Meeting 6 SP23

Intro to Pipelining

@crawfish

Outline

Section 1

Motivation

Goals of Processor Design

- Low latency
- High throughput
- Small size
- Low energy requirements
- etc.

Latency vs. Throughput

- **Latency** refers to the time needed to completely a single task

Latency vs. Throughput

- **Latency** refers to the time needed to completely a single task
- **Throughput** refers to the average number of tasks we can complete per unit time

Latency vs. Throughput

- **Latency** refers to the time needed to completely a single task
- **Throughput** refers to the average number of tasks we can complete per unit time
    - Most relevant for program run-time

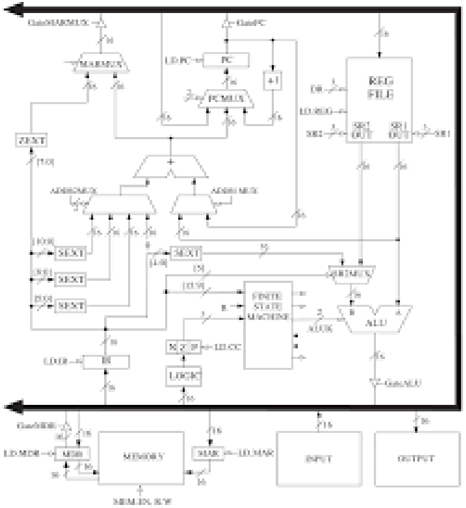
Latency vs. Throughput

- **Latency** refers to the time needed to completely a single task
- **Throughput** refers to the average number of tasks we can complete per unit time
  - Most relevant for program run-time
- Pipelined processor designs optimize for throughput
  - Extra logic required actually causes a slight *increase* in latency

Single Cycle Processors

A **single cycle processor** is one in which an individual instruction takes one clock cycle to execute*.

LC3 Single-Cycle Datapath



Why is this Slow?

- Only one instruction is being processed per clock cycle

Why is this Slow?

- Only one instruction is being processed per clock cycle
- Clock cycles are long

Section 2

Pipelining

Multi-Cycle Processors

A **multi-cycle processor** takes multiple clock cycles to execute a single instruction

Multi-Cycle Processors

A **multi-cycle processor** takes multiple clock cycles to execute a single instruction

- Pipelined processors are a type of multi-cycle processor in which the CPU can have multiple in-progress instructions at once

Multi-Cycle Processors

A **multi-cycle processor** takes multiple clock cycles to execute a single instruction

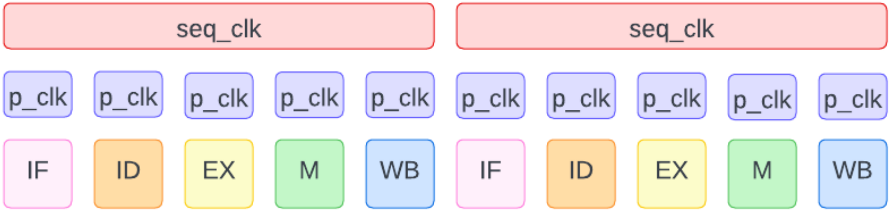
- Pipelined processors are a type of multi-cycle processor in which the CPU can have multiple in-progress instructions at once
- Instruction handling is partitioned into 'units'

Multi-Cycle Processors

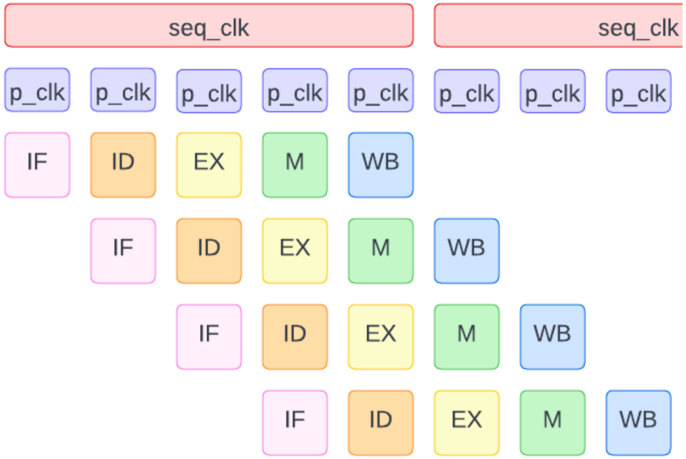
A **multi-cycle processor** takes multiple clock cycles to execute a single instruction

- Pipelined processors are a type of multi-cycle processor in which the CPU can have multiple in-progress instructions at once
- Instruction handling is partitioned into 'units'
- Instruction dispatch is staggered such that each unit is always working on exactly 1 instruction

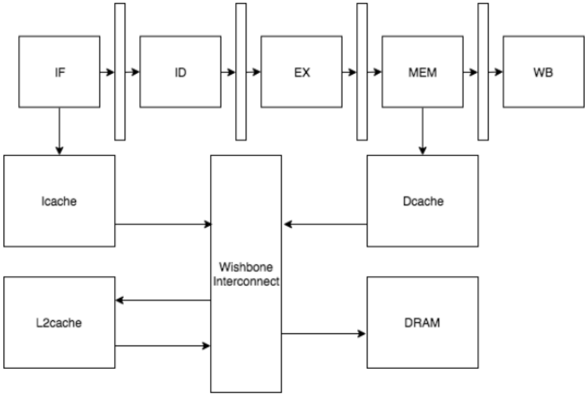
Sequential Execution



Pipelining


seq_clk
seq_clk
p_clk
p_clk
p_clk
p_clk
p_clk
p_clk
p_clk
p_clk
IF
ID
EX
M
WB
IF
ID
EX
M
WB
IF
ID
EX
M
WB
IF
ID
EX
M
WB

LC3 Pipelined Architecture



Instruction Fetch

```
module InstructionFetch(
    input clk,
    input [15:0] pc,
    input stall,
    output [15:0] instr_out,

    // One way read-only memory itf
    mem_itf.read_only_o mem_itf
);
```

Instruction Decode

```
module InstructionDecode(
    input [15:0] instr,

    // From instr 'microcode', produce ctrl signals
    control_signals_itf.control_signals cs_itf
);
```

Execution

```
module ExecutionUnit(
    // reg_a, reg_b are read from register file
    input logic [15:0] reg_a,
    input logic [15:0] reg_b,
    input logic[1:0] opcode,

    // c is forwarded for later writeback
    output logic [15:0] c
);
```

Memory Ops

- Interactions with memory occur here
  - Load / store instructions

Memory Ops

- Interactions with memory occur here
  - Load / store instructions
- Memory operations are not single cycle (even w/ caching)

Memory Ops

- Interactions with memory occur here
  - Load / store instructions
- Memory operations are not single cycle (even w/ caching)
- Less regular access patterns than instruction fetch

Memory Ops

- Interactions with memory occur here
  - Load / store instructions
- Memory operations are not single cycle (even w/ caching)
- Less regular access patterns than instruction fetch
- Memory operations are typically less frequent than arithmetic operations

Memory Ops

- Interactions with memory occur here
  - Load / store instructions
- Memory operations are not single cycle (even w/ caching)
- Less regular access patterns than instruction fetch
- Memory operations are typically less frequent than arithmetic operations
  - Reducing the need to interact with memory is a huge area of optimization

Memory Ops

- Interactions with memory occur here
    - Load / store instructions
- Memory operations are not single cycle (even w/ caching)
- Less regular access patterns than instruction fetch
- Memory operations are typically less frequent than arithmetic operations
    - Reducing the need to interact with memory is a huge area of optimization
- How do we deal w/ long memory operations in our pipeline?

Memory Ops

- Interactions with memory occur here
  - Load / store instructions
- Memory operations are not single cycle (even w/ caching)
- Less regular access patterns than instruction fetch
- Memory operations are typically less frequent than arithmetic operations
  - Reducing the need to interact with memory is a huge area of optimization
- How do we deal w/ long memory operations in our pipeline?
  - Stall :(

Write Back

- Instruction results are 'committed' to register file
- Input comes from either execution stage or register file depending on control signals

What are some problems with this design?

```
; R4 = 0x1, R5 = 0x2
ADD R4, R5, R0
ADD R4, R4, R4
NOP
NOP
NOP ; R4 = 0x6?
```

Hazards

- **RAW:** Read After Write
  - If we 'read' data from the register file (or memory) too soon after it is written, we may encounter a race condition where the returned data is stale
  - Write has not yet been 'committed'

Hazards

- **RAW:** Read After Write
  - If we 'read' data from the register file (or memory) too soon after it is written, we may encounter a race condition where the returned data is stale
  - Write has not yet been 'committed'
- How do we fix this?

Hazards

- **RAW:** Read After Write
  - If we 'read' data from the register file (or memory) too soon after it is written, we may encounter a race condition where the returned data is stale
  - Write has not yet been 'committed'
- How do we fix this?
  - Wait

Hazards

- **RAW:** Read After Write
  - If we 'read' data from the register file (or memory) too soon after it is written, we may encounter a race condition where the returned data is stale
  - Write has not yet been 'committed'
- How do we fix this?
  - Wait
  - Instruction forwarding
- Other hazard types exist in more complicated pipeline structures

Branch and Jump Instructions

- Branch / Jump instructions modify the program counter in an unexpected* way

Branch and Jump Instructions

- Branch / Jump instructions modify the program counter in an unexpected* way
- These changes take effect during the ’write-back’ phase

Branch and Jump Instructions

- Branch / Jump instructions modify the program counter in an unexpected* way
- These changes take effect during the 'write-back' phase
  - Implemented as an addition or replacement of the program counter register

Branch and Jump Instructions

- Branch / Jump instructions modify the program counter in an unexpected* way
- These changes take effect during the ’write-back’ phase
  - Implemented as an addition or replacement of the program counter register
- What about instructions that populate the pipeline in the meantime?
  - Not useful instructions
  - If they modify state (registers, memory) they will cause incorrect future results upon completion

Branch and Jump Instructions

- Stall the pipeline (Wait)

Branch and Jump Instructions

- Stall the pipeline (Wait)
- Branch prediction (Guess)
  - If we guessed wrong, don’t commit anything

Branch and Jump Instructions

- Stall the pipeline (Wait)
- Branch prediction (Guess)
  - If we guessed wrong, don't commit anything
  - Lots of research has been done to optimize branch prediction
  - Code is very repetitive
  - If we guess, we can be right fairly often

Golden Rule of CPU Design: Hide Latency to Increase Throughput

The end. (Now go do some 233 practice problems)