1 RISC-V assembly coding (6 points)

In this exercise we use the RV32I Instruction Set Architecture (ISA) without the multiplication and division extension. We code according the ILP32 Application Binary Interface (ABI) seen during the lectures and the labs, without any exception: any function that we write can be called from any piece of software about which we only know that it 100% respects the same ABI. If needed use the provided RISC-V reference card on assembly language syntax.

In order to check if an unsigned number $N$ is a multiple of 7, without divisions, we can use the following algorithm:

1. If $N$ equals 0 or 7 answer "yes" and stop,
2. if $N$ is less or equal 13 answer "no" and stop,
3. if $N$ is even divide it by 2,
4. else (if $N$ is odd) divide it by 2 and add 4,
5. goto step 1.

Note: in step 4 the division is an integer division ($17/2 = 8$); remember that you cannot use the multiplication and division instructions.

1. In RV32I assembly language code a function `step` that takes a 32-bits unsigned number in register $a0$, implements steps 3 and 4 of the algorithm, and returns the modified number in register $a0$. Examples: `step(5) = 6`, `step(28) = 14`, `step(127) = 67`.

2. In RV32I assembly language code a function `is_multiple_of_7` that implements the complete algorithm and that uses function `step` for steps 3 and 4. `is_multiple_of_7` takes a 32-bits unsigned number in register
a0 and returns 1 in register a0 if the number is a multiple of 7, else 0. Examples: \texttt{is\_multiple\_of\_7(0)} = 1, \texttt{is\_multiple\_of\_7(5)} = 0, \texttt{is\_multiple\_of\_7(28)} = 1, \texttt{is\_multiple\_of\_7(127)} = 0.

3. Assuming each instruction takes exactly one clock cycle to execute what is the Best Case Execution Time (BCET) of your \texttt{is\_multiple\_of\_7} function? For what input value?

4. Assuming each instruction takes exactly one clock cycle to execute what is the Worst Case Execution Time (WCET) of your \texttt{is\_multiple\_of\_7} function? For what input value?

2 Branch prediction (6 points)

Definitions and notations

- Miss-Prediction per executed Branch Instruction (MPBI): the number of times a given branch instruction has been wrongly predicted divided by the total number of times this same branch instruction has been executed. The lower the MPBI, the better the prediction.
- \( M_\infty \): for a given branch instruction, the limit of the MPBI when the number of times the branch instruction is executed tends to infinity, if this limit exists. Undefined if it does not exist.
- Branch outcome: the actual decision (not the prediction) for a given branch instruction; Taken or Not taken, denoted \( T \) and \( N \), respectively.
- Periodic infinite sequence of outcomes: a sequence of outcomes that starts with a finite sequence, the stem, which can be empty, and continues with a finite cycle that repeats infinitely. We represent these sequences as \( \text{STEM(CYCLE)}^* \) where \( \text{STEM} \) is the shortest possible stem and \( \text{CYCLE} \) is the shortest possible cycle. Example: \( T\ N\ N\ N\ T\ T\ T\ T\ T\ \ldots \) is a periodic infinite sequence of outcomes, its shortest possible stem is \( T\ N\ N\ T\ T\ T\ T\ T\ \ldots \), its shortest possible cycle is \( N\ N\ N\ T\ T\ T\ T\ T\ \ldots \) and we represent it as \( T\ N\ N\ T\ T\ T\ T\ T\ \ldots \).

Questions

A branch instruction is predicted using the 2-bits saturating counter (4-states) branch predictor studied during the lecture on pipelines and represented on Figure 1. We assume that the predictor is initialized in the Strong Taken (ST) state and that there is no collision with other branch instructions: the predictor is only used to predict the branch instruction of interest.

![Figure 1: Saturating counter](image-url)
1. Imagine an infinite sequence of outcomes of the branch instruction such that the 2-bits saturating counter has a MPBI equal to 1 (that is, it always predicts wrongly). Represent your sequence using the STEM(CYCLE) notation.

2. Imagine a RV32IM assembly code snippet with a branch instruction that would produce such a sequence. Label $B_1$ the branch instruction of interest. Instead of the 2-bits saturating counter we decide to use the variant predictor, also studied during the lecture on pipelines and represented on Figure 2.

![Variant](image)

3. We assume that the predictor is initialized in the Weak Not taken (WN) state and that there is no collision with other branch instructions: the predictor is only used to predict the branch instruction of interest.

4. Imagine an infinite sequence of outcomes of a branch instruction such that the variant predictor has a MPBI equal to 1. Represent your sequence using the STEM(CYCLE) notation.

5. Instead of the 2-bits saturating counter we decide to use a two-levels prediction strategy with 2-bits local Branch History Shift Registers (BHSR). We still assume that there are no collisions.

6. What would be the $M_\infty$ for branch instruction $B_1$ in your first code snippet, with 2-bits saturating counters?

7. What would be the $M_\infty$ for branch instruction $B_2$ in your second code snippet, with variant predictors?

3 Binary representation of data (4 points)

There are several ways to represent signed integers using bits. In computer systems, the two most frequently encountered are sign and magnitude and two’s complement. In the following we denote $a_{n-1}a_{n-2}\ldots a_1a_0$ the $n$-bits representation of integer $A$. In sign and magnitude $a_{n-1}$ is the sign bit. In two’s complement $a_{n-1}$ is the Most Significant Bit (MSB). In both representations $a_0$ is the Least Significant Bit (LSB).

1. Consider decimal values 12, -59 and -66. We want to represent them all in two’s complement on the same number of bits $m$. What is the minimum value of $m$?
2. Consider decimal values 12, -59 and -66. Convert them in \( m \)-bits two's complement (where \( m \) is your answer to the preceding question).

3. A \( p \)-bits adder is a hardware device that takes two \( p \)-bits inputs, adds them as if they were unsigned integers, and outputs the \( p + 1 \)-bits result. We denote \( A = a_{p-1} \ldots a_1a_0 \), \( B = b_{p-1} \ldots b_1b_0 \) the two \( p \)-bits inputs and \( S = s_p \ldots s_2s_1s_0 \) the \( p + 1 \)-bits output of a \( p \)-bits adder. Example: with a 3-bits adder, if inputs are \( A = 101 \) and \( B = 011 \), the output is \( S = 1000 \). If, instead of considering the inputs and the output as unsigned integers, we consider them as signed numbers represented in *sign and magnitude*, the result is sometimes correct, sometimes not.

- Give an example of two 3-bits *sign and magnitude* inputs for which the output of a 3-bits adder is the correct 4-bits *sign and magnitude* representation of their sum.
- Give an example of two 3-bits *sign and magnitude* inputs for which the output of a 3-bits adder is not the correct 4-bits *sign and magnitude* representation of their sum.
- Express the necessary and sufficient condition on inputs \( A = a_{p-1} \ldots a_1a_0 \) and \( B = b_{p-1} \ldots b_1b_0 \) such that a \( p \)-bits adder outputs the correct *sign and magnitude* representation of their sum.

4. What is the tetradecimal (base 14, symbols 0, 1, 2 \ldots 9, A, B, C, D) representation of decimal value 604?

## 4 Instruction Set Architecture (4 points)

The RV32IM Instruction Set Architecture (ISA) is the one we studied during the lectures and the labs. RV32IM means RISC-V 32 bits Integer with Multiplication and division extension.

1. In the RV32IM ISA there are 32 General Purpose Registers (GPRs). Could we rework this ISA to add more GPRs? What would be the limitations?
2. What is an addressing mode?
3. What addressing modes are supported by the RV32IM ISA?
4. Give an example of an addressing mode that is not supported by the RV32IM ISA.

## 5 RISC-V 5-stages pipeline (4 points)

In the 5-stages RISC-V pipeline represented on Figure 3 we assume that:

- There are no structural pipeline hazards.
- Nothing has been done to solve the other types of pipeline hazards.

We run the assembly program shown on Listing 1.
Figure 3: A 5-stages RISC-V pipeline

1. Identify the various pipeline hazards that may lead to unexpected behavior.
2. For each pipeline hazard:
   - In which class of pipeline hazards does it fall?
   - What technique is the best to deal with it?

Listing 1: Assembly program

1. \texttt{sw t0,0 (t1)} \quad \# \text{mem}[0+t1] \leftarrow t0
2. \texttt{add t0,t0,t2} \quad \# t0 \leftarrow t0+t2
3. \texttt{lw t3,0(t0)} \quad \# t3 \leftarrow \text{mem}[0+t0]
4. \texttt{beq s0,t0,label} \quad \# \text{if s0==t0 \ goto label}
5. \texttt{lw s1,0(s1)} \quad \# s1 \leftarrow \text{mem}[0+s1]
6. \texttt{andi t0,t0,0xff} \quad \# t0 \leftarrow t0 \text{ AND } 0xff
7. \texttt{label: t0,t0,0xff}
8. \ldots
<table>
<thead>
<tr>
<th>Type</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit instruction format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register File</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Aliases</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load/Store Operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical Operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic Operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branching</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pseudo Instructions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Erik Engheim <erik.engheim@ma.com>  

RISC-V Instruction-Set