Register Data Flow Techniques

• **Register Data Flow**
  – Resolving Anti-dependences
  – Resolving Output Dependences
  – Resolving True Data Dependences

• **Tomasulo’s Algorithm** [Tomasulo, 1967]
  – Modified IBM 360/91 Floating-point Unit
  – Reservation Stations
  – Common Data Bus
  – Register Tags
  – Operation of Dependency Mechanisms
The Big Picture

INSTRUCTION PROCESSING CONSTRAINTS

- Resource Contention (Structural Dependences)
- Control Dependences
- Code Dependences
  - (RAW) True Dependences
  - (WAR) Anti-Dependences
- Data Dependences
  - Storage Conflicts
- Output Dependences (WAW)
Each ALU Instruction:

\[ R_i \leftarrow F_n (R_j, R_k) \]

Dest. Funct. Source
Reg. Unit Registers

“Register Transfer”

Need Availability of \( F_n \) (Structural Dependences)

Need Availability of \( R_j, R_k \) (True Data Dependences)

Need Availability of \( R_i \) (Anti-and output Dependences)
Causes of (Register) Storage Conflict

**REGISTER RECYCLING**
- MAXIMIZE USE OF REGISTERS
- MULTIPLE ASSIGNMENTS OF VALUES TO REGISTERS

**OUT OF ORDER ISSUING AND COMPLETION**
- LOSE IMPLIED PRECEDENCE OF SEQUENTIAL CODE
- LOSE 1-1 CORRESPONDENCE BETWEEN VALUES AND REGISTERS

---

First instance of Ri

Second instance of Ri
Contribution to Register Recycling

COMPILER REGISTER ALLOCATION

CODE GENERATION

REG. ALLOCATION

Single Assignment, Symbolic Reg.

Map Symbolic Reg. to Physical Reg.
Maximize Reuse of Reg.

INSTRUCTION LOOPS

For \( k = 1; k \leq 10; k++ \)
\[
t += a[i][k] \times b[k][j];
\]

Reuse Same Set of Reg. in Each Iteration
Overlapped Execution of Different Iterations

"Spill code" (if not enough registers)
Resolving Anti-Dependences

Must Prevent (2) from completing before (1) is dispatched.

(1) R4 ← R3 + 1
(2) R3 ← R5 + 1

STALL DISPATCHING
DELAY DISPATCHING OF (2)
REQUIRE RECHECKING AND REACCESSING

COPY OPERAND
COPY NOT-YET-USED OPERAND TO PREVENT BEING OVERWRITTEN
MUST USE TAG IF ACTUAL OPERAND NOT-YET-AVAILABLE

RENAMED REGISTER
R3 ← …
R3' ← …

HARDWARE ALLOCATION
R3 ← R3
R3' ← R3'
Resolving Output Dependences

Must Prevent (3) from completing before (1) completes.

STALL DISPATCHING/ISSUING

DENOTE OUTPUT DEPENDENCE

HOLD DISPATCHING UNTIL RESOLUTION OF DEPENDENCE

ALLOW DECODING OF SUBSEQUENT INSTRUCTIONS

RENAME REGISTER

HARDWARE ALLOCATION

\[
\begin{align*}
(1) & \quad R3 \leftarrow R3 \text{ op } R5 \\
(3) & \quad R3 \leftarrow R5 + 1
\end{align*}
\]

\[
\begin{align*}
R3 & \leftarrow \ldots \\
R3' & \leftarrow R3 \\
R3' & \leftarrow \ldots \\
R3' & \leftarrow R3'
\end{align*}
\]
Register Renaming

Register Renaming Resolves:

- Anti-Dependences
- Output Dependences

Design of Redundant Registers

Number:
- One
- Multiple

Allocation:
- Fixed for Each Register
- Pooled for all Registers

Location:
- Attached to Register File (Centralized)
- Attached to functional units (Distributed)
Register Renaming in the RIOS-I FPU

FPU Register Renaming

<table>
<thead>
<tr>
<th>OP</th>
<th>T</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAD</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FAD</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Map table 32 x 6

Free List

Head: 32 33 34 35 36 37 38 39

Pending Target Return Queue

7

Simplified FPU Register Model

Incoming FPU instructions pass through a renaming table prior to decode

The 32 architectural registers are remapped to 40 physical registers

Physical register names are used within the FPU

Complex control logic maintains active register mapping

Fload R7 <= Mem[32]

…

<= R7 (actual last use)

Fload R7 <= Mem[32 alloc]
Resolving True Data Dependences

STALL DISPATCHING
ADVANCE INSTRUCTIONS

(1) R2 ← R1 + 1
(2) R3 ← R2
(3) R4 ← R3

1) Read register(s), get “IOU” if not ready
2) Advance to reservation station
3) Wait for “IOU” to show up
4) Execute
Embedded “Data Flow” Engine

- Read register or
- Assign register tag
- Advance instructions to reservation stations

- Monitor reg. tag
- Receive data being forwarded
- Issue when all operands ready

Dispatch Buffer

Dispatch

Reservation Stations

“Dynamic Execution”

Branch

Completion Buffer

Complete

© 2005 Mikko Lipasti
Tomasulo’s Algorithm [Tomasulo, 1967]
IBM 360/91 FPU

- **Multiple functional units (FU’s)**
  - Floating-point add
  - Floating-point multiply/divide
- **Three register files (pseudo reg-reg machine in floating-point unit)**
  - (4) floating-point registers (FLR)
  - (6) floating-point buffers (FLB)
  - (3) store data buffers (SDB)
- **Out of order instruction execution:**
  - After decode the instruction unit passes all floating point instructions (in order) to the floating-point operation stack (FLOS) [actually a queue, not a stack]
  - In the floating point unit, instructions are then further decoded and issued from the FLOS to the two FU’s
- **Variable operation latencies:**
  - Floating-point add: 2 cycles
  - Floating-point multiply: 3 cycles
  - Floating-point divide: 12 cycles
- **Goal:** achieve concurrent execution of multiple floating-point instructions, in addition to achieving one instruction per cycle in instruction pipeline
Dependence Mechanisms

Two Address IBM 360 Instruction Format:

\[ R1 \leftarrow R1 \text{ op } R2 \]

Major dependence mechanisms:

- **Structural (FU) dependence** = \( \rightarrow \) virtual FU’s
  - Reservation stations

- **True dependence** = \( \rightarrow \) pseudo operands + result forwarding
  - Register tags
  - Reservation stations
  - Common data bus (CDB)

- **Anti-dependence** = \( \rightarrow \) operand copying
  - Reservation stations

- **Output dependence** = \( \rightarrow \) register renaming + result forwarding
  - Register tags
  - Reservation stations
  - Common data bus (CDB)
IBM 360/91 FPU

- Storage Bus
- Floating Point Buffers (FLB)
  - 6
  - 5
  - 4
  - 3
  - 2
  - 1

- Instruction Unit
  - Floating Point Registers (FLR)
    - 8
    - 4
    - 2
    - 0
  - Floating Point Buffers (FLB)
    - 6
    - 5
    - 4
    - 3
    - 2
    - 1

- Decoder
- Common Data Bus (CDB)
- Adder
- Result
- Multiply/Divide

- Store Data Buffers (SDB)
- Tags
- Busy Bits
- Stack (FLOS)
- Tags
- Control

© 2005 Mikko Lipasti
Reservation Stations

- Used to collect operands or pseudo operands (tags).
- Associate more than one set of buffering registers (control, source, sink) with each FU, => virtual FU’s.
- Add unit: three reservation stations
- Multiply/divide unit: two reservation stations

<table>
<thead>
<tr>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 implies valid data</td>
<td>Source value</td>
<td>0 implies valid data</td>
<td>Source value</td>
</tr>
</tbody>
</table>
Common Data Bus (CDB)

- CDB is fed by all units that can alter a register (or supply register values) and it feeds all units which can have a register as an operand.

- Sources of CDB:
  - Floating-point buffers (FLB)
  - Two FU’s (add unit and the multiply/divide unit)
  - 6 FLB + 3 addRS + 2 mul/divRS = 11 unique sources
  - 3 physical sources (FLB, adder, mul/div)

- Destinations of CDB:
  - Reservation stations
  - Floating-point registers (FLR)
  - Store data buffers (SDB)
  - (5 RS x 2) + 4 FLR + 3 SDB : CDB has 17 destinations

- Circuit design very challenging
  - 3 physical sources must arbitrate for access to CDB
  - Tag + data must be driven to 17 destinations
Register Tags

• Every source of a register value must be uniquely identified by its own tag value.
  – (6) FLB’s
  – (5) reservation stations (3 with add unit, 2 with multiply/divide unit)
    = = > 4-bit tag is needed to identify the 11 potential sources

• Every destination of a register value must carry a tag field.
  – (5) “sink” entries of the reservation stations
  – (5) “source” entries of the reservation stations
  – (4) FLR’s
  – (3) SDB’s
    = = > a total of 17 tag fields are needed (i.e. 17 places that need tags)
Operation of Dependence Mechanisms

1. Structural (FU) dependence = > virtual FU’s

   – FLOS can hold and decode up to 8 instructions.
   – Instructions are dispatched to the 5 reservation stations (virtual FU’s) even though there are only two physical FU’s.
   – Hence, structural dependence does not stall dispatching.

2. True dependence = > pseudo operands + result forwarding

   – If an operand is available in FLR, it is copied to a res. station entry.
   – If an operand is not available (i.e. there is pending write), then a tag is copied to the reservation station entry instead. This tag identifies the source of the pending write. This instruction then waits in its reservation station for the true dependence to be resolved.
   – When the operand is finally produced by the source (ID of source = tag value), this source unit asserts its ID, i.e. its tag value, on the CDB followed by broadcasting of the operand on the CDB.
   – All the reservation station entries and the FLR entries and SDB entries carrying this tag value in their tag fields will detect a match of tag values and latch in the broadcasted operand from the CDB.
   – Hence, true dependence does not block subsequent independent instructions and does not stall a physical FU. Forwarding also minimizes delay due to true dependence.
### Example 1

#### i: R2 <= R0 + R4

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>1</td>
<td>0</td>
<td>6.0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Adder</td>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>

#### j: R8 <= R0 + R2 (RAW on R2)

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
<th>Busy</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>4</td>
<td>0</td>
<td>6.0</td>
<td>0</td>
<td>0</td>
<td>6.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>x</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Mult/Div</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>j</th>
<th>Busy</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>16.0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>10.0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>2</td>
<td>7.8</td>
</tr>
</tbody>
</table>

© 2005 Mikko Lipasti
Operation of Dependence Mechanisms

3. **Anti-dependence** = > operand copying

– If an operand is available in FLR, it is copied to a reservation station entry.

– By copying this operand to the reservation station, all anti-dependences due to future writes to this same register are resolved.

– Hence, the reading of an operand is not delayed, possibly due to other dependences, and subsequent writes are also not delayed.
### Example 2

#### Cycle #1

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
<th>Busy</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>6.0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>5.0</td>
</tr>
</tbody>
</table>

**DISPATCHED INSTRUCTION(S):**

- **i, j**

#### Cycle #2

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
<th>Busy</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3.5</td>
<td>0</td>
<td>7.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>3.5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**DISPATCHED INSTRUCTION(S):**

- **k**

#### Cycle #3

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
<th>Busy</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3.5</td>
<td>0</td>
<td>7.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>10.0</td>
</tr>
</tbody>
</table>

**DISPATCHED INSTRUCTION(S):**

- **i, j**

---

**i:** $R4 \leftarrow R0 \times R8$

**j:** $R0 \leftarrow R4 \times R2$ (RAW on R4)

**k:** $R2 \leftarrow R2 + R8$ (WAR on R2)
3. **Output dependence** = > register renaming + result forwarding

   - If a register is waiting for a pending write, its tag field will contain the ID, or tag value, of the source for that pending write.

   - When that source eventually produces the result, that result will be written into the register via the CDB.

   - It is possible that prior to the completion of the pending write, another instruction can come along and also has that same register as its destination register.

   - If this occurs, the operands (or pseudo operands) needed by this instruction are still copied to an available reservation station. In addition, the tag field of the destination register of this instruction is updated with the ID of this new reservation station, i.e. the old tag value is overwritten. This will ensure that the said register will get the latest value, i.e. the late completing earlier write cannot overwrite a later write.

   - Hence, the output dependence is resolved without stalling a physical functional unit, not requiring additional buffers to ensure sequential write back to the register file.
What if j causes FP overflow exception?
- where is R4?
- it is lost => imprecise exceptions!

Example 3

Cycle #1

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>1</td>
<td>0</td>
<td>6.0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adder</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DISPATCHED INSTRUCTION(S):

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>4</td>
<td>0</td>
<td>6.0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mult/Div</td>
<td>i</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Busy | Tag | Data |
-----|-----|------|
0    |     | 6.0  |
2    | x   | 3.5  |
4    | x   | 10.0 |
8    |     | 7.8  |

Cycle #2

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>1</td>
<td>0</td>
<td>6.0</td>
<td>4</td>
</tr>
<tr>
<td>k</td>
<td>2</td>
<td>0</td>
<td>6.0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>7.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adder</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DISPATCHED INSTRUCTION(S):

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>4</td>
<td>0</td>
<td>6.0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>7.8</td>
</tr>
<tr>
<td></td>
<td>Mult/Div</td>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>8</td>
<td></td>
<td>7.8</td>
<td></td>
</tr>
</tbody>
</table>

Busy | Tag | Data |
-----|-----|------|
0    |     | 6.0  |
2    | x   | 1    |
4    | x   | 2    |
8    |     | 7.8  |

Cycle #3

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>1</td>
<td>0</td>
<td>6.0</td>
<td>4</td>
</tr>
<tr>
<td>k</td>
<td>2</td>
<td>0</td>
<td>6.0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>7.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adder</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DISPATCHED INSTRUCTION(S):

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>4</td>
<td>0</td>
<td>6.0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>7.8</td>
</tr>
<tr>
<td></td>
<td>Mult/Div</td>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>8</td>
<td></td>
<td>7.8</td>
<td></td>
</tr>
</tbody>
</table>

Busy | Tag | Data |
-----|-----|------|
0    |     | 6.0  |
2    | x   | 1    |
4    | x   | 2    |
8    |     | 7.8  |

© 2005 Mikko Lipasti
Summary of Tomasulo’s Algorithm

- Supports out of order execution of instructions.
- Resolves dependences dynamically using hardware.
- Attempts to delay the resolution of dependencies as late as possible.
- Structural dependence does not stall issuing; virtual FU’s in the form of reservation stations are used.
- Output dependence does not stall issuing; copying of old tag to reservation station and updating of tag field of the register with pending write with the new tag.
- True dependence with a pending write operand does not stall the reading of operands; pseudo operand (tag) is copied to reservation station.
- Anti-dependence does not stall write back; earlier copying of operand awaiting read to the reservation station.
- Can support sequence of multiple output dependences.
- Forwarding from FU’s to reservation stations bypasses the register file.
# Tomasulo vs. Modern OOO

<table>
<thead>
<tr>
<th></th>
<th>IBM 360/91</th>
<th>Modern</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Width</strong></td>
<td>1</td>
<td>14+</td>
</tr>
<tr>
<td><strong>Structural hazards</strong></td>
<td>2 FPU</td>
<td>Many FU</td>
</tr>
<tr>
<td><strong>Single CDB</strong></td>
<td></td>
<td>Many busses</td>
</tr>
<tr>
<td><strong>Anti-dependences</strong></td>
<td>Operand copy</td>
<td>Renamed reg. tag</td>
</tr>
<tr>
<td><strong>Output dependences</strong></td>
<td></td>
<td>Reg. renaming</td>
</tr>
<tr>
<td><strong>True dependences</strong></td>
<td></td>
<td>Tag-based forw. Tag-based forw.</td>
</tr>
<tr>
<td><strong>Exceptions</strong></td>
<td>Imprecise</td>
<td>Precise (ROB)</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>3 x 66” x 15” x 78”</td>
<td>&lt; 1 chip</td>
</tr>
<tr>
<td><strong>60ns cycle time</strong></td>
<td></td>
<td>300ps</td>
</tr>
<tr>
<td><strong>11-12 gate delays per pipe stage</strong></td>
<td>&lt; $1 million</td>
<td></td>
</tr>
<tr>
<td><strong>IMPLEMENTATION</strong></td>
<td></td>
<td>&lt; $100</td>
</tr>
</tbody>
</table>
Example 4

i: \[ R4 \leftarrow R0 + R8 \]

j: \[ R2 \leftarrow R0 \times R4 \]

k: \[ R4 \leftarrow R4 + R8 \]

l: \[ R8 \leftarrow R4 \times R2 \]
Can Tomasulo’s algorithm reach dataflow limit of 8?
Example 4

CYCLE #1

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adder

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Mult/Div

<table>
<thead>
<tr>
<th>BusyTag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.0</td>
</tr>
<tr>
<td>2</td>
<td>3.5</td>
</tr>
<tr>
<td>4</td>
<td>10.0</td>
</tr>
<tr>
<td>8</td>
<td>7.8</td>
</tr>
</tbody>
</table>

DISPATCHED INSTRUCTION(S): ____________

CYCLE #2

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adder

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Mult/Div

<table>
<thead>
<tr>
<th>BusyTag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.0</td>
</tr>
<tr>
<td>2</td>
<td>3.5</td>
</tr>
<tr>
<td>4</td>
<td>10.0</td>
</tr>
<tr>
<td>8</td>
<td>7.8</td>
</tr>
</tbody>
</table>

DISPATCHED INSTRUCTION(S): ____________

CYCLE #3

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adder

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Mult/Div

<table>
<thead>
<tr>
<th>BusyTag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.0</td>
</tr>
<tr>
<td>2</td>
<td>3.5</td>
</tr>
<tr>
<td>4</td>
<td>10.0</td>
</tr>
<tr>
<td>8</td>
<td>7.8</td>
</tr>
</tbody>
</table>

DISPATCHED INSTRUCTION(S): ____________

DISPATCHED INSTRUCTION(S): ____________
## Example 4

**CYCLE #4**

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adder

**CYCLE #5**

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adder

**DISPATCHED INSTRUCTION(S):** ______________

**CYCLE #6**

<table>
<thead>
<tr>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adder

**DISPATCHED INSTRUCTION(S):** ______________
"Dataflow Engine" for Dynamic Execution

- **Dispatch Buffer**
- **Dispatch**
- **Reg. File**
- **Ren. Reg.**
- **Branch**
- **Integer**
- **Integer**
- **Float.-Point**
- **Load/Store**
- **Compl. Buffer (Reorder Buff.)**
- **Complete**

Managed as a queue; Maintains sequential order of all Instructions in flight ("takeoff" = dispatching; "landing" = completion)

Allocate Reorder Buffer entries

Reservation Stations

Forwarding results to Res. Sta. & rename registers

Register Write Back
Instruction Processing Steps

• **DISPATCH:**
  - Read operands from Register File (RF) and/or Rename Buffers (RRB)
  - Rename destination register and allocate RRB entry
  - Allocate Reorder Buffer (ROB) entry
  - Advance instruction to appropriate Reservation Station (RS)

• **EXECUTE:**
  - RS entry monitors bus for register Tag(s) to latch in pending operand(s)
  - When all operands ready, issue instruction into Functional Unit (FU) and deallocate RS entry (no further stalling in execution pipe)
  - When execution finishes, broadcast result to waiting RS entries, RRB entry, and ROB entry

• **COMPLETE:**
  - Update architected register from RRB entry, deallocate RRB entry, and if it is a store instruction, advance it to Store Buffer
  - Deallocate ROB entry and instruction is considered architecturally completed
Reservation Station Implementation

- Reservation Stations: distributed vs. centralized
  - Wakeup: benefit to partition across data types
  - Select: much easier with partitioned scheme
    - Select 1 of n/4 vs. 4 of n
• **Merge RS and ROB => Register Update Unit (RUU)**
  – Inefficient, hard to scale
  – Perhaps of interest only to historians
Scheduling loop

• Wakeup dominated by wire delay
  – More compact RS => higher frequency

Select logic

Wakeup logic

broadcast the tag of the selected inst

issue to FU

Ready - request

selected

grant 0
request 0

grant 1
request 1

……

grant n
request n

ReadyL | TagL | TagR | ReadyR
Scheduler Designs

• Data-Capture Scheduler
  – keep the most recent register value in reservation stations
  – Data forwarding and wakeup are combined
Scheduler Designs

• Non-Data-Capture Scheduler
  – keep the most recent register value in RF (physical registers)
  – Data forwarding and wakeup are decoupled

 Complexity benefits
  ■ smaller RS / faster wakeup path
Mapping to pipeline stages

- AMD K7 (data-capture)

- Pentium 4 (non-data-capture)
Scheduling atomicity & non-data-capture scheduler

- Multi-cycle scheduling loop

- Scheduling atomicity is not maintained
  - Separated by extra pipeline stages (Disp, RF)
  - Unable to issue dependent instructions consecutively

⇒ solution: speculative scheduling
Speculative Scheduling

• Speculatively wakeup dependent instructions even before the parent instruction starts execution
  – Keep the scheduling loop within a single clock cycle

• But, nobody knows what will happen in the future

• Source of uncertainty in instruction scheduling: loads
  – Cache hit / miss
  – Store-to-load aliasing
    → eventually affects timing decisions

• Scheduler assumes that all types of instructions have pre-determined fixed latencies
  – Load instructions are assumed to have a common case (over 90% in general)
    $DL1$ hit latency
  – If incorrect, subsequent (dependent) instructions are replayed
Speculative Scheduling

- **Overview**

  Speculatively issued instructions
  
<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Schedule</th>
<th>Dispatch</th>
<th>RF</th>
<th>Exe</th>
<th>Writeback / Recover</th>
<th>Commit</th>
</tr>
</thead>
</table>

- Unlike the original Tomasulo’s algorithm
  - Instructions are scheduled BEFORE actual execution occurs
  - Assumes instructions have pre-determined fixed latencies
    - ALU operations: fixed latency
    - Load operations: assumes $DL1$ latency (common case)
Selection Logic

• Single FU of each type: arbiter
  – Choose 1:n ready instructions
  – Fairly straightforward logic

• Multiple FUs of a given type: allocator
  – Choose m:n ready instructions
  – Must solve classic “matching” problem
    • Using e.g. iSLIP [Mckeown, IEEE Trans. On Networking, 1999]
  – Not feasible in cycle-time critical loop
  – Instead: instructions bound to FU when inserted
Selection Logic

- Arbiter must guarantee forward progress
  - Select oldest instruction first
  - Algorithm: mask remembers who was waiting when you arrived, defer to them

<table>
<thead>
<tr>
<th>Entry</th>
<th>Valid</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0000</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1100</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0000</td>
</tr>
</tbody>
</table>

1) Entries 2 & 3 wake up: 0011
   2: \((0011 \& 1101) \Rightarrow true\)
   3: \((0011 \& 0100) \Rightarrow false\)

2) 3 is selected, flash clears col 3

3) 2 is selected in next cycle, etc.
Reorder Buffer Implementation

<table>
<thead>
<tr>
<th>Busy</th>
<th>Issued</th>
<th>Finished</th>
<th>Instruction address</th>
<th>Rename register</th>
<th>Speculative</th>
<th>Valid</th>
</tr>
</thead>
</table>

(a)

Next entry to be allocated (tail pointer)

Next instruction to complete (head pointer)

Reorder buffer

(b)

- “Bookkeeping,” physically distributed, many read/write ports
- Can be instruction-grained, or block-grained (4-5 ops)
Register File Speculation & Recovery

• History file (similar to checkpointing)
  – Copy previous value from ARF to HF at dispatch
  – Use HF to reconstruct precise state if needed

• Future file: separate ARF & RRF (lecture notes, PPC 620, Pentium Pro, Core i7, AMD K8, ARM A15)
  – Copy committed value from RRF to ARF
  – Update rename table mapping

• Physical Register File: merged ARF & RRF (MIPS R10000, Pentium 4, Alpha 21264, Power 4-7, Nehalem, Sandybridge, Bulldozer, Bobcat)
  – No copy; simpler datapath (operand always in PRF)
  – Simply “commit” rename table mapping as branches resolve
## Register File Alternatives

<table>
<thead>
<tr>
<th>Register Lifetime</th>
<th>Status</th>
<th>Duration (cycles)</th>
<th>Result stored where?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Future File</td>
</tr>
<tr>
<td>Dispatch Unavail</td>
<td>≥ 1</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>Finish execution</td>
<td>Speculative</td>
<td>≥ 0</td>
<td>FF</td>
</tr>
<tr>
<td>Commit Committed</td>
<td>≥ 0</td>
<td></td>
<td>ARF</td>
</tr>
<tr>
<td>Next def. Dispatched</td>
<td>Committed</td>
<td>≥ 1</td>
<td>ARF</td>
</tr>
<tr>
<td>Next def. Committed</td>
<td>Discarded</td>
<td>≥ 0</td>
<td>Overwritten</td>
</tr>
</tbody>
</table>

- Rename register organization
  - Future file (future updates buffered, later committed)
    - Rename register file
  - History file (old versions buffered, later discarded)
  - Merged (single physical register file)
We showed that PRF is better [ISLPED 07] – nearly everyone now agrees!

- P6 thru Core 2 Duo (Merom): ARF
- Pentium4/Nehalem/Sandybridge, AMD Bulldozer & Bobcat: PRF
- Recent regression: Intel Silvermont, Cortex A15 use ARF ... design team?
Misprediction Recovery

- Branch mispredicts, exceptions: must reclaim allocated resources
  - Load queue, store queue/color, branch color, ROB entry, rename register
- Can reclaim implicitly
  - Tag broadcast: all entities match & release
  - Too expensive for physical register file (PRF)
- Or reclaim explicitly
  - Walk through ROB which contains pointers
  - Follow pointers to release resources
- Also, recover rename mappings
  - Read previous mappings (pending release) and repair map table

<table>
<thead>
<tr>
<th>Valid</th>
<th>PC</th>
<th>Dest PR</th>
<th>Prev PR</th>
<th>Src1</th>
<th>Src2</th>
<th>Imm/target</th>
<th>Issued</th>
<th>Executed</th>
<th>Exception</th>
<th>T/NT Pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x400C</td>
<td>P13</td>
<td>P17</td>
<td>P25</td>
<td>n/a</td>
<td>X80</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>X4008</td>
<td>P14</td>
<td>P22</td>
<td>P31</td>
<td>P5</td>
<td></td>
<td></td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>1</td>
<td>X4004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x4020</td>
<td></td>
<td></td>
<td></td>
<td>NT</td>
</tr>
</tbody>
</table>
Rename Table Implementation

• MAP checkpointing
  – Performance optimization
    • Recovery from branches, exceptions
  – Checkpoint granularity
    • Every instruction (costly)
    • Every branch, play back ROB to get to exception boundary

• RAM vs CAM Map Table
RAM Map Table

- Just a lookup table

Checkpoint size: $n \times \log_2(\text{phys reg})$
Cam Map Table

- CAM search for mappings
  - # rows == number of physical registers
  - Checkpoint only the valid bit column
- Used in Alpha 21264
Summary

• Register dependences
  – True dependences
  – Antidependences
  – Output dependences
• Register Renaming
• Tomasulo’s Algorithm
• Reservation Station Implementation
• Reorder Buffer Implementation
• Register File Implementation
  – History file
  – Future file
  – Physical register file
• Rename Table Implementation