EN2910A: Advanced Computer Architecture
Topic 03: Superscalar core architecture

Prof. Sherief Reda
School of Engineering
Brown University

Material from:
• Mostly from *Modern Processor Design* by Shen and Lipasti
• *Parallel Computer Organization and Design* by Debot, Annavaram and Stenstrom
• *Computer Architecture: A Quantitative Approach* by Hennessy and Patterson
Topics

I. Machines for ILP and superscalar pipeline organization (Oct 1\textsuperscript{st} and Oct 3\textsuperscript{rd})

II. Techniques for superscalar machines
   A. Instruction flow techniques (Oct 8\textsuperscript{th} and Oct 10\textsuperscript{th})
   B. Register data flow techniques (Oct 15\textsuperscript{th} and Oct 17\textsuperscript{th})
   C. Memory data flow techniques (Oct 17\textsuperscript{th} and Oct 22\textsuperscript{nd})

III. SW compilation techniques to expose ILP for superscalar and VLIW machines (Oct 24\textsuperscript{th} and Oct 29\textsuperscript{th})
Pipeline performance

Different instructions may have different CPI

\[
\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)
\]

In pipeline designs, best case CPI = 1, when pipeline is always filled, is rarely achieved because of stalls arising from cache misses, structural/data/control hazards and multi-cycle floating point operations.
Example

<table>
<thead>
<tr>
<th>OP</th>
<th>Freq</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>43%</td>
<td>1</td>
</tr>
<tr>
<td>Load</td>
<td>21%</td>
<td>1</td>
</tr>
<tr>
<td>Store</td>
<td>12%</td>
<td>1</td>
</tr>
<tr>
<td>Branch</td>
<td>24%</td>
<td>3</td>
</tr>
</tbody>
</table>

• Assume stores can execute in 2 cycles by slowing clock 25%
• Should this be implemented?
• Old CPI = 0.43 + 0.21 + 0.12 \times 2 + 0.24 \times 3 = 1.48
• New CPI = 0.43 + 0.21 + 0.12 + 0.24 \times 2 = 1.24
• Speedup = old time/new time
  
  \[
  = \frac{P \times \text{old CPI} \times T}{P \times \text{new CPI} \times 1.15 T}
  \]

  \[
  = \frac{1.48}{1.24 \times 1.25} = 0.95
  \]

• Answer: Don’t make the change
The superscalar proposal

- Go beyond single instruction pipeline, achieve IPC > 1
  Dispatch multiple instructions per cycle

- Provide more generally applicable form of concurrency (not just vectors)
  extract instruction-level parallelism (ILP) from sequential code that is hard to parallelize otherwise
Superscalar performance analysis: speedup over pipeline

\[
speedup = \frac{1}{(1 - f)/s + f/n}
\]

- \(f\): vectorizable portion
- \(s\): minimum ILP
- \(n\): number of EX units

- With high values of \(N\), speedup drops-off rate get significantly as \(f\) departs from 1.
- When \(f \leq 0.75\), it is better to have \(s=2\) and \(n=6\) rather than having \(n = 100\) and \(s = 1\).
How much ILP exists?

<table>
<thead>
<tr>
<th>Source</th>
<th>Value</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>Uht [1986]</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. [1986]</td>
<td>2.79</td>
</tr>
<tr>
<td>Wedig [1982]</td>
<td>3.00</td>
</tr>
<tr>
<td>Butler et al. [1991]</td>
<td>5.8</td>
</tr>
<tr>
<td>Melvin and Patt [1991]</td>
<td>6</td>
</tr>
<tr>
<td>Wall [1991]</td>
<td>7 (Jouppi disagreed)</td>
</tr>
<tr>
<td>Kuck et al. [1972]</td>
<td>8</td>
</tr>
<tr>
<td>Riseman and Foster [1972]</td>
<td>51 (no control dependences)</td>
</tr>
<tr>
<td>Nicolau and Fisher [1984]</td>
<td>90 (Fisher’s optimism)</td>
</tr>
</tbody>
</table>

- Function of the benchmark
- Function of the compiler
- Branches and data dependencies impose practical ILP limit
Machines for ILP

Parameters of ILP machines (Jouppi & Wall ’89):

1. **Operation latency (OL):** number of cycles until the result of an instruction is available for use as an operand by a subsequent instruction.

2. **Machine parallelism (MP):** The maximum number of instructions that can be simultaneously in flight in the pipeline at any one time.

3. **Issue parallelism (IP):** The maximum number of instructions that can be issued in every cycle.
Machines for ILP 1/3

1. Baseline scalar k-stage pipeline:

   OL = 1
   MP = k
   IP = 1

2. Scalar superpipeline machine:
   (m – degree of superpipeline)

   - Each stage divided and pipelined into \( m \) stages
   - frequency is \( m \) times base case
   - OL is \( m \) cycles
   - What is the right \( m \) degree?
Machines for ILP 2/3

3. Superscalar machine: \((n \text{ width})\)
   - Can issue \(n\) instructions per cycle.
   - What is the right \(n\)?
   - Is it better to go super pipeline or go super scalar?

4. Superscalar super pipeline machine: \((n, m)\)
   - Has \(m \times\) frequency compared to baseline.
   - Can issue \(n\) instructions per cycle.
   - Meaningful for \(nm\) ILP
4. VLIW (Very Large Instruction Word) machine:

- Each instruction hundreds of bit long.
- Each instruction specifies many operations.
- Compiler schedules and create the instructions.
- Pros/Cons?

5. Vector machine (SIMD):

- Vector machine could superscalar, superpipeline, or attached unit.
- Need high-level of data parallelism.

[series execution only shown for clarity!]
From scalar to superscalar

- Scalar upper bound on throughput
  - $IPC \leq 1$ or $CPI \geq 1$
  - Solution: wide (superscalar) pipeline

- Rigid unified pipeline
  - Long latency for each instruction
  - Solution: diversified, specialized pipelines

- Rigid pipeline stall policy
  - One stalled instruction stalls all newer instructions
  - Solution: Out-of-order execution, distributed execution pipelines
1. Parallel pipelines

A parallel pipeline with width $s$ can concurrently process up to $s$ instructions in each cycle.

- Logic of each stage increases by $s$. Decoding (& dispatching) increases by more than $s$.
- Interstage communication logic increases by $s^2$.
- Register and memory ports increase by $s$. 
Example of static superscalar: Intel’s Pentium

- Introduced 1993
- Technology 0.8 micron
- Transistors: 3.1 million
- Speed: 66 Mhz
- CISC ISA
- # stages: 5
- Issue width: 2-way static

- Two single-stage asymmetric integer ALUs
- One three-stage floating point
- Decode unit examines pairs of instructions and statically issues them simultaneously if there are no hazards and can be supported by both ALUs; otherwise, only first one is issued.
- Results of two instructions are put back in order when completed and written back to register file or memory
2. Diversified pipelines

Problem of unified pipelines: Each instruction type only required a subset of the stages, but must traverse all stages → idling in unnecessary stages.

Advantages of diversified pipes:
• Customization → efficiency
• Minimum necessary latency

Design Issue:
• Need a dispatch unit/stage; checks for hazards.
• Number and mix of functional units?
Example: Motorola 88110

- 10 functional units. Except for divide, all units are single-cycle latency or pipelined.
- If no hazards → dispatches two instructions per cycle.
- If hazards → dispatches first instruction; second instruction moves to 1st dispatch slot and a new instruction takes 2nd dispatch slot.
- Dispatches in order; execution and writing back could be out-of-order.

Introduced 1992
Technology 1 micron
Transistors: 1.3 million
Speed: 50 Mhz
RISC ISA
# stages: 4
Issue width: 2
3. Dynamic pipelines

- Out-of-order (dynamic) execution
- Replace single-entry pipeline buffer by complex multi-entry pipeline buffers → enable instructions to enter and leave the buffers in different orders.
- What to do with Interrupts and exceptions (e.g., arithmetic exceptions, page faults)?
- Re-order buffer puts back instructions in program order for WB → enables precise exceptions.
- Precise exception handling: synchronized with an instruction; all preceding instructions must complete; subsequent instructions flushed; instruction must resume execution after handling.
Superscalar organization

template conceptual 6-stage superscalar pipeline

- Fetch
- Instruction buffer
- Decode
- Dispatch buffer
- Dispatch
- Issuing buffer
- Execute
- Completion buffer
- Complete
- Store buffer
- Retire
1. Issues in fetching

- **Objective**: Fetch multiple instructions, $s$, per cycle
- @ every cycle: use PC to index I-cache. If group of $s$ instructions are in same cache block $\rightarrow$ done in one cycle.
- **Issues**:
  1. **Misalignment**: If group straddles blocks $\rightarrow$ need two cycles $\rightarrow$ reduces fetch bandwidth; might induce a miss for second block. Solutions for misalignment: static SW compiler techniques and HW alignment.
  2. **Control-changing instructions**: What happens if a branch is in the middle of the fetch group?
Example: IBM RS/6000 HW fetch alignment

- **Goal**: issue four instructions per cycle

- Two-way set associative instruction cache organized into 4 subarrays.
- Each block is 64 bytes (16 instructions) and spans 4 physical rows.
- If misalignment is detected, *T-logic units* autoincrement address and *rotation network* reorders presents instructions in original order.
- **Example**: if PC indexes into subarray 2, T-logic of subarrays 0 & 1 auto increments
2. Issues in decoding

- **Primary Tasks:**
  - Identify individual instructions
  - Determine instruction types
  - Determine dependences between instructions
  - Identify control-flow changing instructions to quickly give feedback to fetch unit.
  - For CISC: translate instructions to one or more µops.

- **Important factors:**
  - ISA (RISC vs CISC)
  - Pipeline width impacts dependency detection hardware and register file complexity.
Example: Pentium Pro fetch/decode unit

- @ each cycle: 16 aligned bytes are delivered to the instruction buffer.
- Instructions within the buffer are identified and aligned in IF stage.
- Three parallel decoders decode simultaneously. 1st decoder can handle and translate any instruction (up to 4 uops), 2nd and 3rd decode only simple instructions (1 uop).
- Two pipeline stages for decoding.

P6 microarchitecture, 1995
Technology: 0.5 micron
Transistors: 5.5 million
Speed: 150 Mhz
#stages: 11
Issue width: 5-way dynamic
3. Instruction dispatching

- All functional units can operate independently in a distributed fashion.
- Dispatching unit must resolve all inter-instruction dependencies and dispatch operations to the appropriate functional units.
- Need to read register operands to dispatch with the operation. What if the operands are not yet ready?
3. Dispatching & issuing

- Reservation station: holds instructions that need to wait for operands or unit availability → takes up slack between decoding and execution stages.
- Centralized reservation station: feeds all functional units
- Distributed stations: instructions are dispatched to reservation stations, and then *issued* to functional unit when appropriate.
4. Instruction execution

• Current trends:
  – More parallelism
  – Deeper pipelines
  – More diversity

• Functional unit types
  – Integer
  – Floating point
  – Load/store Branch
  – Specialized units (e.g., MAC, multimedia)
  – What is the best mix and numbers?

• Additional complexities:
  – Bypass network for forwarding is a interconnect from/to FU inputs and outputs; grows quadratically as a function of number of FUs.
5/6. Issues in completion/retirement

- Completion $\rightarrow$ updating processor state.
- Retiring $\rightarrow$ updating memory.
- Sometimes completion and retiring are used interchangeably.
- Should we retire out-of-order?
  - Precise exceptions
- Solutions:
  - Reorder buffer $\rightarrow$ enter out-of-order, but exit in order.
  - Store buffer
Topics

I. Machines for ILP and superscalar pipeline organization (Oct 1\textsuperscript{st} and Oct 3\textsuperscript{rd})

II. Techniques for superscalar machines
   A. Instruction flow techniques (Oct 8\textsuperscript{th} and Oct 10\textsuperscript{th})
   B. Register data flow techniques (Oct 15\textsuperscript{th} and Oct 17\textsuperscript{th})
   C. Memory data flow techniques (Oct 17\textsuperscript{th} and Oct 22\textsuperscript{nd})

III. SW compilation techniques to expose ILP for superscalar and VLIW machines (Oct 24\textsuperscript{th} and Oct 29\textsuperscript{th})
Superscalar techniques

1. **Instruction flow:**
   - Branches, jumps, calls: predict target, direction
   - Fetch alignment
   - Instruction cache misses

2. **Register data flow:**
   - Register renaming: RAW/WAR/WAW

3. **Memory data flow:**
   - In-order stores: WAR/WAW
   - Store queue: RAW
   - Data cache misses
Performance degradation due to branches

• Control-flow changing instructions include jumps (subroutine calls, returns, and goto) and conditional branches (if-else and loops). Form 20% of the code, with a branch instruction encountered every 6 instructions on the average.

• Number of stall cycles can be dictated from address generation and/or conditional resolution.

• In superscalar, number of empty instruction slots is equal to the number of stalled cycles multiplied by superscalar pipeline width!

• Pipeline stalls → sequential bottleneck of Amdahl’s law
Stalled cycles due to target address generation

- Decode: $PC = PC + \text{DISP} \text{ (adder)}$ (1 cycle penalty)
- Dispatch: $PC = (R2)$ (2 cycle penalty)
- Execute: $PC = (R2) + \text{offset}$ (3 cycle penalty)
Stalled cycles due to condition resolution

- Dispatch: 2 cycle penalty
- Execute: 3 cycle penalty
Branch prediction

- **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)
Superscalar organization with branch predictor

Branch Predictor (using a BTB)

- Prediction
- Spec. target
- Spec. cond.

Fetch
Decode Buffer
Decode
Dispatch Buffer
Dispatch
Reservation Stations
Issue
Branch
Execute
Finish
Completion Buffer

PC(seq.) = FA (fetch address)
PC(seq.)
FA-mux

BTB update (target addr. and history)

FA-mux

to I-cache

S. Reda EN2910A FALL’ 13

33
Target speculation

- **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
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
   - 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
   A. Dynamic BP using Smith algorithm
   B. Two-level adaptive (correlating) BP (Yeh and Patt)
   C. Tournament predictors
A. Dynamic branch prediction

- Branch prediction algorithm is based on FSM. Current state used to generate prediction.
- Branch target address and state updated when branch completes executed and actual outcome is known.
- Branch information could be located in a separate table (pattern history table) also indexed by PC.
B. Two-level adaptive (correlating) BP

- In a 2-level predictor we add history bits to access the btb
- History can be global (all branches) or private (this branch only)
- Notation: the first (capital) letter refers to the type of history; the last letter refers to whether each predictor is private in the table.
- g or g means “global”; p or p means “private”
B. GAg two-level BP

Motivation:
if (a==2) then a:=0;
if (b==2) then b:=0;
if (a!=b) then ---

- Outcomes of most recent branch outcomes is stores in branch history register (BHR); the BHR is a shift register where the outcome of each branch is shifted into the end.
- Branches with the same global history interfere with each other and predicted by the same bits
B. GAp two-level BP

- Histories are shared by all branches but predictors are specific to each branch
B. PAg two-level BP

- Per-address history table and global predictor table
- Histories are tracked for each branch but predictors are shared
- All branches with the same per-address history share the same predictor and interfere with each other’s prediction
B. PAp two-level BP

- Per-address history table and per-address predictor table
C. Tournament predictors

- Predictors consists of two or more predictors and a meta predictor
- All predictors are updated based on branch outcome
- Meta predictor makes a prediction of which predictor will be correct

**Example: Corei7 tournament predictor:**
Chooses best of three predictors:
- Two-bit smith BP
- Global history predictor Gap
- Loop-exit counter (counters predict the exact number of taken branches)
- Miss prediction rate is 4% for integer programs and 3% for FP programs
High-bandwidth fetch mechanisms

- Taken branches and cache alignment problem disrupts fetch BW.

1. Collapsing buffers: Consider C a taken branch instruction with target address E, and E is taken branch with target address is G.

2. Trace cache: a cache that stores the dynamic sequence of instructions.
   - Conventional fetch: 5 cycles
   - Collapsing buffer: 3 cycles
   - Trace cache: 1 cycle
Superscalar techniques

1. **Instruction flow:**
   - Branches, jumps, calls: predict target, direction
   - Fetch alignment
   - Instruction cache misses

2. **Register data flow:**
   - Register renaming: RAW/WAR/WAW

3. **Memory data flow:**
   - In-order stores: WAR/WAW
   - Store queue: RAW
   - Data cache misses
Dynamic scheduling in superscalar

• Rearrange order of instructions to reduce stalls while maintaining data flow

• **Advantages:**
  – Ability to hide the latency of cache misses.
  – Avoids stalls because of data dependencies
  – Compiler doesn’t need to have knowledge of microarchitecture
  – Handles cases where dependencies are unknown at compile time

• **Disadvantages:**
  – Substantial increase in hardware complexity
  – Creates new kind of hazards
  – Complicates speculative execution
  – Complicates exceptions
Register hazards from dynamic scheduling

### Read After Write (RAW) hazard (true dependency)

- A true data dependency because values are transmitted between the instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add</code></td>
<td>$s2, $s1, $s0</td>
</tr>
<tr>
<td><code>sub</code></td>
<td>$s4, $s2, $s3</td>
</tr>
</tbody>
</table>

### Write After Read (WAR) hazard (anti dependency)

- Introduced by OOO
- Just a name dependency – no values being transmitted
- Dependency can be removed by renaming registers (either by compiler or HW)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add</code></td>
<td>$s4, $s2, $s0</td>
</tr>
<tr>
<td><code>sub</code></td>
<td>$s2, $s1, $s3</td>
</tr>
</tbody>
</table>

### Write After Write (WAW) hazard (output dependency)

- Introduced by OOO
- Just a name dependency – no values being transmitted
- Dependency can be removed by renaming registers (either by compiler or HW)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add</code></td>
<td>$s2, $s1, $s0</td>
</tr>
<tr>
<td><code>sub</code></td>
<td>$s2, $t2, $t3</td>
</tr>
</tbody>
</table>
Dynamic scheduling using classic Tomasulo’s algorithm

- First implemented in IBM 360/91.
- Tracks when operands are available and forwarded them to where they are needed → handles RAW hazards.
- Introduces register renaming → eliminates WAW and WAR hazards.
- Out of order execution and out of order committing → can’t handle speculative execution or precise exceptions without extensions to hardware organization.
Organization of dynamic execution engine
Tomasulo’s algorithm

- **Dispatch:**
  - Get next instruction from FIFO queue (dispatch buffer)
  - If a RS is available, dispatch the instruction to the RS with operand values if available; else send pending RS tag.
  - If a RS is not available, then stall (structural hazard).

- **Issue/Execute**
  - If instruction in RS is waiting for operand → monitor the CDB and latch results when tag is identified.
  - When all operands are available, issue the instruction; execution can take a number of cycles.
  - No instruction allowed to initiate execution until all branches that proceed it in program order have been completed.

- **Write result**
  - Write result on CDB and from there to reservation stations and RF.
Example

Adder takes two cycles to compute result
Mult/Div takes three cycles to compute result

\[
\begin{align*}
  w & : R4 \leftarrow R0 + R8 \\
  x & : R2 \leftarrow R0 \times R4 \\
  y & : R4 \leftarrow R4 + R8 \\
  z & : R8 \leftarrow R4 \times R2
\end{align*}
\]
Example

Adder takes two cycles to compute result
Mult/Div takes three cycles to compute result

\[w: \quad R4 \leftarrow R0 + R8\]
\[x: \quad R2 \leftarrow R0 \times R4\]
\[y: \quad R4 \leftarrow R4 + R8\]
\[z: \quad R8 \leftarrow R4 \times R2\]
Example

CYCLE 4

\[
\begin{align*}
  & w: R4 \leftarrow R0 + R8 \\
  & x: R2 \leftarrow R0 \times R4 \\
  & y: R4 \leftarrow R4 + R8 \\
  & z: R8 \leftarrow R4 \times R2
\end{align*}
\]

Dispatched instruction(s): 

CYCLE 5

How were RAW, WAR, and WAW hazards handled in the example?
Hardware-based speculation

- Execute instructions along predicted execution paths but only commit the results if prediction was correct
- Instruction commit: allowing an instruction to update the register file when instruction is no longer speculative
- Need to decouple forwarding of results through CDB and actual updating of register and memory → need an additional piece of hardware to prevent any irrevocable action (i.e. updating state or taking an execution) until an instruction commits
Modern dynamic execution engines

- Reorder buffer (ROB) enables speculative execution and precise exceptions.
- Supports multiple issues and writing using wide CDB.
- Store buffer is now not needed.
Re-order/completion buffer

- A ROB is a FIFO buffer holds the results of instruction between completion and commit. An entry is created for every instruction as it is dispatched from the instruction queue (i.e., dispatch buffer).

- Each entry has four fields: (i) instruction type (e.g., branch/store/register); (ii) destination field (e.g., register number or memory address); (iv) value field and (iv) ready field.

- Register values and memory values are not written until an instruction retires (i.e., when it is the oldest instruction in ROB).

- On misprediction: speculated entries in ROB are cleared.

- On exceptions: not recognized until it is ready to retire.

- If a register does not have its value available → a pointer should be included to the ROB entry # that will hold the latest value. Pointers form the register alias table (RAT).
Dynamic execution with speculation

- **Dispatch**: get an instruction from the instruction queue and dispatch to a reservation station if it has an empty slot and an empty slot exists in ROB. Send values from RF, ROB or instead send ROB entry number tag.

- **Issue**: send to execution unit if all operands are available else monitor CDB.

- **Execute and write to CDB**: broadcast results on CDB with ROB entry tag; update waiting entries in reservation stations and ROB.

- **Retire**: when instruction reaches head of ROB; update RF if instruction is speculated correctly; else flush. Reclaim ROB entry.
Superscalar techniques

1. **Instruction flow:**
   - Branches, jumps, calls: predict target, direction
   - Fetch alignment
   - Instruction cache misses

2. **Register data flow:**
   - Register renaming: RAW/WAR/WAW

3. **Memory data flow:**
   - In-order stores: WAR/WAW
   - Store queue: RAW
   - Data cache misses
Load processing

• Example: LD R1, 100[R2]

1. **Dispatch**: Dispatch from instruction queue (dispatch buffer) to LD/ST reservation station and create entry in ROB. Forward values from RF, ROB or instead forward ROB entry number tag.

2. **Execution**: (1) issue to EX unit to compute target address computation and translate from virtual to physical when source; (2) issue to cache to read value.

3. **Completion**: write the read data to CDB along with the ROB entry tag.

4. **Retiring**: when the load’s ROB entry has both its address and value ready, and it is the oldest instruction in the ROB, then it is retired by writing the load value from ROB to the destination register.
Store processing

• Example: ST R1, 50[R2]

1. **Dispatch**: Dispatch from instruction queue (dispatch buffer) to LD/ST reservation station and create entry in ROB. Forward values from RF, ROB or instead forward ROB entry number tag.

2. **Execution**: issue to EX unit to compute target address computation and translate from virtual to physical when source;

3. **Completion**: write the generated to CDB along with the ROB entry tag.

4. **Retiring**: when the store’s ROB entry has both its address and value ready, and it is the oldest instruction in the ROB, then it is retired by writing the store value into the memory.
Memory data dependences

• Loads and stores can execute out of order with respect to other types of instructions. because we can always check for dependencies by comparing register IDs.

• How about loads and stores with respect to each other?
  – Register dependencies are easy to solve:
    • LD R1, 50[R2]
    • LD R3, 100[R1]
  – Can we resolve memory dependencies?
    • “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.
# Avoiding hazards with memory dependencies

<table>
<thead>
<tr>
<th>ST</th>
<th>R1, 50[R2]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>R3, 50[R4]</td>
</tr>
</tbody>
</table>

Retiring values to memory always occur in order → no WAW because of memory aliasing.

<table>
<thead>
<tr>
<th>LD</th>
<th>R1, 50[R2]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>R3, 50[R4]</td>
</tr>
</tbody>
</table>

Retiring values to memory always occur in order → no WAR because of memory aliasing.

<table>
<thead>
<tr>
<th>LD</th>
<th>R1, 50[R2]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>R3, 50[R4]</td>
</tr>
</tbody>
</table>

No issues with memory aliasing.

<table>
<thead>
<tr>
<th>ST</th>
<th>R1, 50[R2]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>R3, 50[R4]</td>
</tr>
</tbody>
</table>

Potential for RAW hazards because of memory aliasing.
Handling RAW memory hazards

```
ST R1, 50[R2]
.
.
LD R3, 50[R4]
```

Potential for RAW hazards because of memory aliasing.

- Possible pessimistic solution: Execute load and stores in order → superscalar processor can take a big performance hit because there are more loads than stores and OOO of load instructions is a potential source of gain as loads enable further operations on data.

\[ Y(i) = A \times X(i) + Y(i) \]

Loop:

- \( F0 \leftarrow LD, a \)
- \( R4 \leftarrow ADDI, Rx, #512 \); last address
- \( F2 \leftarrow LD, 0(Rx) \); load \( X(i) \)
- \( F2 \leftarrow MULTD, F0, F2 \); \( A \times X(i) \)
- \( F4 \leftarrow LD, 0(Ry) \); load \( Y(i) \)
- \( F4 \leftarrow ADDD, F2, F4 \); \( A \times X(i) + Y(i) \)
- \( O(Ry) \leftarrow SD, F4 \); store into \( Y(i) \)
- \( Rx \leftarrow ADDI, Rx, #8 \); inc. index to \( X \)
- \( Ry \leftarrow ADDI, Ry, #8 \); inc. index to \( Y \)
- \( R20 \leftarrow SUB, R4, Rx \); compute bound
- \( BNZ, R20, Loop \); check if done
OOO execution of loads wrt stores

- Maintain program order for the computations of effective address of a load with respect to earlier stores. That is, before executing effective address of a load, make sure that all prior stores in the ROB have their effective address.

- For OOO execution of loads with respect to stores, associatively compare the LD address against all ROB store entries. Then:
  1. No aliasing; great load can bypass store(s).
  2. Aliasing found & store ROB entry has value → forward directly to load.
  3. Aliasing found but store ROB entry does not have → use ROB entry # as tag for load.
Topics

I. Machines for ILP and superscalar pipeline organization (Oct 1\textsuperscript{st} and Oct 3\textsuperscript{rd})

II. Techniques for superscalar machines
   A. Instruction flow techniques (Oct 8\textsuperscript{th} and Oct 10\textsuperscript{th})
   B. Register data flow techniques (Oct 15\textsuperscript{th} and Oct 17\textsuperscript{th})
   C. Memory data flow techniques (Oct 17\textsuperscript{th} and Oct 22\textsuperscript{nd})

III. SW compilation techniques to expose ILP for superscalar and VLIW machines (Oct 24\textsuperscript{th} and Oct 29\textsuperscript{th})
Compilation techniques to expose ILP

Goals of SW compilation techniques for superscalar:
Maximize the ILP extracted from SW by re-arranging instructions during compilation to avoid stalls during runtime and maximize throughput.

Techniques:
1. Pipeline scheduling
2. Code scheduling
3. Loop unrolling
4. Software pipelining

• VLIW architectures: taking superscalar complexity to SW.
1. Pipeline scheduling

- Pipeline scheduling
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction

- Example:
  for (i=999; i>=0; i=i-1)
  
x[i] = x[i] + s;

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>

Loop: |
| L.D | F0,0(R1) |
| ADD.D | F4,F0,F2 |
| S.D | F4,0(R1) |
| DADDUI | R1,R1,#-8 |
| BNE | R1,R2,Loop |
1. Pipeline scheduling example

Loop:  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result</th>
<th>Instruction</th>
<th>Result</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td></td>
<td>ADD.D F4,F0,F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stall</td>
<td>stall</td>
<td>S.D F4,0(R1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>stall</td>
<td>DADDUI R1,R1,#-8</td>
<td>stall (assume integer load latency is 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BNE R1,R2,Loop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. Code scheduling

Code scheduling: compiler re-arranges order of instruction to reduce stalls while still maintains correctness.

Loop:

L.D       F0,0(R1)
DADDUI R1,R1,#-8
ADD.D     F4,F0,F2
stall     stall
S.D       F4,8(R1)
BNE        R1,R2,Loop

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>
3. Loop unrolling

- **Goals:** (1) reduce loop overhead, and (2) enable better scheduling by allowing instructions from different iterations to be scheduled together.

Loop:

- L.D  F0,0(R1)
- ADD.D F4,F0,F2
- S.D  F4,0(R1); drop DADDUI & BNE
- L.D  F6,-8(R1)
- ADD.D F8,F6,F2
- S.D  F8,-8(R1); drop DADDUI & BNE
- L.D  F10,-16(R1)
- ADD.D F12,F10,F2
- S.D  F12,-16(R1); drop DADDUI & BNE
- L.D  F14,-24(R1)
- ADD.D F16,F14,F2
- S.D  F16,-24(R1)
- DADDUI R1,R1,#-32
- BNE  R1,R2,Loop

---

**note:** number of live registers vs. original loop
3. Loop unrolling + scheduling

- schedule the unrolled loop to avoid stalls:

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Instruction</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F0,0(R1)</td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>F6,-8(R1)</td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>F10,-16(R1)</td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>F14,-24(R1)</td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4,F0,F2</td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F8,F6,F2</td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F12,F10,F2</td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F16,F14,F2</td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>F4,0(R1)</td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>F8,-8(R1)</td>
<td></td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1,R1,#-32</td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>F12,16(R1)</td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>F16,8(R1)</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>R1,R2,Loop</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>
3. Loop unrolling limitations

- Improvement from loop overhead tapers off.
- Works well if loop iterations are independent but problem with loop-carried RAW dependency $\rightarrow$ limits amount of unrolling
- **Example:**
  
  ```c
  for (i=2; i<100; i++) a[i] := a[i-2] + b[i];
  ```

- Consumes architectural registers due to renaming $\rightarrow$ register pressure.
- Code expansion $\rightarrow$ impact I-cache and memory
- What if we do not know the loop limit?
4. Software pipelining

- A software-pipelined loop interleaves instructions from different loop iterations (without no need for unrolling), thus separating the dependent instructions within one iteration of the original loop → less chance of stalling.

- The start-up and finish-up code will correspond to the portions above and below the software-pipelined iteration.
4. Software pipelining example

- What are the startup and cleanup codes?
4. Software pipeline vs. loop unrolling

• The major advantage of software pipelining over straight loop unrolling is that software pipelining consumes less code space.

• Loop unrolling reduces the overhead of the loop—the branch and counter update code → improvement eventually tapers off after a few unrolled iterations.

• Software pipelining reduces the time when the loop is not running at peak speed to once per loop at the beginning and end.

• Because these techniques attack two different types of overhead, the best performance can come from doing both.
VLIW architectures: exporting superscalar HW complexity to SW

[from Fisher et al.]
VLIW Pros and Cons

- Compiler schedules and groups instructions to be issued together in Very Large Instruction Words (VLIW) to maximize ILP.
- Compiler detects and avoids hazards:
  - reorder instructions into issue packets
  - no dependencies within a packet
  - pad with NOPs if necessary
- Simplifies superscalar hardware.
- Works well in data-intensive applications with little control.

- Disadvantages:
  - Some hazards can’t be resolved during compile time → compiler must be conservative → loss in performance compared to dynamic superscalar
  - Poor portability and backward compatibility
Summary of superscalar architectures

• **Pros**: Improved single-thread throughput: hide memory latency; avoid or reduces stalls; and ability to fetch and execute multiple instructions per cycle.

• **Cons**: 
  → impacts silicon area
  → impacts power consumption
  → impacts design complexity → impact clock cycle → deep pipelining → more design complexity for speculation HW.

• **SW compilation techniques enable**
  → more ILP from the same HW
  → simplify HW (e.g., VLIW) at the expense of code portability

• **Final verdict**: great single-thread performance but at the expense of energy efficiency.