Assignment #1
Fall 2022

1. For a certain important application involving matrices, analysis reveals that floating point instructions account for 40% of the total execution time, and the floating point transpose function alone is responsible for 50% of the total floating point time. As a result of this analysis, two mutually exclusive enhancements are being considered: 1) to add a FPTRANS instruction to the current floating point unit that will accelerate the transpose computation by a factor of 4, or 2) to speed up the entire floating point unit by a factor of two. Which proposal will provide the higher performance?

\[
\text{OPT 1: ADD FPTRANS INSTR}
\]
\[
\text{WILL SPEED UP 25% OF TOTAL BY } 5x
\]
\[
S = \frac{1}{1 - \frac{X_{FP} + 2\frac{X_{FP}}{5}}{5}}
\]
\[
= \frac{1}{0.75 + \frac{25}{5}} = \frac{1}{0.25} = 1.33
\]

\[
\text{OPT 2: SPEED UP FP UNIT BY } 2x
\]
\[
S = \frac{1}{0.5 + \frac{0.5}{2}} = \frac{1}{0.75} = 1.33
\]

**Option 2 is better**
2. Your task is to compare the memory efficiency of four different styles of instruction set architectures. The architecture styles are:

1. **Accumulator** - All operations occur between a single register and a memory location.

2. **Memory-memory** - All three operands of each instruction are in memory.

3. **Stack** - All operations occur on the top of the stack. Only *push* and *pop* access memory, and all other instructions remove their operands from the top of stack and replace them with the result. The implementation uses a pair of hardware registers to hold the top two entries on the stack; accesses that use other stack positions require memory references.

4. **Load-store** - All operations occur in registers, and register-to-register instructions have three operands per instruction. There are 16 general purpose registers, making the register specifiers 4 bits long.

To measure memory efficiency, assume the following about all four instruction sets:

- The opcode is always one byte (8 bits).
- All memory addresses are 2 bytes (16 bits).
- All data operands are 4 bytes (32 bits).
- All instructions are an integral number of bytes in length.

There are no other optimizations to reduce memory traffic, and the variables $A$, $B$, $C$ and $D$ are initially in memory.

Invent your own assembly language mnemonics and write the best equivalent assembly language code for the high-level-language fragments given. Write the four code sequences for:

\[
A = B + C;
\]
\[
B = A + C;
\]
\[
D = A - B;
\]

Calculate the instruction bytes fetched and the memory-data bytes transferred. Which architecture is most efficient as measured by code size? Which architecture is most efficient as measured by total memory bandwidth (code + data) required?
1. ACCUMULATOR

<table>
<thead>
<tr>
<th>OP</th>
<th>MEM</th>
<th>INST</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD B</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>ADD C</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>ST A</td>
<td>A=B+C</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>ADD C</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>ST B</td>
<td>B=A+C</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>NEG</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>ADD A</td>
<td>A-B</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>ST D</td>
<td>D=A-B</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

INST BYTES: 22
MEMORY BYTES: 20

2. MEMORY-MEMORY

<table>
<thead>
<tr>
<th>OP</th>
<th>MEM</th>
<th>MEM</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD A, B, C</td>
<td></td>
<td>ADD A, B, C</td>
<td>7</td>
</tr>
<tr>
<td>ADD B, A, C</td>
<td></td>
<td>ADD B, A, C</td>
<td>7</td>
</tr>
<tr>
<td>SUB D, A, B</td>
<td></td>
<td>SUB D, A, B</td>
<td>7</td>
</tr>
</tbody>
</table>

INST BYTES: 21
MEMORY BYTES: 36

3. STACK

<table>
<thead>
<tr>
<th>OP</th>
<th>MEM</th>
<th>MEM</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH B</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>POP A</td>
<td></td>
<td>POP A</td>
<td>3</td>
</tr>
<tr>
<td>PUSH A</td>
<td></td>
<td>PUSH</td>
<td>3</td>
</tr>
<tr>
<td>PUSH C</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td>ADD</td>
<td>3</td>
</tr>
<tr>
<td>POP B</td>
<td></td>
<td>POP B</td>
<td>3</td>
</tr>
<tr>
<td>PUSH A</td>
<td></td>
<td>PUSH A</td>
<td>3</td>
</tr>
<tr>
<td>PUSH B</td>
<td></td>
<td>PUSH B</td>
<td>3</td>
</tr>
<tr>
<td>SUB</td>
<td></td>
<td>SUB</td>
<td>3</td>
</tr>
<tr>
<td>POP D</td>
<td></td>
<td>POP D</td>
<td>3</td>
</tr>
</tbody>
</table>

INST BYTES: 30
MEMORY BYTES: 40
3. STACK (ALT)

```
PUSH B
PUSH C
ADD
ST A ; LEAVE ON STACK
PUSH C
ADD
ST B
PUSH A    | NEG
XCHG
SUB
POP D
```

```
3 3 3 1 3 3 3 0 3 1
```

```
4

4
```

```
25
```

```
28
```

4. LD - ST

```
LD R1, B
LD R2, C
ADD R3, R1, R2
ADD R1, R3, R2
SUB R4, R3, R1
ST R3, A
ST R3, B
ST R4, D
```

```
OP D
```

```
LD R1 X
```

```
4 4
```

```
4
```

```
ADD R3 R1 R2 X
```

```
3 3 0 0
```

```
ST R3 X
```

```
4 4
```

```
29 20
```