Problem 1: New Instruction Encoding

In this problem we would like you to design a new instruction encoding for the set of RISC-V instructions discussed in lecture. The goal of this new encoding is that all instructions have the same type of encoding and that fields in the instruction encoding are not used for different purposes on different instructions. Obviously, to achieve this goal, the instruction encoding will require more than 32 bits. Present a proposed encoding that minimizes the size of each instruction. Showing the format with the various fields and their respective size in bits. Justify your choices.

Solution:

We need 52 bits total. The field size breakdown is as follows:

- opcode: 7 bits
- rd: 5 bits
- rs1: 5 bits
- rs2: 5 bits
- imm: 20 bits
- funct: 10 bits

I/S/B type instructions use 12 bits for immediate while U/J type instructions use 20 bits, so we make the immediate field be 20 bits. I/S/B type instructions use 3 bits for the function field while R type instructions use 10 bits, so we make the function field be 10 bits.

(if optimized we can remove the function field and use the opcode to hardcode the function field and reduce the opcode to the number of bits necessary to encode all possible risc-v instructions, which is \[ \lceil \log_2 47 \rceil = 6 \] for a total of 41 bits instead.)

Problem 2: CPU Architecture

Consider the operation of a pipelined RISC-V CPU that uses a pipelined instruction and a pipelined data cache. The CPU has the following characteristics:
• Although the caches can accept a new address and start a new operation each cycle, they each have a 2 cycle latency and results from a memory read are available at the end of the second cycle.

• The CPU has 7-stage pipeline: IF1, IF2, ID, EX, DM1, DM2, WB.

• Branch resolution is in the ID stage and the pipeline has no branch-predictor. The branch target address is available at the end of the ID stage.

• Unlike the normal RISC-V, the CPU has no architected branch-delay slot, and no architected load-delay slot.

• Stage bypassing (data forwarding) is implemented wherever possible.

• Assume a 100% hit rate in the caches.

• The clock frequency is 1 GHz.

Listed below is a program to be run on the CPU. Without rearranging the code, calculate the performance of the CPU—in total time to run the program. Describe your approach and show your work.

```assembly
add $4, $0, $0
addi $1, $0, 2
addi $2, $0, 0xF000
loop: lw $3, 0($2)
add $4, $4, $3
addi $1, $1, -1
beq $1, $0, done
addi $2, $2, 4
beq $0, $0, loop
done: sw $4, 0($2)
```

**Solution:**

<table>
<thead>
<tr>
<th>Instruction No.</th>
<th>Instruction</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>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>add $4, $0, $0</td>
<td>IF1</td>
<td>IF2</td>
<td>ID</td>
<td>EX</td>
<td>DM1</td>
<td>DM2</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
</tr>
<tr>
<td>2</td>
<td>add $1, $0, 2</td>
<td>IF1</td>
<td>IF2</td>
<td>ID</td>
<td>EX</td>
<td>DM1</td>
<td>DM2</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
</tr>
<tr>
<td>3</td>
<td>lw $3, 0($2)</td>
<td>IF1</td>
<td>IF2</td>
<td>ID</td>
<td>EX</td>
<td>DM1</td>
<td>DM2</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
</tr>
<tr>
<td>4</td>
<td>add $4, $4, $3</td>
<td>IF1</td>
<td>IF2</td>
<td>ID</td>
<td>EX</td>
<td>DM1</td>
<td>DM2</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
</tr>
<tr>
<td>5</td>
<td>add $1, $1, -1</td>
<td>IF1</td>
<td>IF2</td>
<td>ID</td>
<td>EX</td>
<td>DM1</td>
<td>DM2</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
</tr>
<tr>
<td>6</td>
<td>beq $1, $0, done</td>
<td>IF1</td>
<td>IF2</td>
<td>ID</td>
<td>EX</td>
<td>DM1</td>
<td>DM2</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
</tr>
<tr>
<td>7</td>
<td>lw $3, 0($2)</td>
<td>IF1</td>
<td>IF2</td>
<td>ID</td>
<td>EX</td>
<td>DM1</td>
<td>DM2</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
</tr>
<tr>
<td>8</td>
<td>add $2, $2, 4</td>
<td>IF1</td>
<td>IF2</td>
<td>ID</td>
<td>EX</td>
<td>DM1</td>
<td>DM2</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
</tr>
</tbody>
</table>

**Solution:**

9: beq $0, $0, loop
10: beq $0, $0, done
11: beq $0, $0, done
12: beq $0, $0, done
13: beq $0, $0, done
14: beq $0, $0, done
15: beq $0, $0, done
16: beq $0, $0, done
Problem 3: Processor Design

For this problem you will be dealing with a very simple processor with 8-bit instructions, 8 general purpose registers (r0–r7), and a special “accumulator” register, acc. Every instruction is of the same type and comprises an operation code, op, and two register specifiers, src and dst, identifying a source register and a destination register:

```
  7 5 4 3 2 1 0
```

Each instruction executes in the following way: Regfile[dst], acc ← acc OP Regfile[src].

- The result of the operation is stored both in the accumulator and into a general purpose register.
- OP can be one of: add, mult, sub, & pass. The operation pass sends the src register to the accumulator and to the destination register.
- As with the RISC-V, r0 always reads as all 0’s, and writes to it have no effect.

Below is a simple program for computing $a + b \times x$.

For this problem we will not need to specify how instructions get into the instruction memory and how values initially get into the register file.

```
// assembler syntax: operation dst, src
// Assume r1 = x, r2 = a, r3 = b
// Leave result in r4.
pass r0, r1 // copy x to acc
mult r0, r3 // b*x
add r4, r2 // + a
```

1. In the space below write a program (using the assembly syntax demonstrated above) to compute $a + b \times x + c \times x^2$. Try to minimize the number of instructions.

```
// Assume r1 = x, r2 = a, r3 = b, r4 = c
// Leave result in r7.
```
Solution:

```plaintext
// Assume r1 = x, r2 = a, r3 = b, r4 = c
// Leave result in r7.
pass r0, r4 // c in acc
mult r0, r1 // c*x
add r0, r3 // c*x + b
mult r0, r1 // (c*x + b)*x
add r7, r2 // (c*x + b)*x + a = cx^2 + bx + a
```

2. You are given as a library of building blocks a set of block generators (shown below), along with N-bit registers, and simple logic gates and multiplexors:

(a) The instruction memory has asynchronous read.

(b) The true dual-ported memory has asynchronous read and synchronous write. It has internal bypassing, therefore reads and write to the same location on the same cycle result in the read data being equal to the write data (not the old contents).

(c) The ALU ctrl signal determines the ALU operation: +, -, *, or pass A. ALU inputs A and B, and output Y all have the same width (we will ignore overflow in this problem).

Draw a circuit diagram for the entire processor. Indicate the parameters settings for any memory and ROM blocks that you use.

Solution:
3. Now, assume the following delays: memory blocks read or write $\tau = 10$ (independent of size); ALU $\tau = 10$ (independent of width); simple gates and multiplexors $\tau = 1$; register $\tau_{\text{setup}} = 1, \tau_{\text{clk-to-q}} = 1$.

Change your design to increase its frequency of operation by at least a factor of 2. Show your new circuit below.

<table>
<thead>
<tr>
<th>Solution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use at least 3 stage pipeline (Fetch, Decode, and Execute).</td>
</tr>
</tbody>
</table>
4. Now, instead of a processor, we want to build a specialized circuit for computing $a + b \times x + c \times x^2$. All of $a$, $b$, $c$, and $x$ are available as circuit inputs. Using the same library of building blocks, as above, draw a specialized circuit that would minimize the latency of the calculation.

Solution:
Latency is $10 + 10 + 10 = 30$. 