Instruction Set Architecture

Notation:
The numeric value \( x \) is denoted by \(#x\).
Indirect addressing is denoted by \((a)\): the (memory) location whose address is contained in \( a \).
\( ra \), etc. denote registers, \( PC \) is the programme counter and \( AC \) is the accumulator register.
Data movement is denoted by \( a \leftarrow b \): the value of \( b \) is placed in \( a \).

We present a list of instructions typical of a RISC (reduced set instruction computer) machine. Most architectures use two addresses, where the first (or second) argument serves both as the target and one of the sources, e.g., \( \text{ADD } ra, rb \) means \( ra \leftarrow ra + rb \). In the load and store instructions, the addresses may be immediate, direct, register or register indirect addresses. The other instructions use register addressing.

\[
\begin{align*}
\text{LOAD } a1, a2 & \quad a1 \leftarrow a2 \\
\text{STOR } a1, a2 & \quad a1 \leftarrow a2 \\
\text{ADD } ra, rb, rc & \quad ra \leftarrow rb + rc \\
\text{SUB } ra, rb, rc & \quad ra \leftarrow rb - rc \\
\text{MUL } ra, rb, rc & \quad ra \leftarrow rb \times rc \\
\text{DIV } ra, rb, rc & \quad ra \leftarrow rb / rc \\
\text{AND } ra, rb, rc & \quad ra \leftarrow rb \& rc \\
\text{OR } ra, rb, rc & \quad ra \leftarrow rb \mid rc \\
\text{NOT } ra, rb & \quad ra \leftarrow \neg rb \\
\text{ASH } ra, rb, rc & \quad ra \leftarrow rb \text{ (arbitrarily) shifted by } rc \text{ positions} \\
\text{LSH } ra, rb, rc & \quad ra \leftarrow rb \text{ (logically) shifted by } rc \text{ positions} \\
\text{BR } ra & \quad PC \leftarrow ra \\
\text{BEQ } ra, rb, rc & \quad PC \leftarrow ra \text{ if } rb \text{ is equal to } rc \\
\text{BNE } ra, rb, rc & \quad PC \leftarrow ra \text{ if } rb \text{ is not equal to } rc \\
\text{BLT } ra, rb, rc & \quad PC \leftarrow ra \text{ if } rb \text{ is less than } rc \\
\text{BGT } ra, rb, rc & \quad PC \leftarrow ra \text{ if } rb \text{ is greater than } rc \\
\text{BLE } ra, rb, rc & \quad PC \leftarrow ra \text{ if } rb \text{ is less than or equal to } rc \\
\text{BGE } ra, rb, rc & \quad PC \leftarrow ra \text{ if } rb \text{ is greater than or equal to } rc
\end{align*}
\]

Most instructions have a floating-point version when special floating-point registers are used. Some instructions have a version where a specific value is the operand (immediate addressing), e.g., \( \text{MOV } ra, #X \) means \( ra \leftarrow #X \) and \( \text{BR } #I1 \) means to jump to instruction \( I1 \) (i.e., load it into \( PC \)).

We will use a five-stage pipeline: IF (instruction fetch), ID (instruction decode), RR (register read), EX (execute instruction), WB (write back result into register). We will assume that for data-movement instructions the data transfer between the CPU and main memory happens in the execute stage. Note that for some instructions (e.g., \( \text{MOV } ra, #X \)) some of the pipeline stages (e.g., RR) are not needed.
Example

Write an assembly program that computes \(((10 \times 8) + 4) - 7)^2\) and show the delay slots when executed on a five-stage pipeline. Try to minimize the number of registers used.

We make the following assumptions:

1. Instructions are executed in the order they appear in the code.

2. Instructions like `MOV r1, #10` skip the RR and WB stages.

3. If an instruction cannot be further processed (because of some dependency) then it stalls the pipeline.

Here is the code:

I1  MOV r1, #10
I2  MOV r2, #8
I3  MUL r1, r1, r2
I4  MOV r2, #4
I5  ADD r1, r1, r2
I6  MOV r2, #7
I7  SUB r1, r1, r2
I8  MUL r1, r1, r1

Here is the diagram showing delay slots:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>I7</td>
<td>I8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>I7</td>
<td>I8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>I7</td>
<td>I8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>I7</td>
<td>I8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>I7</td>
<td>I8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Exercise**

Consider the computation \((M_1 \times M_2) + M_3 - M_4\) where \(M_i\) indicates the content of a memory location. Write an assembly program for the above computation using minimum number of registers. Show the delay slots when the code is executed on a five-stage pipeline.

You can make the following assumptions:

1. Instructions are executed in the order they appear in the code.
2. Instructions like `MOV r1, M1` skip the RR stage.
3. There is an internal cache where decoded instructions can be stored until they can be further processed.

**Comprehensive Exercise**

Consider the computation \((M_1 \times M_2) + (M_3 \times M_4)\).

1. Write a program in assembly language that performs the above computation. Try to use a minimal number of registers (assume that initially all registers contain 0).

2. Draw a diagram showing the execution of your program in a five-stage pipeline using in-order-issue in-order-completion.

3. Identify the dependencies in your code.

4. Draw a diagram showing the execution of your program in a five-stage pipeline using in-order-issue out-of-order-completion.

5. Remove as many dependencies as possible from your code (by using register renaming) and reorder the code to minimize the number of delay slots when executed on a five-stage pipeline. Show the pipeline activity in a diagram.

You can assume that there is no resource conflict between fetching an instruction and executing a data transfer instruction.