1 Data Reuse in Load-Store vs Memory-Memory machines (30 pt)

Consider two machines as described below.

Machine X
A three-address memory-memory machine takes two memory locations as sources and one memory location as the destination. For this problem, you may assume that its ISA allows the following operations: (each instruction is encoded in seven bytes, data operands are 4 bytes):
- OP M3, M1, M2 - Performs a binary operation OP on the values stored at memory locations M1 and M2 and stores the result back into memory location M3 (M3 = M1 OP M2).
- OP M3, M1, Immediate - Performs a binary operation OP on the value stored at memory location M1 and value Immediate and stores the result back into memory location M3 (M3 = M1 OP Immediate).

Machine Y
A three-address load-store machine whose sources and destination are registers. Values are loaded into registers using memory operations (The MIPS is an example of a three-address load-store machine). For this problem, you may assume that this machine has 32 registers and its ISA allows the following operations (each instruction is encoded in four bytes, as in MIPS, data operands are 4 bytes):
- OP R3, R1, R2 - Performs a binary operation OP on the values stored at registers R1 and R2 and stores the result back into register R3 (R3 = R1 OP R2).
- OP R3, R1, Immediate - Performs a binary operation OP on the value stored at registers R1 and immediate value Immediate and stores the result back into register R3 (R3 = R1 OP Immediate).
- LD R1, M - Loads the value at memory location M into register R1.
- ST R2, M - Stores the value in register R2 into memory location M.

Consider the following two high-level language code fragments.

Code Segment A
A = A + 1;
B = B + 2;
C = C + 3;

Code Segment B
B = B + A;
C = B + A;
A = B - C;

(a) For each of the two code segments A and B, write assembly language code sequences for each of the two machines X and Y. Try to reuse values in registers when possible.
(b) Compute the total number of bytes (instruction + data) transferred for each of the two code segments A and B, for each of the two machines X and Y described above. Fill in the table below with the total number of bytes transferred.

**Total number of bytes transferred**

<table>
<thead>
<tr>
<th></th>
<th>Code Segment A</th>
<th>Code Segment B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) For each code segment, (i) which machine has lower number of total bytes (instruction + data) transferred? (ii) Is this the same for both the code segments? (iii) If yes, explain why this is the case? If no, why is there a difference?

## 2 Fixed Length vs Variable Length ISAs (20 pt)

Consider the following two ISAs for a load/store machine.

1. The first is a fixed length ISA that has the following instruction encoding.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand 1 (Destination)</th>
<th>Operand 2 (Source 1)</th>
<th>Operand 3 (Source 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg/Imm</td>
<td>Reg/Imm</td>
<td>Reg/Imm</td>
<td></td>
</tr>
</tbody>
</table>

An opcode is 1 byte. Each operand is 1 byte.

All register/register and register/immediate operations take 1 cycle, while all load and store instructions take 4 cycles.

2. The second is a variable length ISA that the following instruction encoding.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand 1 (Destination)</th>
<th>Operand 2 (Source 1)</th>
<th>Operand 3 (Source 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg/Imm</td>
<td>Reg/Imm</td>
<td>Reg/Imm</td>
<td></td>
</tr>
</tbody>
</table>

The opcode is 1 byte. Each operand (register/immediate) is 1 byte. **Note that operand 3 is optional.** If an instruction does not need the third operand, it is not used. In other words, the third operand is not a part of every instruction.

The variable length of this ISA increases its decode complexity. Therefore, all instructions take one cycle longer than the previously described fixed length ISA. i.e., All register/register and register/immediate operations take 2 cycles, while all load and store instructions take 5 cycles.

Consider the following assembly language code sequence (Please see comment to see what each line of code does.)

```assembly
ADD r3, r1, r2 // r3 = r1+r2
SLL r3, 0x2 // r3 = r3 << 2
MOV r5, 0xa // r5 = 0x0a
STW r3, (r5) // MEMORY[r5] = r3
```

(a) What would be the code size for the given assembly language sequence, for each of the two ISAs?

(b) What is the number of cycles taken to execute the code sequence for each of the two ISAs?

(c) Which of these two ISAs has lower code size for the given assembly language sequence? Why?

(d) Which of these two ISAs has lower execution time for the given assembly language sequence? Why?
3 Addressing Modes (30 pt)

We covered the following addressing modes in lecture 3.

- Absolute
- Register indirect
- Based (Base + Offset)
- Indexed
- Memory indirect
- Auto increment/decrement

Consider the following high-level language code segments:

(a) `uint8_t a[100]; // a is allocated in memory
   for (i = 0; i < 100; i ++)
   {
      a[i] = 10;
   }

(b) `int a[100]; // a is allocated in memory
    for (i = 0; i < 100; i ++)
    {
      a[i] = 10;
    }

(c) `int *p; // p is allocated in memory
    *p = 100;

Assume that in the first two code segments, a register contains the address of the start of the array, and in the last snippet, a register contains the address of the pointer p.

(a) For each of the above three high-level language code segments, which of the addressing modes listed above, do you think, would lead to the minimum number of instructions (Note: No addressing mode might fit perfectly. You might require other instructions for address computation)?

(b) Write MIPS assembly code for each of the three high-level language code segments (using the addressing modes that MIPS supports).

4 Programmer vs Compiler vs Microarchitect (20 pt)

For each of the following, state who is this good for, and who is this bad for? Explain why. Please state any assumptions you make clearly.

Assume that the goal of the programmers, compilers, and microarchitect is to maximize performance.

(a) **More addressing modes**
   - HLL Programmer (who programs in a high-level language like Java):
   - Assembly Programmer (who programs in assembly language):
   - Compiler:
   - Microarchitect:

(b) **Variable length ISA**
   - HLL Programmer (who programs in a high-level language like Java):
   - Assembly Programmer (who programs in assembly language):
   - Compiler:
   - Microarchitect:
(c) **ISA orthogonality**

HLL Programmer (who programs in a high-level language like Java):
Assembly Programmer (who programs in assembly language):
Compiler:
Microarchitect:

(d) **High-level data types**

HLL Programmer (who programs in a high-level language like Java):
Assembly Programmer (who programs in assembly language):
Compiler:
Microarchitect:

5 **Addressability (15 pt)**

Say we have 32 megabytes of storage. Calculate the number of bits required to address a location if

(a) (i) the ISA is bit-addressable
   (ii) the ISA is byte-addressable
   (iii) the ISA is 32-bit addressable
   (iv) the ISA is 128-bit addressable

(b) Let us imagine for a moment that MIPS did not have the byte and halfword load/store instructions, but only **Lw** and **Sw**. Write MIPS code to access an arbitrary byte at the address in register $8$ and leave that byte (zero-extended) in register $9$ at the LSB position. You are only allowed to perform load/store accesses to addresses that are 4-byte (32-bit) aligned. Assume that the MIPS processor is little-endian, as specified in our labs.

6 **Microarchitecture vs ISA (20 pt)**

(a) Briefly explain the difference between the microarchitecture level and the ISA level in the transformation hierarchy. What information does the compiler need to know about the microarchitecture of the machine in order to compile the program correctly?

(b) Classify the following attributes of a machine as either a property of its microarchitecture or ISA:

   (i) The machine does not have a subtract instruction.
   (ii) The ALU of the machine does not have a subtract unit.
   (iii) The machine does not have condition codes.
   (iv) A 5-bit immediate can be specified in an ADD instruction.
   (v) It takes $n$ cycles to execute an ADD instruction.
   (vi) There are 8 general purpose registers.
   (vii) A 2-to-1 mux feeds one of the inputs to ALU.
   (viii) The register file has one input and two output ports.
7  Microcoded Machines (30 pt)

Microcoding is a powerful technique to perform complex computations on a simple datapath. In this problem, you will sketch out a microcode-controlled single-bus implementation of the MIPS R2000 ISA.

In a single-bus processor, the primary interconnect between state elements and functional units is a shared bus. In each cycle, only one unit can drive data on the bus, while all other units may listen and selectively latch the data. Below is the starting point of a single-bus microarchitecture.

Assume that there is a microcode control ROM that can drive all of the control signals shown based on a current state and specify a next state. As an example, below is a possible microcode sequence for the three-register ADD instruction. Assume that other states which are not shown handle instruction fetch and PC update. Thus, at the initial state shown here, the instruction register already contains the instruction.

```
<table>
<thead>
<tr>
<th>State</th>
<th>nextState</th>
<th>lePC</th>
<th>enPC</th>
<th>leIR</th>
<th>enIR</th>
<th>leA</th>
<th>leB</th>
<th>ALU</th>
<th>select_field</th>
<th>leIDX</th>
<th>RF</th>
<th>R</th>
<th>RF</th>
<th>W</th>
<th>enRF</th>
<th>leMAR</th>
<th>MEM</th>
<th>RF</th>
<th>Mem</th>
<th>W</th>
<th>Mem</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_0</td>
<td>ADD_1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Latch IR into IDX</td>
</tr>
<tr>
<td>ADD_1</td>
<td>ADD_2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Read RF[rs] into A</td>
</tr>
<tr>
<td>ADD_2</td>
<td>ADD_3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Latch rt into IDX</td>
</tr>
<tr>
<td>ADD_3</td>
<td>ADD_4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Read RF[rt] into B</td>
</tr>
<tr>
<td>ADD_4</td>
<td>ADD_5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Latch rd into IDX</td>
</tr>
<tr>
<td>ADD_5</td>
<td>Fetch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A+B into RF[rd]</td>
</tr>
</tbody>
</table>
```

For this homework problem, you will implement a new instruction, ADDM, in microcode. ADDM performs an addition where one operand is loaded from memory and the other operand comes from a register, and the result is stored back into a register. It is an R-type instruction with the following semantics:

\[
RF[rd] = M[RF[rs]] + RF[rt]
\]

In other words, the instruction loads the word in memory at the address specified by register rs, then adds the loaded value to the value in register rt, and stores the result in register rd. Write out a microcode sequence like the one in the table above for the ADDM instruction. You can assume that when accessing memory (with the MEM_R signal asserted), the microcode sequencer will stall until the memory provides the data.