Chapter 5: Superscalar: Memory Data Flow
Modern Processor Design: Fundamentals of Superscalar Processors
Data and control dependencies

- The **sequential execution model** assumes that
  - 1. Instructions are executed atomically
  - 2. in the order specified by the program.
- Dependencies among instructions may hinder parallel execution of multiple instructions.
- **Parallel instruction execution**
  - 1. ILP
  - 2. Parallelizing compilers
- **Hazards** → parallel execution of several instructions without considering data and control dependencies can lead to wrong results
Data dependency

• Data dependency ➜ when an instructions refers to data in a previous one, the two instructions cannot be executed in parallel.

• RAW – Read After Write
  I1  a=9
  I2  b = a
  I3  c = b  I3 is dependent on I2

• WAR - Write After Read (anti-dependence). An instruction requires a value that is later updated.
  I1  a = 9
  I2  b = a + 5  I2 is anti-dependent on I3
  I3  a = 3

This dependency can be removed by introducing an additional variable and all instructions can now be executed in parallel
  I1  a = 9
  I2  a1 = a
  I3  b = a1 + 5
  I4  a = 3
Data dependency

- **WAW - write-after-write.** The ordering of instructions affects the final output value of a variable.
  
  I1  \[ a = 3 \]
  I2  \[ b = a + 9 \]
  I3  \[ a = 7 \]  
  If I3 is executed before I1 the results are affected

  WAW or output dependencies are *name dependencies*. Renaming variables, solves the problem:
  
  1  \[ a_1 = 3 \]
  2  \[ b = a_1 + 9 \]
  3  \[ a = 7 \]
Control dependency

• An instruction I2 is control dependent of another instruction I1 if I1 determines if I2 should be executed or not.

  I1    if (a == b)
  I2    a = a + b  \( I2 \text{ is control dependent of } I1 \)
  I3    b = a + b

Two conditions:
• there exists a path Q from P1 to P2 such that every instruction Pi ≠ P1 within Q will be followed by P2 in each possible path to the end of the program and
• P1 will not necessarily be followed by P2, i.e. there is an execution path from P1 to the end of the program that does not go through P2.
Memory Data Flow

- Memory Data Flow
  - Memory Data Dependences
  - Load Bypassing
  - Load Forwarding
  - Speculative Disambiguation
  - The Memory Bottleneck
## Memory Data Dependences

- Besides branches, long memory latencies are one of the biggest performance challenges today.

<table>
<thead>
<tr>
<th>WAW</th>
<th>WAR</th>
<th>RAW</th>
</tr>
</thead>
<tbody>
<tr>
<td>store X</td>
<td>load X</td>
<td>store X</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>store X</td>
<td>store X</td>
<td>load X</td>
</tr>
</tbody>
</table>
Memory dependency and OOO execution

- To preserve sequential (in-order) state in the data caches and external memory (so that recovery from exceptions is possible) **stores are performed in order**. This takes care of output and anti-dependences to memory locations.

- However, **loads can be issued out of order** with respect to stores if the out-of-order loads check for data dependences with respect to previous, pending stores.
Memory Data Dependences

- "Memory Aliasing" = Two memory references involving the same memory location (collision of two memory addresses).
- "Memory Disambiguation" = Determining whether two memory references will alias or not (whether there is a dependence or not).
- Memory Dependency Detection:
  - Must compute effective addresses of both memory references
  - Effective addresses can depend on run-time data and other instructions
  - Comparison of addresses require much wider comparators

Example code:

(1) STORE \( \text{mem}[r1 + 12] = r2 \) \( \text{addr} = r1 + 12 = V \)
(2) ADD
(3) LOAD \( r3 = \text{Mem}[r10+4] \) \( \text{addr} = r10 + 4 = W \)
(4) LOAD \( r3 = \text{Mem}[X] \) \( \text{addr} = X \)
(5) LOAD \( r5 = \text{Mem}[r13] \) \( \text{addr} = r13 = V \)
(6) ADD
(7) STORE \( \text{Mem}[W] = r0 \) \( \text{addr} = W \)

Decoding is not enough to determine memory dep. => requires addr computation
Handling Memory Dependencies

- Total Order of Loads and Stores
- Load Bypassing
- Load Forwarding (Bypassing != Forwarding in this context)
- Optimizing Load/Store Disambiguation
Total Order of Loads and Stores

• Keep all loads and stores totally in order with respect to each other.
• However, loads and stores can execute out of order with respect to other types of instructions.
• Consequently, stores are held for all previous instructions (store buffer), and loads are held for stores.
  – I.e. stores performed at commit point
  – Sufficient to prevent wrong branch path stores since all prior branches now resolved
Load Bypassing

- Loads can be allowed to bypass stores (if no aliasing).
- Two separate reservation stations and address generation units are employed for loads and stores.
- Store addresses still need to be computed before loads can be issued to allow checking for load dependences. If dependence cannot be checked, e.g. store address cannot be determined, then all subsequent loads are held until address is valid (conservative).
- Stores are kept in ROB until all previous instructions complete; and kept in the store buffer until gaining access to cache port.
  - Store buffer is “future file” for memory
Load Bypassing
Load Forwarding

- If a subsequent load has a dependence on a store still in the store buffer, it need not wait till the store is issued to the data cache.
- The load can be directly satisfied from the store buffer if the address is valid and the data is available in the store buffer.
- This avoids the latency of accessing the data cache.
Load Forwarding

Reservation station

Store unit

Load unit

Tag match

Address

Data

Data cache

Match/no match

If match: forward to destination register

(Finished) Store buffer

(Completed) Store buffer

match
Optimizing Load/Store Disambiguation

- Non-speculative load/store disambiguation
  1. Loads wait for addresses of all prior stores
  2. Full address comparison
  3. Bypass if no match, forward if match
- (1) can limit performance:

```
load r5, MEM[r3] ← cache miss
store r7, MEM[r5] ← RAW for agen, stalled
...
load r8, MEM[r9] ← independent load stalled
```
Speculative Disambiguation

• What if aliases are rare?
  1. Loads don’t wait for addresses of all prior stores
  2. Full address comparison of stores that are ready
  3. Bypass if no match, forward if match
  4. Check all store addresses when they commit
     - No matching loads – speculation was correct
     - Matching bypassed load – incorrect speculation
  5. Replay starting from incorrect load

• This is called a load-replay trap
  - Same mechanism used to keep machine sequentially consistent in multiprocessor configurations
  - P0
  - A=1
  - Read B
  - P1
  - B=1
  - Read A
Speculative Memory Disambiguation

- Reservation station
- Store unit
  - (Finished) Store buffer
  - (Completed) Store buffer
  - data
  - addr
  - Tag match
  - At store completion
- Load unit
  - addr
  - data
  - Tag match
  - Address
  - Data
  - At finish: update renamed register
  - At completion: update architected register
- Match/no match
- Match/no match
  - If match: flush aliased load and all trailing instructions
- Data cache
Use of Prediction

• If aliases are rare: static prediction
  – Predict no alias every time
    • Why even implement forwarding? PowerPC 620 doesn’t
  – Pay misprediction penalty rarely
• If aliases are more frequent: dynamic prediction
  – Use PHT-like history table for loads
    • If alias predicted: delay load
    • If aliased pair predicted: forward from store to load
      – More difficult to predict pair [store sets, Alpha 21264]
  – Pay misprediction penalty rarely
• Memory cloaking [Moshovos, Sohi]
  – Predict store/load pair
  – Directly copy store data register to load target register
  – Reduce data transfer latency to absolute minimum
    • Integer code improves ~4%, floating point ~3%
Load/Store Disambiguation Discussion

- **RISC ISA:**
  - Many registers, most variables allocated to registers
  - Aliases are rare
  - Most important to not delay loads (bypass)
  - Alias predictor may/may not be necessary

- **CISC ISA:**
  - Few registers, many operands from memory
  - Aliases much more common, forwarding necessary
  - Incorrect load speculation should be avoided
  - If load speculation allowed, predictor probably necessary

- **Address translation:**
  - Can’t use virtual address (must use physical)
  - Wait till after TLB lookup is done
  - Or, use subset of untranslated bits (page offset)
    - Safe for proving inequality (bypassing OK)
    - Not sufficient for showing equality (forwarding not OK)