CSE 502: Computer Architecture

Review
Course Overview (1/2)

• Caveat 1: I’m (kind of) new here.
• Caveat 2: This is a (somewhat) new course.

• Computer Architecture is
  ... the science and art of selecting and interconnecting hardware and software components to create computers ...

...
Course Overview (2/2)

• This course is hard, roughly like CSE 506
  – In CSE 506, you learn what’s inside an OS
  – In CSE 502, you learn what’s inside a CPU

• This is a project course
  – Learn why things are the way they are, first hand
  – We will “build” emulators of CPU components
Policy and Projects

• Probably different from other classes
  – Much more open, but much more strict
    • Most people followed the policy
    • Some did not
  – Resembles the “real world”
    • You’re here because you want to learn and to be here
    • If you managed to get your partner(s) to do the work
      – You’re probably good enough to do it at your job too
        » The good: You might make a good manager
        » The bad: You didn’t learn much

• Time mgmt. often more important than tech. skill
  – If you started early, you probably have an A already
**Amdahl’s Law**

\[
\text{Speedup} = \frac{\text{time}_{\text{without enhancement}}}{\text{time}_{\text{with enhancement}}}
\]

An enhancement speeds up fraction \( f \) of a task by factor \( S \)

\[
\text{time}_{\text{new}} = \text{time}_{\text{orig}} \cdot ( (1-f) + \frac{f}{S} )
\]

\[
S_{\text{overall}} = \frac{1}{(1-f) + \frac{f}{S}}
\]
The *Iron Law* of Processor Performance

\[
\text{Time} = \frac{\text{Total Work In Program}}{\text{CPI or } 1/\text{IPC}} \times \frac{\text{Cycle Time}}{\text{Microarchitecture}} \times \frac{\text{Program Instructions}}{\text{Microarchitecture, Process Tech}}
\]

Architects target CPI, but *must* understand the others.
Averaging Performance Numbers (2/2)

- **Arithmetic**: times
  - proportional to time
  - e.g., latency

- **Harmonic**: rates
  - inversely proportional to time
  - e.g., throughput

- **Geometric**: ratios
  - unit-less quantities
  - e.g., speedups
Power vs. Energy

• **Power**: instantaneous rate of energy transfer
  – Expressed in Watts
  – In Architecture, implies conversion of electricity to heat
  – Power(Comp1+Comp2)=Power(Comp1)+Power(Comp2)

• **Energy**: measure of using power for some time
  – Expressed in Joules
  – power * time (joules = watts * seconds)
  – Energy(OP1+OP2)=Energy(OP1)+Energy(OP2)
ISA: A contract between HW and SW

• **ISA**: Instruction Set Architecture
  – A well-defined hardware/software interface

• The “contract” between software and hardware
  – Functional definition of operations supported by hardware
  – Precise description of how to invoke all features

• No guarantees regarding
  – How operations are implemented
  – Which operations are fast and which are slow (and when)
  – Which operations take more energy (and which take less)
Components of an ISA

• Programmer-visible states
  – Program counter, general purpose registers, memory, control registers

• Programmer-visible behaviors
  – What to do, when to do it

Example “register-transfer-level” description of an instruction

if imem[pc]==“add rd, rs, rt”
then
  pc \leftarrow pc+1
  gpr[rd]=gpr[rs]+gpr[rt]

• A binary encoding

ISAs last forever, don’t add stuff you don’t need
Locality Principle

• Recent past is a good indication of near future

_Spatial Locality:_ If you looked something up, it is very likely you will look up something nearby soon

_Temporal Locality:_ If you looked something up, it is very likely that you will look it up again soon
Caches

• An *automatically managed* hierarchy

• Break memory into *blocks* (several bytes) and transfer data to/from cache in blocks
  – *spatial locality*

• Keep recently accessed blocks
  – *temporal locality*
Fully-Associative Cache

- Keep blocks in cache frames
  - data
  - state (e.g., valid)
  - address tag

What happens when the cache runs out of space?
The 3 C’s of Cache Misses

• **Compulsory**: Never accessed before
• **Capacity**: Accessed long ago and already replaced
• **Conflict**: Neither compulsory nor capacity
• **Coherence**: (In multi-cores, become owner to write)
Cache Size

- Cache size is data capacity (don’t count tag and state)
  - Bigger can exploit temporal locality better
  - Not always better

- Too large a cache
  - Smaller is faster \( \rightarrow \) bigger is slower
  - Access time may hurt critical path

- Too small a cache
  - Limited temporal locality
  - Useful data constantly replaced
Block Size

• Block size is the data that is
  – Associated with an address tag
  – Not necessarily the unit of transfer between hierarchies

• Too small a block
  – Don’t exploit spatial locality well
  – Excessive tag overhead

• Too large a block
  – Useless data transferred
  – Too few total blocks
    • Useful data frequently replaced
Direct-Mapped Cache

- Use middle bits as index
- Only one tag comparison

```
```
**N-Way Set-Associative Cache**

Note the additional bit(s) moved from index to tag.
Associativity

- Larger associativity
  - lower miss rate (fewer conflicts)
  - higher power consumption

- Smaller associativity
  - lower cost
  - faster hit time

hit rate vs. associativity

~5 for L1-D
Parallel vs Serial Caches

• Tag and Data usually separate (tag is smaller & faster)
  – State bits stored along with tags
    • *Valid* bit, “LRU” bit(s), ...

Parallel access to Tag and Data reduces latency (good for L1)

Serial access to Tag and Data reduces power (good for L2+)
Physically-Indexed Caches

- Core requests are VAs
- Cache index is PA[15:6]
  - VA passes through TLB
  - D-TLB on critical path
- Cache tag is PA[63:16]
- If index size < page size
  - Can use VA for index
Virtually-Indexed Caches

- Core requests are VAs
- Cache index is VA[15:6]
- Cache tag is PA[63:16]

- Why not tag with VA?
  - Cache flush on ctx switch
- Virtual aliases
  - Ensure they don’t exist
  - ... or check all on miss
Inclusion

• Core often accesses blocks not present on chip
  – Should block be allocated in L3, L2, and L1?
    • Called *Inclusive* caches
    • Waste of space
    • Requires forced evict (e.g., force evict from L1 on evict from L2+)
  – Only allocate blocks in L1
    • Called *Non-inclusive* caches (who not “exclusive”?)
    • Must write back clean lines

• Some processors combine both
  – L3 is inclusive of L1 and L2
  – L2 is non-inclusive of L1 (like a large victim cache)
Parity & ECC

• Cosmic radiation can strike at any time
  – Especially at high altitude
  – Or during solar flares
• What can be done?
  – **Parity**
    • 1 bit to indicate if sum is odd/even (detects single-bit errors)
  – **Error Correcting Codes (ECC)**
    • 8 bit code per 64-bit word
    • Generally **SECDED (Single-Error-Correct, Double-Error-Detect)**
• Detecting errors on clean cache lines is harmless
  – Pretend it’s a cache miss
SRAM vs. DRAM

- SRAM = Static RAM
  - As long as power is present, data is retained
- DRAM = Dynamic RAM
  - If you don’t do anything, you lose the data

- SRAM: 6T per bit
  - built with normal high-speed CMOS technology
- DRAM: 1T per bit (+1 capacitor)
  - built with special DRAM process optimized for density
DRAM Chip Organization

- Low-Level organization is very similar to SRAM
- Cells are only single-ended
  - Reads *destructive*: contents are erased by reading
- **Row buffer** holds read data
  - Data in row buffer is called a *DRAM row*
    - Often called “page” - not necessarily same as OS page
  - Read gets entire row into the buffer
  - Block reads always performed out of the row buffer
    - Reading a whole row, but accessing one block
    - Similar to reading a cache line, but accessing one word
All banks within the rank share all address and control pins.

All banks are independent, but can only talk to one bank at a time.

x8 means each DRAM outputs 8 bits, need 8 chips for DDRx (64-bit).

Why 9 chips per rank? 64 bits data, 8 bits ECC.

Dual-rank x8 (2Rx8) DIMM
AMAT with MLP

• If ...
cache hit is 10 cycles (core to L1 and back)memory access is 100 cycles (core to mem and back)

• Then ...
at 50% miss ratio, avg. access: $0.5 \times 10 + 0.5 \times 100 = 55$

• Unless MLP is $> 1.0$, then...
at 50% mr, 1.5 MLP, avg. access: \(\frac{0.5 \times 10 + 0.5 \times 100}{1.5} = 37\)
at 50% mr, 4.0 MLP, avg. access: \(\frac{0.5 \times 10 + 0.5 \times 100}{4.0} = 14\)

In many cases, MLP dictates performance
Memory Controller (1/2)

![Diagram of a memory controller with queues, scheduler, buffer, and channels]

- Read Queue
- Write Queue
- Response Queue
- Scheduler
- Buffer
- Channel 0
- Channel 1
Memory Controller (2/2)

• Memory controller connects CPU and DRAM
• Receives requests after cache misses in LLC
  – Possibly originating from multiple cores

• Complicated piece of hardware, handles:
  – DRAM Refresh
  – Row-Buffer Management Policies
  – Address Mapping Schemes
  – Request Scheduling
Address Mapping Schemes

• Example Open-page Mapping Scheme:
  
  
  High Parallelism: [row rank bank column channel offset]
  Easy Expandability: [channel rank row bank column offset]

• Example Close-page Mapping Scheme:
  
  High Parallelism: [row column rank bank channel offset]
  Easy Expandability: [channel rank row column bank offset]
Memory Request Scheduling

- **Write buffering**
  - Writes can wait until reads are done

- **Queue DRAM commands**
  - Usually into per-bank queues
  - Allows easily reordering ops. meant for same bank

- **Common policies:**
  - *First-Come-First-Served (FCFS)*
  - *First-Ready—First-Come-First-Served (FR-FCFS)*
Prefetching (1/2)

- Fetch block ahead of demand
- Target compulsory, capacity, (& coherence) misses
  - Not conflict: prefetched block would conflict

- Big challenges:
  - Knowing “what” to fetch
    - Fetching useless blocks wastes resources
  - Knowing “when” to fetch
    - Too early → clutters storage (or gets thrown out before use)
    - Fetching too late → defeats purpose of “pre”-fetching
Prefetching (2/2)

- Without prefetching:

- With prefetching:

- Or:

  Prefetching must be **accurate** and **timely**
Next-Line (or Adjacent-Line) Prefetching

- On request for line X, prefetch X+1 (or X^0x1)
  - Assumes spatial locality
    - Often a good assumption
  - Should stop at physical (OS) page boundaries
- Can often be done efficiently
  - Adjacent-line is convenient when next-level block is bigger
  - Prefetch from DRAM can use bursts and row-buffer hits
- Works for I$ and D$
  - Instructions execute sequentially
  - Large data structures often span multiple blocks

Simple, but usually not timely
Next-N-Line Prefetching

- On request for line X, prefetch X+1, X+2, ..., X+N
  - N is called "prefetch depth" or "prefetch degree"

- Must carefully tune depth N. Large N is ...
  - More likely to be useful (correct and timely)
  - More aggressive → more likely to make a mistake
    - Might evict something useful
  - More expensive → need storage for prefetched lines
    - Might delay useful request on interconnect or port

Still simple, but more timely than Next-Line
Stride Prefetching

- Access patterns often follow a **stride**
  - Accessing column of elements in a matrix
  - Accessing elements in array of structs
- Detect stride $S$, prefetch depth $N$
  - Prefetch $X+1 \cdot S$, $X+2 \cdot S$, ..., $X+N \cdot S$
“Localized” Stride Prefetchers

- Store PC, last address, last stride, and count in RPT
- On access, check **RPT (Reference Prediction Table)**
  - Same stride? → count++ if yes, count-- or count=0 if no
  - If count is high, prefetch (last address + stride*N)

<table>
<thead>
<tr>
<th>PC</th>
<th>Last Address</th>
<th>Stride</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x409A34</td>
<td>A+3N</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>0x409A38</td>
<td>X+3N</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>0x409A40</td>
<td>Y+2N</td>
<td>N</td>
<td>1</td>
</tr>
</tbody>
</table>

PCa: 0x409A34  Load R1 = [R2]
PCb: 0x409A38  Load R3 = [R4]
PCc: 0x409A40  Store [R6] = R5

If confident about the stride (count > C_min), prefetch (A+4N)
Evaluating Prefetchers

• Compare against larger caches
  – Complex prefetcher vs. simple prefetcher with larger cache

• Primary metrics
  – **Coverage**: prefetched hits / base misses
  – **Accuracy**: prefetched hits / total prefetches
  – **Timeliness**: latency of prefetched blocks / hit latency

• Secondary metrics
  – **Pollution**: misses / (prefetched hits + base misses)
  – Bandwidth: total prefetches + misses / base misses
  – Power, Energy, Area...
Before there was pipelining...

Single-cycle control: hardwired
- Low CPI (1)
- Long clock period (to accommodate slowest instruction)

Multi-cycle control: micro-programmed
- Short clock period
- High CPI

Can we have both low CPI and short clock period?
Pipelining

- Start with multi-cycle design
- When insn0 goes from stage 1 to stage 2
  ... insn1 starts stage 1
- Each instruction passes through all stages
  ... but instructions enter and leave at faster rate

Can have as many insns in flight as there are stages
Instruction Dependencies

• Data Dependence
  – *Read-After-Write (RAW)* (only true dependence)
    • Read must wait until earlier write finishes
  – *Anti-Dependence (WAR)*
    • Write must wait until earlier read finishes (avoid clobbering)
  – *Output Dependence (WAW)*
    • Earlier write can’t overwrite later write

• Control Dependence (a.k.a. Procedural Dependence)
  – Branch condition must execute before branch target
  – Instructions after branch cannot run before branch
Pipeline Terminology

• **Pipeline Hazards**
  – Potential violations of program dependencies
  – Must ensure program dependencies are not violated

• **Hazard Resolution**
  – Static method: performed at compile time in software
  – Dynamic method: performed at runtime using hardware
  – Two options: Stall (costs perf.) or Forward (costs hw.)

• **Pipeline Interlock**
  – Hardware mechanism for dynamic hazard resolution
  – Must detect and enforce dependencies at runtime
Balancing Pipeline Stages

Coarser-Grained Machine Cycle:
- 4 machine cyc / instruction

<table>
<thead>
<tr>
<th>Stage</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF &amp; ID</td>
<td>8 units</td>
</tr>
<tr>
<td>OF</td>
<td>9 units</td>
</tr>
<tr>
<td>EX</td>
<td>5 units</td>
</tr>
<tr>
<td>WB</td>
<td>9 units</td>
</tr>
</tbody>
</table>

- # stages = 4
- $T_{cyc} = 9$ units

Finer-Grained Machine Cycle:
- 11 machine cyc / instruction

<table>
<thead>
<tr>
<th>Stage</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>1 unit</td>
</tr>
<tr>
<td>ID</td>
<td>2 units</td>
</tr>
<tr>
<td>OF</td>
<td>2 units</td>
</tr>
<tr>
<td>OF</td>
<td>2 units</td>
</tr>
<tr>
<td>OF</td>
<td>2 units</td>
</tr>
<tr>
<td>EX</td>
<td>1 unit</td>
</tr>
<tr>
<td>EX</td>
<td>1 unit</td>
</tr>
<tr>
<td>WB</td>
<td>1 unit</td>
</tr>
<tr>
<td>WB</td>
<td>1 unit</td>
</tr>
<tr>
<td>WB</td>
<td>1 unit</td>
</tr>
</tbody>
</table>

- # stages = 11
- $T_{cyc} = 3$ units
IPC vs. Frequency

• 10-15% IPC not bad if frequency can double

<table>
<thead>
<tr>
<th>Frequency</th>
<th>IPC</th>
<th>BIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GHz</td>
<td>2.0</td>
<td>2 BIPS</td>
</tr>
<tr>
<td>2 GHz</td>
<td>1.7</td>
<td>3.4 BIPS</td>
</tr>
</tbody>
</table>

• Frequency doesn’t double
  – Latch/pipeline overhead
  – Stage imbalance
Architectures for Instruction Parallelism

• Scalar pipeline (baseline)
  – Instruction/overlap parallelism = D
  – Operation Latency = 1
  – Peak IPC = 1.0
Superscalar Machine

• Superscalar (pipelined) Execution
  – Instruction parallelism = $D \times N$
  – Operation Latency = 1
  – Peak IPC = $N$ per cycle

![Graph showing 12 time steps with different instructions overlapped]
RISC ISA Format

• Fixed-length
  – MIPS all insts are 32-bits/4 bytes

• Few formats
  – MIPS has 3: R (reg, reg, reg), I (reg, reg, imm), J (addr)
  – Alpha has 5: Operate, Op w/ Imm, Mem, Branch, FP

• Regularity across formats (when possible/practical)
  – MIPS & Alpha opcode in same bit-position for all formats
  – MIPS rs & rt fields in same bit-position for R and I formats
  – Alpha ra/fa field in same bit-position for all 5 formats
Superscalar Decode for RISC ISAs

- Decode X insns. per cycle (e.g., 4-wide)
  - Just duplicate the hardware
  - Instructions aligned at 32-bit boundaries
CISC ISA

• RISC focus on fast access to information
  – Easy decode, I$, large RF’s, D$

• CISC focus on max expressiveness per min space
  – Designed in era with fewer transistors, chips
  – Each memory access very expensive
    • Pack as much work into as few bytes as possible
    • More “expressive” instructions
      – Better potential code generation in theory
      – More complex code generation in practice
## ADD in RISC ISA

<table>
<thead>
<tr>
<th>Mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>ADD R4, R3, R2</td>
<td>R4 = R3 + R2</td>
</tr>
</tbody>
</table>
## ADD in CISC ISA

<table>
<thead>
<tr>
<th>Mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>ADD R4, R3</td>
<td>R4 = R4 + R3</td>
</tr>
<tr>
<td>Immediate</td>
<td>ADD R4, #3</td>
<td>R4 = R4 + 3</td>
</tr>
<tr>
<td>Displacement</td>
<td>ADD R4, 100(R1)</td>
<td>R4 = R4 + Mem[100+R1]</td>
</tr>
<tr>
<td>Register Indirect</td>
<td>ADD R4, (R1)</td>
<td>R4 = R4 + Mem[R1]</td>
</tr>
<tr>
<td>Indexed/Base</td>
<td>ADD R3, (R1+R2)</td>
<td>R3 = R3 + Mem[R1+R2]</td>
</tr>
<tr>
<td>Direct/Absolute</td>
<td>ADD R1, (1234)</td>
<td>R1 = R1 + Mem[1234]</td>
</tr>
<tr>
<td>Memory Indirect</td>
<td>ADD R1, @(R3)</td>
<td>R1 = R1 + Mem[Mem[R3]]</td>
</tr>
<tr>
<td>Auto-Increment</td>
<td>ADD R1,(R2)+</td>
<td>R1 = R1 + Mem[R2]; R2++</td>
</tr>
<tr>
<td>Auto-Decrement</td>
<td>ADD R1, -(R2)</td>
<td>R2--; R1 = R1 + Mem[R2]</td>
</tr>
</tbody>
</table>
RISC (MIPS) vs CISC (x86)

\[ \text{lui R1, Disp[31:16]} \]
\[ \text{ori R1, R1, Disp[15:0]} \]
\[ \text{add R1, R1, R2} \]
\[ \text{shli R3, R3, 3} \]
\[ \text{add R3, R3, R1} \]
\[ \text{lui R1, Imm[31:16]} \]
\[ \text{ori R1, R1, Imm[15:0]} \]
\[ \text{st [R3], R1} \]

\text{MOV [EBX+EAX*8+Disp], Imm}

8 insns. at 32 bits each \textbf{vs} 1 insn. at 88 bits: \textbf{2.9x!}
x86 Encoding

• Basic x86 Instruction:

- Prefixes: 0-4 bytes
- Opcode: 1-2 bytes
- Mod R/M: 0-1 bytes
- SIB: 0-1 bytes
- Displacement: 0/1/2/4 bytes
- Immediate: 0/1/2/4 bytes

- Shortest Inst: 1 byte
- Longest Inst 15 bytes

- Opcode has flag indicating Mod R/M is present
  - Most instructions use the Mod R/M byte
  - Mod R/M specifies if optional SIB byte is used
  - Mod R/M and SIB may specify additional constants

Instruction length not known until after decode
Instruction Cache Organization

- To fetch N instructions per cycle...
  - L1-I line must be wide enough for N instructions
- PC register selects L1-I line
- A *fetch group* is the set of insns. starting at PC
  - For N-wide machine, [PC,PC+N-1]
Fetch Misalignment

• Now takes two cycles to fetch N instructions
Fragmentation due to Branches

- Fetch group is aligned, cache line size > fetch group
  - Taken branches still limit fetch width
Types of Branches

• Direction:
  – Conditional vs. Unconditional

• Target:
  – PC-encoded
    • PC-relative
    • Absolute offset
  – Computed (target derived from register)

Need direction and target to find next fetch group
Branch Prediction Overview

• Use two hardware predictors
  – Direction predictor guesses if branch is taken or not-taken
  – Target predictor guesses the destination PC

• Predictions are based on history
  – Use previous behavior as indication of future behavior
  – Use historical context to disambiguate predictions
Direction vs. Target Prediction

• Direction: 0 or 1
• Target: 32- or 64-bit value
• Turns out targets are generally easier to predict
  – Don’t need to predict N-t target
  – T target doesn’t usually change
• Only need to predict taken-branch targets
• Prediction is really just a “cache”
  – *Branch Target Buffer (BTB)*
Branch Target Buffer (BTB)

- Branch PC
- Valid Bit
- Branch Instruction Address (Tag)
- Branch Target Address
- Next Fetch PC
- Hit?
Fewer bits to compare, but prediction may alias
If target too far or PC rolls over, will mispredict
Branches Have Locality

• If a branch was previously taken...
  – There’s a good chance it’ll be taken again

```c
for(i=0; i < 100000; i++)
{
    /* do stuff */
}
```

This branch will be taken 99,999 times in a row.
Last Outcome Predictor

• Do what you did last time

```
0xDC08: for(i=0; i < 100000; i++) {
    0xDC44: if( (i % 100) == 0 ) tick();
    0xDC50: if( (i & 1) == 1 ) odd();
}
```
Saturating Two-Bit Counter

- Predict N-t
- Predict T
- Transition on T outcome
- Transition on N-t outcome

FSM for Last-Outcome Prediction

FSM for 2bC (2-bit Counter)
Typical Organization of 2bC Predictor

PC → hash

32 or 64 bits

log₂ n bits

n entries/counters

table update

Prediction

FSM Update Logic

Actual outcome
Track the *History* of Branches

- **prev = 1**  
  - Counter if prev=0: 3 3  
  - Counter if prev=1: 3 3  
  - Prediction = T

- **prev = 0**  
  - Counter if prev=0: 3 2  
  - Counter if prev=1: 3 2  
  - Prediction = T

- **prev = 1**  
  - Counter if prev=0: 3 2  
  - Counter if prev=1: 3 3  
  - Prediction = T

- **prev = 1**  
  - Counter if prev=0: 3 3  
  - Counter if prev=1: 3 3  
  - Prediction = T
Deeper History Covers More Patterns

- Counters learn “pattern” of prediction

\[
egin{align*}
001 & \rightarrow 1; 011 & \rightarrow 0; 110 & \rightarrow 0; 100 & \rightarrow 1 \\
00110011001\ldots & (0011)^* 
\end{align*}
\]
Predictor Training Time

- Ex: prediction equals opposite for 2\textsuperscript{nd} most recent
  - Hist Len = 2
  - 4 states to train:
    - $NN \rightarrow T$
    - $NT \rightarrow T$
    - $TN \rightarrow N$
    - $TT \rightarrow N$
  - Hist Len = 3
  - 8 states to train:
    - $NNN \rightarrow T$
    - $NNT \rightarrow T$
    - $NTN \rightarrow N$
    - $NTT \rightarrow N$
    - $TNN \rightarrow T$
    - $TNT \rightarrow T$
    - $TTN \rightarrow N$
    - $TTT \rightarrow N$
Predictor Organizations

- **PC Hash**
  - Different pattern for each branch PC

- **PC Hash**
  - Shared set of patterns

- **PC Hash**
  - Mix of both
Two-Level Predictor Organization

- **Branch History Table (BHT)**
  - $2^a$ entries
  - $h$-bit history per entry
- **Pattern History Table (PHT)**
  - $2^b$ sets
  - $2^h$ counters per set
- **Total Size in bits**
  - $h \times 2^a + 2^{(b+h)} \times 2$

Each entry is a 2-bit counter
Combined Indexing

• “gshare” (S. McFarling)
OoO Execution

- **Out-of-Order execution (OoO)**
  - Totally in the hardware
  - Also called Dynamic scheduling

- Fetch many instructions into *instruction window*
  - Use branch prediction to speculate past branches

- Rename regs. to avoid false deps. (WAW and WAR)

- Execute insns. as soon as possible
  - As soon as deps. (regs and memory) are known

- Today’s machines: 100+ insns. scheduling window
Superscalar != Out-of-Order

A: R1 = Load 16[R2]
B: R3 = R1 + R4
C: R6 = Load 8[R9]
D: R5 = R2 – 4
E: R7 = Load 20[R5]
F: R4 = R4 – 1
G: BEQ R4, #0

1-wide
In-Order

2-wide
In-Order

1-wide
Out-of-Order

2-wide
Out-of-Order

A cache miss

B cache miss

C

D

E

F

G

A

C

D

E

F

G

B

5 cycles

7 cycles

8 cycles

10 cycles
Review of Register Dependencies

Read-After-Write
A: R1 = R2 + R3
B: R4 = R1 * R4

Write-After-Read
A: R1 = R3 / R4
B: R3 = R2 * R4

Write-After-Write
A: R1 = R2 + R3
B: R1 = R3 * R4
Register Renaming

- **Register renaming** (in hardware)
  - “Change” register names to eliminate WAR/WAW hazards
  - Arch. registers (r1,f0...) are names, not storage locations
  - Can have more locations than names
  - Can have multiple active versions of same name

- How does it work?
  - Map-table: maps names to most recent locations
  - On a write: allocate new location, note in map-table
  - On a read: find location of most recent write via map-table
Tomasulo’s Algorithm

- **Reservation Stations** (RS): instruction buffer
- Common data bus (CDB): broadcasts results to RS
- Register renaming: removes WAR/WAW hazards
- Bypassing (not shown here to make example simpler)
Tomasulo Data Structures
Where is the “register rename”? 

- Value *copies* in RS (V1, V2) 
- Insn. stores correct input values in its own RS entry 
- “Free list” is implicit (allocate/deallocate as part of RS)
Precise State

• Speculative execution requires
  – (Ability to) abort & restart at every branch
  – Abort & restart at every load

• Synchronous (exception and trap) events require
  – Abort & restart at every load, store, divide, ...

• Asynchronous (hardware) interrupts require
  – Abort & restart at every ??

• Real world: bite the bullet
  – Implement abort & restart at every insn.
  – Called \textit{precise state}
Complete and Retire

- **Complete (C):** insns. write results into ROB
  - Out-of-order: don’t block younger insns.
- **Retire (R):** a.k.a. *commit*, graduate
  - ROB writes results to register file
  - In-order: stall back-propagates to younger insns.
P6 Data Structures

Map Table

Dispatch

RS

T

T+

CDB.T

V1

V2

value

FU

Regfile

CDB.V

ROB

R

value

Head

Retire

Tail

Dispatch
MIPS R10K: Alternative Implementation

- One big **physical register file** holds all data - no copies
  - Register file close to FUs → small and fast data path
  - ROB and RS “on the side” used only for control and tags
Executing Memory Instructions

- If R1 != R7
  - Then Load R8 gets correct value from cache
- If R1 == R7
  - Then Load R8 should get value from the Store
  - *But it didn’t!*

But there was a later load…
Memory Disambiguation Problem

• Ordering problem is a data-dependence violation
• Imprecise memory worse than imprecise registers

• Why can’t this happen with non-memory insts?
  – Operand specifiers in non-memory insns. are absolute
    • “R1” refers to one specific location
  – Operand specifiers in memory insns. are ambiguous
    • “R1” refers to a memory location specified by the value of R1.
    • When pointers (e.g., R1) change, so does this location
Two Problems

• Memory disambiguation on loads
  – Do earlier unexecuted stores to the same address exist?
    • Binary question: answer is yes or no

• Store-to-load forwarding problem
  – I’m a load: Which earlier store do I get my value from?
  – I’m a store: Which later load(s) do I forward my value to?
    • Non-binary question: answer is one or more insn. identifiers
Load/Store Queue (1/2)

- **Load/store queue (LSQ)**
  - Completed stores write to LSQ
  - When store retires, head of LSQ written to L1-D
    - (or write buffer)
  - When loads execute, access LSQ and L1-D in parallel
    - Forward from LSQ if older store with matching address
Load/Store Queue (2/2)

Almost a “real” processor diagram
Loads Execute When ...

• Most aggressive approach
• Relies on fact that store→load forwarding is rare
• Greatest potential IPC – loads never stall

• Potential for incorrect execution
  – Need to be able to “undo” bad loads
Detecting Ordering Violations

• Case 1: Older store execs before younger load
  – No problem; if same address st → ld forwarding happens

• Case 2: Older store execs after younger load
  – Store scans all younger loads
  – Address match → ordering violation
Loads Checking for Earlier Stores

- On Load dispatch, find data from earlier Store

Diagram:

- Address Bank:
  - ST 0x4000
  - ST 0x4000
  - ST 0x4120
  - LD 0x4000

- Data Bank:

- Logic:
  - Valid store
  - Addr match
  - No earlier matches

- CSE502: Computer Architecture

- Need to adjust this so that load need not be at bottom, and LSQ can wrap-around

- If |LSQ| is large, logic can be adapted to have log delay
Data Forwarding

- On execute Store (STA+STD), check for later Loads

This is ugly, complicated, slow, and power hungry
Data-Capture Scheduler

- Dispatch: read available operands from ARF/ROB, store in scheduler
- Commit: Missing operands filled in from bypass
- Issue: When ready, operands sent directly from scheduler to functional units
Scheduling Loop or *Wakeup-Select Loop*

- **Wake-Up Part:**
  - Executing insn notifies dependents
  - Waiting insns. check if all deps are satisfied
    - If yes, “wake up” instruction

- **Select Part:**
  - Choose which instructions get to execute
    - More than one insn. can be ready
    - Number of functional units and memory ports are limited
Interaction with Execution

Payload RAM

Select Logic

D S_L S_R A
Simple Scheduler Pipeline

A: Select → Payload → Execute
   - tag broadcast
   - result broadcast

B: Wakeup → Capture
   - enable capture on tag match

C: Wakeup → Capture
   - tag broadcast
   - enable capture

Cycle i → Cycle i+1

Very long clock cycle
Deeper Scheduler Pipeline

A: Select | Payload | Execute
  tag broadcast

B: Wakeup
  enable capture
  Capture

C: Wakeup
  enable capture
  Capture

Cycle i | Cycle i+1 | Cycle i+2 | Cycle i+3

Faster, but Capture & Payload on same cycle
Very Deep Scheduler Pipeline

A: Select → Payload → Execute
B: Select → Select → Payload → Execute
C: Wakeup
D: Wakeup

A&B both ready, only A selected, B bids again
A→C and C→D must be bypassed, B→D OK without bypass

Dependent instructions can’t execute back-to-back
Non-Data-Capture Scheduler

- Fetch & Dispatch
- Scheduler
- ARF
- PRF
- Functional Units
  - Physical register update

- Fetch & Dispatch
- Scheduler
- Unified PRF
- Functional Units
  - Physical register update
Pipeline Timing

Data-Capture

Select → Payload → Execute

Wakeup → Select → Payload → Execute

Non-Data-Capture

Select → Payload → Read Operands from PRF → Execute

Wakeup → Select → Payload → Read Operands from PRF → Execute

“Skip” Cycle

Substantial increase in schedule-to-execute latency
Handling Multi-Cycle Instructions

- Add R1 = R2 + R3
- Xor R4 = R1 ^ R5
- Mul R1 = R2 × R3
- Add R4 = R1 + R5

Instructions can’t execute *too early*
Non-Deterministic Latencies

- Real situations have unknown latency
  - Load instructions
    - Latency $\in \{\text{L1\_lat, L2\_lat, L3\_lat, DRAM\_lat}\}$
    - DRAM\_lat is not a constant either, queuing delays
  - Architecture specific cases
    - PowerPC 603 has “early out” for multiplication
    - Intel Core 2’s has early out divider also
Load-Hit Speculation

- Caches work pretty well
  - Hit rates are high (otherwise we wouldn’t use caches)
  - Assume all loads hit in the cache

What to do on a cache miss?
Simple Select Logic

Scheduler Entries

S entries yields $O(S)$ gate delay

$\text{Grant}_0 = 1$

$\text{Grant}_1 = \neg \text{Bid}_0$

$\text{Grant}_2 = \neg \text{Bid}_0 \land \neg \text{Bid}_1$

$\text{Grant}_3 = \neg \text{Bid}_0 \land \neg \text{Bid}_1 \land \neg \text{Bid}_2$

$\text{Grant}_{n-1} = \neg \text{Bid}_0 \land \ldots \land \neg \text{Bid}_{n-2}$

$O(\log S)$ gates
Implementing Oldest First Select

Must broadcast grant age to instructions
Problems in N-of-M Select

\[ O(\log M) \text{ gate delay / select} \]

\[ O(N \log M) \text{ delay} \]
Select Binding

Not-Quite-Oldest-First:
Ready insns are aged 2, 3, 4
Issued insns are 2 and 4

Wasted Resources:
3 instructions are ready
Only 1 gets to issue
Execution Ports

• Divide functional units into P groups
  – Called “ports”
• Area only $O(P^2M \log M)$, where $P << F$
• Logic for tracking bids and grants less complex (deals with P sets)
Decentralized RS

• Natural split: INT vs. FP

Often implies non-ROB based physical register file:

One “unified” integer PRF, and one “unified” FP PRF, each managed separately with their own free lists
Higher Complexity not Worth Effort

- Performance

- Made sense to go Superscalar/OOO: good ROI

- Very little gain for substantial effort

- “Effort”

- Scalar In-Order
- Moderate-Pipe Superscalar/OOO
- Very-Deep-Pipe Aggressive Superscalar/OOO
**SMP Machines**

- **SMP** = Symmetric Multi-Processing
  - Symmetric = All CPUs have “equal” access to memory
- OS seems multiple CPUs
  - Runs one process (or thread) on each CPU
MP Workload Benefits

- Task A
- Task B

3-wide OOO CPU
4-wide OOO CPU
3-wide OOO CPU
2-wide OOO CPU
2-wide OOO CPU

Benefit

runtime
... If Only One Task Available

- 3-wide OOO CPU
- 4-wide OOO CPU
- 3-wide OOO CPU
- 2-wide OOO CPU

Task A

Benefit

No benefit over 1 CPU

Performance degradation!

Idle

runtime
Chip-Multiprocessing (CMP)

- Simple SMP on the same chip
  - CPUs now called “cores” by hardware designers
  - OS designers still call these “CPUs”
On-chip Interconnects (1/4)

• Today, (Core+L1+L2) = “core”
  – (L3+I/O+Memory) = “uncore”

• How to interconnect multiple “core”s to “uncore”?

• Possible topologies
  – Bus
  – Crossbar
  – Ring
  – Mesh
  – Torus
On-chip Interconnects (2/4)

• Possible topologies
  – Bus
  – Crossbar
  – Ring
  – Mesh
  – Torus
On-chip Interconnects (3/4)

• Possible topologies
  – Bus
  – Crossbar
  – Ring
  – Mesh
  – Torus

• 3 ports per switch
• Simple and cheap
• Can be bi-directional to reduce latency
On-chip Interconnects (4/4)

• Possible topologies
  – Bus
  – Crossbar
  – Mesh
  – Torus

• Up to 5 ports per switch

*Tiled* organization combines core and cache
Multi-Threading

- Uni-Processor: 4-6 wide, lucky if you get 1-2 IPC
  - Poor utilization of transistors
- SMP: 2-4 CPUs, but need independent threads
  - Poor utilization as well (if limited tasks)
- \{Coarse-Grained, Fine-Grained, Simultaneous\}-MT
  - Use single large uni-processor as a multi-processor
    - Core provide multiple hardware contexts (threads)
      - Per-thread PC
      - Per-thread ARF (or map table)
  - Each core appears as multiple CPUs
    - OS designers still call these “CPUs”
Scalar Pipeline

Time

Dependencies limit functional unit utilization
Superscalar Pipeline

Higher performance than scalar, but lower utilization
Limited utilization when running one thread
Coarse-Grained Multithreading

<table>
<thead>
<tr>
<th>Time</th>
<th>Hardware Context Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Only good for long latency ops (i.e., cache misses)
Fine-Grained Multithreading

Time

Saturated workload -> Lots of threads

Unsaturated workload -> Lots of stalls

Intra-thread dependencies still limit performance
Simultaneous Multithreading

Max utilization of functional units
Paired vs. Separate Processor/Memory?

- **Separate CPU/memory**
  - *Uniform memory access* (UMA)
    - Equal latency to memory
  - Low peak performance

- **Paired CPU/memory**
  - *Non-uniform memory access* (NUMA)
    - Faster local memory
    - Data placement matters
  - High peak performance
Issues for Shared Memory Systems

• Two big ones
  – Cache coherence
  – Memory consistency model

• Closely related

• Often confused
Cache Coherence: The Problem

- Variable A initially has value 0
- P1 stores value 1 into A
- P2 loads A from memory and sees old value 0

Need to do something to keep P2's cache coherent
Simple MSI Protocol

Cache Actions:
- Load, Store, Evict

Bus Actions:
- BusRd, BusRdX
- BusInv, BusWB, BusReply

Usable coherence protocol
Coherence vs. Consistency

• Coherence concerns only one memory location
• Consistency concerns ordering for all locations

• A Memory System is Coherent if
  – Can serialize all operations to that location
    • Operations performed by any core appear in program order
  – Read returns value written by last store to that location

• A Memory System is Consistent if
  – It follows the rules of its Memory Model
    • Operations on memory locations appear in some defined order
Sequential Consistency (SC)

Processors issue memory ops in program order. P1, P2, and P3 are processors with memory operations. The switch randomly set after each memory op defines a single sequential order among all ops. Memory

Defines Single Sequential Order Among All Ops.
Mutex Example w/ Store Buffer

**P1**
lockA: A = 1;
if (B != 0)
    { A = 0; goto lockA; }
/* critical section*/
A = 0;

**P2**
lockB: B=1;
if (A != 0)
    { B = 0; goto lockB; }
/* critical section*/
B = 0;

Does not work
Relaxed Consistency Models

• Sequential Consistency (SC):
  – $R \rightarrow W$, $R \rightarrow R$, $W \rightarrow R$, $W \rightarrow W$

• Total Store Ordering (TSO) relaxes $W \rightarrow R$
  – $R \rightarrow W$, $R \rightarrow R$, $W \rightarrow W$

• Partial Store Ordering relaxes $W \rightarrow W$ (coalescing WB)
  – $R \rightarrow W$, $R \rightarrow R$

• Weak Ordering or Release Consistency (RC)
  – All ordering explicitly declared
    • Use $fences$ to define boundaries
    • Use $acquire$ and $release$ to force flushing of values
Good Luck!