CMPE550 Exam Review

4 questions out of 6 questions

(Must answer first 2 questions and 2 from remaining 4)

• Instruction Dependencies and graphs (quiz 3)
• In-order Floating Point/Multicycle Pipelining (quiz 2)
• Improving Instruction-Level Parallelism (ILP).
  – Loop-unrolling (quiz 3)
• Dynamic Pipeline Scheduling.
  – The Tomasulo Algorithm (quiz 4)
• Multiple Instruction Issue (CPI < 1): Superscalar vs. VLIW
• Dynamic Hardware-Based Speculation (quiz 5)
• Loop-Level Parallelism (LLP).
  – Making loop iterations parallel (quiz 6)
  – Software Pipelining (Symbolic Loop-Unrolling)
• Cache & Memory Performance. (quiz 7)
• I/O & System Performance. (quiz 8)
Data Hazard/Dependence Classification

**True Data Dependence**

$I$ (Write)

$J$ (Read)

Read after Write (RAW)
if data dependence is violated

$I$ (Write)

$J$ (Write)

Write after Write (WAW)
if output dependence is violated

**Shared Operand**

A name dependence:
output dependence

$I$ (Write)

$J$ (Read)

Write after Read (WAR)
if antidependence is violated

$I$ (Read)

$J$ (Write)

Program Order

No dependence

$I$ (Read)

$J$ (Read)

Read after Read (RAR) not a hazard

CMPE550 - Shaaban

#2 Exam Review Fall 2015 11-23-2015
Instruction Dependence Example

Can instruction 3 (first S.D) be moved just after instruction 4 (second L.D)?
How about moving 3 after 5 (the second ADD.D)? If not what dependencies are violated?

What happens if we rename F0 to F6 and F4 to F8 in instructions 4, 5, 6?

Example Code

1  L.D  F0, 0 (R1)
2  ADD.D  F4, F0, F2
3  S.D  F4, 0(R1)
4  L.D  F0, -8(R1)
5  ADD.D  F4, F0, F2
6  S.D  F4, -8(R1)

Date Dependence:
(1, 2)  (2, 3)  (4, 5)  (5, 6)

Output Dependence:
(1, 4)  (2, 5)

Anti-dependence:
(2, 4)  (3, 5)

Can instruction 4 (second L.D) be moved just after instruction 1 (first L.D)?
If not what dependencies are violated?
Control Dependencies

- Control dependence determines the **ordering of an instruction with respect to a branch (control) instruction.**
- Every instruction in a program except those in the very first basic block of the program is **control dependent on some set of branches.**

1. An instruction which is control dependent on a branch cannot be moved before the branch so that its execution is **no longer controlled by the branch.**

2. An instruction which is **not control dependent** on the branch **cannot be moved** so that its execution is **controlled by the branch** (in the then portion).
   → Both scenarios lead a control dependence violation (control hazard).

- It’s possible in some cases to violate these constraints and still have correct execution.
- Example of control dependence in the then part of an if statement:

```plaintext
if p1 {
    S1;  // S1 is control dependent on p1
    S1 is control dependent on p1
}

S2;  // S2 is control dependent on p2 but not on p1

If p2 {
    S2;  // What happens if S1 is moved here?
    S2;
}
```

*In Fourth Edition Chapter 2.1
(In Third Edition Chapter 3.1)*

**Control Dependence Violation = Control Hazard**
Floating Point/Multicycle Pipelining in MIPS

• Completion of MIPS EX stage floating point arithmetic operations in one or two cycles is impractical since it requires:
  • A much longer CPU clock cycle, and/or
  • An enormous amount of logic.

• Instead, the floating-point pipeline will allow for a longer latency (more EX cycles than 1).

• Floating-point operations have the same pipeline stages as the integer instructions with the following differences:
  – The EX cycle may be repeated as many times as needed (more than 1 cycle).
  – There may be multiple floating-point functional units.
  – A stall will occur if the instruction to be issued either causes a structural hazard for the functional unit or cause a data hazard.

• The latency of functional units is defined as the number of intervening cycles between an instruction producing the result and the instruction that uses the result (usually equals stall cycles with forwarding used).

• The initiation or repeat interval is the number of cycles that must elapse between issuing an instruction of a given type.
Extending The MIPS Pipeline: Multiple Outstanding Floating Point Operations

Latency = 0
Initiation Interval = 1
Pipelined

Latency = 3
Initiation Interval = 1
Pipelined

Latency = 6
Initiation Interval = 1
Pipelined

Hazards:
RAW, WAW possible
WAR Not Possible
Structural: Possible
Control: Possible

In-Order Single-Issue MIPS Pipeline with FP Support

(A pipeline that supports multiple outstanding FP operations.)

A pipelined CPU with pipelined FP units = Super-pipelined CPU

In Appendix A
FP Code RAW Hazard Stalls Example
(with full data forwarding in place)

FP Multiply Functional Unit has 7 EX cycles (and 6 cycle latency 6 = 7-1)
FP Add Functional Unit has 4 EX cycles (and 3 cycle latency 3 = 4-1)

L.D F4, 0(R2)
MUL.D F0, F4, F6
ADD.D F2, F0, F8
S.D F2, 0(R2)

Program Order

6 stall cycles which equals latency of FP multiply functional unit

When run on In-Order Single-Issue MIPS Pipeline with FP Support
With FP latencies/initiation intervals given above

Third stall due to structural hazard in MEM stage

11-23-2015
Increasing Instruction-Level Parallelism (ILP)

• A common way to increase parallelism among instructions is to exploit parallelism among iterations of a loop (i.e. Loop Level Parallelism, LLP).

• This is accomplished by unrolling the loop either statically by the compiler, or dynamically by hardware, which increases the size of the basic block present. This resulting larger basic block provides more instructions that can be scheduled or re-ordered by the compiler to eliminate more stall cycles.

• In this loop every iteration can overlap with any other iteration. Overlap within each iteration is minimal.

\[
\text{for (i=1; i<=1000; i=i+1;)}
\]
\[
x[i] = x[i] + y[i];
\]

• In vector machines, utilizing vector instructions is an important alternative to exploit loop-level parallelism,

• Vector instructions operate on a number of data items. The above loop would require just four such instructions.

4 vector instructions:

- Load Vector X
- Load Vector Y
- Add Vector X, X, Y
- Store Vector X

Independent (parallel) loop iterations: A result of high degree of data parallelism

In Fourth Edition Chapter 2.2 (In Third Edition Chapter 4.1)
MIPS Loop Unrolling Example

• For the loop:

for (i=1000; i>0; i=i-1)

\[
x[i] = x[i] + s;
\]

The straightforward MIPS assembly code is given by:

Loop:  L.D             F0, 0 (R1)           ;F0=array element
ADD.D        F4, F0, F2           ;add scalar in F2 (constant)
S.D               F4, 0(R1)            ;store result
DADDUI     R1, R1, # -8        ;decrement pointer 8 bytes
BNE             R1, R2,Loop      ;branch R1!=R2

R1 is initially the address of the element with highest address. 8(R2) is the address of the last element to operate on.

X[i] array of double-precision floating-point numbers (8-bytes each)

R1 initially points here
R2 points here

High Memory
Low Memory

First element to compute
Last element to compute

Program Order

Basic block size = 5 instructions

In Fourth Edition Chapter 2.2
(In Third Edition Chapter 4.1)

Initial value of R1 = R2 + 8000

CMPE550 - Shaaban

#9 Exam Review  Fall 2015  11-23-2015 (quiz 3)
MIPS FP Latency For Loop Unrolling Example

- All FP units assumed to be pipelined.
- The following FP operations latencies are used:

<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>

Other Assumptions:
- Branch resolved in decode stage, Branch penalty = 1 cycle
- Full forwarding is used
- Single Branch delay Slot
- Potential structural hazards ignored

In Fourth Edition Chapter 2.2 (In Third Edition Chapter 4.1)
Loop Unrolling Example (continued)

• This loop code is executed on the MIPS pipeline as follows:
  (Branch resolved in decode stage, Branch penalty = 1 cycle, Full forwarding is used)

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0, 0(R1)</td>
<td>1</td>
</tr>
<tr>
<td>stall</td>
<td>2</td>
</tr>
<tr>
<td>ADD.D F4, F0, F2</td>
<td>3</td>
</tr>
<tr>
<td>stall</td>
<td>4</td>
</tr>
<tr>
<td>stall</td>
<td>5</td>
</tr>
<tr>
<td>S.D F4, 0 (R1)</td>
<td>6</td>
</tr>
<tr>
<td>DADDUI R1, R1, # -8</td>
<td>7</td>
</tr>
<tr>
<td>stall</td>
<td>8</td>
</tr>
<tr>
<td>BNE R1,R2, Loop</td>
<td>9</td>
</tr>
<tr>
<td>stall</td>
<td>10</td>
</tr>
</tbody>
</table>

No scheduling
(Resulting stalls shown)

Scheduled with single delayed branch slot:
(Resulting stalls shown)

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0, 0(R1)</td>
<td>1</td>
</tr>
<tr>
<td>DADDUI R1, R1, # -8</td>
<td>2</td>
</tr>
<tr>
<td>ADD.D F4, F0, F2</td>
<td>3</td>
</tr>
<tr>
<td>stall</td>
<td>4</td>
</tr>
<tr>
<td>stall</td>
<td>5</td>
</tr>
<tr>
<td>BNE R1,R2, Loop</td>
<td>6</td>
</tr>
<tr>
<td>S.D F4,8(R1)</td>
<td>7</td>
</tr>
</tbody>
</table>

10 cycles per iteration

6 cycles per iteration

10/6 = 1.7 times faster

Due to resolving branch in ID

Program Order

CMPE550 - Shaaban

In Fourth Edition Chapter 2.2
(In Third Edition Chapter 4.1)

• Ignoring Pipeline Fill Cycles
• No Structural Hazards

#11 Exam Review Fall 2015 11-23-2015
Loop Unrolling Example (continued)

- The resulting loop code when four copies of the loop body are unrolled without reuse of registers.
- The size of the basic block increased from 5 instructions in the original loop to 14 instructions.

Three branches and three decrements of R1 are eliminated.

Load and store addresses are changed to allow DADDUI instructions to be merged.

The unrolled loop runs in 28 cycles assuming each L.D has 1 stall cycle, each ADD.D has 2 stall cycles, the DADDUI 1 stall, the branch 1 stall cycle, or 28/4 = 7 cycles to produce each of the four elements.

Load and store addresses are changed to allow DADDUI instructions to be merged.

The unrolled loop runs in 28 cycles assuming each L.D has 1 stall cycle, each ADD.D has 2 stall cycles, the DADDUI 1 stall, the branch 1 stall cycle, or 28/4 = 7 cycles to produce each of the four elements.

No scheduling

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Iteration</th>
<th>Instruction</th>
<th>Resulting stalls shown</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L.D F0, 0(R1)</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADD.D F4, F0, F2</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SD F4,0 (R1) ; drop DADDUI &amp; BNE</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LD F6, -8(R1)</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ADDD F8, F6, F2</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>SD F8, -8 (R1), ; drop DADDUI &amp; BNE</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>LD F10, -16(R1)</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ADDD F12, F10, F2</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>SD F12, -16 (R1) ; drop DADDUI &amp; BNE</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>LD F14, -24 (R1)</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>ADDD F16, F14, F2</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>SD F16, -24(R1)</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>DADDUI R1, R1, # -32</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>BNE R1, R2, Loop</td>
<td>Stall</td>
<td></td>
</tr>
</tbody>
</table>

Note use of different registers for each iteration (register renaming)

i.e. unrolled four times

In Fourth Edition Chapter 2.2 (In Third Edition Chapter 4.1)
Loop Unrolling Example (continued)

When scheduled for pipeline

Loop:

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

The execution time of the loop has dropped to 14 cycles, or $14/4 = 3.5$ clock cycles per element compared to 7 before scheduling and 6 when scheduled but unrolled.

Speedup = $6/3.5 = 1.7$

Unrolling the loop exposed more computations that can be scheduled to minimize stalls by increasing the size of the basic block from 5 instructions in the original loop to 14 instructions in the unrolled loop.

Note: No stalls

In Fourth Edition Chapter 2.2 (In Third Edition Chapter 4.1)
Dynamic Pipeline Scheduling

Dynamic instruction scheduling is accomplished by:

- Dividing the Instruction Decode ID stage into two stages:
  - **Issue**: Decode instructions, check for **structural hazards**.
    - A record of data dependencies is constructed as instructions are issued.
    - This creates a dynamically-constructed dependency graph for the window of instructions in-flight (being processed) in the CPU.
  - **Read operands**: Wait until data hazard conditions, if any, are resolved, then read operands when available (then start execution).
    (All instructions pass through the **issue stage in order** but can be stalled or pass each other in the read operands stage).
    - In the instruction fetch stage IF, fetch an additional instruction every cycle into a latch or several instructions into an instruction queue.
    - Increase the number of functional units to meet the demands of the additional instructions in their EX stage.

- Two approaches to dynamic scheduling:
  1. Dynamic scheduling with the **Scoreboard** used first in CDC6600 (1963)
  2. The **Tomasulo approach** pioneered by the IBM 360/91 (1966)

---

Fourth Edition: Appendix A.7, Chapter 2.4
(Third Edition: Appendix A.8, Chapter 3.2)

CDC660 is the world’s first “Supercomputer” Cost: $7 million in 1963

CMPE550 - Shaaban

Always done in program order

Can be done out of program order
Tomasulo Algorithm Vs. Scoreboard

- Control & buffers *distributed* with Functional Units (FUs) Vs. centralized in Scoreboard:
  - FU buffers are called "reservation stations" which have pending instructions and operands and other instruction status info (including data dependencies).
  - Reservations stations are sometimes referred to as "physical registers" or "renaming registers" as opposed to architecture registers specified by the ISA.
- ISA Registers in instructions are replaced by either values (if available) or pointers (renamed) to reservation stations (RS) that will supply the value later:
  - This process is called *register renaming*.
    - Register renaming eliminates WAR, WAW hazards (name dependence).
  - Allows for a *hardware-based* version of loop unrolling.
  - More reservation stations than ISA registers are possible, leading to optimizations that compilers can’t achieve and prevents the number of ISA registers from becoming a bottleneck.
- Instruction results go (forwarded) from RSs to RSs, *not through registers*, over *Common Data Bus (CDB)* that broadcasts results to all waiting RSs (dependant instructions).
- Loads and Stores are treated as FUs with RSs as well.

In Fourth Edition: Chapter 2.4 (In Third Edition: Chapter 3.2)
Dynamic Scheduling: The Tomasulo Approach

The basic structure of a MIPS floating-point unit using Tomasulo’s algorithm

Pipelined FP units are used here

In Fourth Edition: Chapter 2.4
(In Third Edition: Chapter 3.2)
Reservation Station (RS) Fields

- **Op**: Operation to perform in the unit (e.g., + or –)
- **Vj, Vk**: Value of Source operands S1 and S2
  - Store buffers have a single V field indicating result to be stored.
- **Qj, Qk**: Reservation stations producing source registers. (value to be written).
  - No ready flags as in Scoreboard; Qj,Qk=0 => ready.
  - Store buffers only have Qi for RS producing result.
- **A**: Address information for loads or stores. Initially immediate field of instruction then effective address when calculated.
- **Busy**: Indicates reservation station is busy.
- **Register result status**: Qi Indicates which Reservation Station will write each register, if one exists.
  - Blank (or 0) when no pending instruction (i.e. RS) exist that will write to that register.

Register bank behaves like a reservation station

*In Fourth Edition: Chapter 2.4 (In Third Edition: Chapter 3.2)*
Three Stages of Tomasulo Algorithm

1. **Issue**: Get instruction from pending Instruction Queue (IQ).
   - Instruction issued to a free reservation station (RS) (no structural hazard).
   - Selected RS is marked busy.
   - Control sends available instruction operands values (from ISA registers) to assigned RS.
   - Operands not available yet are renamed to RSs that will produce the operand (register renaming). (Dynamic construction of data dependency graph)

2. **Execution (EX)**: Operate on operands.
   - When both operands are ready then start executing on assigned FU.
   - If all operands are not ready, watch Common Data Bus (CDB) for needed result (forwarding done via CDB). (i.e. wait on any remaining operands, no RAW)

3. **Write result (WB)**: Finish execution.
   - Write result on Common Data Bus (CDB) to all awaiting units (RSs)
   - Mark reservation station as available.

   - **Normal data bus**: data + destination ("go to" bus).
   - **Common Data Bus (CDB)**: data + source ("come from" bus):
     - 64 bits for data + 4 bits for Functional Unit source address.
     - Write data to waiting RS if source matches expected RS (that produces result).
     - Does the result forwarding via broadcast to waiting RSs.

---

Stage 0 Instruction Fetch (IF): No changes, in-order

Always done in program order

In Fourth Edition: Chapter 2.4
(In Third Edition: Chapter 3.2)

Including destination register

CMPE550 - Shaaban

Note: No WB for stores

#18 Exam Review Fall 2015 11-23-2015
Tomasulo Approach Example

Using the same code used in the scoreboard example to be run on the Tomasulo configuration given earlier:

### Tomasulo Functional Units

- **L.D** F6, 34(R2)
- **L.D** F2, 45(R3)
- **MUL. D** F0, F2, F4
- **SUB. D** F8, F6, F2
- **DIV. D** F10, F0, F6
- **ADD. D** F6, F8, F2

### Pipelined Functional Units

- **Real Data Dependence (RAW)**
- **Anti-dependence (WAR)**
- **Output Dependence (WAW)**

L.D processing takes two cycles: EX, MEM (only one cycle in scoreboard example)

### RSs and EX Cycles

<table>
<thead>
<tr>
<th>Type</th>
<th># of RSs</th>
<th>EX Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Floating Point Multiply/divide</td>
<td>2</td>
<td>10/40</td>
</tr>
<tr>
<td>Floating Point add</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

In Fourth Edition: Chapter 2.5 (In Third Edition: Chapter 3.3)
Tomasulo Example: Cycle 57

<table>
<thead>
<tr>
<th>Instruction status</th>
<th>Execution</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>j</td>
<td>k</td>
</tr>
<tr>
<td>L.D F6 34+ R2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>L.D F2 45+ R3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>MUL.D F0 F2 F4</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>SUB.D F8 F6 F2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>DIV.D F10 F0 F6</td>
<td>5</td>
<td>56</td>
</tr>
<tr>
<td>ADD.D F6 F8 F2</td>
<td>6</td>
<td>10</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>0</td>
<td>Add1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</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>0</td>
<td>Mult1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</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>57</td>
<td>FU</td>
<td>M*F4</td>
<td>M(45+R3)</td>
<td>(M–M)+M()</td>
<td>M()–M()</td>
<td>M*F4/M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- We have:
  - In-order issue,
  - Out-of-order execution, completion

(quiz 4)
Tomasulo Loop Example
(Hardware-Based Version of Loop-Unrolling)

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

- Assume FP Multiply takes 4 execution clock cycles.
- Assume first load takes 8 cycles (possibly due to a cache miss), second load takes 4 cycles (cache hit).
- Assume R1 = 80 initially.
- Assume DADDUI only takes one cycle (issue)
- Assume branch resolved in issue stage (no EX or CDB write)
- Assume branch is predicted taken and no branch misprediction.
- No branch delay slot is used in this example.
- Stores take 4 cycles (ex, mem) and do not write on CDB
- We’ll go over the execution to complete first two loop iterations.

Expanded from loop example in Chapter 2.5 (Third Edition Chapter 3.3)
Loop Example Cycle 20

(First two Loop iterations done)

<table>
<thead>
<tr>
<th>Instruction status</th>
<th>Execution</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>j</td>
<td>k</td>
</tr>
<tr>
<td>L.D</td>
<td>F0</td>
<td>0</td>
</tr>
<tr>
<td>MUL.D</td>
<td>F4</td>
<td>F0</td>
</tr>
<tr>
<td>S.D</td>
<td>F4</td>
<td>0</td>
</tr>
<tr>
<td>L.D</td>
<td>F0</td>
<td>0</td>
</tr>
<tr>
<td>MUL.D</td>
<td>F4</td>
<td>F0</td>
</tr>
<tr>
<td>S.D</td>
<td>F4</td>
<td>0</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>
<th>Code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Add1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L.D</td>
</tr>
<tr>
<td>0</td>
<td>Add2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MUL.D</td>
</tr>
<tr>
<td>0</td>
<td>Add3</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S.D</td>
</tr>
<tr>
<td>1</td>
<td>Mult1</td>
<td>Yes</td>
<td>MULTD</td>
<td>M(64)</td>
<td>R(F2)</td>
<td></td>
<td></td>
<td>DADDUI</td>
</tr>
<tr>
<td>0</td>
<td>Mult2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BNE</td>
</tr>
</tbody>
</table>

Second S.D done (No write on CDB for stores) Second loop iteration done Issue fourth iteration L.D (to RS Load1)
## Tomasulo Loop Example Timing Diagram

### Table: Cycle

| Iteration | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|-----------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| MUL.D     | I |   |   |   |   |   |   |   |   |    | E | E | E | E | W |   |   |   |   |   |   |   |   |
| S.D.      | I |   |   |   |   |   |   |   |   |    |   | E | E | E | E |   |   |   |   |   |   |   |   |
| DADDUI    | I |   |   |   |   |   |   |   |   |    |   |   |   |   |   |   |   |   |   |   |   |   |   |
| BNE       | I |   |   |   |   |   |   |   |   |    |   |   |   |   |   |   |   |   |   |   |   |   |   |

### Diagram:

- **Iteration 1:**
  - L.D. issued at cycle 1.
  - MUL.D delayed until cycle 3 when mul RS is available.
  - 3rd L.D write delayed one cycle.

- **Iteration 2:**
  - L.D. issued at cycle 2.
  - MUL.D issued at cycle 3.
  - 3rd MUL.D issue delayed until mul RS is available.
  - 3rd L.D write delayed one cycle.

- **Iteration 3:**
  - L.D. issued at cycle 4.

- **Iteration 4:**
  - L.D. issued at cycle 5.

---

I = Issue  E = Execute  W = Write Result on CDB
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...
  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:

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

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

Unrolled and scheduled loop from loop unrolling example

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

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

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

#26   Exam Review   Fall 2015  11-23-2015
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></td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></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 F4,0(R1)</td>
<td>S.D F8,-8(R1)</td>
<td>ADD.D F28,F26,F2</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>S.D F12,-16(R1)</td>
<td>S.D F16,-24(R1)</td>
<td></td>
<td></td>
<td>DADDUI R1,R1,#-56</td>
<td>7</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>
<td>8</td>
</tr>
<tr>
<td>S.D F28,8(R1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</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)
Multiple Instruction Issue with Dynamic Scheduling Example

Assumptions:

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

A sufficient number of reservation stations is available.

Total two integer units available:
- One integer unit (for ALU, 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 in EX (resolved)

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:

```
Loop: L.D F0,0(R1); F0=array element
ADD.D F4,F0,F2; add scalar in F2
S.D F4,0(R1); store result
DADDIU R1,R1,#-8; decrement pointer
BNE R1,R2,LOOP; 8 bytes (per DW) branch R1!=R2
```

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.

Three Loop Iterations on Restricted 2-way Superscalar Tomasulo

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Instructions</th>
<th>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>
</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>
</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>DADDUI R1,R1,#-8</td>
<td>2</td>
<td>4</td>
<td></td>
<td>5</td>
<td>Wait for ALU</td>
</tr>
<tr>
<td>1</td>
<td>BNE R1,R2,Loop</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td>Wait for DADDUI</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 BNE complete</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>
</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>
</tr>
<tr>
<td>2</td>
<td>DADDUI R1,R1,#-8</td>
<td>5</td>
<td>9</td>
<td></td>
<td>10</td>
<td>Wait for ALU</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 DADDUI</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>
</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>
</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>
</tr>
<tr>
<td>3</td>
<td>DADDUI R1,R1,#-8</td>
<td>8</td>
<td>14</td>
<td></td>
<td></td>
<td>Wait for ALU</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 DADDUI</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 exeexecute 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.

FP ADD has 3 execution cycles
Branches single issue

Only one CDB is actually needed in this case.

19 cycles to complete three iterations
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

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.
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>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>
</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>
</tr>
<tr>
<td>1</td>
<td>S.D F4,0(R1)</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td>4</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></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></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></td>
<td>12</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></td>
<td>7</td>
<td>Executes earlier</td>
</tr>
<tr>
<td>3</td>
<td>BNE R1,R2,Loop</td>
<td>6</td>
<td>8</td>
<td></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></td>
<td>15</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></td>
<td>10</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.

16 cycles here vs. 19 cycles (with two integer units)

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

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

Dynamic Hardware-Based Speculation

(Speculative Execution Processors, Speculative Tomasulo)

- **Combines:**
  - Dynamic hardware-based branch prediction
  - 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.
  - If a branch turns out as mispredicted all such dynamically speculated instructions must be prevented from changing the state of the machine (registers, memory).

**Why?**
- **i.e. before branch is resolved**
- **i.e. Dynamic speculative execution**
- **i.e speculated instructions must be cancelled**

**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).**
- **Precise exceptions are possible since instructions must commit in order.**
  - **i.e instructions forced to complete (commit) in program order**

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

Speculative Execution + Tomasulo’s Algorithm

= 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.
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:

Loop:  
LD R2,0(R1) ; R2 = array element  
DADDIU R2,R2,#1 ; increment R2  
SD R2,0(R1) ; store result  
DADDIU R1,R1,#4 ; increment pointer  
BNE R2,R3,LOOP ; branch if not last element

Integer code
Ex = 1 cycle

Assumptions:
- i.e issue up to 2 instructions and commit up to 2 instructions per cycle

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table as in Figure 3.27 for the first three iterations of this loop for both machines. Assume that up to two instructions of any type can commit per clock.

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

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

Branches still single issue

**Answer: Without Speculation**

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Instructions</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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 **L.D** 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.

---

**Branches Still Single Issue**

19 cycles to complete three iterations.
**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>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>8</td>
<td>Commit in order</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R1,R1,#4</td>
<td>2</td>
<td>3</td>
<td></td>
<td>4</td>
<td>8</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>1</td>
<td>BNE R2,R3,LOOP</td>
<td>3</td>
<td>7</td>
<td></td>
<td>8</td>
<td>8</td>
<td></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>9</td>
<td>No execute delay</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R2.R2,#1</td>
<td>4</td>
<td>8</td>
<td></td>
<td>9</td>
<td>10</td>
<td>Wait for LW</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>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R1,R1,#4</td>
<td>5</td>
<td>6</td>
<td></td>
<td>11</td>
<td>11</td>
<td>Commit in order</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>Wait for DADDIU</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>14</td>
<td>14</td>
<td>Executes 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**

14 cycles here (with speculation) vs. 19 without speculation

Arrows show data dependencies
Loop-Level Parallelism (LLP) Analysis

- Loop-Level Parallelism (LLP) analysis focuses on whether data accesses in later iterations of a loop are data dependent on data values produced in earlier iterations and possibly making loop iterations independent (parallel).

  e.g. in
  for (i=1; i<=1000; i++)
  
  the computation in each iteration is independent of the previous iterations and the loop is thus parallel. The use of \( X[i] \) twice is within a single iteration.
  \( \Rightarrow \) Thus loop iterations are parallel (or independent from each other).

Classification of Data Dependencies in Loops:

- **Loop-carried Data Dependence**: A data dependence between different loop iterations (data produced in an earlier iteration used in a later one).
- **Not Loop-carried Data Dependence**: Data dependence within the same loop iteration.
- LLP analysis is important in software optimizations such as loop unrolling since it usually requires loop iterations to be independent (and in vector processing).
- LLP analysis is normally done at the source code level or close to it since assembly language and target machine code generation introduces loop-carried name dependence in the registers used in the loop.
  - Instruction level parallelism (ILP) analysis, on the other hand, is usually done when instructions are generated by the compiler.

LLP Analysis Example 1

- In the loop:

```c
for (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i];  /* S1 */
    B[i+1] = B[i] + A[i+1];} /* S2 */
```

(Where A, B, C are distinct non-overlapping arrays)

- **S2** uses the value A[i+1], computed by **S1** in the same iteration. This data dependence is within the same iteration (not a loop-carried dependence). (i.e. S1 → S2 on A[i+1]) Not loop-carried dependence

  ⇒ does not prevent loop iteration parallelism.

- **S1** uses a value computed by **S1** in the earlier iteration, since iteration i computes A[i+1] read in iteration i+1 (loop-carried dependence, prevents parallelism). The same applies for **S2** for B[i] and B[i+1].

  (i.e. S1 → S1 on A[i]  Loop-carried dependence
   S2 → S2 on B[i]  Loop-carried dependence)

  ⇒ These two data dependencies are loop-carried spanning more than one iteration (two iterations) preventing loop parallelism.

In this example the loop carried dependencies form two dependency chains starting from the very first iteration and ending at the last iteration.

![Dependency Graph](image_url)
LLP Analysis Example 2

- In the loop:

```c
for (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i]; /* S1 */
    B[i+1] = C[i] + D[i]; /* S2 */
}
```

- **S1** uses the value **B[i]** computed by **S2** in the previous iteration (loop-carried dependence)
- This dependence is not circular:
  - **S1** depends on **S2** but **S2** does not depend on **S1**.
  - Can be made parallel by replacing the code with the following:

```c
for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
```

```c
```

```c
B[101] = C[100] + D[100];  /* Loop Completion code */
```

---

**Dependency Graph**

**Iteration #** → **i** → **i+1**

- **S1**
- **S2**

**Loop-carried Dependence**

- S2 → S1 on B[i]

---

**4th Edition: Appendix G.2 (3rd Edition: Chapter 4.4) (quiz 6)**
LLP Analysis Example 2

Original Loop:

```
for (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i]; /* S1 */
    B[i+1] = C[i] + D[i]; /* S2 */
}
```

Modified Parallel Loop:

```

for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

Not Loop Carried Dependence

Loop Completion code

(one less iteration)
ILP Compiler Support: Software Pipelining (Symbolic Loop Unrolling)

– A compiler technique where loops are reorganized:
  
  • Each new iteration is made from instructions selected from a number of independent iterations of the original loop.

  i.e parallel iterations

– The instructions are selected to separate dependent instructions within the original loop iteration.

– No actual loop-unrolling is performed.
  
  • A software equivalent to the Tomasulo approach?

– Requires:
  
  • Additional start-up code to execute code left out from the first original loop iterations.

  • Additional finish code to execute instructions left out from the last original loop iterations.

This static optimization is done at machine code level

4th Edition: Appendix G.3 (3rd Edition: Chapter 4.4)
Software Pipelining (Symbolic Loop Unrolling) Example

Show a software-pipelined version of the code:

Before: Unrolled 3 times

1. L.D  F0, 0 (R1)
2. ADD.D F4, F0, F2
3. S.D  F4, 0 (R1)
4. L.D  F0, -8 (R1)
5. ADD.D F4, F0, F2
6. S.D  F4, -8 (R1)
7. L.D  F0, -16 (R1)
8. ADD.D F4, F0, F2
9. S.D  F4, -16 (R1)
10. DADDUI R1, R1, # -24
11. BNE  R1, R2, LOOP

After: Software Pipelined Version

1. L.D  F0, 0 (R1)
2. ADD.D F4, F0, F2
3. S.D  F4, 0 (R1)
4. ADD.D F4, F0, F2
5. BNE  R1, R2, LOOP

No actual loop unrolling is done (do not rename registers)

3 times because chain of dependence of length 3 instructions exist in body of original loop i.e. L.D → ADD.D → S.D

2 fewer loop iterations
Software Pipelining Example Illustrated

Assuming 6 original iterations (for illustration purposes):

\[
\begin{align*}
&L.D \quad F_0,0 \ (R1) \\
&ADD.D \quad F_4,F_0,F_2 \\
&S.D \quad F_4,0 \ (R1)
\end{align*}
\]

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
L.D & L.D & L.D & L.D & L.D & L.D \\
ADD.D & ADD.D & ADD.D & ADD.D & ADD.D & ADD.D \\
\end{array}
\]

4 Software Pipelined loop iterations (2 fewer iterations)

Loop Body of software Pipelined Version
Basic Cache Concepts

- Cache is the first level of the memory hierarchy once the address leaves the CPU and is searched first for the requested data.

- If the data requested by the CPU is present in the cache, it is retrieved from cache and the data access is a cache hit otherwise a cache miss and data must be read from main memory.

- On a cache miss a block of data must be brought in from main memory to cache to possibly replace an existing cache block.

- The allowed block addresses where blocks can be mapped (placed) into cache from main memory is determined by cache placement strategy.

- Locating a block of data in cache is handled by cache block identification mechanism: Tag matching.

- On a cache miss choosing the cache block being removed (replaced) is handled by the block replacement strategy in place.

- When a write to cache is requested, a number of main memory update strategies exist as part of the cache write policy.

(Review from 350)
Memory Hierarchy Performance:
Average Memory Access Time (AMAT), Memory Stall cycles

- **The Average Memory Access Time (AMAT):** The number of cycles required to complete an average memory access request by the CPU.
- **Memory stall cycles per memory access:** The number of stall cycles added to CPU execution cycles for one memory access.

- Memory stall cycles per average memory access = (AMAT - 1)
- For ideal memory: AMAT = 1 cycle, this results in zero memory stall cycles.
- Memory stall cycles per average instruction =
  \[
  \text{Number of memory accesses per instruction} \times \text{Memory stall cycles per average memory access}
  \]
  \[
  = (1 + \text{fraction of loads/stores}) \times (\text{AMAT} - 1)
  \]

Base CPI = \( \text{CPI}_{\text{execution}} \) = CPI with ideal memory

\[
\text{CPI} = \text{CPI}_{\text{execution}} + \text{Mem Stall cycles per instruction}
\]

cycles = CPU cycles

(Review from 350)
Cache Performance:
Single Level L1 Princeton (Unified) Memory Architecture

CPUtime = Instruction count x CPI x Clock cycle time

CPI_{execution} = CPI with ideal memory

CPI = CPI_{execution} + Mem Stall cycles per instruction

Mem Stall cycles per instruction =
    Memory accesses per instruction x Memory stall cycles per access

Assuming no stall cycles on a cache hit (cache access time = 1 cycle, stall = 0)

Cache Hit Rate = H1      Miss Rate = 1 - H1

Memory stall cycles per memory access = Miss rate x Miss penalty = (1 - H1) x M

AMAT = 1 + Miss rate x Miss penalty = 1 + (1 - H1) x M

Memory accesses per instruction = (1 + fraction of loads/stores)

Miss Penalty = M = the number of stall cycles resulting from missing in cache
    = Main memory access time - 1

Thus for a unified L1 cache with no stalls on a cache hit:

CPI = CPI_{execution} + (1 + fraction of loads/stores) x (1 - H1) x M

AMAT = 1 + (1 - H1) x M
Memory Access Tree:
For Unified Level 1 Cache

CPU Memory Access

L1 Hit:
% = Hit Rate = H1
Hit Access Time = 1
Stall cycles per access = 0
Stall = H1 x 0 = 0
(No Stall)

L1 Miss:
% = (1- Hit rate) = (1-H1)
Access time = M + 1
Stall cycles per access = M
Stall = M x (1-H1)

AMAT = H1 x 1 + (1-H1) x (M+1) = 1 + M x (1-H1)

Stall Cycles Per Access = AMAT - 1 = M x (1-H1)
CPI = CPI_{execution} + (1 + fraction of loads/stores) x M x (1-H1)

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
H1 = Level 1 Hit Rate
1 - H1 = Level 1 Miss Rate

(Review from 350)
Cache Performance:

Single Level L1 Harvard (Split) Memory Architecture

For a CPU with separate or split level one (L1) caches for instructions and data (Harvard memory architecture) and no stalls for cache hits:

\[
\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}
\]

\[
\text{CPI} = \text{CPI}_{\text{execution}} + \text{Mem Stall cycles per instruction}
\]

\[
\text{Mem Stall cycles per instruction} = \text{Instruction Fetch Miss rate} \times M + \text{Data Memory Accesses Per Instruction} \times \text{Data Miss Rate} \times M
\]

\[
M = \text{Miss Penalty} = \text{stall cycles per access to main memory resulting from missing in cache}
\]

\[
\text{CPI}_{\text{execution}} = \text{base CPI with ideal memory}
\]

Miss rate = 1 − data H1

Miss rate = 1 − instruction H1
Cache Write Strategies

1 Write Though: Data is written to both the cache block and to a block of main memory.
   - The lower level always has the most updated data; an important feature for I/O and multiprocessing.
   - Easier to implement than write back.
   - A write buffer is often used to reduce CPU write stall while data is written to memory.

2 Write Back: Data is written or updated only to the cache block. The modified or dirty cache block is written to main memory when it’s being replaced from cache.
   - Writes occur at the speed of cache
   - A status bit called a dirty or modified bit, is used to indicate whether the block was modified while in cache; if not the block is not written back to main memory when replaced.
   - Advantage: Uses less memory bandwidth than write through.
Cache Write Miss Policy

- Since data is usually not needed immediately on a write miss two options exist on a cache write miss:

  **Write Allocate:**  
  (Bring old block to cache then update it)
  The missed cache block is loaded into cache on a write miss followed by write hit actions.  
  i.e. A cache block frame is allocated for the block to be modified (written-to)

  **No-Write Allocate:**
  i.e. A cache block frame is not allocated for the block to be modified (written-to)
  The block is modified in the lower level (lower cache level, or main memory) and not loaded (written or updated) into cache.

While any of the above two write miss policies can be used with either write back or write through:

- Write back caches always use write allocate to capture subsequent writes to the block in cache.
- Write through caches usually use no-write allocate since subsequent writes still have to go to memory.
Memory Access Tree, Unified L₁

Write Through, No Write Allocate, No Write Buffer

CPU Memory Access

- **Read**
  - L₁ Read Hit: Hit Access Time = 1
  - L₁ Read Miss:
    - Access Time: M + 1
    - Stalls Per access = M
    - Stalls = % reads x (1 - H₁) x M

- **Write**
  - L₁ Write Hit:
    - Access Time: M + 1
    - Stalls Per access = M
    - Stalls = % write x H₁
  - L₁ Write Miss:
    - Access Time: M + 1
    - Stalls Per access = M
    - Stalls = % write x (1 - H₁) x M

Stall Cycles Per Memory Access = % reads x (1 - H₁) x M + % write x M

AMAT = 1 + % reads x (1 - H₁) x M + % write x M

CPI = CPI_{execution} + (1 + fraction of loads/stores) x Stall Cycles per access

Stall Cycles per access = AMAT - 1

- **Exercise:**
  - Create memory access tree for split level 1

Assuming:
- Ideal access on a read hit, no stalls

**Symbols and Definitions:**
- M = Miss Penalty
- H₁ = Level 1 Hit Rate
- 1 - H₁ = Level 1 Miss Rate
- CPI_{execution} = CPI_{execution}
- Stall Cycles per access = AMAT - 1

---

**Notes:**
- Miss Penalty = stall cycles per access resulting from missing in cache
- Miss Time = Main memory access time
- Level 1 Hit Rate = H₁
- Level 1 Miss Rate = 1 - H₁
Reducing Write Stalls For Write Though Cache Using Write Buffers

• To reduce write stalls when write though is used, a write buffer is used to eliminate or reduce write stalls:
  
  – **Perfect write buffer:** All writes are handled by write buffer, no stalling for writes
  
  – **In this case** (for unified L1 cache):
    
    Stall Cycles Per Memory Access = \( \% \text{ reads x (1 - H1 ) x M} \)
    
    (i.e No stalls at all for writes)
  
  – **Realistic Write buffer:** A percentage of write stalls are not eliminated when the write buffer is full.
  
  – **In this case** (for unified L1 cache):
    
    Stall Cycles/Memory Access = \( ( \% \text{ reads x (1 - H1 )} + \% \text{ write stalls not eliminated} ) \) x M
Write Through Cache Performance Example

- A CPU with \( \text{CPI}_{\text{execution}} = 1.1 \) Mem accesses per instruction = 1.3
- Uses a unified L1 Write Through, No Write Allocate, with:
  1. No write buffer.
  2. Perfect Write buffer
  3. A realistic write buffer that eliminates 85% of write stalls
- Instruction mix: 50% arith/logic, 15% load, 15% store, 20% control
- Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.

\[
\text{CPI} = \text{CPI}_{\text{execution}} + \text{mem stalls per instruction}
\]

\[
\% \text{ reads} = \frac{1.15}{1.3} = 88.5\% \quad \% \text{ writes} = \frac{.15}{1.3} = 11.5\%
\]

1. With No Write Buffer:
   Mem Stalls/ instruction = 1.3 x 50 x (88.5% x 1.5% + 11.5%) = 8.33 cycles
   CPI = 1.1 + 8.33 = 9.43

2. With Perfect Write Buffer (all write stalls eliminated):
   Mem Stalls/ instruction = 1.3 x 50 x (88.5% x 1.5%) = 0.86 cycles
   CPI = 1.1 + 0.86 = 1.96

3. With Realistic Write Buffer (eliminates 85% of write stalls)
   Mem Stalls/ instruction = 1.3 x 50 x (88.5% x 1.5% + 15% x 11.5%) = 1.98 cycles
   CPI = 1.1 + 1.98 = 3.08
Memory Access Tree Unified L₁
Write Back, With Write Allocate

CPU Memory Access

1 or 100%

L₁ Hit:
% = H₁
Hit Access Time = 1
Stalls = 0

H₁

L₁ Miss

(1-H₁)

(1-H₁) x % clean

(1-H₁) x % dirty

L₁ Miss, Clean
Access Time = M +1
Stalls per access = M
Stall cycles = M x (1-H₁) x % clean

L₁ Miss, Dirty
Access Time = 2M +1
Stalls per access = 2M
Stall cycles = 2M x (1-H₁) x % dirty

2M needed to Write Dirty Block and Read new block

Stall Cycles Per Memory Access = (1-H₁) x (M x % clean + 2M x % dirty)

AMAT = 1 + Stall Cycles Per Memory Access

CPI = CPI_{execution} + (1 + fraction of loads/stores) x Stall Cycles per access

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
H₁ = Level 1 Hit Rate
1 - H₁ = Level 1 Miss Rate
Write Back Cache Performance Example

- A CPU with $CPI_{\text{execution}} = 1.1$ uses a unified L1 with write back, with write allocate, and the probability a cache block is dirty = 10%.
- Instruction mix: 50% arith/logic, 15% load, 15% store, 20% control.
- Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.

$$CPI = CPI_{\text{execution}} + \text{mem stalls per instruction}$$

$$= M$$

Mem Stalls per instruction =

$$\text{Mem accesses per instruction} \times \text{Stalls per access}$$

Mem accesses per instruction = $1 + 0.3 = 1.3$

Stalls per access $= (1-H1) \times (M \times \% \text{ clean} + 2M \times \% \text{ dirty})$

Stalls per access $= 1.5\% \times (50 \times 90\% + 100 \times 10\%) = 0.825$ cycles

AMAT $= 1 + \text{stalls per access} = 1 + 0.825 = 1.825$ cycles

Mem Stalls per instruction $= 1.3 \times 0.825 = 1.07$ cycles

$$CPI = 1.1 + 1.07 = 2.17$$

The ideal CPU with no misses is $2.17/1.1 = 1.97$ times faster.
Memory Access Tree Structure
For Separate Level 1 Caches, Write Back, With Write Allocate
(AKA Split)

CPU Memory Access

% Instructions

1 or 100%

% data

Instruction

% instructions

x (1 - Instruction H1)

Instruction L1 Miss:
Access Time = M + 1
Stalls per access = M
Stalls = M x % instructions x (1 - Instruction H1)

Instruction L1 Hit:
Hit Access Time = 1
Stalls = 0

% instructions x

Instruction H1

Data

Data L1 Miss:
Hit Access Time = 1
Stalls = 0

Data L1 Hit:
Hit Access Time = 1
Stalls = 0

% data x (1 - Data H1)
% data x Data H1

Data L1 Miss, Clean
Access Time = M + 1
Stalls per access = M
Stall cycles = M x % data x (1 - Data H1) x % clean

Data L1 Miss, Dirty
Access Time = 2M + 1
Stalls per access = 2M
Stall cycles = 2M x % data x (1 - Data H1) x % dirty

Exercise: Find expression for: Stall cycles per average memory access, AMAT

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
Data H1 = Level 1 Data Hit Rate
Instruction H1 = Level 1 Instruction Hit Rate
1 - Data H1 = Level 1 Data Miss Rate
1 - Instruction H1 = Level 1 Instruction Miss Rate
% Instructions = Percentage or fraction of instruction fetches out of all memory accesses
% Data = Percentage or fraction of data accesses out of all memory accesses
% Clean = Percentage or fraction of data L1 misses that are clean
% Dirty = Percentage or fraction of data L1 misses that are dirty = 1 - % Clean

Assuming:
Ideal access on a hit in L1

CMPE550 - Shaaban
#58 Exam Review Fall 2015 11-23-2015
2 Levels of Cache: $L_1$, $L_2$

**Basic Design Rule for $L_1$ Cache:**
K.I.S.S  
(e.g. low degree of associativity  
and capacity to keep it fast)

**$L_2$ has slower access time  
than $L_1$ (5-8 cycles typical)  
But has more capacity  
and higher associativity**

**CPU**

Hit Rate= $H_1$
Hit Access Time = 1 cycle (No Stall)
Stalls for hit access = $T_1 = 0$

**$L_1$ Cache**

Local Hit Rate= $H_2$
Stalls per hit access= $T_2$
Hit Access Time = $T_2 + 1$ cycles

**Main Memory**

Slower (longer access time) than $L_2$

$L_1$ = Level 1 Cache
$L_2$ = Level 2 Cache

**Memory access penalty, $M$  
(stalls per main memory access)**
Access Time = $M + 1$

**Goal of multi-level Caches:**
Reduce the effective miss penalty incurred by level 1 cache misses  
by using additional levels of cache that capture some of these misses.
Thus hiding more main memory latency and reducing AMAT further
Miss Rates For Multi-Level Caches

- **Local Miss Rate:** This rate is the number of misses in a cache level divided by the number of memory accesses to this level (i.e., those memory accesses that reach this level).
  
  \[
  \text{Local Hit Rate} = 1 - \text{Local Miss Rate}
  \]

- **Global Miss Rate:** The number of misses in a cache level divided by the total number of memory accesses generated by the CPU.

- Since level 1 receives all CPU memory accesses, for level 1:
  
  \[
  \text{Local Miss Rate} = \text{Global Miss Rate} = 1 - H_1
  \]

- For level 2 since it only receives those accesses missed in 1:
  
  \[
  \begin{align*}
  \text{Local Miss Rate} &= \text{Miss rate}_{L_2} = 1 - H_2 \\
  \text{Global Miss Rate} &= \text{Miss rate}_{L_1} \times \text{Local Miss rate}_{L_2} \\
  &= (1 - H_1) \times (1 - H_2)
  \end{align*}
  \]

For Level 3, global miss rate?
2-Level Cache (Both Unified) Performance (Ignoring Write Policy)

CPUtime = IC x (CPI_{execution} + Mem Stall cycles per instruction) x C

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

For a system with 2 levels of unified cache, assuming no penalty when found in L1 cache: \( T_1 = 0 \)

Stall cycles per memory access =

\[
[\text{miss rate } L_1] \times[\text{Hit rate } L_2 \times \text{Hit time } L_2] + \text{Miss rate } L_2 \times \text{Memory access penalty} = (1-H_1) \times H_2 \times T_2 + (1-H_1)(1-H_2) \times M
\]

L1 Miss, L2 Hit

Here we assume \( T_1 = 0 \) (no stall on L1 hit)

H1 = L1 Hit Rate
T1 = stall cycles per L1 access hit
H2 = Local L2 Hit Rate
T2 = stall cycles per L2 access hit

L1 Miss, L2 Miss:
Must Access Main Memory

CPI = CPI_{execution} + (1 + fraction of loads and stores) x stall cycles per access
= CPI_{execution} + (1 + fraction of loads and stores) x (AMAT – 1)
2-Level Cache (Both Unified) Performance
Memory Access Tree  (Ignoring Write Policy)

CPU Stall Cycles Per Memory Access

**CPU Memory Access**

- **L1 Hit:** Hit Access Time = 1
  - Stalls = H1 x 0 = 0  (No Stall)
  - L1 Hit: % = (1 - H1)

- **L1 Miss:** % = (1 - H1)

- **L1 Miss, L2 Hit:** Hit Access Time = T2 + 1
  - Stalls per L2 Hit = T2
  - Stalls = (1 - H1) x H2 x T2

- **L1 Miss, L2 Miss:** Access Time = M + 1
  - Stalls per access = M
  - Stalls = (1 - H1)(1 - H2) x M

Stall cycles per memory access = (1 - H1) x H2 x T2 + (1 - H1)(1 - H2) x M

AMAT = 1 + (1 - H1) x H2 x T2 + (1 - H1)(1 - H2) x M

CPI = CPI_{\text{execution}} + (1 + \text{fraction of loads and stores}) \times \text{stall cycles per access}
= CPI_{\text{execution}} + (1 + \text{fraction of loads and stores}) \times (AMAT - 1)

Assuming:
- Ideal access on a hit in L1
  - T1 = 0

CMPE550 - Shaaban
Unified Two-Level Cache Example

- CPU with \( \text{CPI}_{\text{execution}} = 1.1 \) running at clock rate = 500 MHz
- 1.3 memory accesses per instruction.
- With two levels of cache (both unified)
- \( L_1 \) hit access time = 1 cycle (no stall on a hit, \( T_1 = 0 \)), a miss rate of 5%
- \( L_2 \) hit access time = 3 cycles (\( T_2 = 2 \) stall cycles per hit) with local miss rate 40%,
- Memory access penalty, \( M = 100 \) cycles (stalls per access). Find CPI ...

\[ \text{CPI} = \text{CPI}_{\text{execution}} + \text{Mem Stall cycles per instruction} \]

With No Cache, \( \text{CPI} = 1.1 + 1.3 \times 100 = 131.1 \)

With single \( L_1 \), \( \text{CPI} = 1.1 + 1.3 \times 0.05 \times 100 = 7.6 \)

Mem Stall cycles per instruction = Mem accesses per instruction \( \times \) Stall cycles per access

Stall cycles per memory access = \( (1-H_1) \times H_2 \times T_2 + (1-H_1)(1-H_2) \times M \)

\[ = 0.05 \times 0.6 \times 2 + 0.05 \times 0.4 \times 100 \]

\[ = 0.06 + 2 = 2.06 \text{ cycles} \]

\[ \text{AMAT} = 2.06 + 1 = 3.06 \text{ cycles} \]

Mem Stall cycles per instruction = Mem accesses per instruction \( \times \) Stall cycles per access

\[ = 2.06 \times 1.3 = 2.678 \text{ cycles} \]

\[ \text{CPI} = 1.1 + 2.678 = 3.778 \]

\[ \text{Speedup} = 7.6/3.778 = 2 \]

Compared to CPU with L1 only

\[ \text{CPI} = \text{CPI}_{\text{execution}} + (1 + \text{fraction of loads and stores}) \times \text{stall cycles per access} = \text{CPI}_{\text{execution}} + (1 + \text{fraction of loads and stores}) \times (\text{AMAT} - 1) \]
Memory Access Tree Structure For 2-Level Cache
(Separate Level 1 Caches, Unified Level 2)

CPU Memory Access

% Instructions = Percentage or fraction of instruction fetches out of all memory accesses
% Data = Percentage or fraction of data accesses out of all memory accesses

For L1:
- T1 = Stalls per hit access to level 1
- Data H1 = Level 1 Data Hit Rate
- Instruction H1 = Level 1 Instruction Hit Rate

For L2:
- T2 = Stalls per access to level 2
- H2 = Level 2 local hit Rate

M = Miss Penalty = stall cycles per access resulting from missing in cache level 2
M + 1 = Miss Time = Main memory access time

Exercise: In terms of the parameters below, complete the memory access tree and find the expression for stall cycles per memory access.
Common Write Policy For 2-Level Cache

- **Write Policy For Level 1 Cache:**
  - Usually Write through to Level 2. (not write through to main memory just to L2)
  - Write allocate is used to reduce level 1 read misses.
  - Use write buffer to reduce write stalls to level 2.

- **Write Policy For Level 2 Cache:**
  - Usually write back with write allocate is used.
    - To minimize memory bandwidth usage.

- The above 2-level cache write policy results in inclusive L2 cache since the content of L1 is also in L2
  - Common in the majority of all CPUs with 2-levels of cache
  - As opposed to exclusive L1, L2 (e.g AMD Athlon XP, A64)

\[
\text{L1} \quad \text{L2}
\]

**As if we have a single level of cache with one portion (L1) is faster than remainder (L2)**

**i.e what is in L1 is not duplicated in L2**
2-Level (Both Unified) Memory Access Tree

L1: Write Through to L2, Write Allocate, With Perfect Write Buffer
L2: Write Back with Write Allocate

Assuming:
Ideal access on a hit in L1
T1 = 0

CPU Memory Access

L1 Hit:
Hit Access Time = 1
Stalls Per access = 0

L1 Miss:
(1-H1) x H2

L1 Miss, L2 Hit:
Hit Access Time = T2 + 1
Stalls per L2 Hit = T2
Stalls = (1-H1) x H2 x T2

L1 Miss, L2 Miss

(1-H1) x (1-H2)

L1 Miss, L2 Miss, Clean
Access Time = M + 1
Stalls per access = M
Stall cycles = M x (1-H1) x (1-H2) x % clean

Stall cycles per memory access = (1-H1) x H2 x T2 + M x (1-H1) x (1-H2) x % clean + 2M x (1-H1) x (1-H2) x % dirty

AMAT = 1 + Stall Cycles Per Memory Access

CPI = CPI_{execution} + (1 + fraction of loads and stores) x Stall Cycles per access

CMPE550 - Shaaban
Two-Level (Both Unified) Cache Example With Write Policy

- CPU with $CPI_{\text{execution}} = 1.1$ running at clock rate = 500 MHz
- 1.3 memory accesses per instruction. Two levels of cache (both unified)
  
  For $L_1$:
  - Cache operates at 500 MHz (no stall on L1 Hit, $T_1 = 0$) with a miss rate of $1 - H_1 = 5\%$
  - Write though to L2 with perfect write buffer with write allocate
  
  For $L_2$:
  - Hit access time = 3 cycles ($T_2 = 2$ stall cycles per hit) local miss rate $1 - H_2 = 40\%$
  - Write back to main memory with write allocate
  - Probability a cache block is dirty = 10\%

- Memory access penalty, $M = 100$ cycles.
- Create memory access tree and find, stalls per memory access, AMAT, CPI.

Stall cycles per memory access =

\[
(1 - H_1) \times H_2 \times T_2 + \frac{(1 - H_1) \times (1 - H_2) \times (\% \text{ clean} \times M + \% \text{ dirty} \times 2M)}{1 - H_1} = 0.05 \times 0.6 \times 2 + 0.05 \times 0.4 \times (0.9 \times 100 + 0.1 \times 200)
\]

\[
= 0.06 + 0.02 \times 110 = 0.06 + 2.2 = 2.26
\]

- $AMAT = 2.26 + 1 = 3.26$ cycles

Mem Stall cycles per instruction = Mem accesses per instruction $\times$ Stall cycles per access

\[
= 2.26 \times 1.3 = 2.938 \text{ cycles}
\]

$CPI = 1.1 + 2.938 = 4.038 = 4$
Memory Access Tree For Two-Level (Both Unified) Cache Example With Write Policy

L1: Write Through to L2, Write Allocate, With Perfect Write Buffer
L2: Write Back with Write Allocate

CPU Memory Access

L1 Hit:
Hit Access Time = 1
Stalls Per access = 0

H1 = 0.95 or 95%
(1-H1) = 0.05 or 5%

L1 Miss:
(1-H1) x H2 = 0.05 x 0.6 = 0.03 or 3%

L1 Miss, L2 Hit:
Hit Access Time = T2 +1 = 3 cycles
Stalls per L2 Hit = T2 = 2 cycles
Stalls = (1-H1) x H2 x T2
= 0.03 x 2 = 0.06 cycles

(1-H1) x (1-H2) = 0.05 x 0.4 = 0.02 or 2%

L1 Miss, L2 Miss
Access Time = M +1 = 101 cycles
Stalls per access = M
Stall cycles = M x (1 -H1) x (1-H2) x % clean
= 100 x 0.018 = 1.8 cycles

(1-H1) x (1-H2) x % clean
= 0.02 x 0.9 = 0.018 or 1.8%

L1 Miss, L2 Miss, Clean
Access Time = M +1 = 101 cycles
Stalls per access = M
Stall cycles = M x (1 -H1) x (1-H2) x % clean
= 100 x 0.018 = 1.8 cycles

(1-H1) x (1-H2) x % clean
= 0.02 x 0.9 = 0.018 or 1.8%

L1 Miss, L2 Miss, Dirty
Access Time = 2M +1 = 200 + 1 = 201 cycles
Stalls per access = 2M
Stall cycles = 2M x (1-H1) x (1-H2) x % dirty
= 200 x 0.002 = 0.4 cycles

(1-H1) x (1-H2) x % dirty
= 0.02 x 0.1 = 0.002 or 0.2%

Stall cycles per memory access = (1-H1) x H2 x T2 + M x (1-H1) x (1-H2) x % clean + 2M x (1-H1) x (1-H2) x % dirty
= 0.06 + 1.8 + 0.4 = 2.26 cycles

AMAT = 1 + Stall cycles per memory access = 1 + 2.26 = 3.26 cycles
Stall cycles per instruction = (1 + fraction of loads/stores) x Stall Cycles per access
= 1.3 x 2.26 = 2.938 cycles
CPI = CPI_{execution} + Stall cycles per instruction = 1.1 + 2.938 = 4.038
Memory Access Tree Structure For 2-Level Cache (Separate Level 1 Caches, Unified Level 2)
L1: Write Through to L2, Write Allocate, With Perfect Write Buffer   L2: Write Back with Write Allocate

CPU Memory Access

<table>
<thead>
<tr>
<th>% Instructions</th>
<th>% data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Data</td>
</tr>
<tr>
<td>L1 Hit</td>
<td></td>
</tr>
<tr>
<td>L1 Miss</td>
<td></td>
</tr>
<tr>
<td>Instruction L1 Hit:</td>
<td>Instruction L1 Miss:</td>
</tr>
<tr>
<td>L2 Hit</td>
<td>L2 Miss</td>
</tr>
<tr>
<td>L2 Miss Clean</td>
<td>L2 Miss Dirty</td>
</tr>
<tr>
<td>L2 Miss Dirty</td>
<td>L2 Miss Clean</td>
</tr>
<tr>
<td>L2 Miss Dirty</td>
<td>L2 Miss Dirty</td>
</tr>
</tbody>
</table>

Exercise: In terms of the parameters below, complete the memory access tree and find the expression for stall cycles per memory access

For L1:
- $T_1 =$ Stalls per hit access to level 1
- $D_{H1} =$ Level 1 Data Hit Rate
- $I_{H1} =$ Level 1 Instruction Hit Rate

For L2:
- $T_2 =$ Stalls per access to level 2
- $H_2 =$ Level 2 local hit Rate
- $1 - H_2 =$ Level 2 local miss rate
- $D_{H2} =$ Level 2 Data Hit Rate
- $I_{H2} =$ Level 2 Instruction Hit Rate

% Clean = Percentage or fraction of data L2 misses that are clean
% Dirty = Percentage or fraction of L2 misses that are dirty = 1 - % Clean

$M =$ Miss Penalty = stall cycles per access resulting from missing in cache level 2
$M + 1 =$ Miss Time = Main memory access time
3 Levels of Cache

**CPU**

- Hit Rate = $H_1$
- Hit Access Time = 1 cycle (No Stall)
- Stalls for hit access = $T_1 = 0$

**L1 Cache**

- Local Hit Rate = $H_2$
- Stalls per hit access = $T_2$
- Hit Access Time = $T_2 + 1$ cycles

**L2 Cache**

- Slower than L1 (5-8 cycles typical)
- But has more capacity and higher associativity
- Local Hit Rate = $H_3$
- Stalls per hit access = $T_3$
- Hit Access Time = $T_3 + 1$ cycles

**L3 Cache**

- Slower than L2 (12-20 cycles typical)
- But has more capacity and higher associativity

**Main Memory**

- Slower than L3
- Memory access penalty, $M$
  
  - (stalls per main memory access)
  
  - Access Time = $M + 1$

CPI = $CPI_{\text{execution}} + (1 + \text{fraction of loads and stores}) \times \text{stall cycles per access}$

= $CPI_{\text{execution}} + (1 + \text{fraction of loads and stores}) \times (\text{AMAT} - 1)$

**Basic Design Rule for L1 Cache:**

K.I.S.S

(e.g. low degree of associativity and capacity to keep it fast)

**Assuming**

Ideal access on a hit in L1

**Multi-Level Cache:**

- Slower than L1 (5-8 cycles typical)
  - But has more capacity and higher associativity

- Slower than L2 (12-20 cycles typical)
  - But has more capacity and higher associativity

---

CMPE550 - Shaaban

#70 Exam Review Fall 2015 11-23-2015
3-Level (All Unified) Cache Performance

CPU\text{time} = IC \times (CPI_{\text{execution}} + \text{Mem Stall cycles per instruction}) \times C

Mem Stall cycles per instruction = Mem accesses per instruction \times Stall cycles per access

- For a system with 3 levels of cache, assuming no penalty when found in L_1 cache: \quad (T_1 = 0)

Stall cycles per memory access =

\[ [\text{miss rate } L_1] \times [\text{Hit rate } L_2 \times \text{Hit time } L_2 +\text{Miss rate } L_2 \times (\text{Hit rate } L_3 \times \text{Hit time } L_3 + \text{Miss rate } L_3 \times \text{Memory access penalty})] =\]

\[ (1-H1) \times H2 \times T2 + (1-H1) \times (1-H2) \times H3 \times T3 + (1-H1)(1-H2)(1-H3) \times M \]

CPI = CPI_{\text{execution}} + (1 + \text{fraction of loads and stores}) \times \text{stall cycles per access}

= CPI_{\text{execution}} + (1 + \text{fraction of loads and stores}) \times (\text{AMAT} - 1)
3-Level (All Unified) Cache Performance

Memory Access Tree (Ignoring Write Policy)

CPU Stall Cycles Per Memory Access

CPU Memory Access

L1 Hit:
- Hit Access Time = 1
- Stalls Per access = T1 = 0
- Stalls = H1 x 0 = 0
  (No Stall)

L1 Miss:
- % = (1-H1)

L1 Miss, L2 Hit:
- Hit Access Time = T2 + 1
- Stalls per L2 Hit = T2
- Stalls = (1-H1) x H2 x T2

L1 Miss, L2 Miss:
- % = (1-H1)(1-H2)

L1 Miss, L2 Hit, L3 Hit:
- Hit Access Time = T3 + 1
- Stalls per L2 Hit = T3
- Stalls = (1-H1) x H2 x T2 x H3 x T3

Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1) x (1-H2) x H3 x T3 + (1-H1)(1-H2)(1-H3) x M

AMAT = 1 + Stall cycles per memory access

CPI = CPI_{execution} + (1 + fraction of loads and stores) x stall cycles per access
= CPI_{execution} + (1 + fraction of loads and stores) x (AMAT - 1)

T1 = Stalls per hit access for Level 1
T2 = Stalls per hit access for Level 2
T3 = Stalls per hit access for Level 3

Here we assume ideal access on a hit in L1
Thus T1 = 0

Global Hit Rate for Level 2

Global Hit Rate for Level 3

Global Miss Rate for Level 2

Global Miss Rate for Level 3

CMPE550 - Shaaban

#72 Exam Review Fall 2015 11-23-2015
Three-Level (All Unified) Cache Example

- CPU with CPI_{\text{execution}} = 1.1 running at clock rate = 500 MHz
- 1.3 memory accesses per instruction.
- L_1 cache operates at 500 MHz (no stalls on a hit in L1) with a miss rate of 5%
- L_2 hit access time = 3 cycles (T2= 2 stall cycles per hit), local miss rate 40%
- L_3 hit access time = 6 cycles (T3= 5 stall cycles per hit), local miss rate 50%,
- Memory access penalty, M= 100 cycles (stall cycles per access). Find CPI.

With No Cache, \[ \text{CPI} = 1.1 + 1.3 \times 100 = 131.1 \]

With single L_1, \[ \text{CPI} = 1.1 + 1.3 \times 0.05 \times 100 = 7.6 \]

With L1, L2 \[ \text{CPI} = 1.1 + 1.3 \times (0.05 \times 0.6 \times 2 + 0.05 \times 0.4 \times 100) = 3.778 \]

\[ \text{CPI} = \text{CPI}_{\text{execution}} + \text{Mem Stall cycles per instruction} \]

Mem Stall cycles per instruction = Mem accesses per instruction \times Stall cycles per access

Stall cycles per memory access \[ = (1-H_1) \times H_2 \times T_2 + (1-H_1) \times (1-H_2) \times H_3 \times T_3 + (1-H_1)(1-H_2)(1-H_3) \times M \]
\[ = 0.05 \times 0.6 \times 2 + 0.05 \times 0.4 \times 0.5 \times 5 + 0.05 \times 0.4 \times 0.5 \times 100 \]
\[ = 0.06 + 0.05 + 1 = 1.11 \]

AMAT = 1.11 + 1 = 2.11 cycles (vs. AMAT = 3.06 with L1, L2, vs. 5 with L1 only)

\[ \text{CPI} = 1.1 + 1.3 \times 1.11 = 2.54 \]

Speedup compared to L1 only \[ = 7.6/2.54 = 3 \]

Speedup compared to L1, L2 \[ = 3.778/2.54 = 1.49 \]
Main Memory

- Main memory generally utilizes Dynamic RAM (DRAM), which use a single transistor to store a bit, but require a periodic data refresh by reading every row increasing cycle time.  
  DRAM: Slow but high density
- Static RAM may be used for main memory if the added expense, low density, high power consumption, and complexity is feasible (e.g. Cray Vector Supercomputers).
- Main memory performance is affected by:
  - **Memory latency**: Affects cache miss penalty, M. Measured by:
    - **Memory Access time**: The time it takes between a memory access request is issued to main memory and the time the requested information is available to cache/CPU.
    - **Memory Cycle time**: The minimum time between requests to memory (greater than access time in DRAM to allow address lines to be stable)
  - **Peak Memory bandwidth**: The maximum sustained data transfer rate between main memory and cache/CPU.
    - In current memory technologies (e.g Double Data Rate SDRAM) published peak memory bandwidth does not take account most of the memory access latency.
    - This leads to achievable realistic memory bandwidth < peak memory bandwidth
Basic Memory Bandwidth Improvement/Miss Penalty (M) Latency Reduction Techniques

• **Wider Main Memory (CPU-Memory Bus):**

  Memory bus width is increased to a number of words (usually up to the size of a cache block).
  - Memory bandwidth is proportional to memory bus width.
    - e.g. Doubling the width of cache and memory doubles potential memory bandwidth available to the CPU.
    - The miss penalty is reduced since fewer memory bus accesses are needed to fill a cache block on a miss.

• **Interleaved (Multi-Bank) Memory:**

  Memory is organized as a number of independent banks.
  - Multiple interleaved memory reads or writes are accomplished by sending memory addresses to several memory banks at once or pipeline access to the banks.
  - **Interleaving factor:** Refers to the mapping of memory addresses to memory banks. Goal reduce bank conflicts.
    - e.g. using 4 banks (width one word), bank 0 has all words whose address is:
      
      \[(\text{word address mod} \ 4) = 0\]

  The above two techniques can also be applied to any cache level to reduce cache hit time and increase cache bandwidth.
Memory Bank Interleaving  
(Multi-Banked Memory)

Can be applied at: 1- DRAM chip level (e.g. SDRAM, DDR)  2- DRAM module level  3- DRAM channel level

Access Pattern without Interleaving: (One Memory Bank)

Pipeline access to different memory banks to increase effective bandwidth

Access Pattern with 4-way Interleaving:

Bank interleaving does not reduce latency of accesses to the same bank

We can Access Bank 0 again

Number of banks ≥ Number of cycles to access word in a bank
Memory Width, Interleaving: Performance Example
(i.e multiple memory banks)

| Given the following system parameters with single unified cache level L1 (ignoring write policy): |
| Block size= 1 word Memory bus width= 1 word Miss rate =3% M = Miss penalty = 32 cycles |
| (4 cycles to send address 24 cycles access time, 4 cycles to send a word to CPU) |
| Memory access/instruction = 1.2 CPI_{\text{execution}} (ignoring cache misses) = 2 |
| Miss rate (block size = 2 word = 8 bytes ) = 2% Miss rate (block size = 4 words = 16 bytes) = 1% |

- The CPI of the base machine with 1-word blocks = 2 + (1.2 x 0.03 x 32) = 3.15 (For Base system)

Increasing the block size to two words (64 bits) gives the following CPI: (miss rate = 2%)

- 32-bit bus and memory, no interleaving, M = 2 x 32 = 64 cycles CPI = 2 + (1.2 x 0.02 x 64) = 3.54
- 32-bit bus and memory, interleaved, M = 4 + 24 + 8 = 36 cycles CPI = 2 + (1.2 x 0.02 x 36) = 2.86
- 64-bit bus and memory, no interleaving, M = 32 cycles CPI = 2 + (1.2 x 0.02 x 32) = 2.77

Increasing the block size to four words (128 bits); resulting CPI: (miss rate = 1%)

- 32-bit bus and memory, no interleaving , M = 4 x 32 = 128 cycles CPI = 2 + (1.2 x 0.01 x 128) = 3.54
- 32-bit bus and memory, interleaved , M = 4 + 24 + 16 = 44 cycles CPI = 2 + (1.2 x 0.01 x 44) = 2.53
- 64-bit bus and memory, no interleaving, M = 2 x 32 = 64 cycles CPI = 2 + (1.2 x 0.01 x 64) = 2.77
- 64-bit bus and memory, interleaved, M = 4 + 24 + 8 = 36 cycles CPI = 2 + (1.2 x 0.01 x 36) = 2.43
- 128-bit bus and memory, no interleaving, M = 32 cycles CPI = 2 + (1.2 x 0.01 x 32) = 2.38

Miss Penalty = M = Number of CPU stall cycles for an access missed in cache and satisfied by main memory
Program Steady-State Bandwidth-Usage Example

- In the previous example with three levels of cache (all unified, ignore write policy)
- CPU with CPI_{execution} = 1.1 running at clock rate = 500 MHz
- 1.3 memory accesses per instruction.
- L_1 cache operates at 500 MHz (no stalls on a hit in L1) with a miss rate of 5%
- L_2 hit access time = 3 cycles (T_2 = 2 stall cycles per hit), local miss rate 40%
- L_3 hit access time = 6 cycles (T_3 = 5 stall cycles per hit), local miss rate 50%,
- Memory access penalty, M = 100 cycles (stall cycles per access to deliver 32 bytes from main memory to CPU)

- We found the CPI:
  With No Cache, CPI = 1.1 + 1.3 \times 100 = 131.1
  With single L_1, CPI = 1.1 + 1.3 \times 0.05 \times 100 = 7.6
  With L_1, L_2 CPI = 1.1 + 1.3 \times (0.05 \times 0.6 \times 2 + 0.05 \times 0.4 \times 100) = 3.778
  With L_1, L_2, L_3 CPI = 1.1 + 1.3 \times 1.11 = 2.54

Assuming that all cache blocks are 32 bytes

For each of the three cases with cache:
A. What is the peak (or maximum) number of memory accesses and effective peak bandwidth for each cache level and main memory?
B. What is the total number of memory accesses generated by the CPU per second?
C. What percentage of these memory accesses reach each cache level/memory and what percentage of each cache level/memory bandwidth is used by the CPU?
3-Level (All Unified) Cache Performance

Memory Access Tree (Ignoring Write Policy)

CPU Stall Cycles Per Memory Access

**CPU Memory Access** (100%)

- **L1 Hit:**
  - Hit Access Time = 1
  - Stalls Per access = 0
  - Stalls = H1 x 0 = 0
  - (No Stall)

- **L1 Miss:**
  - % = (1-H1) = 0.05 or 5%

**L1 Hit, L2 Hit:**

- Hit Access Time = T2 + 1 = 3
- Stalls per L2 Hit = T2 = 2
- Stalls = (1-H1) x H2 x T2
  - = 0.05 x 0.6 x 2 = 0.06

**L1 Miss, L2 Miss:**

- Miss Penalty = M = 100
- Stalls = (1-H1)(1-H2)(1-H3) x M
  - = 0.01 x 100 = 1 cycle

- Stall cycles per memory access
  - = (1-H1) x H2 x T2 + (1-H1) x (1-H2) x H3 x T3 + (1-H1)(1-H2)(1-H3) x M +1
  - = 0.06 + 0.05 + 1 = 1.11

**AMAT**

- AMAT = 1 + Stall cycles per memory access
  - = 1 + 1.11 = 2.11 cycles

- T2 = 2 cycles = Stalls per hit access for Level 2
- T3 = 5 cycles = Stalls per hit access for Level 3
- M = Memory Miss Penalty = M = 100 cycles

**H1** = 95% or 0.95
- T1 = 0 cycles

**H2** = 60% or 0.6
- T2 = 2 cycles

**H3** = 50% or 0.5
- T3 = 5 cycles

**M** = 100 cycles

**CPI** = CPIExecution + (1 + fraction of loads and stores) x stalls per access

- CPI = 1.1 + 1.3 x 1.11 = 2.54
A. What is the peak (or maximum) number of memory accesses and effective peak bandwidth for each cache level and main memory?

- L1 cache requires 1 CPU cycle to deliver 32 bytes, thus:
  Maximum L1 accesses per second = 500 x 10^6 accesses/second
  Maximum effective L1 bandwidth = 32 x 500 x 10^6 = 16,000 x 10^6 = 16 x 10^9 byes/sec

- L2 cache requires 3 CPU cycles to deliver 32 bytes, thus:
  Maximum L2 accesses per second = 500/3 x 10^6 = 166.67 x 10^6 accesses/second
  Maximum effective L2 bandwidth = 32 x 166.67 x 10^6 = 5,333.33 x 10^6 = 5.33 x 10^9 byes/sec

- L3 cache requires 6 CPU cycles to deliver 32 bytes, thus:
  Maximum L3 accesses per second = 500/6 x 10^6 = 83.33 x 10^6 accesses/second
  Maximum effective L3 bandwidth = 32 x 166.67 x 10^6 = 2,666.67 x 10^6 = 2.67 x 10^9 byes/sec

- Memory requires 101 CPU cycles (101 = M+1 = 100+1) to deliver 32 bytes, thus:
  Maximum main memory accesses per second = 500/101 x 10^6 = 4.95 x 10^6 accesses/second
  Maximum effective main memory bandwidth = 32 x 4.95 x 10^6 = 158.42 x 10^6 byes/sec
Program Steady-State Bandwidth-Usage Example

• For CPU with L1 Cache:
  B. What is the total number of memory accesses generated by the CPU per second?
  
  - The total number of memory accesses generated by the CPU per second =
    (memory access/instruction) \times \text{clock rate} / \text{CPI} = 1.3 \times 500 \times 10^6 / \text{CPI} = 650 \times 10^6 / \text{CPI}
  - With single L1 cache CPI was found = 7.6
    - CPU memory accesses = \frac{650 \times 10^6}{7.6} = 85 \times 10^6 \text{ accesses/sec}

C. What percentage of these memory accesses reach each cache level/memory and what percentage of each cache level/memory bandwidth is used by the CPU?

• For L1:
  The percentage of CPU memory accesses that reach L1 = 100%
  L1 Cache bandwidth usage = 32 \times 85 \times 10^6 = 2,720 \times 10^6 = 2.7 \times 10^9 \text{ byes/sec}
  Percentage of L1 bandwidth used = \frac{2,720}{16,000} = 0.17 \text{ or } 17\%
    (or by just dividing CPU accesses / peak L1 accesses = 85/500 = 0.17 = 17\%)

• For Main Memory:
  The percentage of CPU memory accesses that reach main memory = (1-H1) = 0.05 \text{ or } 5%
  Main memory bandwidth usage = 0.05 \times 32 \times 85 \times 10^6 = 136 \times 10^6 \text{ byes/sec}
  Percentage of main memory bandwidth used = \frac{136}{158.42} = 0.8585 \text{ or } 85.85\%
**Program Steady-State Bandwidth-Usage Example**

- **For CPU with L1, L2 Cache:**
  
  **B. What is the total number of memory accesses generated by the CPU per second?**
  
  The total number of memory accesses generated by the CPU per second = (memory access/instruction) x clock rate / CPI = 1.3 x 500 x 10⁶ / CPI = 650 x 10⁶ / CPI

  - With L1, L2 cache CPI was found = 3.778
    
    CPU memory accesses = 650 x 10⁶ / 3.778 = 172 x 10⁶ accesses/sec

  Vs. With L1 only = 85 x 10⁶ accesses/sec

- **C. What percentage of these memory accesses reach each cache level/memory and what percentage of each cache level/memory bandwidth is used by the CPU?**
  
  **For L1:**
  
  The percentage of CPU memory accesses that reach L1 = 100%

  L1 Cache bandwidth usage = 32 x 172 x 10⁶ = 5,505 x 10⁶ = 5.505 x10⁹ byes/sec

  Percentage of L1 bandwidth used = 5,505 / 16,000 = 0.344 or 34.4%

  (or by just dividing CPU accesses / peak L1 accesses = 172/500 = 0.344 = 34.4%)

  Vs. With L1 only = 17%

  **For L2:**
  
  The percentage of CPU memory accesses that reach L2 = (I-H1) = 0.05 or 5%

  L2 Cache bandwidth usage = 0.05x 32 x 172 x 10⁶ = 275.28 x 10⁶ byes/sec

  Percentage of L2 bandwidth used = 275.28 / 5,333.33 = 0.0516 or 5.16%

  (or by just dividing CPU accesses that reach L2 / peak L2 accesses = 0.05 x 172/166.67 = 0.0516= 5.16%)

  **For Main Memory:**
  
  The percentage of CPU memory accesses that reach main memory = (1-H1) x (1-H2) = 0.05 x 0.4 = 0.02 or 2%

  Main memory bandwidth usage = 0.02 x 32 x 172 x 10⁶ = 110.11 x 10⁶ byes/sec

  Percentage of main memory bandwidth used = 110.11 / 158.42 = 0.695 or 69.5%

  Vs. With L1 only = 85.5%

**Exercises:**

- What if Level 1 (L1) is split?
- What if Level 2 (L2) is write back with write allocate?
Program Steady-State Bandwidth-Usage Example

- **For CPU with L1, L2, L3 Cache:**

  **B. What is the total number of memory accesses generated by the CPU per second?**

  - The total number of memory accesses generated by the CPU per second = (memory access/instruction) x clock rate / CPI = 1.3 x 500 x 10^6 / CPI = 650 x 10^6 / CPI

  - With L1, L2, L3 cache CPI was found = 2.54

    - CPU memory accesses = 650 x 10^6 / 2.54 = 255.9 x 10^6 accesses/sec

  - With L1, L2 = 172 x 10^6 accesses/sec

  **C. What percentage of these memory accesses reach each cache level/memory and what percentage of each cache level/memory bandwidth is used by the CPU?**

  - **For L1:**

    - The percentage of CPU memory accesses that reach L1 = 100%
    - L1 Cache bandwidth usage = 32 x 255.9 x 10^6 = 8,188 x 10^6 = 8.188 x 10^9 byes/sec
    - Percentage of L1 bandwidth used = 8,188 / 16,000 = 0.5118 or 51.18%
    - 172/500 = 0.344 = 34.4%

  - **For L2:**

    - The percentage of CPU memory accesses that reach L2 = (1-H1) = 0.05 or 5%
    - L2 Cache bandwidth usage = 0.05 x 32 x 255.9 x 10^6 = 409.45 x 10^6 byes/sec
    - Percentage of L2 bandwidth used = 409.45 / 5,333.33 = 0.077 or 7.7%
    - 0.05 x 255.9 / 166.67 = 12.795 / 166.67 = 0.077 = 7.7%

  - **For L3:**

    - The percentage of CPU memory accesses that reach L3 = (1-H1)x (1-H2) = 0.02 or 2%
    - L3 Cache bandwidth usage = 0.02 x 32 x 255.9 x 10^6 = 163.78 x 10^6 byes/sec
    - Percentage of L3 bandwidth used = 163.78 / 2,666.67 = 0.061 or 6.1%
    - 0.02 x 255.9 / 83.33 = 5.118 / 83.33 = 0.061 = 6.1%

  - **For Main Memory:**

    - The percentage of CPU memory accesses that reach main memory = (1-H1) x (1-H2) x (1-H3)= .05 x .4 x .5 = 0.01 or 1%
    - Main memory bandwidth usage = 0.01 x 32 x 255.9 x 10^6 = 81.89 x 10^6 byes/sec
    - Percentage of main memory bandwidth used = 110.11 / 158.42 = 0.517 or 51.7%

**Exercises:**
- What if Level 1 (L1) is split?
- What if Level 3 (L3) is write back with write allocate?
I/O Performance Metrics/Modeling

- **Diversity**: The variety of I/O devices that can be connected to the system.
- **Capacity**: The maximum number of I/O devices that can be connected to the system.

**I/O Performance Metrics/Modeling**: The producer (CPU, human etc.) creates tasks to be performed and places them in a task buffer (queue); the server (I/O device or controller) takes tasks from the queue and performs them.

**I/O Performance Metrics**:

- **I/O Throughput**: The maximum data rate that can be transferred to/from an I/O device or sub-system, or the maximum number of I/O tasks or transactions completed by I/O in a certain period of time
  
  ⇒ Maximized when task queue is never empty (server always busy).

- **I/O Latency or response time**: The time an I/O task takes from the time it is placed in the task buffer or queue until the server (I/O system) finishes the task. Includes I/O device service time and buffer waiting (or queuing time).
  
  ⇒ Minimized when task queue is always empty (no queuing time).

  Response Time = Service Time + Queuing Time

CMPE550 - Shaaban

4th Edition: Chapter 6.1, 6.2, 6.4, 6.5
3rd Edition: Chapter 7.1-7.3, 7.7, 7.8
Producer-Server Model

Throughput vs. Response Time

Response Time = Time\text{System} = Time\text{Queue} + Time\text{Server} = T_q + T_{\text{ser}}

Queue almost empty most of the time
Less time in queue

Queue full most of the time
More time in queue

i.e Utilization = U ranges from 0 to 1 (0 % to 100%)
Magnetic Disks

Characteristics:

- Diameter (form factor): 2.5in - 5.25in
- Rotational speed: 3,600RPM-15,000 RPM
- Tracks per surface.
- Sectors per track: Outer tracks contain more sectors.
- Recording or Areal Density: Tracks/in X Bits/in
- Cost Per Megabyte. Bits/ Inch²
- Seek Time: (2-12 ms) Current Areal Density ~ 100 Gbits / Inch²
  The time needed to move the read/write head arm.
  Reported values: Minimum, Maximum, Average.
- Rotation Latency or Delay: (2-8 ms)
  The time for the requested sector to be under the read/write head. (~ time for half a rotation)
- Transfer time: The time needed to transfer a sector of bits.
- Type of controller/interface: SCSI, EIDE (PATA, SATA)
- Disk Controller delay or time.
- Average time to access a sector of data =
  average seek time + average rotational delay + transfer time
  + disk controller overhead
  (ignoring queuing time)

Access time = average seek time + average rotational delay
Basic Disk Performance Example

- Given the following Disk Parameters:
  - Average seek time is 5 ms
  - Disk spins at 10,000 RPM
  - Transfer rate is 40 MB/sec
- Controller overhead is 0.1 ms
- Assume that the disk is idle, so no queuing delay exist.
- What is Average Disk read or write service time for a 500-byte (.5 KB) Sector?

\[
\text{Average service time} = 5 \text{ ms} + \frac{0.5}{10000 \text{ RPM}/60} + \frac{0.5 \text{ KB}}{40 \text{ MB/s}} + 0.1 \text{ ms}
\]

\[
= 5 + 0.03 + 0.13 + 0.1 = 5.26 \text{ ms}
\]

\[
T_{\text{service}} = \text{Disk Service Time for this request}
\]

Here: 1KBytes = 10^3 bytes, MByte = 10^6 bytes, 1 GByte = 10^9 bytes
I/O Performance & Little’s Queuing Law

Given: An I/O system in equilibrium (input rate is equal to output rate) and:

- \( T_{\text{ser}} \): Average time to service a task = 1/Service rate
- \( T_q \): Average time per task in the queue
- \( T_{\text{sys}} \): Average time per task in the system, or the response time, the sum of \( T_{\text{ser}} \) and \( T_q \) thus \( T_{\text{sys}} = T_{\text{ser}} + T_q \)
- \( r \): Average number of arriving tasks/sec (i.e. task arrival rate)
- \( L_{\text{ser}} \): Average number of tasks in service.
- \( L_q \): Average length of queue
- \( L_{\text{sys}} \): Average number of tasks in the system, the sum of \( L_q \) and \( L_{\text{ser}} \)

**Little’s Law states:**

\[
L_{\text{sys}} = r \times T_{\text{sys}} \quad \text{(applied to system)}
\]
\[
L_q = r \times T_q \quad \text{(applied to queue)}
\]

**Server utilization** = \( u = \frac{r}{\text{Service rate}} = r \times T_{\text{ser}} \)

\( u \) must be between 0 and 1 otherwise there would be more tasks arriving than could be serviced

Here a server is the device (i.e. hard drive) and its I/O controller (IOC)
A Little Queuing Theory: M/G/1 and M/M/1

**Assumptions:**
- System in equilibrium
- Time between two successive arrivals in line are random
- Server can start on next customer immediately after prior finishes
- No limit to the queue: works First-In-First-Out
- Afterward, all customers in line must complete; each avg $T_{ser}$

- Described “memoryless” or Markovian request arrival
  (M for C=1 exponentially random), General service distribution (no restrictions), 1 server: $M/G/1$ queue

- When Service times have C = 1, $M/M/1$ queue

$$T_q = T_{ser} \times u / (1 - u)$$

- $T_{ser}$ average time to service a task
- $r$ average number of arriving tasks/second
- $u$ server utilization (0..1): $u = r \times T_{ser}$
- $T_q$ average time/task in queue
- $T_{sys}$ Average time per task in the system
- $L_q$ average length of queue: $L_q = r \times T_q$
- $L_{sys}$ Average number of tasks in the system

$$T_{sys} = T_q + T_{ser}$$

$$L_{sys} = r \times T_{sys}$$

In textbook page 385 (3rd Edition: page 726)
Single Queue + Multiple Servers (Disks/Controllers)

I/O Modeling: \( M/M/m \) Queue

- I/O system with Markovian request arrival rate \( r \) tasks/sec
- A single queue serviced by \( m \) servers (disks + controllers) each with Markovian Service rate = \( 1/T_{ser} \)
  (and requests are distributed evenly among servers)

\[
T_q = T_{ser} \times \frac{u}{m (1 - u)}
\]

where \( u = \frac{r \times T_{ser}}{m} \)

- \( m \) number of servers
- \( T_{ser} \) average time to service a task
- \( u \) server utilization (0..1): \( u = \frac{r \times T_{ser}}{m} \)
- \( T_q \) average time/task in queue
- \( T_{sys} = T_{ser} + T_q \) Time in system (mean response time)
- \( L_q \) average length of queue: \( L_q = r \times T_q \)
- \( L_{sys} \) Average number of tasks in the system

Please Note:
We will use this simplified formula for M/M/m not the book version 4th Edition on page 388
(3rd Edition: page 729)

i.e as if the \( m \) servers are a single server with an effective service time of \( T_{ser}/m \)
I/O Queuing Performance: An M/M/1 Example

- A processor sends 40 disk I/O requests per second, requests & service are exponentially distributed, average disk service time = 20 ms

- On average:
  - What is the disk utilization $u$?
  - What is the average time spent in the queue, $T_q$?
  - What is the average response time for a disk request, $T_{sys}$?
  - What is the number of requests in the queue $L_q$? In system, $L_{sys}$?

- We have:
  - $r$: average number of arriving requests/second = 40
  - $T_{ser}$: average time to service a request = 20 ms (0.02s)

- We obtain:
  - $u$: server utilization: $u = r \times T_{ser} = 40/s \times 0.02s = 0.8$ or 80%
  - $T_q$: average time/request in queue = $T_{ser} \times u / (1 - u)$
    - $T_q = 20 \times 0.8/(1-0.8) = 20 \times 0.8/0.2 = 20 \times 4 = 80$ ms (0.08s)
  - $T_{sys}$: average time/request in system: $T_{sys} = T_q + T_{ser} = 80 + 20 = 100$ ms
  - $L_q$: average length of queue: $L_q = r \times T_q$
    - $L_q = 40/s \times 0.08s = 3.2$ requests in queue
  - $L_{sys}$: average # tasks in system: $L_{sys} = r \times T_{sys} = 40/s \times 0.1s = 4$
Example: Determining the System Performance Bottleneck (ignoring I/O queuing delays)

• Assume the following system components:
  – 500 MIPS CPU
  – 16-byte wide memory system with 100 ns cycle time
  – 200 MB/sec I/O bus
  – 20, 20 MB/sec SCSI-2 buses, with 1 ms controller overhead
  – 5 disks per SCSI bus: 8 ms seek, 7,200 RPMS, 6MB/sec (100 disks total)

• Other assumptions
  – All devices/system components can be used to 100% utilization
  – Average I/O request size is 16 KB
  – I/O Requests are assumed spread evenly on all disks.
  – OS uses 10,000 CPU instructions to process a disk I/O request
  – Ignore disk/controller queuing delays.
    (i.e. \( u = 1 \))
    (Since I/O queuing delays are ignored here 100% disk utilization is allowed)

• What is the average IOPS?
• What is the average I/O bandwidth?
• What is the average response time per IO operation?
The performance of I/O systems is determined by the system component with the lowest performance (the system performance bottleneck):

- **CPU**: \((500 \text{ MIPS})/(10,000 \text{ instructions per I/O}) = 50,000 \text{ IOPS}\)
  
  \[\text{CPU time per I/O} = \frac{10,000}{500,000,000} = 0.02 \text{ ms}\]

- **Main Memory**: \((16 \text{ bytes})/(100 \text{ ns} \times 16 \text{ KB per I/O}) = 10,000 \text{ IOPS}\)
  
  \[\text{Memory time per I/O} = \frac{1}{10,000} = 0.1 \text{ ms}\]

- **I/O bus**: \((200 \text{ MB/sec})/(16 \text{ KB per I/O}) = 12,500 \text{ IOPS}\)

- **SCSI-2**: \((20 \text{ buses})/((1 \text{ ms} + (16 \text{ KB})/(20 \text{ MB/sec})) \text{ per I/O}) = 11,111 \text{ IOPS}\)
  
  \[\text{SCSI bus time per I/O} = 1 \text{ ms} + \frac{16}{20} \text{ ms} = 1.8 \text{ ms}\]

- **Disks**: \((100 \text{ disks})/((8 \text{ ms} + 0.5/(7200 \text{ RPMS}) + (16 \text{ KB})/(6 \text{ MB/sec})) \text{ per I/O}) = 6700 \text{ IOPS}\)

\[T_{\text{disk}} = (8 \text{ ms} + 0.5/(7200 \text{ RPMS}) + (16 \text{ KB})/(6 \text{ MB/sec}) = 8 + 4.2 + 2.7 = 14.9 \text{ ms}\]

The disks limit the I/O performance to 6700 IOPS

- The average I/O bandwidth is 6700 IOPS x (16 KB/sec) = 107.2 MB/sec

- **Response Time Per I/O** = \(T_{\text{cpu}} + T_{\text{memory}} + T_{\text{scsi}} + T_{\text{disk}} = 0.02 + 0.1 + 1.8 + 14.9 = 16.82 \text{ ms}\)

---

**Example: Determining the System I/O Bottleneck**

(ignoring queuing delays)

- **Determining the system performance bottleneck**
  - CPU: \((500 \text{ MIPS})/(10,000 \text{ instructions per I/O}) = 50,000 \text{ IOPS}\)
    
    \[\text{CPU time per I/O} = \frac{10,000}{500,000,000} = 0.02 \text{ ms}\]
  - **Main Memory**: \((16 \text{ bytes})/(100 \text{ ns} \times 16 \text{ KB per I/O}) = 10,000 \text{ IOPS}\)
    
    \[\text{Memory time per I/O} = \frac{1}{10,000} = 0.1 \text{ ms}\]
  - **I/O bus**: \((200 \text{ MB/sec})/(16 \text{ KB per I/O}) = 12,500 \text{ IOPS}\)
  - **SCSI-2**: \((20 \text{ buses})/((1 \text{ ms} + (16 \text{ KB})/(20 \text{ MB/sec})) \text{ per I/O}) = 11,111 \text{ IOPS}\)
  
    \[\text{SCSI bus time per I/O} = 1 \text{ ms} + \frac{16}{20} \text{ ms} = 1.8 \text{ ms}\]
  - **Disks**: \((100 \text{ disks})/((8 \text{ ms} + 0.5/(7200 \text{ RPMS}) + (16 \text{ KB})/(6 \text{ MB/sec})) \text{ per I/O}) = 6700 \text{ IOPS}\)

\[T_{\text{disk}} = (8 \text{ ms} + 0.5/(7200 \text{ RPMS}) + (16 \text{ KB})/(6 \text{ MB/sec}) = 8 + 4.2 + 2.7 = 14.9 \text{ ms}\]

The disks limit the I/O performance to 6700 IOPS

- The average I/O bandwidth is 6700 IOPS x (16 KB/sec) = 107.2 MB/sec

- **Response Time Per I/O** = \(T_{\text{cpu}} + T_{\text{memory}} + T_{\text{scsi}} + T_{\text{disk}} = 0.02 + 0.1 + 1.8 + 14.9 = 16.82 \text{ ms}\)

---

Here: 1KBytes = 10^3 bytes, MByte = 10^6 bytes, 1 GByte = 10^9 bytes

**CMPE550 - Shaaban**

#93 Exam Review Fall 2015 11-23-2015
Example: Determining the I/O Bottleneck

Accounting for I/O Queue Time (M/M/m queue)

• Assume the following system components:
  – 500 MIPS CPU
  – 16-byte wide memory system with 100 ns cycle time
  – 200 MB/sec I/O bus
  – 20, 20 MB/sec SCSI-2 buses, with 1 ms controller overhead
  – 5 disks per SCSI bus: 8 ms seek, 7,200 RPMS, 6MB/sec (100 disks)

• Other assumptions
  – All devices used to 60% utilization (i.e. \( u = 0.6 \)).
  – Treat the I/O system as an M/M/m queue.
  – I/O Requests are assumed spread evenly on all disks.
  – Average I/O size is 16 KB
  – OS uses 10,000 CPU instructions to process a disk I/O request

• What is the average IOPS? What is the average bandwidth?
• Average response time per IO operation?

Here: \( m = 100 \) 

\[ \text{i.e I/O throughput} \]

\[ \text{Here: } 1\text{KBytes} = 10^3 \text{ bytes, MByte} = 10^6 \text{ bytes, 1 GByte} = 10^9 \text{ bytes} \]
Example: Determining the I/O Bottleneck

Accounting For I/O Queue Time (M/M/m queue)

- The performance of I/O systems is still determined by the system component with the lowest performance (the system performance bottleneck):
  - CPU: \((500 \text{ MIPS})/(10,000 \text{ instr. per I/O}) \times .6 = 30,000 \text{ IOPS}\)
    
    CPU time per I/O = \(10,000 / 500,000,000 = .02 \text{ ms}\)
  - Main Memory: \((16 \text{ bytes})/(100 \text{ ns} \times 16 \text{ KB per I/O}) \times .6 = 6,000 \text{ IOPS}\)
    
    Memory time per I/O = \(1/10,000 = .1 \text{ ms}\)
  - I/O bus: \((200 \text{ MB/sec})/(16 \text{ KB per I/O}) \times .6 = 12,500 \text{ IOPS}\)
  - SCSI-2: \((20 \text{ buses})/((1 \text{ ms} + (16 \text{ KB})/(20 \text{ MB/sec})) \times .6 = 6,666.6 \text{ IOPS}\)
    
    SCSI bus time per I/O = \(1 \text{ ms} + 16/20 \text{ ms} = 1.8 \text{ ms}\)
  - Disks: \((100 \text{ disks})/((8 \text{ ms} + 0.5/(7200 \text{ RPMS}) + (16 \text{ KB})/(6 \text{ MB/sec})) \times .6 = 6,700 \times .6 = 4020 \text{ IOPS}\)
    
    \(T_{\text{ser}} = (8 \text{ ms} + 0.5/(7200 \text{ RPMS}) + (16 \text{ KB})/(6 \text{ MB/sec}) = 8 + 4.2 + 2.7 = 14.9 \text{ ms}\)

- The disks limit the I/O performance to \(r = 4020 \text{ IOPS}\)
- The average I/O bandwidth is \(4020 \text{ IOPS} \times (16 \text{ KB/sec}) = 64.3 \text{ MB/sec}\)
- \(T_q = T_{\text{ser}} \times u / [m(1-u)] = 14.9 \text{ ms} \times .6 / [100 \times .4] = .22 \text{ ms}\)
- **Response Time** = \(T_{\text{ser}} + T_{q} + T_{\text{cpu}} + T_{\text{memory}} + T_{\text{scsi}} = 14.9 + .22 + .02 + .1 + 1.8 = 17.04 \text{ ms}\)