Advanced Computer Architecture

Part I: General Purpose
Exploiting ILP Dynamically

Paolo.Ienne@epfl.ch
EPFL - I&C - LAP
ILP? The Traditional Way

(Let’s Make It Fast!)
Speed: Main Goal in General Purpose Computer Architecture

- **Reduce delay per gate** → **Technology** (~ x1.2/year)
- **Improve architecture** → **Parallelism** (~ x1.3/year)

**Architectural and organisational ideas** are the main performance drivers since the mid-1980s.
Clock Rate Does Not Grow Much (Anymore!)
Sources of Parallelism

- **Bit-level**
  - Wider processor datapaths (8→16→32→64…)

- **Word-level (SIMD)**
  - Vector processors
  - Multimedia instruction sets (Intel’s MMX and SSE, Sun’s VIS, etc.)

- **Instruction-level**
  - Pipelining
  - Superscalar
  - VLIW and EPIC

- **Task- and Application-levels…**
  - Explicit parallel programming
  - Multiple threads
  - Multiple applications…

This lesson: **ILP = Instruction Level Parallelism**
Starting Point (Programmer Model)

- Sequential multicycle processor

- Cycles
  - 1:
  - 2:
  - 3:

Instructions
ILP?

Cycles
Instructions

?
First Step: Pipelining

Simplest form of **Instruction Level Parallelism** (ILP): Several instructions are being executed at once
Simple Pipeline
Simple Pipelining

Scope for parallelism is limited:

- **Control hazards** limit the usability of the pipeline
  - Must squash fetched and decoded instruction following a branch

- **Data hazards** limit the usability of the pipeline
  - Whenever the next instruction cannot be executed, the pipeline is stalled and no new useful work is done until the “problem” is solved (e.g., cache miss)

- **Rigid sequencing**
  - Special “slots” for everything even if sometimes useless (e.g., MEM before WB)
  - Every instruction must be coerced to the same framework
  - Structural hazards avoided “by construction”
Simple Pipeline with Forwarding
ILP So Far...

Cycles

Instructions

Pipelining

Standard

AdvCompArch — Exploiting ILP Dynamically

© Ienne 2003-12
Dynamic Scheduling: The Idea

- Extend the scope to extract parallelism:
  
  ```
  divd       $f0, $f2, $f4
  addd       $f10, $f0, $f8
  subd       $f12, $f8, $f14
  ```

- Why not to execute `subd` while `addd` waits for the result of `divd`?

- Relax a fundamental rule: instructions can be executed **out of program order**! (but the result must still be correct...)}
Break the Rigidity of the Basic Pipelining

- **Continue fetching and decoding** even and especially if one cannot execute previous instructions
- **Keep writeback waiting** if there is a structural hazard, without slowing down execution

Solution:

- **Split the tasks** in independent units/pipelines
  - Fetch and decode
  - Execute
  - Writeback

- Clearly, instructions will now produce results **out-of-order (OOO)**
Dynamically Scheduled Processor

Diagram showing the components of a dynamically scheduled processor, including F, D, RS, ALU, RF, ROB, MEM, and W. Arrows indicate the flow of data and control signals.
Reservation Stations

**Fetch&Decode Unit and Register File**
(1) Fetched operation descriptions and
(2a) known operands (from RF)
or (2b) source-operation tags

**All Execution Units**
(1) Tags of the executed operations
and (2) corresponding results

**Reservation Station**

**Dependent Execution Unit**
(1) Description of operations ready to execute
with (2) corresponding tags and (3) operands
Reservation Stations

- A reservation station checks that the operands are available (RAW) and that the Execution Unit is free (Structural Hazards), then starts execution.

<table>
<thead>
<tr>
<th>ALU1:</th>
<th>1</th>
<th>addd</th>
<th>–</th>
<th>MUL3</th>
<th>0xa87f b351</th>
<th>???</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU2:</td>
<td>1</td>
<td>subd</td>
<td>ALU1</td>
<td>–</td>
<td>???</td>
<td>0xffff fee1</td>
</tr>
<tr>
<td>ALU3:</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reservation Stations

- Unavailable operands are identified by the name of the entry in the reservation station in charge of the originating instruction
- Implicit register renaming, thus removing WAR and WAW hazards
- New results are seen at their inputs through special result bus(es)
- Writeback into the registers is not on the critical execution path
A Problem with Exceptions...

- Precise exceptions
  - Reordering at commit; user view is that of a fully in-order processor

- Imprecise exceptions
  - No reordering; out-of-order completion visible to the user
  - The OS/programmer must be aware of the problem and take appropriate action (e.g., execute again the complete subroutine where the problem occurred)

```
Precise

andi $t4, $t2, 0xff
andi $t5, $t3, 0xff
addi $v0, $v0, 1
srl $t2, $t2, 8

Imprecise

andi $t4, $t2, 0xff
andi $t5, $t3, 0xff
addi $v0, $v0, 3
srl $t2, $t2, 8
```
Out-of-order Commitment and Exceptions

- Exception handlers should know exactly where a problem has occurred, especially for nonterminating exceptions (e.g., page fault) so that they can correct the problem and resume exactly where the exception occurred.

- Of course, one assumes that everything before the faulty instruction was executed and everything after was not.

- With OOO dynamic execution it might no longer be true…
Reordering

- **Fundamental observation**: a processor can do *whatever it wants* provided that it gives the appearance of sequential execution (i.e., the architectural machine state is updated in program order)

- New phase: COMMIT or RETIRE or GRADUATE (besides the usual F, D, E, W)

- This observation is fundamental because it allows many techniques (precise interrupts, speculation, multithreading, etc.)
Dynamically Scheduled Processor

Instruction Fetch & Decode Unit

Reservation Stn. Reservation Stn. Reservation Stn. Reservation Stn.

ALU FP Unit Branch Unit Load/Store Unit

Commit Unit

Register File

Computation advances independently from machine state updates

Machine state is updated in order
Reorder Buffer

**Fetch&Decode Unit**
(1) Fetched-operation tags in original order, (2) destination register or address, and (3) PC

**All Execution Units**
(1) Tags of the executed operations and (2) corresponding results

**Commit Unit (Reorder Buffer)**

**Register File and Memory**
For each instruction, in the original fetch order, (1) destination register or address and (2) value to write
Reordering Instructions at Writeback

- Needs a reorder buffer in the Commit Unit

<table>
<thead>
<tr>
<th>PC</th>
<th>Tag</th>
<th>Register</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x1000 0004</td>
<td>—</td>
<td>$f3</td>
<td>0x627f ba5a</td>
</tr>
<tr>
<td>1</td>
<td>0x1000 0008</td>
<td>ALU1</td>
<td>0xa87f b351</td>
<td>???</td>
</tr>
<tr>
<td>1</td>
<td>0x1000 000c</td>
<td>MUL2</td>
<td>$f5</td>
<td>???</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Origins of Reordering

- Paper Smith & Pleszkun on precise interrupts, 1988
Second Step: Dynamic Scheduling

- Tangible amount of ILP now possible
- What’s next?!
ILP So Far...

- Pipelining
- Dynamic Scheduling

Cycles
Instructions
Superscalar Execution

- Why not more than one instruction beginning execution (issued) per cycle?
- Key requirements are
  - Fetching more instruction in a cycle: no big difficulty provided that the instruction cache can sustain the bandwidth
  - Decide on data and control dependencies: dynamic scheduling already takes care of this
Superscalar Processor

Instruction Fetch & Decode Unit
(Multiple Instructions per Cycle)

Reservation Stn.

Reservation Stn.

Reservation Stn.

Reservation Stn.

ALU 1

ALU 2

FP Unit

Branch Unit

Load/Store Unit

Register File

Commit Unit
(Multiple Instructions per Cycle)

Multiple Buses
Third Step: Superscalar Execution

IF | ID | EX1 | EX2 | EX3 | WB
---|----|-----|-----|-----|-----
1: | IF | ID  | EX1 | EX2 | EX3 | WB
2: | IF | ID  | EX1 | EX2 | EX3 | MEM
3: | IF | ID  | EX1 | MEM | WB  |
4: | IF | ID  | EX1 | EX2 | EX3 | EX4 | EX5 | WB
5: | IF | ID  | EX1 | MEM |     |
6: | IF | ID  | EX1 | EX2 | EX3 |     | WB  |
7: | IF | ID  | EX1 |     | MEM |
8: | IF | ID  | EX1 | EX2 | EX3 | EX4 | EX5 | EX6 | WB

Instructions
Several Steps in Exploiting ILP

- Standard
- Pipelining
- Dynamic Scheduling
- Superscalar
References on ILP

- AQA 5th ed., Appendix C
- CAR, Chapter 4—Introduction
Register Renaming

(How Do I Get Rid of WAR and WAW?!...)
Register Renaming

- Importance of removing WAR and WAW dependences with “close-to-ideal” instruction windows (2K entries) and maximum issue rate (64 per cycle)

Source: AQA, © Morgan Kaufman 1996
# A Little History of (Modern) Renaming

## First: IBM 360/91 (1967, FP partial renaming)

<table>
<thead>
<tr>
<th>Year</th>
<th>Processor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1967</td>
<td>IBM 360/91</td>
<td>FP partial renaming</td>
</tr>
</tbody>
</table>

**Source:** Sima, © IEEE 2000

---

**Note:**
- PPC designates PowerPC.
- The Nx586 has scalar issue for CISC instructions but a 3-way superscalar core for converted RISC instructions.
- The issue rate of the Power2 and P2SC is 6 along the sequential path while only 4 immediately after a branch.

---

---

---
Main Dimensions in Renaming Policies

1. Scope of register renaming
   - Simple: only some classes of registers are renamed (e.g., integer or FP only)

2. Layout of the renamed registers
   - Where are they?

3. Method of register mapping
   - Allocation, tracking, and deallocation

4. Rename rate
   - How many instructions can be renamed at once?
Where Are the Rename Registers?

Four possibilities:

1. Merged rename and architectural RF
2. Split rename and architectural RFs
3. Renamed values in the reorder buffer
4. Renamed values in the reservation stations (a.k.a. shelving buffers)
Four Possible Locations for Rename Registers

- Merged architectural and rename register file
  - Method of operand fetching
  - Method of updating the program status
  - Power1 (1990)
  - Power2 (1993)
  - ES/9000 (1992)
  - Nx586 (1994)
  - PMI (Sparc64, 1995)
  - R10000 (1996)
  - R12000 (1999)
  - Alpha 21264 (1998)

- Rename register file
  - PowerPC 603 (1993)
  - PowerPC 604 (1995)
  - PowerPC 620 (1996)
  - Power3 (1998)
  - PA 8000 (1996)
  - PA 8200 (1997)
  - PA 8500 (1999)

- Architectural register file
  - ROB
  - Am290000 superscalar (1995)
  - K6 (1997)
  - Pentium Pro (1995)
  - Pentium II (1997)
  - Pentium III (1999)

- shelving buffers
  - Architectural register file

Source: Sima, © IEEE 2000
Dynamically Scheduled Processor

Instruction Fetch & Decode Unit

Reservation Stn.

ALU

Reservation Stn.

FP Unit

Reservation Stn.

Branch Unit

Reservation Stn.

Load/Store Unit

Commit Unit

Register File

Architectural Registers

Rename Registers
Typical ROB

<table>
<thead>
<tr>
<th>PC</th>
<th>Tag</th>
<th>Register</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x1000 0004</td>
<td>—</td>
<td>$f3</td>
<td>0x627f ba5a</td>
</tr>
<tr>
<td>1</td>
<td>0x1000 0008</td>
<td>ALU1</td>
<td>0xa87f b351</td>
<td>???</td>
</tr>
<tr>
<td>1</td>
<td>0x1000 000c</td>
<td>MUL2</td>
<td>$f5</td>
<td>???</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- from F&D Unit
- from EUs
- to MEM and RF

© Ienne 2003-12
Possible States of each Register in a Merged File

- Initialized (remaining registers)
- Architectural register is reclaimed
- Instruction is completed
- Instruction is canceled
- Entry is allocated to an issued instruction
- RB, not valid
- RB, valid

Source: Sima, © IEEE 2000
State Transitions in a Merged File

- Initialisation:
  - First N registers are “AR”, others “Available”

1. **Available** → **Renamed Invalid**
   - Instruction enters the Reservation Stations and/or the ROB: register allocated for the result (i.e., register uninitialised)

2. **Renamed Invalid** → **Renamed Valid**
   - Instruction completes (i.e., register initialised)

3. **Renamed Valid** → **Architectural Register**
   - Instruction commits (i.e., register “exists”)

4. **Architectural Register** → **Available**
   - Another instruction commits to the same AR (i.e., register is dead)

5. **Renamed Invalid** and **Renamed Valid** → **Available**
   - Squashing
Tracking the Mapping: Where is Physically an Architectural Register?

Mapping in a Mapping Table

Mapping in the Rename Buffer

Source: Sima, © IEEE 2000
Reminder: Reservation Stations

- A reservation station checks that the operands are available (RAW) and that the Execution Unit is free (Control), then starts execution.

![Diagram showing reservation stations and operands]

### ALU1:
- 1
- Operation: `addd`
- Tags: `-`
- Results: `MUL3`, `0xa87f b351`, `??`

### ALU2:
- 1
- Operation: `subd`
- Tags: `ALU1`
- Results: `??`, `0xffff fee1`

### ALU3:
- 0

From F&D Unit:
- Operation `Op`
- Tags `Tag1` and `Tag2`
- Arguments `Arg1` and `Arg2`
Remark the complexity of the Mapping Table:

- 4-issue processor (→ 4x above scheme in parallel)
- 16 parallel accesses: 16 read ports and 4 write ports!
State Transitions Replaced by Copying in Stand-alone RRF

- **Initialisation:**
  - All Rename Registers are “Available”

1. **Available \(\rightarrow\) Renamed Invalid**
   - Instruction enters the Reservation Stations and/or the ROB: register allocated for the result (i.e., register uninitialised)

2. **Renamed Invalid \(\rightarrow\) Renamed Valid**
   - Instruction completes (i.e., register initialised)

3. **Renamed Valid \(\rightarrow\) Available**
   - Instruction commits (i.e., register “exists”)
   - Value is copied in the Architectural RF

4. **Renamed Invalid** and **Renamed Valid \(\rightarrow\) Available**
   - Squashing (no copy to the Architectural RF)
State of the Rename Registers in the Commit Unit (ROB)

<table>
<thead>
<tr>
<th>PC</th>
<th>Tag</th>
<th>Register</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0x0000</td>
<td>0x627f ba5a</td>
</tr>
<tr>
<td>1</td>
<td>ALU1</td>
<td>$f3</td>
<td>0x1000 0004</td>
<td>0x627f ba5a</td>
</tr>
<tr>
<td>1</td>
<td>MUL2</td>
<td>$f5</td>
<td>0x1000 000c</td>
<td>0x627f ba5a</td>
</tr>
</tbody>
</table>

Available

Renamed Valid

Renamed Invalid

from F&D Unit

from EUs

head to MEM and RF
MIPS R10000:
32 AR, 64 PhR, Merged Register File

Mapping Table:

Free Register Table:
Up to 32 empty PhR

Status Table:
Invalid PhR
MIPS R10000: Information Flow

1. **Available → Renamed Invalid**
   - Read new PhR from top of Free Register Table
   - Create new mapping \( \text{LogDest} \rightarrow \text{Dest} \) in the Mapping Table
   - Set corresponding Busy-Bit (=invalid) in the Status Table

2. **Renamed Invalid → Renamed Valid**
   - Write PhR \( \text{Dest} \) indicated in the I-Queue
   - Reset corresponding Busy-Bit (=valid) in the Status Table
   - Mark as Done in the corresponding entry in the ROB

3. **Renamed Valid → Architectural Register**
   - Implicit (removal of historical mapping \( \text{LogDest} \rightarrow \text{Dest} \))

4. **Architectural Register → Available**
   - Free PhR indicated by \( \text{OldDest} \) in the entry removed from the ROB

5. **Renamed Invalid and Renamed Valid → Available**
   - Restore mapping from all squashed ROB entries (from tail to head) as \( \text{LogDest} \rightarrow \text{Dest} \)
   - Reset corresponding Busy-Bit (=valid) in the Status Table
How Many Rename Registers?

- In-Flight instructions:
  \[ N_{in-flight} = N_{RS} + N_{EU} + N_{LD} + N_{ST} \]

- Rename Registers:
  \[ N_{rename} \leq N_{RS} + N_{EU} + N_{LD} \]

- ROB size:
  \[ N_{ROB} \leq N_{in-flight} \]

Note: if strictly < then structural stalls can occur
## Number of Rename Registers

<table>
<thead>
<tr>
<th>Processor type (year of volume shipment)</th>
<th>Type of rename buffer</th>
<th>No. of rename buffers</th>
<th>Issue rate</th>
<th>Width of dispatch window (wdw)</th>
<th>Total no. of rename buffers (nr)</th>
<th>Reorder width (nROB)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RISC processors</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 603 (1993) Ren. reg. file</td>
<td>FX N/A FP 4</td>
<td>3</td>
<td>3</td>
<td>N/A</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>PowerPC 620 (1996) Ren. reg. file</td>
<td>FX 8 FP 8</td>
<td>4</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>R10000 (1996) Merged</td>
<td>FX 32 FP 32</td>
<td>4</td>
<td>48</td>
<td>64</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Alpha 21264 (1998) Merged</td>
<td>FX 48 FP 41</td>
<td>4</td>
<td>36</td>
<td>89</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>PA 8000 (1986) Ren. reg. file</td>
<td>FX 56 FP 56</td>
<td>4</td>
<td>56</td>
<td>112</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td><strong>x86 (CISC processors)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium Pro (1995) In the ROB</td>
<td>FX 40 FP 3</td>
<td>3(^2)</td>
<td>20(^1)</td>
<td>40</td>
<td>40(^1)</td>
<td>40(^1)</td>
</tr>
<tr>
<td>Pentium II (1997) In the ROB</td>
<td>FX 40 FP 3</td>
<td>3(^2)</td>
<td>20(^1)</td>
<td>40</td>
<td>40(^1)</td>
<td>40(^1)</td>
</tr>
<tr>
<td>K5 (1995) In the ROB</td>
<td>FX 16 FP 4(^1)</td>
<td>4(^2)</td>
<td>11(^1) (?)</td>
<td>16</td>
<td>16(^1)</td>
<td>16(^1)</td>
</tr>
<tr>
<td>K6 (1996) In the ROB</td>
<td>FX 74 FP 7(^2)</td>
<td>7(^2)</td>
<td>24(^1)</td>
<td>24</td>
<td>24(^1)</td>
<td>24(^1)</td>
</tr>
<tr>
<td>M3 (2000 expected) Merged</td>
<td>FX 32 N/A 3(^2)</td>
<td>3(^2)</td>
<td>56(^1)</td>
<td>N/A</td>
<td>32(^2)</td>
<td></td>
</tr>
</tbody>
</table>

1. RISC operations
2. x86 instructions (on average, produce 1.3 to 1.9 RISC operations\(^2\))

\(?\) Questionable data
N/A Not available

---

Source: Sima, © IEEE 2000
Actual Choices in Commercial Implementations

Basic alternatives of register remaining

Merged architectural and rename register file
Separate rename register files
Renaming within the ROB
Renaming within the shelving buffers

Basic alternatives

Using a mapping table
Mapping within the RBs
Using a mapping table
Mapping within the RBs
Using a mapping table
Mapping within the RBs
Using a mapping table
Mapping within the RBs

Implementation schemes

Issue-bound operand fetching
Dispatch-bound operand fetching
Issue-bound operand fetching
Dispatch-bound operand fetching
Issue-bound operand fetching
Dispatch-bound operand fetching
Issue-bound operand fetching
Dispatch-bound operand fetching

Proposals

Keller (1996)\(^6\)
Smith-Pleszkun (1987)\(^42\)
Johnson (1987)\(^43\)
Sohi, Vajapeyam (1987)\(^44\)

Processors

Power1 (1990)
ES9000 (1992)
Power 2 (1993)
P2SC (1996)
Nx586 (1994)
R10000 (1996)
R12000 (1999)
M3 (2000)
PMI (1995) (Sparc64)

PowerPC 603 (1993)
PowerPC 604 (1995)
PowerPC 620 (1996)
PA 8000 (1996)
PA 8200 (1997)
Power3 (1998)
PA 8500 (1999)

PentiumPro (1995)
Pentium II (1997)
Pentium III (1999)
AMD29000 (1995)*
K6 (1995)
Lighting (1991)
K6* (1997)

*The shelving buffers are also implemented in the ROB. The resulting unit is occasionally called the DRIS.
### Current High-End Processors

No renaming **only in UltraSparc:** Use of register windows for integers makes it too difficult to implement renaming (and → no OOO either!)

**Nor in Itanium, of course...**

---

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel 1-core Xeon</th>
<th>AMD 1-core Opteron 854</th>
<th>Intel 2-core Xeon X5270</th>
<th>AMD 2-core Opteron 8224SE</th>
<th>Intel 4-core Xeon X7350</th>
<th>AMD 4-core Opteron 8360SE</th>
<th>Intel 6-core Xeon X7460</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bit-width</strong></td>
<td>32/64-bit</td>
<td>32/64-bit</td>
<td>32/64-bit</td>
<td>32/64-bit</td>
<td>32/64-bit</td>
<td>32/64-bit</td>
<td>32/64-bit</td>
</tr>
<tr>
<td><strong>Cores/chip x</strong></td>
<td>1 x 2</td>
<td>1 x 1</td>
<td>2 x 1</td>
<td>2 x 1</td>
<td>4 x 1</td>
<td>4 x 1</td>
<td>6 x 1</td>
</tr>
<tr>
<td><strong>Clock Rate</strong></td>
<td>3.80GHz</td>
<td>2.80GHz</td>
<td>3.50GHz</td>
<td>3.20GHz</td>
<td>2.93GHz</td>
<td>2.50GHz</td>
<td>2.67GHz</td>
</tr>
<tr>
<td><strong>I/D or Unified</strong></td>
<td>12K/16K - 2M - N/A</td>
<td>64K/64K - 3K - 2M - 6M - N/A</td>
<td>2 x 64K/64K - 6M - N/A</td>
<td>4 x 32K/32K - 2M - 4M - N/A</td>
<td>4 x 64K/64K - 4M - N/A</td>
<td>6 x 32K/32K - 3M - 16M</td>
<td></td>
</tr>
<tr>
<td><strong>Execution Rate/Core</strong></td>
<td>3 Instructions</td>
<td>3 Instructions</td>
<td>1 Complex + 3 Simple</td>
<td>3 Instructions</td>
<td>1 Complex + 3 Simple</td>
<td>3 Instructions</td>
<td>1 Complex + 3 Simple</td>
</tr>
<tr>
<td><strong>Pipeline Stages</strong></td>
<td>31</td>
<td>12 int/17 fp</td>
<td>90</td>
<td>72</td>
<td>90</td>
<td>72</td>
<td>90</td>
</tr>
<tr>
<td><strong>Out of Order</strong></td>
<td>12/17</td>
<td>12/17</td>
<td>12/17</td>
<td>12/17</td>
<td>12/17</td>
<td>12/17</td>
<td>12/17</td>
</tr>
<tr>
<td><strong>Memory Bus</strong></td>
<td>800MHz</td>
<td>6.4GB/s</td>
<td>1.33GHz</td>
<td>10.6GB/s</td>
<td>10.6GHz</td>
<td>10.6GB/s</td>
<td>10.6GHz</td>
</tr>
<tr>
<td><strong>IC Process</strong></td>
<td>90nm</td>
<td>90nm</td>
<td>90nm</td>
<td>90nm</td>
<td>65nm</td>
<td>65nm</td>
<td>45nm</td>
</tr>
<tr>
<td><strong>Transistor</strong></td>
<td>160M</td>
<td>120M</td>
<td>410M</td>
<td>233M</td>
<td>463M</td>
<td>1900M</td>
<td></td>
</tr>
<tr>
<td><strong>List Price (Intro)</strong></td>
<td>$900</td>
<td>$1,151</td>
<td>$1,172</td>
<td>$2,146</td>
<td>$3,301</td>
<td>$3,149</td>
<td>$3,729</td>
</tr>
<tr>
<td><strong>Power (Max)</strong></td>
<td>110W</td>
<td>93W</td>
<td>80W</td>
<td>120W</td>
<td>130W</td>
<td>105W</td>
<td>130W</td>
</tr>
<tr>
<td><strong>Availability</strong></td>
<td>3Q05</td>
<td>3Q05</td>
<td>3Q05</td>
<td>3Q07</td>
<td>3Q07</td>
<td>2Q08</td>
<td>4Q08</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>1-2 Chips</td>
<td>2-4 Chips</td>
<td>1-2 Chips</td>
<td>1-4 Chips</td>
<td>1-4 Chips</td>
<td>2-4 Chips</td>
<td>1-4 Chips</td>
</tr>
</tbody>
</table>

**SPECint/2006 [Cores]:**


**SPECint/2006_rate [Cores]:**


**Code Name:**

- Itanium: Sun
- K8: AMD
- Netburst: Intel
- Athlon: AMD
- Wolfson: AMD
- Santa Rosa: AMD
- Tigerton: AMD
- Barcelona: AMD
- Dunnington: AMD

---

All SPEC scores are lower. *Score measured at 4.20GHz (net 5.0GHz).*
References on Register Renaming

- AQA 5th ed., Appendix C and Chapter 3
- PA, Sections 6.3, 6.4, and 6.5
- CAR, Chapter 5—Introduction
Prediction and Speculation

(Don’t Know It? Don’t Wait but Guess...)
Prediction & Speculation: The Idea

- Some operation takes awfully long?
- The processor needs the result to proceed?
  - To fetch the next instruction, one needs to know which one must be fetched
  - To perform a computation, one needs the operands

**Don’t wait!!!**

1. Make a guess (⇒ Predict) and
2. Proceed tentatively (⇒ Speculate)
General Problems

1. How do I make a good guess?
   - Remember some history…

2. What do I do if the guess was wrong?
   - Undo speculatively-executed instructions (“squash”)
   - May cost nothing—e.g.,
     - Squash some results
   - May cost something—e.g.,
     - Empty pipelines
     - Restore saved state
     - Execute compensation code
Branch Prediction

- The idea:
  Start fetching and decoding before knowing the outcome of a branch
- Rudimentary form of speculation
  - Guess if branch will be taken or not
  - Tentative PC advancements do not affect machine state (no execution) → Backing up is easy…
Branch Prediction

Branch outcome and additional info

Predicted direction (Taken/Not Taken)

Current PC

Branch Predictor Logic

Predicted target address
Branch Target Buffers

One needs to know if a just fetched and yet undecoded instruction is a branch and what is the destination (computed branch, relative address, return, etc.)
More Complex But Cheaper Branch Target Buffers

PC (Branch Address)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Typical Cache/TLB organisations

<table>
<thead>
<tr>
<th>Tag (31..8)</th>
<th>Target Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1234 56</td>
<td>0xa123 fee4</td>
</tr>
<tr>
<td>0x1235 ef</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag (31..8)</th>
<th>Target Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5678 23</td>
<td>0x7834 3847</td>
</tr>
<tr>
<td>0x1235 78</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Which Strategy to Predict?

- **Static** predictions: ignore history
  1. Never-taken or always-taken
  2. Always-taken-backward (e.g., loops)
  3. Compiler-specified, etc.
  - Still a form of **dynamic** control speculation, because the squashing process is done in hardware

- **Dynamic** prediction: learn from history
  - Record how often a branch was taken in the past
Which Strategy to Predict Dynamically?

1. Same outcome as last time
   - Keep **one bit of history** per recently visited branches
     - Needs an associative memory \( \rightarrow \) expensive
   - Keep **one bit of history** per hashed address
     - Needs only a RAM \( \rightarrow \) inexpensive
     - Different branches alias \( \rightarrow \) mistakes, but we are only guessing, anyway…

2. Same outcome as last few times (hysteresis)
   - Keep a **two-bit saturating history counter** per hashed address; use sign as a predictor
     - Tuned to **for** loops: one misprediction is normal (last iteration) and should not modify the successive prediction (first iteration of a new execution)
## Branch History Table

<table>
<thead>
<tr>
<th>PC (Branch Address)</th>
<th>One-bit Prediction</th>
<th>Two-bit Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 8 7 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0000:</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0000 0001:</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>0000 0010:</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>0000 0011:</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>1111 1110:</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1111 1111:</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>

- **One-bit Prediction**: 0 for not taken, 1 for taken.
- **Two-bit Prediction**: 01 for not taken, 10 for taken, 11 for taken, 00 for not taken.
Two-bit Prediction Scheme

Slightly modified saturating two-bit counter
Two mispredictions → Strong reversal
(e.g., UltraSPARC-I)
Prediction Accuracy

Mispredictions

Source: AQA, © Morgan Kaufman 1996
More Complex Strategies...

- Important novelty in speculative techniques: One does not care to be always right but just to be **right most of the time**!
- Cheap but clever techniques can be used…

### Pattern History Table (PHT):

- **n-bit standard predictors**

### (m,n) Branch Predictor Buffer

- Exploit correlation:
- A global m-bit predictor uses the outcome of the last two branches to select one among four different predictors
Return Address Stack

- Special elementary case of branch prediction:
  - Small stack (e.g., 8-16 values)
  - Each call (CALL, JAL, etc.) pushes a value
  - Each return (RET, JP $ra, etc.) pops a predicted return address

- Functionally identical to the “real” stack but avoids any SP manipulation, memory accesses, argument bypassing, etc.
Dynamic Control Speculation

- The next step after Dynamic Branch Prediction is **Control Speculation**
  - Limited scope if one limits to fetch and decode speculatively
  - More aggressively, one could execute too...

- If instructions are issued and execute before the branch target is known, one needs to avoid changing the state of the processor until the correctness of the prediction is assessed or to restore the state preceding the branch.
Remember Reordering Problem?

- **Fundamental observation**: a processor can do whatever it wants provided that it gives the appearance of sequential execution (i.e., the architectural machine state is updated in program order)

- Idea was to commit in order, so that one could pretend something was not executed in case of an exception in previous instructions

  ➔ **Prediction and Speculation?**!

  **Yes!** Prediction: “no exceptions will occur”
Double Use of the Reorder Buffer

- OOO execution was hidden from the user by preventing instruction to commit until in-order execution would have taken place.
  - If there is an exception, one squashes all uncommitted instructions in the reorder buffer which followed the instruction which raised the exception.

- One can easily extend this functionality to squash all uncommitted instructions following a mispredicted branch!
Commit Unit Now Includes Also Branches

- Something more in the Commit Unit to check branch outcomes

<table>
<thead>
<tr>
<th></th>
<th>PC</th>
<th>Tag</th>
<th>Register</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0x1000 0004</td>
<td></td>
<td></td>
<td>0x627f ba5a</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x1000 0008</td>
<td>BR1</td>
<td>0x2012 1111</td>
<td>???</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x2012 1111</td>
<td>MUL2</td>
<td></td>
<td>???</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Misprediction Has High Cost $\Rightarrow$ Lots of Efforts in Improving Accuracy

- Pipelines become more and more deep (e.g., up to 22-24 cycles in Pentium 4)
- Issue width grows (typically 3-8)
- Large number of in-flight instructions (up to 100-200!)
- Many predicted branches in-flight at once
- Probability of executing speculatively something useful reduces quickly

$$p_{\text{tot}} = \prod_{i}^{all \ pred} p_i$$
### Current High-End Processors

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel 1-core</th>
<th>AMD 1-core</th>
<th>Intel 2-core</th>
<th>AMD 2-core</th>
<th>Intel 4-core</th>
<th>AMD 4-core</th>
<th>Intel 6-core</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xeon</td>
<td>Opteron 854</td>
<td>Xeon X5270 ¹</td>
<td>Opteron 8224SE</td>
<td>Xeon X7350 ²</td>
<td>Opteron 8360SE ³</td>
<td>Xeon X7460 ⁴</td>
</tr>
<tr>
<td><strong>Bit-width</strong></td>
<td>32/64-bit</td>
<td>32/64-bit</td>
<td>32/64-bit</td>
<td>32/64-bit</td>
<td>32/64-bit</td>
<td>32/64-bit</td>
<td>32/64-bit</td>
</tr>
<tr>
<td><strong>Cores/chip x</strong></td>
<td>1 x 2</td>
<td>1 x 1</td>
<td>2 x 1</td>
<td>2 x 1</td>
<td>4 x 1</td>
<td>4 x 1</td>
<td>6 x 1</td>
</tr>
<tr>
<td><strong>Threads/core</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Clock Rate</strong></td>
<td>3.80GHz</td>
<td>2.80GHz</td>
<td>3.50GHz</td>
<td>3.20GHz</td>
<td>2.93GHz</td>
<td>2.50GHz</td>
<td>2.67GHz</td>
</tr>
<tr>
<td><strong>Cache: L1-L2-L3 -</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12K/16K/2K</td>
<td>64K/64K/64K</td>
<td>2 x 32K/32K</td>
<td>2 x 64K/64K</td>
<td>4 x 32K/32K</td>
<td>4 x 512K/64K</td>
<td>6 x 32K/32K</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Execution Rate/Core</strong></td>
<td>3 Instructions</td>
<td>3 Instructions</td>
<td>1 Complex + 3 Simple</td>
<td>3 Instructions</td>
<td>1 Complex + 3 Simple</td>
<td>3 Instructions</td>
<td>1 Complex + 3 Simple</td>
</tr>
<tr>
<td><strong>Pipe Stages</strong></td>
<td>31</td>
<td>12 int / 17 fp</td>
<td>31</td>
<td>12 int / 17 fp</td>
<td>31</td>
<td>12 int / 17 fp</td>
<td>31</td>
</tr>
<tr>
<td><strong>Out of Order</strong></td>
<td>126</td>
<td>72</td>
<td>30</td>
<td>72</td>
<td>96</td>
<td>72</td>
<td>96</td>
</tr>
<tr>
<td><strong>Memory Bus</strong></td>
<td>800MHz</td>
<td>6.4GB/s</td>
<td>1333MHz</td>
<td>10.6GB/s</td>
<td>1066MHz</td>
<td>10.6GB/s</td>
<td>1066MHz</td>
</tr>
<tr>
<td><strong>Package</strong></td>
<td>LGA-775</td>
<td>uPGA 940</td>
<td>LGA-771</td>
<td>LGA-1207</td>
<td>LGA-771</td>
<td>LGA-1207</td>
<td>LGA-771</td>
</tr>
<tr>
<td><strong>I/O Process</strong></td>
<td>90nm 7M</td>
<td>90nm 9M</td>
<td>45nm</td>
<td>90nm 9M</td>
<td>65nm 8M</td>
<td>65nm 11M</td>
<td>45nm</td>
</tr>
<tr>
<td><strong>Die Size</strong></td>
<td>100nm²</td>
<td>107mm²</td>
<td>106mm²</td>
<td>137mm²</td>
<td>227mm²</td>
<td>134mm²</td>
<td>50mm²</td>
</tr>
<tr>
<td><strong>Transistor</strong></td>
<td>160M</td>
<td>120M</td>
<td>410M</td>
<td>233M</td>
<td>463M</td>
<td>1900M</td>
<td></td>
</tr>
<tr>
<td><strong>List Price (Intro)</strong></td>
<td>$903</td>
<td>$1,164</td>
<td>$1,772</td>
<td>$2,180</td>
<td>$2,301</td>
<td>$2,149</td>
<td>$2,729</td>
</tr>
<tr>
<td><strong>Power (Max)</strong></td>
<td>110W</td>
<td>93W</td>
<td>80W</td>
<td>120W</td>
<td>130W</td>
<td>105W</td>
<td>130W</td>
</tr>
<tr>
<td><strong>Availability</strong></td>
<td>3Q05</td>
<td>3Q05</td>
<td>3Q08</td>
<td>3Q07</td>
<td>3Q07</td>
<td>2Q08</td>
<td>4Q08</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>1–2 Chips</td>
<td>2–4 Chips</td>
<td>1–2 Chips</td>
<td>1–4 Chips</td>
<td>1–4 Chips</td>
<td>2–4 Chips</td>
<td>1–4 Chips</td>
</tr>
<tr>
<td><strong>SPECint/fp2006 [Cores]</strong></td>
<td>11.4/11.7 (2)</td>
<td>11.2/12.1 (2)</td>
<td>26.5/25.5 (4)</td>
<td>14.1/14.2 (8)</td>
<td>21.7/18.9 (16)</td>
<td>14.4/18.5 (8)</td>
<td>22.0/22.3 (24)</td>
</tr>
<tr>
<td><strong>SPECfp2006_rate [Cores]</strong></td>
<td>20.9/18.8 (2)</td>
<td>41.4/45.6 (4)</td>
<td>84.9/57.7 (4)</td>
<td>105/96.7 (8)</td>
<td>184/108 (16)</td>
<td>170/156 (16)</td>
<td>274/142 (24)</td>
</tr>
<tr>
<td><strong>Code Name</strong></td>
<td>K8/2000</td>
<td>Athlon 64</td>
<td>Wolfson</td>
<td>Santa Rosa</td>
<td>Tigrerton</td>
<td>Barcelona</td>
<td>Dunnington</td>
</tr>
<tr>
<td><strong>Microarchitecture</strong></td>
<td>Netburst</td>
<td>K8</td>
<td>Core2</td>
<td>Core</td>
<td>K10</td>
<td>Core</td>
<td></td>
</tr>
</tbody>
</table>

### Performance Characteristics

- **Processor**
  - Intel Core 2: 2.90GHz
  - Intel Core i7: 3.33GHz
- **Cores/chip x Threads/core**
  - 2 x 2
- **Clock Rate**
  - 1.60GHz
- **Cache: L1-L2-L3 - I/O or Unified**
  - 2 x 16K/256K - 1M/256K - 12/Mon
- **Execution Rate/Core**
  - 6.8 Issue
- **Pipe Stages**
  - 8
- **Out of Order**
  - None
- **Memory Bus**
  - 850MHz
- **Package**
  - uPGA-775
- **I/O Process**
  - 900nm 7M
- **Die Size**
  - 100mm²
- **Transistor**
  - 160M
- **List Price (Intro)**
  - $903
- **Power (Max)**
  - 110W
- **Availability**
  - 3Q05
- **Scalability**
  - 1–2 Chips

*All SPEC scores are base.*
Speculation Is Not Necessarily a Run-Time Concept

- **Dynamic**: in hardware, no interaction whatsoever from the compiler
  - Binary code is unmodified
- **Static**: in software, planned beforehand by the compiler
  - Binary code is written in such a way as to do speculation (with or without some hardware support in the ISA)
Static Control Speculation Example

- We need to compute:
  \[ \text{if (A==0) A=B; else A=A+4;} \]

- In assembly:
  
  \[
  \begin{align*}
  &\text{LW } R1, 0(R3) \quad ; \text{load A} \\
  &\text{BNEZ } R1, L1 \quad ; \text{test A, possibly skip then} \\
  &\text{LW } R1, 0(R2) \quad ; \text{‘then’ clause: load B} \\
  &J \quad L2 \quad ; \text{skip else} \\
  &L1: \text{ADD } R1, R1, 4 \quad ; \text{‘else’ clause: compute A+4} \\
  &L2: \text{SW } 0(R3), R1 \quad ; \text{store new A}
  \end{align*}
  \]

- If we know that the ‘then’ clause is almost always executed, can we optimise this code?
Static Control Speculation Example

- We could speculatively start earlier to load B into another register and, if needed, squash the value with the right one.

- In assembly:
  
  ```assembly
  LW R1, 0(R3) ; load A
  LW R14, 0(R2) ; speculative load B
  BEQZ R1, L3 ; test A, possibly skip else
  ADD R14, R1, 4 ; ‘else’ clause: compute A+4
  L3: SW 0(R3), R14 ; store new A
  ```

- Advantages: now we load B while the test is performed (→ in parallel)

- Any problem? As usual: exceptions…
Exceptions and Speculation

Some ways to handle exceptions in speculative execution:

- **Static renaming**: Hardware and operating systems cooperatively ignore exceptions.
- **Poison bits**: Mark results as speculative and delay exception at first use.
- **Speculative instructions**: Mark instruction as speculative and do not commit the result until speculation is solved.
Static Renaming and Hardware-Software Cooperation

- Back to our example:

```
LW R1, 0(R3) ; load A
LW R14, 0(R2) ; speculative load B
BEQZ R1, L3 ; test A, possibly skip else
ADD R14, R1, 4 ; ‘else’ clause: compute A+4
L3: SW 0(R3), R14 ; store new A
```

- OS “helps” with two policies:
  - Nonterminating exceptions (e.g., Page Fault): resume independently from speculativeness $\rightarrow$ performance penalty, but execution ok
  - Terminating exceptions (e.g., Divide by Zero): ignore and return an undefined value $\rightarrow$ if it was speculated, it will be unused

- Problem: nonspeculative terminating exceptions?
Static Renaming and Poison Bits

- Special marker for speculative instructions:
  
  \[
  \begin{align*}
  \text{LW} & \quad R1, 0(R3) \quad ; \text{load } A \\
  \text{LW}^* & \quad R14, 0(R2) \quad ; \text{speculative load } B \\
  \text{BEQZ} & \quad R1, L3 \quad ; \text{test } A, \text{ possibly skip else} \\
  \text{ADD} & \quad R14, R1, 4 \quad ; \text{‘else’ clause: compute } A+4 \\
  \text{L3: SW} & \quad 0(R3), R14 \quad ; \text{store new } A; \text{ report exceptions}
  \end{align*}
  \]

- The processor knows the load is speculative and turns on R14’s Poison Bit if it raises a terminating exception, and suppresses the exception.

- The add, if executed, resets the R14’s Poison Bit.

- When R14 is used, a deferred terminal exception is raised if its Poison Bit is set.
Example:
Speculative Loads in Itanium (I)

- **Goal**: move loads as early as possible, even **speculatively** before preceding branches (i.e., without being sure they are really needed)

```plaintext
<some code>
(p1) br.cond somewhere
// ------ barrier
ld r1 = [r2]
<some code using r1>
```

Exceptions?

```plaintext
ld r1 = [r2]  // load could be speculated
// if old value r1 not needed
// <- neither here nor in "somewhere"

<some code>
(p1) br.cond somewhere
// ------ barrier
<some code using r1>  // but...
```
Example:
Speculative Loads in Itanium (II)

- Speculative loads and deferred exceptions to explicit compiler-generated fix-up code

```plaintext
ld.s r1 = [r2] // speculative loads do not raise exceptions but mark the register with the additional NaT bit

<some code>
<some code using r1> // NaT is propagated in further calculations, which also defer exceptions

(pl) br.cond somewhere // ------ barrier

<some more code using r1>
chk.s r1, fix_code_r1 // call exception handler if needed // to fix-up execution
```
Predication (= Guarded) Execution

- A special form of static control speculation?
  “I cannot make a good prediction? I will avoid gambling and will do both”

- A bit more than that: removes control flow change altogether

- Not always a good idea: compiler trade-off
  - (Almost) free if one uses execution units which where not used otherwise (e.g., because of limited ILP)
  - Not free at all in the general case: more than needed is always executed
Weak- and Strong-dependence Models

- Typical model for data dependences is:
  - No dependence (B does not depend on A)
  - Strong dependence (B depends on A)
    - If A and B are executed OOO, one must be always right about the dependence

- Weak-dependence model:
  - A dependence can be temporarily violated (predicted negative and speculated) if means are in place to recover correct execution
    - If A and B are executed OOO, one must be mostly right about the dependence
Dynamic Data Dependence Prediction

Examples:

- **Memory Dependence Prediction**: predicts whether a load is dependent on a pending store (memory aliasing or disambiguation)

- **Alias Prediction**: predicts which pending store contains the right value for a load
Static Data Dependence
Speculation (I)

- Potential RAW dependencies through memory are to be conservatively assumed as real dependencies → Loss of useful reordering possibilities

- **Goal**: move loads as early as possible, even **speculatively** before preceding stores (i.e., without being sure that the value is right)

```c
<some code>
st [r3] = r4
// ------ barrier
ld r1 = [r2]
<some code using r1>
```

```c
ld r1 = [r2]  // load could be speculated...
<some code>
st [r3] = r4  // ...but if r2==r3, r1 is WRONG!
// ------ barrier
<some code using r1>
```
Speculative Loads get executed but mark the destination register as “speculatively” loaded and track subsequent stores for a conflict.

```
ld.a r1 = [r2] // speculative loads are normal
    // but mark always the register
    // with the additional NaT bit
<some code>
<some code using r1> // NaT is propagated in further
    // calculations
st [r3] = r4 // successive stores are checked
    // to see if they rewrite locations
    // which were object of speculative
    // loads
// ------ barrier
<some more code using r1>
chk.a r1, fix_code_r1 // if violated RAW dependence, call
    // special fix-up routine
```

Important advantage because loads (slow operations) can now be started earlier.
Dynamic Data Value Prediction

- **Examples:**
  - **Source Operand Value Prediction:** predict quasi-constant input operands
    - Many constant values during program execution
    - History table recording last value
  - **Value Stride Prediction:** predict constant increments across input operands
    - History table recording stride between last two values
  - **Load Addresses and Load Values**
# Types of Prediction & Speculation

<table>
<thead>
<tr>
<th></th>
<th><strong>Dynamic</strong> (by the hardware)</th>
<th><strong>Static</strong> (by the compiler)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exceptions</strong></td>
<td>OOO execution and reordering</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Imprecise exceptions in DBT (e.g., Transmeta Crusoe)</td>
<td>—</td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td>Branch Prediction (AQA, 4.3)</td>
<td>Trace Scheduling (AQA, 4.4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hyperblocks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Predication (AQA, 4.6)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speculative Loads (e.g., Itanium)</td>
</tr>
<tr>
<td><strong>Data Availability</strong></td>
<td>Virtual memory</td>
<td>—</td>
</tr>
<tr>
<td><strong>Data Dependence</strong></td>
<td>Research?</td>
<td>Advanced Loads (e.g., Itanium)</td>
</tr>
<tr>
<td><strong>Data Value</strong></td>
<td>Research</td>
<td>Dynamic compilers (e.g., DyC, Calpa)</td>
</tr>
</tbody>
</table>

Not all of these are traditionally called “speculation”!
References on Prediction & Speculation

- AQA 5th ed., Chapter 3 and Appendix H
- PA, Sections 4.3 and 5.3
Simultaneous Multithreading

(How do I fill my issue slots?!...)
Sources of Parallelism

- Bit-level
  - Wider processor datapaths (8→16→32→64…)

- Word-level (SIMD)
  - Vector processors
  - Multimedia instruction sets (Intel’s MMX and SSE, Sun’s VIS, etc.)

- Instruction-level
  - Pipelining
  - Superscalar
  - VLIW and EPIC

- Task- and Application-levels...
  - Explicit parallel programming
  - Multiple threads
  - Multiple applications…
Simple Sequential Processor

- **op 1**
- **op 2**
- **op 3**
Pipelined Processor
Superscalar Processor
OOO Superscalar Processor

functional units

| cycles   |   |   |
|-----------|   |   |
| op 1      | op 2 |   |
| op 4      |   | op 5 |
| op 3      |   | op 6 (br) |
| op 11     |   | op 7 |

AdvCompArch — Exploiting ILP Dynamically

© Ienne 2003-12
Speculative Execution

The diagram illustrates the concept of speculative execution in computer architecture. Each row represents a sequence of operations (ops) that are executed in parallel by functional units. The columns represent the progression of cycles, where each cycle advances one step in the execution of the operations.

- **Operations (Ops)**: The operations are labeled as `op 1`, `op 2`, `op 3`, `op 4`, `op 5`, `op 6 (br)`, `op 7 ?`, `op 8 ?`, `op 9 ?`, `op 10 ?`, `op 11`, `op 12`, and `op 13`.
- **Functional Units**: These are represented along the horizontal axis, indicating the available units for execution in each cycle.

The diagram shows how operations are speculated and executed in parallel, with some operations marked as `?` to indicate that their outcomes are pending and may lead to branching or other computational paths.
## Limits of ILP

<table>
<thead>
<tr>
<th>Cycles</th>
<th>op 1</th>
<th>op 2</th>
<th>op 4</th>
<th>op 10?</th>
<th>op 5</th>
<th>op 3</th>
<th>op 6 (br)</th>
<th>op 7?</th>
<th>op 8?</th>
<th>op 9?</th>
<th>op 11</th>
<th>op 12</th>
<th>op 13</th>
</tr>
</thead>
</table>

*Note: The table represents the functional units over cycles with some operations marked with a question mark.*
Sources of Unused Issue Slots

Percent of Total Issue Cycles

Applications

alvinn
doduc
eqntott
espresso
fpppp
hydro2d
li
mdljdp2
mdljsp2
nasa7
ora
su2cor
swm
tomcatv

composite

memory conflict
long fp
short fp
long integer
short integer
load delays
control hazards
branch misprediction
dcache miss
icache miss
dtlb miss
itlb miss
processor busy

Source: Tullsen et al., © IEEE 1995
Horizontal and Vertical Waste

Horizontal Waste: 39%
Vertical Waste: 61%

Source: Tullsen et al., © IEEE 1995

AdvCompArch — Exploiting ILP Dynamically © Ienne 2003-12
Multithreading: The Idea

Rather than enlarging the **depth** of the instruction window (more speculation with **lowering confidence**)!, enlarge its **“width”**.

→ fetch from multiple threads!
Basic Needs of a Multithreaded Processor

- Processor must be aware of several independent states, one per each thread:
  - Program Counter
  - Register File (and Flags)
  - (Memory)

- Either multiple resources in the processor or a fast way to switch across states
Thread Scheduling

When one switches thread?
Which thread will be run next?

- Simple interleaving options:
  - Cycle-by-cycle multithreading
    - Round-robin selection between a set of threads
  - Block multithreading
    - Keep executing a thread until something happens
      - Long latency instruction found
      - Some indication of scheduling difficulties
      - Maximum number of cycles per thread executed
Cycle-by-cycle Interleaving (or Fine-Grain) Multithreading

- functional units
- cycles
- context switches
- op 1, op 2, op 5
- op 1, op 2
- op 5, op 2, op 1
- op 3, op 4, op 6
- op 3, op 7, op 4
- op 3
- op 11, op 10, op 8
- op 11
- op 6, op 9
- op 5, op 10
- op 6, op 9, op 7, op 4
## Block Interleaving (or Coarse-Grain) Multithreading

<table>
<thead>
<tr>
<th>Cycles</th>
<th>functional units</th>
<th>Context switches</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op 1</td>
<td>op 2</td>
<td>op 5</td>
</tr>
<tr>
<td>op 3</td>
<td>op 4</td>
<td>op 6</td>
</tr>
<tr>
<td></td>
<td>op 8</td>
<td></td>
</tr>
<tr>
<td>op 1</td>
<td>op 2</td>
<td></td>
</tr>
<tr>
<td>op 3</td>
<td>op 7</td>
<td>op 4</td>
</tr>
<tr>
<td></td>
<td>op 8</td>
<td></td>
</tr>
<tr>
<td>op 5</td>
<td>op 2</td>
<td>op 1</td>
</tr>
<tr>
<td>op 6</td>
<td>op 9</td>
<td>op 7</td>
</tr>
<tr>
<td></td>
<td>op 7</td>
<td>op 3</td>
</tr>
</tbody>
</table>

© Ienne 2003-12
Fundamental Requirement

- Key issue in general-purpose processors which has prevented for many years multithreaded techniques to become commercially relevant.

It is not acceptable that single-thread performance goes significantly down or at all.
Problems of Cycle-by-Cycle Multithreading

- Null time to switch context
  - Multiple Register Files

- No need for forwarding paths if threads supported are more than pipeline depth!
  - Simple(r) hardware

- Fills well short vertical waste (other threads hide latencies ~ no. of threads)

- Fills much less well long vertical waste (the thread is rescheduled no matter what)

- Does not reduce significantly horizontal waste (per thread, the instruction window is not much different...)

- Significant deterioration of single thread job
Block Interleaving Techniques

Block Interleaving

Static
- Explicit Switch
- Implicit Switch
  - Switch-on-Load
  - Switch-on-Store
  - Switch-on-Branch

Dynamic
- Explicit Switch
  - Conditional-Switch
- Implicit Switch
  - Switch-on-Miss
  - Switch-on-Use
  - Switch-on-Signal
    (a.k.a. Lazy-Switch-on-Miss)
    (interrupt, trap, …)
Problems of Block Multithreading

- Scheduling of threads not self-evident:
  - What happens if thread #1 executes perfectly well and leaves no gap?
  - Explicit techniques require ISA modifications ➔ Bad...

- More time allowable for context switch
- Fills very well long vertical waste (other threads come in)
- Fills poorly short vertical waste (if not sufficient to switch context)
- Does not reduce almost at all horizontal waste
Simultaneous Multithreading (SMT): The Idea

![Diagram showing Simultaneous Multithreading (SMT) operations and cycles.](image-url)
Several Simple Scheduling Possibilities

- Prioritised scheduling?
  - Thread #0 schedules freely
  - Thread #1 is allowed to use #0 empty slots
  - Thread #2 is allowed to use #0 and #1 empty slots, etc.

- Fair scheduling?
  - All threads compete for resources
  - If several threads want the same resource, round-robin assignment
Superscalar Processor

- Instruction Fetch & Decode Unit
  - (Multiple Instructions per Cycle)

- Multiple Buses

- Reservation Stn.
- ALU 1
- ALU 2
- FP Unit
- Branch Unit
- Load/Store Unit

- Commit Unit
  - (Multiple Instructions per Cycle)

- IQ

- Multiple Buses

- Register File
### Reorder Buffer

<table>
<thead>
<tr>
<th></th>
<th>PC</th>
<th>Tag</th>
<th>Register</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x1000 0004</td>
<td>—</td>
<td>$f3</td>
<td></td>
<td>0x627f ba5a</td>
</tr>
<tr>
<td>1</td>
<td>0x1000 0008</td>
<td>ALU1</td>
<td></td>
<td>0xa87f b351</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x1000 000c</td>
<td>MUL2</td>
<td></td>
<td>$f5</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Register Renaming

(between fetch/decode and commit)

- **head** from EUs to MEM and RF
- **tail** from F&D Unit

© Ienne 2003-12
What Must Be Added to a Superscalar to Achieve SMT?

- Multiple program counters (= threads) and a policy for the instruction fetch unit(s) to decide which thread(s) to fetch
- Multiple or larger register file(s) with at least as many registers as logical registers for all threads
- Multiple instruction retirement (e.g., per thread squashing)
  - No changes needed in the execution path

And also:
- Thread-aware branch predictors (BTBs, etc.)
- Per-thread Return Address Stacks
SMT Processor as a Natural Extension of a Superscalar

Instruction Fetch & Decode Unit (Multiple Instructions per Cycle)

Reservation Stn.

ALU 1

ALU 2

FP Unit

Branch Unit

Load/Store Unit

Commit Unit (Multiple Instructions per Cycle)

Register File(s)

PC

IQ
Reorder Buffer Remembers the Thread of Origin

- Some changes to the reorder buffer in the Commit Unit—e.g.:

```
<table>
<thead>
<tr>
<th>PC</th>
<th>Tag</th>
<th>Register</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000 0004</td>
<td>2</td>
<td>$f3</td>
<td></td>
<td>0x627f ba5a</td>
</tr>
<tr>
<td>0x1000 0008</td>
<td>2 ALU1</td>
<td></td>
<td>0xa87f b351</td>
<td>???</td>
</tr>
<tr>
<td>0x2001 1234</td>
<td>1 MUL3</td>
<td>$f3</td>
<td></td>
<td>???</td>
</tr>
<tr>
<td>0x1000 000c</td>
<td>2 MUL2</td>
<td>$f5</td>
<td></td>
<td>???</td>
</tr>
</tbody>
</table>
```

Architectural Register Identifier:
Reg # + Thread #
Reservation Stations

- Reservation stations do not need to know which thread an instruction belongs to.
- Remember: operand sources are renamed—physical regs, tags, etc.

<table>
<thead>
<tr>
<th>ALU1:</th>
<th>ALU2:</th>
<th>ALU3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><code>addd</code></td>
<td><code>subd</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>ALU1</code></td>
<td></td>
</tr>
<tr>
<td><code>MUL3</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>0xa87f b351</code></td>
<td><code>???</code></td>
<td><code>0xffff fee1</code></td>
</tr>
</tbody>
</table>

From F&D Unit:
- Op
- Tag1
- Tag2
- Arg1
- Arg2

From EUs and RF:
- Thread #1?!
- Thread #2?!
Does It Work?!
Main Results on Implementability
SMT vs. Superscalar

From [Tullsen] un96:

- Instruction scheduling not more complex
- Register File datapaths not more complex (but much larger register file!)
- Instruction Fetch Throughput is attainable even without more fetch bandwidth
- Unmodified cache and branch predictors are appropriate also for SMT
- SMT achieves better results than aggressive superscalar
Where to Fetch?

- **Static** solutions: Round-robin
  - Each cycle 8 instructions from 1 thread
  - Each cycle 4 instructions from 2 threads, 2 from 4,…
  - Each cycle 8 instructions from 2 threads, and forward as many as possible from #1 then when long latency instruction in #1 pick rest from #2

- **Dynamic** solutions: Check execution queues!
  - Favour threads with minimal # of in-flight branches
  - Favour threads with minimal # of outstanding misses
  - **Favour threads with minimal # of in-flight instructions**
  - Favour threads with instructions far from queue head
What to Issue?

- Not **exactly** the same as in superscalars...
  - In superscalar: oldest is the best (least speculation, more dependent ones waiting, etc.)
  - In SMT not so clear: branch-speculation level and optimism (cache-hit speculation) vary across threads

- One can think of many selection strategies:
  - Oldest first
  - Cache-hit speculated last
  - Branch speculated last
  - Branches first...

- Important result: **doesn’t matter too much!**

⇒ Issue Logic (critical in superscalars) can be left alone
Importance of Accurate Branch Prediction in SMT vs. Superscalar

- Reduce the impact of Branch Prediction was one of the qualitative initial motivations

Results from [Tullsen] un96:

- Perfect branch prediction advantage
  - 25% at 1 thread
  - 15% at 4 threads
  - 9% at 8 threads

- Losses due to suppression of speculative execution
  - -7% at 8 threads
  - -38% at 1 thread (speculation was a good idea...)

© Ienne 2003-12
Bottlenecks
Sources of Unused Issue Slots

- Completion queue not very relevant (remember: this is out of the execution path…)
- **Rename register count** important
- Most critical: number of register **writeback ports**

SMT for **utilisation rate (EUs)** not **bandwidth**!
Bottlenecks

- Fetch and memory throughput are still bottlenecks
  - Fetch: branches, etc.
  - Memory not addressed
- Performance vs. # of rename registers (8T) in addition to the architectural ones
  - Infinite: +2%
  - 100: ref.
  - 90: -1%
  - 80: -3%
  - 70: -6%
- Register file access time likely limit to # of threads

Source: Tulsien et al., © IEEE 1996

IPC vs. # threads
200 physical registers
Superscalars are cheap only for relatively small issue bandwidth, then quickly down.

SMT improves significantly the picture already with 2 threads and maximum moves to larger issue bandwidths with more threads.
Introduction of SMT in Commercial Processors

- Compaq Alpha 21464 (EV8)
  - 4T SMT
  - Project killed June 2001

- Intel Pentium IV (Xeon)
  - 2T SMT
  - Availability since 2002
    (already there before, but not enabled)
  - 10-30% gains expected

- SUN Ultra III
  - 2-core CMP, 4T SMT
Intel SMT: Xeon Hyper-Threading Pipeline

Front-end (TC hit)

- IP
- Trace Cache
- Duplicated resources

Front-end (TC miss)

- I-Fetch Queue
- OR
- L2 Access
- ITLB
- L2 Access
- Decoded Queue
- Decode
- Trace Cache
- IP

OOO Execution

- Uop Queue
- Rename
- Register Read
- Execute
- L1 D-Cache
- Register Write
- Retire

Resources:
- Freely shared resources
- Split resources
- Time-shared resources

Source: Marr et al., © Intel 2002

AdvCompArch — Exploiting ILP Dynamically

© Ienne 2003-12
What happens when there is only one thread? What does the OS when there is nothing to do? Ahem…

Four modes: Low-power, ST0, ST1, and MT
Intel SMT: Xeon Hyper-Threading
Goals and Results

- Minimum additional cost: SMT = approx. 5% area
- No impact on single-thread performance
  - Recombine partitioned resources
- Fair behaviour with 2 threads

Source: Marr et al., © Intel 2002
And Now, What’s Next?

- Key ingredients for success so far:
  - Maximise compatibility, no info from programmers beyond straight sequential code and coarse threads
  - Aggressive prediction and speculation of anything predictable
  - Use irregular, fine-grained parallelism (ILP): it is “easier” to extract, can be done at runtime,…

- Problems:
  - Branch prediction accuracy hard to improve
  - Hard to exploit ILP any further within a thread
SMT Research Directions

- Dynamic Multithreading (DMT) [AkkariNov98]
  - Automatic generation of threads from loops (backward branches) and procedure call

- Execution on a “typical” SMT microarchitecture
- Speculative execution of threads with interthread data dependence and value speculation
SMT Research Directions

- What Dynamic Multithreading represents?
  - Introduces, from a single program, not only out-of-order Issue but now also out-of-order Fetch
  - Lowers the dependence on correct Branch Prediction
- Experiments show good advantages even without more fetch bandwidth

Source: Akkary et al., © IEEE 1998
References on Simultaneous Multithreading

- AQA 5th ed., Chapter 3
- PA, Sections 6.3, 6.4, and 6.5
- CAR, Chapter 5—Introduction
- D. M. Tullsen et al., *Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor*, ISCA, 1996
- H. Marr et al., *Hyper-Threaded Technology Architecture and Microarchitecture*, Intel Technology Journal, Q1, 2002