**Part A: An Initial CPU Time Example**

**Question 1:**

**Preface:**
We can modify the datapath from Lecture 02-03 of the 3-instruction processor to add an instruction that performs an ALU operation on any two memory locations and stores the result in a register file location.

(We would need to add 2 multiplexers – ALU_Mux1 and ALU_Mux2 – to the inputs to the ALU to select between an input from the register file and an input from data memory.)

Note that if we wanted to perform the operation \(d(4) = d(5) + d(6) + d(7)\) with the old datapath (i.e. before the enhancement) we would need a total of 6 instructions. However, with the new datapath, we would only need 4 instructions:

<table>
<thead>
<tr>
<th>Un-enhanced Solution</th>
<th>Enhanced Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R1, (d(5))</td>
<td>Add R1, (d(5)), (d(6))</td>
</tr>
<tr>
<td>MOV R2, (d(6))</td>
<td>MOV R2, (d(7))</td>
</tr>
<tr>
<td>MOV R3, (d(7))</td>
<td>Add R3, R1, R2</td>
</tr>
<tr>
<td>Add R4, R1, R2</td>
<td>MOV (d(4)), R3</td>
</tr>
<tr>
<td>Add R5, R4, R3</td>
<td></td>
</tr>
<tr>
<td>MOV (d(4)), R5</td>
<td></td>
</tr>
</tbody>
</table>

From the standpoint of instruction count alone, the enhanced solution looks better. But is it?

**Part A:**
Assume that each instruction really does just take 3 CCs to execute (1 each for fetch, decode, and execute). Also, assume that clock rates of both 1 GHz and 2 GHz are possible. Calculate the CPU time of the un-enhanced and enhanced design assuming the 2 different clock rates. What is the potential “swing” in performance?

We can start with the CPU time formula – remember that execution time is the best metric...

\[
\text{CPU Time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycles}}
\]

| Time (un-enhanced, 1 GHz) | \((6 \text{ ins})(3 \text{ CCs})(1 \times 10^{-9} \text{ s})\) | \(= 1.8 \times 10^{-8} \text{ s} = 18 \text{ ns}\) |
| Time (un-enhanced, 2 GHz) | \((6 \text{ ins})(3 \text{ CCs})(0.5 \times 10^{-9} \text{ s})\) | \(= 9.0 \times 10^{-9} \text{ s} = 9 \text{ ns}\) |

- (above is easy to compare; instruction count, CPI are constants)

*** Be sure that you understand where \(1 \times 10^{-9} \text{ s}\) and \(0.5 \times 10^{-9} \text{ s}\) comes from ***
Time (enhanced, 1 GHz) = (4 ins)(3 CCs)(1 x 10^{-9}s) = 1.2 x 10^{-8} s = 12 ns
- (comparing this to the un-enhanced, 10 MHz version – its better to improve clock rate)

Time (enhanced, 2 GHz) = (4 ins)(3 CCs)(0.5 x 10^{-9}s) = 6.0 x 10^{-9} s = 6 ns
- (faster clock rate, fewer instructions = best)

Part B:
In reality, an instruction that requires a memory reference will require more clock cycles than an instruction that operates on data that’s just in registers. If the new ADD instruction requires 5 clock cycles, what is the average CPI for the different instruction mixes shown above?

- Here, need to take into account % of instructions with different CCs
  - For un-enhanced, easy: 100% x (3) = 3 CCs/instruction

- For enhanced, we can see that 1 instruction out of 4 requires 5 CCs
  - Therefore (0.75)(3) + (0.25)(5) = 3.5 CCs/instruction

- Note, CPI not as good (3.5 vs. 3.0)
  - So, what’s the advantage? Enhanced version uses fewer instructions...

Part C:
Repeat Part A given the assumptions of Part B.

Recall base case:
Time (un-enhanced, 1 GHz) = (6 ins)(3 CCs)(1 x 10^{-9}s) = 1.8 x 10^{-8} s = 18 ns
Time (un-enhanced, 2 GHz) = (6 ins)(3 CCs)(0.5 x 10^{-9}s) = 9.0 x 10^{-9} s = 9 ns

Time (enhanced, 1 GHz) – (4 instructions) x (3.5 CC / instruction) x (1.0 x 10^{-9} s) = 1.4 x 10^{-8} s
- Compare to last time – (1.2 x 10^{-8} s)
- Therefore, with greater CPI, enhanced version is ~16.7% slower!
  - Although still better...

Time (enhanced, 2 GHz) – (4 instructions) x (3.5 CC / instruction) x (0.5 x 10^{-9} s) = 7.0 x 10^{-9} s

Before:
- Time (un-enhanced, 2 GHz) = 9.0 x 10^{-9} s
- Time (enhanced, 2 GHz) = 6 x 10^{-9} s
- 50 % speedup

After:
- Time (un-enhanced, 2 GHz) = 9.0 x 10^{-9} s
- Time (enhanced, 2 GHz) = 7.0 x 10^{-9} s
- 28.5% speedup
- Not as good – but more realistic.
**Question 2:**
You are given two implementations of the same Instruction Set Architecture (ISA):

<table>
<thead>
<tr>
<th>Machine</th>
<th>Cycle Time</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1 ns</td>
<td>2.0</td>
</tr>
<tr>
<td>B</td>
<td>2 ns</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Part A:
What does “two implementations of the same ISA” really mean anyway?

- Instruction count will be the same
- Hence, possible instructions to translate code to is the same on both machines
  - Therefore only one way to do i++ for example
- Then, how can CPI be different?
  - 1 example:
    - memory-to-register (load); path from M → R = 2 CCs or 1 CC
    - HW / organization based – see Venn diagram

Part B:
Which machine is faster? By how much?

- \[ t_a = n \times 2.0 \times 1.0 \text{ ns} = 2.0(n) \text{ ns} \]
- \[ t_b = n \times 1.2 \times 2.0 \text{ ns} = 2.4(n) \text{ ns} \]

\[ \frac{24}{20} = 1.2X \text{ faster} \]
Part B: The Impact of the Compiler

Question 1:
A compiler designer is trying to decide between two code sequences for a particular machine. The machine supports three classes of instructions: A, B, and C.

(Note A might be ALU instructions – like Adds, B might be Jumps, and C might be Loads and Stores).

- Class A takes 3 clock cycle to execute
- Class B takes 5 clock cycles to execute
- Class C takes 6 clock cycles to execute

We now have two sequences of instructions made up of Class A, B, and C instructions respectively.

Let's assume that:
- Sequence 1 contains: 200 A’s, 100 B’s, and 200 C’s
- Sequence 2 contains: 400 A’s, 100 B’s, and 50 C’s

Questions:
- Which sequence is faster?
- By how much?
- What is the CPI of each?

Recall CPU Time = \[ \text{CPU Time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}} \]

- No information give about clock rate – therefore, we can assume its X
- Instructions / program (sequence 1) = 500
- Instructions / program (sequence 2) = 550

What's the CPI?

CPI (Seq 1) = \[
(200/500)(3) + (100/500)(5) + (200/500)(6)
= (0.4 \times 3) + (0.2 \times 5) + (0.4 \times 6)
= 4.6
\]

CPI (Seq 2) = \[
(400/550)(3) + (100/550)(5) + (50/550)(6)
= ((0.727) \times 3) + ((0.182) \times 5) + ((0.091) \times 6)
= 3.636
\]

Time (1) = 500 \times 4.6 \times X = 2300X

Time (2) = 550 \times 3.636 \times X = 2000X

Therefore, \[ \frac{2300X}{2000X} = 1.15 \times \text{X faster} \]
**Part C: Bad Benchmarks**

**Question 1:**
Two compilers are being tested for a 1 GHz machine with 3 classes of instructions A, B, and C – again, requiring 1, 2, and 3 clock cycles respectively.

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Cycles per Instruction (CPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compiler</th>
<th># of A instructions</th>
<th># of B instructions</th>
<th># of C instructions</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5 M</td>
<td>1 M</td>
<td>1 M</td>
<td>7 M</td>
</tr>
<tr>
<td>2</td>
<td>10 M</td>
<td>1 M</td>
<td>1 M</td>
<td>12 M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Cycles from type A</th>
<th>Cycles from type B</th>
<th>Cycles from type C</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(5x10^6 inst.)(1 CC/inst.) 5x10^6 CCs</td>
<td>(1x10^6 inst.)(2 CC/inst.) 2x10^6 CCs</td>
<td>(1x10^6 inst.)(3 CC/inst.) 3x10^6 CCs</td>
<td>10 M</td>
</tr>
<tr>
<td>2</td>
<td>(10x10^6 inst.)(1 CC/inst.) 10x10^6 CCs</td>
<td>(1x10^6 inst.)(2 CC/inst.) 2x10^6 CCs</td>
<td>(1x10^6 inst.)(3 CC/inst.) 3x10^6 CCs</td>
<td>15 M</td>
</tr>
</tbody>
</table>

Which sequence will produce more millions of instructions per clock cycle (MIPS)?

Seq 1 – Millions instructions/s = \( \frac{(7 \times 10^6 \text{ instructions})}{(10 \times 10^6 \text{ cycles}) \times 1 \times 10^{-9} \text{s/CC}} \)

= 700 million instructions/s

(therefore MIPS rating is 700)

Seq 2 – Millions instructions/s = \( \frac{(12 \times 10^6 \text{ instructions})}{(15 \times 10^6 \text{ cycles}) \times 1 \times 10^{-9} \text{s/CC}} \)

= 800 million instructions/s

(therefore MIPS rating is 800)

**Is sequence 2 seemingly better?**

Which sequence is faster?

CPU time\(_1\) = \( \frac{(7 \text{ M inst.}) \times ((5/7)(1) + (1/7)(2) + (1/7)(3)) \text{ CC / inst.}}{0.01 \text{ s}} \times 10^{-9} \text{ s / CC} \)

CPU time\(_1\) = \( \frac{(12 \text{ M inst.}) \times ((10/12)(1) + (1/12)(2) + (1/12)(3)) \text{ CC/inst.}}{0.015 \text{ s}} \times 10^{-9} \text{ s / CC} \)

More MIPS, more time – sequence 1 solves problem “more efficiently”
**Part D: Other Examples**

**Question 1:**
Let's assume that we have a CPU that executes the following mix of instructions:
- 43% are ALU operations (i.e. adds, subtracts, etc.) that take 1 CC
- 21% are Load instructions (i.e. that bring data from memory to a register) that take 1 CC
- 12% are Store instructions (i.e. that write data in a register to memory) that take 2 CCs
- 24% are Jump instructions (i.e. that help to implement conditionals, etc.) that take 2 CCs

What happens if we implement 1 CC stores at the expense of a 15% slower clock? Is this change a good idea?

\[
\text{CPU time (v1):} \quad = \quad (\text{# of instructions}) \times ((0.43 \times 1) + (0.21 \times 1) + (0.12 \times 2) + (0.24 \times 2)) \times \text{clock}
\]
\[
= \quad I \times (1.36) \times \text{clock}
\]

\[
\text{CPU time (v2):} \quad = \quad (\text{# of instructions}) \times ((0.43 \times 1) + (0.21 \times 1) + (0.12 \times 1) + (0.24 \times 2)) \times (\text{clock} \times 1.15)
\]
\[
= \quad I \times (1.24) \times (1.15 \times \text{clock})
\]
\[
= \quad I \times 1.426 \times \text{clock}
\]

\[
v_2 \text{ is } 1.426 / 1.36 = \approx 5\% \text{ slower}
\]

**Question 2:**
Assume that you have the following mix of instructions with average CPIs:

<table>
<thead>
<tr>
<th>% of Mix</th>
<th>Average CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>47%</td>
</tr>
<tr>
<td>Load</td>
<td>19%</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
</tr>
<tr>
<td>Store</td>
<td>14%</td>
</tr>
</tbody>
</table>

The clock rate for this machine is 1 GHz.

You want to improve the performance of this machine, and are considering redesigning your multiplier to reduce the average CPI of multiply instructions. (Digress – why do multiplies take longer than adds?)

If you make this change, the CPI of multiply instructions would drop to 6 (from 8). The percentage of ALU instructions that are multiply instructions is 23%. How much will performance improve by?
**First**, need to calculate a basis for comparison:
- Let the number of instructions = \( I \)
  - (We’re only changing the HW, not the code so the number of instructions per program will remain constant.)

Then, the CPI for this instruction mix is:

\[
\text{CPI}_{\text{avg}} = (0.47)(6.7) + (0.19)(7.9) + (0.2)(5) + (0.14)(7.1)
\]
\[
= 3.15 + 1.5 + 1 + 1
\]
\[
= 6.65
\]

\[
\text{CPU Time (base)} = I \times 6.65 \times (1 \times 10^{-9})
\]
\[
= 6.65 \times 10^{-9} (I)
\]

Next...
- To evaluate the impact of the new multiplier, we need to calculate a new average CPI for ALU instructions

*We know that the OLD ALU CPI is:*

\[
\text{CPI}_{\text{ALU-old}} = (0.23)\text{(multiply)} + (0.77)\text{(non-multiply)}
\]
\[
6.7 = (0.23)(8) + (0.77)\text{(non-multiply)}
\]

We can solve for (non-multiply) – which equals 6.31.

*Now, we can calculate a new ALU CPI:*

\[
\text{CPI}_{\text{ALU-new}} = (0.23)\text{(multiply)} + (0.77)\text{(non-multiply)}
\]
\[
= (0.23)(6) + (0.77)(6.31)
\]
\[
= 6.24
\]

Finally... we can calculate a new CPI and CPU time:

\[
\text{CPI}_{\text{new}} = (0.47)(6.24) + (0.19)(7.9) + (0.2)(5) + (0.14)(7.1)
\]
\[
= 2.68 + 1.5 + 1 + 1
\]
\[
= 6.41
\]

\[
\text{CPU Time (new)} = I \times 6.41 \times (1 \times 10^{-9})
\]
\[
= 6.41 \times 10^{-9} (I)
\]

The speedup with the new multiplier is then:  \( \frac{6.65}{6.41} \) – or 3.7%
Part E: Amdahl’s Law Examples

Question 1:
Consider 4 potential applications of the Amdahl’s Law Formula:

1. 95% of a task/program/etc. is improved by 10%
2. 5% of a task/program/etc. is improved by 10X
3. 5% of a task/program/etc. is infinitely improved
4. 95% of a task/program/etc. is infinitely improved

For all 4 cases, what is the overall speedup of the task?

Recall Amdahl’s Law Formula:
\[
\text{speedup} = \frac{1}{(1 - f_{\text{enhanced}}) + \frac{f_{\text{enhanced}}}{\text{speedup}_{\text{enhanced}}}}
\]

Case 1:
\[
\text{speedup} = \frac{1}{(1 - 0.95) + \frac{0.95}{1.1}} = 1.094
\]
- Here, there is a 9.4% speedup.
- Because the enhancement does not affect the whole program, we don't get 10% – but because it's widely applied, we get close.

Case 2:
\[
\text{speedup} = \frac{1}{(1 - 0.05) + \frac{0.05}{10}} = 1.047
\]
- Here, there is a 4.7% speedup.
- Because the enhancement is limited in scope, there is limited improvement.

Case 3:
\[
\text{speedup} = \frac{1}{(1 - 0.05) + \frac{0.05}{\infty}} = 1.052
\]
- Here, there is a 5.2% speedup.
- Same as Case 3. Because the enhancement is limited in scope, there is limited improvement.

Case 4
\[
\text{speedup} = \frac{1}{(1 - 0.95) + \frac{0.95}{\infty}} = 20
\]
- Only if enhancement almost everywhere do you see big speedup – and then only 20X!
- (If 1,000,000 – still see about 20X – therefore “lose” 50,000X of improvement)
**Question 2:**
Let’s suppose that we have 2 design options to choose from:

1. We can make part of a task 20X faster than it was before; this part of the task constitutes 10% of the overall task time.
2. We can make 85% of the task 1.2X faster.

**Part A:**
Which is better?

To answer, we need to calculate 2 parameters:
1. % of task that will run faster / how much faster it will run
2. Part of task that will be the same as before

<table>
<thead>
<tr>
<th></th>
<th>(i)</th>
<th>(ii)</th>
<th>(i) + (ii)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>0.1 / 20 = 0.005</td>
<td>1-0.1</td>
<td>0.005 + 0.9 = 0.905</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.85 / 1.2 = 0.708</td>
<td>1-0.85</td>
<td>0.708 + 0.15 = 0.858</td>
</tr>
<tr>
<td></td>
<td>Think of this column as the new component of execution time</td>
<td>This is the part of the task that takes the same as before.</td>
<td></td>
</tr>
</tbody>
</table>

Can then divide normalized old execution time by the result to get speedup:

For Case 1: \( \frac{1}{0.905} = 1.105 = 10.5\% \)

For Case 2: \( \frac{1}{0.858} = 1.166 = 16.5\% \)

Therefore Case 2 is better – b/c it improves almost *everything*

**Part B:**
Question – how much / what % of code must be sped up by 20X to match performance of Case 2?

\[
1.165 = \frac{1}{(1 - f_{\text{enhanced}}) + \frac{f_{\text{enhanced}}}{20}}
\]

If we solve for fraction_{enhanced}, we get 0.149 – i.e. 14.9% of the code/task must run 20X faster instead of 10%.
**Part F: Lab Example**

How do you use output from the SimpleScalar tool to calculate execution time?

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim_CPI</td>
<td>1.4196 # cycles per instruction</td>
</tr>
<tr>
<td>sim_IPC</td>
<td>0.7044 # instructions per cycle</td>
</tr>
<tr>
<td>sim_num_insn</td>
<td>1220754 # total number of instructions committed</td>
</tr>
<tr>
<td>sim_num_loads</td>
<td>316121 # total number of loads committed</td>
</tr>
<tr>
<td>sim_num_stores</td>
<td>151151.0000 # total number of stores committed</td>
</tr>
<tr>
<td>sim_num_branches</td>
<td>94135 # total number of branches committed</td>
</tr>
<tr>
<td>dl1.accesses</td>
<td>477290 # total number of accesses</td>
</tr>
<tr>
<td>dl1.hits</td>
<td>471924 # total number of hits</td>
</tr>
<tr>
<td>dl1.misses</td>
<td>5366 # total number of misses</td>
</tr>
<tr>
<td>dl1.miss_rate</td>
<td>0.0112 # miss rate (i.e., misses/ref)</td>
</tr>
<tr>
<td>il1.accesses</td>
<td>1637165 # total number of accesses</td>
</tr>
<tr>
<td>il1.hits</td>
<td>1636752 # total number of hits</td>
</tr>
<tr>
<td>il1.misses</td>
<td>413 # total number of misses</td>
</tr>
<tr>
<td>il1.miss_rate</td>
<td>0.0003 # miss rate (i.e., misses/ref)</td>
</tr>
</tbody>
</table>

*Can multiply number of instructions by CPI.*