Review, #1

- Technology is changing rapidly:
  - Capacity 2x in 3 years 2x in 3 years
  - Speed 2x in 1.5 years
  - Logic 2x in 3 years 2x in 3 years
  - DRAM 4x in 3 years 2x in 10 years
  - Disk 4x in 3 years 2x in 10 years
  - Processor (n.a.) 2x in 1.5 years
- What was true five years ago is not necessarily true now.
- Execution time is the REAL measure of computer performance!
  - Not clock rate, not CPI
- "X is n times faster than Y" means:
  \[ \frac{\text{ExTime}(y)}{\text{ExTime}(X)} = \frac{\text{Performance}(X)}{\text{Performance}(Y)} \]

Today:
Quick review of everything you should have learned

Computer Performance

<table>
<thead>
<tr>
<th></th>
<th>CPU time (Seconds)</th>
<th>Inst Count</th>
<th>CPI</th>
<th>Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>X</td>
<td>X</td>
<td>(X)</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Inst. Set.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Technology</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Cycles Per Instruction (Throughput)

"Average Cycles per Instruction"

\[ CPI = \frac{\text{CPU time} \times \text{Clock Rate}}{\text{Instruction Count}} \]

\[ \text{CPU time} = \text{Cycle Time} \times \sum_j CPI_j \times I_j \]

\[ CPI = \sum_j CPI_j \times F_j \quad \text{where} \quad F_j = \frac{I_j}{\text{Instruction Count}} \]

"Instruction Frequency"
Example: Calculating CPI bottom up

Run benchmark and collect workload characterization (simulate, machine counters, or sampling)

<table>
<thead>
<tr>
<th>Base Machine (Reg / Reg)</th>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI(i)</th>
<th>(% Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>1</td>
<td>(33%)</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>(27%)</td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>2</td>
<td>.2</td>
<td>(13%)</td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>(27%)</td>
<td></td>
</tr>
</tbody>
</table>

Typical Mix of instruction types in program

Design guideline: Make the common case fast

MIPS 1% rule: only consider adding an instruction if it is shown to add 1% performance improvement on reasonable benchmarks.

Example: Branch Stall Impact

- Assume CPI = 1.0 ignoring branches (ideal)
- Assume solution was stalling for 3 cycles
- If 30% branch, Stall 3 cycles on 30%

<table>
<thead>
<tr>
<th>Other Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI(i)</th>
<th>(% Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>1</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>(27%)</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>2</td>
<td>.2</td>
<td>(13%)</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>(27%)</td>
</tr>
</tbody>
</table>

⇒ new CPI = 1.9

- New machine is 1/1.9 = 0.52 times faster (i.e. slow!)

SPEC: System Performance Evaluation

Cooperative

- First Round 1989
  - 10 programs yielding a single number (“SPECmarks”)
- Second Round 1992
  - SPECint92 (8 integer programs) and SPECfp92 (14 floating point programs)
    - Compiler Flags unlimited. March 93 of DEC 4000 Model 610:
      - spice: unix.c:/def=(sysv,has_bcopy,"bcopy(a,b,c)=memcpy(b,a,c)"
      - wave5: /ali=(all,dcom=nat)/ag=a/ur=4/ur=200
      - nasa7: /norecu/ag=a/ur=4/ur2=200/lc=blas
- Third Round 1995
  - new set of programs: SPECint95 (8 integer programs) and SPECfp95 (10 floating point)
    - “benchmarks useful for 3 years”
    - Single flag setting for all programs: SPECint_base95, SPECfp_base95
- Fourth Round 2000: 26 apps
  - analysis and simulation programs
  - Compression: bzip2, gzip,
  - Integrated circuit layout, ray tracing, lots of others

Integrated Circuits Costs

\[
\text{IC cost} = \frac{\text{Die cost} + \text{Testing cost} + \text{Packaging cost}}{\text{Final test yield}}
\]

\[
\text{Die cost} = \text{Wafer cost} \times \frac{\text{Dies per Wafer}}{\text{Die yield}}
\]

\[
\frac{\text{Dies per wafer}}{\text{Die yield}} = \frac{(\text{Wafer, diam}/2)^2 \times \text{Die Area}}{\text{Die Area} - \text{Test_Die Area} - \text{Defect Density} - \text{Die area}^4}
\]

A "Typical" RISC

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
  - Single address mode for load/store: base + displacement
    - no indirectation
- Simple branch conditions
- Delayed branch

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3
Example: MIPS (- DLX)

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs1</th>
<th>Rs2</th>
<th>Rd</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>12</td>
<td>16 15</td>
</tr>
</tbody>
</table>

Register-Register

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs1</th>
<th>Rs2</th>
<th>Rd</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>12</td>
<td>16 15</td>
</tr>
</tbody>
</table>

Register-Immediate

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs1</th>
<th>Rs2</th>
<th>Rd</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>12</td>
<td>16 15</td>
</tr>
</tbody>
</table>

Branch

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs1</th>
<th>Rs2</th>
<th>Rd</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>12</td>
<td>16 15</td>
</tr>
</tbody>
</table>

Jump / Call

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs1</th>
<th>Rs2</th>
<th>Rd</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>12</td>
<td>16 15</td>
</tr>
</tbody>
</table>

Datapath vs Control

- Datapath: Storage, FU, interconnect sufficient to perform the desired functions
  - Inputs are Control Points
  - Outputs are signals
- Controller: State machine to orchestrate operation on the data path

Approaching an ISA

- Instruction Set Architecture
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- Meaning of each instruction is described by RTL on architected registers and memory
- Given technology constraints assemble adequate datapath
  - Architectured storage mapped to actual storage
  - Function units to do all the required operations
  - Possible additional storage (eg. MAR, MBR, ...)
  - Interconnect to move information among regs and FUs
- Map each instruction to sequence of RTLs
- Collate sequences into symbolic controller state transition diagram (STD)
- Lower symbolic STD to control points
- Implement controller

5 Steps of DLX Datapath

1. Instruction Fetch
2. Instruction Decode
3. Register Fetch
4. Execute
5. Memory Access

Inst. Set Processor Controller

- IR <= mem[PC];
- PC <= PC + 4
- A <= Reg[IRrs];
- B <= Reg[IRrt];
- rslt <= A opIRop B
- Reg[IRrd] <= WB
- WB <= rslt
5 Steps of DLX Datapath

Figure 3.4, Page 134

- Instruction Fetch
- Instr. Decode
- Execute
- Memory Access
- Write Back

Visualizing Pipelining

Figure 3.3, Page 133

- Instruction Fetch
- Instruction Decode
- Register Fetch
- ALU
- Address Calculation
- Memory Access
- Write Back

CS 252 Administrivia

- Review: Chapters 1-2, App A
- CS 152 home page, maybe "Computer Organization and Design (COD)2/e"
  - If did take a class, be sure COD Chapters 2, 5, 6, 7 are familiar
  - Copies in Bechtel Library on 2-hour reserve
- Resources for course on web site:
  - Check out the ISCA (International Symposium on Computer Architecture) 25th year retrospective on web site.
  - Look for "Additional reading" below text-book description
  - Pointers to previous CS152 exams and resources
  - Lots of old CS252 material
  - Interesting pointers at bottom. Check out the: WWW Computer Architecture Home Page
- Great ISA debate on tuesday

Pipelining is not quite that easy!

- Limits to pipelining: 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)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

One Memory Port/Structural Hazards

Figure 3.6, Page 142

- Memory Access
- Write Back

One Memory Port/Structural Hazards

Time (clock cycles)

- Load
- Instr 1
- Instr 2
- Instr 3
- Instr 4

- Data stationary control
  - Local decode for each instruction phase / pipeline stage

How do you “bubble” the pipe?
### Speed Up Equation for Pipelining

\[
\text{ Speedup } = \frac{\text{Ideal CPI} \times \text{Pipeline depth} \times \text{Cycle Time}_{\text{unpipelined}}}{\text{Ideal CPI} + \text{Pipeline stall CPI} \times \text{Cycle Time}_{\text{pipelined}}}
\]

For simple RISC pipeline, CPI = 1:

\[
\text{ Speedup } = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}
\]

### Example: Dual-port vs. Single-port

- Machine A: Dual ported memory ("Harvard Architecture")
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

\[
\text{SpeedUp}_A = \frac{\text{Pipeline Depth}}{1 + 0} \times \frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{pipe}}}
\]

\[
\text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{1 + 0.4 \times 1} \times \frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{unpipe}} / 1.05}
\]

\[
\text{SpeedUp}_A = \text{Pipeline Depth} \times 1.05 = 0.75 \times \text{Pipeline Depth}
\]

\[
\frac{\text{SpeedUp}_A}{\text{SpeedUp}_B} = \frac{\text{Pipeline Depth}}{0.75 \times \text{Pipeline Depth}} = 1.33
\]

- Machine A is 1.33 times faster

### Data Hazard on R1

Three Generic Data Hazards

- **Read After Write (RAW)**
  - Instr\textsubscript{J} tries to read operand before Instr\textsubscript{I} writes it
  - Caused by a "dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

- **Write After Read (WAR)**
  - Instr\textsubscript{J} writes operand before Instr\textsubscript{I} reads it
  - Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1".
  - Can't happen in DLX 5 stage pipeline because:
    - All instructions take 5 stages, and
    - Reads are always in stage 2, and
    - Writes are always in stage 5

- **Write After Write (WAW)**
  - Instr\textsubscript{J} writes operand before Instr\textsubscript{I} writes it.
  - Called an "output dependence" by compiler writers. This also results from the reuse of name "r1".
  - Can't happen in DLX 5 stage pipeline because:
    - All instructions take 5 stages, and
    - Writes are always in stage 5
  - Will see WAR and WAW in more complicated pipes
Forwarding to Avoid Data Hazard

Time (clock cycles)

\[
\begin{align*}
\text{add} & \ r1, r2, r3 \\
\text{sub} & \ r4, r1, r3 \\
\text{and} & \ r6, r1, r7 \\
\text{or} & \ r8, r1, r9 \\
\text{xor} & \ r10, r1, r11
\end{align*}
\]

What circuit detects and resolves this hazard?

HW Change for Forwarding

Figure 3.20, Page 161

Data Hazard Even with Forwarding

Time (clock cycles)

\[
\begin{align*}
\text{lw} & \ r1, 0(r2) \\
\text{sub} & \ r4, r1, r6 \\
\text{and} & \ r6, r1, r7 \\
\text{or} & \ r8, r1, r9
\end{align*}
\]

How is this detected?

Software Scheduling to Avoid Load Hazards

Try producing fast code for
\[
\begin{align*}
a & = b + c; \\
d & = e - f;
\end{align*}
\]
assuming a, b, c, d, e, and f in memory.

Slow code:  Fast code:
\[
\begin{align*}
\text{LW} & \ Rb, b \quad \text{LW} \ Rb, b \\
\text{LW} & \ Rc, c \quad \text{LW} \ Rc, c \\
\text{ADD} & \ Ra,Rb,Rc \quad \text{ADD} \ Ra,Rb,Rc \\
\text{SW} & \ a,Ra \quad \text{ADD} \ Ra,Rb,Rc \\
\text{LW} & \ Re,e \quad \text{LW} \ Re,f \\
\text{SW} & \ RF,RF \quad \text{SW} \ a,Ra \\
\text{SUB} & \ Rd,Ra,Rf \quad \text{SUB} \ Rd,Ra,Rf \\
\text{SW} & \ d,Rd \quad \text{SW} \ d,Rd
\end{align*}
\]

Compiler optimizes for performance. Hardware checks for safety.

Control Hazard on Branches

Three Stage Stall

\[
\begin{align*}
10: & \ \text{beq} \ r1, r3, 36 \\
14: & \ \text{and} \ r2, r3, r5 \\
18: & \ \text{or} \ r6, r1, r7 \\
22: & \ \text{add} \ r8, r1, r9 \\
36: & \ \text{xor} \ r10, r1, r11
\end{align*}
\]

What do you do with the 3 instructions in between?

How do you do it?

Where is the "commit"?
Branch Stall Impact

- If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- DLX branch tests if register = 0 or ≠ 0
- DLX Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear
#2: Predict Branch Not Taken
  - Execute successor instructions in sequence
  - "Squash" instructions in pipeline if branch actually taken
  - Advantage of late pipeline state update
  - 47% DLX branches not taken on average
  - PC+4 already calculated, so use it to get next instruction
#3: Predict Branch Taken
  - 53% DLX branches taken on average
  - But haven’t calculated branch target address in DLX
  - DLX still incurs 1 cycle branch penalty
  - Other machines: branch target known before outcome

Delayed Branch

- Where to get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: only valuable when branch taken
  - From fall through: only valuable when branch not taken
  - Canceling branches allow more slots to be filled

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)

Evaluating Branch Alternatives

<table>
<thead>
<tr>
<th></th>
<th>Stall pipeline</th>
<th>Predict taken</th>
<th>Predict not taken</th>
<th>Delayed branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall CPI</td>
<td>1.42</td>
<td>1.14</td>
<td>1.09</td>
<td>0.5</td>
</tr>
<tr>
<td>Unpipelined speedup</td>
<td>3.5</td>
<td>4.4</td>
<td>4.5</td>
<td>4.6</td>
</tr>
<tr>
<td>Pipelined speedup</td>
<td>1.0</td>
<td>1.26</td>
<td>1.29</td>
<td>1.31</td>
</tr>
</tbody>
</table>

Conditional & Unconditional = 14%, 65% change PC
Now, Review of Memory Hierarchy

Recap: Who Cares About the Memory Hierarchy?

Levels of the Memory Hierarchy

The Principle of Locality

Memory Hierarchy: Terminology

Cache Measures
Simplest Cache: Direct Mapped

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4 Byte Direct Mapped Cache</td>
</tr>
<tr>
<td>1</td>
<td>Cache Index</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

- Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, etc.
  - In general: any memory location whose 2 LSBs of the address are 0s
  - Address<1:0> => cache index

- Which one should we place in the cache?

- How can we tell which one is in the cache?

1 KB Direct Mapped Cache, 32B blocks

- For a $2^N$ byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = $2^M$)

Two-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operate in parallel (N typically 2 to 4)

- Example: Two-way set associative cache
  - Cache Index selects a "set" from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result

Disadvantage of Set Associative Cache

- N-way Set Associative Cache v. Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data

  - Data comes AFTER Hit/ Miss

- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:

  - Possible to assume a hit and continue. Recover later if miss.

4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level?  
  (Block placement)

- Q2: How is a block found if it is in the upper level?  
  (Block identification)

- Q3: Which block should be replaced on a miss?  
  (Block replacement)

- Q4: What happens on a write?  
  (Write strategy)
Q2: How is a block found if it is in the upper level?
- Tag on each block
  - No need to check index or block offset
- Increasing associativity shrinks index, expands tag

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
</tr>
</tbody>
</table>

Q3: Which block should be replaced on a miss?
- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

Assoc: 2-way 4-way 8-way

<table>
<thead>
<tr>
<th>Size</th>
<th>LRU</th>
<th>Ran</th>
<th>LRU</th>
<th>Ran</th>
<th>LRU</th>
<th>Ran</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
<td>5.3%</td>
<td>4.4%</td>
<td>5.0%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
<td>1.7%</td>
<td>1.4%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
<td>1.13%</td>
<td>1.12%</td>
<td>1.12%</td>
</tr>
</tbody>
</table>

Q4: What happens on a write?
- Write through—The information is written to both the block in the cache and to the block in the lower-level memory.
- Write back—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - Is block clean or dirty?
- Pros and Cons of each?
  - WT: read misses cannot result in writes
  - WB: no repeated writes to same location
- WT always combined with write buffers so that don’t wait for lower level memory

Write Buffer for Write Through
- A Write Buffer is needed between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: writes content of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
- Memory system designer’s nightmare:
  - Store frequency (w.r.t. time) -> 1 / DRAM write cycle
  - Write buffer saturation

Impact of Memory Hierarchy on Algorithms
- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?
- Quicksort: fastest comparison based sorting algorithm when all keys fit in memory
- Radix sort: also called “linear time” sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys
- For Alphastation 250, 32 byte blocks, direct mapped L2 2MB cache, 8 byte keys, from 4000 to 4000000

Quick sort vs. Radix as vary number keys: Instructions

Page 10
Quicksort vs. Radix as vary number keys:

- Instrs & Time
- Cache misses

What is proper approach to fast algorithms?

A Modern Memory Hierarchy

- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.

Three Advantages of Virtual Memory

- Translation:
  - Program can be given consistent view of memory, even though physical memory is scrambled
  - Only the most important part of program ("Working Set") must be in physical memory.
  - Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later.

- Protection:
  - Different threads (or processes) protected from each other.
  - Different pages can be given special behavior
  - Kernel data protected from User programs
  - Far more "viruses" under Microsoft Windows

- Sharing:
  - Can map same physical page to multiple users

Issues in Virtual Memory System Design

What is the size of information blocks that are transferred from secondary to main storage (M)? ⇒ page size

Which region of M is to hold the new block ⇒ placement policy

How do we find a page when we look for it? ⇒ block identification

Block of information brought into M, and M is full, then some region of M must be released to make room for the new block ⇒ replacement policy

What do we do on a write? ⇒ write policy

Missing item fetched from secondary memory only on the occurrence of a fault ⇒ demand load policy

What is virtual memory?

- Virtual memory ⇒ treat memory as a cache for the disk
- Terminology: blocks in this cache are called "Pages"
  - Typical size of a page: 5K — 8K
- Page table maps virtual page numbers to physical frames
  - "PTE" = Page Table Entry

What is proper approach to fast algorithms?

Set size in keys

Quick sort

Radix sort

Instructions

Cache misses

Quick sort

Radix sort

Time

Instructions vs. Time

Cache misses vs. Set size in keys

Set size in keys

Quick sort

Radix sort

Instructions

Cache misses
Large Address Spaces

Two-level Page Tables

32-bit address:

\( \text{P1 index P2 index page offset} \)

\( 4 \text{ bytes} \)

* 2 GB virtual address space
* 4 KB of PTE2
  * paget, index
* 4 KB of PTE1

\( 4 \text{ bytes} \)

\[ \text{4KB} \]

\[ \text{PTEs} \]

What about a 48-64 bit address space?

Translation Look-Aside Buffers

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped.

TLBs are usually small, typically not more than 128 - 256 entries even on high-end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.

Translation with a TLB

CPU

VA

hit

PA

miss

CPU

TLB

Lookup

Cache

Translation

hit

miss

Main Memory

data

Translation with a TLB

Overlapped Cache & TLB Access

If cache hit AND (cache tag = PA) then deliver data to CPU
ELSE IF [cache miss OR (cache tag = PA)] and TLB hit THEN access memory with the PA from the TLB
ELSE do standard VA translation

Overlapped access only works as long as the address bits used to index into the cache do not change as the result of VA translation. This usually limits things to small caches, large page sizes, or high n-way set associative caches if you want a large cache.

Example: suppose everything the same except that the cache is increased to 8 K bytes instead of 4 K:

Overlapped Cache & TLB Access

Problems With Overlapped TLB Access

This bit is changed by VA translation, but is needed for cache lookup.

Solutions:
- go to 8K byte page sizes;
- go to 2 way set associative cache; or

Summary #1/5: Control and Pipelining

- Control VIA State Machines and Microprogramming
- Just overlap tasks; easy if tasks are independent
- Speed Up \( \leq \) Pipeline Depth; if ideal CPI is 1, then:

\[
\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{required}}{\text{Cycle Time}_{pipeline}}
\]

- Hazards limit performance on computers:
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
  - Control: delayed branch, prediction

Summary #2/5: Caches

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- Three Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Capacity Misses: increase cache size
  - Conflict Misses: increase cache size and/or associativity. Nightmare Scenario: ping pong effect!
- Write Policy:
  - Write Through: needs a write buffer. Nightmare: WB saturation
  - Write Back: control can be complex
Summary #3/5: The Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins

Summary #4/5: TLB, Virtual Memory

- Caches, TLBs, Virtual Memory all understood by examining how they deal with 4 questions: 1) Where can block be placed? 2) How is block found? 3) What block is replaced on miss? 4) How are writes handled?
  - Page tables map virtual address to physical address
  - TLBs are important for fast translation
  - TLB misses are significant in processor performance
    - funny times, as most systems can't access all of 2nd level cache without TLB misses!

Summary #5/5: Memory Hierachy

- Virtual memory was controversial at the time: can SW automatically manage 64KB across many programs?
  - 1000X DRAM growth removed the controversy
- Today VM allows many processes to share single memory without having to swap all processes to disk; today VM protection is more important than memory hierarchy
- Today CPU time is a function of (ops, cache misses) vs. just f(ops):
  What does this mean to Compilers, Data structures, Algorithms?