Readings

- General introduction and basic concepts
  - Hennessy and Patterson, Sections 2.1-2.10 (inclusive).

- Modern designs

- Seminal papers
Reviews Due

- Due September 30
Last Time …

- Branch mispredictions
- Out-of-order execution
  - Register renaming
  - Tomasulo’s algorithm
Today and the Next Related Lectures

- Exploiting Instruction Level Parallelism (ILP)
- More in-depth out-of-order execution
- Superscalar processing
- Better instruction supply and control flow handling
Tomasulo’s Machine: IBM 360/91
Tomasulo’s Algorithm

- If reservation station available before renaming
  - Instruction + renamed operands (source value/tag) inserted into the reservation station
  - Only rename if reservation station is available
- Else stall
- While in reservation station, each instruction:
  - Watches common data bus (CDB) for tag of its sources
  - When tag seen, grab value for the source and keep it in the reservation station
  - When both operands available, instruction ready to be dispatched
- Dispatch instruction to the Functional Unit when instruction is ready
- After instruction finishes in the Functional Unit
  - Arbitrate for CDB
  - Put tagged value onto CDB (tag broadcast)
  - Register file is connected to the CDB
    - Register contains a tag indicating the latest writer to the register
    - If the tag in the register file matches the broadcast tag, write broadcast value into register (and set valid bit)
  - Reclaim rename tag
    - no valid copy of tag in system!
Register Renaming and OoO Execution

- Architectural registers dynamically renamed
  - Mapped to reservation stations

```
<table>
<thead>
<tr>
<th>tag</th>
<th>value</th>
<th>valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>-</td>
<td>V0</td>
</tr>
<tr>
<td>R1</td>
<td>S2</td>
<td>newV1V1</td>
</tr>
<tr>
<td>R2</td>
<td>-</td>
<td>V2</td>
</tr>
<tr>
<td>R3</td>
<td>S0</td>
<td>newV3V3</td>
</tr>
<tr>
<td>R4</td>
<td>-</td>
<td>V4</td>
</tr>
<tr>
<td>R5</td>
<td>-</td>
<td>V5</td>
</tr>
<tr>
<td>R6</td>
<td>-</td>
<td>V6</td>
</tr>
<tr>
<td>R7</td>
<td>S4</td>
<td>V7</td>
</tr>
<tr>
<td>R8</td>
<td>-</td>
<td>V8</td>
</tr>
<tr>
<td>R9</td>
<td>-</td>
<td>V9</td>
</tr>
</tbody>
</table>
```

```
IMUL R3 ← R1, R2  IMUL S0 ← V1, V2
ADD R3 ← R3, R1  ADD S1 ← S0, V4
ADD R1 ← R6, R7  ADD S2 ← V6, V7
IMUL R3 ← R6, R8  IMUL S3 ← V6, V8
ADD R7 ← R3, R9  ADD S4 ← S3, V9
```

```
<table>
<thead>
<tr>
<th>Src1 tag</th>
<th>Src1 value</th>
<th>V?</th>
<th>Src2 tag</th>
<th>Src2 value</th>
<th>V?</th>
<th>Ctl</th>
<th>S?</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nul</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>S0</td>
<td></td>
<td>Retired</td>
<td>Entry Deallocated</td>
<td>ent</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>S2</td>
<td>-</td>
<td></td>
<td>Completed</td>
<td>Wait for Retirement</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>-</td>
<td></td>
<td>BROADCAST</td>
<td>S3 and new3V3</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>S4</td>
<td>S3</td>
<td>new3V3</td>
<td>0</td>
<td>-</td>
<td>V9</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```
In-order vs. Out-of-order Dispatch

- **In order dispatch:**

  - Tomasulo:

  - 16 vs. 12 cycles
An Exercise

MUL R3 ← R1, R2
ADD R5 ← R3, R4
ADD R7 ← R2, R6
ADD R10 ← R8, R9
MUL R11 ← R7, R10
ADD R5 ← R5, R11

- Assume ADD (4 cycle execute), MUL (6 cycle execute)
- Assume one adder and one multiplier
- How many cycles
  - in a non-pipelined machine
  - in an in-order-dispatch pipelined machine with future file and reorder buffer
  - in an out-of-order dispatch pipelined machine with future file and reorder buffer
Summary of OOO Execution Concepts

- Renaming eliminates false dependencies

- Tag broadcast enables value communication between instructions → dataflow

- An out-of-order engine dynamically builds the dataflow graph of a piece of the program
  - which piece?
    - Limited to the instruction window
  - Can we do it for the whole program? Why would we like to?
    - How can we have a large instruction window efficiently?
Some Implementation Issues

- Back to back dispatch of dependent operations
  - When should the tag be broadcast?

- Is it a good idea to have the reservation stations also be the reorder buffer?

- Is it a good idea to have values in the register alias table?

- Is it a good idea to store values in reservation stations?

- Is it a good idea to have a single, centralized reservation stations for all FUs?

- Idea: Decoupling different buffers
  - Many modern processors decouple these buffers
  - Many have distributed reservation stations across FUs
Buffer Decoupling in Pentium Pro

Figure courtesy of Prof. Wen-mei Hwu, University of Illinois
Buffer Decoupling in Pentium 4
Pentium Pro vs. Pentium 4

Basic Pentium III Processor Misprediction Pipeline

1. Fetch
2. Fetch
3. Decode
4. Decode
5. Decode
6. Rename
7. ROB Rd
8. Rdy/Sch
9. Dispatch
10. Exec

Basic Pentium 4 Processor Misprediction Pipeline

1. TC Nxt IP
2. TC Fetch
3. Drive
4. Alloc
5. Rename
6. Que
7. Sch
8. Sch
9. Disp
10. Disp
11. RF
12. RF
13. RF
14. Ex
15. Flgs
16. Br Ck
17. Br Ck
18. Drive
19. Drive
20. Drive

Pentium III

ROB
Data Status

RAT
EAX
EBX
ECX
EDX
ESI
EDI
ESP
EBP

RRF

NetBurst

RF
Data

ROB
Status

Frontend RAT
EAX
EBX
ECX
EDX
ESI
EDI
ESP
EBP

Retirement RAT
EAX
EBX
ECX
EDX
ESI
EDI
ESP
EBP

Pointers to architectural register values
Required/Recommended Reading


Buffer-Decoupled OoO Designs

- Register Alias Table contains only tags and valid bits
  - Tag is a pointer to the physical register containing the value

- During renaming, instruction allocates entries in different buffers (as needed):
  - Reorder buffer
  - Physical register file
  - Reservation stations (centralized vs. distributed)
  - Store/load buffer

- Tradeoffs?
Physical Register Files

- Register values stored only in physical register files (architectural register state is part of this file), e.g. Pentium 4

  + Smaller physical register file than ROB: Not all instructions write to registers \(\rightarrow\) more area efficient
  
  + No duplication of data values in reservation stations, arch register file, reorder buffer, rename table (space efficient)
  
  + Only one write and read path for register data values (simpler control for register read and writes)

  -- Accessing register state always requires indirection (higher latency)
  
  - Instructions with ready operands need to access register file
  
  - Copying architectural register state takes time
Physical Register Files in Pentium 4

Physical Register Files in Alpha 21264

Alpha 21264 Pipelines

Source: Microprocessor Report, 10/28/96
Centralized Reservation Stations (Pentium Pro)
Distributed Reservation Stations

- Many processors: PowerPC 604, Pentium 4
Centralized vs. Distributed Reservation Stations

- **Centralized (monolithic):**
  - + Reservation stations not statically partitioned (can adapt to changes in instruction mix, e.g., 100% adds)
  - -- All entries need to have all fields even though some fields might not be needed for some instructions (e.g. branches, stores, etc)
  - -- Number of ports = issue width
  - -- More complex control and routing logic

- **Distributed:**
  - + Each RS can be specialized to the functional unit it serves (more area efficient)
  - + Number of ports can be 1 (RS per functional unit)
  - + Simpler control and routing logic
  - -- Other RS’s cannot be used if one functional unit’s RS is full (static partitioning)
Issues in Scheduling Logic (I)

- What if multiple instructions become ready in the same cycle?
  - Why does this happen?
    - An instruction with multiple dependents
    - Multiple instructions can complete in the same cycle

- Which one to schedule?
  - Oldest
  - Random
  - Most dependents first
  - Most critical first (longest latency dependency chain)

- Does not matter for performance unless the active window is very large
Issues in Scheduling Logic (II)

- When to schedule the dependents of a multi-cycle execute instruction?
  - One cycle before the completion of the instruction
  - Example: IMUL, Pentium 4, 3-cycle ADD

- When to schedule the dependents of a variable-latency execute instruction?
  - A load can hit or miss in the data cache
  - Option 1: Schedule dependents assuming load will hit
  - Option 2: Schedule dependents assuming load will miss

- When do we take out an instruction from a reservation station?
Scheduling of Load Dependents

- Assume load will hit
  - + No delay for dependents (load hit is the common case)
  - -- Need to squash and re-schedule if load actually misses

- Assume load will miss (i.e. schedule when load data ready)
  - + No need to re-schedule (simpler logic)
  - -- Significant delay for load dependents if load hits

- Predict load hit/miss
  - + No delay for dependents on accurate prediction
  - -- Need to predict and re-schedule on misprediction

What to Do with Dependents on a Load Miss? (I)

- A load miss can take hundreds of cycles
- If there are many dependent instructions on a load miss, these can clog the scheduling window
- Independent instructions cannot be allocated reservation stations and scheduling can stall
- How to avoid this?

- Idea: Move miss-dependent instructions into a separate buffer
  - Example: Pentium 4’s “scheduling loops”
But, dependents still hold on to the physical registers

Cannot scale the size of the register file indefinitely since it is on the critical path

Possible solution: Deallocate physical registers of dependents

Questions

- Why is OoO execution beneficial?
  - What if all operations take single cycle?
  - Latency tolerance: OoO execution tolerates the latency of multi-cycle operations by executing independent operations concurrently

- What if the first IMUL was a cache-miss LD?
  - If it took 500 cycles, how large of a scheduling window do we need to continue decoding?
  - How many cycles of latency can OoO tolerate?
  - What limits the latency tolerance scalability of Tomasulo’s algorithm?
    - Active/instruction window size: determined by register file, scheduling window, reorder buffer, store buffer, load buffer