ES4 EE3400 02/04/2025

These exercises are from H&P, Chapter 4. References like §4.8 point to the sections of that book. The questions related to RISC-V pipeline as described in the book, unless stated otherwise.

1. [4.25] Consider the following loop.

```
LOOP: lw lw
add addi bnez
x10, 0(x13)
x11, 8(x13)
x12, x10, x11
x13, x13, 16
x12, LOOP
```

Assume that perfect branch prediction is used (no stalls due to control hazards), that there are no delay slots, that the pipeline has full forwarding support, and that branches are resolved in the EX (as opposed to the ID) stage.
4.25.1 [10] <§4.8> Show a pipeline execution diagram for the first two iterations of this loop.
4.25.2 [10] <§4.8> Mark pipeline stages that do not perform useful work. How often while the pipeline is full do we have a cycle in which all five pipeline stages are doing useful work? (Begin with the cycle during which the addi is in the IF stage. End with the cycle during which the bnez is in the IF stage.)

2. [4.27] Problems in this exercise refer to the following sequence of instructions, and assume that it is executed on a five-stage pipelined datapath:

```
add x15, x12, x11
lw x13, 8(x15)
lw x12, 0(x2)
or x13, x15, x13
sw x13, 0(x15)
```

4.27.1 [5] <§4.8> If there is no forwarding or hazard detection, insert NOPs to ensure correct execution.
4.27.2 [10] <§4.8> Now, change and/or rearrange the code to minimize the number of NOPs needed. You can assume register x17 can be used to hold temporary values in your modified code.
4.27.3 [10] <§4.8> If the processor has forwarding, but we forgot to implement the hazard detection unit, what happens when the original code executes?
4.27.4 [20] <§4.8> If there is forwarding, for the first seven cycles during the execution of this code, specify which signals are asserted in each cycle by hazard detection and forwarding units in Figure 4.76.
4.27.5 [10] <§4.8> If there is no forwarding, what new input and output signals do we need for the hazard detection unit in Figure 4.76? Using this instruction sequence as an example, explain why each signal is needed.

3. [4.29] This exercise examines the accuracy of various branch predictors for the following repeating pattern (e.g., in a loop) of branch outcomes: T, NT, T, T, NT.
4.29.1 [5] <§4.9> What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?
4.29.2 [5] <§4.9> What is the accuracy of the 2-bit predictor for the first four branches in this pattern, assuming that the predictor starts off in the bottom left state from Figure 4.78 (predict not taken)?
4.29.3 [10] <§4.9> What is the accuracy of the 2-bit predictor if this pattern is repeated forever?
4.29.4 [30] <§4.9> Design a predictor that would achieve a perfect accuracy if this pattern is repeated forever. You predictor should be a sequential circuit with one output that provides a prediction (1 for taken, 0 for not taken) and no inputs other than the clock and the control signal that indicates that the instruction is a conditional branch.
4.29.5 [10] <§4.9> What is the accuracy of your predictor from 4.29.4 if it is given a repeating pattern that is the exact opposite of this one?
4.29.6 [20] <§4.9> Repeat 4.29.4, but now your predictor should be able to eventually (after a warm-up period during which it can make wrong predictions) start perfectly predicting both this pattern and its opposite. Your predictor should have an input that tells it what the real outcome was. Hint: this input lets your predictor determine which of the two repeating patterns it is given.

4. You are tasked with optimizing a dense **matrix-matrix multiplication (MMM)** kernel for a modern multi-core processor and evaluating its performance using the **Roofline Model**.
Consider a **square matrix multiplication** of size N×N: C=A×B, where A,B,C are **double-precision floating-point** matrices stored in **row-major order**.
Your target hardware has the following characteristics:

- Peak computational performance: **500 GFLOP/s** (assuming double precision, fused multiply-add instructions).
- Memory bandwidth: **50 GB/s** from the main memory.

A. Baseline Implementation & Roofline Model Analysis

- Implement a naïve MMM kernel using **three nested loops** (row-major ordering).

- Calculate the **operational intensity (OI)**:
  OI=Floating Point Operations (FLOPs) / Bytes Transferred from Memory
- Is it memory bound or compute bound?
- Suggest modifications which can push the implementation to be memory bound or compute bound (consider caches, vectorised implementation, and other algorithmic implementations listed on https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm).

5. In one the 5 stage pipeline for RISC-V, the following exception types must be handled:

- **Instruction Address Misalignment** (IF Stage)
- **Illegal Instruction** (ID Stage)
- **Arithmetic Exception (e.g., division by zero)** (EX Stage)
- **Load/Store Address Misalignment** (MEM Stage)

- **Memory Access Fault** (MEM Stage)

The processor should handle them via the **SEPC** and **Scause (exception cause register)**.
A. **Exception Detection and Handling in the Pipeline:** Modify the pipeline to detect exceptions at the appropriate stage. Ensure that multiple exceptions in a single instruction are handled correctly (e.g., a misaligned instruction fetch should take priority over an arithmetic exception).
B. **Implementing Pipeline Flush and Control Transfer** Introduce a **flush mechanism** to clear all instructions **after the faulting instruction** in the pipeline. Modify the control unit to redirect execution to the **SEPC.** Ensure the processor can **resume execution** correctly after the exception handler completes.
C. Handling Multiple Exceptions and Interrupts If a pipeline flush is already triggered for an exception, ensure that other exceptions in later stages do not override it.