1 SPIM Simulator (5pt)

There isn’t a solution per se to this. The expectation is that you are familiar with SPIM/XSPIM and are using it to debug your labs and homeworks.

2 ISA (25 pt)

Code size:
Each instruction has an opcode and a set of operands
- The opcode is always 1 byte (8 bits).
- All register operands are 1 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.

Memory Bandwidth:
Memory bandwidth = code size + amount of data transfer
Amount of data transfer = number of data references * 4 bytes

(a), (b)

<table>
<thead>
<tr>
<th>Instruction Set Architecture</th>
<th>Opcode</th>
<th>Operands</th>
<th>I-bytes</th>
<th>D-bytes</th>
<th>Total Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero-address</td>
<td>PUSH B</td>
<td>A</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>PUSH C</td>
<td>A</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>POP A</td>
<td></td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>PUSH A</td>
<td></td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>PUSH C</td>
<td></td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>B</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>POP B</td>
<td>A</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>PUSH A</td>
<td>B</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>PUSH B</td>
<td></td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>SUB</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>POP D</td>
<td></td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td>36</td>
<td>66</td>
</tr>
</tbody>
</table>
Instruction Set Architecture | Opcode | Operands | I-bytes | D-bytes | Total Bytes
--- | --- | --- | --- | --- | ---
One-address
| LOAD | B | 3 | 4 |
| ADD | C | 3 | 4 |
| STORE | A | 3 | 4 |
| ADD | C | 3 | 4 |
| STORE | B | 3 | 4 |
| LOAD | A | 3 | 4 |
| SUB | B | 3 | 4 |
| STORE | D | 3 | 4 |
|  |  |  |  |  | 24 32 56
Two-address
| SUB | A, A | 5 | 12 |
| ADD | A, B | 5 | 12 |
| ADD | A, C | 5 | 12 |
| SUB | B, B | 5 | 12 |
| ADD | B, A | 5 | 12 |
| ADD | B, C | 5 | 12 |
| SUB | D, D | 5 | 12 |
| ADD | D, A | 5 | 12 |
| SUB | D, B | 5 | 12 |
|  |  |  |  |  | 45 108 153
Three-address
Memory-Memory
| ADD | A, B, C | 7 | 12 |
| ADD | B, A, C | 7 | 12 |
| SUB | D, A, B | 7 | 12 |
|  |  |  |  |  | 21 36 57
Load-Store
| LD | R1, B | 4 | 4 |
| LD | R2, C | 4 | 4 |
| ADD | R1, R1, R2 | 4 |
| ST | R1, A | 4 | 4 |
| ADD | R3, R1, R2 | 4 |
| ST | R3, B | 4 | 4 |
| SUB | R3, R1, R3 | 4 |
| ST | R3, D | 4 | 4 |
|  |  |  |  |  | 32 20 52

(c) The three-address memory-memory machine is the most efficient as measured by code size - 21 bytes.

(d) The three-address load-store machine is the most efficient as measured by total memory bandwidth (code + data) - 52 bytes.

3  MIPS Assembly (50 pt)

3.1  MIPS assembly function foo() (25 pt)

```assembly
foo: int foo(int *A, int n){
    int s;
    addiu $sp, $sp, -12 // allocate stack
    sw $16, 0($sp)
    add $16, $4, $0 // r16 for A
    sw $17, 4($sp) // use callee saved
    add $17, $5, $0 // r17 for n
```
sw  $31, 8($sp) // save return address

_branch: if (n >= 2) {
    slt  $2, $16, 2 // use $2 as temp
    bne $2, $0, false

__true:
    s=foo(A,n-1)
    addi $5, $17, -1 // $4=A, $5=n-1
    jal foo

    sll $3, $17, 2 // use $3 as temp; s=s+A[n-2];
    add $3, $3, $16 // compute &A[n] + compute byteoffset n*4
    lw $3, -8($3) // load A[n-2]
    add $2, $2, $3 // compute s=s+A[n-2]
    j _join // jump to merge

__false: else {

    addi $2, $0, 1 // set s=1
    s = 1;
}

__join:
    sll $3, $17, 2 // compute byteoffset n*4
    add $3, $3, $16 // compute &A[n]
    sw $2, 0($3) // store s ($2) to A[n]

__done:
    lw $31, 8($sp) // restore $31
    lw $17, 4($sp) // restore $17
    lw $16, 0($sp) // restore $16
    addiu $sp, $sp, 12 // restore stack pointer
    jr $31 // return to caller

3.2 MIPS assembly for REP MOVSB (25 pt)
Assume: $1 = ECX, $2 = ESI, $3 = EDI

(a) beq $1, $0, AfterLoop // If counter is zero, skip

CopyLoop:
    lb $4, 0($2) // Load 1 byte
    sb $4, 0($3) // Store 1 byte
    addiu $2, $2, 1 // Increase source pointer by 1 byte
    addiu $3, $3, 1 // Increase destination pointer by 1 byte
    addiu $1, $1, -1 // Decrement counter
    bne $1, $0, CopyLoop // If not zero, repeat

AfterLoop:
    Following instructions

(b) The size of the MIPS assembly code is 4 bytes × 7 = 28 bytes, as compared to 2 bytes for x86 REP MOVSB.

(c) The count (value in ECX) is 0x01000000 = 16777216. Therefore, loop body is executed (16777216 * 6) = 100663292. Total instructions executed = 100663292 + 1 (beq instruction outside of the loop) = 100663297.

(d) The count (value in ECX) is 0x0 = 0. Therefore, loop body is executed (0 * 6) = 0. Total instructions executed = 0 + 1 (beq instruction outside of the loop) = 1
4 Performance (15 pt)

For P&H Fourth Edition
1.5.4 a) Execution time = ((1000 × 1) + ((400 + 100) × 10) + (50 × 3))/3e9 = 2.05μ seconds
1.5.4 b) Execution time = ((1500 × 1) + ((300 + 100) × 10) + (100 × 3))/3e9 = 1.93μ seconds

1.5.5 a) Execution time = ((1000 × 1) + ((400 + 100) × 2) + (50 × 3))/3e9 = 0.72μ seconds
1.5.5 b) Execution time = ((1500 × 1) + ((300 + 100) × 2) + (100 × 3))/3e9 = 0.87μ seconds

1.5.6 a) Execution time (when compute reduced in half) = ((500 × 1) + ((400 + 100) × 2) + (50 × 3))/3e9 = 0.55μ seconds
Speedup = 0.72/0.55 = 1.30

1.5.6 b) Execution time (when compute reduced in half) = ((750 × 1) + ((300 + 100) × 2) + (100 × 3))/3e9 = 0.62μ seconds
Speedup = 0.87/0.62 = 1.40

For P&H Revised Fourth Edition
1.5.4 a) Execution time = ((600 × 1) + ((600 + 200) × 10) + (50 × 3))/3e9 = 2.91μ seconds
1.5.4 b) Execution time = ((900 × 1) + ((500 + 100) × 10) + (200 × 3))/3e9 = 2.5μ seconds

1.5.5 a) Execution time = ((600 × 1) + ((600 + 200) × 2) + (50 × 3))/3e9 = 0.78μ seconds
1.5.5 b) Execution time = ((900 × 1) + ((500 + 100) × 2) + (200 × 3))/3e9 = 0.9μ seconds

1.5.6 a) Execution time (when compute reduced in half) = ((300 × 1) + ((600 + 200) × 2) + (50 × 3))/3e9 = 0.68μ seconds
Speedup = 0.78/0.68 = 1.14

1.5.6 b) Execution time (when compute reduced in half) = ((450 × 1) + ((500 + 100) × 2) + (200 × 3))/3e9 = 0.75μ seconds
Speedup = 0.9/0.75 = 1.20

For P&H Fourth Edition and P&H Revised Fourth Edition
1.12.4 CPU time = No of Instructions × CPI/Clock rate
If CPI and clock rate do not change, the CPU time increase is equal to the increase in the number of instructions, that is 10%.

1.12.5 CPU time (before) = No of Instructions × CPI/Clock rate
CPU time (after) = 1.1 × No of Instructions × 1.05 × CPI/Clock rate = 1.155 Therefore, the CPU time is increased by 15.5%.

5 Subroutine Call (15 pt)

2 and 3 support subroutine nesting; 3 supports recursion.

In 1, the return address is overwritten in the processor register, on the call to the nested subroutine. So, 1 does not support either subroutine nesting or recursion.
In 2, the return address is stored in a different memory location for each subroutine. Therefore, it can support nesting (a subroutine can call another subroutine). However, a subroutine cannot call itself, as this would write the return address into the same memory location.

In 3, the return address is pushed on the stack. So, return addresses can be pushed on the stack during both nested and recursive calls and popped off as each nested/recursive call returns.