EECC551 Review

• Instruction In-order Pipeline Performance.
• Instruction-Level Parallelism (ILP).
  – Loop-unrolling
• Dynamic Pipeline Scheduling.
  – The Tomasulo Algorithm
• Dynamic Branch Prediction.
• Multiple Instruction Issue (CPI < 1): Superscalar vs. VLIW
• Dynamic Hardware-Based Speculation
• Loop-Level Parallelism (LLP).
  – Making loop iterations parallel
• Cache & Memory Performance.
• I/O System Performance.
# Simple MIPS In-Order Pipelined Integer Instruction Processing

MIPS Pipeline Stages:
- **IF** = Instruction Fetch
- **ID** = Instruction Decode
- **EX** = Execution
- **MEM** = Memory Access
- **WB** = Write Back

<table>
<thead>
<tr>
<th>Instruction Number</th>
<th>Clock Number</th>
<th>Time in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

First instruction, I Completed

Last instruction, I+4 completed

Time to fill the pipeline
Pipeline Hazards

• Hazards are situations in pipelining which prevent the next instruction in the instruction stream from executing during the designated clock cycle.

• Hazards reduce the ideal speedup gained from pipelining and are classified into three classes:
  – *Structural hazards*: Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.
  – *Data hazards*: Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
  – *Control hazards*: Arise from the pipelining of conditional branches and other instructions that change the PC
Performance of Pipelines with Stalls

- Hazards in pipelines may make it necessary to stall the pipeline by one or more cycles and thus degrading performance from the ideal CPI of 1.

\[
\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}
\]

- Where stall cycles per instruction is:
  
  Structural hazard stalls per instruction +  
  Data hazard stalls per instruction +  
  Control hazard stalls per instruction +  
  Memory stalls per instruction +  
  TLB-miss stalls per instruction
A set of instructions that depend on the DADD result uses forwarding paths to avoid the data hazard.
Load/Store Forwarding Example

Forwarding of operand required by stores during MEM
Data Hazard Classification

Given two instructions $I$, $J$, with $I$ occurring before $J$ in an instruction stream:

- **RAW (read after write):** A true data dependence
  $J$ tried to read a source before $I$ writes to it, so $J$ incorrectly gets the old value.

- **WAW (write after write):** A name dependence
  $J$ tries to write an operand before it is written by $I$
  The writes end up being performed in the wrong order.

- **WAR (write after read):** A name dependence
  $J$ tries to write to a destination before it is read by $I$,
  so $I$ incorrectly gets the new value.

- **RAR (read after read):** Not a hazard.
Data Hazard Classification

$I$ (Write) → $J$ (Read) → Shared Operand → Read after Write (RAW)

$I$ (Write) → $I$ (Write) → Shared Operand → Write after Write (WAW)

$I$ (Read) → $J$ (Write) → Shared Operand → Write after Read (WAR)

$I$ (Read) → $J$ (Read) → Shared Operand → Read after Read (RAR) not a hazard
Data Hazards Requiring Stall Cycles

• In some code sequence cases, potential data hazards cannot be handled by bypassing. For example:

  LD R1, 0 (R2)
  DSUB R4, R1, R5
  AND R6, R1, R7
  OR R8, R1, R9

• The LD (load double word) instruction has the data in clock cycle 4 (MEM cycle).

• The DSUB instruction needs the data of R1 in the beginning of that cycle.

• Hazard prevented by hardware pipeline interlock causing a stall cycle.
LD R1,0(R1)

DSUB R4, R1,R5

AND R6, R1, R7

OR R8, R1, R9

The load interlock causes a stall to be inserted at clock cycle 4, delaying the SUB instruction and those that follow by one cycle.
Compiler Instruction Scheduling Example

• For the code sequence:
  
  \[ a = b + c \]
  
  \[ d = e - f \]

• Assuming loads have a latency of one clock cycle, the following code or pipeline compiler schedule eliminates stalls:

Original code with stalls:

- LD    Rb,b
- LD    Rc,c
- DADD  Ra,Rb,Rc
- SD    Ra,a
- LD    Re,e
- LD    Rf,f
- DSUB  Rd,Re,Rf
- SD    Rd,d

Scheduled code with no stalls:

- LD    Rb,b
- LD    Rc,c
- DADD  Ra,Rb,Rc
- LD    Re,e
- DADD  Ra,Rb,Rc
- LD    Rf,f
- SD    Ra,a
- DSUB  Rd,Re,Rf
- SD    Rd,d
Control Hazards

• When a conditional branch is executed it may change the PC and, without any special measures, leads to stalling the pipeline for a number of cycles until the branch condition is known.

• In current MIPS pipeline, the conditional branch is resolved in the MEM stage resulting in three stall cycles as shown below:

<table>
<thead>
<tr>
<th>Branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch successor</td>
<td>IF</td>
<td>ID</td>
<td>stall</td>
<td>stall</td>
<td>IF</td>
</tr>
<tr>
<td>Branch successor + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch successor + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch successor + 3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch successor + 4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch successor + 5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Three clock cycles are wasted for every branch for current MIPS pipeline
Modified MIPS Pipeline: Conditional Branches Completed in ID Stage

The stall from branch hazards can be reduced by moving the zero test and branch target calculation into the ID phase of the pipeline.
Compile-Time Reduction of Branch Penalties

- One scheme discussed earlier is to flush or freeze the pipeline by whenever a conditional branch is decoded by holding or deleting any instructions in the pipeline until the branch destination is known (zero pipeline registers, control lines).

- Another method is to predict that the branch is not taken where the state of the machine is not changed until the branch outcome is definitely known. Execution here continues with the next instruction; stall occurs here when the branch is taken.

- Another method is to predict that the branch is taken and begin fetching and executing at the target; stall occurs here if the branch is not taken.

- Delayed Branch: An instruction following the branch in a branch delay slot is executed whether the branch is taken or not.
### Delayed Branch Example

<table>
<thead>
<tr>
<th>Untaken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch delay instruction ((i + 1))</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 3)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 4)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Taken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch delay instruction ((i + 1))</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

The behavior of a delayed branch is the same whether or not the branch is taken.
Pipeline Performance Example

• Assume the following MIPS instruction mix:

<table>
<thead>
<tr>
<th>Type</th>
<th>Frequency</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith/Logic</td>
<td>40%</td>
<td>30% of which 25% are followed immediately by</td>
</tr>
<tr>
<td></td>
<td></td>
<td>an instruction using the loaded value</td>
</tr>
<tr>
<td>Load</td>
<td>30%</td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>20% of which 45% are taken</td>
</tr>
</tbody>
</table>

• What is the resulting CPI for the pipelined MIPS with forwarding and branch address calculation in ID stage when using a branch not-taken scheme?

• CPI = Ideal CPI + Pipeline stall clock cycles per instruction

\[
\begin{align*}
\text{CPI} &= 1 + \text{stalls by loads} + \text{stalls by branches} \\
&= 1 + 0.3 \times 0.25 \times 1 + 0.2 \times 0.45 \times 1 \\
&= 1 + 0.075 + 0.09 \\
&= 1.165
\end{align*}
\]
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.
• 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.
  – 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.

(In Appendix A)
Extending The MIPS In-order Integer Pipeline: Multiple Outstanding Floating Point Operations

Latency = 6
Initiation Interval = 1
Pipelined

Latency = 3
Initiation Interval = 1
Pipelined

Latency = 0
Initiation Interval = 1

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

A pipeline that supports multiple outstanding FP operations.

(In Appendix A)
Pipeline Characteristics With FP

• Instructions are still processed in-order in IF, ID, EX at the rate of instruction per cycle.
• Longer RAW hazard stalls likely due to long FP latencies.
• Structural hazards possible due to varying instruction times and FP latencies:
  – FP unit may not be available; divide in this case.
  – MEM, WB reached by several instructions simultaneously.
• WAW hazards can occur since it is possible for instructions to reach WB out-of-order.
• WAR hazards impossible, since register reads occur in-order in ID.
• Instructions are allowed to complete out-of-order requiring special measures to enforce precise exceptions.
FP Code RAW Hazard Stalls Example
(with full data forwarding in place)

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

6 stall cycles which equals latency of FP add functional unit

Third stall due to structural hazard in MEM stage

(In Appendix A)
FP Code Structural Hazards Example

- MULTD F0, F4, F6
  - IF ID M1 M2 M3 M4 M5 M6 M7 MEM WB
  - . . . (integer)
  - IF ID EX MEM WB
  - . . . (integer)
  - IF ID EX MEM WB

- ADDD F2, F4, F6
  - IF ID A1 A2 A3 A4 MEM WB
  - . . . (integer)
  - IF ID EX MEM WB
  - . . . (integer)
  - IF ID EX MEM WB

- LD F2, 0(R2)
  - IF ID EX MEM WB

(In Appendix A)
Pipelining and Exploiting Instruction-Level Parallelism (ILP)

- Pipelining increases performance by overlapping the execution of independent instructions.

- The CPI of a real-life pipeline is given by:

  Pipeline CPI = Ideal Pipeline CPI + Structural Stalls + RAW Stalls
  + WAR Stalls + WAW Stalls + Control Stalls + Memory Stalls + TLB stalls

- A *basic instruction block* is a straight-line code sequence with no branches in, except at the entry point, and no branches out except at the exit point of the sequence.

- The amount of parallelism in a basic block is limited by instruction dependence present and size of the basic block.

- In typical integer code, dynamic branch frequency is about 15% (average basic block size of 7 instructions).

(In Chapter 3.1)
Increasing Instruction-Level Parallelism

• 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.

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

  ```
  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.

(In 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
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.

(In Chapter 4.1)
MIPS FP Latency Assumptions Used In Chapter 4

- 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>

(In Chapter 4.1)
Loop Unrolling Example (continued)

- This loop code is executed on the MIPS pipeline as follows:

  No scheduling

<table>
<thead>
<tr>
<th>Instruction</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>

  With delayed branch scheduling

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0, 0(R1)</td>
<td></td>
</tr>
<tr>
<td>DADDUI R1, R1, # -8</td>
<td>6</td>
</tr>
<tr>
<td>ADD.D F4, F0, F2</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>S.D F4,8(R1)</td>
<td>10</td>
</tr>
</tbody>
</table>

6 cycles per iteration

10/6 = 1.7 times faster

(In Chapter 4.1)
Loop Unrolling Example (continued)

- The resulting loop code when four copies of the loop body are unrolled without reuse of registers:

  No scheduling

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

  Three branches and three decrements of R1 are eliminated.

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

  The loop runs in 28 assuming each L.D has 1 stall cycle, each ADD.D has 2 stall cycles, the DADDUI 1 stall, the branch 1 stall cycles, or 7 cycles for each of the four elements.
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 3.5 clock cycles per element compared to 6.8 before scheduling and 6 when scheduled but unrolled.

Unrolling the loop exposed more computation that can be scheduled to minimize stalls.

(In Chapter 4.1)
Loop Unrolling Requirements

• In the loop unrolling example, the following guidelines were followed:
  – Determine that it was legal to move S.D after DADDUI and BNE; find the S.D offset.
  – Determine that unrolling the loop would be useful by finding that the loop iterations were independent.
  – Use different registers to avoid constraints of using the same registers (WAR, WAW).
  – Eliminate extra tests and branches and adjust loop maintenance code.
  – Determine that loads and stores can be interchanged by observing that they are independent from different loops.
  – Schedule the code, preserving any dependencies needed to give the same result as the original code.

(In Chapter 4.1)
Instruction Dependencies

• Determining instruction dependencies is important for pipeline scheduling and to determine the amount of parallelism in the program to be exploited.

• If two instructions are parallel, they can be executed simultaneously in the pipeline without causing stalls; assuming the pipeline has sufficient resources.

• Instructions that are dependent are not parallel and cannot be reordered.

• Instruction dependencies are classified as:
  - Data dependencies
  - Name dependencies
  - Control dependencies

(In Chapter 3.1)
Instruction Data Dependencies

An instruction \( j \) is data dependent on another instruction \( i \) if:

- Instruction \( i \) produces a result used by instruction \( j \), resulting in a direct RAW hazard, or
- Instruction \( j \) is data dependent on instruction \( k \) and instruction \( k \) is data dependent on instruction \( i \) which implies a chain of RAW hazard between the two instructions.

Example: The arrows indicate data dependencies and point to the dependent instruction which must follow and remain in the original instruction order to ensure correct execution.

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

(In Chapter 3.1)
Instruction Name Dependencies

- A name dependence occurs when two instructions use the same register or memory location, called a name.
- No flow of data exist between the instructions involved in the name dependency.
- If instruction $i$ precedes instruction $j$ then two types of name dependencies can occur:
  - An antidependence occurs when $j$ writes to a register or memory location and $i$ reads and instruction $i$ is executed first. This corresponds to a WAR hazard.
  - An output dependence occurs when instruction $i$ and $j$ write to the same register or memory location resulting in a WAW hazard and instruction execution order must be observed.
Renaming the registers used for each copy of the loop body are renamed, only true dependencies remain:

| Loop: L.D   | F0, 0 (R1) |
| ADD.D   | F4, F0, F2 |
| S.D | F4, 0(R1) |
| L.D   | F0, -8(R1) |
| ADD.D   | F4, F0, F2 |
| S.D | F4, -8(R1) |
| L.D   | F0, -16(R1) |
| ADD.D   | F4, F0, F2 |
| S.D | F4, -16 (R1) |
| L.D   | F0, -24 (R1) |
| ADD.D   | F4, F0, F2 |
| S.D | F4, -24(R1) |
| DADDUI   | R1, R1, # -32 |
| BNE   | R1, R2, Loop |

(In Chapter 4.1)
Control Dependencies

- Determines the ordering of an instruction with respect to a branch instruction.
- Every instruction except in the first basic block of the program is control dependent on some set of branches.
- An instruction which is control dependent on a branch cannot be moved before the branch.
- 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).
- 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:

```java
if (p1) {
    S1; // S1 is control dependent on p1
}
If (p2) {
    S2; // S2 is control dependent on p2 but not on p1
}
```

(In Chapter 3.1)
Control Dependence Example

The unrolled loop code with the branches still in place is shown here.

Branch conditions are complemented here to allow the fall-through to execute another loop.

BEQZ instructions prevent the overlapping of iterations for scheduling optimizations.

Moving the instructions requires a change in the control dependencies present.

Removing the branches changes the control dependencies present and makes optimizations possible.

Loop:

L.D F0, 0 (R1)
ADD.D F4, F0, F2
S.D F4,0 (R1)
DADDUI R1, R1, # -8
BNE R1, R2, exit
L.D F6, 0 (R1)
ADD.D F8, F6, F2
S.D F8, 0 (R1)
DADDUI R1, R1, # -8
BNE R1, R2, exit
L.D F10, 0 (R1)
ADD.D F12, F10, F2
S.D F12,0 (R1)
DADDUI R1, R1, # -8
BNE R1, R2,exit
L.D F14, 0 (R1)
ADD.D F16, F14, F2
S.D F16, 0 (R1)
SUBI R1, R1, # -8
BNE R1, R2,Loop

exit:
Reduction of Data Hazards Stalls with Dynamic Scheduling

• So far we have dealt with data hazards in instruction pipelines by:
  – Result forwarding and bypassing to reduce latency and hide or reduce the effect of true data dependence.
  – Hazard detection hardware to stall the pipeline starting with the instruction that uses the result.
  – Compiler-based static pipeline scheduling to separate the dependent instructions minimizing actual hazards and stalls in scheduled code.

• Dynamic scheduling:
  – Uses a hardware-based mechanism to rearrange instruction execution order to reduce stalls at runtime.
  – Enables handling some cases where dependencies are unknown at compile time.
  – Similar to the other pipeline optimizations above, a dynamically scheduled processor cannot remove true data dependencies, but tries to avoid or reduce stalling.

(In Appendix A.8, Chapter 3.2)
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.
  - Read operands: Wait until data hazard conditions, if any, are resolved, then read operands when available.
    (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 dynamic scheduling approaches exist:
- Dynamic scheduling with a Scoreboard used first in CDC6600
- The Tomasulo approach pioneered by the IBM 360/91

(In Appendix A.8, Chapter 3.2)
Dynamic Scheduling With A Scoreboard

- The scoreboard is a hardware mechanism that maintains an execution rate of one instruction per cycle by executing an instruction as soon as its operands are available and no hazard conditions prevent it.
- It replaces ID, EX, WB with four stages: ID1, ID2, EX, WB
- Every instruction goes through the scoreboard where a record of data dependencies is constructed (corresponds to instruction issue).
- A system with a scoreboard is assumed to have several functional units with their status information reported to the scoreboard.
- If the scoreboard determines that an instruction cannot execute immediately it executes another waiting instruction and keeps monitoring hardware units status and decide when the instruction can proceed to execute.
- The scoreboard also decides when an instruction can write its results to registers (hazard detection and resolution is centralized in the scoreboard).

(In Appendix A.8)
The basic structure of a MIPS processor with a scoreboard

(In Appendix A.8)
Instruction Execution Stages with A Scoreboard

1 **Issue (ID1):** If a functional unit for the instruction is available, the scoreboard issues the instruction to the functional unit and updates its internal data structure; **structural** and **WAW** hazards are resolved here. (this replaces part of **ID** stage in the conventional MIPS pipeline).

2 **Read operands (ID2):** The scoreboard monitors the availability of the source operands. A source operand is available when no earlier active instruction will write it. When all source operands are available the scoreboard tells the functional unit to *read* all operands from the registers (no forwarding supported) and start execution (**RAW** hazards resolved here dynamically). This completes ID.

3 **Execution (EX):** The functional unit starts execution upon receiving operands. When the results are ready it notifies the scoreboard (replaces **EX, MEM** in MIPS).

4 **Write result (WB):** Once the scoreboard senses that a functional unit completed execution, it checks for **WAR** hazards and stalls the completing instruction if needed otherwise the write back is completed.

(In Appendix A.8)
Three Parts of the Scoreboard

1  **Instruction status:** Which of 4 steps the instruction is in.

2  **Functional unit status:** Indicates the state of the functional unit (FU). Nine fields for each functional unit:
   - **Busy** Indicates whether the unit is busy or not
   - **Op** Operation to perform in the unit (e.g., + or –)
   - **Fi** Destination register
   - **Fj, Fk** Source-register numbers
   - **Qj, Qk** Functional units producing source registers Fj, Fk
   - **Rj, Rk** Flags indicating when Fj, Fk are ready
     (set to Yes after operand is available to read)

3  **Register result status:** Indicates which functional unit will write to each register, if one exists. Blank when no pending instructions will write that register.

(In Appendix A.8)
## A Scoreboard Example

The following code is run on the MIPS with a scoreboard given earlier with:

```
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
```

<table>
<thead>
<tr>
<th>Functional Unit (FU)</th>
<th># of FUs</th>
<th>EX Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Floating Point Multiply</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Floating Point add</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Floating point Divide</td>
<td>1</td>
<td>40</td>
</tr>
</tbody>
</table>

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

All functional units are not pipelined
### Scoreboard Example: Cycle 62

#### Instruction status

<table>
<thead>
<tr>
<th>Instruction</th>
<th>$j$</th>
<th>$k$</th>
<th>Read</th>
<th>Execution</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F6</td>
<td>34+</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>L.D F2</td>
<td>45+</td>
<td>R3</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>MUL.D F0</td>
<td>F2</td>
<td>F4</td>
<td>6</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>SUB.D F8</td>
<td>F6</td>
<td>F2</td>
<td>7</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>DIV.D F10</td>
<td>F0</td>
<td>F6</td>
<td>8</td>
<td>21</td>
<td>61</td>
</tr>
<tr>
<td>ADD.D F6</td>
<td>F8</td>
<td>F2</td>
<td>13</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

#### Functional unit status

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>$S_i$</th>
<th>$S_j$</th>
<th>$F_i$</th>
<th>$F_j$</th>
<th>$Q_i$</th>
<th>$Q_j$</th>
<th>$R_i$</th>
<th>$R_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Integer</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></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>
<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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Add</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Divide</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Register result status

<table>
<thead>
<tr>
<th>Clock</th>
<th>FU</th>
</tr>
</thead>
<tbody>
<tr>
<td>62</td>
<td></td>
</tr>
</tbody>
</table>

---

- **We have:**
  - In-order issue,
  - Out-of-order execute and commit

---
Dynamic Scheduling: The Tomasulo Algorithm

• Developed at IBM and first implemented in IBM’s 360/91 mainframe in 1966, about 3 years after the debut of the scoreboard in the CDC 6600.

• Dynamically schedule the pipeline in hardware to reduce stalls.

• Differences between IBM 360 & CDC 6600 ISA.
  – IBM has only 2 register specifiers/instr vs. 3 in CDC 6600.
  – IBM has 4 FP registers vs. 8 in CDC 6600.

• Current CPU architectures that can be considered descendants of the IBM 360/91 which implement and utilize a variation of the Tomasulo Algorithm include:

RISC CPUs: Alpha 21264, HP 8600, MIPS R12000, PowerPC G4
RISC-core x86 CPUs: AMD Athlon, Pentium III, 4, Xeon ….
Tomasulo Algorithm Vs. Scoreboard

- Control & buffers *distributed* with Function Units (FU) Vs. centralized in Scoreboard:
  - FU buffers are called “reservation stations” which have pending instructions and operands and other instruction status info.
  - 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 to reservation stations (RS) that will supply the value later:
  - This process is called *register renaming*.
  - Avoids WAR, WAW hazards.
  - Allows for *hardware-based* 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) to FUs from RSs, *not through registers*, over *Common Data Bus (CDB)* that broadcasts results to all FUs.

- Loads and Stores are treated as FUs with RSs as well.

- Integer instructions can go past branches, allowing FP ops beyond basic block in FP queue.

(In Chapter 3.2)
Dynamic Scheduling: The Tomasulo Approach

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

(In Chapter 3.2)
Reservation Station 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 and FU are busy.

• **Register result status:**  Qi Indicates which functional unit will write each register, if one exists.
  – Blank (or 0) when no pending instructions exist that will write to that register.

(In Chapter 3.2)
### Three Stages of Tomasulo Algorithm

1. **Issue**: Get instruction from pending Instruction Queue.
   - Instruction issued to a free reservation station (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).

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).

3. **Write result (WB)**: Finish execution.
   - Write result on Common Data Bus to all awaiting units
   - 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.

(In Chapter 3.2)
Tomasulo Approach Example

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

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

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

Pipelined 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

Real Data Dependence (RAW) →
Anti-dependence (WAR) ←
Output Dependence (WAW) ↔
### Tomasulo Example: Cycle 57

(vs 62 cycles for scoreboard)

<table>
<thead>
<tr>
<th>Instruction status</th>
<th>Execution complete</th>
<th>Write Result</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction</strong></td>
<td><strong>j</strong></td>
<td><strong>k</strong></td>
<td><strong>Issue</strong></td>
<td><strong>Result</strong></td>
</tr>
<tr>
<td>L.D F6 34+ R2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>Load1</td>
</tr>
<tr>
<td>L.D F2 45+ R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2</td>
</tr>
<tr>
<td>MUL.D F0 F4</td>
<td>3</td>
<td>15</td>
<td>16</td>
<td>Load3</td>
</tr>
<tr>
<td>SUB.D F8 F6 F2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>DIV.D F10 F6 F2</td>
<td>5</td>
<td>56</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>ADD.D F6 F8 F2</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

#### Reservation Stations

<table>
<thead>
<tr>
<th>S1</th>
<th>S2</th>
<th>RS for j</th>
<th>RS for k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Instruction Block done

- Again we have:
- In-order issue,
- Out-of-order execution, completion

#### 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>
Tomasulo Loop Example

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 Multiply takes 4 clocks.
- Assume first load takes 8 clocks (possibly due to a cache miss), second load takes 4 clocks (hit).
- Assume R1 = 80 initially.
- Assume branch is predicted taken.
- We’ll go over the execution of the first two loop iterations.

(In Chapter 3.2)
## Loop Example Cycle 21

### Instruction status

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>iteration</th>
<th>Issue</th>
<th>Write</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F0</td>
<td>0</td>
<td>R1</td>
<td>1</td>
<td>9</td>
<td>10</td>
<td>Load1</td>
</tr>
<tr>
<td>MUL.D</td>
<td>F4</td>
<td>F0</td>
<td>F2</td>
<td>1</td>
<td>2</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>S.D</td>
<td>F4</td>
<td>0</td>
<td>R1</td>
<td>1</td>
<td>3</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>L.D</td>
<td>F0</td>
<td>0</td>
<td>R1</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>MUL.D</td>
<td>F4</td>
<td>F0</td>
<td>F2</td>
<td>2</td>
<td>7</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>S.D</td>
<td>F4</td>
<td>0</td>
<td>R1</td>
<td>2</td>
<td>8</td>
<td>20</td>
<td>21</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>L.D</td>
<td>F0</td>
<td>0(R1)</td>
<td></td>
<td></td>
<td>F0, 0(R1)</td>
</tr>
<tr>
<td>0</td>
<td>Add2</td>
<td>No</td>
<td>MUL.D</td>
<td>F4</td>
<td>F0,F2</td>
<td></td>
<td></td>
<td>S.D F4, 0(R1)</td>
</tr>
<tr>
<td>0</td>
<td>Add3</td>
<td>No</td>
<td>S.D</td>
<td>F4</td>
<td>0(R1)</td>
<td></td>
<td></td>
<td>DADDUI</td>
</tr>
<tr>
<td>0</td>
<td>Mult1</td>
<td>Yes</td>
<td>MULTD</td>
<td></td>
<td></td>
<td>R(F2)</td>
<td></td>
<td>Load3</td>
</tr>
<tr>
<td>0</td>
<td>Mult2</td>
<td>No</td>
<td>BNE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R1, R1, #8, loop</td>
</tr>
</tbody>
</table>

### Register result status

<table>
<thead>
<tr>
<th>Clock</th>
<th>R1</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>21</td>
<td>56</td>
<td>Qi</td>
<td>Load3</td>
<td>Mult1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**EECC551 - Shaaban**

#52 Exam Review Fall 2002 10-31-2002
Branch Target Buffer (BTB)

• Effective branch prediction requires the target of the branch at an early pipeline stage.

• One can use additional adders to calculate the target, as soon as the branch instruction is decoded. This would mean that one has to wait until the ID stage before the target of the branch can be fetched, taken branches would be fetched with a one-cycle penalty (this was done in the enhanced MIPS pipeline Fig A.24).

• To avoid this problem one can use a Branch Target Buffer (BTB). A typical BTB is an associative memory where the addresses of taken branch instructions are stored together with their target addresses.

• Some designs store \( n \) prediction bits as well, implementing a combined BTB and BHT.

• Instructions are fetched from the target stored in the BTB in case the branch is predicted-taken and found in BTB. After the branch has been resolved the BTB is updated. If a branch is encountered for the first time a new entry is created once it is resolved.

• Branch Target Instruction Cache (BTIC): A variation of BTB which caches also the code of the branch target instruction in addition to its address. This eliminates the need to fetch the target instruction from the instruction cache or from memory.
Basic Branch Target Buffer (BTB)

PC of instruction to fetch

Look up

Predicted PC

Number of entries in branch-target buffer

A branch-target buffer.

Yes: then instruction is branch and predicted PC should be used as the next PC

No: instruction is not predicted to be branch. Proceed normally

Branch predicted taken or untaken
One-Level Bimodal Dynamic Branch Predictors

• One-level or bimodal branch prediction uses only one level of branch history.
• These mechanisms usually employ a table which is indexed by lower bits of the branch address.
• The table entry consists of $n$ history bits, which form an $n$-bit automaton or saturating counters.
• Smith proposed such a scheme, known as the Smith algorithm, that uses a table of two-bit saturating counters.
• One rarely finds the use of more than 3 history bits in the literature.
• Two variations of this mechanism:
  – Decode History Table: Consists of directly mapped entries.
  – Branch History Table (BHT): Stores the branch address as a tag. It is associative and enables one to identify the branch instruction during IF by comparing the address of an instruction with the stored branch addresses in the table (similar to BTB).
One-Level Bimodal Branch Predictors

Decode History Table (DHT)

- High bit determines branch prediction
  - 0 = Not Taken
  - 1 = Taken

N Low Bits of Branch Address

Table has $2^N$ entries.

Example:

For $N = 12$

Table has $2^N = 2^{12}$ entries

$= 4096 = 4k$ entries

Number of bits needed = $2 \times 4k = 8k$ bits
Correlating Two-Level Dynamic GAp Branch Predictors

- Improve branch prediction by looking not only at the history of the branch in question but also at that of other branches using two levels of branch history.
- Uses two levels of branch history:
  - First level (global):
    - Record the global pattern or history of the $m$ most recently executed branches as taken or not taken. Usually an $m$-bit shift register.
  - Second level (per branch address):
    - $2^m$ prediction tables, each table entry has $n$ bit saturating counter.
    - The branch history pattern from first level is used to select the proper branch prediction table in the second level.
    - The low $N$ bits of the branch address are used to select the correct prediction entry within a the selected table, thus each of the $2^m$ tables has $2^N$ entries and each entry is 2 bits counter.
    - Total number of bits needed for second level = $2^m \times n \times 2^N$ bits
- In general, the notation: $(m,n)$ GAp predictor means:
  - Record last $m$ branches to select between $2^m$ history tables.
  - Each second level table uses $n$-bit counters (each table entry has $n$ bits).
- Basic two-bit single-level Bimodal BHT is then a $(0,2)$ predictor.
Organization of A Correlating Two-level GAp (2,2) Branch Predictor

First Level

- 2-bit global branch history
- Low 4 bits of address
- Selects correct entry in table

Second Level

- High bit determines branch prediction
  - 0 = Not Taken
  - 1 = Taken
- Low 4 bits of address selects correct table
- XX prediction

\[ m = \# \text{ of branches tracked in first level} = 2 \]
Thus \[ 2^m = 2^2 = 4 \] tables in second level

\[ N = \# \text{ of low bits of branch address used} = 4 \]
Thus each table in 2nd level has \[ 2N = 24 = 16 \] entries

\[ n = \# \text{ number of bits of 2nd level table entry} = 2 \]

Number of bits for 2nd level
\[ = 4 \times 2 \times 16 = 128 \text{ bits} \]

A (2,2) branch-prediction buffer uses a two-bit global history to choose from among four predictors for each branch address.
Multiple Instruction Issue: CPI < 1

- To improve a pipeline’s CPI to be better [less] than one, and to utilize ILP better, a number of independent instructions have to be issued in the same pipeline cycle.

- Multiple instruction issue processors are of two types:
  - **Superscalar:** A number of instructions (2-8) is issued in the same cycle, scheduled statically by the compiler or dynamically (Tomasulo).
    - PowerPC, Sun UltraSparc, Alpha, HP 8000 ...
  - **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).
    - Joint HP/Intel agreement (Itanium, Q4 2000).
    - Intel Architecture-64 (IA-64) 64-bit address:
      - Explicitly Parallel Instruction Computer (EPIC): Itanium.

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

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

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two-issue statically scheduled pipeline in operation
FP instructions assumed to be adds
Unrolled Loop Example for Scalar Pipeline

1 Loop: L.D F0,0 (R1)
2     L.D F6,−8 (R1)
3     L.D F10,−16 (R1)
4     L.D F14,−24 (R1)
5     ADD.D F4,F0,F2
6     ADD.D F8,F6,F2
7     ADD.D F12,F10,F2
8     ADD.D F16,F14,F2
9     S.D F4,0 (R1)
10    S.D F8,−8 (R1)
11    DADDUI R1,R1,#−32
12    S.D F12,−16 (R1)
13    BNE R1,R2,LOOP
14    S.D F16,8 (R1) ; 8−32 = −24

14 clock cycles, or 3.5 per iteration
### Loop Unrolling in 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>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F0,0(R1)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>L.D F6,-8(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>3</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>ADD.D F8,F6,F2</td>
<td>4</td>
</tr>
<tr>
<td>L.D F14,-24(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td>5</td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>ADD.D F16,F14,F2</td>
<td>6</td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>ADD.D F20,F18,F2</td>
<td>7</td>
</tr>
<tr>
<td>S.D F8,-8(R1)</td>
<td></td>
<td></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 (+1 due to SS)**
- **12 clocks, or 2.4 clocks per iteration (1.5X)**
- **7 issue slots wasted**
## 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>L.D F6,-8(R1)</td>
<td>L.D F14,-24(R1)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td>ADD.D F20,F18,F2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>ADD.D F26,F2</td>
<td>S.D F8,-8(R1)</td>
<td>ADD.D F20,F20,F2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>S.D F12,-16(R1)</td>
<td>S.D F16,-24(R1)</td>
<td>S.D F24,F24,F2</td>
<td>S.D F28,F26,F2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>S.D F20,24(R1)</td>
<td>S.D F24,16(R1)</td>
<td>DADDUI R1,R1,#-56</td>
<td>S.D F28,8(R1)</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>S.D F28,8(R1)</td>
<td>BNE R1, R2, LOOP</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
Average: 2.5 ops per clock, 50% efficiency
Note: Needs more registers in VLIW (15 vs. 6 in Superscalar)
Hardware Support for Extracting More Parallelism

• Compiler ILP techniques (loop-unrolling, software Pipelining etc.) are not effective to uncover maximum ILP when branch behavior is not well known at compile time.

• Hardware ILP techniques:
  – **Conditional or Predicted Instructions:** An extension to the instruction set with instructions that turn into no-ops if a condition is not valid at run time.
  – **Speculation:** An instruction is executed before the processor knows that the instruction should execute to avoid control dependence stalls:
    • **Static Speculation** by the compiler with hardware support:
      – The compiler labels an instruction as speculative and the hardware helps by ignoring the outcome of incorrectly speculated instructions.
      – Conditional instructions provide limited speculation.
    • **Dynamic Hardware-based Speculation:**
      – Uses dynamic branch-prediction to guide the speculation process.
      – Dynamic scheduling and execution continued passed a conditional branch in the predicted branch direction.
Dynamic Hardware-Based Speculation

• Combines:
  – Dynamic hardware-based branch prediction
  – Dynamic Scheduling: of multiple instructions to execute out of order.

• 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 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).

  • Addition of commit (retire or re-ordering) stage and forcing instructions to commit in their order in the code (i.e to write results to registers or memory).
  • Precise exceptions are possible since instructions must commit in order.
Hardware-Based Speculation

Speculative Execution + Tomasulo’s Algorithm
Four Steps of Speculative Tomasulo Algorithm

1. **Issue** — Get an instruction from FP Op 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** — 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** — Finish execution (WB)
   
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.

4. **Commit** — 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 (sometimes called “graduation”)

   ⇒ Instructions issue in order, execute (EX), write result (WB) out of order, but must commit in order.
Static Compiler Optimization Techniques

• We already examined the following static compiler techniques aimed at improving pipelined CPU performance:
  – Static pipeline scheduling (in ch 4.1).
  – Loop unrolling (ch 4.1).
  – Static branch prediction (in ch 4.2).
  – Static multiple instruction issue: VLIW (in ch 4.3).
  – Conditional or predicted instructions (in ch 4.5)

• Here we examine two additional static compiler-based techniques (in ch 4.4):
  – Loop-Level Parallelism (LLP) analysis:
    • Detecting and enhancing loop iteration parallelism
      – GCD test.
    – Software pipelining (Symbolic loop unrolling).
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.

  e.g. in
  for (i=1; i<=1000; i++)
      x[i] = x[i] + s;

  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.

  ⇒Thus loop iterations are parallel (or independent from each other).

• Loop-carried Dependence: A data dependence between different loop iterations (data produced in earlier iteration used in a later one).

• LLP analysis is normally done at the source code level or close to it since assembly language and target machine code generation introduces a loop-carried name dependence in the registers used for addressing and incrementing.

• 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).
  
  \( \Rightarrow \) does not prevent loop iteration parallelism.

- **S1** uses a value computed by S1 in an 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 \( B[i+1] \)
  
  \( \Rightarrow \) These two dependences are loop-carried spanning more than one iteration preventing loop parallelism.
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];
}
B[101] = C[100] + D[100];    Loop Completion code
```
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];
ILP Compiler Support:
Loop-Carried Dependence Detection

- Compilers can increase the utilization of ILP by better detection of instruction dependencies.

- To detect loop-carried dependence in a loop, the GCD test can be used by the compiler, which is based on the following:

- If an array element with index: \( a \times i + b \) is stored and element: \( c \times i + d \) of the same array is loaded where index runs from \( m \) to \( n \), a dependence exist if the following two conditions hold:

  1. There are two iteration indices, \( j \) and \( k \), \( m \leq j \leq K \leq n \) (within iteration limits)

  2. The loop stores into an array element indexed by:

     \[ a \times j + b \]

     and later loads from the same array the element indexed by:

     \[ c \times k + d \]

     Thus:

     \[ a \times j + b = c \times k + d \]
The Greatest Common Divisor (GCD) Test

- If a loop carried dependence exists, then:

  \[
  \text{GCD}(c, a) \text{ must divide } (d-b)
  \]

Example:

```latex
for(i=1; i<=100; i=i+1) {
    x[2*i+3] = x[2*i] * 5.0;
}
```

\(a = 2\) \(b = 3\) \(c = 2\) \(d = 0\)

\[
\text{GCD}(a, c) = 2
\]

\[
d - b = -3
\]

2 does not divide -3 \(\Rightarrow\) No dependence possible.
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 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.

- On a cache miss the cache block being removed 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.
Unified vs. Separate Level 1 Cache

- **Unified Level 1 Cache** (Princeton Memory Architecture).
  A single level 1 cache is used for both instructions and data.

- **Separate instruction/data Level 1 caches** (Harvard Memory Architecture):
  The level 1 ($L_1$) cache is split into two caches, one for instructions ($L_1$ I-cache) and the other for data ($L_1$ D-cache).
Cache 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.
- For ideal memory: AMAT = 1 cycle, this results in zero memory stall cycles.
- Memory stall cycles per average memory access = (AMAT - 1)
- Memory stall cycles per average instruction =
  
  Memory stall cycles per average memory access
  x Number of memory accesses per instruction
  
  = (AMAT - 1) x (1 + fraction of loads/stores)

  Instruction Fetch
Cache Performance
Princeton (Unified) Memory Architecture

For a CPU with a single level (L1) of cache for both instructions and data (Princeton memory architecture) and no stalls for cache hits:

Total CPU time = (CPU execution clock cycles + Memory stall clock cycles) x clock cycle time

Memory stall clock cycles =
(Reads x Read miss rate x Read miss penalty) + (Writes x Write miss rate x Write miss penalty)

If write and read miss penalties are the same:
Memory stall clock cycles =
Memory accesses x Miss rate x Miss penalty
Cache Performance

Princeton (Unified) Memory Architecture

CPU time = Instruction count x CPI x Clock cycle time

CPI_{execution} = CPI with ideal memory

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

CPU time = Instruction Count x (CPI_{execution} + Mem Stall cycles per instruction) x Clock cycle time

Mem Stall cycles per instruction = Mem accesses per instruction x Miss rate x Miss penalty

CPU time = IC x (CPI_{execution} + Mem accesses per instruction x Miss rate x Miss penalty) x Clock cycle time

Misses per instruction = Memory accesses per instruction x Miss rate

CPU time = IC x (CPI_{execution} + Misses per instruction x Miss penalty) x Clock cycle time
Memory Access Tree
For Unified Level 1 Cache

CPU Memory Access

L1 Hit:
\[ \text{L1 Hit: } \quad \% = \text{Hit Rate} = H_1 \]
Access Time = 1
Stalls = \( H_1 \times 0 = 0 \)
(No Stall)

L1 Miss:
\[ \% = (1 - \text{Hit rate}) = (1 - H_1) \]
Access time = \( M + 1 \)
Stall cycles per access = \( M \times (1 - H_1) \)

AMAT = \( H_1 \times 1 + (1 - H_1) \times (M+1) = 1 + M \times (1 - H_1) \)

Stall Cycles Per Access = AMAT - 1 = \( M \times (1 - H_1) \)

\( M = \text{Miss Penalty} \)
\( H_1 = \text{Level 1 Hit Rate} \)
\( 1 - H_1 = \text{Level 1 Miss Rate} \)
Cache Impact On Performance: An Example

Assuming the following execution and cache parameters:

- Cache miss penalty = 50 cycles
- Normal instruction execution CPI ignoring memory stalls = 2.0 cycles
- Miss rate = 2%
- Average memory references/instruction = 1.33

\[
\text{CPU time} = \text{IC} \times [\text{CPI}_{\text{execution}} + \text{Memory accesses/instruction} \times \text{Miss rate} \times \text{Miss penalty}] \times \text{Clock cycle time}
\]

\[
\text{CPU time with cache} = \text{IC} \times (2.0 + (1.33 \times 2\% \times 50)) \times \text{clock cycle time}
\]

\[
= \text{IC} \times 3.33 \times \text{Clock cycle time}
\]

→ Lower CPI_{execution} increases the impact of cache miss clock cycles
Cache Performance Example

- Suppose a CPU executes at Clock Rate = 200 MHz (5 ns per cycle) with a single level of cache.
- \( \text{CPI}_{\text{execution}} = 1.1 \)
- Instruction mix: 50% arith/logic, 30% load/store, 20% control
- Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.

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

\[
\text{Mem Stalls per instruction} = \text{Mem accesses per instruction} \times \text{Miss rate} \times \text{Miss penalty}
\]

\[
\text{Mem accesses per instruction} = 1 + .3 = 1.3
\]

\[
\text{Mem Stalls per instruction} = 1.3 \times .015 \times 50 = 0.975
\]

\[
\text{CPI} = 1.1 + .975 = 2.075
\]

The ideal memory CPU with no misses is \( \frac{2.075}{1.1} = 1.88 \) times faster
Cache Performance
Harvard 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:

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

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

\[ \text{CPU time} = \text{Instruction Count} \times (\text{CPI}_{\text{execution}} + \text{Mem Stall cycles per instruction}) \times \text{Clock cycle time} \]

\[ \text{Mem Stall cycles per instruction} = \]

\[ \text{Instruction Fetch Miss rate} \times \text{Miss Penalty} + \]

\[ \text{Data Memory Accesses Per Instruction} \times \text{Data Miss Rate} \times \text{Miss Penalty} \]
Memory Access Tree
For Separate Level 1 Caches

CPU Memory Access

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

Instruction L1 Miss:
- Access Time = M + 1
- Stalls Per access
- % instructions \times (1 - Instruction H1) \times M

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

Data L1 Miss:
- Access Time: M + 1
- Stalls per access:
- % data \times (1 - Data H1) \times M

Stall Cycles Per Access = \% Instructions \times (1 - Instruction H1) \times M + \% data \times (1 - Data H1) \times M

AMAT = 1 + Stall Cycles per access
Cache Performance Example

To compare the performance of either using a 16-KB instruction cache and a 16-KB data cache as opposed to using a unified 32-KB cache, we assume a hit to take one clock cycle and a miss to take 50 clock cycles, and a load or store to take one extra clock cycle on a unified cache, and that 75% of memory accesses are instruction references. Using the miss rates for SPEC92 we get:

Overall miss rate for a split cache = (75% x 0.64%) + (25% x 6.74%) = 2.1%

From SPEC92 data a unified cache would have a miss rate of 1.99%

Average memory access time = 1 + stall cycles per access
= 1 + % instructions x (Instruction miss rate x Miss penalty)
+ % data x (Data miss rate x Miss penalty)

For split cache:
Average memory access time_{split} = 1 + 75% x (0.64% x 50) + 25% x (6.47% x 50) = 2.05 cycles

For unified cache:
Average memory access time_{unified} = 1 + 75% x (1.99% x 50) + 25% x (1 + 1.99% x 50)
= 2.24 cycles
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 bit, is used to indicate whether the block was modified while in cache; if not the block is not written to main memory.
   - 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:
The cache block is loaded on a write miss followed by write hit actions.

No-Write Allocate:
The block is modified in the lower level (lower cache level, or main memory) and not loaded 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 $L_1$
Write Through, No Write Allocate, No Write Buffer

CPU Memory Access

Read

$L_1$

- L1 Read Hit: Access Time = 1
- L1 Read Miss: Access Time = $M + 1$
- Stalls Per access: \( \% \text{ reads} \times (1 - H_1) \times M \)

Write

- L1 Write Hit: Access Time: $M + 1$
- Stalls Per access: \( \% \text{ write} \times (H_1) \times M \)
- L1 Write Miss: Access Time: $M + 1$
- Stalls per access: \( \% \text{ write} \times (1 - H_1) \times M \)

(A write buffer eliminates some or all these stalls)

Stall Cycles Per Memory Access = \( \% \text{ reads} \times (1 - H_1) \times M + \% \text{ write} \times M \)

AMAT = \( 1 + \% \text{ reads} \times (1 - H_1) \times M + \% \text{ write} \times M \)
Memory Access Tree Unified L₁
Write Back, With Write Allocate

CPU Memory Access

Read

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

L₁ Read Miss

Clean
Access Time = M +1
Stall cycles = M x (1-H₁) x % reads x % clean

Dirty
Access Time = 2M +1
Stall cycles = 2M x (1-H₁) x % read x % dirty

Write

L₁ Write Hit:
% write x H₁
Access Time = 1
Stalls = 0

L₁ Write Miss

Clean
Access Time = M +1
Stall cycles = M x (1-H₁) x % write x % clean

Dirty
Access Time = 2M +1
Stall cycles = 2M x (1-H₁) x % read x % dirty

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

AMAT = 1 + Stall Cycles Per Memory Access
Write Through Cache Performance Example

- A CPU with $CPI_{\text{execution}} = 1.1$ uses a unified L1 Write Through, No Write Allocate and no write buffer.
- 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.

\[
CPI = CPI_{\text{execution}} + \frac{\text{mem stalls per instruction}}{
\frac{\text{mem accesses per instruction}}{\text{stalls per access}}
\times \text{mem accesses per instruction}}
\]

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 = \% reads x miss rate x Miss penalty + \% write x Miss penalty

% reads = 1.15/1.3 = 88.5\% \quad % \text{writes} = .15/1.3 = 11.5\%

Stalls per access = 50 \times (88.5\% \times 1.5\% + 11.5\%) = 6.4 \text{ cycles}

Mem Stalls per instruction = 1.3 \times 6.4 = 8.33 \text{ cycles}

AMAT = 1 + 8.33 = 9.33 \text{ cycles}

CPI = 1.1 + 8.33 = 9.43

The ideal memory CPU with no misses is $9.43/1.1 = 8.57$ times faster.
Write Back Cache Performance Example

- A CPU with $CPI_{\text{execution}} = 1.1$ uses a unified L1 with write back, 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.

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

Mem Stalls per instruction =

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

Mem accesses per instruction = \(1 + .3 = 1.3\)

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\%) = .825 \text{ cycles}

Mem Stalls per instruction = 1.3 \times .825 = 1.07 \text{ cycles}

AMAT = 1 + 1.07 = 2.07 \text{ cycles}

CPI = 1.1 + 1.07 = 2.17

The ideal CPU with no misses is \(2.17/1.1 = 1.97\) times faster
Impact of Cache Organization: \textit{An Example}

\textbf{Given:}

- A perfect CPI with cache = 2.0 \quad \text{Clock cycle} = 2 \text{ ns}
- 1.3 memory references/instruction \quad \text{Cache size} = 64 \text{ KB with}
- Cache miss penalty = 70 \text{ ns, no stall on a cache hit}
- One cache is direct mapped with miss rate = 1.4%
- The other cache is two-way set-associative, where:
  - CPU time increases 1.1 times to account for the cache selection multiplexor
  - Miss rate = 1.0%

\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}
\text{Average memory access time}_{1\text{-way}} = 2.0 + (.014 \times 70) = 2.98 \text{ ns}
\text{Average memory access time}_{2\text{-way}} = 2.0 \times 1.1 + (.010 \times 70) = 2.90 \text{ ns}

\text{CPU time} = \text{IC} \times [\text{CPI}_{\text{execution}} + \text{Memory accesses/instruction} \times \text{Miss rate} \times \text{Miss penalty}] \times \text{Clock cycle time}
\text{CPU time}_{1\text{-way}} = \text{IC} \times (2.0 \times 2 + (1.3 \times .014 \times 70)) = 5.27 \times \text{IC}
\text{CPU time}_{2\text{-way}} = \text{IC} \times (2.0 \times 2 \times 1.1 \times (1.3 \times 0.01 \times 70)) = 5.31 \times \text{IC}

\rightarrow \text{In this example, 1-way cache offers slightly better performance with less complex hardware.}
2 Levels of Cache: $L_1, L_2$

- **CPU**
  - **$L_1$ Cache**
    - Hit Rate = $H_1$, Hit time = 1 cycle
    - (No Stall)
  - **$L_2$ Cache**
    - Hit Rate = $H_2$, Hit time = $T_2$ cycles

- **Main Memory**

  *Memory access penalty, $M$*
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. Local Hit Rate = 1 - 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:**
  – Local Miss Rate = Global Miss Rate = 1 - H1

• **For level 2 since it only receives those accesses missed in 1:**
  – Local Miss Rate = Miss rate\(_{L2}\) = 1 - H2
  – Global Miss Rate = Miss rate\(_{L1}\) x Miss rate\(_{L2}\)
    = (1 - H1) x (1 - H2)
2-Level Cache Performance

CPUtil = IC \times (CPI_{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 2 levels of cache, assuming no penalty when found in L_1 cache:

Stall cycles per memory access =

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

\[(1-H_1) \times H_2 \times T_2 + (1-H_1)(1-H_2) \times M\]

L1 Miss, L2 Hit

L1 Miss, L2 Miss:
Must Access Main Memory
2-Level Cache Performance
Memory Access Tree

CPU Stall Cycles Per Memory Access

CPU Memory Access

L₁
L₁ Hit:
Stalls = H₁ x 0 = 0
(No Stall)

L₂
L₂ Hit:
(1-H₁) x H₂ x T₂
L₂ Miss:
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
Two-Level Cache Example

- CPU with \( \text{CPI}_{\text{execution}} = 1.1 \) running at clock rate = 500 MHZ
- 1.3 memory accesses per instruction.
- \( L_1 \) cache operates at 500 MHZ with a miss rate of 5%
- \( L_2 \) cache operates at 250 MHZ with local miss rate 40%, \((T_2 = 2 \text{ cycles})\)
- Memory access penalty, \( M = 100 \) cycles. 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

\[
\text{Stall cycles per memory access} = (1-H_1) \times H_2 \times T_2 + (1-H_1)(1-H_2) \times M
\]

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

\[
= 0.06 + 2 = 2.06
\]

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

\[
= 2.06 \times 1.3 = 2.678
\]

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

\[
\text{Speedup} = 7.6/3.778 = 2
\]
3 Levels of Cache

CPU

L1 Cache
Hit Rate = H_1, Hit time = 1 cycle

L2 Cache
Hit Rate = H_2, Hit time = T_2 cycles

L3 Cache
Hit Rate = H_3, Hit time = T_3

Main Memory

Memory access penalty, M
3-Level Cache Performance

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

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

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

Stall cycles per memory access =

\[
\left[ \text{miss rate } L_1 \right] \times \left[ \text{Hit rate } L_2 \times \text{Hit time } L_2 \right. \\
+ \left. \text{Miss rate } L_2 \times (\text{Hit rate } L_3 \times \text{Hit time } L_3 \right. \\
+ \left. \text{Miss rate } L_3 \times \text{Memory access penalty} \right] = \\
(1-H1) \times H2 \times T2 \\
+ (1-H1) \times (1-H2) \times H3 \times T3 \\
+ (1-H1)(1-H2)(1-H3)x M
\]

L1 Miss, L2 Miss: Must Access Main Memory
L1 Miss, L2 Hit
L2 Miss, L3 Hit
### 3-Level Cache Performance

#### Memory Access Tree

**CPU Stall Cycles Per Memory Access**

- **CPU Memory Access**
  - **L1**
    - **Hit:**
      - Stalls = $H_1 \times 0 = 0$
      - (No Stall)
    - **Miss:**
      - $\% = (1 - H_1)$
  
  - **L2**
    - **Hit:**
      - $(1 - H_1) \times H_2 \times T_2$
    - **Miss:**
      - $\% = (1 - H_1)(1 - H_2)$

  - **L3**
    - **Hit:**
      - $(1 - H_1) \times (1 - H_2) \times H_3 \times T_3$
    - **Miss:**
      - $(1 - H_1)(1 - H_2)(1 - H_3) \times M$

Stall cycles per memory access = $(1 - H_1) \times H_2 \times T_2 + (1 - H_1) \times (1 - H_2) \times H_3 \times T_3 + (1 - H_1)(1 - H_2)(1 - H_3) \times M$

**AMAT = 1 + Stall cycles per memory access**
Three-Level Cache Example

- 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 with a miss rate of 5%
- L_2 cache operates at 250 MHZ with a local miss rate 40%, (T_2 = 2 cycles)
- L_3 cache operates at 100 MHZ with a local miss rate 50%, (T_3 = 5 cycles)
- Memory access penalty, M = 100 cycles. Find CPI.

With No Cache, CPI = 1.1 + 1.3 x 100 = 131.1

With single L_1, CPI = 1.1 + 1.3 x 0.05 x 100 = 7.6

With L_1, L_2, CPI = 1.1 + 1.3 x (0.05 x 0.6 x 2 + 0.05 x 0.4 x 100) = 3.778

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

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

Stall cycles per memory access = (1-H_1) x H_2 x T_2 + (1-H_1) x (1-H_2) x H_3 x T_3 + (1-H_1)(1-H_2) (1-H_3)x M
= 0.05 x 0.6 x 2 + 0.05 x 0.4 x 0.5 x 5 + 0.05 x 0.4 x 0.5 x 100
= 0.097 + 0.0075 + 0.00225 = 1.11

CPI = 1.1 + 1.3 x 1.11 = 2.54

Speedup compared to L_1 only = 7.6/2.54 = 3

Speedup compared to L_1, L_2 = 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.

• 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. Measured by:
    
    • **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.
    
    • **Cycle time:** The minimum time between requests to memory (greater than access time in DRAM to allow address lines to be stable)

  – **Memory bandwidth:** The maximum sustained data transfer rate between main memory and cache/CPU.
Typical timing at 133 MHZ (PC133 SDRAM) : 5-1-1-1
For bus width = 64 bits = 8 bytes  Max. Bandwidth = 133 x 8 = 1064 Mbytes/sec
It takes = 5+1+1+1 = 8 memory cycles or 7.5 ns x 8 = 60 ns to read 32 byte cache block
Minimum Read Miss penalty for CPU running at 1 GHZ = 7.5 x 8 = 60 CPU cycles
Memory Bandwidth Improvement Techniques

- **Wider Main Memory:**
  Memory width is increased to a number of words (usually the size of a cache block).
  ⇒ Memory bandwidth is proportional to memory width.
  e.g. Doubling the width of cache and memory doubles memory bandwidth

- **Simple Interleaved Memory:**
  Memory is organized as a number of banks each one word wide.
  – Simultaneous multiple word memory reads or writes are accomplished by sending memory addresses to several memory banks at once.
  – Interleaving factor: Refers to the mapping of memory addressess to memory banks.
  e.g. using 4 banks, bank 0 has all words whose address is:
    \[(\text{word address mod } 4) = 0\]
Memory Interleaving

Access Pattern without Interleaving:

- Start Access for D1
- Start Access for D2
  - D1 available

Access Pattern with 4-way Interleaving:

Access Bank 0
Access Bank 1
Access Bank 2
Access Bank 3

We can Access Bank 0 again
Number of banks ≥ Number of cycles to access word in a bank
Memory Width, Interleaving: An Example

Given the following system parameters with single cache level $L_1$:

Block size = 1 word  Memory bus width = 1 word  Miss rate = 3%  Miss penalty = 32 cycles  
(4 cycles to send address  24 cycles access time/word,  4 cycles to send a word)

Memory access/instruction = 1.2  Ideal CPI (ignoring cache misses) = 2  
Miss rate (block size = 2 word) = 2%  Miss rate (block size = 4 words) = 1%

- The CPI of the base machine with 1-word blocks = $2 + (1.2 \times 0.03 \times 32) = 3.15$
- Increasing the block size to two words gives the following CPI:
  - 32-bit bus and memory, no interleaving = $2 + (1.2 \times 0.02 \times 2 \times 32) = 3.54$
  - 32-bit bus and memory, interleaved = $2 + (1.2 \times 0.02 \times (4 + 24 + 8) = 2.86$
  - 64-bit bus and memory, no interleaving = $2 + (1.2 \times 0.02 \times 1 \times 32) = 2.77$

- Increasing the block size to four words; resulting CPI:
  - 32-bit bus and memory, no interleaving = $2 + (1.2 \times 0.01 \times 4 \times 32) = 3.54$
  - 32-bit bus and memory, interleaved = $2 + (1.2 \times 0.01 \times (4 +24 + 16) = 2.53$
  - 64-bit bus and memory, no interleaving = $2 + (1.2 \times 0.01 \times 2 \times 32) = 2.77$
Main Memory Bandwidth-Usage Example

- In the example with three levels of cache
- 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 with a miss rate of 5%
- L\(_2\) cache operates at 250 MHZ with a local miss rate 40%, (T\(_2\) = 2 cycles)
- L\(_3\) cache operates at 100 MHZ with a local miss rate 50%, (T\(_3\) = 5 cycles)
- Memory access penalty, M = 100 cycles.

- We found the 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 L\(_1\), L\(_2\) \(\text{CPI} = 1.1 + 1.3 \times (0.05 \times 0.6 \times 2 + 0.05 \times 0.4 \times 100) = 3.778\)
  - With L\(_1\), L\(_2\), L\(_3\) \(\text{CPI} = 1.1 + 1.3 \times 1.11 = 2.54\)

Assuming:
- instruction size = data size = 4 bytes, all cache blocks are 32 bytes and

For each of the three cases with cache:
- What is the total number of memory accesses generated by the CPU per second?
- What is the percentage of these memory accesses satisfied by main memory?
- Percentage of main memory bandwidth used by the CPU?
Main Memory Bandwidth-Usage Example

- Memory requires 100 CPU cycles = 200 ns to deliver 32 bytes, thus total main memory bandwidth = 32 bytes / (200 ns) = 160 x 10^6 bytes/sec

- 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 single L1 = 650 x 10^6 / 7.6 = 85 x 10^6 accesses/sec
  - With L1, L2 = 650 x 10^6 / 3.778 = 172 x 10^6 accesses/sec
  - With L1, L2, L3 = 650 x 10^6 / 2.54 = 255 x 10^6 accesses/sec

- The percentage of these memory accesses satisfied by main memory:
  - With single L1 = L1 miss rate = 5%
  - With L1, L2 = L1 miss rate x L2 miss rate = .05 x .4 = 2%
  - with L1, L2, L3 = L1 miss rate x L2 miss rate x L3 miss rate = .05 x .4 x .5 = 1%

- Memory Bandwidth used
  - With single L1 = 32 bytes x 85x10^6 accesses/sec x .05 = 136 x10^6 bytes/sec
    or 136/160 = 85 % of total memory bandwidth
  - With L1, L2 = 32 bytes x 172 x10^6 accesses/sec x .02 = 110 x10^6 bytes/sec
    or 110/160 = 69 % of total memory bandwidth
  - With L1, L2, L3 = 32 bytes x 255 x10^6 accesses/sec x .01 = 82 x10^6 bytes/sec
    or 82/160 = 51 % of total memory bandwidth
Virtual Memory

Benefits

– Illusion of having more physical main memory
– Allows program relocation
– Protection from illegal memory access

Virtual address

31 30 29 28 27 ............... 15 14 13 12 11 10 9 8 ........ 3 2 1 0

Virtual page number  Page offset

Translation

29 28 27 ............... 15 14 13 12 11 10 9 8 ........ 3 2 1 0

Physical page number  Page offset

Physical address
Speeding Up Address Translation:
Translation Lookaside Buffer (TLB)

- TLB: A small on-chip fully-associative cache used for address translations.
- If a virtual address is found in TLB (a TLB hit), the page table in main memory is not accessed.

![Diagram of TLB and page table interactions]
CPU Performance with Real TLBs

When a real TLB is used with a TLB miss rate and a TLB miss penalty is used:

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

Where:

- Mem Stalls per instruction = Mem accesses per instruction \times mem stalls per access
- Similarly:
- TLB Stalls per instruction = Mem accesses per instruction \times TLB stalls per access
  \quad \text{TLB stalls per access} = \text{TLB miss rate} \times \text{TLB miss penalty}

Example:
Given: \( \text{CPI}_{\text{execution}} = 1.3 \)  \( \text{Mem accesses per instruction} = 1.4 \)
Mem stalls per access = .5  \( \text{TLB miss rate} = .3\% \)  \( \text{TLB miss penalty} = 30 \text{ cycles} \)

What is the resulting CPU CPI?

- Mem Stalls per instruction = 1.4 \times .5 = .7 \text{ cycles/instruction}
- TLB stalls per instruction = 1.4 \times (\text{TLB miss rate} \times \text{TLB miss penalty})
  = 1.4 \times .003 \times 30 = .126 \text{ cycles/instruction}
- CPI = 1.3 + .7 + .126 = 2.126
System Components

CPU Core
1 GHz - 3.0 GHz
4-way Superscaler
RISC or RISC-core (x86):
  Deep Instruction Pipelines
  Dynamic scheduling
  Multiple FP, integer FUs
  Dynamic branch prediction
  Hardware speculation

SDRAM
PC100/PC133
100-133MHz
64-128 bits wide
2-way interleaved
~ 900 MBYTES/SEC (64bit)

Double Date Rate (DDR) SDRAM
PC2100
133MHz DDR
64-128 bits wide
4-way interleaved
~2.1 GBYTES/SEC (64bit)

RAMbus DRAM (RDRAM)
400MHZ DDR
16 bits wide (32 banks)
~ 1.6 GBYTES/SEC

CPU

Caches

Memory

Chipset

Main I/O Bus
Example: PCI, 33-66MHz
32-64 bits wide
133-528 MB
PCI-X 133MHz 64-bits wide
1066 MB

I/O Subsystem

Disks
Displays
Keyboards

I/O Controllers

NICs

Networks

I/O Devices

Chipset

North Bridge

South Bridge

Bus Adapter

Memory Bus

Front System Bus

Examples: Alpha, AMD K7: EV6, 200-333MHz
Intel PII, PIII: GTL+ 133 MHz
Intel P4 533 MHz

TIME(workload) = TIME(CPU) + TIME(I/O) - TIME(Overlap)

CPU Core
1 GHz - 3.0 GHz
4-way Superscaler
RISC or RISC-core (x86):
  Deep Instruction Pipelines
  Dynamic scheduling
  Multiple FP, integer FUs
  Dynamic branch prediction
  Hardware speculation

SDRAM
PC100/PC133
100-133MHz
64-128 bits wide
2-way interleaved
~ 900 MBYTES/SEC (64bit)

Double Date Rate (DDR) SDRAM
PC2100
133MHz DDR
64-128 bits wide
4-way interleaved
~2.1 GBYTES/SEC (64bit)

RAMbus DRAM (RDRAM)
400MHZ DDR
16 bits wide (32 banks)
~ 1.6 GBYTES/SEC

CPU

Caches

Memory

Chipset

North Bridge

South Bridge

Bus Adapter

Memory Bus

Front System Bus

Examples: Alpha, AMD K7: EV6, 200-333MHz
Intel PII, PIII: GTL+ 133 MHz
Intel P4 533 MHz

TIME(workload) = TIME(CPU) + TIME(I/O) - TIME(Overlap)
I/O Performance Metrics

- **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.
- **Producer/server Model of I/O**: 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 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
  \[\Rightarrow\] Maximized when task buffer is never empty.
- **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 buffer waiting or queuing time.
  \[\Rightarrow\] Maximized when task buffer is always empty.

In textbook: Ch. 7.1-7.3, 7.7, 7.8
Producer-Server Model

Response Time = \( T_{\text{System}} = T_{\text{Queue}} + T_{\text{Server}} \)

Throughput vs. Response Time

Throughput vs. Response Time

Percent of maximum throughput (bandwidth)

Response time (latency) in ms

300
200
100
0
0%
20%
40%
60%
80%
100%
Magnetic Disks

Characteristics:

- Diameter: 2.5in - 5.25in
- Rotational speed: 3,600RPM-10,000 RPM
- Tracks per surface.
- Sectors per track: Outer tracks contain more sectors.
- Recording or Areal Density: Tracks/in × Bits/in
- Cost Per Megabyte.
- Seek Time: The time needed to move the read/write head arm.
  Reported values: Minimum, Maximum, Average.
- Rotation Latency or Delay:
  The time for the requested sector to be under the read/write head.
- Transfer time: The time needed to transfer a sector of bits.
- Type of controller/interface: SCSI, EIDE
- Disk Controller delay or time.
- Average time to access a sector of data =
  average seek time + average rotational delay + transfer time +
  disk controller overhead
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 time for a 512-byte Sector?

\[
\text{Ave. seek + ave. rot delay + transfer time + controller overhead} = 5 \text{ ms} + \frac{0.5}{7200 \text{ RPM/60}} + \frac{0.5 \text{ KB/40 MB/s}}{40} + 0.1 \text{ ms} \\
= 8.11 \text{ ms}
\]

This time is service time \( T_{\text{ser}} \) for this task used for queuing delay computation.
Introduction to Queuing Theory

• Concerned with long term, steady state than in startup:
  - where => Arrivals = Departures

• **Little’s Law:**
  Mean number tasks in system = arrival rate x mean response time

• Applies to any system in equilibrium, as long as nothing in the black box is creating or destroying tasks.
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
- \( T_q \): 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 \( T_q \)
- \( r \): Average number of arriving tasks/sec
- \( L_{\text{ser}} \): 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: \( L_{\text{sys}} = r \times T_{\text{sys}} \)

Server utilization = \( u = \frac{r}{T_{\text{ser}}} \) 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.
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 $T_{\text{ser}}$

• 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

\[
T_q = T_{\text{ser}} \times r \times u / (1 - u)
\]

$T_{\text{ser}}$ average time to service a task

$r$ average number of arriving tasks/second

$u$ server utilization (0..1): $u = r \times T_{\text{ser}}$

$T_q$ average time/task in queue

$T_{\text{sys}}$ Average time per task in the system $T_{\text{sys}} = T_q + T_{\text{ser}}$

$L_q$ average length of queue: $L_q = r \times T_q$

$L_{\text{sys}}$ Average number of tasks in the system $L_{\text{sys}} = r \times T_{\text{sys}}$
Multiple Disk/Controller I/O Modeling: M/M/M/m Queue

- I/O system with Markovian request arrival rate $r$
- A single queue serviced by $m$ servers (disks + controllers) each with Markovian Service rate $= 1/ T_{ser}$

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

\[
u = r \times T_{ser} / m
\]

- $m$ number of servers
- $T_{ser}$ average time to service a task
- $u$ server utilization (0..1): $u = r \times T_{ser} / m$
- $T_q$ average time/task in queue
- $T_{sys}$ Average time per task in the system $T_{sys} = T_q + T_{ser}$
- $L_q$ average length of queue: $L_q = r \times T_q$
- $L_{sys}$ Average number of tasks in the system $L_{sys} = r \times T_{sys}$
I/O Queuing Performance: An Example

- A processor sends 10 x 8KB disk I/O requests per second, requests & service are exponentially distributed, average disk service time = 20 ms

- On average:
  - How utilized is the disk, 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 = 10
  - $T_{ser}$ average time to service a request = 20 ms (0.02s)

- We obtain:
  - $u$ server utilization: $u = r x T_{ser} = 10/s x .02s = 0.2$
  - $T_q$ average time/request in queue = $T_{ser} x u / (1 - u)$
    - $= 20 x 0.2/(1-0.2) = 20 x 0.25 = 5$ ms (0.005s)
  - $T_{sys}$ average time/request in system: $T_{sys} = T_q + T_{ser} = 25$ ms
  - $L_q$ average length of queue: $L_q = r x T_q$
    - $= 10/s x .005s = 0.05$ requests in queue
  - $L_{sys}$ average # tasks in system: $L_{sys} = r x T_{sys} = 10/s x .025s = 0.25$
Example: Determining the System I/O Bottleneck

- 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

- Other assumptions
  - All devices used to 100% capacity, always have average values
  - Average I/O size is 16 KB
  - OS uses 10,000 CPU instructions for a disk I/O
  - Ignore disk/controller queuing delays.

- What is the average IOPS? What is the average bandwidth?
Example: Determining the I/O Bottleneck

• The performance of I/O systems is determined by the portion with the lowest I/O bandwidth
  – CPU: \((500 \text{ MIPS})/(10,000 \text{ instr. per I/O}) = 50,000 \text{ IOPS}\)
  – Main Memory: \((16 \text{ bytes})/(100 \text{ ns} \times 16 \text{ KB per I/O}) = 10,000 \text{ IOPS}\)
  – I/O bus: \((200 \text{ MB/sec})/(16 \text{ KB per I/O}) = 12,500 \text{ IOPS}\)
  – SCSI-2: \(((20 \text{ buses})/((1 \text{ ms} + (16 \text{ KB})/(20 \text{ MB/sec})) \text{ per I/O}) = 11,120 \text{ IOPS}\)
  – Disks: \(((100 \text{ disks})/((8 \text{ ms} + 0.5/(7200 \text{ RPMS}) + (16 \text{ KB})/(6 \text{ MB/sec})) \text{ per I/O}) = 6,700 \text{ IOPS}\)

• In this case, the disks limit the I/O performance to 6,700 IOPS

• The average I/O bandwidth is
  – \(6,700 \text{ IOPS} \times (16 \text{ KB/sec}) = 107.2 \text{ MB/sec}\)
Example: Determining the I/O Bottleneck
Accounting For I/O Queue Time

• 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

• Other assumptions
  – All devices used to 60% capacity.
  – Treat the I/O system as an M/M/m queue.
  – Requests are assumed spread evenly on all disks.
  – Average I/O size is 16 KB
  – OS uses 10,000 CPU instructions for a disk I/O

• What is the average IOPS? What is the average bandwidth?
• Average response time per IO operation?
Example: Determining the I/O Bottleneck

Accounting For I/O Queue Time

• The performance of I/O systems is still determined by the system component with the lowest I/O bandwidth
  – CPU : (500 MIPS)/(10,000 instr. per I/O) x .6 = 30,000 IOPS
    CPU time per I/O = 10,000 / 500,000,000 = .02 ms
  – Main Memory : (16 bytes)/(100 ns x 16 KB per I/O) x .6 = 6,000 IOPS
    Memory time per I/O = 1/10,000 = .1ms
  – I/O bus: (200 MB/sec)/(16 KB per I/O) x .6 = 12,500 IOPS
  – SCSI-2: (20 buses)/((1 ms + (16 KB)/(20 MB/sec)) per I/O) = 7,500 IOPS
    SCSI bus time per I/O = 1ms + 16/20 ms = 1.8ms
  – Disks: (100 disks)/((8 ms + 0.5/(7200 RPMS) + (16 KB)/(6 MB/sec)) per I/O) x .6 = 6,700 x .6 = 4020 IOPS
    \( T_{ser} = (8 \text{ ms} + 0.5/(7200 \text{ RPMS}) + (16 \text{ KB})/(6 \text{ MB/sec}) \) = 8+4.2+2.7 = 14.9ms

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