Beyond static pipelines and single threads

- Dynamic scheduling.
- Exception handling.
- Multi-threading and multi-core.
- Cache coherency.
Dynamic Scheduling
Recall: Out of Order Execution

Goal: Issue instructions out of program order

Example:

... so let ADDD go first

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Seconds</th>
<th>Instructions</th>
<th>Cycles</th>
<th>Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td></td>
<td>F2, 34(R2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td></td>
<td>F4, 45(R3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTD</td>
<td></td>
<td>F6, F4, F2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ADDD</td>
<td></td>
<td>F8, F2, F2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

MULTD waiting on F4 to load ...
Dynamic Scheduling: Enables Out-of-Order

Goal: Enable out-of-order by breaking pipeline in two: fetch and execution.

Example: IBM Power 5:

I-fetch and decode: like static pipelines

Today’s focus: execution unit
Recall: WAR and WAW hazards ...

Write After Read (WAR) hazards. Instruction $I_2$ expects to write over a data value after an earlier instruction $I_1$ reads it. But instead, $I_2$ writes too early, and $I_1$ sees the new value.

Write After Write (WAW) hazards. Instruction $I_2$ writes over data an earlier instruction $I_1$ also writes. But instead, $I_1$ writes after $I_2$, and the final data value is incorrect.

Dynamic scheduling eliminates WAR and WAW hazards, making out-of-order execution tractable.
Dynamic Scheduling: A mix of 3 ideas

Imagine: an endless supply of registers ...

Top-down idea: Registers that may be written only once (but may be read many times) eliminate WAW and WAR hazards.

Mid-level idea: An instruction waiting for an operand to execute may trigger on the (single) write to the associated register. (eliminates RAW hazards)

Bottom-up idea: To support “snooping” on register writes, attach all machine elements to a common bus.

Robert Tomasulo, IBM, 1967. FP unit for IBM 360/91
Register Renaming

Imagine: an endless supply of registers??
How???
Consider this simple loop ...

Loop: LD F0,0(R1) ;F0= array element
      ADDD F4,F0,F2 ;add scalar from F2
      SD F4,0(R1) ;store result
      SUBI R1,R1,8 ;decrement pointer 8B (DW)
      BNEZ R1,Loop ;branch R1!=zero
      NOP ;delayed branch slot

Every pass through the loop introduces the potential for WAW and/or WAR hazards for F0, F4, and R1.

(Note: F registers are floating point registers. F0 is not equal to the constant 0, but instead is a normal register just like F1, F2, ...).
Given an endless supply of registers ...

Rename “architected registers” (Ri, Fi) to new “physical registers” (PRi, PFi) on each write.

```
ADDI R1, R0, 64
Loop:
  LD F0, 0(R1)
  ADDD F4, F0, F2
  SD F4, 0(R1)
  SUBI R1, R1, 8
  BNEZ R1, Loop
  NOP

  ADDI PR01, PR00, 64
  LD PF00 0(PR01)
  ADDD PF04, PF00, PF02
  SD PF04, 0(PR01)
  SUBI PR11, PR01, 8
  BEQZ PR11 ENDLOOP
```

**What was gained?**

An instruction may execute once all of its source registers have been written.
Bus-Based CPUs
A common bus == long wires == slow?

Pipelines in theory

Wires are short, so clock periods can be short.

“wiring by abutment”

Long wires are the price we paid to avoid stalls.

Pipelines in practice

Conjecture:
If processor speed is limited by long wires, let's do a design that fully uses the semantics of long wires by using a bus.
A bus-based multi-cycle computer

If we add too many functional units, one bus is too long, too slow. Solutions: more buses, faster electrical signalling

Common Data Bus <data id#, data value>

(1) Only one unit writes at a time (one source).
(2) All units may read the written values (many destinations), if interested in id#.
Data-Driven Execution

(Associative Control)

Caveat: In comparison to static pipelines, there is great diversity in dynamic scheduling implementations. Presentation that follows is a composite, and does not reflect any specific machine.
Recall: IBM Power 5 block diagram ...

Queues between instruction fetch and execution.

MP = “Mapping” from architected registers to physical registers (renaming).

ISS = Instruction Issue
Instructions placed in “Reorder Buffer”

Each line holds physical <src1, src2, dest> registers for an instruction, and controls when it executes.

Reorder Buffer

<table>
<thead>
<tr>
<th>Inst #</th>
<th>src1 #</th>
<th>src1 val</th>
<th>src2 #</th>
<th>src2 val</th>
<th>dest #</th>
<th>dest val</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[...]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution engine works on the physical registers, not the architecture registers.

Common Data Bus: <reg #, reg val>
## Circular Reorder Buffer: A closer look

### Next instr to “commit”, (complete).

### Instruction opcode
- Add next inst, in program order.
- Physical register numbers
- Valid bits for values
- Copies of physical register values

<table>
<thead>
<tr>
<th>Inst #</th>
<th>Op</th>
<th>U</th>
<th>E</th>
<th>#1</th>
<th>#2</th>
<th>#d</th>
<th>P1</th>
<th>P2</th>
<th>Pd</th>
<th>P1 value</th>
<th>P2 value</th>
<th>Pd value</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>ADD</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>OR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>SUB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Execute bit (0 if waiting ...
- Use bit (1 if line is in use)

- Next instr to “commit”, (complete).
- Physical register numbers
- Valid bits for values
- Copies of physical register values

- Instruction opcode
- Execute bit (0 if waiting ...
- Use bit (1 if line is in use)
Example: The life of  ADD R3, R1, R2

Issue: R1 “renamed” to PR21, whose value (13) was set by an earlier instruction. R2 renamed to PR22; it has not been written. R3 renamed to PR23.

<table>
<thead>
<tr>
<th>Inst#</th>
<th>Op</th>
<th>U</th>
<th>E</th>
<th>#1</th>
<th>#2</th>
<th>#d</th>
<th>P1</th>
<th>P2</th>
<th>Pd</th>
<th>P1 value</th>
<th>P2 value</th>
<th>Pd value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Add</td>
<td>1</td>
<td>0</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

A write to PR22 appears on the bus, value 87. Both operands are now known, so 13 and 87 sent to ALU.

<table>
<thead>
<tr>
<th>Inst#</th>
<th>Op</th>
<th>U</th>
<th>E</th>
<th>#1</th>
<th>#2</th>
<th>#d</th>
<th>P1</th>
<th>P2</th>
<th>Pd</th>
<th>P1 value</th>
<th>P2 value</th>
<th>Pd value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Add</td>
<td>1</td>
<td>1</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>13</td>
<td>87</td>
<td>-</td>
</tr>
</tbody>
</table>

ALU does the add, writing < PR23, 100 > onto the bus.

<table>
<thead>
<tr>
<th>Inst#</th>
<th>Op</th>
<th>U</th>
<th>E</th>
<th>#1</th>
<th>#2</th>
<th>#d</th>
<th>P1</th>
<th>P2</th>
<th>Pd</th>
<th>P1 value</th>
<th>P2 value</th>
<th>Pd value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Add</td>
<td>1</td>
<td>1</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td>87</td>
<td>100</td>
</tr>
</tbody>
</table>
More details (many are still overlooked)

Q. Why are we storing each physical register value several times in the reorder buffer? Quick access.
Exceptions and Interrupts

**Exception:** An unusual event happens to an instruction during its execution. **Examples:** divide by zero, undefined opcode.

**Interrupt:** Hardware signal to switch the processor to a new instruction stream. **Example:** a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting).
Challenge: Precise Interrupt / Exception

**Definition:**

*It must appear as if an interrupt is taken between two instructions* (say $I_i$ and $I_{i+1}$)

- the effect of all instructions up to and including $I_i$ is totally complete
- no effect of any instruction after $I_i$ has taken place

The interrupt handler either aborts the program or restarts it at $I_{i+1}$.

**Follows from the “contract” between the architect and the programmer ...**
Precise Exceptions in Static Pipelines

Key observation: architected state only change in memory and register write stages.
Key observation: Only the architected state needs to be precise, not the physical register state. So, we delay removing instructions from the reorder buffer until we are ready to “commit” to that state changing the architected registers.
Add completion logic to data path ...

To sustain CPI < 1, must be able to do multiple issues, commits, and reorder buffer execution launches and writes per cycle.

Not surprising design and validation teams are so large.
Power 5: By the numbers ...

Fetch up to 8 instructions per cycle.

Dispatch up to 5 instructions per cycle

Execute up to 8 instructions per cycle

Instruction fetch

Branch redirects

Interrupts and flushes

Out-of-order processing

Branch pipeline

Load/store pipeline

Fixed-point pipeline

Floating-point pipeline

Dispatch up to 5 instructions per cycle

Up to 200 instructions “in flight”

240 physical registers (120 int + 120 FP)

A thread may commit up to 5 instructions per cycle.
Note: Good branch prediction required
Because so many stages between predict and result!

BP = Branch prediction. On IBM Power 5, quite complex ... uses a predictor to predict the best branch prediction algorithm!
Recap: Dynamic Scheduling

Three big ideas: register renaming, data-driven detection of RAW resolution, bus-based architecture.

Very complex, but enables many things: out-of-order execution, multiple issue, loop unrolling, etc.

Has saved architectures that have a small number of registers: IBM 360 floating-point ISA, Intel x86 ISA.
Throughput Computing
Recall: Throughput and multiple threads

Goal: Use multiple instruction streams to improve (1) throughput of machines that run many programs (2) multi-threaded program execution time.

Example: Sun Niagara (32 instruction streams on a chip).

Difficulties: Gaining full advantage requires rewriting applications, OS, libraries.

Ultimate limiter: Amdahl’s law (application dependent). Memory system performance.
Throughput Computing

**Multithreading:** Interleave instructions from separate threads on the same hardware. Seen by OS as several CPUs.

**Multi-core:** Integrating several processors that (partially) share a memory system on the same chip.
Multi-Threading

(Static Pipelines)
Recall: Bypass network prevents stalls

Instead of bypass: Interleave threads on the pipeline to prevent stalls...
**Multithreading**

How can we guarantee no dependencies between instructions in a pipeline?

- One way is to interleave execution of instructions from different program threads on the same pipeline.

---

**Simple Multithreaded Pipeline**

Have to carry thread select down the pipeline to ensure correct state bits read/written at each pipe stage.

---

**Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe**

T1: LW r1, 0(r2)
T2: ADD r7, r1, r4
T3: XORI r5, r4, #12
T4: SW 0(r7), r5

T1: LW r5, 12(r1)

Last instruction in a thread always completes writeback before the next instruction in the same thread reads the register file.

---

**Introduction in 1964 by Seymour Cray**

4 CPUs, each run at 1/4 clock.

---

Many variants...
Multi-Threading
(Dynamic Scheduling)
Power 4 (predates Power 5 shown earlier)

Single-threaded predecessor to Power 5. 8 execution units in out-of-order engine, each may issue an instruction each cycle.
For most apps, most execution units lie idle

Observation:
Most hardware in an out-of-order CPU concerns physical registers. Could several instruction threads share this hardware?

Simultaneous Multi-threading ...

<table>
<thead>
<tr>
<th>Cycle</th>
<th>M</th>
<th>M</th>
<th>FX</th>
<th>FX</th>
<th>FP</th>
<th>FP</th>
<th>BR</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle</th>
<th>M</th>
<th>M</th>
<th>FX</th>
<th>FX</th>
<th>FP</th>
<th>FP</th>
<th>BR</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes
**Power 4**

Branch redirects

Instruction fetch

IF  IC  BP

D0  D1  D2  D3  Xfer  GD

Instruction crack and group formation

Interrupts and flushes

**Power 5**

Branch redirects

Instruction fetch

IF  IC  BP

D0  D1  D2  D3  Xfer  GD

Instruction crack and group formation

Interrupts and flushes

2 commits

(architected register sets)

2 fetch (PC), 2 initial decodes

Out-of-order processing

MP  ISS  RF  EX

BR  WB  Xfer

LD/ST

MP  ISS  RF  EA  DC  Fmt

MP  ISS  RF  EX

CP  WB  Xfer

FP  WB  Xfer

F6  WB  Xfer

Branch pipeline

Load/store pipeline

Fixed-point pipeline

Floating-point pipeline

2 fetches

2 initial decodes

 uc Regents Fall 2008 © UCB
Why only 2 threads? With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck.
Power 5 thread performance ...

Relative priority of each thread controllable in hardware.

For balanced operation, both threads run slower than if they “owned” the machine.
Multi-Core
Recall: Superscalar utilization by a thread

For an 8-way superscalar.

Observation: In many cases, the on-chip cache and DRAM I/O bandwidth is also underutilized by one CPU. So, let 2 cores share them.
Most of Power 5 die is shared hardware
Core-to-core interactions stay on chip

(1) Threads on two cores that use shared libraries conserve L2 memory.

(2) Threads on two cores share memory via L2 cache operations. Much faster than 2 CPUs on 2 chips.
Sun Niagara
The case for Sun’s Niagara ...

For an 8-way superscalar.

Observation:

Some apps struggle to reach a CPI == 1.

For throughput on these apps, a large number of single-issue cores is better than a few superscalars.
Niagara (original): 32 threads on one chip

8 cores:
- Single-issue, 1.2 GHz
- 6-stage pipeline
- 4-way multi-threaded
- Fast crypto support

Shared resources:
- 3MB on-chip cache
- 4 DDR2 interfaces
- 32G DRAM, 20 Gb/s
- 1 shared FP unit
- GB Ethernet ports

Die size: 340 mm² in 90 nm.
Power: 50–60 W

Sources: Hot Chips, via EE Times, Infoworld.
J Schwartz weblog (Sun COO)
The board that booted Niagara first-silicon

Source: J Schwartz weblog (then Sun COO, now CEO)
Used in Sun Fire T2000: “Coolthreads”

Claim: server uses 1/3 the power of competing servers.

Web server benchmarks used to position the T2000 in the market.
Project Blackbox

A data center in a 20-ft shipping container. Servers, air-conditioners, power distribution.
Just hook up network, power, and water ...
Holds 250 T1000 servers.

2000 CPU cores, 8000 threads.
Cache Coherency
Two CPUs, two caches, shared DRAM ...

CPU0:
LW R2, 16(R0)

CPU1:
LW R2, 16(R0)

CPU1:
SW R0, 16(R0)

View of memory no longer "coherent".

Loads of location 16 from CPU0 and CPU1 see different values!

What to do?

Write-through caches
The simplest solution ... one cache!

CPUs do not have internal caches.

Only one cache, so different values for a memory address cannot appear in 2 caches!

Multiple caches banks support read/writes by both CPUs in a switch epoch, unless both target same bank.

In that case, one request is stalled.
Not a complete solution ... good for L2.

For modern clock rates, access to shared cache through switch takes 10+ cycles.

Using shared cache as the L1 data cache is tantamount to slowing down clock 10X for LWs. Not good.

This approach was a complete solution in the days when DRAM row access time and CPU clock period were well matched.
Modified form: Private L1s, shared L2

Thus, we need to solve the cache coherency problem for L1 cache.

Advantages of shared L2 over private L2s:

- Processors communicate at cache speed, not DRAM speed.
- Constructive interference, if both CPUs need same data/instr.

Disadvantage: CPUs share BW to L2 cache...
Cache coherency goals...

1. Only one processor at a time has write permission for a memory location.

2. No processor can load a stale copy of a location after a write.
Implementation: Snoopy Caches

Each cache has the ability to "snoop" on memory bus transactions of other CPUs.

The bus also has mechanisms to let a CPU intervene to stop a bus transaction, and to invalidate cache lines of other CPUs.
Writes from 10,000 feet ...

For write-thru caches ...

To a first-order, reads will “just work” if write-thru caches implement this policy.

A “two-state” protocol (cache lines are “valid” or “invalid”).

1. Writing CPU takes control of bus.

2. Address to be written is invalidated in all other caches.

Reads will no longer hit in cache and get stale data.

3. Write is sent to main memory.

Reads will cache miss, retrieve new value from main memory.
Limitations of the write-thru approach

Every write goes to the bus.

Total bus write bandwidth does not support more than 2 CPUs, in modern practice.

To scale further, we need to use write-back caches.

Write-back big trick: keep track of whether other caches also contain a cached line. If not, a cache has an “exclusive” on the line, and can read and write the line as if it were the only CPU.

For details, take CS 152 and CS 252 ...
Simultaneous Multithreading: Instructions streams can share an out-of-order engine economically.

Multi-core: Once instruction-level parallelism run dry, thread-level parallelism is a good use of die area.
Next Monday:

This Friday:
Thanksgiving Holiday, No Meeting

Final Presentation
Fri, Dec 5