Instruction Set Architecture

We present a list of instructions typical of a RISC (reduced set instruction computer) machine. In data-movement and control instructions, the addresses may be immediate #X, direct memory M, register r, or register indirect (r) addresses. Data-processing instructions use immediate or register addressing. PC is the programme counter and a <- b indicates that the value of b is placed in a.

```
LOAD a, b       a <- b
STOR a, b       a <- b
ADD a, b, c     a <- b + c
SUB a, b, c     a <- b - c
MUL a, b, c     a <- b * c
DIV a, b, c     a <- b / c
AND a, b, c     a <- b & c
OR a, b, c      a <- b | c
NOT a, b        a <- !b
ASH a, b, c     r <- b (arithmetically) shifted by c positions
LSH a, b, c     a <- b (logically) shifted by c positions
BR a            PC <- a
BEQ a, b, c     PC <- a if b is equal to c
BNE a, b, c     PC <- a if b is not equal to c
BLT a, b, c     PC <- a if b is less than c
BGT a, b, c     PC <- a if b is greater than c
BLE a, b, c     PC <- a if b is less than or equal to c
BGE a, b, c     PC <- a if b is greater than or equal to c
```

Most instructions have floating-point versions when special floating-point registers are used (but we will not need these).

Most architectures use two addresses, where the first (or second) argument serves both as the target and one of the sources, e.g., ADD ra, rb means ra <- ra + rb. For branch instructions BR X means to jump to instruction X (i.e., load the address of instruction X 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., LOAD 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. Assume that when an instruction cannot be further processed (e.g., because of some dependency or resource conflict), then it stalls the pipeline.

Here is the code:

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

We make the following assumptions:

1. Instructions are processed strictly in the order they appear in the code.
2. Instructions like LOAD r1, #10 skip the RR and EX stages, and 10 is written into r1 in the WB stage.

Here is the diagram showing delay slots:

```

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| IF| I1| I2| I3| I4| I5| I5| I5| I5| I5| I5  | I5  | I5  | I5  | I5  | I5  | I5  | I5  | I5  |
| ID| I1| I2| I3| I4| I4| I5| I5| I5| I5| I5  | I5  | I5  | I5  | I5  | I5  | I5  | I5  | I5  |
| RR|   |   |   |   | I3| I5| I5| I5| I5| I5  | I5  | I5  | I5  | I5  | I5  | I5  | I5  | I5  |
| EX| I1| I2| I3| I4| I5| I5| I5| I5| I5| I5  | I5  | I5  | I5  | I5  | I5  | I5  | I5  | I5  |
| WB| I1| I2| I3| I4| I5| I5| I5| I5| I5| I5  | I5  | I5  | I5  | I5  | I5  | I5  | I5  | I5  |
```

Comments: In cycles 14 and 15, I8 has to wait (r1 is not ready yet).

Alternatively, we can assume that an instruction can jump the queue, provided that this does not alter the outcome of the computation.

```

<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>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
</tr>
<tr>
<td>ID</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I4</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
</tr>
<tr>
<td>RR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I3</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
</tr>
<tr>
<td>EX</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
</tr>
<tr>
<td>WB</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I5</td>
</tr>
</tbody>
</table>
```

Comments: In cycle 7, I5 has to wait (r1 is not ready yet), in cycle 10, I7 has to wait (r2 is not ready yet).
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.

Assume that there is an internal cache, where fetched and decoded instructions can be stored until they can be further processed.

Comprehensive Exercise

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

1. Write a program in assembly language that performs the above computation. Try to use a minimal number of registers.

2. Draw a diagram showing the execution of your program on 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 on 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 an onboard instruction cache from which the instructions are fetched (so there is no resource conflict between fetching an instruction and executing a data-transfer instruction).