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.
Assembly code sequences for code segments A and B for machines X and Y

**Code Segment A**

*Machine X*

ADD A, A, 1  
ADD B, B, 2  
ADD C, C, 3

*Machine Y*

LD R1, A  
ADD R1, R1, 1  
ST R1, A  
LD R2, B  
ADD R2, R2, 2  
ST R2, B  
LD R3, C  
ADD R3, R3, 3  
ST R3, C

**Code Segment B**

*Machine X*

ADD B, B, A  
ADD C, B, A  
SUB A, B, C

*Machine Y*

LD R1, A  
LD R2, B  
ADD R3, R1, R2  
ADD R4, R3, R1  
SUB R5, R3, R4  
ST R3, B  
ST R4, C  
ST R5, A

(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.

<table>
<thead>
<tr>
<th></th>
<th>Code Segment A</th>
<th>Code Segment B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine X</td>
<td>45</td>
<td>57</td>
</tr>
<tr>
<td>Machine Y</td>
<td>60</td>
<td>52</td>
</tr>
</tbody>
</table>

(c) For each code segment,

(i) Which machine has lower number of total bytes (instruction + data) transferred?

For code segment A, machine X has lower number of total bytes transferred. For code segment B, machine Y has lower number of total bytes transferred.

(ii) Is this the same for both the code segments?

No. For code segment A, this is machine X and for code segment B, this is machine Y.
(iii) If yes, explain why this is the case? If no, why is there a difference?

No.

Code segment A has no data reuse. It just adds an immediate value to A, B and C each, and does not reuse the values of A, B or C. A memory-memory machine needs just three instructions to express code segment A. Each instruction fetches a memory location, adds an immediate value to it and stores it back. On the other hand, a load-store machine incurs several instructions to load the values of A, B and C into registers and store them back. This is unnecessary and merely increases the number of instructions (and hence number of instruction bytes transferred) for code segment A, as there is no reuse of data.

Code segment B reuses the values of A and B. In this case, a load-store machine loads A and B into registers first and reuses the values stored in registers. On the other hand, a memory-memory machine does repeated memory accesses to memory to fetch A and B every time they are used. Therefore, a load-store machine reduces the number of data bytes transferred significantly, for code segment B, resulting in a smaller number of total bytes transferred.

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 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 (Optional)</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.)

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?

**Fixed Length ISA** - \(4 \times 4 = 16\) bytes

**Variable Length ISA** - \(1 \times 4\) (ADD instruction) +\(3 \times 3\) (other instructions) = 13 bytes

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

**Fixed Length ISA** - \(3 \times 1\) (other instructions) +\(1 \times 4\) (STW instruction) = 7 cycles

**Variable Length ISA** - \(3 \times 2\) (other instructions) +\(1 \times 5\) (STW instruction) = 11 cycles
(c) Which of these two ISAs has lower code size for the given assembly language sequence? Why?
The variable length ISA has lower code size, because it uses fewer bytes to encode the instructions in the given code segment.

(d) Which of these two ISAs has lower execution time for the given assembly language sequence? Why?
The fixed length ISA has lower execution time, because the decode complexity and hence the number of cycles to decode an instruction is smaller.

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)?

(a) Auto increment
(b) Based, Indexed
(c) Memory indirect

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

(a) `$t0 contains the address of the start of array a.

```
ADDIU $t1, $0, 10 // Load value 10 into t1
ADDIU $t2, $0, 100 // Load array size - 100 into t2
Load:
SB $t1, 0($t0) // Load value 10 into array a
```
ADDIU $t0, $t0, 1 // Increment array address
ADDIU $t2, $t2, -1 // Decrement array size counter
BNE $t2, $0, Load // If array size counter is not 0, load

(b) $t0 contains the address of the start of array a.

ADDIU $t1, $0, 10 // Load value 10 into t1
ADDIU $t2, $0, 100 // Load array size - 100 into t2
Load:
SW $t1, 0($t0) // Load value 10 into array a
ADDIU $t0, $t0, 4 // Increment array address by 4 bytes
ADDIU $t2, $t2, -1 // Decrement array size counter
BNE $t2, $0, Load // If array size counter is not 0, load

(c) $t0 contains the address of pointer p.

LW $t1, 0($t0)
ADDIU $t2, $0, 100
SW $t2, 0($t1)

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.

There is no single correct answer to this question. We wanted to see how you think based on what we saw in lecture. Below is one possible answer.

(a) More addressing modes

HLL Programmer (who programs in a high-level language like Java):
Good. Different kinds of accesses can be encoded more concisely with different instructions. Smaller code size and faster execution (assuming the complexity in microarchitecture does not decrease clock speed).
Assembly Programmer (who programs in assembly language):
Good. Different kinds of accesses can be encoded more concisely with different instructions. Smaller code size and faster execution (assuming the complexity in microarchitecture does not decrease clock speed). Possibly easier/less tedious to code accesses to arrays and other data structures in memory.
Compiler:
Possibly bad: More work, as the compiler now has to recognize code patterns that can use different addressing modes to pick the best matching mode. Also possibly good: if addressing modes are orthogonal (can be used with every instruction), the compiler does not need to generate complex address-calculation code and can encode array accesses directly.
Microarchitect:
Bad. Has to support the different addressing modes in hardware. Higher complexity, which can affect performance, if not designed carefully.

(b) Variable length ISA

HLL Programmer (who programs in a high-level language like Java):
Good. Smaller code size which could lead to higher performance (assuming the decode complexity in microarchitecture does not decrease clock speed).
Assembly Programmer (who programs in assembly language):
Good. Smaller code size which could lead to higher performance (assuming the decode complexity in microarchitecture does not decrease clock speed).
Compiler:
Good. Smaller code size. which could lead to higher performance (if the compiler leverages information about the length of different instructions).
Microarchitect:
Bad. Has to implement complex decoding logic. Optimizing the microarchitecture for performance can become more difficult due to decode complexity.

(c) ISA orthogonality
HLL Programmer (who programs in a high-level language like Java):
Doesn't matter much: the HLL programmer does not deal with the instruction set directly.
Assembly Programmer (who programs in assembly language):
Good. Ease of programming. Can use any instruction with any addressing mode.
Compiler:
Good. Can use any instruction with any addressing mode. No need to encode complex constraints in the instruction selection algorithms.
Microarchitect:
Possibly bad: Has to implement all addressing modes for all instructions. Likely more complex hardware. (Might not be too bad if address calculation is factored out as a separate step or hardware unit.) Also bad: leads to larger instructions to encode all possible addressing modes (which could increase code size and reduce performance).

(d) High-level data types
HLL Programmer (who programs in a high-level language like Java):
Depends. If high-level data types can be exposed to HLL programmer in an usable and portable way, this could lead to higher performance with hardware-accelerated datatypes. Otherwise, may not be very useful (e.g., BCD data type in x86 not used by most programmers.)
Assembly Programmer (who programs in assembly language):
Good. Can directly express a high-level data type such as, say a linked list, and process it using a single/few instructions, rather than tedious sequences of simpler instructions. No concerns about portability as in HLL case.
Compiler:
Good/Bad. Can directly express a high-level data type such as, say a linked list, and process it using a single/few instructions. But, could lead to additional complexity in instruction selection, and could potentially hinder compiler optimizations that can be done with more primitive data types.
Microarchitect:
Bad (assuming no software translation layer). Has to handle the complexity of supporting high-level data types in hardware. Increases hardware complexity and can reduce performance if not carefully designed.

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
    28 bits
(ii) the ISA is byte-addressable
    25 bits
(iii) the ISA is 32-bit addressable
    23 bits
(iv) the ISA is 128-bit addressable

21 bits

(b) Let us imagine for a moment that MIPS did not have the byte and halfword load/store instructions, but only \texttt{LW} and \texttt{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.

\begin{verbatim}
andi $10, $8, 3 // $10 = offset (0 = LSByte, 3 = MSByte)
subu $11, $8, $10 // $11 = aligned word address
lw $9, 0($11) // $9 = word containing the byte we want
sll $10, 3 // $10 = offset in bits (0 = LSByte, 24 = MSByte)
srlv $9, $10 // shift desired byte to LSByte position
andi $9, $9, 0xff // mask off the rest
\end{verbatim}

6 Microarchitecture vs ISA (20 pt)

(a) Briefly explain the difference between the microarchitecture level and the ISA level in the transformation hierarchy.

The ISA level is the interface a machine exposes to the software. The microarchitecture is the actual underlying implementation of the machine. Therefore, the microarchitecture and changes to the microarchitecture are transparent to the compiler/programmer (except in terms of performance), while changes to the ISA affect the compiler/programmer.

What information does the compiler need to know about the microarchitecture of the machine in order to compile the program correctly?

None.

(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.

ISA

(ii) The ALU of the machine does not have a subtract unit.

Microarchitecture

(iii) The machine does not have condition codes.

ISA

(iv) A 5-bit immediate can be specified in an ADD instruction.

ISA

(v) It takes \( n \) cycles to execute an ADD instruction.

Microarchitecture

(vi) There are 8 general purpose registers.

ISA

(vii) A 2-to-1 mux feeds one of the inputs to ALU.

Microarchitecture

(viii) The register file has one input and two output ports.

Microarchitecture
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.

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.

**Microcode for ADDM instruction**

<table>
<thead>
<tr>
<th>State</th>
<th>lastStateOp</th>
<th>irPC</th>
<th>irIR</th>
<th>irA</th>
<th>irB</th>
<th>irALU</th>
<th>irOP</th>
<th>irenALU</th>
<th>irselect_field</th>
<th>ireIDX</th>
<th>ireMAR_M</th>
<th>ireMAR_M</th>
<th>ireRF</th>
<th>ireMem</th>
<th>ireMemComment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDM</td>
<td>ADDM_1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>21 (rs)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Latch rs into IDX</td>
</tr>
<tr>
<td>ADDM_1</td>
<td>ADDM_2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>21 (rs)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Read RF[rs] into A</td>
</tr>
<tr>
<td>ADDM_2</td>
<td>ADDM_3</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>Latch rt into IDX</td>
</tr>
<tr>
<td>ADDM_3</td>
<td>ADDM_4</td>
<td>0</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>Read RF[rt] into B</td>
</tr>
<tr>
<td>ADDM_4</td>
<td>ADDM_5</td>
<td>0</td>
<td>0</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>Latch rd into IDX</td>
</tr>
<tr>
<td>ADDM_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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A+B into RF[rd]</td>
</tr>
</tbody>
</table>

---

8