ECE/CS 757: Advanced Computer Architecture II

Instructor: Mikko H Lipasti

Spring 2013

University of Wisconsin-Madison

Lecture notes based on slides created by John Shen, Mark Hill, David Wood, Guri Sohi, Jim Smith, Natalie Enright Jerger, Michel Dubois, Murali Annavaram, Per Stenström and probably others
Review of 752

- Iron law
- Beyond pipelining
- Superscalar challenges
  - Instruction flow
  - Register data flow
  - Memory Dataflow
- Modern memory interface
Iron Law

Processor Performance = \frac{Time}{Program}

= \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Time}{Cycle}

(code size) (CPI) (cycle time)

Architecture --> Implementation --> Realization

Compiler Designer  Processor Designer  Chip Designer
Iron Law

• Instructions/Program
  – Instructions executed, not static code size
  – Determined by algorithm, compiler, ISA

• Cycles/Instruction
  – Determined by ISA and CPU organization
  – Overlap among instructions reduces this term

• Time/cycle
  – Determined by technology, organization, clever circuit design
Our Goal

• Minimize time, which is the product, NOT isolated terms

• Common error to miss terms while devising optimizations
  – E.g. ISA change to decrease instruction count
  – BUT leads to CPU organization which makes clock slower

• Bottom line: terms are inter-related
Pipelined Design

• Motivation:
  – Increase throughput with little increase in hardware.

  Bandwidth or Throughput = Performance

• Bandwidth (BW) = no. of tasks/unit time
• For a system that operates on one task at a time:
  – BW = 1/delay (latency)
• BW can be increased by pipelining if many operands exist which need the same operation, i.e. many repetitions of the same task are to be performed.
• Latency required for each task remains the same or may even increase slightly.
Ideal Pipelining

• Bandwidth increases linearly with pipeline depth
• Latency increases by latch delays

BW = ~(1/n)

BW = ~(2/n)

BW = ~(3/n)
Example: Integer Multiplier

- 16x16 combinational multiplier
  - ISCAS-85 C6288 standard benchmark
  - Tools: Synopsys DC/LSI Logic 110nm gflxp ASIC

[Source: J. Hayes, Univ. of Michigan]
Example: Integer Multiplier

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Delay</th>
<th>MPS</th>
<th>Area (FF/wiring)</th>
<th>Area Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinational</td>
<td>3.52ns</td>
<td>284</td>
<td>7535 (--/1759)</td>
<td></td>
</tr>
<tr>
<td>2 Stages</td>
<td>1.87ns</td>
<td>534 (1.9x)</td>
<td>8725 (1078/1870)</td>
<td>16%</td>
</tr>
<tr>
<td>4 Stages</td>
<td>1.17ns</td>
<td>855 (3.0x)</td>
<td>11276 (3388/2112)</td>
<td>50%</td>
</tr>
<tr>
<td>8 Stages</td>
<td>0.80ns</td>
<td>1250 (4.4x)</td>
<td>17127 (8938/2612)</td>
<td>127%</td>
</tr>
</tbody>
</table>

- Pipeline efficiency
  - 2-stage: nearly double throughput; marginal area cost
  - 4-stage: 75% efficiency; area still reasonable
  - 8-stage: 55% efficiency; area more than doubles
- Tools: Synopsys DC/LSI Logic 110nm gflxp ASIC
Pipelining Idealisms

• Uniform subcomputations
  – Can pipeline into stages with equal delay
  – Balance pipeline stages

• Identical computations
  – Can fill pipeline with identical work
  – Unify instruction types

• Independent computations
  – No relationships between work units
  – Minimize pipeline stalls

• Are these practical?
  – No, but can get close enough to get significant speedup
Instruction Pipelining

• The “computation” to be pipelined.
  – Instruction Fetch (IF)
  – Instruction Decode (ID)
  – Operand(s) Fetch (OF)
  – Instruction Execution (EX)
  – Operand Store (OS)
  – Update Program Counter (PC)
• Based on “obvious” subcomputations
Pipelining Idealisms

☑ Uniform subcomputations
  - Can pipeline into stages with equal delay
  - Balance pipeline stages

☑ Identical computations
  - Can fill pipeline with identical work
  - Unify instruction types (example in 752 notes)

• Independent computations
  - No relationships between work units
  - Minimize pipeline stalls
The implied sequential precedences are an overspecification. It is sufficient but not necessary to ensure program correctness.
Program Data Dependences

• True dependence (RAW)
  – j cannot execute until i produces its result

• Anti-dependence (WAR)
  – j cannot write its result until i has read its sources

• Output dependence (WAW)
  – j cannot write its result until i has written its result
Control Dependences

• Conditional branches
  – Branch must execute to determine which instruction to fetch next
  – Instructions following a conditional branch are control dependent on the branch instruction
Resolution of Pipeline Hazards

• Pipeline hazards
  – Potential violations of program dependences
  – Must ensure program dependences are not violated

• Hazard resolution
  – Static: compiler/programmer guarantees correctness
  – Dynamic: hardware performs checks at runtime

• Pipeline interlock
  – Hardware mechanism for dynamic hazard resolution
  – Must detect and enforce dependences at runtime
IBM RISC Experience
[Agerwala and Cocke 1987]

• Internal IBM study: Limits of a scalar pipeline?
• Memory Bandwidth
  – Fetch 1 instr/cycle from I-cache
  – 40% of instructions are load/store (D-cache)
• Code characteristics (dynamic)
  – Loads – 25%
  – Stores 15%
  – ALU/RR – 40%
  – Branches – 20%
    • 1/3 unconditional (always taken)
    • 1/3 conditional taken, 1/3 conditional not taken
IBM Experience

• Cache Performance
  – Assume 100% hit ratio (upper bound)
  – Cache latency: $I = D = 1$ cycle default

• Load and branch scheduling
  – Loads
    • 25% cannot be scheduled (delay slot empty)
    • 65% can be moved back 1 or 2 instructions
    • 10% can be moved back 1 instruction
  – Branches
    • Unconditional – 100% schedulable (fill one delay slot)
    • Conditional – 50% schedulable (fill one delay slot)
CPI Optimizations

• Goal and impediments
  – CPI = 1, prevented by pipeline stalls

• No cache bypass of RF, no load/branch scheduling
  – Load penalty: 2 cycles: 0.25 x 2 = 0.5 CPI
  – Branch penalty: 2 cycles: 0.2 x 2/3 x 2 = 0.27 CPI
  – Total CPI: 1 + 0.5 + 0.27 = 1.77 CPI

• Bypass, no load/branch scheduling
  – Load penalty: 1 cycle: 0.25 x 1 = 0.25 CPI
  – Total CPI: 1 + 0.25 + 0.27 = 1.52 CPI
More CPI Optimizations

• Bypass, scheduling of loads/branches
  – Load penalty:
    • 65% + 10% = 75% moved back, no penalty
    • 25% => 1 cycle penalty
    • \(0.25 \times 0.25 \times 1 = 0.0625\) CPI
  – Branch Penalty
    • 1/3 unconditional 100% schedulable => 1 cycle
    • 1/3 cond. not-taken, => no penalty (predict not-taken)
    • 1/3 cond. Taken, 50% schedulable => 1 cycle
    • 1/3 cond. Taken, 50% unschedulable => 2 cycles
    • \(0.25 \times [1/3 \times 1 + 1/3 \times 0.5 \times 1 + 1/3 \times 0.5 \times 2] = 0.167\)

• Total CPI: \(1 + 0.063 + 0.167 = 1.23\) CPI
Simplify Branches

- Assume 90% can be PC-relative
  - No register indirect, no register access
  - Separate adder (like MIPS R3000)
  - Branch penalty reduced
- Total CPI: $1 + 0.063 + 0.085 = 1.15$ CPI = 0.87 IPC

<table>
<thead>
<tr>
<th>PC-relative</th>
<th>Schedulable</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes (90%)</td>
<td>Yes (50%)</td>
<td>0 cycle</td>
</tr>
<tr>
<td>Yes (90%)</td>
<td>No (50%)</td>
<td>1 cycle</td>
</tr>
<tr>
<td>No (10%)</td>
<td>Yes (50%)</td>
<td>1 cycle</td>
</tr>
<tr>
<td>No (10%)</td>
<td>No (50%)</td>
<td>2 cycles</td>
</tr>
</tbody>
</table>
Limits of Pipelining

• IBM RISC Experience
  – Control and data dependences add 15%
  – Best case CPI of 1.15, IPC of 0.87
  – Deeper pipelines (higher frequency) magnify dependence penalties

• This analysis assumes 100% cache hit rates
  – Hit rates approach 100% for some programs
  – Many important programs have much worse hit rates
Processor Performance

Processor Performance = \frac{\text{Time}}{\text{Program}}

\begin{align*}
\text{Processor Performance} &= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}} \\
&= \frac{\text{(code size)}}{\text{(CPI)}} \times \frac{\text{(cycle time)}}
\end{align*}

- In the 1980’s (decade of pipelining):
  - CPI: 5.0 => 1.15
- In the 1990’s (decade of superscalar):
  - CPI: 1.15 => 0.5 (best case)
- In the 2000’s (decade of multicore):
  - Core CPI unchanged; chip CPI scales with #cores
## Limits on Instruction Level Parallelism (ILP)

<table>
<thead>
<tr>
<th>Study</th>
<th>ILP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weiss and Smith [1984]</td>
<td>1.58</td>
</tr>
<tr>
<td>Sohi and Vajapeyam [1987]</td>
<td>1.81</td>
</tr>
<tr>
<td>Tjaden and Flynn [1970]</td>
<td>1.86  (Flynn’s bottleneck)</td>
</tr>
<tr>
<td>Tjaden and Flynn [1973]</td>
<td>1.96</td>
</tr>
<tr>
<td>Uht [1986]</td>
<td>2.00</td>
</tr>
<tr>
<td>Smith et al. [1989]</td>
<td>2.00</td>
</tr>
<tr>
<td>Jouppi and Wall [1988]</td>
<td>2.40</td>
</tr>
<tr>
<td>Johnson [1991]</td>
<td>2.50</td>
</tr>
<tr>
<td>Acosta et al. [1986]</td>
<td>2.79</td>
</tr>
<tr>
<td>Wedig [1982]</td>
<td>3.00</td>
</tr>
<tr>
<td>Butler et al. [1991]</td>
<td>5.8</td>
</tr>
<tr>
<td>Melvin and Patt [1991]</td>
<td>6</td>
</tr>
<tr>
<td>Wall [1991]</td>
<td>7 (Jouppi disagreed)</td>
</tr>
<tr>
<td>Kuck et al. [1972]</td>
<td>8</td>
</tr>
<tr>
<td>Riseman and Foster [1972]</td>
<td>51 (no control dependences)</td>
</tr>
<tr>
<td>Nicolau and Fisher [1984]</td>
<td>90 (Fisher’s optimism)</td>
</tr>
</tbody>
</table>
Superscalar Proposal

• Go beyond single instruction pipeline, achieve IPC > 1
• Dispatch multiple instructions per cycle
• Provide more generally applicable form of concurrency (not just vectors)
• Geared for sequential code that is hard to parallelize otherwise
• Exploit fine-grained or instruction-level parallelism (ILP)
Limitations of Scalar Pipelines

• Scalar upper bound on throughput
  – IPC <= 1 or CPI >= 1

• Inefficient unified pipeline
  – Long latency for each instruction

• Rigid pipeline stall policy
  – One stalled instruction stalls all newer instructions
Parallel Pipelines

(a) No Parallelism  (b) Temporal Parallelism

(c) Spatial Parallelism  (d) Parallel Pipeline
Power4 Diversified Pipelines

I-Cache

Fetch Q

Decode

BR Scan

BR Predict

FP Issue Q

FP1 Unit

FP2 Unit

FX/LD 1 Issue Q

FX1 Unit

LD1 Unit

LD2 Unit

StQ

D-Cache

FX/LD 2 Issue Q

FX2 Unit

CR Unit

BR Unit

Reorder Buffer

BR/CR Issue Q

CR Unit

BR Unit
Rigid Pipeline Stall Policy

Bypassing of Stalled Instruction Not Allowed

Backward Propagation of Stalling
Dynamic Pipelines

IF

ID

RD

Dispatch Buffer

EX

ALU

MEM1

FP1

BR

MEM2

FP2

FP3

Reorder Buffer

WB

( in order )

( out of order )

( out of order )

( in order )
Limitations of Scalar Pipelines

• Scalar upper bound on throughput
  – IPC <= 1 or CPI >= 1
  – Solution: wide (superscalar) pipeline

• Inefficient unified pipeline
  – Long latency for each instruction
  – Solution: diversified, specialized pipelines

• Rigid pipeline stall policy
  – One stalled instruction stalls all newer instructions
  – Solution: Out-of-order execution, distributed execution pipelines
High-IPC Processor Evolution

Desktop/Workstation Market

- **Scalar RISC Pipeline**
  - 1980s: MIPS, SPARC, Intel 486
- **2-4 Issue In-order**
  - Early 1990s: IBM RIOS-I, Intel Pentium
- **Limited Out-of-Order**
  - Mid 1990s: PowerPC 604, Intel P6
- **Large ROB Out-of-Order**
  - 2000s: DEC Alpha 21264, IBM Power4/5, AMD K8

**1985 – 2005: 20 years, 100x frequency**

Mobile Market

- **Scalar RISC Pipeline**
  - 2002: ARM11
- **2-4 Issue In-order**
  - 2005: Cortex A8
- **Limited Out-of-Order**
  - 2009: Cortex A9
- **Large ROB Out-of-Order**
  - 2011: Cortex A15

**2002 – 2011: 10 years, 10x frequency**
Superscalar Overview

• Instruction flow
  – Branches, jumps, calls: predict target, direction
  – Fetch alignment
  – Instruction cache misses

• Register data flow
  – Register renaming: RAW/WAR/WAW

• Memory data flow
  – In-order stores: WAR/WAW
  – Store queue: RAW
  – Data cache misses
High-IPC Processor

Instruction Flow

Instruction Buffer

Branch Predictor

FETCH

DECODE

I-cache

D-cache

Branch Predictor

Instruction Flow

Memory Data Flow

Register Data Flow

Integer

Floating-point

Media

Memory

EXECUTE

(ROB)

Reorder Buffer (ROB)

COMMIT

D-cache

Store Queue

Reorder Buffer (ROB)

COMMIT

D-cache

Store Queue
Goal and Impediments

• Goal of Instruction Flow
  – Supply processor with maximum number of useful instructions every clock cycle

• Impediments
  – Branches and jumps
  – Finite I-Cache
    • Capacity
    • Bandwidth restrictions
## Limits on Instruction Level Parallelism (ILP)

<table>
<thead>
<tr>
<th>Study</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weiss and Smith [1984]</td>
<td>1.58</td>
</tr>
<tr>
<td>Sohi and Vajapeyam [1987]</td>
<td>1.81</td>
</tr>
<tr>
<td>Tjaden and Flynn [1970]</td>
<td>1.86 (Flynn’s bottleneck)</td>
</tr>
<tr>
<td>Tjaden and Flynn [1973]</td>
<td>1.96</td>
</tr>
<tr>
<td>Uht [1986]</td>
<td>2.00</td>
</tr>
<tr>
<td>Smith et al. [1989]</td>
<td>2.00</td>
</tr>
<tr>
<td>Jouppi and Wall [1988]</td>
<td>2.40</td>
</tr>
<tr>
<td>Johnson [1991]</td>
<td>2.50</td>
</tr>
<tr>
<td>Acosta et al. [1986]</td>
<td>2.79</td>
</tr>
<tr>
<td>Wedig [1982]</td>
<td>3.00</td>
</tr>
<tr>
<td>Butler et al. [1991]</td>
<td>5.8</td>
</tr>
<tr>
<td>Melvin and Patt [1991]</td>
<td>6</td>
</tr>
<tr>
<td>Wall [1991]</td>
<td>7 (Jouppi disagreed)</td>
</tr>
<tr>
<td>Kuck et al. [1972]</td>
<td>8</td>
</tr>
<tr>
<td>Riseman and Foster [1972]</td>
<td>51 (no control dependences)</td>
</tr>
<tr>
<td>Nicolau and Fisher [1984]</td>
<td>90 (Fisher’s optimism)</td>
</tr>
</tbody>
</table>
Speculative Execution

• Riseman & Foster showed potential
  – But no idea how to reap benefit
• 1979: Jim Smith patents branch prediction at Control Data
  – Predict current branch based on past history
• Today: virtually all processors use branch prediction
Instruction Flow

Objective: Fetch multiple instructions per cycle

• Challenges:
  – Branches: unpredictable
  – Branch targets misaligned
  – Instruction cache misses

• Solutions
  – Prediction and speculation
  – High-bandwidth fetch logic
  – Nonblocking cache and prefetching
Disruption of Instruction Flow

Instruction/Decode Buffer

Decode

Dispatch Buffer

Dispatch

Reservation/Store Buffer

Complete

Retire
Branch Prediction

• Target address generation → Target Speculation
  – Access register:
    • PC, General purpose register, Link register
  – Perform calculation:
    • +/- offset, autoincrement

• Condition resolution → Condition speculation
  – Access register:
    • Condition code register, General purpose register
  – Perform calculation:
    • Comparison of data register(s)
Target Address Generation

- Fetch
- Decode Buffer
- Decode
- Dispatch Buffer
- Dispatch
- Store Buffer
- Complete
- Retire
- Reservation Stations
- Completion Buffer
- Complete
- Store Buffer
- Retire

PC-rel.
Reg. ind.
Reg. ind. with offset
Issue
Branch
Execute
Finish

Mikko Lipasti
University of Wisconsin
Branch Instruction Speculation

Prediction

Spec. target

Branch Predictor (using a BTB)

Spec. cond.

BTB update (target addr. and history)

Fetch

Decode Buffer

Decode

Dispatch Buffer

Dispatch

Reservation Stations

Issue

Branch

Execute

Finish

Completion Buffer

FA-mux

PC(seq.) = FA (fetch address)

to I-cache

PC(seq.)

Fetch

Decode

Dispatch

Completion Buffer

Mikko Lipasti

University of Wisconsin
Hardware Smith Predictor

- Widely employed: Intel Pentium, PowerPC 604, MIPS R10000, etc.
Cortex A15: Bi-Mode Predictor

- PHT partitioned into T/NT halves
  - Selector chooses source
- Reduces negative interference, since most entries in PHT$_0$ tend towards NT, and most entries in PHT$_1$ tend towards T

15% of A15 Core Power!
Branch Target Prediction

- Does not work well for function/procedure returns
- Does not work well for virtual functions, switch statements

Mikko Lipasti - University of Wisconsin
Branch Speculation

- **Leading Speculation**
  - Done during the Fetch stage
  - Based on potential branch instruction(s) in the current fetch group
- **Trailing Confirmation**
  - Done during the Branch Execute stage
  - Based on the next Branch instruction to finish execution
Branch Speculation

- Start new correct path
  - Must remember the alternate (non-predicted) path
- Eliminate incorrect path
  - Must ensure that the mis-speculated instructions produce no side effects
Mis-speculation Recovery

• **Start new correct path**
  1. Update PC with computed branch target (if predicted NT)
  2. Update PC with sequential instruction address (if predicted T)
  3. Can begin speculation again at next branch

• **Eliminate incorrect path**
  1. Use tag(s) to **deallocate** resources occupied by speculative instructions
  2. **Invalidate** all instructions in the decode and dispatch buffers, as well as those in reservation stations
Parallel Decode

• Primary Tasks
  – Identify individual instructions (!)
  – Determine instruction types
  – Determine dependences between instructions

• Two important factors
  – Instruction set architecture
  – Pipeline width
Pentium Pro Fetch/Decode

Macroinstruction bytes from IFU

Instruction buffer (16 bytes)

To next address calculation

uROM

Decoder 0

Decoder 1

Decoder 2

4 uops

1 uop

1 uop

Branch address calculation

uop queue (6)

Mikko Lipasti - University of Wisconsin
Dependence Checking

- Trailing instructions in fetch group
  - Check for dependence on leading instructions
Summary: Instruction Flow

• Fetch group alignment

• Target address generation
  – Branch target buffer

• Branch condition prediction

• Speculative execution
  – Tagging/tracking instructions
  – Recovering from mispredicted branches

• Decoding in parallel
High-IPC Processor

Instruction Flow

Register Data Flow

Memory Data Flow
Register Data Flow

• Parallel pipelines
  – Centralized instruction fetch
  – Centralized instruction decode

• Diversified execution pipelines
  – Distributed instruction execution

• Data dependence linking
  – Register renaming to resolve true/false dependences
  – Issue logic to support out-of-order issue
  – Reorder buffer to maintain precise state
Issue Queues and Execution Lanes

ARM Cortex A15

Source: theregister.co.uk
Program Data Dependences

- True dependence (RAW)
  - j cannot execute until i produces its result
- Anti-dependence (WAR)
  - j cannot write its result until i has read its sources
- Output dependence (WAW)
  - j cannot write its result until i has written its result

\[ D(i) \cap R(j) \neq \phi \]

\[ R(i) \cap D(j) \neq \phi \]

\[ D(i) \cap D(j) \neq \phi \]
Register Data Dependences

• Program data dependences cause hazards
  – True dependences (RAW)
  – Antidependences (WAR)
  – Output dependences (WAW)

• When are registers read and written?
  – Out of program order!
  – Hence, any and all of these can occur

• Solution to all three: register renaming
Register Renaming: WAR/WAW

- Widely employed (Core i7, Cortex A15, ...)
- Resolving WAR/WAW:
  - Each register write gets unique “rename register”
  - Writes are committed in program order at Writeback
  - WAR and WAW are not an issue
    - All updates to “architected state” delayed till writeback
    - Writeback stage always later than read stage
  - Reorder Buffer (ROB) enforces in-order writeback

| Add R3 <= … | P32 <= … |
| Sub R4 <= … | P33 <= … |
| And R3 <= … | P35 <= … |
Register Renaming: RAW

• In order, at dispatch:
  – Source registers checked to see if “in flight”
    • Register map table keeps track of this
    • If not in flight, can be read from the register file
    • If in flight, look up “rename register” tag (IOU)
  – Then, allocate new register for register write

Add R3 <= R2 + R1    P32 <= P2 + P1
Sub R4 <= R3 + R1    P33 <= P32 + P1
And R3 <= R4 & R2    P35 <= P33 + P2
Register Renaming: RAW

- Advance instruction to instruction queue
  - Wait for rename register tag to trigger issue

- Issue queue/reservation station enables out-of-order issue
  - Newer instructions can bypass stalled instructions
Physical Register File

- Used in the MIPS R10000 pipeline, Intel Sandy/Ivybridge
- All registers in one place
  - Always accessed right before EX stage
  - No copying to real register file
Managing Physical Registers

- What to do when all physical registers are in use?
  - Must release them somehow to avoid stalling
  - Maintain *free list* of “unused” physical registers

- Release when no more uses are possible
  - Sufficient: next write commits
High-IPC Processor

Instruction Flow

Register Data Flow

Memory Data Flow

Mikko Lipasti-University of Wisconsin
Memory Data Flow

- Resolve WAR/WAW/RAW memory dependences
  - MEM stage can occur out of order
- Provide high bandwidth to memory hierarchy
  - Non-blocking caches
Memory Data Dependences

• Besides branches, long memory latencies are one of the biggest performance challenges today.

• To preserve sequential (in-order) state in the data caches and external memory (so that recovery from exceptions is possible) stores are performed in order. This takes care of antidependences and output dependences to memory locations.

• However, loads can be issued out of order with respect to stores if the out-of-order loads check for data dependences with respect to previous, pending stores.

<table>
<thead>
<tr>
<th>WAW</th>
<th>WAR</th>
<th>RAW</th>
</tr>
</thead>
<tbody>
<tr>
<td>store X</td>
<td>load X</td>
<td>store X</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>store X</td>
<td>store X</td>
<td>load X</td>
</tr>
</tbody>
</table>
Memory Data Dependences

• **“Memory Aliasing”** = Two memory references involving the same memory location (collision of two memory addresses).

• **“Memory Disambiguation”** = Determining whether two memory references will alias or not (whether there is a dependence or not).

• **Memory Dependency Detection:**
  – Must compute effective addresses of both memory references
  – Effective addresses can depend on run-time data and other instructions
  – Comparison of addresses require much wider comparators

Example code:

(1) STORE V
(2) ADD
(3) LOAD W
(4) LOAD X
(5) LOAD V
(6) ADD
(7) STORE W

RAW, WAR
Memory Data Dependences

- **WAR/WAW**: stores commit in order
  - Hazards not possible.
- **RAW**: loads must check pending stores
  - Store queue keeps track of pending store addresses
  - Loads check against these addresses
  - Similar to register bypass logic
  - Comparators are 32 or 64 bits wide (address size)
- **Major source of complexity in modern designs**
  - Store queue lookup is position-based
  - What if store address is not yet known? Stall all trailing ops
Optimizing Load/Store Disambiguation

- Non-speculative load/store disambiguation
  1. Loads wait for addresses of all prior stores
  2. Full address comparison
  3. Bypass if no match, forward if match

(1) can limit performance:

```plaintext
load r5, MEM[r3] ← cache miss
store r7, MEM[r5] ← RAW for agen, stalled
...
load r8, MEM[r9] ← independent load stalled
```
Speculative Disambiguation

- What if aliases are rare?
  1. Loads don’t wait for addresses of all prior stores
  2. Full address comparison of stores that are ready
  3. Bypass if no match, forward if match
  4. Check all store addresses when they commit
     - No matching loads – speculation was correct
     - Matching unbypassed load – incorrect speculation
  5. Replay starting from incorrect load
Speculative Disambiguation: Load Bypass

- i1 and i2 issue in program order
- i2 checks store queue (no match)
Speculative Disambiguation: Load Forward

- i1 and i2 issue in program order
- i2 checks store queue (match=>forward)
Speculative Disambiguation: Safe Speculation

i1: st R3, MEM[R8]: ??
i2: ld R9, MEM[R4]: ??

Agen
Mem

Load Queue
i2: ld R9, MEM[R4]: x400C

Store Queue
i1: st R3, MEM[R8]: x800A

Reorder Buffer

• i1 and i2 issue out of program order
• i1 checks load queue at commit (no match)
Speculative Disambiguation: Violation

- i1 and i2 issue out of program order
- i1 checks load queue at commit (match)
  - i2 marked for replay
Use of Prediction

- If aliases are rare: static prediction
  - Predict no alias every time
    - Why even implement forwarding? PowerPC 620 doesn’t
  - Pay misprediction penalty rarely
- If aliases are more frequent: dynamic prediction
  - Use PHT-like history table for loads
    - If alias predicted: delay load
    - If aliased pair predicted: forward from store to load
      - More difficult to predict pair [store sets, Alpha 21264]
  - Pay misprediction penalty rarely
- Memory cloaking [Moshovos, Sohi]
  - Predict load/store pair
  - Directly copy store data register to load target register
  - Reduce data transfer latency to absolute minimum
Load/Store Disambiguation Discussion

- **RISC ISA:**
  - Many registers, most variables allocated to registers
  - Aliases are rare
  - Most important to not delay loads (bypass)
  - Alias predictor may/may not be necessary

- **CISC ISA:**
  - Few registers, many operands from memory
  - Aliases much more common, forwarding necessary
  - Incorrect load speculation should be avoided
  - If load speculation allowed, predictor probably necessary

- **Address translation:**
  - Can’t use virtual address (must use physical)
  - Wait till after TLB lookup is done
  - Or, use subset of untranslated bits (page offset)
    - Safe for proving inequality (bypassing OK)
    - Not sufficient for showing equality (forwarding not OK)
Increasing Memory Bandwidth


RS's → Branch, Integer, Integer, Float.- Point

Reorder Buff. → Complete

Store Buff. → Retire

Load/ Store

Data Cache

Expensive to duplicate

Complex, concurrent FSMs

Missed loads
Coherent Memory Interface

• Load Queue
  – Tracks inflight loads for aliasing, coherence

• Store Queue
  – Defers stores until commit, tracks aliasing

• Storethrough Queue or Write Buffer or Store Buffer
  – Defers stores, coalesces writes, must handle RAW

• MSHR
  – Tracks outstanding misses, enables lockup-free caches [Kroft ISCA 91]

• Snoop Queue
  – Buffers, tracks incoming requests from coherent I/O, other processors

• Fill Buffer
  – Works with MSHR to hold incoming partial lines

• Writeback Buffer
  – Defers writeback of evicted line (demand miss handled first)
Split Transaction Bus

(a) Simple bus with atomic transactions

(b) Split-transaction bus with separate requests and responses

- “Packet switched” vs. “circuit switched”
- Release bus after request issued
- Allow multiple concurrent requests to overlap memory latency
- Complicates control, arbitration, and coherence protocol
  - Transient states for pending blocks (e.g. “req. issued but not completed”)
Memory Consistency

- How are memory references from different processors interleaved?
- If this is not well-specified, synchronization becomes difficult or even impossible
  - ISA must specify consistency model
- Common example using Dekker’s algorithm for synchronization
  - If load reordered ahead of store (as we assume for a baseline OOO CPU)
  - Both Proc0 and Proc1 enter critical section, since both observe that other’s lock variable (A/B) is not set
- If consistency model allows loads to execute ahead of stores, Dekker’s algorithm no longer works
  - Common ISAs allow this: IA-32, PowerPC, SPARC, Alpha
Sequential Consistency [Lamport 1979]

- Processors treated as if they are interleaved processes on a single time-shared CPU
- All references must fit into a total global order or interleaving that does not violate any CPU’s program order
  - Otherwise sequential consistency not maintained
- Now Dekker’s algorithm will work
- Appears to preclude any OOO memory references
  - Hence precludes any real benefit from OOO CPUs
High-Performance Sequential Consistency

- Coherent caches isolate CPUs if no sharing is occurring
  - Absence of coherence activity means CPU is free to reorder references
- Still have to order references with respect to misses and other coherence activity (snoops)
- Key: use speculation
  - Reorder references speculatively
  - Track which addresses were touched speculatively
  - Force replay (in order execution) of such references that collide with coherence activity (snoops)
High-Performance Sequential Consistency

- Load queue records all speculative loads
- Bus writes/upgrades are checked against LQ
- Any matching load gets marked for replay
- At commit, loads are checked and replayed if necessary
  - Results in machine flush, since load-dependent ops must also replay
- Practically, conflicts are rare, so expensive flush is OK
Maintaining Precise State

- Out-of-order execution
  - ALU instructions
  - Load/store instructions

- In-order completion/retirement
  - Precise exceptions

- Solutions
  - Reorder buffer retires instructions in order
  - Store queue retires stores in order
  - Exceptions can be handled at any instruction boundary by reconstructing state out of ROB/SQ
  - Load queue monitors remote stores
Superscalar Summary
Landscape of Microprocessor Families

**Performance**$_{CPU} = \frac{Frequency}{PathLength \times CPI}$

**Data source** www.spec.org
Review of 752

✓ Iron law
✓ Beyond pipelining
✓ Superscalar challenges
  ✓ Instruction flow
  ✓ Register data flow
  ✓ Memory Dataflow
✓ Modern memory interface
• What was not covered
  – Memory hierarchy (review later)
  – Virtual memory (read 4.4 in book)
  – Power & reliability (read ch. 2 in book)
  – Many implementation/design details
  – Etc.
  – Multithreading (coming up next)