CS250P: Computer Systems Architecture
Pipelining

Sang-Woo Jun
Fall 2022

Large amount of material adapted from MIT 6.004, “Computation Structures”, Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”, and CS 152 Slides by Isaac Scherson
State of our understanding

- Complex logic has high propagation delay
  - Which leads to lower clock speed
- Naturally, we must trade-off complexity of the processor vs. clock speed
  - Is this true?

- Q1. Can we make complex processors run at higher clock speeds
- Q2. Will higher clock speeds actually lead to higher performance
Eight great ideas

- Design for Moore’s Law
- Use abstraction to simplify design
- Make the common case fast
- Performance via parallelism
- Performance via pipelining
- Performance via prediction
- Hierarchy of memories
- Dependability via redundancy

But before we start...
Performance Measures

- Two metrics when designing a system

1. Latency: The delay from when an input enters the system until its associated output is produced
2. Throughput: The rate at which inputs or outputs are processed

- The metric to prioritize depends on the application
  - Embedded system for airbag deployment? **Latency**
  - General-purpose processor? **Throughput**
Performance of Combinational Circuits

- For combinational logic
  - latency = $t_{PD}$
  - throughput = $1/t_{PD}$

Is this an efficient way of using hardware?

Source: MIT 6.004 2019 L12
Pipelined Circuits

- Pipelining by adding registers to hold F and G’s output
  - Now F & G can be working on input $X_{i+1}$ while H is performing computation on $X_i$
  - A 2-stage pipeline!
  - For input X during clock cycle j, corresponding output is emitted during clock j+2.

Source: MIT 6.004 2019 L12
Pipelined Circuits

F and G not doing work! Just holding output data

Latency

<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unpipelined</td>
<td>45</td>
<td>1/45</td>
</tr>
<tr>
<td>2-stage pipelined</td>
<td>50</td>
<td>1/25</td>
</tr>
</tbody>
</table>

(Worse!) (Better!)

Source: MIT 6.004 2019 L12
Pipeline conventions

Definition:
- A well-formed K-Stage Pipeline ("K-pipeline") is an acyclic circuit having exactly K registers on every path from an input to an output.
- A combinational circuit is thus a 0-stage pipeline.

Composition convention:
- Every pipeline stage, hence every K-Stage pipeline, has a register on its output (not on its input).

Clock period:
- The clock must have a period $t_{CLK}$ sufficient to cover the longest register to register propagation delay plus setup time.

K-pipeline latency = $K \times t_{CLK}$

K-pipeline throughput = $1 / t_{CLK}$

Source: MIT 6.004 2019 L12
Ill-formed pipelines

- Is the following circuit a K-stage pipeline? No

- Problem:
  - Some paths have different number of registers
  - Values from different input sets get mixed! -> Incorrect results
    - $B(Y_{t-1}, A(X_t)) \leftarrow$ Mixing values from $t$ and $t-1$
A pipelining methodology

- **Step 1:**
  - Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

- **Step 2:**
  - Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction.
  - These lines demarcate pipeline stages.

- **Step 3:**
  - Add a pipeline register at every point where a separating line crosses a connection.

**Strategy:** Try to break up high-latency elements, make each pipeline stage as low-latency as possible!

Source: MIT 6.004 2019 L12
Pipelining example

- 1-pipeline improves neither L nor T
- T improved by breaking long combinational path, allowing faster clock
- Too many stages cost L, not improving T
- Back-to-back registers are sometimes needed for well-formed pipelines

<table>
<thead>
<tr>
<th></th>
<th>LATENCY</th>
<th>THROUGHPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>1-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>2-pipe:</td>
<td>4</td>
<td>1/2</td>
</tr>
<tr>
<td>3-pipe:</td>
<td>6</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Source: MIT 6.004 2019 L12
Hierarchical pipelining

- Pipelined systems can be hierarchical
  - Replacing a slow combinational component with a k-pipe version may allow faster clock

- In the example:
  - 4-stage pipeline, T=1

Source: MIT 6.004 2019 L12
Sample pipelining problem

Pipeline the following circuit for maximum throughput while minimizing latency.

- Each module is labeled with its latency

What is the best latency and throughput achievable?

Source: MIT 6.004 2019 L12
Sample pipelining problem

- $t_{\text{CLK}} = 4$
- $T = \frac{1}{4}$
- $L = 4 \times 4 = 16$
Aside: When pipelines are not deterministic

- Lock-step pipelines are great when modules are deterministic
  - Good for carefully scheduled circuits like a well-optimized microprocessor

- What if the latency of F is non-deterministic?
  - At some cycles, F’s pipeline register may hold invalid value
  - Pipeline register must be tagged with a valid flag
  - How many pipeline registers should we add to G? Max possible latency?
  - What if F and G are both non-deterministic? How many registers?
Aside: FIFOs (First-In First-Out)

- Queues in hardware
  - Static size (because it’s hardware)
  - User checks whether full or empty before enqueue or dequeue
  - Enqueue/dequeue in single cycle regardless of size or occupancy

- MUX! Large FIFO has long propagation delay
Counting cycles: Benefits of an elastic pipeline

- Assume F and G are multi-cycle, internally pipelined modules
  - If we don’t know how many pipeline stages F or G has, how do we ensure correct results?

- Elastic pipeline allows correct results regardless of latency
  - If \( L(F) = L(G) \), enqueued data available at very next cycle (acts like single register)
  - If \( L(F) = L(G) + 1 \), FIFO acts like two pipelined registers
  - What if we made a 4-element FIFO, but \( L(F) = L(G) + 4 \)?
    - G will block! Results will still be correct!
    - ... Just slower! How slow?

```
L <- Latency in cycles
```

Measuring pipeline performance

- Latency of F is 3, Latency of G is 1, and we have a 2-element FIFO
  - What would be the performance of this pipeline?

- One pipeline “bubble” every four cycles
  - Duty cycle of \( \frac{3}{4} \)!
Aside: Little’s law

- \( L = \lambda W \)
  - \( L \): Number of requests in the system
  - \( \lambda \): Throughput
  - \( W \): Latency

Imagine a DMV office! \( L \): Number of booths. (Not number of chairs in the room)

- In our pipeline example
  - \( L = 3 \) (limited by pipeline depth of \( G \))
  - \( W = 4 \) (limited by pipeline depth of \( F \))
  - As a result: \( \lambda = \frac{3}{4}! \)

How do we improve performance?
- Larger FIFO, or
- Replicate \( G \)! (round-robin use of \( G1 \) and \( G2 \))
CS250P: Computer Systems Architecture Processor Microarchitecture – Pipelining

Sang-Woo Jun
Fall 2022

Large amount of material adapted from MIT 6.004, “Computation Structures”, Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”, and CS 152 Slides by Isaac Scherson
Course outline

- Part 1: The Hardware-Software Interface
  - What makes a ‘good’ processor?
  - Assembly programming and conventions

- Part 2: Recap of digital design
  - Combinational and sequential circuits
  - How their restrictions influence processor design

- Part 3: Computer Architecture
  - Simple and pipelined processors
  - Computer Arithmetic
  - Caches and the memory hierarchy

- Part 4: Computer Systems
  - Operating systems, Virtual memory
How to build a computing machine?

- Pretend the computers we know and love have never existed
- We want to build an automatic computing machine to solve mathematical problems
- Starting from (almost) scratch, where you have transistors and integrated circuits but no existing microarchitecture
  - No PC, no register files, no ALU
- How would you do it? Would it look similar to what we have now?
Aside: Dataflow architecture

- Instead of traversing over instructions to execute, all instructions are independent, and are each executed whenever operands are ready.
  - Programs are represented as graphs (with dependency information).

Did not achieve market success, (why?) but the ideas are now everywhere e.g., Out-of-Order microarchitecture.

Figure 2. A dataflow graph representation of $\text{sum} = \sum_{i=1}^{N} f(i)$. 

A “static” dataflow architecture
The von Neumann Model

- Almost all modern computers are based on the von Neumann model
  - John von Neumann, 1945

- Components
  - Main memory, where both data and programs are held
  - Processing unit, which has a program counter and ALU
  - Storage and I/O to communicate with the outside world
Key Idea: Stored-Program Computer

- Very early computers were programmed by manually adjusting switches and knobs of the individual programming elements
  - (e.g., ENIAC, 1945)
- von Neumann Machines instead had a general-purpose CPU, which loaded its instructions also from memory
  - Express a program as a sequence of coded instructions, which the CPU fetches, interprets, and executes
    - “Treating programs as data”

Similar in concept to a universal Turing machine (1936)
Example: Harvard Mark 1

- Built 1944 (near the end of WW2) using switches, relays, shafts, etc
  - Used to crunch numbers for Manhattan project
  - Programmed by John von Neumann and others
Example: Harvard Mark 1

- Slow by today standards!
  - 3 Additions/s, 6 secs for mults, etc

![Photo: ArnoldReinhold, Wikimedia commons](https://upload.wikimedia.org/wikipedia/commons/thumb/2/26/Harvard_Mark_1.jpg/1280px-Harvard_Mark_1.jpg)

Data also entered via switches

Programs/data entered through tape, no control flow instructions!
(Loops meant physically gluing tape into loops)
Another example: MITS Altair (1978)

- Built using Intel 8080 @ 2 MHz
- Only input are front panel switches
- Only output are front panel LEDs
- First successful personal computer
- Bill Gates sold his first software
  - Altair BASIC
  - Tape reader expansion

Actually a bad example... Programs were entered via switches/tape but 8080 had control flow instructions!)
von Neumann and Turing machine

Turing machine is a mathematical model of computing machines
  - Proven to be able to compute any mechanically computable functions
  - Anything an algorithm can compute, it can compute

Components include
  - An infinite tape (like memory) and a header which can read/write a location
  - A state transition diagram (like program) and a current location (like pc)
    - State transition done according to current value in tape

Only natural that computer designs gravitate towards provably universal models
Once we decide on the stored program computer paradigm
  o With program counter (PC) pointing to encoded programs in memory
Then it becomes an issue of deciding the programming abstraction
  o Instruction set architecture, which we talked about
Then, it becomes an issue of executing it quickly and efficiently
  o Microarchitecture! – Improving performance/efficiency/etc while maintaining ISA abstraction
  o Which is the core of this class, starting now
The classic RISC pipeline

- Many early RISC processors had very similar structure
  - MIPS, SPARC, etc...
  - Major criticism of MIPS is that it is too optimized for this 5-stage pipeline
- RISC-V is also typically taught using this structure as well
Remember:
Super simplified processor operation

\[ \text{inst} = \text{mem}[\text{PC}] \]
\[ \text{next\_PC} = \text{PC} + 4 \]

if ( inst.type == STORE ) \[ \text{mem}[\text{rf}[\text{inst.arg1}]] = \text{rf}[\text{inst.arg2}] \]
if ( inst.type == LOAD ) \[ \text{rf}[\text{inst.arg1}] = \text{mem}[\text{rf}[\text{inst.arg2}]] \]
if ( inst.type == ALU ) \[ \text{rf}[\text{inst.arg1}] = \text{alu}(\text{inst.op}, \text{rf}[\text{inst.arg2}], \text{rf}[\text{inst.arg3}]) \]
if ( inst.type == COND ) \[ \text{next\_PC} = \text{rf}[\text{inst.arg1}] \]

\[ \text{PC} = \text{next\_PC} \]
The classic RISC pipeline

- Fetch: Request instruction fetch from memory
- Decode: Instruction decode & register read
- Execute: Execute operation or calculate address
- Memory: Request memory read or write
- Writeback: Write result (either from execute or memory) back to register

Why these 5 stages? Why not 1 or 6?
A high-level view of computer architecture

- CPU
- Instruction cache
- Data cache
- Shared cache
- DRAM

Low latency (~1 cycle)
High latency (100s~1000s of cycles)

Will deal with caches in detail later!
Designing a microprocessor

Many, many constraints processors are optimize for, but for now:

- Constraint 1: Circuit timing
  - Processors are complex! How do we organize the pipeline to process instructions as fast as possible?

- Constraint 2: Memory access latency
  - Register files can be accessed as a combinational circuit, but it is small
  - All other memory have high latency, and must be accessed in separate request/response
    - Memory can have high throughput, but also high latency

Memory will be covered in detail later!
The most basic microarchitecture

- Because memory is not combinational, our RISC ISA requires at least three disjoint stages to handle
  - Instruction fetch
  - Instruction receive, decode, execute (ALU), register file access, memory request
  - If mem read, write read to register file

- Three stages can be implemented as a Finite State Machine (FSM)

Will this processor be fast? Why or why not?
Limitations of our simple microarchitecture

- Stage two is disproportionately long
  - Very long critical path, which limits the clock speed of the whole processor
  - Stages are “not balanced”

- Note: we have not pipelined things yet!
Limitations of our simple microarchitecture

- Let’s call our stages Fetch(“F”), Execute(“E”), and Writeback (“W”)
- Speed of our simple microarchitecture, assuming:
  - Clock-synchronous circuits, single-cycle memory
- Lots of time not spent doing useful work!
  - Can pipelining help with performance?

![Diagram of pipeline stages](image)
Pipelined processor introduction

- Attempt to pipeline our processor using pipeline registers/FIFOs

- Much better latency and throughput!
  - Average CPI reduced from 3 to 1!
  - Still lots of time spent not doing work. Can we do better?

* We will see soon why pipelining a processor isn’t this simple

Note we need a memory interface with two concurrent interfaces now! (For fetch and execute) Remember instruction and data caches!
Building a balanced pipeline

- Must reduce the critical path of Execute
- Writing ALU results to register file can be moved to “Writeback”
  - Most circuitry already exists in writeback stage
  - No instruction uses memory load and ALU at the same time
    - RISC!
Building a balanced pipeline

- Divide execute into multiple stages
  - “Decode”
    - Extract bit-encoded values from instruction word
    - Read register file
  - “Execute”
    - Perform ALU operations
  - “Memory”
    - Request memory read/write

- No single critical path which reads and writes to register file in one cycle

Results in a small number of stages with relatively good balance!
Ideally balanced pipeline performance

- Clock cycle: 1/5 of total latency
- Circuits in all stages are always busy with useful work
Aside: Real-world processors have wide range of pipeline stages

<table>
<thead>
<tr>
<th>Name</th>
<th>Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVR/PIC microcontrollers</td>
<td>2</td>
</tr>
<tr>
<td>ARM Cortex-M0</td>
<td>3</td>
</tr>
<tr>
<td>Apple A9 (Based on ARMv8)</td>
<td>16</td>
</tr>
<tr>
<td>Original Intel Pentium</td>
<td>5</td>
</tr>
<tr>
<td>Intel Pentium 4</td>
<td>30+</td>
</tr>
<tr>
<td>Intel Core (i3,i5,i7,...)</td>
<td>14+</td>
</tr>
<tr>
<td>RISC-V Rocket</td>
<td>6</td>
</tr>
</tbody>
</table>

Designs change based on requirements!
Will our pipeline operate correctly?
A problematic example

- What should be stored in data+8? 3, right?

```
la t0 data
lw s0, 0(t0)
lw s1, 4(t0)
add s2, s0, s1
sw s2, 8(t0)
data:
> .word 1 2
```

- Assuming zero-initialized register file, our pipeline will write zero
  Why? “Hazards”