Superscalar Organization

Prof. Mikko H. Lipasti
University of Wisconsin-Madison

Lecture notes based on notes by John P. Shen
Updated by Mikko Lipasti
Limitations of Scalar Pipelines

- Scalar upper bound on throughput
  - $\text{IPC} \leq 1$ or $\text{CPI} \geq 1$

- 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
Intel Pentium Parallel Pipeline

IF

D1

D2

EX

WB

U - Pipe

V - Pipe
Diversified Pipelines

IF | ID | RD | EX | WB

IF
ID
RD
EX
WB

ALU | MEM1 | FP1 | BR

MEM2 | FP2

FP3

• • •
Power4 Diversified Pipelines

I-Cache

Fetch Q

Decode

FP
FP Issue Q
FP1 Unit
FP2 Unit

FX/LD 1
FX Issue Q
FX1 Unit
LD1 Unit

FX/LD 2
FX Issue Q
LD2 Unit
FX2 Unit

BR/CR
BR Issue Q
CR Unit
BR Unit

Reorder Buffer

PC

BR Scan

BR Predict

D-Cache

StQ
Rigid Pipeline Stall Policy

Bypassing of Stalled Instruction Not Allowed

Backward Propagation of Stalling
Dynamic Pipelines

IF  ID  RD  WB  ALU  MEM1  FP1  BR  MEM2  FP2  FP3  EX  Dispatch  Buffer  Reorder  Buffer  WB

(in order)  (out of order)  (in order)  (out of order)  (out of order)
Interstage Buffers

(a)

Stage $i$
Buffer (1)
Stage $i+1$

(b)

Stage $i$
Buffer ($n$)
Stage $i+1$

Stage $i$
Buffer ($\geq n$)
Stage $i+1$

(c)
Limitations of Scalar Pipelines

• Scalar upper bound on throughput
  • IPC $\leq 1$ or CPI $\geq 1$
    • 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

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

1985 – 2005: 20 years, 100x frequency

Mobile Market

<table>
<thead>
<tr>
<th>Scalar RISC Pipeline</th>
<th>2-4 Issue In-order</th>
<th>Limited Out-of-Order</th>
<th>Large ROB Out-of-Order</th>
</tr>
</thead>
</table>

2002 – 2011: 10 years, 10x frequency

Mikko Lipasti - University of Wisconsin
High-IPC Processor

Instruction Flow

I-cache

DECODE

D-cache

Branch Predictor

FETCH

Instruction Buffer

Flow

Instruction

Buffer

Register

Data

Flow

Integer

Floating-point

Media

Memory

EXECUTE

(ROB)

Memory Data Flow

Reorder Buffer (ROB)

COMMIT

Store Queue

D-cache

Mikko Lipasti-University of Wisconsin

13
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

only 3 instructions fetched
I-Cache Organization

1 cache line = 1 physical row

1 cache line = 2 physical rows
Fetch Alignment

PC = XX000001

Row decoder

00

001

... 

111

00 01 10 11

Fetch group

Row width
RIOS-I Fetch Hardware

Odd directory sets A & B

Even directory sets A & B

TLB hit and buffer control logic

Instruction buffer network

Interlock, dispatch, branch, execution logic
Disruption of Instruction Flow
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)
Branch Instruction Speculation

Branch Predictor (using a BTB)

Prediction
Spec. target
Spec. cond.

BTB update (target addr. and history)

Fetch
PC(seq.) = FA (fetch address)

Decode Buffer

Dispatch Buffer
Decode

Dispatch

Issue

Reservation Stations

Execute

Finish

Completion Buffer
Hardware Smith Predictor

- Widely employed: Intel Pentium, PowerPC 604, MIPS R10000, etc.
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
Parallel Decode

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

• Two important factors
  • Instruction set architecture
  • Pipeline width
Predecoding in the AMD K5

- Now commonly employed in loop buffers, decoded instruction caches (uop caches)
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

Memory Data Flow

Register 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
Necessity of Instruction Dispatch
Centralized Reservation Station

Dispatch (issue)

Centralized reservation station (dispatch buffer)

Execute

Completion buffer
Issues in Instruction Execution

• Current trends
  • More parallelism ← bypassing very challenging
  • Deeper pipelines
  • More diversity

• Functional unit types
  • Integer
  • Floating point
  • Load/store ← most difficult to make parallel
  • Branch
  • Specialized units (media)
    • Very wide datapaths (256 bits/register or more)
Bypass Networks

- $O(n^2)$ interconnect from/to FU inputs and outputs
- Associative tag-match to find operands
- Solutions (hurt IPC, help cycle time)
  - Use RF only (IBM Power4) with no bypass network
  - Decompose into clusters (Alpha 21264)
Specialized units

- Intel Pentium 4 staggered adders
  - Fireball
- Run at 2x clock frequency
- Two 16-bit bitslices
- Dependent ops execute on half-cycle boundaries
- Full result not available until full cycle later
Specialized units

- FP multiply-accumulate
  \[ R = (A \times B) + C \]
- Doubles FLOP/instruction
- Lose RISC instruction format symmetry:
  - 3 source operands
- Widely used
Media Data Types

• Subword parallel vector extensions
  • Media data (pixels, quantized datum) often 1-2 bytes
  • Several operands packed in single 32/64b register
    {a,b,c,d} and {e,f,g,h} stored in two 32b registers
  • Vector instructions operate on 4/8 operands in parallel
  • New instructions, e.g. sum of abs. differences (SAD)
    \[ m_e = |a - e| + |b - f| + |c - g| + |d - h| \]

• Substantial throughput improvement
  • Usually requires hand-coding of critical loops
  • Shuffle ops (gather/scatter of vector elements)
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

Source: theregister.co.uk

Mikko Lipasti-University of Wisconsin
High-IPC Processor

Instruction Flow

Register Data Flow

Memory Data Flow
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

• WAR/WAW: stores commit in order
  • Hazards not possible.
• RAW: loads must check pending stores
  • Store queue keeps track of pending stores
  • Loads check against these addresses
  • Similar to register bypass logic
  • Comparators are 64 bits wide
  • Must consider position (age) of loads and stores

• Major source of complexity in modern designs
  • Store queue lookup is position-based
  • What if store address is not yet known?
Increasing Memory Bandwidth


RS’s
- Branch
- Integer
- Integer
- Float.-Point
- Load/Store
- Load/Store

Expensive to duplicate
Complex, concurrent FSMs
Missed loads

Data Cache → Complete → Retire → Store Buff. → Reorder Buff.
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
Summary: A High-IPC Processor

I-cache
FETCH
DECODE
COMMIT
D-cache
Branch Predictor
Instruction Flow

Integer
Floating-point
Media
EXECUTE

Memory
Reorder Buffer (ROB)
COMMIT
D-cache
Store Queue

Register Data Flow

Mikko Lipasti-University of Wisconsin
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