IR \leftarrow \text{Memory} \\
&PC \leftarrow A + 4 \\
&\text{dispatch on Opcode} & \text{# start microcode}
\end{align*}
\]

ALU:

\[
\begin{align*}
&A \leftarrow \text{Reg[rs]} & \\
&B \leftarrow \text{Reg[rt]} & \\
&\text{Reg[rd]} \leftarrow \text{func}(A,B) \\
&\text{do instruction fetch}
\end{align*}
\]

ALUi:

\[
\begin{align*}
&A \leftarrow \text{Reg[rs]} & \\
&B \leftarrow \text{Imm} & \text{# sign extension} \\
&\text{Reg[rt]} \leftarrow \text{Opcode}(A,B) \\
&\text{do instruction fetch}
\end{align*}
\]
CISC Bus-Based MIPS Datapath

Microinstruction: register to register transfer (17 control signals)

MA ← PC means RegSel = PC; enReg=yes; ldMA= yes
B ← Reg[rt] means RegSel = rt; enReg=yes; ldB = yes
RISC Hard-wired MIPS Datapath
Visualizing the Pipeline
Hazards and Limits to Pipelining

Structural Hazards
- Hardware cannot support this combination of instructions.
- Solution: stall pipeline (interlocks)

Data Hazards
- Instruction depends on result of prior instruction still in pipeline
- Solution: forward data, stall pipeline

Control Hazards
- Instruction fetch depends on decision about control flow
- Example: compute branches early in pipeline, predict branches
**Tomasulo & Out-of-order**

**Out-of-order Execution**
- Dynamically schedule instructions
- Execute instructions when dependences resolved

**Tomasulo’s Algorithm**
- Queue instructions until operands ready (reservation stations, ROB)
- Rename to eliminate write hazards (rename table, physical registers)

**Precise Interrupts/Exceptions**
- Instructions execute/comcomplete out-of-order
- Instructions commit in-order via reorder buffer
- Check for exceptions when committing instruction
Memory

- **DRAM**: access dense array of slow memory with a command protocol
- **SRAM**: access smaller array of fast memory on processor die
- **Virtual Memory**: translate applications’ virtual addresses into physical addresses, providing better memory management and protection
-- Chip organized into 4-8 logical banks, which can be accessed in parallel
-- Access DRAM with activate, read/write, precharge commands
Caches

Caches exploit predictable patterns

Temporal Locality
Caches remember the contents of recently accessed locations

Spatial Locality
Caches fetch blocks of data nearby recently accessed locations
Placement Policy

Line Number

Memory

Set Number

Cache

Fully Associative anywhere

(2-way) Set Associative anywhere in set 0 \((12 \mod 4)\)

Direct Mapped only into block 4 \((12 \mod 8)\)

Line 12 can be placed
Direct-Mapped Cache

- Tag
- Index
- Line Offset

- $t$
- $k$
- $b$

- $V$, Tag, Data Line

- $2^k$ lines

- HIT

- Data Word or Byte
Average Memory Access Time

$$\text{AMAT} = [\text{Hit Time}] + [\text{Miss Prob.}] \times [\text{Miss Penalty}]$$
- Miss Penalty equals AMAT of next cache/memory/storage level.
- AMAT is recursively defined

To improve performance
- Reduce the hit time (e.g., smaller cache)
- Reduce the miss rate (e.g., larger cache)
- Reduce the miss penalty (e.g., optimize the next level)
Caches and Code

Restructuring code affects data access sequences
- Group data accesses together to improve spatial locality
- Re-order data accesses to improve temporal locality

Prevent data from entering the cache
- Useful for variables that are only accessed once
- Requires SW to communicate hints to HW.
- Example: “no-allocate” instruction hints

Kill data that will never be used again
- Streaming data provides spatial locality but not temporal locality
- If particular lines contain dead data, use them in replacement policy.
- Toward software-managed caches
Caches and Code

for(i=0; i < N; i++)
    a[i] = b[i] * c[i];

for(i=0; i < N; i++)
    d[i] = a[i] * c[i];

for(i=0; i < N; i++)
{
    a[i] = b[i] * c[i];
    d[i] = a[i] * c[i];
}

What type of locality does this improve?
Parallelism

Instruction-level Parallelism (ILP)
- multiple instructions in-flight
- hardware-scheduled: (1) pipelining, (2) out-of-order execution
- software-scheduled: (3) VLIW

Data-level Parallelism (DLP)
- multiple, identical operations on data arrays/streams
- (1) vector processors, (2) GPUs
- (3) single-instruction, multiple-data (SIMD) extensions

Thread-level Parallelism (TLP)
- multiple threads of control
- if a thread stalls, issue instructions from other threads
- (1) multi-threading, (2) multiprocessors
VLIW and ILP (SW-managed)

- Multiple operations packed into one instruction format
- Instruction format is fixed, each slot supports particular instruction type
- Constant operation latencies are specified (e.g., 1 cycle integer op)
- Software schedules operations into instruction format, guaranteeing
  (1) Parallelism within an instruction – no RAW checks between ops
  (2) No data use before ready – no data interlocks/stalls
Vectors and DLP

**SCALAR**
(1 operation)

\[ r1 + r2 = r3 \]

add \( r3, r1, r2 \)

**VECTOR**
(N operations)

\[ v1 + v2 = v3 \]

vadd.vv \( v3, v1, v2 \)
Multithreading and TLP

- Superscalar
- Fine-Grained
- Coarse-Grained
- Multiprocessing
- Simultaneous Multithreading

- Thread 1
- Thread 2
- Thread 3
- Thread 4
- Thread 5
- Idle slot

Time (processor cycle)
Shared-memory Multiprocessors

- Provide a shared-memory abstraction
- Enables familiar and efficient programmer interface

Memory System
Shared-memory Multiprocessors

- Provide a shared-memory abstraction
- Enables familiar and efficient programmer interface
Shared-memory Multiprocessors

- Provide a shared-memory abstraction
- Enables familiar and efficient programmer interface
Challenges in Shared Memory

Cache Coherence
- “Common Sense”
- \( P1-Read[X] \rightarrow P1-Write[X] \rightarrow P1-Read[X] \) \hspace{1cm} Read returns \( X \)
- \( P1-Write[X] \rightarrow P2-Read[X] \) \hspace{1cm} Read returns value written by \( P1 \)
- \( P1-Write[X] \rightarrow P2-Write[X] \) \hspace{1cm} Writes serialized
All \( P \)'s see writes in same order

Synchronization
- Atomic read/write operations

Memory Consistency
- What behavior should programmers expect from shared memory?
- Provide a formal definition of memory behavior to programmer
- Example: When will a written value be seen?
- Example: \( P1-Write[X] \ll<10\text{ps}> \) \( P2-Read[X] \). What happens?
Coherence Protocols

Implement protocol for every cache line.
Compare, contrast snoopy and directory protocols [[Stanford Dash]]
**Solution: Test-and-set instruction**

- Add single instruction for load-test-store (t&s R1, lock)
- Test-and-set atomically executes
  
  ```
  ld R1, lock;       # load previous lock value
  st 1, lock;       # store 1 to set/acquire
  ```

- If lock initially free (0), t&s acquires lock (sets to 1)
- If lock initially busy (1), t&s does not change it
- Instruction is un-interruptible/atomic by definition

```plaintext
Inst-0   t&s R1, lock   # atomically load, check, and set lock=1
Inst-1   bnez R1       # if previous value of R1 not 0, acquire unsuccessful
....
Inst-n   stw R1, 0     # atomically release lock
```
Sequential Consistency (SC)

Definition of Sequential Consistency
Formal definition of programmers’ expected view of memory

1. Each processor \( P \) sees its own loads/stores in program order.

2. Each processor \( P \) sees \( \neg P \) loads/stores in program order.

3. All processors see the same global load/store ordering. \( P \) and \( \neg P \) loads/stores may be interleaved into some order. But all processors see the same interleaving/ordering.

Definition of Multiprocessor Ordering [Lamport]
Multi-processor ordering corresponds to some sequential interleaving of uniprocessor orderings. Multiprocessor ordering should be indistinguishable from multi-programmed uniprocessor.
Computer architecture is HW/SW interface. Consider classes on both sides of this interface.
Looking Forward

Energy-efficiency
• Technology limitations motivate new architectures for efficiency
  • Ex: specialization, heterogeneity, management

Technology
• Emerging technologies motivate new architectures for capability
  • Ex: memory (phase change), networks (optical),

Reliability and Security
• Variations in fabrication, design process motivate new safeguards
  • Ex: tunable structures, trusted bases

Multiprocessors
• Abundant transistors, performance goals motivate parallel computing
  • Ex: parallel programming, coherence/consistency, management