Register Data Flow

Prof. Mikko H. Lipasti
University of Wisconsin-Madison

Lecture notes based on notes by John P. Shen
Updated by Mikko Lipasti

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
Data Dependences
WAR: True Dependences
(WAR) Anti-Dependences
Output Dependences (WAW)
Storage Conflicts

INSTRUCTION EXECUTION MODEL

Each ALU Instruction:

- Register Transfer
- Read
- Write
- Execute

INSTRUCTION LOOPS

Single Assignment, Symbolic Reg.
Map Symbolic Reg. to Physical Reg.
Maximize Reuse of Reg.

Contribution to Register Recycling

COMPILER REGISTER ALLOCATION

Single Assignment, Symbolic Reg.
Map Symbolic Reg. to Physical Reg.
Maximize Reuse of Reg.

For (i=1; i<= 10; i++)
   t := a[i] * b[i];

Spill code (if not enough registers)

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

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
Resolving Anti-Dependences

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

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

Renumbering

Register Renaming

Resolving True Data Dependences

STALL DISPATCHING
ADVANCE INSTRUCTIONS

ECE/CS 725: Advanced Computer Architecture I
Tomasulo's Algorithm [Tomasulo, 1967]

Dependence Mechanisms
Two Address IBM 360 Instruction Format:
R1 <-- R1 op R2

Major dependence mechanisms:
- Structural (FU) dependence ⇒ virtual FU's
  - Reservation stations
- True dependence ⇒ pseudo operands ⇒ result forwarding
  - Register tags
  - Reservation stations
  - Common data bus (CDB)
- Anti-dependence ⇒ operand copying
  - Reservation stations
- Output dependence ⇒ register renaming ⇒ result forwarding
  - Register tags
  - Reservation stations
  - Common data bus (CDB)

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 Source</th>
<th>Valid Data</th>
<th>Source Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>Source</td>
<td>Source Value</td>
</tr>
</tbody>
</table>

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)
  - (6) floating-point registers (FLR)
  - (6) floating-point buffers (FLB)
  - (6) 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

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 FLR + 3 addRS + 2 multiRS = 11 unique sources
  - 3 physical sources (FLR, 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
- Electrically 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.
  - (i) FLR's
  - (s) reservation stations ([1 with add unit, 2 with multiply/divide unit]
  - x + y bit tag is needed to identify the 11 potential sources

- Every destination of a register value must carry a tag field.
  - (i) "sink" entries of the reservation stations
  - (s) "source" entries of the reservation stations
  - (d) SDB's
  - x + y > 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
   - FLR's 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 (virtual entries carrying this tag value in their tag fields) will detect a match of their 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: R2 <- R0 + R4 (RAW on R2)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Example 2: R4 <- R0 + R4 (RAW on R4)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

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 3: R0 <- R4 * R2 (WAR on R2)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ID</th>
<th>Tag</th>
<th>Sink</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Operation of Dependence Mechanisms

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 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.
Example 3

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>0</td>
<td></td>
<td></td>
<td>6.0</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>6.0</td>
<td></td>
<td></td>
<td>0</td>
<td>7.8</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>0</td>
<td>6.0</td>
</tr>
</tbody>
</table>

Example 3

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>0</td>
<td></td>
<td></td>
<td>6.0</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>0</td>
<td>7.8</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>0</td>
<td>10.0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td>0</td>
<td>7.8</td>
</tr>
</tbody>
</table>

Example 3

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>0</td>
<td></td>
<td></td>
<td>6.0</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>0</td>
<td>7.8</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td>0</td>
<td>10.0</td>
</tr>
</tbody>
</table>


can Tomasulo’s algorithm reach dataflow limit of 8?

Summary of Tomasulo’s Algorithm

- Supports out of order execution of instructions.
- Resolves dependences dynamically using hardware.
- Attempts to delay the resolution of dependences as late as possible.
- Structural dependence does not stall issuing; virtual FUs 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 FUs 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>Width</td>
<td>Peak IPC = 1</td>
<td>4+</td>
</tr>
<tr>
<td>Structural hazards</td>
<td>2 FPU</td>
<td>Many FU</td>
</tr>
<tr>
<td>Anti-dependences</td>
<td>Single CD8</td>
<td>Many busses</td>
</tr>
<tr>
<td>Output dependences</td>
<td>Renamed reg. tag</td>
<td>Reg. renaming</td>
</tr>
<tr>
<td>True dependences</td>
<td>Tag-based forw.</td>
<td>Tag-based forw.</td>
</tr>
<tr>
<td>Exceptions</td>
<td>Imprecise</td>
<td>Precise (ROB)</td>
</tr>
<tr>
<td>Implementation</td>
<td>3 x 66” x 15&quot; x 78”</td>
<td>1 chip</td>
</tr>
<tr>
<td></td>
<td>60ns cycle time</td>
<td>300ps</td>
</tr>
<tr>
<td></td>
<td>11-12 gate delays</td>
<td>&lt; $100</td>
</tr>
<tr>
<td></td>
<td>per pipe stage</td>
<td>&lt;$1 million</td>
</tr>
</tbody>
</table>

Example 4

i: R4 ← R0 + R8
j: R2 ← R0 * R4
k: R4 ← R4 + R8
l: R8 ← R4 * R2
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
disallocate 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

Reorder Buffer Implementation

- Merge RS and ROB => Register Update Unit (RUU)
  - Inefficient, hard to scale
  - Perhaps of interest only to historians

Reorder Buffer Implementation

- “Bookkeeping”
  - Can be instruction-grained, or block-grained (4–5 ops)
Data Capture Reservation Station

- Reservation Stations
  - Data capture vs. no data capture
  - Latter leads to "speculative scheduling"

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>Dispatch</td>
<td>Unavail</td>
<td>≥ 1</td>
<td>N/A</td>
</tr>
<tr>
<td>Finish execution</td>
<td>Speculative</td>
<td>≥ 0</td>
<td>FF, ARF, PRF</td>
</tr>
<tr>
<td>Commit</td>
<td>Committed</td>
<td>≥ 0</td>
<td>ARF, ARF, PRF</td>
</tr>
<tr>
<td>Next def. Dispatched</td>
<td>Committed</td>
<td>≥ 1</td>
<td>ARF, HP, PRF</td>
</tr>
<tr>
<td>Next def. Committed</td>
<td>Discarded</td>
<td>≥ 0</td>
<td>Overwritten, Discarded, Reclaimed</td>
</tr>
</tbody>
</table>

- Rename register organization
  - Future file (future updates buffered, later committed)
  - History file (old versions buffered, later discarded)
  - Merged (single physical register file)

Register File Commit

- Register Commit
  - History file (similar to checkpointing – covered later)
    - 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 604/620, Pentium Pro, Core 2 Duo, AMD K8)
    - 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

ARF vs. PRF

- We showed that PRF is better [ISLPED 07] – everyone now agrees!
- P6 thru Core 2 Duo (Merom): ARF
- Pentium4/Nehalem/Sandybridge, AMD Bulldozer & Bobcat: PRF

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 entries 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

Rename Table Implementation

- MAP checkpointing
  - Performance optimization
    - Recovery from branches, exceptions
  - Checkpoint granularity
    - Every instruction
    - 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 (# arch reg) x log₂(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