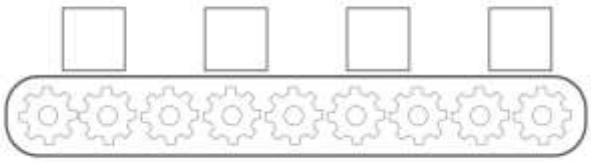
Lab Activity #1:

Basic 5-Stage Pipeline



In the following weeks, we will design a pipelined processor microarchitecture for the OTTER (RV32I instruction set architecture). Your processor will be capable of executing simple C programs that do not use system calls, just as our multi-cycle microarchitecture from 233 but with higher performance.

In this week's lab, we will first build a basic 5-stage pipeline, assuming there are no hazards (control, data, structural). In the following lab, we will then add hazard detection, control, and forwarding logic to our pipelines to efficiently handle data and control hazards that are a regular part of most programs.

After completing our full pipeline, we will characterize the performance (and power/area) of our designs against our baseline OTTER (multi-cycle design from 233). As a friendly competition, we will compare groups to see who designs the processor with the best speedup and lowest energy over our benchmarks from lab0!

Learning Objectives:

- Understand basics of instruction set architecture
- Understand a basic pipelined processor microarchitecture
- Abstraction levels, including register-transfer-level modeling
- Design principles including modularity, hierarchy, and encapsulation
- Design patterns including control/datapath split and pipelined control

Introduction: *Pipelining* is an implementation technique whereby multiple instructions are overlapped in execution. Pipelining takes advantage of parallelism that exists among the actions needed to execute an instruction. Today, pipelining is the key implantation technique used to make fast processors and highly efficient processors that cost less than a dollar.

A pipeline is like an assembly line. In an automobile assembly line, there are many steps, each contributing something to the construction of the car. Each step operates in parallel with the other steps, although on a different car. In a computer assembly line, different steps are completing different parts of the different instructions in parallel. Eahc of these steps is called a *stage*. The stages are connected one to the next to form a pipe – instructions enter at one end, progress through the stages, and exit at the other end, just as cars would in an assembly line.

The *throughput* of an instruction pipeline is determined by how often an instruction exits the pipeline. Because the pipe stages are connected together, all the stages must be ready to proceed at the same time, just as we would require in an assembly line. The time required between moving between moving an instruction one step down the pipeline is a processor cycle. Because all stages proceed at the same time, the length of a processor cycle is determined by the time required for the slowest pipe stage. Ideally the pipeline would have balanced pipeline stages, but usually the stages are never perfectly balanced.

Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream. It has a substantial advantage that, unlike some speedup techniques, it is not visible to the programmer.

Assignment:

1. Implement a 5-stage pipeline implementation of the OTTER. Your pipelined processor should have 5 stages:

    - F- fetch instructions, increment PC;
    - D-decode instructions, read register operands, handle jumps;
    - EX – arithmetic operations, address generation, branch comparison;
    - M – access data memory;
    - W – write register file.

    By design (.e.g dual port memory), our architecture does not have any control hazards. However, data and control hazards are very much a likely occurrence when executing an arbitrary program. That said, for this lab, you can assume that the programs executed on your pipeline do not contain data hazards (e.g. data dependencies –specifically, RAW dependencies - between instructions in the pipeline at any time) or control hazards (e.g. control flow instructions – branches or jumps). Note that these assumptions then mean that your pipeline for this lab can fetch a new instruction every clock cycle. You can find sample test programs for this lab on Canvas. In our next lab we will add architectural support so that the pipeline can more efficiently handle these hazards.

    Tip: generate control signals in the Decode state for all instructions. Use meaningful names for pipeline registers (e.g. ex_pc, m_pc, w_pc, etc.) Feel free to use the struct (instr_t) included in the pipeline template on canvas for holding your decoded signals and connecting them between pipeline stages (i.e. the pipeline registers).

Deliverables:

1. Short paragraph that provides an overview for all of your designs.
2. Answers to all questions, including spreadsheet/graphs showing your measurements.
3. HDL code for all designs