Implementing out-of-order execution processors

IBM 360/91
High performance substrate
Historical perspective

1960: IBM Stretch
1962: ILLIAC II
1967: CDC 6600
1961: IBM Stretch
1962: ILLIAC II
1967: CDC 6600
1974: Data flow
1976: Cray 1
1977: DEC VAX
1980: Berkeley RISC
1981: Stanford MIPS
1983: Yale VLIW
1985: Berkeley HPS
1980: Berkeley RISC
1981: Stanford MIPS
1983: Yale VLIW
1985: Berkeley HPS
1992: IBM PowerPC 600
1995: Intel Pentium Pro
1996: MIPS R10000
1998: DEC Alpha 21264

Pipeline
1960
Out-of-order
1970
RISC
1980
VLIW
1990
Superscalar
2000
Era of Supercomputers
Era of RISC
Era of Superscalar
Historical Context: CDC6600

- Mainframe computer in 1964
- Superscalar design with 10 parallel functional units
- Functional units not pipelined
- Instructions fetched and issued faster than execution
Scoreboarding

• Scoreboard
  • A central control to determine dependencies and prevent hazards

• Steps:
  • Issue
    • Prevents WAW and Structural hazards
  • Read Operands
    • Leads to OOO
  • Execution
    • Followed by notification to scoreboard
  • Write result
    • Checks for WAR
Architecture

Structural hazard: delaying the issue

WAW data hazard: delaying the issue

RAW data hazard: wait until the values of the source registers are available in the registers

WAR data hazard: delaying the write if a WAR hazard exists
Parts of Scoreboard

• Instructional status
  • Indicates which of the 4 steps an instruction is in

• Functional unit status
  • State of functional unit
  • 9 such states. Eg. busy state

• Register result status
  • Indicates the functional unit that will write each register
## Scoreboard Structure

### Instruction status

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Read Operands</th>
<th>Ex complete</th>
<th>Write result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LF F6 34(R2)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>LF F2 45(R3)</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTF F0 F2 F4</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBF F8 F5 F2</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVF F10 F0 F6</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDF F6 F8 F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Functional unit status

<table>
<thead>
<tr>
<th>FU no</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Dest</th>
<th>Src1</th>
<th>Src2</th>
<th>P1</th>
<th>P2</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Integer</td>
<td>Y</td>
<td>loadf</td>
<td>F2</td>
<td>R3</td>
<td></td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Mult1</td>
<td>Y</td>
<td>multf</td>
<td>F0</td>
<td>F2</td>
<td>F4</td>
<td>1</td>
<td></td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>Mult2</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Add</td>
<td>Y</td>
<td>subf</td>
<td>F8</td>
<td>F6</td>
<td>F2</td>
<td></td>
<td>1</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>5</td>
<td>Divide</td>
<td>Y</td>
<td>divf</td>
<td>F10</td>
<td>F0</td>
<td>F6</td>
<td>2</td>
<td></td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

### Register result status

<table>
<thead>
<tr>
<th>FU no</th>
<th>F0</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
<th>F12</th>
<th>……</th>
<th>F30</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
Scoreboarding Limitations

• Number of entries in scoreboard
  • Determines look ahead for independent instructions

• Number and types of functional units
  • Affect structural dependences

• Centralized control
  • Only 1 instruction can be issued at a time

• Low throughput
  • Stalls for WAW and WAR
Historical Context: IBM 360/91

- Tomasulo's Algorithm implemented for the Floating Point operations
- It had only 2 functional units: 1 adder and 1 multiplier/divider
- Had only 4 double precision FP registers
Tomasulo's Goals

- The design must identify existence of a dependency
- It must sequence the instructions correctly
- It must allow independent instructions to overlap
Examples

RAW Hazard
LD F0 FLB1
MD F0 FLB2
• It is a true dependency
• Second operation must not proceed until the first one is complete.
• F0 cannot be used until the recent operations using it as sink are complete

Independent Instruction
LD F0, FLB1
MD F2, FLB2
Tomasulo's Algorithm w.o CDB

- Maintaining precedence using control bits on registers (busy bit scheme) for true dependencies
  - Set control bit when register is a sink
  - Transmit data to waiting unit when register gets result

- Achieving parallelism through use of different registers is programmer's responsibility for WAW and WAR

- Meets the dependency goals but not performance goal
  - There is a stall for data dependences
  - Programmer resolves false dependences in code
Figure 1 Data registers and transfer paths without CDB.
Reservation Stations

• To efficiently utilize execution units during stalls for true dependences

• Example:

LD F0, D   F0=D
LD F2, C   F2=C
LD F4, B   F4=B
MD F0, E   F0 = D * E
AD F2, F0  F2 = C + D * E
AD F4, A   F4 = A + B
AD F2, F4  F2 = A + B + C + D * E
Removing False Dependences

Common Data Bus
- Efficiently moves data to allow concurrency
- Every unit that alters a register feeds into CDB
- Every unit that requires a register is fed by CDB
- These units are recognized by identifier called tag

Register Renaming
- Tagging is the mechanism
- Removes false dependences
  - WAW is resolved since register keeps track of last operation tag that updated it
  - WAR is resolved using in-order decode
Figure 4  Data registers and transfer paths, including CDB and reservation stations.
Details on register renaming

- Output dependence (WAW hazard)
  - Scoreboard: Instruction issue is stalled
  - Tomasulo: Resolved by changing the pointer to the reservation for pending update

- Anti dependence (WAR hazard)
  - Scoreboard: Write back is stalled
  - Tomasulo: Resolved by early dispatch with register values
Steps in Tomasulo's Algorithm

- **Issue**
  - Instruction issued in-order
  - Issue to the reservation station with the operands or track the FUs that will produce operands
  - Stall if no reservation station is available

- **Execute**
  - Instructions are executed when all operands become available
  - Many instructions executed simultaneously

- **Write results**
  - Results are written to CDB
  - CDB writes to registers and reservation stations
Limitations of Tomasulo's Algorithm

- The number of CDBs limits bandwidth
  - Increasing CDBs increases complexity and cost
- Hard to debug because of imprecise interrupts

➡️ Dynamic scheduling with in-order commit HPS
## Scoreboard vs. Tomasulo

<table>
<thead>
<tr>
<th></th>
<th>Scoreboard</th>
<th>Tomasulo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue</td>
<td>When FU free</td>
<td>When RS free</td>
</tr>
<tr>
<td>Read operands</td>
<td>From reg file</td>
<td>From reg file, CDB</td>
</tr>
<tr>
<td>Write operands</td>
<td>To reg file</td>
<td>To CDB</td>
</tr>
<tr>
<td>Structural hazards</td>
<td>Functional units</td>
<td>Reservation stations</td>
</tr>
<tr>
<td>WAW, WAR hazards</td>
<td>Problem</td>
<td>No problem</td>
</tr>
<tr>
<td>Register renaming</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Instructions</td>
<td>No limit</td>
<td>1 per cycle (per CDB)</td>
</tr>
<tr>
<td>completing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions</td>
<td>1 (per set of read ports)</td>
<td>No limit</td>
</tr>
<tr>
<td>beginning exec</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Slide source: “Instruction Level Parallelism - Tomasulo” lecture notes by Dean Tullsen*
Summary

Principles:
• In-order execution for RAW hazards
• Renaming registers for WAR and WAW

Components:
• Reservation Stations
  • Buffer operands for instructions waiting to execute
  • Virtual registers implementing register renaming
• Common Data Bus
  • Hardware implementation for concurrency with multiple FUs
  • Use tags for broadcasting data
  • Allow more than one instruction to reach execution stage simultaneously
Historical perspective revisited

1960: IBM Stretch
1962: ILLIAC II
1967: CDC 6600
1969: Cray 1
1970: Data flow
1974: Pipeline
1976: Cray 1
1977: DEC VAX
1980: Berkeley RISC
1981: Stanford MIPS
1983: Yale VLIW
1985: Berkeley HPS
1992: IBM PowerPC 600
1995: Intel Pentium Pro
1996: MIPS R10000
1998: DEC Alpha 21264

Era of Supercomputers
Era of RISC
Era of Superscalar
HPS as restricted data flow

completed!
Requirements for high performance

- High degree of HW concurrency available
- Well utilized HW concurrency
Instruction set architecture of HPS

- Fixed 32 bit instruction length  
  - RISC - like
- Two operations per instruction  
  - Can be dependent or independent of each other  
  - VLIW - like
- 16 architectural registers  
  - 4 special registers  
  - 4 safe registers  
  - 8 unsafe registers  
  - CISC - like
Instruction handling in HPS

• Instructions can be fetched from…
  – Instruction cache
    • Once fetched, instruction is decoded for both execution and refill of node cache
  – Node cache
    • Design concept is similar to trace cache
    • It stores instructions in decoded form
    • It holds up to 1K entries

• HPS continues to execute beyond branches
  – Speculative execution - more on this later
Register renaming revisited

$2 \leftarrow 0 + 3$
$1 \leftarrow 0 \times 2$
$2 \leftarrow 2 + 3$
$3 \leftarrow 2 \times 1$

Register Alias Table

<table>
<thead>
<tr>
<th>RDY</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>Yes</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

Value Buffer

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
</tr>
<tr>
<td>0x1</td>
</tr>
<tr>
<td>0x2</td>
</tr>
<tr>
<td>0x3</td>
</tr>
<tr>
<td>0x4</td>
</tr>
<tr>
<td>0x5</td>
</tr>
<tr>
<td>0x6</td>
</tr>
<tr>
<td>0x7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>6.0</td>
</tr>
<tr>
<td>0x1</td>
<td>3.5</td>
</tr>
<tr>
<td>0x2</td>
<td>10.0</td>
</tr>
<tr>
<td>0x3</td>
<td>7.8</td>
</tr>
<tr>
<td>0x4</td>
<td>13.8</td>
</tr>
<tr>
<td>0x5</td>
<td>82.8</td>
</tr>
<tr>
<td>0x6</td>
<td>21.6</td>
</tr>
<tr>
<td>0x7</td>
<td></td>
</tr>
</tbody>
</table>

Node Table

<table>
<thead>
<tr>
<th>node op</th>
<th>Result tag</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>add</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>mult</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>mult</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

Fired

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8</td>
</tr>
<tr>
<td>0xF</td>
</tr>
</tbody>
</table>
Designing node tables
Decoupled architecture

Front-end
- Instruction fetch
- Instruction decode
- Branch prediction
- Trace cache
- ... and more

Node table
- "the slack"

Back-end
- Multiple execution units
- Data memory access
- Out-of-order execution
- In-order commit
- ... and more

I-cache

D-cache
Retirement mechanism (1)

- Instructions retire in-order in HPS
  - Retirement finalizes the state (Reg/Mem) changes made by the instruction
  - Why is in-order retirement enforced in modern processors?
    - Precise need to restart for I/O and timer interrupts
    - Recovering from page fault
    - Easier debugging
    - Graceful recovery from arithmetic exceptions
## Retirement mechanism (2)

### Active Instruction Table

<table>
<thead>
<tr>
<th>Op 1</th>
<th>Op 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDY</td>
<td>TAG</td>
</tr>
<tr>
<td>Y</td>
<td>32</td>
</tr>
</tbody>
</table>

### Result distribution

<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Instructions

- **Inst i-1**: `add $0, $1, $2` & `sub $5, $6, $3`
  - Retired
- **Inst i**: `div $1, $5, $0` & `mult $2, $5, $0`
  - Retired
- **Inst i+1**: `beq $7, $4, 100` & `store $8, 100(2)`
  - Retired
- **Inst i+2**: `add $9, $10, $12` & `store $7, 200(2)`
  - Exception
Speculative execution (1)

• Name of the game
  – Guess branch outcome and execute as if the prediction was correct

• Basic idea
  – If turns out to correct: confirm state changes
  – If not: revert back to the state when prediction was made
    • This means that to speculatively execute, the state at the time of prediction must be backed up
Speculative execution (2)

- When correct
  - Allow next branch to proceed
  - Mark branch op as ready in AIT

- When incorrect
  - Redirect instruction stream
  - Allow next branch to proceed
  - Restore RAT entries
  - Invalidate node table entries younger than branch
  - Invalidate mem buffer entries younger than branch
  - Invalidate AIT younger than branch

old

<table>
<thead>
<tr>
<th>Inst i-1</th>
<th>div $1, $5, $0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst i</td>
<td>beq $7, $4, 100</td>
</tr>
<tr>
<td>Inst i+1</td>
<td>add $9, $10, $12</td>
</tr>
<tr>
<td>Inst i+2</td>
<td>store $8, 100($2)</td>
</tr>
<tr>
<td>Inst i+3</td>
<td>mult $2, $5, $0</td>
</tr>
<tr>
<td>Inst i+4</td>
<td>bne $2, $1, 200</td>
</tr>
</tbody>
</table>

new
Performance results (1)

- Evaluated systems on RTL simulator
  - RISC, RISC-opt, HPSm, HPSm-opt
- Reasoning for picking out benchmarks
  - Small enough to do hand-translation
  - Procedure / branch intensive
  - Well-performed on RISC II

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RISC A</th>
<th>RISC B</th>
<th>HPSm A</th>
<th>HPSm B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Towers(18)</td>
<td>3,729ms</td>
<td>2,171ms</td>
<td>1,075ms</td>
<td>852ms</td>
</tr>
<tr>
<td>Acker(3,6)</td>
<td>3,164ms</td>
<td>2,484ms</td>
<td>230ms</td>
<td>207ms</td>
</tr>
<tr>
<td>Quicksort</td>
<td>867ms</td>
<td>764ms</td>
<td>175ms</td>
<td>175ms</td>
</tr>
<tr>
<td>BenchE</td>
<td>492us</td>
<td>337us</td>
<td>198us</td>
<td>116us</td>
</tr>
<tr>
<td>BenchF</td>
<td>178us</td>
<td>106us</td>
<td>50us</td>
<td>50us</td>
</tr>
<tr>
<td>BenchH</td>
<td>188us</td>
<td>148us</td>
<td>67us</td>
<td>56us</td>
</tr>
</tbody>
</table>
Performance results (2)

- **Cycle time**
  - HPS cycle time: 100ns
  - RISC II cycle time: 330ns

- **Reasoning behind different cycle times**
  - Large register file of RISC II makes it slow
  - HPSm is equipped with faster cache memory

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RISC A</th>
<th>RISC B</th>
<th>HPSm A</th>
<th>HPSm B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Towers(18)</td>
<td>11.30M</td>
<td>6.59M</td>
<td>10.75M</td>
<td>8.52M</td>
</tr>
<tr>
<td>Acker(3,6)</td>
<td>9.39M</td>
<td>7.12M</td>
<td>2.30M</td>
<td>2.07M</td>
</tr>
<tr>
<td>Quicksort</td>
<td>2.63M</td>
<td>2.31M</td>
<td>1.75M</td>
<td>1.75M</td>
</tr>
<tr>
<td>BenchE</td>
<td>1.50K</td>
<td>1.02K</td>
<td>1.98K</td>
<td>1.16K</td>
</tr>
<tr>
<td>BenchF</td>
<td>0.54K</td>
<td>0.32K</td>
<td>0.50K</td>
<td>0.50K</td>
</tr>
<tr>
<td>BenchH</td>
<td>0.57K</td>
<td>0.45K</td>
<td>0.67K</td>
<td>0.56K</td>
</tr>
</tbody>
</table>
Summary of HPS

• Precursor to modern superscalar μP
  – Multiple functional units
  – Multiple instruction issue
  – Out-of-order execution, in-order commit
  – Speculative execution
Types of Dependences

- Data Dependence
- Name Dependence
- Control Dependence
Data Dependences

• Instructions depend on each other for actual data
• True dependence
• Ex:

  Loop:  L.D  F0, 0(R1)
          ADD.D  F4, F0, F2
          S.D  F4,0(R1)

• Correct order of execution needs to be ensured
Name Dependences

- Arise when instructions use the same register but there is no actual data flow
- False dependence
- Types:
  - Antidependence: \( j \) writes to location that \( i \) reads
  - Output dependence: \( i \) and \( j \) write to the same location
- Can be resolved with register renaming
Control Dependence

- The ordering for control blocks need to be maintained
- Ex: a statement from \textit{then} block cannot be executed before the corresponding \textit{if} condition
- Necessary for maintaining correctness of the code
- The correct order of execution needs to be maintained
Types of Hazards

- **Structural Hazard**
  - Due to finite nature of resources

- **Data Hazard**
  - WAR due to antidependence
  - RAW due to true data dependence
  - WAW due to output dependence

- **Control Hazard**
Overcoming Hazards Statically

• Techniques such as forwarding
• Drawbacks:
  o Limited applicability
  o Unnecessary stalls and reduced pipeline throughput

• Compiler scheduling in loop unrolling and other techniques
• Drawbacks
  o Too many registers
  o Leads to large code size
Overcoming Hazards Dynamically

• Independent instructions continue on stalls through OOO

• Techniques:
  o Scoreboarding
  o Tomasulo's Algorithm