1. CPI < 1? How?

→ Multiple issue processors:
  • VLIW (Very Long Instruction Word)
  • Superscalar processors

2. What if dynamic branch prediction is wrong?

→ Speculative Tomasulo Processor

Or delay start of execution of instructions following a branch until after the branch is resolved.
Evolution of Microprocessor Performance

So far we examined static & dynamic techniques to improve the performance of single-issue (scalar) pipelined CPU designs including: static & dynamic scheduling, static & dynamic branch predication. Even with these improvements, the restriction of issuing a single instruction per cycle still limits the ideal CPI = 1.

We next examine the two approaches to achieve a CPI < 1 by issuing multiple instructions per cycle:

- Superscalar CPUs
- Very Long Instruction Word (VLIW) CPUs.

Single-issue Processor = Scalar Processor
Instructions Per Cycle (IPC) = 1/CPI
Parallelism in Microprocessor VLSI Generations

Bit-level parallelism
Multiple micro-operations per cycle (multi-cycle non-pipelined)
Not Pipelined
CPI >> 1

Instruction-level
Single-issue Pipelined
CPI = 1

Thread-level (?) (TLP)
Simultaneous Multithreading SMT:
e.g. Intel's Hyper-threading
Chip-Multiprocessors (CMPs)
e.g IBM Power 4, 5
AMD Athlon64 X2
Intel Pentium D

Superscalar Processors
Very Long Instruction Word (VLIW) ISA/Processors
CPI < 1

Multi-core Processors

Transistors


100,000,000
10,000,000
1,000,000
100,000
10,000
1,000

Superscalar /VLIW CPI < 1
Pentium
R10000
R3000
R2000
i80386
i4004
i8080
i8086
i80286

Thread Level Parallelism (TLP)

Pipelining
Multiple Instruction Issue

Multiple micro-operations per cycle (multi-cycle non-pipelined)

Simultaneous Multithreading SMT:
e.g. Intel's Hyper-threading
Chip-Multiprocessors (CMPs)
e.g IBM Power 4, 5
AMD Athlon64 X2
Intel Pentium D

Superscalar Processors
Very Long Instruction Word (VLIW) ISA/Processors
CPI < 1

Multi-core Processors

Transistors


100,000,000
10,000,000
1,000,000
100,000
10,000
1,000

Superscalar /VLIW CPI < 1
Pentium
R10000
R3000
R2000
i80386
i4004
i8080
i8086
i80286

Thread Level Parallelism (TLP)

Pipelining
Multiple Instruction Issue

Multiple micro-operations per cycle (multi-cycle non-pipelined)

Simultaneous Multithreading SMT:
e.g. Intel's Hyper-threading
Chip-Multiprocessors (CMPs)
e.g IBM Power 4, 5
AMD Athlon64 X2
Intel Pentium D

Superscalar Processors
Very Long Instruction Word (VLIW) ISA/Processors
CPI < 1

Multi-core Processors

Transistors


100,000,000
10,000,000
1,000,000
100,000
10,000
1,000

Superscalar /VLIW CPI < 1
Pentium
R10000
R3000
R2000
i80386
i4004
i8080
i8086
i80286

Thread Level Parallelism (TLP)

Pipelining
Multiple Instruction Issue

Multiple micro-operations per cycle (multi-cycle non-pipelined)

Simultaneous Multithreading SMT:
e.g. Intel's Hyper-threading
Chip-Multiprocessors (CMPs)
e.g IBM Power 4, 5
AMD Athlon64 X2
Intel Pentium D

Superscalar Processors
Very Long Instruction Word (VLIW) ISA/Processors
CPI < 1

Multi-core Processors

Transistors


100,000,000
10,000,000
1,000,000
100,000
10,000
1,000

Superscalar /VLIW CPI < 1
Pentium
R10000
R3000
R2000
i80386
i4004
i8080
i8086
i80286

Thread Level Parallelism (TLP)

Pipelining
Multiple Instruction Issue

Multi-core Processors
Multiple Instruction Issue: CPI < 1

• To improve a pipeline’s CPI to be better [less] than one, and to better exploit Instruction Level Parallelism (ILP), a number of instructions have to be issued in the same cycle.

• Multiple instruction issue processors are of two types:
  1. **Superscalar:** A number of instructions (2-8) is issued in the same cycle, scheduled statically by the compiler or -more commonly- dynamically (Tomasulo).
     - PowerPC, Sun UltraSparc, Alpha, HP 8000, Intel PII, III, 4 ...
     - No ISA Support Needed
  2. **VLIW (Very Long Instruction Word):** A fixed number of instructions (3-6) are formatted as one long instruction word or packet (statically scheduled by the compiler).
     - Example: Explicitly Parallel Instruction Computer (EPIC)
     - Originally a joint HP/Intel effort.
     - ISA: Intel Architecture-64 (IA-64) 64-bit address:
     - IA-64

• Limitations of the approaches:
  - Available ILP in the program (both).
  - Specific hardware implementation difficulties (superscalar).
  - VLIW optimal compiler design issues.

CPI < 1 or Instructions Per Cycle (IPC) > 1
Simple Statically Scheduled Superscalar Pipeline

- Two instructions can be issued per cycle (static two-issue or 2-way superscalar).
- One of the instructions is integer (including load/store, branch). The other instruction is a floating-point operation.
  - This restriction reduces the complexity of hazard checking.
- Hardware must fetch and decode two instructions per cycle.
- Then it determines whether zero (a stall), one or two instructions can be issued (in decode stage) per cycle.

<table>
<thead>
<tr>
<th>Instruction Type</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>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>

Two-issue statically scheduled pipeline in operation
FP instructions assumed to be adds (EX takes 3 cycles)

Ideal CPI = 0.5  Ideal Instructions Per Cycle (IPC) = 2
Intel IA-64: VLIW “Explicitly Parallel Instruction Computing (EPIC)”

- Three 41-bit instructions in 128 bit “Groups” or bundles; an instruction bundle template field (5-bits) determines if instructions are dependent or independent and statically specifies the functional units to used by the instructions:
  - Smaller code size than old VLIW, larger than x86/RISC
  - Groups can be linked to show dependencies of more than three instructions.

- 128 integer registers + 128 floating point registers

- Hardware checks dependencies (interlocks ⇒ binary compatibility over time)

- Predicated execution: An implementation of conditional instructions used to reduce the number of conditional branches used in the generated code ⇒ larger basic block size

  - **IA-64**: Name given to instruction set architecture (ISA).
  - **Itanium**: Name of the first implementation (2001).

In VLIW dependency analysis is done statically by the compiler not dynamically in hardware (Tomasulo)
Intel/HP EPIC VLIW Approach

original source code

compiler

Instruction Dependency Analysis

Expose Instruction Parallelism (dependency analysis)

Optimize

Instruction 2
Instruction 1
Instruction 0
Template

128-bit bundle

127
0

Instruction 2
41 bits

Instruction 1
41 bits

Instruction 0
41 bits

Template
5 bits

Instruction Dependency Graph

Template field has static assignment/scheduling information

Sequential Code

Exploit Instruction Parallelism: Generate VLIWs

CMPE550 - Shaaban
IA-64 Instruction Types

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Description</th>
<th>Execution Unit Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Integer ALU</td>
<td>I-unit or M-unit</td>
</tr>
<tr>
<td>I</td>
<td>Non-integer ALU</td>
<td>I-unit</td>
</tr>
<tr>
<td>M</td>
<td>Memory</td>
<td>M-unit</td>
</tr>
<tr>
<td>F</td>
<td>Floating Point</td>
<td>F-unit</td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
<td>B-unit</td>
</tr>
<tr>
<td>L+X</td>
<td>Extended</td>
<td>I-unit/B-unit</td>
</tr>
</tbody>
</table>

Information on static assignment of instructions to functional units and instruction typed in a bundle contained in the template field.
IA-64 Template Use

- The template specifies the functional units for the three operations (instructions) in the instruction bundle.
  - Part of static scheduling

- Possible instruction combinations:
  - M-unit, I-unit, I-unit
  - M-unit, L-unit, X-unit
  - M-unit, M-unit, I-unit
  - M-unit, F-unit, I-unit
  - M-unit, M-unit, F-unit
  - M-unit, I-unit, B-unit
  - M-unit, B-unit, B-unit
  - B-unit, B-unit, B-unit
  - M-unit, M-unit, B-unit
  - M-unit, F-unit, B-unit
IA-64 Instruction Format Example:
Type A (Integer ALU) Instruction Format

i.e. R-Type

Register-register format:

41 Bits

40  37  36  35  34  33  32  29  28  27  26  20  19  13  12  6  5  0

8  X2a  Ve  X4  X2b  R3  R2  R1  qp
4  1  2  1  4  2  7  7  7  6

- 8 is the major opcode for this instruction type.
- X2a, X2b, Ve, and X4 are opcode extensions.
- qp is the predicate register (or predication flag) assigned to this operation (64 such flags)

Predication: Any instruction can be cancelled (turned into a no-op) based on the value of one of 64 predication flags (qp)
Purpose: To reduce number of branches in code (larger basic blocks)
X(i) = X(i) + S

Unrolled Loop Example for Scalar (single-issue) Pipeline

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  F4,0 (R1)
10     S.D  F8,-8 (R1)
11     DADDUI  R1,R1,#-32
12     S.D  F12,16(R1)
13     BNE  R1,R2,LOOP
14     S.D  F16,8 (R1) ; 8-32 = -24

14 clock cycles, or 3.5 per original iteration (result) (unrolled four times)

Latency:
L.D to ADD.D: 1 Cycle
ADD.D to S.D: 2 Cycles

Unrolled (4-times) and scheduled loop from loop unrolling example in lecture # 3 (slide 11)

Recall that loop unrolling exposes more ILP by increasing size of resulting basic block

No stalls in code above: CPI = 1 (ignoring initial pipeline fill cycles)
Loop Unrolling in 2-way Superscalar Pipeline:
(1 Integer, 1 FP/Cycle)

<table>
<thead>
<tr>
<th>Integer instruction</th>
<th>FP instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>L.D F6,-8(R1)</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>3</td>
</tr>
<tr>
<td>L.D F14,-24(R1)</td>
<td>ADD.D F8,F6,F2</td>
<td>4</td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td>5</td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>ADD.D F16,F14,F2</td>
<td>6</td>
</tr>
<tr>
<td>S.D F8,-8(R1)</td>
<td>ADD.D F20,F18,F2</td>
<td>7</td>
</tr>
<tr>
<td>S.D F12,-16(R1)</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>DADDUI R1,R1,#-40</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>S.D F16,-24(R1)</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>BNE R1,R2,LOOP</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

- Unrolled 5 times to avoid delays and expose more ILP (unrolled one more time)
- 12 cycles, or 12/5 = 2.4 cycles per iteration (3.5/2.4 = 1.5X faster than scalar)
- CPI = 12/17 = .7 worse than ideal CPI = .5 because 7 issue slots are wasted

Recall that loop unrolling exposes more ILP by increasing basic block size

Scalar Processor = Single-issue Processor

CMPE550 - Shaaban
Empty or wasted issue slots can be defined as either vertical waste or horizontal waste:

- **Vertical waste** is introduced when the processor issues no instructions in a cycle.
- **Horizontal waste** occurs when not all issue slots can be filled in a cycle.

**Superscalar/VLIW Architecture Limitations:**

**Issue Slot Waste Classification**

- Example:
  - 4-Issue Superscalar
  - Ideal IPC = 4
  - Ideal CPI = 0.25

Also applies to VLIW

Result of issue slot waste: Actual Performance << Peak Performance
Loop Unrolling in VLIW Pipeline
(2 Memory, 2 FP, 1 Integer / Cycle)

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op/branch</th>
<th>Clock</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>
<td>1</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
<td>2</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>
<td>3</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>
<td>4</td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>S.D F8,-8(R1)</td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>S.D F12, -16(R1)</td>
<td>S.D F16,-24(R1)</td>
<td>ADD.D F28,F26,F2</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>S.D F20, 24(R1)</td>
<td>S.D F24,16(R1)</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>S.D F28, 8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>DADDUI R1,R1,#-56</td>
<td>8</td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays and expose more ILP
7 results in 9 cycles, or 1.3 cycles per iteration
(2.4/1.3 = 1.8X faster than 2-issue superscalar, 3.5/1.3 = 2.7X faster than scalar)
Average: about 23/9 = 2.55 IPC (instructions per clock cycle) Ideal IPC =5,
CPI = .39  Ideal CPI = .2  thus about 50% efficiency, 22 issue slots are wasted
Note: Needs more registers in VLIW (15 vs. 6 in Superscalar)

Scalar Processor = Single-Issue Processor

5-issue VLIW
Ideal CPI = 0.2
IPC = 5

Scalar Processor = Single-Issue Processor

CMPE550 - Shaaban
4th Edition: Chapter 2.7 pages 116-117
(3rd Edition: Chapter 4.3 pages 317-318)

9/7 = 1.3 cycles per original iteration
Superscalar Tomasulo-based Dynamic Scheduling

- The **Tomasulo** dynamic scheduling algorithm is extended to issue more than one instruction per cycle.
- However the restriction that instructions must issue in program order still holds to avoid violating instruction dependencies (construct correct dependency graph dynamically).
  
  - The result of issuing multiple instructions in one cycle should be the same as if they were single-issued, one instruction per cycle.

- How to issue two instructions and keep in-order instruction issue for Tomasulo?
  
  - **Simplest Method:** Restrict Type of Instructions Issued Per Cycle
  - To simplify the issue logic, issue one integer + one floating-point instruction per cycle (for a 2-way superscalar).
    
    - 1 Tomasulo control for integer, 1 for floating point.

  - FP loads/stores might cause a dependency between integer and FP issue:
    
    - Replace load reservation stations with a load queue; operands must be read in the order they are fetched (program order).
    - Replace store reservation stations with a store queue; operands must be written in the order they are fetched.

    - **Load checks addresses in Store Queue to avoid RAW violation**
      
      - (get load value from store queue if memory address matches)
    - **Store checks addresses in Load Queue to avoid WAR, and checks Store Queue to avoid WAW.**

    (the above load/store queue checking is also applicable to single-issue Tomasulo to take care of memory RAW, WAR, WAW). More on this later..
Superscalar Dynamic Scheduling

Three techniques can be used to support multiple instruction issue in Tomasulo without putting restrictions on the type of instructions issued per cycle:

1. **Issue at a higher clock rate so that issue remains in order.**
   - For example for a **2-Issue** superscalar issue at 2X Clock Rate.

   ![Diagram showing 2-Issue superscalar issue at 2X Clock Rate]

2. **Widen the issue logic to handle multiple instruction issue**
   - All possible dependencies between instructions to be issues are detected at once and the result of the multiple issue matches in-order issue.

   ![Diagram showing 4-Issue superscalar]

**Why?**

For correct dynamic construction of dependency graph:
The result of issuing multiple instructions in one cycle should be the same as if they were single-issued, one instruction per cycle.
Superscalar Dynamic Scheduling

To avoid increasing the CPU clock cycle time in the last two approaches, multiple instruction issue can be split into two pipelined issue stages:

- **Issue Stage One:** Decide how many instructions can issue simultaneously checking dependencies within the group of instructions to be issued + available RSs, ignoring instructions already issued.

- **Issue Stage Two:** Examine dependencies among the selected instructions from the group and the those already issued.

- This approach is usually used in dynamically-scheduled wide superscalars that can issue four or more instructions per cycle.

- Splitting the issue into two pipelined staged increases the CPU pipeline depth and increases branch penalties
  - This increases the importance of accurate dynamic branch prediction methods.

- Further pipelining of issue stages beyond two stages may be necessary as CPU clock rates are increased.

- The dynamic scheduling/issue control logic for superscalars is generally very complex growing at least quadratically with issue width.
  - e.g. 4 wide superscalar -> 4x4 = 16 times complexity of single issue CPU

Complexity

More control logic complexity/area/power
Multiple Instruction Issue with Dynamic Scheduling Example

Assumptions:

1. Restricted 2-way superscalar: 1 integer, 1 FP Issue Per Cycle

2. A sufficient number of reservation stations is available.

3. Total two integer units available:
   One integer unit (for ALU, effective address)
   One integer unit for branch condition

4. 2 CDBs

5. Execution cycles:
   Integer: 1 cycle
   Load: 2 cycles (1 ex + 1 mem)
   FP add: 3 cycles

6. Any instruction following a branch cannot start execution until after branch condition is evaluated in EX (resolved)

7. Branches are single issued, no delayed branch, perfect branch prediction

Example

Consider the execution of the following simple loop, which adds a scalar in F2 to each element of a vector in memory. Use a MIPS pipeline extended with Tomasulo’s algorithm and with multiple issue:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Integer</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F0,0(R1)</td>
<td>; F0 = array element</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4,F0,F2</td>
<td>; add scalar in F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F4,0(R1)</td>
<td>; store result</td>
</tr>
<tr>
<td>DADDIU</td>
<td>R1,R1,#-8</td>
<td>; decrement pointer</td>
</tr>
<tr>
<td>BNE</td>
<td>R1,R2,LOOP</td>
<td>; 8 bytes (per DW)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>; branch R1!=R2</td>
</tr>
</tbody>
</table>

Assume that both a floating-point and an integer operation can be issued on every clock cycle, even if they are dependent. Assume one integer functional unit is used for both ALU operations and effective address calculations and a separate pipelined FP functional unit for each operation type. Assume that Issue and Write Results take one cycle each and that there is dynamic branch-prediction hardware and a separate functional unit to evaluate branch conditions. As in most dynamically scheduled processors, the presence of the Write Results stage means that the effective instruction latencies will be one cycle longer than in a simple in-order pipeline. Thus, the number of cycles of latency between a source instruction and an instruction consuming the result is one cycle for integer ALU operations, two cycles for loads, and three cycles for FP add. Create a table showing when each instruction issues, begins execution, and writes its result to the CDB for the first three iterations of the loop. Assume two CDBs and assume that branches single issue (no delayed branches) but that branch prediction is perfect. Also show the resource usage for the integer unit, the floating-point unit, the data cache, and the two CDBs.
Multiple Instruction Issue with Dynamic Scheduling Example

**Answer**  The loop will be dynamically unwound, and, whenever possible, instructions will be issued in pairs. The execution timing is shown in Figure 3.25. Figure 3.26 shows the resource utilization. The loop will continue to fetch and issue a new loop iteration every three clock cycles, and sustaining one iteration every three cycles would lead to an IPC of $5/3 = 1.67$. The instruction execution rate, however, is lower: By looking at the Execute stage we can see that the sustained instruction completion rate is $15/16 = 0.94$. Assuming the branches are perfectly predicted, the issue unit will eventually fill all the reservation stations and will stall.

The throughput improvement versus a single-issue pipeline is small because there is only one floating-point operation per iteration and, thus, the integer pipeline becomes a bottleneck.
### Three Loop Iterations on Restricted 2-way Superscalar Tomasulo

#### FP EX = 3 cycles

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Instructions</th>
<th>In Order</th>
<th>(Start) Issues at</th>
<th>(Start) Executes</th>
<th>Memory access at</th>
<th>Write CDB at</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L.D F0,0(R1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>First issue</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ADD.D F4,F0,F2</td>
<td>1</td>
<td>5</td>
<td></td>
<td>8</td>
<td>Wait for L.D</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S.D F4,0(R1)</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td>5</td>
<td>Wait for ADD.D</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R1,R1,#-8</td>
<td>2</td>
<td>4</td>
<td></td>
<td>5</td>
<td>Wait for ALU</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BNE R1,R2,Loop</td>
<td>3</td>
<td>6</td>
<td></td>
<td>9</td>
<td>Wait for DADDIU</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L.D F0,0(R1)</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>13</td>
<td>Wait for ADD.D</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADD.D F4,F0,F2</td>
<td>4</td>
<td>10</td>
<td></td>
<td></td>
<td>Wait for L.D</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S.D F4,0(R1)</td>
<td>5</td>
<td>8</td>
<td>14</td>
<td>13</td>
<td>Wait for ADD.D</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R1,R1,#-8</td>
<td>5</td>
<td>9</td>
<td></td>
<td>10</td>
<td>Wait for ALU</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BNE R1,R2,Loop</td>
<td>6</td>
<td>11</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>L.D F0,0(R1)</td>
<td>7</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>Wait for BNE complete</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ADD.D F4,F0,F2</td>
<td>7</td>
<td>15</td>
<td></td>
<td>18</td>
<td>Wait for L.D</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S.D F4,0(R1)</td>
<td>8</td>
<td>13</td>
<td>19</td>
<td>15</td>
<td>Wait for ADD.D</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R1,R1,#-8</td>
<td>8</td>
<td>14</td>
<td></td>
<td></td>
<td>Wait for ALU</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BNE R1,R2,Loop</td>
<td>9</td>
<td>16</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.25** The clock cycle of issue, execution, and writing result for a dual-issue version of our Tomasulo pipeline. The Write Result stage does not apply to either stores or branches, since they do not write any registers. We assume a result is written to the CDB at the end of the clock cycle it is available in. This figure also assumes a wider CDB. For L.D and S.D, the execution is effective address calculation. For branches, the execute cycle shows when the branch condition can be evaluated and the prediction checked; we assume that this can happen as early as the cycle after issue, if the operands are available. Any instructions following a branch cannot start execution until after the branch condition has been evaluated. We assume one memory unit, one integer pipeline, and one FP adder. If two instructions could use the same functional unit at the same point, priority is given to the “older” instruction. Note that the load of the next iteration performs its memory access before the store of the current iteration.

---

**Only one CDB is actually needed in this case.**

19 cycles to complete three iterations

---

**FP ADD** has 3 execution cycles
Branches single issue

For instructions after a branch: Execution starts after branch is resolved
<table>
<thead>
<tr>
<th>Clock number</th>
<th>Integer ALU</th>
<th>FP ALU</th>
<th>Data cache</th>
<th>CDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1 / L.D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1 / S.D</td>
<td>1 / L.D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1 / DADDIU</td>
<td></td>
<td></td>
<td>1 / L.D</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>1 / ADD.D</td>
<td></td>
<td>1 / DADDIU</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2 / L.D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2 / S.D</td>
<td>2 / L.D</td>
<td>1 / ADD.D</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2 / DADDIU</td>
<td>1 / S.D</td>
<td>2 / L.D</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>2 / ADD.D</td>
<td></td>
<td>2 / DADDIU</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>3 / L.D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>3 / S.D</td>
<td>3 / L.D</td>
<td>2 / ADD.D</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>3 / DADDIU</td>
<td>2 / S.D</td>
<td>3 / L.D</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>3 / ADD.D</td>
<td></td>
<td>3 / DADDIU</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>3 / ADD.D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>3 / S.D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.26 Resource usage table for the example shown in Figure 3.25. The entry in each box shows the opcode and iteration number of the instruction that uses the functional unit heading the column at the clock cycle corresponding to the row. Only a single CDB is actually required and that is what we show.

Only one CDB is actually needed in this case.
Multiple Instruction Issue with Dynamic Scheduling Example

Assumptions:
The same loop in previous example
On restricted 2-way superscalar:
1 integer, 1 FP Issue Per Cycle
A sufficient number of reservation stations is available.
Total three integer units
one for ALU, one for effective address
One integer unit for branch condition
2 CDBs
Execution cycles:
Integer: 1 cycle
Load: 2 cycles (1 ex + 1 mem)
FP add: 3 cycles
Any instruction following a branch cannot start execution until after branch condition is evaluated
Branches are single issued, no delayed branch, perfect branch prediction

Example
Consider the execution of the same loop on a two-issue processor, but, in addition, assume that there are separate integer functional units for effective address calculation and for ALU operations. Create a table as in Figure 3.25 for the first three iterations of the same loop and another table to show the resource usage.

Figure 3.27 shows the improvement in performance: The loop executes in 5 clock cycles less (11 versus 16 execution cycles). The cost of this improvement is both a separate address adder and the logic to issue to it; note that, in contrast to the earlier example, a second CDB is needed. As Figure 3.28 shows this example has a higher instruction execution rate but lower efficiency as measured by the utilization of the functional units.

Three factors limit the performance (as shown in Figure 3.27) of the two-issue dynamically scheduled pipeline:

1. There is an imbalance between the functional unit structure of the pipeline and the example loop. This imbalance means that it is impossible to fully use the FP units. To remedy this, we would need fewer dependent integer operations per loop. The next point is a different way of looking at this limitation.

2. The amount of overhead per loop iteration is very high: two out of five instructions (the DADDIU and the BNE) are overhead. In the next chapter we look at how this overhead can be reduced.

3. The control hazard, which prevents us from starting the next L.D before we know whether the branch was correctly predicted, causes a one-cycle penalty on every loop iteration. The next section introduces a technique that addresses this limitation.

Previous example repeated with one more integer ALU (3 total)
Same three loop Iterations on Restricted 2-way Superscalar Tomasulo but with Three integer units (one for ALU, one for effective address calculation, one for branch condition)

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Instructions</th>
<th>In Order</th>
<th>(Start) Executes</th>
<th>Memory access at</th>
<th>Write CDB at</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L.D F0,0(R1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>First issue</td>
</tr>
<tr>
<td>1</td>
<td>ADD.D F4,F0,F2</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td></td>
<td>Wait for L.D</td>
</tr>
<tr>
<td>1</td>
<td>S.D F4,0(R1)</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td></td>
<td>Wait for ADD.D</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R1,R1,#-8</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td>Executes earlier</td>
</tr>
<tr>
<td>1</td>
<td>BNE R1,R2,Loop</td>
<td>3</td>
<td>5</td>
<td>12</td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>L.D F0,0(R1)</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>Wait for BNE complete</td>
</tr>
<tr>
<td>2</td>
<td>ADD.D F4,F0,F2</td>
<td>4</td>
<td>9</td>
<td>12</td>
<td></td>
<td>Wait for L.D</td>
</tr>
<tr>
<td>2</td>
<td>S.D F4,0(R1)</td>
<td>5</td>
<td>7</td>
<td>13</td>
<td></td>
<td>Wait for ADD.D</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R1,R1,#-8</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td>Executes earlier</td>
</tr>
<tr>
<td>2</td>
<td>BNE R1,R2,Loop</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>L.D F0,0(R1)</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>Wait for BNE complete</td>
</tr>
<tr>
<td>3</td>
<td>ADD.D F4,F0,F2</td>
<td>7</td>
<td>12</td>
<td>15</td>
<td></td>
<td>Wait for L.D</td>
</tr>
<tr>
<td>3</td>
<td>S.D F4,0(R1)</td>
<td>8</td>
<td>10</td>
<td>16</td>
<td></td>
<td>Wait for ADD.D</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R1,R1,#-8</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
<td>Executes earlier</td>
</tr>
<tr>
<td>3</td>
<td>BNE R1,R2,Loop</td>
<td>9</td>
<td>11</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
</tbody>
</table>

**Figure 3.27** The clock cycle of issue, execution, and writing result for a dual-issue version of our Tomasulo pipeline with separate functional units for integer ALU operations and effective address calculation, which also uses a wider CDB. The extra integer ALU allows the DADDIU to execute earlier, in turn allowing the BNE to execute earlier, and thereby starting the next iteration earlier.


For instructions after a branch: Execution starts after branch is resolved

Both CDBs are used here (in cycles 4, 8)

16 cycles here vs. 19 cycles (with two integer units)
## Resource Usage Table for Example:

<table>
<thead>
<tr>
<th>Clock number</th>
<th>Integer ALU</th>
<th>Address adder</th>
<th>FP ALU</th>
<th>Data cache</th>
<th>CDB #1</th>
<th>CDB #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>1 / L.D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1 / DADDIU</td>
<td>1 / S.D</td>
<td></td>
<td>1 / L.D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 / L.D</td>
<td>1 / DADDIU</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 / ADD.D</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2 / DADDIU</td>
<td>2 / L.D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>2 / S.D</td>
<td></td>
<td>2 / L.D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 / ADD.D</td>
<td>2 / L.D</td>
</tr>
<tr>
<td>9</td>
<td>3 / DADDIU</td>
<td>3 / L.D</td>
<td></td>
<td>2 / ADD.D</td>
<td>1 / S.D</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 / L.D</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 / ADD.D</td>
<td>2 / ADD.D</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>2 / S.D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 / ADD.D</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>3 / S.D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Figure 3.28** Resource usage table for the example shown in Figure 3.27, using the same format as Figure 3.26.

Both CDBs are used here (in cycles 4, 8)

---

*3rd Edition: page 225 (not in 4th Edition)*
Speculation (Speculative Execution)

- Compiler ILP techniques (loop-unrolling, software Pipelining etc.) are not effective to uncover maximum ILP when branch behavior is not well known at compile time.
- Full exploitation of the benefits of dynamic branch prediction and further reduction of the impact of branches on performance can be achieved by using speculation:

  - **Speculation:** An instruction is executed before the processor knows that the instruction should execute to avoid control dependence stalls (i.e. branch not resolved yet):

    - **Static Speculation** by the compiler with hardware support:
      - The compiler labels an instruction as speculative and the hardware helps by ignoring the outcome of incorrectly speculated instructions.
      - Conditional instructions provide limited speculation.

    - **Dynamic Hardware-based Speculation**:
      - Uses dynamic branch-prediction to guide the speculation process.
      - Dynamic scheduling and execution continued passed a conditional branch in the predicted branch direction.

Here we focus on hardware-based speculation using Tomasulo-based dynamic scheduling enhanced with speculation (**Speculative Tomasulo**).

- The resulting processors are usually referred to as **Speculative Processors**.

What if dynamic branch prediction is wrong?
Dynamic Hardware-Based Speculation

(Speculative Execution Processors, Speculative Tomasulo)

- Combines:
  1. Dynamic hardware-based branch prediction
  2. Dynamic Scheduling: issue multiple instructions in order and execute out of order. (Tomasulo)

- Continue to dynamically issue, and execute instructions passed a conditional branch in the dynamically predicted branch direction, before control dependencies are resolved.
  - This overcomes the ILP limitations of the basic block size.
  - Creates **dynamically speculated instructions** at run-time with no ISA/compiler support at all. i.e Dynamic speculative execution
  - If a branch turns out as mispredicted all such dynamically speculated instructions must be prevented from changing the state of the machine (registers, memory).
  - Precise exceptions are possible since instructions **must commit in order**. i.e instructions forced to complete (commit) in program order

**How?**

- Addition of commit (retire, completion, or re-ordering) stage and forcing instructions to commit in their order in the code (i.e to write results to registers or memory in program order).

**Why?**

- If a branch is mispredicted, the speculated instructions must be stopped and invalidated. i.e speculated instructions must be cancelled

---

4th Edition: Chapter 2.6, 2.8 (3rd Edition: Chapter 3.7)
Hardware-Based Speculation

Speculative Execution + Tomasulo’s Algorithm

\[ \text{Speculative Tomasulo} \]

Speculative Tomasulo-based Processor

Four Steps of Speculative Tomasulo Algorithm

1. **Issue** — (In-order) Get an instruction from Instruction Queue
   
   If a reservation station and a reorder buffer slot are free, issue instruction & send operands & reorder buffer number for destination (this stage is sometimes called “dispatch”)

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

3. **Write result** — (out-of-order) Finish execution (WB)
   
   Write on Common Data Bus (CDB) to all awaiting FUs & reorder buffer; mark reservation station available.

4. **Commit** — (In-order) Update registers, memory with reorder buffer result
   
   - When an instruction is at head of reorder buffer & the result is present, update register with result (or store to memory) and remove instruction from reorder buffer.
   
   - A mispredicted branch at the head of the reorder buffer flushes the reorder buffer (cancels speculated instructions after the branch)

   ⇒ Instructions issue in order, execute (EX), write result (WB) out of order, but must commit in order.
Hardware-Based Speculation Example

Example  Assume the same latencies for the floating-point functional units as in earlier examples: add is 2 clock cycles, multiply is 10 clock cycles, and divide is 40 clock cycles. Using the code segment below, the same one we used to generate Figure 3.4, show what the status tables look like when the MUL.D is ready to go to commit.

<table>
<thead>
<tr>
<th>Program Order</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F6,34(R2)</td>
</tr>
<tr>
<td>L.D</td>
<td>F2,45(R3)</td>
</tr>
<tr>
<td>MUL.D</td>
<td>F0,F2,F4</td>
</tr>
<tr>
<td>SUB.D</td>
<td>F8,F6,F2</td>
</tr>
<tr>
<td>DIV.D</td>
<td>F10,F0,F6</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F6,F8,F2</td>
</tr>
</tbody>
</table>

Show speculated single-issue Tomasulo status when MUL.D is ready to commit (commit done in program order)

Answer  The result is shown in the three tables in Figure 3.30. Notice that although the SUB.D instruction has completed execution, it does not commit until the MUL.D commits. The reservation stations and register status field contain the same basic information that they did for Tomasulo’s algorithm (see page 189 for a description of those fields). The differences are that reservation station numbers are replaced with ROB entry numbers in the Qj and Qk fields, as well as in the register status fields, and we have added the Dest field to the reservation stations. The Dest field designates the ROB number that is the destination for the result produced by this reservation station entry.

Single-issue speculative Tomasulo Example

Figure 3.30 At the time the MUL.D is ready to commit, only the two L.D instructions have committed, although several others have completed execution. The MUL.D is at the head of the ROB, and the two L.D instructions are there only to ease understanding. The SUB.D and ADD.D instructions will not commit until the MUL.D instruction commits, although the results of the instructions are available and can be used as sources for other instructions. The DIV.D is in execution, but has not completed solely due to its longer latency than MUL.D. The Value column indicates the value being held; the format #X is used to refer to a value field of ROB entry X. Reorder buffers 1 and 2 are actually completed, but are shown for informational purposes. We do not show the entries for the load-store queue, but these entries are kept in order.

Single-issue speculative Tomasulo Example
Hardware-Based Speculation Example

Example

Consider the code example used earlier for Tomasulo’s algorithm and shown in Figure 3.6 in execution:

<table>
<thead>
<tr>
<th>Loop:</th>
<th>L.D</th>
<th>F0,0(R1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MUL.D</td>
<td>F4,F0,F2</td>
</tr>
<tr>
<td></td>
<td>S.D</td>
<td>F4,0(R1)</td>
</tr>
<tr>
<td></td>
<td>DADDIU</td>
<td>R1,R1,#-8</td>
</tr>
<tr>
<td></td>
<td>BNE</td>
<td>R1,R2,Loop ;branches if R1≠R2</td>
</tr>
</tbody>
</table>

Next instruction to commit → First iteration instructions committed

Assume that we have issued all the instructions in the loop twice. Let’s also assume that the L.D and MUL.D from the first iteration have committed and all other instructions have completed execution. Normally, the store would wait in the ROB for both the effective address operand (R1 in this example) and the value (F4 in this example). Since we are only considering the floating-point pipeline, assume the effective address for the store is computed by the time the instruction is issued.

Answer

The result is shown in the two tables in Figure 3.31.

Because neither the register values nor any memory values are actually written until an instruction commits, the processor can easily undo its speculative actions when a branch is found to be mispredicted. Suppose that in the previous example (see Figure 3.31), the branch BNE is not taken the first time. The instructions prior to the branch will simply commit when each reaches the head of the ROB; when the branch reaches the head of that buffer, the buffer is simply cleared and the processor begins fetching instructions from the other path.

Single-issue speculative Tomasulo

L.D. and MUL.D of first iteration have committed, other instructions completed execution

<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 F0,0(R1)</td>
<td>Commit</td>
<td>F0</td>
<td>Mem[0 + Regs[R1]]</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>MUL.D F4,F0,F2</td>
<td>Commit</td>
<td>F4</td>
<td>#1×Regs[F2]</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>S.D F4,0(R1)</td>
<td>Write result</td>
<td>0 + Regs[R1]</td>
<td>#2</td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>DADDIU R1,R1,#-8</td>
<td>Write result</td>
<td>R1</td>
<td>Regs[R1]−8</td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td>BNE R1,R2,Loop</td>
<td>Write result</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>yes</td>
<td>L.D F0,0(R1)</td>
<td>Write result</td>
<td>F0</td>
<td>Mem[#4]</td>
</tr>
<tr>
<td>7</td>
<td>yes</td>
<td>MUL.D F4,F0,F2</td>
<td>Write result</td>
<td>F4</td>
<td>#6×Regs[F2]</td>
</tr>
<tr>
<td>8</td>
<td>yes</td>
<td>S.D F4,0(R1)</td>
<td>Write result</td>
<td>0+#4</td>
<td>#7</td>
</tr>
<tr>
<td>9</td>
<td>yes</td>
<td>DADDIU R1,R1,#-8</td>
<td>Write result</td>
<td>R1</td>
<td>#4−8</td>
</tr>
<tr>
<td>10</td>
<td>yes</td>
<td>BNE R1,R2,Loop</td>
<td>Write result</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What if branch was mispredicted?

<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>7</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></td>
</tr>
</tbody>
</table>

Figure 3.31 Only the L.D and MUL.D instructions have committed, although all the others have completed execution. Hence, no reservation stations are busy and none are shown. The remaining instructions will be committed as fast as possible. The first two reorder buffers are empty, but are shown for completeness.

Single-issue speculative Tomasulo
Multiple Issue with Speculation Example
(2-way superscalar with no restriction on issue instruction type)

Example

Consider the execution of the following loop, which searches an array, on a two-issue processor, once without speculation and once with speculation:

<table>
<thead>
<tr>
<th>Loop</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>R2,0(R1)</td>
<td>;R2=array element</td>
</tr>
<tr>
<td>DADDIU</td>
<td>R2,R2,#1</td>
<td>;increment R2</td>
</tr>
<tr>
<td>SD</td>
<td>R2,0(R1)</td>
<td>;store result</td>
</tr>
<tr>
<td>DADDIU</td>
<td>R1,R1,#4</td>
<td>;increment pointer</td>
</tr>
<tr>
<td>BNE</td>
<td>R2,R3,LOOP</td>
<td>;branch if not last element</td>
</tr>
</tbody>
</table>

Integer code
Ex = 1 cycle

A sufficient number of reservation stations and reorder (commit) buffer entries are available.
Branches still single issue

Assumptions:

A sufficient number of reservation stations and reorder (commit) buffer entries are available. + A sufficient number of Functional Units (FUs)/ALUs

Answer

Figures 3.33 and 3.34 show the performance for a two-issue dynamically scheduled processor, without and with speculation. In this case, where a branch is a key potential performance limitation, speculation helps significantly. The third branch in the speculative processor executes in clock cycle 13, while it executes in clock cycle 19 on the nonspeculative pipeline. Because the completion rate on the nonspeculative pipeline is falling behind the issue rate rapidly, the nonspeculative pipeline will stall when a few more iterations are issued. The performance of the nonspeculative processor could be improved by allowing load instructions to

**Answer: Without Speculation**

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Instructions</th>
<th>In Order</th>
<th>Issues at clock cycle number</th>
<th>Executes at clock cycle number</th>
<th>Data Memory access at clock cycle number</th>
<th>Write CDB at clock cycle number</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD</td>
<td>R2, 0(R1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>First issue</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU</td>
<td>R2, R2,#1</td>
<td>1</td>
<td>5</td>
<td></td>
<td>6</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>1</td>
<td>SD</td>
<td>R2, 0(R1)</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU</td>
<td>R1, R1,#4</td>
<td>2</td>
<td>3</td>
<td></td>
<td>4</td>
<td>Execute directly</td>
</tr>
<tr>
<td>1</td>
<td>BNE</td>
<td>R2, R3,LOOP</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>LD</td>
<td>R2, 0(R1)</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU</td>
<td>R2, R2,#1</td>
<td>4</td>
<td>11</td>
<td></td>
<td>12</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>2</td>
<td>SD</td>
<td>R2, 0(R1)</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU</td>
<td>R1, R1,#4</td>
<td>5</td>
<td>8</td>
<td></td>
<td>9</td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>2</td>
<td>BNE</td>
<td>R2, R3,LOOP</td>
<td>6</td>
<td>13</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>LD</td>
<td>R2, 0(R1)</td>
<td>7</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU</td>
<td>R2, R2,#1</td>
<td>7</td>
<td>17</td>
<td></td>
<td>18</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>3</td>
<td>SD</td>
<td>R2, 0(R1)</td>
<td>8</td>
<td>15</td>
<td>19</td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU</td>
<td>R1, R1,#4</td>
<td>8</td>
<td>14</td>
<td></td>
<td>15</td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>3</td>
<td>BNZ</td>
<td>R2, R3,LOOP</td>
<td>9</td>
<td>19</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
</tbody>
</table>

**Figure 3.33** The time of issue, execution, and writing result for a dual-issue version of our pipeline without speculation. Note that the LD following the BNE cannot start execution earlier, because it must wait until the branch outcome is determined. This type of program, with data-dependent branches that cannot be resolved earlier, shows the strength of speculation. Separate functional units for address calculation, ALU operations, and branch condition evaluation allow multiple instructions to execute in the same cycle.

For instructions after a branch: Execution starts after branch is resolved (No speculation)

CMPE550 - Shaaban

#34 lec # 6 Fall 2014 10-6-2014
Answer: 2-way Superscalar Tomasulo With Speculation

2-way Speculative Superscalar Processor: Issue and commit up to 2 instructions per cycle

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Instructions</th>
<th>Issues at clock number</th>
<th>Executes at clock number</th>
<th>Data Memory Read access at clock number</th>
<th>Write CDB at clock number</th>
<th>Commits at clock number</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD R2,0(R1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>First issue</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R2,R2,#1</td>
<td>1</td>
<td>5</td>
<td></td>
<td>6</td>
<td>7</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>1</td>
<td>SD R2,0(R1)</td>
<td>2</td>
<td>3</td>
<td></td>
<td>4</td>
<td>7</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R1,R1,#4</td>
<td>2</td>
<td>3</td>
<td></td>
<td>8</td>
<td>8</td>
<td>Commit in order</td>
</tr>
<tr>
<td>1</td>
<td>BNE R2,R3,LOOP</td>
<td>3</td>
<td>7</td>
<td></td>
<td>9</td>
<td>9</td>
<td>No execute delay</td>
</tr>
<tr>
<td>2</td>
<td>LD R2,0(R1)</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R2,R2,#1</td>
<td>4</td>
<td>8</td>
<td></td>
<td>9</td>
<td>9</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>SD R2,0(R1)</td>
<td>5</td>
<td>6</td>
<td></td>
<td>10</td>
<td>10</td>
<td>Commit in order</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R1,R1,#4</td>
<td>5</td>
<td>6</td>
<td></td>
<td>7</td>
<td>11</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>BNE R2,R3,LOOP</td>
<td>6</td>
<td>10</td>
<td></td>
<td>11</td>
<td>11</td>
<td>Commit in order</td>
</tr>
<tr>
<td>3</td>
<td>LD R2,0(R1)</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>Earliest possible</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R2,R2,#1</td>
<td>7</td>
<td>11</td>
<td></td>
<td>12</td>
<td>13</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>3</td>
<td>SD R2,0(R1)</td>
<td>8</td>
<td>9</td>
<td></td>
<td>13</td>
<td>13</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R1,R1,#4</td>
<td>8</td>
<td>9</td>
<td></td>
<td>10</td>
<td>14</td>
<td>Execute earlier</td>
</tr>
<tr>
<td>3</td>
<td>BNE R2,R3,LOOP</td>
<td>9</td>
<td>13</td>
<td></td>
<td>14</td>
<td>14</td>
<td>Wait for DADDIU</td>
</tr>
</tbody>
</table>

**Figure 3.34** The time of issue, execution, and writing result for a dual-issue version of our pipeline with speculation. Note that the `L.D` following the `BNE` can start execution early because it is speculative.

Branches Still Single Issue

Arrows show data dependencies

14 cycles here (with speculation) vs. 19 without speculation
Data Memory Access Dependency Checking/Handling In Dynamically Scheduled Processors

- Renaming in Tomasulo-based dynamically scheduled processors eliminates name dependence for register access but not for data memory access.
- Thus both true data dependencies and name dependencies must be detected to ensure correct ordering of data memory accesses for correct execution.

One possible solution:
- For Loads: Check store queue/buffers to ensure no data dependence violation (RAW hazard)
- For Stores:
  - Check store queue/buffers to ensure no output name dependence violation (WAW hazard)
  - Check load queue/buffers to ensure no anti-dependence violation (WAR hazard)

Related discussion in 4th edition page 102 (3rd edition page 195)

Note: Since instructions issue in program order, all pending load/store instructions in load/store queues are before the current load/store instruction in program order.
Data Access Name Dependency Examples:

**Anti-dependence Example:**

I    L.D. F6, 0(R1)

J    S.D. F4, 0(R1)

We have an address match here:
Instruction J (S.D.) must occur after I (i.e. write to memory by J must occur after read from memory by I) to prevent anti-dependence violation (WAR hazard).

**Output-dependence Example:**

I    S.D. F4, 0(R1)

J    S.D. F6, 0(R1)

We have an address match here:
Instruction J (Second S.D.) must occur last (i.e. write to memory last) to prevent output-dependence violation (WAW hazard).

**True Data Dependence Example:**

I    S.D. F4, 0(R1)

J    L.D. F6, 0(R1)

We have an address match here:
Instruction J (L.D.) gets the value of I (S.D) from store buffer to prevent data dependence violation (RAW hazard).

What about memory access dependency checking in speculative Tomasulo?
Limits to Multiple Instruction Issue Machines

• Inherent limitations of ILP:
  – If 1 branch exist for every 5 instruction: How to keep a 5-way superscalar/VLIW processor busy?
  – Latencies of unit adds complexity to the many operations that must be scheduled every cycle.
  – For maximum performance multiple instruction issue requires about:
    Pipeline Depth x No. Functional Units
    active instructions at any given cycle.  
    i.e to achieve 100% Functional unit utilization

• Hardware implementation complexities:
  – Duplicate FUs for parallel execution are needed, more CDBs.
  – More instruction bandwidth is essential.
  – Increased number of ports to Register File (datapath bandwidth):
    • VLIW example needs 7 read and 3 write for Int. Reg.
      & 5 read and 3 write for FP reg
  – Increased ports to memory (to improve memory bandwidth).
  – Superscalar issue/decoding complexity may impact pipeline clock rate, depth.
Empty or wasted issue slots can be defined as either vertical waste or horizontal waste:

- **Vertical waste** is introduced when the processor issues no instructions in a cycle.
- **Horizontal waste** occurs when not all issue slots can be filled in a cycle.

### Issue Slot Waste Classification

#### Example:

- **4-Issue Superscalar**
  - Ideal IPC = 4
  - Ideal CPI = 0.25

- **Instructions Per Cycle**: IPC = 1/CPI

**Result of issue slot waste:** Actual Performance << Peak Performance

- Full Stall?
- Partial Stall?
- How About 8-Issue?

- **Horizontal waste**: 9 slots
- **Vertical waste**: 12 slots
Sources of Unused Issue Cycles in an 8-issue Superscalar Processor.

Processor busy represents the utilized issue slots; all others represent wasted issue slots.

61% of the wasted cycles are vertical waste, the remainder are horizontal waste.

Workload: SPEC92 benchmark suite.

Superscalar Architecture Limitations:

All possible causes of wasted issue slots, and latency-hiding or latency reducing techniques that can reduce the number of cycles wasted by each cause.

<table>
<thead>
<tr>
<th>Source of Wasted Issue Slots</th>
<th>Possible Latency-Hiding or Latency-Reducing Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction tlb miss, data tlb miss</td>
<td>decrease the TLB miss rates (e.g., increase the TLB sizes); hardware instruction prefetching; hardware or software data prefetching; faster servicing of TLB misses</td>
</tr>
<tr>
<td>I cache miss</td>
<td>larger, more associative, or faster instruction cache hierarchy; hardware instruction prefetching</td>
</tr>
<tr>
<td>D cache miss</td>
<td>larger, more associative, or faster data cache hierarchy; hardware or software prefetching; improved instruction scheduling; more sophisticated dynamic execution</td>
</tr>
<tr>
<td>branch misprediction</td>
<td>improved branch prediction scheme; lower branch misprediction penalty</td>
</tr>
<tr>
<td>control hazard</td>
<td>speculative execution; more aggressive if-conversion</td>
</tr>
<tr>
<td>load delays (first-level cache hits)</td>
<td>shorter load latency; improved instruction scheduling; dynamic scheduling</td>
</tr>
<tr>
<td>short integer delay</td>
<td>improved instruction scheduling</td>
</tr>
<tr>
<td>long integer, short fp, long fp delays</td>
<td>(multiply is the only long integer operation, divide is the only long floating point operation) shorter latencies; improved instruction scheduling</td>
</tr>
<tr>
<td>memory conflict</td>
<td>(accesses to the same memory location in a single cycle) improved instruction scheduling</td>
</tr>
</tbody>
</table>

Main Issue: One Thread leads to limited ILP (cannot fill issue slots)

Solution: Exploit Thread Level Parallelism (TLP) within a single microprocessor chip:

How?

Simultaneous Multithreaded (SMT) Processor:
- The processor issues and executes instructions from a number of threads creating a number of logical processors within a single physical processor
  e.g. Intel’s HyperThreading (HT), each physical processor executes instructions from two threads

Chip-Multiprocessors (CMPs):
- Integrate two or more complete processor cores on the same chip (die)
- Each core runs a different thread (or program)
- Limited ILP is still a problem in each core
  (Solution: combine this approach with SMT)

Current Dual-Core Chip-Multiprocessor (CMP) Architectures

Single Die
Shared L2 Cache

Cores communicate using shared cache
(Lowest communication latency)

Examples:
IBM POWER4/5
Intel Pentium Core Duo (Yonah), Conroe
(Core 2), Sun UltraSparc T1 (Niagara)
AMD Barcelona (quad-core, 2nd half 2007)

Single Die
Private Caches
Shared System Interface

Cores communicate using on-chip
Interconnects (shared system interface)

Examples:
AMD Dual Core Opteron,
Athlon 64 X2
Intel Itanium2 (Montecito)

Two Dice – Shared Package
Private Caches
Private System Interface

Cores communicate over external
Front Side Bus (FSB)
(Highest communication latency)

Example:
Intel Pentium D, Quad cores

Source: Real World Technologies,
http://www.realworldtech.com/page.cfm?ArticleID=RWT101405234615
Example Six-Core Processor: AMD Phenom II X6

Six processor cores sharing 6 MB of level 3 (L3) cache
Eight processor cores sharing 24 MB of level 3 (L3) cache
Each core is 2-way SMT (2 threads per core), for a total of 16 threads

Example Eight-Core CMP: Intel Nehalem-EX

CMP = Chip-Multiprocessor
AKA Multi-Core Processor
Example 100-Core CMP: Tilera TILE-Gx Processor

No shared cache

Communication Links

On-Chip Network (Switch)

Network-on-Chip (NoC) Example

CMPE550 - Shaaban

For more information see: http://www.tilera.com/