Lecture 17: Instruction Level Parallelism
-- Hardware Speculation
and VLIW (Static Superscalar)

CSE 564 Computer Architecture Summer 2017

Department of Computer Science and
Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan
Topics for Instruction Level Parallelism

- ILP Introduction, Compiler Techniques and Branch Prediction
  - 3.1, 3.2, 3.3
- Dynamic Scheduling (OOO)
  - 3.4, 3.5 and C.5, C.6 and C.7 (FP pipeline and scoreboard)
- Hardware Speculation and Static Superscalar/VLIW
  - 3.6, 3.7
- Dynamic Scheduling, Multiple Issue and Speculation
  - 3.8, 3.9
- ILP Limitations and SMT
  - 3.10, 3.11, 3.12
Not Every Stage Takes only one Cycle

- **FP EXE Stage**
  - Multi-cycle Add/Mul
  - Nonpipelined for DIV

- **MEM Stage**

---

*Figure C.41* The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating
Issues of Multi-Cycle in Some Stages

- The divide unit is not fully pipelined
  - structural hazards can occur
    » need to be detected and stall incurred.

- The instructions have varying running times
  - the number of register writes required in a cycle can be > 1

- Instructions no longer reach WB in order
  - Write after write (WAW) hazards are possible
    » Note that write after read (WAR) hazards are not possible, since the register reads always occur in ID.

- Instructions can complete in a different order than they were issued (out-of-order complete)
  - causing problems with exceptions

- Longer latency of operations
  - stalls for RAW hazards will be more frequent.
Hazards and Forwarding for Longer-Latency Pipeline

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F4,0(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL.D F0,F4,F6</td>
<td>IF</td>
<td>ID</td>
<td>Stall</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>M7</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D F2,F0,F8</td>
<td>IF</td>
<td>Stall</td>
<td>ID</td>
<td>Stall</td>
<td>Stall</td>
<td>Stall</td>
<td>Stall</td>
<td>Stall</td>
<td>Stall</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F2,0(R2)</td>
<td>IF</td>
<td>Stall</td>
<td>Stall</td>
<td>Stall</td>
<td>Stall</td>
<td>Stall</td>
<td>Stall</td>
<td>Stall</td>
<td>Stall</td>
<td>ID</td>
<td>EX</td>
<td>Stall</td>
<td>Stall</td>
<td>Stall</td>
<td>Stall</td>
<td>MEM</td>
<td></td>
</tr>
</tbody>
</table>

Clock cycle number

**Figure C.37** A typical FP code sequence showing the stalls arising from RAW hazards. The longer pipeline substantially raises the frequency of stalls versus the shallower integer pipeline. Each instruction in this sequence is dependent on the previous and proceeds as soon as data are available, which assumes the pipeline has full bypassing and forwarding. The S.D must be stalled an extra cycle so that its MEM does not conflict with the ADD.D. Extra hardware could easily handle this case.
Problems Arising From Writes

- If we issue one instruction per cycle, how can we avoid structural hazards at the writeback stage and out-of-order writeback issues?
- WAW Hazards

Figure C.38 Three instructions want to perform a write-back to the FP register file simultaneously, as shown in clock cycle 11. This is not the worst case, since an earlier divide in the FP unit could also finish on the same clock. Note that although the MUL.D, ADD.D, and L.D all are in the MEM stage in clock cycle 10, only the L.D actually uses the memory, so no structural hazard exists for MEM.
A load instruction followed by an immediate use results in a 2-cycle stall. Normal forwarding paths can be used after 2 cycles, so the DADD and DSUB get the value by forwarding after the stall. The OR instruction gets
3-Cycle Branch Delay when Taken
## Instruction Scheduling

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_1$</td>
<td>FDIV.D</td>
<td>f6,</td>
<td>f6,</td>
<td>f4</td>
<td></td>
</tr>
<tr>
<td>$I_2$</td>
<td>FLD</td>
<td>f2,</td>
<td>45(x3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_3$</td>
<td>FMULT.D</td>
<td>f0,</td>
<td>f2,</td>
<td>f4</td>
<td></td>
</tr>
<tr>
<td>$I_4$</td>
<td>FDIV.D</td>
<td>f8,</td>
<td>f6,</td>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>$I_5$</td>
<td>FSUB.D</td>
<td>f10,</td>
<td>f0,</td>
<td>f6</td>
<td></td>
</tr>
<tr>
<td>$I_6$</td>
<td>FADD.D</td>
<td>f6,</td>
<td>f8,</td>
<td>f2</td>
<td></td>
</tr>
</tbody>
</table>

**Valid orderings:**

**in-order**

$L_1$ $L_2$ $L_3$ $L_4$ $L_5$ $L_6$

**out-of-order**

$L_2$ $L_1$ $L_3$ $L_4$ $L_5$ $L_6$

**out-of-order**

$L_1$ $L_2$ $L_3$ $L_5$ $L_4$ $L_6$
Register Renaming

- Example:
  - DIV.D     F0,F2,F4
  - ADD.D     F6,F0,F8
  - S.D       F6,0(R1)
  - SUB.D     F8,F10,F14
  - MUL.D     F6,F10,F8

- Now only RAW hazards remain, which can be strictly ordered
How important is renaming?
Consider execution without it

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD</td>
<td>F2, 34(R2)</td>
</tr>
<tr>
<td>2</td>
<td>LD</td>
<td>F4, 45(R3)</td>
</tr>
<tr>
<td>3</td>
<td>MULTD</td>
<td>F6, F4, F2</td>
</tr>
<tr>
<td>4</td>
<td>SUBD</td>
<td>F8, F2, F2</td>
</tr>
<tr>
<td>5</td>
<td>DIVD</td>
<td>F4, F2, F8</td>
</tr>
<tr>
<td>6</td>
<td>ADDD</td>
<td>F10, F6, F4</td>
</tr>
</tbody>
</table>

In-order: 1 (2,1) . . . . . . 2 3 4 4 3 5 . . . 5 6 6 6
Out-of-order: 1 (2,1) 4 4 . . . . . 2 3 . . 3 5 . . . 5 6 6 6

*Out-of-order execution did not allow any significant improvement!**
Instruction-level Parallelism via Renaming

In-order: \( 1 \ (2, 1) \ . \ . \ . \ . \ . \ . \ 2 \ 3 \ 4 \ 4 \ 3 \ 5 \ . \ . \ . \ 5 \ 6 \ 6 \)

Out-of-order: \( 1 \ (2, 1) \ 4 \ 4 \ 5 \ . \ . \ . \ 2 \ (3, 5) \ 3 \ 6 \ 6 \)

Any antidependence can be eliminated by renaming. 

(renaming \( \Rightarrow \) additional storage)

Can be done either in Software or Hardware
Hardware Solution

- **Dynamic Scheduling**
  - Out-of-order execution and completion

- **Data Hazard via Register Renaming**
  - Dynamic RAW hazard detection and scheduling in data-flow fashion
  - Register renaming for WRW and WRA hazard (name conflict)

- **Implementations**
  - Scoreboard (CDC 6600 1963)
    - Centralized register renaming
  - Tomasulo’s Approach (IBM 360/91, 1966)
    - Distributed control and renaming via reservation station, load/store buffer and common data bus (data+source)
Organizations of Tomasulo’s Algorithm

- Load/Store buffer
- Reservation station
- Common data bus
Three Stages of Tomasulo Algorithm

1. **Issue**—get instruction from FP Op Queue
   - If reservation station free (no structural hazard), control issues instr & sends operands (renames registers).

2. **Execution**—operate on operands (EX)
   - When both operands ready then execute; if not ready, watch Common Data Bus for result

3. **Write result**—finish execution (WB)
   - Write on Common Data Bus to all awaiting units; mark reservation station available

- Normal data bus: data + destination (“go to” bus)
- **Common data bus**: data + source (“come from” bus)
  - 64 bits of data + 4 bits of Functional Unit source address
  - Write if matches expected Functional Unit (produces result)
  - Does the broadcast
Tomasulo Example Cycle 3

**Instruction status:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>Issue</th>
<th>Comp</th>
<th>Result</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F6</td>
<td>34+</td>
<td>R2</td>
<td>1</td>
<td>3</td>
<td>Load1</td>
<td>Yes 34+R2</td>
</tr>
<tr>
<td>LD</td>
<td>F2</td>
<td>45+</td>
<td>R3</td>
<td>2</td>
<td></td>
<td>Load2</td>
<td>Yes 45+R3</td>
</tr>
<tr>
<td>MULTD</td>
<td>F0</td>
<td>F2</td>
<td>F4</td>
<td>3</td>
<td></td>
<td>Load3</td>
<td>No</td>
</tr>
<tr>
<td>SUBD</td>
<td>F8</td>
<td>F6</td>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVD</td>
<td>F10</td>
<td>F0</td>
<td>F6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD</td>
<td>F6</td>
<td>F8</td>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Reservation Stations:**

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Vk</th>
<th>Qj</th>
<th>Qk</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add3</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mult1</td>
<td>Yes</td>
<td>MULTD</td>
<td>R(F4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mult2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register result status:**

<table>
<thead>
<tr>
<th>Clock</th>
<th>F0</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
<th>F12</th>
<th>...</th>
<th>F30</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Note:** registers names are removed ("renamed") in Reservation Stations
Register Renaming Summary

- **Purpose of Renaming:** removing “Anti-dependencies”
  - Get rid of WAR and WAW hazards, since these are not “real” dependencies

- **Implicit Renaming:** i.e. Tomasulo
  - Registers changed into values or response tags
  - We call this “implicit” because space in register file may or may not be used by results!

- **Explicit Renaming:** more physical registers than needed by ISA.
  - Rename table: tracks current association between architectural registers and physical registers
  - Uses a translation table to perform compiler-like transformation on the fly

- **With Explicit Renaming:**
  - All registers concentrated in single register file
  - Can utilize bypass network that looks more like 5-stage pipeline
  - Introduces a register-allocation problem
    - Need to handle branch misprediction and precise exceptions differently, but ultimately makes things simpler
Explicit Register Renaming

- Tomasulo provides * Implicit Register Renaming *
  - User registers renamed to reservation station tags

- Explicit Register Renaming:
  - Use *physical* register file that is larger than number of registers specified by ISA

- Keep a translation table:
  - ISA register => physical register mapping
  - When register is written, replace table entry with new register from freelist.
  - Physical register becomes free when not being used by any instructions in progress.

- Pipeline can be exactly like “standard” DLX pipeline
  - IF, ID, EX, etc....

- Advantages:
  - Removes all WAR and WAW hazards
  - Like Tomasulo, good for allowing full out-of-order completion
  - Allows data to be fetched from a single register file
  - Makes speculative execution/precise interrupts easier:
    » All that needs to be “undone” for precise break point is to undo the table mappings
Explicit Renaming Support Includes:

- Rapid access to a table of translations
- A physical register file that has more registers than specified by the ISA
- Ability to figure out which physical registers are free.
  - No free registers $\Rightarrow$ stall on issue
- Thus, register renaming doesn’t require reservation stations.

Many modern architectures use explicit register renaming + Tomasulo-like reservation stations to control execution.
  - R10000, Alpha 21264, HP PA8000
HARDWARE SPECULATION: ADDRESSING CONTROL HAZARDS
Control Hazard from Branches: Three Stage Stall if Taken

10: BEQ R1, R3, 36
14: AND R2, R3, R5
18: OR R6, R1, R7
22: ADD R8, R1, R9
36: XOR R10, R1, R11

What do you do with the 3 instructions in between?
How do you do it?
Control Hazards

- Break the instruction flow
- Unconditional Jump
- Conditional Jump
- Function call and return
- Exceptions
Independent “Fetch” unit

Stream of Instructions
To Execute

- Instruction Fetch with Branch Prediction
- Out-Of-Order Execution Unit

Correctness Feedback
On Branch Results

- Instruction fetch decoupled from execution
- Often issue logic (+ rename) included with Fetch
Branches must be resolved quickly

- **The loop-unrolling example**
  - we relied on the fact that branches were under control of “fast” integer unit in order to get overlap!

- **Loop:**
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>F0</th>
<th>F4</th>
<th>F2</th>
<th>R1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNEZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **What happens if branch depends on result of multd??**
  - We completely lose all of our advantages!
  - Need to be able to “predict” branch outcome.
    - If we were to predict that branch was taken, this would be right most of the time.

- Problem **much** worse for superscalar (issue multiple instrs per cycle) machines!
Precision Exception

- Out-of-order completion

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Set(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIV.D</td>
<td>F0,F2,F4</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F10,F10,F8</td>
</tr>
<tr>
<td>SUB.D</td>
<td>F12,F12,F14</td>
</tr>
</tbody>
</table>

- ADD.D completed before DIV.D which raises exception (e.g. divide by zero)
  - When handled by exception handler, the state of the machine does not represents where the exception is raised
Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!

How much work is lost if pipeline doesn’t follow correct instruction flow?

~ Loop length x pipeline width
Reducing Control Flow Penalty

- **Software solutions**
  - **Eliminate branches - loop unrolling**
    » Increases the run length
  - **Reduce resolution time - instruction scheduling**
    » Compute the branch condition as early as possible (of limited value)

- **Hardware solutions**
  - **Find something else to do - delay slots**
    » Replaces pipeline bubbles with useful work (requires software cooperation)

- **Branch prediction**
  - **Speculative execution of instructions beyond the branch**
Branch Prediction

- **Motivation:**
  - Branch penalties limit performance of deeply pipelined processors
  - Modern branch predictors have high accuracy: (>95%) and can reduce branch penalties significantly

- **Required hardware support**
  - Branch history tables (Taken or Not)
  - Branch target buffers, etc. (Target address)
Mispredict Recovery

In-order execution machines:
- Assume no instruction issued after branch can write-back before branch resolves
- Kill all instructions in pipeline behind mispredicted branch

Out-of-order execution:
- Multiple instructions following branch in program order can complete before branch resolves
Mispredict Recovery

- Keep result computation separate from commit
- Kill instructions following branch in pipeline
- Restore state to state following branch

Hardware Speculation = Prediction + Mispredict Recovery
Speculative Execution

- **Speculative**: issued, executed, but not yet committed

1. **Fetch**: Instruction bits retrieved from cache.
2. **Decode**: Instructions placed in appropriate issue (aka “dispatch”) stage buffer.
3. **Execute**: Instructions and operands sent to execution units. When execution completes, all results and exception flags are available.
4. **Commit**: Instruction irrevocably updates architectural state (aka “graduation” or “completion”).
In-Order Commit for Control Hazards

- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (⇒ out-of-order completion)
- Commit (write-back to architectural state, i.e., regfile & memory) is in-order

Temporary storage needed to hold results before commit (shadow registers and store buffers)
Reorder Buffer

**Idea:**
- record instruction issue order
- Allow them to execute out of order
- Reorder them so that they commit in-order

**On issue:**
- Reserve slot at tail of ROB
- Record dest reg, PC
- Tag u-op with ROB slot

**Done execute**
- Deposit result in ROB slot
- Mark exception state

**WB head of ROB**
- Check exception, handle
- Write register value, or
- Commit the store
Reorder Buffer + Forwarding

- **Idea:**
  - Forward uncommitted results to later uncommitted operations

- **Trap**
  - Discard remainder of ROB

- **Opfetch / Exec**
  - Match source reg against all dest regs in ROB
  - Forward last (once available)
Reorder Buffer + Forwarding + Speculation

- **Idea:**
  - Issue branch into ROB
  - Mark with prediction
  - Fetch and issue predicted instructions speculatively
  - Branch must resolve before leaving ROB
  - Resolve correct
    » Commit following instr
  - Resolve incorrect
    » Mark following instr in ROB as invalid
    » Let them clear
How do you find the latest version of a register?
- As specified by Smith paper, need associative comparison network
- Could use future file or just use the register result status buffer to track which specific reorder buffer has received the value

Need as many ports on ROB as register file
Hardware Speculation in Tomasulo Algorithm

- + Reorder Buffer
- - Store Buffer
  - Integrated in ROF
Four Steps of Speculative Tomasulo

1. Issue—get instruction from FP Op Queue
   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called “dispatch”)

2. Execution—operate on operands (EX)
   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called “issue”)

3. Write result—finish execution (WB)
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.

4. Commit—update register with reorder result
   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called “graduation”)
Instruction In-order Commit

- Also called completion or graduation
- In-order commit
  - In-order issue
  - Out-of-order execution
  - Out-of-order completion
- Three cases when an instr reaches the head of ROB
  - Normal commit: when an instruction reaches the head of the ROB and its result is present in the buffer
    - The processor updates the register with the result and removes the instruction from the ROB.
  - Committing a store:
    - is similar except that memory is updated rather than a result register.
  - A branch with incorrect prediction
    - indicates that the speculation was wrong.
    - The ROB is flushed and execution is restarted at the correct successor of the branch.
Example with ROB and Reservation (Dynamic Scheduling and Speculation)

- MUL.D is ready to commit

After SUB.D completes execution, if exception happens by MUL.D ....
In-order Commit with Branch

Loop:  
L.D    F0,0(R1)  
MUL.D  F4,F0,F2  
S.D    F4,0(R1)  
DADDIU R1,R1,#-8  
BNE    R1,R2,Loop ;branches if R1

<table>
<thead>
<tr>
<th>Entry</th>
<th>Busy</th>
<th>Instruction</th>
<th>State</th>
<th>Destination</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No</td>
<td>L.D</td>
<td>Commit</td>
<td>F0</td>
<td>Mem[0 + Regs[R1]]</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>MUL.D</td>
<td>Commit</td>
<td>F4</td>
<td>#1 × Regs[F2]</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>S.D</td>
<td>Write result</td>
<td>0 + Regs[R1]</td>
<td>#2</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>DADDIU</td>
<td>Write result</td>
<td>R1</td>
<td>Regs[R1] – 8</td>
</tr>
<tr>
<td>5</td>
<td>Yes</td>
<td>BNE</td>
<td>Write result</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Yes</td>
<td>L.D</td>
<td>Write result</td>
<td>F0</td>
<td>Mem[#4]</td>
</tr>
<tr>
<td>7</td>
<td>Yes</td>
<td>MUL.D</td>
<td>Write result</td>
<td>F4</td>
<td>#6 × Regs[F2]</td>
</tr>
<tr>
<td>8</td>
<td>Yes</td>
<td>S.D</td>
<td>Write result</td>
<td>0 + #4</td>
<td>#7</td>
</tr>
<tr>
<td>9</td>
<td>Yes</td>
<td>DADDIU</td>
<td>Write result</td>
<td>R1</td>
<td>#4 – 8</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>BNE</td>
<td>Write result</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FP register status

<table>
<thead>
<tr>
<th>Field</th>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reorder #</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Busy</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

In the reorder buffer, entry 1 and entry 2 are flushed due to branch prediction. Entry 1 is flushed because of an IF misprediction.
Memory Disambiguation: RAW Hazards in memory

- Question: Given a load that follows a store in program order, are the two related?
  - (Alternatively: is there a RAW hazard between the store and the load)?

  Eg: 
  \[
  \begin{align*}
  \text{st} & \quad 0(R2), R5 \\
  \text{ld} & \quad R6, 0(R3)
  \end{align*}
  \]

- Can we go ahead and start the load early?
  - Store address could be delayed for a long time by some calculation that leads to R2 (divide?).
  - We might want to issue/begin execution of both operations in same cycle.
  - Today: Answer is that we are not allowed to start load until we know that address \( 0(R2) \neq 0(R3) \)
  - Other advanced techniques: We might guess at whether or not they are dependent (called “dependence speculation”) and use reorder buffer to fixup if we are wrong.
Hardware Support for Memory Disambiguation

- Need buffer to keep track of all outstanding stores to memory, in program order.
  - Keep track of address (when becomes available) and value (when becomes available)
  - FIFO ordering: will retire stores from this buffer in program order

- When issuing a load, record current head of store queue (know which stores are ahead of you).

- When have address for load, check store queue:
  - If any store prior to load is waiting for its address, stall load.
  - If load address matches earlier store address (associative lookup), then we have a memory-induced RAW hazard:
    » store value available ⇒ return value
    » store value not available ⇒ return ROB number of source
  - Otherwise, send out request to memory

- Actual stores commit in order, so no worry about WAR/WAW hazards through memory.
Relationship between precise interrupts, branch and speculation:

- Speculation is a form of guessing
  - Branch prediction, data prediction
  - If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
  - This is exactly same as precise exceptions!

- Branch prediction is a very important!
  - Need to “take our best shot” at predicting branch direction.
  - If we issue multiple instructions per cycle, lose lots of potential instructions otherwise:
    » Consider 4 instructions per cycle
    » If take single cycle to decide on branch, waste from 4 - 7 instruction slots!

- Technique for both precise interrupts/exceptions and speculation: *in-order completion or commit*
  - This is why reorder buffers in all new processors
Dynamic Scheduling and Speculation

- **ILP Maximized (a restricted data-flow)**
  - In-order issue
  - Out-of-order execution
  - Out-of-order completion
  - In-order commit

- **Data Hazards**
  - Input operands-driven dynamic scheduling for RAW hazard
  - Register renaming for handling WAR and WAW hazards

- **Control Hazards (Branching, Precision Exception)**
  - Branch prediction and in-order commit

- **Implementation: Tomasulo**
  - Reservation stations and Reorder buffer
  - Other solutions as well (scoreboard, history table)
MULTIPLE ISSUE VIA VLIW/STATIC SUPERSCALAR
Multiple Issue

- Issue multiple instructions in one cycle
- Three major types (VLIW and superscalar)
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - Dynamically scheduled superscalar processors

Superscalar
- Variable # of instr per cycle
- In-order execution for static superscalar
- Out-of-order execution for dynamic superscalar

VLIW
- Issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallel-ism among instructions explicitly indicated by the instruction.
- Inherently statically scheduled by the compiler
- Intel/HP IA-64 architecture, named EPIC—explicitly parallel instruction computer
  » Appendix H,
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Issue structure</th>
<th>Hazard detection</th>
<th>Scheduling</th>
<th>Distinguishing characteristic</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Static</td>
<td>In-order execution</td>
<td>Mostly in the embedded space:</td>
</tr>
<tr>
<td>(static)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MIPS and ARM, including the ARM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Cortex-A8</td>
</tr>
<tr>
<td>Superscalar</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic</td>
<td>Some out-of-order execution, but no speculation</td>
<td>None at the present</td>
</tr>
<tr>
<td>(dynamic)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Superscalar</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic with speculation</td>
<td>Out-of-order execution with speculation</td>
<td>Intel Core i3, i5, i7; AMD Phenom;</td>
</tr>
<tr>
<td>(speculative)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IBM Power 7</td>
</tr>
<tr>
<td>VLIW/LIW</td>
<td>Static</td>
<td>Primarily software</td>
<td>Static</td>
<td>All hazards determined and indicated by compiler</td>
<td>Most examples are in signal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(often implicitly)</td>
<td>processing, such as the TI C6x</td>
</tr>
<tr>
<td>EPIC</td>
<td>Primarily static</td>
<td>Primarily software</td>
<td>Mostly static</td>
<td>All hazards determined and indicated explicitly by the</td>
<td>Itanium</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>compiler</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 3.15 The five primary approaches in use for multiple-issue processors and the primary characteristics that distinguish them.* This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. Appendix H focuses on compiler-based approaches. The EPIC approach, as embodied in the IA-64 architecture, extends many of the concepts of the early VLIW approaches, providing a blend of static and dynamic approaches.
VLIW and Static Superscalar

- Very similar in terms of the requirements for compiler and hardware support
- We will discuss VLIW

- Very Long Instruction Word (VLIW)
  - packages the multiple operations into one very long instruction
VLIW: Very Large Instruction Word

- Each “instruction” has explicit coding for multiple operations
  - In IA-64, grouping called a “packet”
  - In Transmeta, grouping called a “molecule” (with “atoms” as ops)

- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 1 integer operation/branch, 2 FP ops, 2 Memory refs
    - 16 to 24 bits per field => 5*16 or 80 bits to 5*24 or 120 bits wide
  - Need compiling technique that schedules across several branches
Recall: Unrolled Loop that Minimizes Stalls for Scalar

1 Loop: L.D F0,0(R1)
2 L.D F6,-8(R1)
3 L.D F10,-16(R1)
4 L.D F14,-24(R1)
5 ADD.D F4,F0,F2
6 ADD.D F8,F6,F2
7 ADD.D F12,F10,F2
8 ADD.D F16,F14,F2
9 S.D 0(R1),F4
10 S.D -8(R1),F8
11 S.D -16(R1),F12
12 DSUBUI R1,R1,#32
13 BNEZ R1,LOOP
14 S.D 8(R1),F16 ; 8–32 = -24

14 clock cycles, or 3.5 per iteration

for (i=999; i>=0; i=i–1)
\[ x[i] = x[i] + s; \]
Loop Unrolling in VLIW

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
Average: 2.5 ops per clock, 50% efficiency

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP operation 2</th>
<th>Integer operation/branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>S.D F8,-8(R1)</td>
<td>ADD.D F28,F26,F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F12,-16(R1)</td>
<td>S.D F16,-24(R1)</td>
<td></td>
<td>DADDUI R1,R1,#-56</td>
<td></td>
</tr>
<tr>
<td>S.D F20,24(R1)</td>
<td>S.D F24,16(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F28,8(R1)</td>
<td></td>
<td></td>
<td>BNE R1,R2,Loop</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.16 VLIW instructions that cycles assuming no branch delay; norrations in 9 clock cycles, or 2.5 operation, is about 60%. To achieve this loop. The VLIW code sequence a MIPS processor can use as few as two l

<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 Unrolling in VLIW

- **Unroll 8 times**
  - **Enough registers**

8 results in 9 clocks, or 1.125 clocks per iteration

**Average:** 2.89 (26/9) ops per clock, 58% efficiency (26/45)

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP operation 2</th>
<th>Integer operation/branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td>L.D</td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>S.D F8,-8(R1)</td>
<td>ADD.D F28,F26,F2</td>
<td>ADD.D</td>
<td></td>
</tr>
<tr>
<td>S.D F12,-16(R1)</td>
<td>S.D F16,-24(R1)</td>
<td></td>
<td>DADDUI R1,R1,#-56</td>
<td></td>
</tr>
<tr>
<td>S.D F20,24(R1)</td>
<td>S.D F24,16(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F28,8(R1)</td>
<td>S.D</td>
<td></td>
<td>BNE R1,R2,Loop</td>
<td></td>
</tr>
</tbody>
</table>

---

**Figure 3.16** VLIW instructions that cycles assuming no branch delay; narrations in 9 clock cycles, or 2.5 operations, about 60%. To achieve this loop. The VLIW code sequence about MIPS processor can use as few as two instructions producing result | Instruction using result | Latency in clock cycles |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></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 Unrolling in VLIW

- **Unroll 10 times**
  - **Enough registers**

10 results in 10 clocks, or 1 clock per iteration

**Average:** 3.2 ops per clock (32/10), 64% efficiency (32/50)

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP operation 2</th>
<th>Integer operation/branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td>L.D ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>L.D ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>S.D F8,-8(R1)</td>
<td>ADD.D F28,F26,F2</td>
<td>ADD.D</td>
<td>DADDUI R1,R1,#-56</td>
</tr>
<tr>
<td>S.D F12,-16(R1)</td>
<td>S.D F16,-24(R1)</td>
<td>ADD.D</td>
<td>ADD.D</td>
<td></td>
</tr>
<tr>
<td>S.D F20,24(R1)</td>
<td>S.D F24,16(R1)</td>
<td></td>
<td></td>
<td>BNE R1,R2,Loop</td>
</tr>
<tr>
<td>S.D F28,8(R1)</td>
<td>S.D</td>
<td></td>
<td></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>
Problems with 1st Generation VLIW

- **Increase in code size**
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding

- **Operated in lock-step; no hazard detection HW**
  - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might prediction function units, but caches hard to predict

- **Binary code compatibility**
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code
Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- **IA-64**: instruction set architecture
  - 128 64-bit integer regs + 128 82-bit floating point regs
    » Not separate register files per functional unit as in old VLIW
  - Hardware checks dependencies (interlocks ⇒ binary compatibility over time)

- 3 Instructions in 128 bit “bundles”; field determines if instructions dependent or independent
  - Smaller code size than old VLIW, larger than x86/RISC
  - Groups can be linked to show independence > 3 instr

- Predicated execution (select 1 out of 64 1-bit flags) ⇒ 40% fewer mispredictions?

- Speculation Support:
  - deferred exception handling with “poison bits”
  - Speculative movement of loads above stores + check to see if incorrect

- **Itanium™** was first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 µ process

- **Itanium 2™** is name of 2nd implementation (2005)
  - 6-wide, 8-stage pipeline at 1666Mhz on 0.13 µ process
  - Caches: 32 KB I, 32 KB D, 128 KB L2I, 128 KB L2D, 9216 KB L3
Summary

- **VLIW: Explicitly Parallel, Static Superscalar**
  - Requires advanced and aggressive compiler techniques
  - Trace Scheduling: Select primary “trace” to compress + fixup code

- **Other aggressive techniques**
  - **Boosting:** Moving of instructions above branches
    - Need to make sure that you get same result (i.e. do not violate dependencies)
    - Need to make sure that exception model is same (i.e. not unsafe)

- **Itanium/EPIC/VLIW is not a breakthrough in ILP**
  - If anything, it is as complex or more so than a dynamic processor