CSCE 4610: Computer Architecture

Exam 1 Stats
High = 100 Average = 60 Std. Dev = 22

#1. Note we have two modes that are speeded up: \( f_1 \) and \( f_2 \) by \( s_1 \) and \( s_2 \)

a). The portion that is not speeded up = \( 1 - f_1 - f_2 \)
New execution time = \( (1 - f_1 - f_2) + (f_1/s_1) + (f_2/s_2) \)
Speed up = \( 1/(\text{new execution time}) \)

b). Using \( f_1 = 0.5 \) and \( f_2 = 0.25 \) and \( s_1 = 10 \) and \( s_2 = 5 \),
speed up = \( 1/[0.25 + 0.5/10 + 0.25/5] = 1/0.35 = 2.857 \)

c). Generalized express

new execution time = \( (1-f_1-f_2-\ldots-f_n) + (f_1/s_1) + (f_2/s_2) + \ldots + (f_n/s_n) \)
Speedup = \( 1/(\text{new execution time}) \)

CSCE 4610 March 7, 2017

#2. Here we are not looking for average memory access times
We need to find out how many bytes per second are requested by CPU

CPU generates \( 10^9 \) requests per second
only 2% of these requests go to memory (on cache miss)
25% of misses lead to write-back
So each request on a miss causes 1.25 memory requests
Each request transfers 32 bytes

So CPU requests \( 0.02*10^9*1.25*32 = 800*10^6 = 0.8*10^9 \) bytes from memory

Since memory can support \( 10^9 \) bytes per second, CPU is using 80% of that bandwidth

#3 We have seen several example of this kind

```c
int i, j, b[100], c[100], a[100][100];
for (i=0; i<100; i++)
    c[i]=0.0;
for (j=0; j<100; j=j+1)
    c[i] = a[i][j]*b[j];
```
CSCE 4610: Computer Architecture

a) With 16 byte cache blocks (each cache line holds 4 data items)
For a, we cause $100 \times (100/4) = 2500$ misses
For b and c we cause $100/4 = 25$ misses each
Total misses = 2550 misses

Total number of access.
We access is element of a only once for a total of 100*100 accesses for a
We access each c[i] 101 times for a total of 101*100 accesses for c
We access each b[j] 100 times for a total of 100*100 accesses for b
Total accesses = 30,100
Miss rate = 2550/30100 = 0.0847 or 8.47%

b). With 32 bytes (8 data items per cache line)
100 elements of a row of a accesses in the inner loop cause
$100/8 = 12.5$ or 13 misses
Total misses caused by a = 100*13 = 1300
For b and c we cause $100/8 = 12.5$ but rounded off to 13 misses each
Total misses = 1326 and miss rate = 1326/30100 = 4.4%

#4. We are given
• The miss rate when using 4-way associative cache is 0.001.
• When using way-prediction, we have a 90% success and it costs 1 ns. However on a mis-prediction, but if the data is in one of the other ways, it will cost 2ns.
• If we use conventional 4-way associative cache, the clock speed will be slower by 10% (compared with way-prediction). So, the hit time is 1.2ns.
• Miss penalty is 10ns (with or without way-prediction).

i). Note on a hit we have two cases: if way-prediction is correct it costs 1ns; if way-prediction is wrong it costs 2ns. On a miss(0.1%), we take 10ns

Average Memory access time = (hit-time) + (miss-rate)*(miss-penalty)
= 1*[1*0.9+2*0.1] + 0.001*10= 1.11

ii). Here the clock is slow, but we spend 1.1ns on hit and 10ns on miss.

Average memory access time = hit-time + (miss-rate)*(miss-penalty)
= 1.1ns + 0.001*10 = 1.1 ns
CSCE 4610: Computer Architecture

If you are having difficulty with the material, see me
Work on more problems from textbook
try problems for other books (listed as reference books)

Is there anything I can do to help you?

Review
Data paths and control paths
Pipelined implementation
buffers between pipeline stages

Pipelined design – all instructions have to go through all 5 stages

Each instruction takes 50ns to complete
But on average, we can complete 1 instruction every 10ns

What are the advantages of pipelines?

Consider the following examples

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>10ns</td>
</tr>
<tr>
<td>Decode/fetch registers</td>
<td>5ns</td>
</tr>
<tr>
<td>ALU execution</td>
<td>5ns</td>
</tr>
<tr>
<td>Load/Store</td>
<td>10ns</td>
</tr>
<tr>
<td>Write results</td>
<td>5ns</td>
</tr>
</tbody>
</table>

Speed up = number of pipelines stages (after the pipeline is full)

CSCE 4610/4610: Computer Architecture

Suppose, one of the pipeline stage is very slow compared to the other stages

We have two choices to improve the performance of a pipeline

Super-Pipelines -- more pipeline stages
Super-Scalars -- more pipelines

In principle both should provide the same speed ups
but there are issues to be considered

Cache misses and branch penalties for superpipelines
Instruction level parallelism and data dependencies for superscalars
CSCE 4610/4610: Computer Architecture

Reasons for not achieving a speed up that is equal to the number of pipeline stages

3 type of “hazards” that require pipeline stalls

1. Structural Hazards due to resource dependencies
2. Data hazards due to data dependencies
3. Control hazards due to branch instructions

Hazards become more complex to deal with in superscalars out of order execution of instructions

Compilers can help in avoiding some of these hazards
- reordering of instructions
- renaming variables
- branch prediction
- delayed branches

CPI for pipeline machines = 1 + (avg number of stalls per instruction)

Ideal pipeline has a CPI of 1

CSCE 4610/4610: Computer Architecture

**Structural Hazard:** Consider a unified cache (in older processors)
- IF and MEM stages need to access cache
- So on Load/Store instructions, we need to stall IF stage

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>Decode</th>
<th>Exec</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW $s2, 0($s1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add $s3, $s1, $s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addi $s1, $s1, 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE $s3, $s2, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>Decode</th>
<th>Exec</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW $s2, 0($s1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add $s3, $s1, $s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addi $s1, $s1, 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE $s3, $s2, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CSCE 4610 March 7, 2017
CSCE 4610/4610: Computer Architecture

If 40% of instructions are load/store, then we have an average of 0.4 stalls per instruction. CPI = 1 + 0.4 = 1.4 cycles

If we use separate caches, we may have to account for added hardware complexity (i.e., slower clock) and higher cache misses (particularly for data caches)

Note we may have other types of structural hazards
Say only one Floating point multiply unit
And we have two consecutive multiply instructions
We will discuss these types of issues later

Data Hazards:
Consider the following example:

```
sub R2, R1, R3
and R12, R2, R5
or R13, R6, R2
add R14, R2, R2
sw R15, 100(R2)
```

When is the result from sub written to R2?

Write-Back stage.

But and instruction already read its register operands when it reached ID, therefore getting old value from R2

---

CSCE 4610 March 7, 2017

---

CSCE 4610/4610: Computer Architecture

```
Time (in clock cycles)
Value of register $s2:

Program execution order (in instructions):
sub $7, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $3
sw $15, 100($2)
```

CSCE 4610 March 7, 2017
But remember that actual result of sub is available when it completes EX stage
Can we take the result of an instruction from EX stage, without having to wait for write-back?

Data (or Result) forwarding
Data Hazards:

Consider another example code

\[
\begin{align*}
\text{DADD} & \quad \text{R1, R2, R3} \\
\text{DSUB} & \quad \text{R4, R1, R5} \\
\text{AND} & \quad \text{R6, R1, R7} \\
\text{OR} & \quad \text{R8, R1, R9} \\
\text{XOR} & \quad \text{R10, R1, R11}
\end{align*}
\]

Which instructions must be stalled?

Note XOR is not stalled since WB writes the result before XOR’s ID reads the value of R1 and R11 in cycle 6.

Data forwarding

1. Need to detect the hazard among instructions
2. Need to decide on when (and where) to forward

Detection of data hazards:

We use the information in pipestage latches

For example:

\[
\begin{align*}
1. & \quad \text{ADD} \quad \text{R1, R2, R3} \\
2. & \quad \text{SUB} \quad \text{R4, R1, R5}
\end{align*}
\]

When instruction 2 is in decode, instruction 1 is in Execute

\[
\text{IF} = \text{ID}(\text{IR}(ls)) = \text{ID}/\text{EX}(\text{IR}(ld))
\]

indicates that the destination of the instruction in EX is the first Source of the instruction in ID.
CSCE 4610/4610: Computer Architecture

Likewise IF/ID(IR(rs2)) = ID/EX(IR(rd)) indicates that the destination of the instruction in EX is the second source of the instruction in ID.

To stall an instruction in the pipeline:
1. Send no-op down the pipeline.
2. Prevent "next" instruction overwriting stalled instruction.
   This can be easily achieved using write-enable inputs associated with latches.

We need to similarly establish conditions to detect hazards:
- between Load and subsequent Arithmetic operations,
- between arithmetic operations and subsequent store
- between arithmetic operations and branch/compare instructions.

Data Forwarding

When instruction 1 completes in EX, instruction 2 completes ID. So we can write the output of ALU back to ID/EX stage for the next instruction.

ID/EX(src1) \rightarrow ALU-Out

What about the dependency between instruction 1 and 3?

1. ADD R1, R2, R3
2. ............
3. AND R6, R1, R7

When instruction 3 is in ID, instruction 1 is in MEM; so we need to forward the result form EX/MEM pipe latches back to ID/EX latches.

We need to detect dependency as follows:

IF/ID(IR(src1)) = EX/MEM(IR(rd))

Then forward

ID/EX(src1) \rightarrow EX/MEM(result)

Likewise need to check for second source.
Data Forwarding

Note we may have a data hazard either on the first operand or the second, so we need to test for both cases.

It may not always be possible to forward data and eliminate all stalls.

Consider:

1. LW \textbf{R4}, 0 (R1)
2. ADD R1, R4, R3

By the time instruction 1 completes execution (in MEM), instruction 2 is already in EX. Therefore, we can’t forward data for R4 in this case; a stall is required.

Too late for the next instruction
CSCE 4610/4610: Computer Architecture

Can we design pipelines to permit forwarding of data from Load to subsequent instructions?

- **LD** \( S_1, (S_2) \): Load data from \((R_2)+0\) into \(R_1\)
- **ADD** \( S_4, S_1, S_3 \): \(R_4 = R_1 + R_3\)

We need an extra adder.

---

This table shows all conditions to test and forward data (original pipeline):
An example (Problem C.1 from Textbook)

Loop: 
- LD R1, 0(R2)
- DADDI R1, R1, #1
- SD 0(R2), R1
- DADDI R2, R2, #4
- DSUB R4, R3, R2
- BNEZ R4, Loop

First – no data forwarding
(Assume registers can be written and read in the same cycle, during writeback)

Need 16 cycles to complete one iteration of the loop
(LD-F in cycles 1 and 17)
CSCE 4610/4610: Computer Architecture

The loop is repeated 99 times
(R3 is 396 larger than R2; and R2 is incremented by 4 each time)

So we need 16*99 cycles to complete the loop.

Now let us consider data forwarding between instructions.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DADDI</td>
<td>F</td>
<td>S</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DADDI</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSUB</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNEZ</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTSIDE</td>
<td>F</td>
<td>S</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

Now we need only 9 cycles to complete one iteration