Superscalar Techniques
Part 1: Instruction Flow

Instruction Flow Techniques
• Goal and Impediments
• Branch Types and Implementations
• What's So Bad About Branches?
• What are Control Dependences?
• Impact of Control Dependences on Performance
• Improving I-Cache Performance

Instruction Flow in Context

Goal and Impediments
• Goal of Instruction Flow
  – Supply processor with maximum number of useful instructions every clock cycle
• Impediments
  – Branches and jumps
  – I-Cache limitations
    • hit rate
    • width
    • alignment
    • etc.
Branch Types and Implementation

1. Types of Branches
   A. Conditional or Unconditional
   B. Save PC? Save other processor state?
   C. How is target computed?
      - constant target (immediate, PC-relative)
      - variable target (register, register + offset)

2. Branch Architectures
   A. Condition code or condition registers
   B. Register

What’s So Bad About Branches?

- Effects of Branches
  - Fragmentation of I-Cache lines
  - Need to determine branch outcome
  - Need to determine branch target
  - Use up execution resources

What’s So Bad About Branches?

Problem: Fetch stalls until outcome is determined

Solutions:
- Minimize delay
  - Move instructions determining branch condition away from branch
- Make use of delay
  - Non-speculative:
    - Fill delay slots with useful safe instructions
    - Execute both paths (eager execution)
  - Speculative:
    - Predict branch outcome

Problem: Fetch stalls until branch target is determined

Solutions:
- Minimize delay
- Generate branch target early
- Make use of delay: Predict branch target
  - Single target (constant—only need to compute once)
  - Multiple targets (variable—but may use previous target value as prediction for next time).
Control Dependences

- **Control Flow Graph**
  - Shows possible paths of control flow through basic blocks

```
main:
  addi r2, r0, A
  addi r3, r0, B
  addi r4, r0, C
  addi r5, r0, N
  addi r10, r0, r0
  bge r10, r5, end
loop:
  lw r20, 0(r2)
  lw r21, 0(r3)
  bge r20, r21, T1
  sw r21, 0(r4)
  b T2
T1:
  sw r20, 0(r4)
T2:
  addi r10, r10, 1
  addi r2, r2, 4
  addi r3, r3, 4
  addi r4, r4, 4
  blt r10, r5, loop
```

- **Control Dependence**
  - Node B is CD on Node A if A determines whether B executes

```
main:
  addi r2, r0, A
  addi r3, r0, B
  addi r4, r0, C
  addi r5, r0, N
  addi r10, r0, r0
  bge r10, r5, end
loop:
  lw r20, 0(r2)
  lw r21, 0(r3)
  bge r20, r21, T1
  sw r21, 0(r4)
  b T2
T1:
  sw r20, 0(r4)
T2:
  addi r10, r10, 1
  addi r2, r2, 4
  addi r3, r3, 4
  addi r4, r4, 4
  blt r10, r5, loop
```

**Limits on Instruction Level Parallelism (ILP)**

<table>
<thead>
<tr>
<th>Reference</th>
<th>ILP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weiss and Smith [1984]</td>
<td>1.58</td>
</tr>
<tr>
<td>Sohi and Vajapeyam [1987]</td>
<td>1.81</td>
</tr>
<tr>
<td>Tjaden and Flynn [1970]</td>
<td>1.86 (Flynn's bottleneck)</td>
</tr>
<tr>
<td>Tjaden and Flynn [1973]</td>
<td>1.96</td>
</tr>
<tr>
<td>UN [1988]</td>
<td>2.00</td>
</tr>
<tr>
<td>Smith et al. [1989]</td>
<td>2.00</td>
</tr>
<tr>
<td>Jouppi and Wall [1988]</td>
<td>2.40</td>
</tr>
<tr>
<td>Johnson [1991]</td>
<td>2.50</td>
</tr>
<tr>
<td>Acosta et al. [1996]</td>
<td>2.50</td>
</tr>
<tr>
<td>Melina [1992]</td>
<td>2.79</td>
</tr>
<tr>
<td>Butler et al. [1991]</td>
<td>3.00</td>
</tr>
<tr>
<td>Mock and Pat [1991]</td>
<td>5.8</td>
</tr>
<tr>
<td>Wall [1991]</td>
<td>7 (Jouppi disagreed)</td>
</tr>
<tr>
<td>Rose et al. [1991]</td>
<td>8</td>
</tr>
<tr>
<td>Riseman and Foster [1972]</td>
<td>51 (no control dependences)</td>
</tr>
<tr>
<td>Nicolas and Fisher [1984]</td>
<td>80 (Fisher's optimism)</td>
</tr>
</tbody>
</table>

**Riseman and Foster’s Study**

- 7 benchmark programs on CDC-3600
- Assume infinite machines
  - Infinite memory and instruction stack
  - Infinite register file
  - Infinite functional units
- True dependencies only at dataflow limit
- If lookahead bounded to single basic block, speedup is 1.72 (Flynn’s bottleneck)
- If one can bypass branches (hypothetically), then: a theoretical ILP of 51 was determined by Riseman and Foster

**Riseman and Foster’s Study**

- 7 benchmark programs on CDC-3600
- Assume infinite machines
  - Infinite memory and instruction stack
  - Infinite register file
  - Infinite functional units
- True dependencies only at dataflow limit
- If lookahead bounded to single basic block, speedup is 1.72 (Flynn’s bottleneck)
- If one can bypass branches (hypothetically), then: a theoretical ILP of 51 was determined by Riseman and Foster
Program Control Flow

- Implicit Sequential Control Flow
  - Static Program Representation
    - Control Flow Graph (CFG)
    - Nodes = basic blocks
    - Edges = Control flow transfers
  - Physical Program Layout
    - Mapping of CFG to linear program memory
    - Implied sequential control flow
- Dynamic Program Execution
  - Traversal of the CFG nodes and edges (e.g. loops)
  - Traversal dictated by branch conditions
- Dynamic Control Flow
  - Deviates from sequential control flow
  - Disrupts sequential fetching
  - Can stall IF stage and reduce I-fetch bandwidth

Disruption of Sequential Control Flow

- Target address generation → Target Speculation
  - Access register:
    - PC, General purpose register, Link register
  - Perform calculation:
    - +/- offset, autoincrement, autodecrement
- Condition resolution → Condition speculation
  - Access register:
    - Condition code register, General purpose register
  - Perform calculation:
    - Comparison of data register(s)
### Target Address Generation

- **Fetch**
- **Decode**
- **Dispatch**
- **Reservation**
- **Issue**
- **Execute**
- **Complete**
- **Retire**
- **Store Buffer**
- **Completion Buffer**

### Condition Resolution

- **Fetch**
- **Decode**
- **Dispatch**
- **Reservation**
- **Issue**
- **Execute**
- **Complete**
- **Retire**
- **Store Buffer**
- **Completion Buffer**

### Branch Instruction Speculation

- Prediction
- Branch Prediction (using a BTB)
- BTB update (target address and history)
- Execute
- Finish
- Completion Buffer

### Branch/Jump Target Prediction

- Branch Target Buffer: small cache in fetch stage
  - Previously executed branches, address, taken history, target(s)
- Fetch stage compares current FA against BTB
  - If match, use prediction
  - If predict taken, use BTB target
- When branch executes, BTB is updated
- Optimization:
  - Size of BTB: increases hit rate
  - Prediction algorithm: increase accuracy of prediction

**Branch Inst. Information**

- Branch Target Address for predict.
- Branch Target Address (most recent)
Branch Prediction: Condition Speculation

1. Biased Not Taken
   - Hardware prediction
   - Does not affect ISA
   - Not effective for loops

2. Software Prediction
   - Extra bit in each branch instruction
     - Set to 0 for not taken
     - Set to 1 for taken
   - Bit set by compiler or user; can use profiling
   - Static prediction, same behavior every time

3. Prediction based on branch offset
   - Positive offset: predict not taken
   - Negative offset: predict taken

4. Prediction based on dynamic history

UCB Study [Lee and Smith, 1984]

- Benchmarks used
  - 26 programs (IBM 370, DEC PDP-11, CDC 6400)
  - 6 workloads (4 IBM, 1 DEC, 1 CDC)
  - Used trace-driven simulation

- Branch types
  - Unconditional: always taken or always not taken
  - Subroutine call: always taken
  - Loop control: usually taken
  - Decision: either way, if-then-else
  - Computed goto: always taken, with changing target
  - Supervisor call: always taken
  - Execute: always taken (IBM 370)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>IBM1</th>
<th>IBM2</th>
<th>IBM3</th>
<th>IBM4</th>
<th>DEC</th>
<th>CDC</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>0.64</td>
<td>0.65</td>
<td>0.70</td>
<td>0.54</td>
<td>0.73</td>
<td>0.77</td>
<td>0.67</td>
</tr>
<tr>
<td>NT</td>
<td>0.36</td>
<td>0.34</td>
<td>0.29</td>
<td>0.46</td>
<td>0.26</td>
<td>0.22</td>
<td>0.32</td>
</tr>
</tbody>
</table>

Branch Prediction: Condition Speculation

1. Biased Not Taken
   - Hardware prediction
   - Does not affect ISA
   - Not effective for loops

2. Software Prediction
   - Extra bit in each branch instruction
     - Set to 0 for not taken
     - Set to 1 for taken
   - Bit set by compiler or user; can use profiling
   - Static prediction, same behavior every time

3. Prediction based on branch offset
   - Positive offset: predict not taken
   - Negative offset: predict taken

4. Prediction based on dynamic history

Branch Prediction Function

- Prediction function F(X1, X2, ...)
  - X1 – opcode type
  - X2 – history
- Prediction effectiveness based on opcode only, or history

<table>
<thead>
<tr>
<th>Opcode only</th>
<th>IBM1</th>
<th>IBM2</th>
<th>IBM3</th>
<th>IBM4</th>
<th>DEC</th>
<th>CDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>66</td>
<td>69</td>
<td>71</td>
<td>55</td>
<td>80</td>
<td>78</td>
</tr>
<tr>
<td>History 1</td>
<td>92</td>
<td>95</td>
<td>87</td>
<td>80</td>
<td>97</td>
<td>82</td>
</tr>
<tr>
<td>History 2</td>
<td>93</td>
<td>97</td>
<td>91</td>
<td>83</td>
<td>98</td>
<td>91</td>
</tr>
<tr>
<td>History 3</td>
<td>94</td>
<td>97</td>
<td>91</td>
<td>84</td>
<td>98</td>
<td>94</td>
</tr>
<tr>
<td>History 4</td>
<td>95</td>
<td>97</td>
<td>92</td>
<td>84</td>
<td>98</td>
<td>95</td>
</tr>
<tr>
<td>History 5</td>
<td>95</td>
<td>97</td>
<td>92</td>
<td>84</td>
<td>98</td>
<td>96</td>
</tr>
</tbody>
</table>
Example Prediction Algorithm

- Hardware table remembers last 2 branch outcomes
  - History of past several branches encoded by FSM
  - Current state used to generate prediction
- Results:

<table>
<thead>
<tr>
<th></th>
<th>IBM1</th>
<th>IBM2</th>
<th>IBM3</th>
<th>IBM4</th>
<th>DEC</th>
<th>CDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>93</td>
<td>97</td>
<td>91</td>
<td>83</td>
<td>98</td>
<td>91</td>
<td></td>
</tr>
</tbody>
</table>

Other Prediction Algorithms

- Combining prediction accuracy with BTB hit rate (86.5% for 128 sets of 4 entries each), branch prediction can provide the net prediction accuracy of approximately 80%. This implies a 5-20% performance enhancement.

IBM Study [Nair, 1992]

- Branch processing on the IBM RS/6000
  - Separate branch functional unit
  - Five different branch types
    - b: unconditional branch
    - bl: branch and link (subroutine calls)
    - bc: conditional branch
    - bcr: conditional branch using link register (returns)
    - bcc: conditional branch using count register
  - Overlap of branch instructions with other instructions
    - Zero cycle branches
  - Two causes for branch stalls
    - Unresolved conditions
    - Branches downstream too close to unresolved branches

Branch Instruction Distribution

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>b %</th>
<th>bl %</th>
<th>bc %</th>
<th>bcr %</th>
<th>3 cyc</th>
<th>2 cyc</th>
<th>1 cyc</th>
</tr>
</thead>
<tbody>
<tr>
<td>spice2g6</td>
<td>7.86</td>
<td>0.30</td>
<td>12.58</td>
<td>0.32</td>
<td>13.82</td>
<td>3.12</td>
<td>0.76</td>
</tr>
<tr>
<td>doduc</td>
<td>1.00</td>
<td>0.94</td>
<td>8.22</td>
<td>1.01</td>
<td>10.14</td>
<td>1.76</td>
<td>2.02</td>
</tr>
<tr>
<td>matrix300</td>
<td>0.00</td>
<td>0.00</td>
<td>14.50</td>
<td>0.00</td>
<td>0.68</td>
<td>0.22</td>
<td>0.20</td>
</tr>
<tr>
<td>tomcatv</td>
<td>0.00</td>
<td>0.00</td>
<td>6.10</td>
<td>0.00</td>
<td>0.24</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>gcc</td>
<td>2.30</td>
<td>1.32</td>
<td>15.50</td>
<td>1.81</td>
<td>22.46</td>
<td>9.48</td>
<td>4.85</td>
</tr>
<tr>
<td>espresso</td>
<td>3.61</td>
<td>0.58</td>
<td>19.85</td>
<td>0.68</td>
<td>37.37</td>
<td>1.77</td>
<td>0.31</td>
</tr>
<tr>
<td>li</td>
<td>2.41</td>
<td>1.92</td>
<td>14.36</td>
<td>1.91</td>
<td>31.55</td>
<td>3.44</td>
<td>1.37</td>
</tr>
<tr>
<td>eqntott</td>
<td>0.91</td>
<td>0.47</td>
<td>32.87</td>
<td>0.51</td>
<td>5.01</td>
<td>11.01</td>
<td>0.80</td>
</tr>
</tbody>
</table>
Exhaustive Search for Optimal 2-bit Predictor

- There are \(2^{20}\) possible state machines of 2-bit predictors
- Some machines are uninteresting, pruning them out reduces the number of state machines to 5248
- For each benchmark, determine prediction accuracy for all the predictor state machines
- Find optimal 2-bit predictor for each application

### Table: Prediction Accuracy (Overall CPI Overhead)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>3 bit</th>
<th>2 bit</th>
<th>1 bit</th>
<th>0 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>spice2g6</td>
<td>97.0 (0.009)</td>
<td>97.0 (0.009)</td>
<td>96.2 (0.013)</td>
<td>78.6 (0.031)</td>
</tr>
<tr>
<td>doduc</td>
<td>94.2 (0.003)</td>
<td>94.3 (0.003)</td>
<td>90.2 (0.004)</td>
<td>69.2 (0.022)</td>
</tr>
<tr>
<td>gcc</td>
<td>89.7 (0.025)</td>
<td>89.1 (0.026)</td>
<td>86.0 (0.033)</td>
<td>50.0 (0.128)</td>
</tr>
<tr>
<td>espresso</td>
<td>89.5 (0.045)</td>
<td>89.1 (0.047)</td>
<td>87.2 (0.054)</td>
<td>58.5 (0.176)</td>
</tr>
<tr>
<td>li</td>
<td>88.3 (0.042)</td>
<td>86.8 (0.048)</td>
<td>82.5 (0.063)</td>
<td>62.4 (0.142)</td>
</tr>
<tr>
<td>eqntott</td>
<td>89.3 (0.028)</td>
<td>87.2 (0.033)</td>
<td>82.9 (0.046)</td>
<td>78.4 (0.049)</td>
</tr>
</tbody>
</table>

- Branch history table size: Direct-mapped array of 2\(^k\) entries
- Some programs, like gcc, have over 7000 conditional branches
- In collisions, multiple branches share the same predictor
  - Constructive interference
  - Destructive interference
- Marginal gains beyond 1K entries (for these programs)
Yeh & Patt Michigan Study, 1992

- **Two-level adaptive branch prediction:**
  - **First level:** History of last \( k \) branches encountered
  - **Second level:** branch behavior of the last \( s \) occurrences of the specific pattern of these \( k \) branches
  - Use a Branch History Register (BHR) in conjunction with a Pattern History Table (PHT)

- **Example:** \( (k=8, s=6) \)
  - Last \( k \) branches with the behavior \( (11100101) \)
  - History at the entry \( (11100101) \) is \( [101010] \)
  - Using history, branch prediction algorithm predicts direction of the branch

- **Effectiveness:**
  - Averaging 97% accuracy for SPEC
  - Used in the Intel P6 and AMD K6

---

**Global BHR Scheme (GAs)**

- **Branch Address:** \( j \) bits
- **Branch History Shift Register (BHSR):** \( k \) bits
- **BHT of \( 2 \times 2^j \):**

---

**Per-branch BHSR (PAs)**

- **Branch Address:** \( j \) bits
- **Branch History Shift Register (BHSR):** \( k \) bits
- **BHT of \( 2 \times 2^j \):**
**Register Data Flow Techniques**

- **Register Data Flow**
  - Resolving Anti-dependences (WAR)
  - Resolving Output Dependences (WAW)
  - Resolving True Data Dependences (RAW)

- **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
- [WAR] True Dependences
- [WAR] Anti-Dependences
- Output Dependences (WAR)
- Storage Conflicts
- Data Dependences
- Code Dependences

---

**Register Data Flow**

```
Each ALU Instruction:
Ri Fn (Rj, Rk)  (Rj, Rk)
Dest.  Funct.  Unit
Reg.  Source

INSTRUCTION EXECUTION MODEL:

R0 R1 R2...

FU0 FU1...

Registers

FU0 Functional Units

Need Availability of Fs (Structural Dependences)
Need Availability of Rj, Rk (True Data Dependences)
Need Availability of Ri (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

\[
\begin{array}{c}
\text{Ri} \\
\text{•} \\
\text{••} \\
\text{•} \\
\text{•} \\
\text{••} \\
\text{•}\end{array} \rightarrow \begin{array}{c}
\text{DEF} \\
\text{USE} \\
\text{USE} \\
\text{USE} \\
\text{USE} \\
\text{DEF} \\
\text{DEF} \end{array}
\]

**Contribution to Register Recycling**

**COMPILER REGISTER ALLOCATION**
- Single Assignment, Symbolic Reg.
- Map Symbolic Reg. to Physical Reg.
- Maximize Reuse of Reg.

**CODE GENERATION**
- Instruction loops
- Single Assignment, Symbolic Reg.
- Map Symbolic Reg. to Physical Reg.
- Maximize Reuse of Reg.
- Overlapped Execution of Different Iterations

**INSTRUCTION LOOPS**

- $i \leftarrow a[i][k] \cdot b[k][j];$
- $t := a[i][k] \cdot b[k][j];$
- Reuse Same Set of Reg. in Each Iteration

Resolving Anti-Dependences

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

- (1) $R4 \Rightarrow R3 + 1$
- (2) $R5 \Rightarrow R3 + 1$

**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

**RENAME REGISTER**
- Hardware allocation

Resolving Output Dependences

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

- (1) $R3 \Rightarrow R5$
- (2) $R5 \Rightarrow R3$
- (3) $R3 \Rightarrow R5 + 1$

**STALL DISPATCHING/ISSUING**
- Denote output dependence
- Hold dispatching until resolution of dependence
- Allow decoding of subsequent instructions

**RENAME REGISTER**
- Hardware allocation
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
- Attached to Register File (Centralized)
- Attached to functional units (Distributed)

Architected Registers
- Physical Registers

Register Renaming in the RIOS-I FPU

FPU Register Renaming

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.

Resolving True Data Dependences

(1) $R_2 \rightarrow R_1 + 1$
(2) $R_3 \rightarrow R_2$
(3) $R_4 \rightarrow R_3$

STALL DISPATCHING
ADVANCE INSTRUCTIONS
"DYNAMIC EXECUTION"

Reservation Station + Complex Forwarding
Out-of-order (OoO) Execution
Try to Approach the "Data-Flow Limit"

Embedded "Data Flow" Engine

- Read register or assign register tag
- Advance instructions to reservation stations
- Receive data being forwarded
- Issue when all operands ready
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).
  - 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 ←− 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

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)
- Destinations of CDB:
  - Reservation stations
  - Floating-point registers (FLR)
  - Store data buffers (SDB)

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
  - (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**

\[ i: R2 \leftarrow R0 + R4 \]
\[ j: R8 \leftarrow R0 + R2 \]

**Operation of Dependence Mechanisms**

3. **Anti-dependence**  \( \rightarrow \) 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**

\[ i: R4 \leftarrow R0 \times R8 \]
\[ j: R0 \leftarrow R4 \times R2 \]
\[ k: R2 \leftarrow R2 + R8 \]

**Operation of Dependence Mechanisms**

3. **Output dependence**  \( \rightarrow \) 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.
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>IBM 360/91</th>
<th>Modern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>Peak IPC = 1</td>
</tr>
<tr>
<td>Structural hazards</td>
<td>2 FPU Single CDB</td>
</tr>
<tr>
<td>Anti-dependences</td>
<td>Operand copy</td>
</tr>
<tr>
<td>Output dependences</td>
<td>Renamed reg. tag</td>
</tr>
<tr>
<td>True dependences</td>
<td>Tag-based forw.</td>
</tr>
<tr>
<td>Exceptions</td>
<td>Imprecise</td>
</tr>
<tr>
<td>Implementation</td>
<td>3 x 66” x 15” x 78” 60ns cycle time 11-12 gate delays per pipe stage &gt;$1 million</td>
</tr>
</tbody>
</table>

Example 4

i: R4 <- R0 * R8
j: R2 <- R0 + R4
k: R4 <- R0 + R8
l: R8 <- R4 * R2
Example 4

CYCLE #1
ID | Tag Sink | Tag Source | Busy/Tag Data
---|----------|------------|----------------
1  | 2        | 3          | Adder
3  | 1        | 4          | Adder
5  | 2        | 3          | Adder

DISPATCHED INSTRUCTIONS:

CYCLE #2
ID | Tag Sink | Tag Source | Busy/Tag Data
---|----------|------------|----------------
1  | 2        | 3          | Adder
3  | 1        | 4          | Adder
5  | 2        | 3          | Adder

DISPATCHED INSTRUCTIONS:

CYCLE #3
ID | Tag Sink | Tag Source | Busy/Tag Data
---|----------|------------|----------------
1  | 2        | 3          | Adder
3  | 1        | 4          | Adder
5  | 2        | 3          | Adder

DISPATCHED INSTRUCTIONS:

"Dataflow Engine" for Dynamic Execution

Dispatch Buffer
Reg. Write Back
Allocate Recorder Buffer entries

Reservation Stations
Forwarding results to Res. Sta. & rename registers

Compl. Buffer (Reorder Buff.)
Managed as a queue; Maintains sequential order of all instructions in flight ("takeoff" = dispatching; "landing" = completion)
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 architectural 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

- Reorder Buffer
  - “Bookkeeping”
  - Can be instruction-grained, or block-grained (4-5 ops)

Data Capture Reservation Station

- Reservation Stations
  - Data capture vs. no data capture
Memory Data Flow

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

- Basic Memory Hierarchy Review

Memory Data Dependences

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

| WAR | RAW | RAW |
| store X | load X | store X |
| : | : | : |
| store X | store X | load X |

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 V
2. ADD
3. LOAD W
4. LOAD X
5. LOAD V
6. ADD
7. STORE W

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, 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
  - How would you build “history file” for memory?

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 Bypassing of Stores with Forwarding

The DAXPY Example

Optimizing Load/Store Disambiguation

Performance Gains From Weak Ordering

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 unbypassed load – incorrect speculation
  5. Replay starting from incorrect load

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 load/store pair
  - Directly copy store data register to load target register
  - Reduce data transfer latency to absolute minimum

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)

The Memory Bottleneck
Load/Store Processing

1. Effective Address Generation:
   - Must wait on register value
   - Must perform address calculation

2. Address Translation:
   - Must access TLB
   - Can potentially induce a page fault (exception)

For Loads:
- D-cache Access (Read)
  - Can potentially induce a D-cache miss
  - Check aliasing against store buffer for possible load forwarding
  - If bypassing store, must be flagged as “speculative” load until completion

For Stores:
- D-cache Access (Write)
  - When completing must check aliasing against “speculative” loads
  - After completion, wait in store buffer for access to D-cache
  - Can potentially induce a D-cache miss

Easing The Memory Bottleneck

Memory Bottleneck Techniques

Dynamic Hardware (Microarchitecture):
- Use Non-blocking D-cache (need missed-load buffers)
- Use Multiple Load/Store Units (need multiported D-cache)
- Use More Advanced Caches (victim cache, stream buffer)
- Use Hardware Prefetching (need load history and stride detection)

Static Software (Code Transformation):
- Insert Prefetch or Cache-Touch Instructions (mask miss penalty)
- Array Blocking Based on Cache Organization (minimize misses)
- Reduce Unnecessary Load/Store Instructions (redundant loads)
- Software Controlled Memory Hierarchy (expose it to above DSI)

Advanced Memory Hierarchy

- Better miss rate: victim caches
- Reducing miss costs through software restructuring
- Higher bandwidth: Lock-up free caches, superscalar caches
- Beyond simple blocks
- Two level caches
- Prefetching, software prefetching
- Main Memory, DRAM
- Virtual Memory, TLBs
- Interaction of caches, virtual memory
Jouppi’s Victim Cache

- Targeted at conflict misses
- Victim cache: a small fully associative cache
  - Holds victims replaced in direct-mapped or lowassoc
  - LRU replacement
  - A miss in cache + a hit in victim cache
    => move line to main cache
- Poor man’s associativity
  - Not all sets suffer conflicts; provide limited capacity for conflicts

Software Restructuring

- If column-major (Fortran)
  - \( x[i+1, j] \) follows \( x[i,j] \) in memory
  - \( x[i,j+1] \) long after \( x[i,j] \) in memory
- Poor code
  - for \( i = 1 \), rows
    - for \( j = 1 \), columns
      - \( \text{sum} = \text{sum} + x[i,j] \)
- Conversely, if row-major (C/C++)
- Poor code
  - for \( j = 1 \), columns
    - for \( i = 1 \), rows
      - \( \text{sum} = \text{sum} + x[i,j] \)
Superscalar Caches

- Increasing issue width => wider caches
- Parallel cache accesses are harder than parallel functional units
- Fundamental difference:
  - Caches have state, functional units don’t
  - Operation thru one port affects future operations thru others
- Several approaches used
  - True multi-porting
  - Multiple cache copies
  - Virtual multi-porting
  - Multi-banking (interleaving)

Multiple Cache Copies

- Used in DEC Alpha 21164, IBM Power4
- Independent load paths
- Single shared store path
  - May be exclusive with loads, or internally dual-ported
  - Bottleneck, not practically scalable beyond 2 paths
- Provides some fault-tolerance:
  - Parity protection per copy
  - Parity error: restore from known-good copy
  - Avoids more complex ECC (no RMW for subword writes), still provides SEC

Virtual Multiporting

- Used in IBM Power2 and DEC 21264
  - 21264 wave pipelining - pipeline wires WITHOUT latches
- Time-share a single port
- Requires very careful array design to guarantee balanced paths
  - Second access cannot catch up with first access
- Probably not scalable beyond 2 ports
- Complicates and reduces benefit of speed binning

Multi-banking or Interleaving

- Used in Intel Pentium (8 banks)
- Need routing network
- Must deal with bank conflicts
  - Bank conflicts not known till address generated
  - Difficult in non-data-capture machine with speculative scheduling
- Replay: looks just like a cache miss
  - Sensitive to bank interleave: fine-grained vs. coarse-grained
- Spatial locality: many temporally local references to same block
  - Combine these with a “row buffer” approach?
Combined Schemes

• Multiple banks with multiple ports
• Virtual multiporting of multiple banks
• Multiple ports and virtual multiporting
• Multiple banks with multiply virtually multiported ports
• Complexity!
• No good solution known at this time
  – Current generation superscalars get by with 1-3 ports

Beyond Simple Blocks

• Break blocks into
  – Address block associated with tag
  – Transfer block to/from memory (subline, sub-block)
• Large address blocks
  – Decrease tag overhead
  – But allow fewer blocks to reside in cache (fixed mapping)

Beyond Simple Blocks

• Larger transfer block
  – Exploit spatial locality
  – Amortize memory latency
  – But take longer to load
  – Replace more data already cached (more conflicts)
  – Cause unnecessary traffic
• Typically used in large L2/L3 caches to limit tag overhead
• Sublines tracked by MSHR during pending fill

Latency vs. Bandwidth

• Latency can be handled by
  – Hiding (or tolerating) it - out of order issue, nonblocking cache
  – Reducing it – better caches
• Parallelism helps to hide latency
  – MLP – multiple outstanding cache misses overlapped
• But increases bandwidth demand
• Latency ultimately limited by physics
Latency vs. Bandwidth

- Bandwidth can be handled by “spending” more (hardware cost)
  - Wider buses, interfaces
  - Banking/interleaving, multiporting
- Ignoring cost, a well-designed system should never be bandwidth-limited
  - Can’t ignore cost!
- Bandwidth improvement usually increases latency
  - No free lunch
- Hierarchies decrease bandwidth demand to lower levels
  - Serve as traffic filters: a hit in L1 is filtered from L2
- Parallelism puts more demand on bandwidth
- If average b/w demand is not met => infinite queues
  - Bursts are smoothed by queues
- If burst is much larger than average => long queue
  - Eventually increases delay to unacceptable levels

Prefetching

- Even “demand fetching” prefetches other words in block
  - Spatial locality
- Prefetching is useless
  - Unless a prefetch costs less than demand miss
- Ideally, prefetches should
  - Always get data before it is referenced
  - Never get data not used
  - Never prematurely replace data
  - Never interfere with other cache activity

Software Prefetching

- Use compiler to try to
  - Prefetch early
  - Prefetch accurately
- Prefetch into
  - Register (binding)
    - Use normal loads? ROB fills up, fetch stalls
    - What about page faults? Exceptions?
  - Caches (non-binding) – preferred
    - Needs ISA support

- For example:
do j= 1, cols
do ii = 1 to rows by BLOCK
  prefetch (kx[i][j]+BLOCK) # prefetch one block ahead
do i = ii to ii + BLOCK-1
  sum = sum + x[i][j]
- How many blocks ahead should we prefetch?
  - Affects timeliness of prefetches
  - Must be scaled based on miss latency
Hardware Prefetching

- **What to prefetch**
  - One block spatially ahead
  - N blocks spatially ahead
  - Based on observed stride

- **When to prefetch**
  - On every reference
    - Hard to find if block to be prefetched already in the cache
  - On every miss
    - Better than doubling block size
  - Tagged
    - Prefetch when prefetched item is referenced

Prefetching for Pointer-based Data Structures

- **What to prefetch**
  - Next level of tree: n+1, n+2, n+?
    - Entire tree? Or just one path
  - Next node in linked list: n+1, n+2, n+?
  - Jump-pointer prefetching
    - Markov prefetching

- **How to prefetch**
  - Software places jump pointers in data structure
  - Hardware scans blocks for pointers

Stream or Prefetch Buffers

- Prefetching causes capacity and conflict misses (pollution)
  - Can displace useful blocks
- Aimed at compulsory and capacity misses
- Prefetch into buffers, NOT into cache
  - On miss start filling stream buffer with successive lines
  - Check both cache and stream buffer
    - Hit in stream buffer => move line into cache (promote)
    - Miss in both => clear and refill stream buffer
- Performance
  - Very effective for L1-caches, less for D-caches
  - Multiple buffers to capture multiple streams (better for D-caches)
- Can use with any prefetching scheme to avoid pollution

Multilevel Caches

- Ubiquitous in high-performance processors
  - Gap between L1 (core frequency) and main memory too high
  - Level 2 usually on chip, level 3 on or off-chip, level 4 off chip
- Inclusion in multilevel caches
  - Multi-level inclusion holds if L2 cache is superset of L1
  - Can handle virtual address synonyms
  - Filter coherence traffic: if L2 misses, L1 needn’t see snoop
  - Makes L1 writes simpler
    - For both write-through and write-back
Multilevel Inclusion

- Example: local LRU not sufficient to guarantee inclusion
  - Assume L1 holds two and L2 holds three blocks
  - Both use local LRU
- Final state: L1 contains 1, L2 does not
  - Inclusion not maintained
- Different block sizes also complicate inclusion

Multilevel Miss Rates

- Miss rates of lower level caches
  - Affected by upper level filtering effect
  - LRU becomes LRM, since “use” is “miss”
  - Can affect miss rates, though usually not important
- Miss rates reported as:
  - Miss per instruction
  - Global miss rate
  - Local miss rate
  - “Solo” miss rate
  - L2 cache sees all references (unfiltered by L1)

Inclusion takes effort to maintain
- Make L2 cache have bits or pointers giving L1 contents
- Invalidate from L1 before replacing from L2
- In example, removing 1 from L2 also removes it from L1
- Number of pointers per L2 block
  - L2 blocksize/L1 blocksize
- Reading list: [Wang, Baer, Levy ISCA 1989]