Pipelined Implementation
Part 1

CMPU 224 – Computer Organization
Jason Waterman
Real-World Pipelines: Car Washes

Sequential

Pipelined

• Idea
  • Divide process into independent stages
    • Soap, rinse, wax, buff, dry
  • Move objects through stages in sequence
  • At any given times, multiple objects being processed

• Throughput
  • Number of customers served per unit time

• Latency
  • The time required to service an individual customer
Computational Example

- System
  - Computation requires total of 300 picoseconds
  - Additional 20 picoseconds to save result in register
  - Must have clock cycle of at least 320 ps

Delay = 320 ps
Throughput = 3.125 GIPS (1/320 ps)
3-Way Pipelined Version

• System
  • Divide combinational logic into 3 blocks of 100 ps each
  • Can begin new operation as soon as previous one passes through stage A
    • Begin new operation every 120 ps
    • Increase in throughput: $8.333 / 3.125 = 2.667$ times
  • Overall latency increases
    • 360 ps from start to finish
    • Increase in latency: $360 / 320 = 1.12$ times

Delay = 360 ps
Throughput = 8.333 GIPS (1/120 ps)
Pipeline Diagrams

- **Sequential**
  - Cannot start new operation until previous one completes

- **3-Way Pipelined**
  - Up to 3 operations in process simultaneously
Operating a Pipeline

Clock

OP1
A B C

OP2
A B C

OP3
A B C

Time
Operating a Pipeline

Clock

OP1
A B C

OP2
A B C

OP3
A B C

Time

0 120 240 360 480 640

100 ps
20 ps
100 ps
20 ps
100 ps
20 ps

Comb. logic A
Reg

Comb. logic B
Reg

Comb. logic C
Reg

Clock
### Operating a Pipeline

A pipeline in computer architecture is a method of processing tasks in a sequential manner where the output of one task becomes the input of the next. This allows for the efficient execution of multiple tasks simultaneously, reducing the overall processing time. The diagram illustrates how tasks (OP1, OP2, OP3) are executed in a pipeline over time.

<table>
<thead>
<tr>
<th>Clock</th>
<th>OP1</th>
<th>OP2</th>
<th>OP3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>120</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>240</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>360</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>480</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>640</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Comb. logic**: Combinational logic performs operations on inputs to produce outputs.
- **Reg**: Register stores the intermediate results.

The diagram shows that the time for each operation is as follows:
- OP1: 100 ps
- OP2: 20 ps
- OP3: 100 ps

The total time for processing all operations is the sum of these times.

---

**Clock**

- 241

**Comb. logic**

- A
- B
- C

**Reg**

- A
- B
- C
Operating a Pipeline

Clock

OP1

OP2

OP3

Time

0 120 240 360 480 640

100 ps 20 ps 100 ps 20 ps 100 ps 20 ps

Comb. logic A

Reg

Comb. logic B

Reg

Comb. logic C

Reg

Clock
Operating a Pipeline

Clock

<table>
<thead>
<tr>
<th>Time</th>
<th>OP1</th>
<th>OP2</th>
<th>OP3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>120</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>240</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>360</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>480</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>640</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

Comb. logic A

100 ps

Reg

Comb. logic B

100 ps

Reg

Comb. logic C

100 ps

Reg

20 ps

Clock
Limitations: Nonuniform Delays

- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Delay = 510 ps (450 + 60)
Throughput = 5.88 GIPS (1/170 ps)
Limitations: Register Overhead

- As we deepen the pipeline, overhead of loading registers becomes more significant.
- Percentage of clock cycle spent loading register for our example:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining (15 or more stages)
Data Dependencies

- System
  - Each operation depends on result from preceding one
Data Hazards

- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system
Data Dependencies in Processors

- Result from one instruction used as operand for another
  - Read-after-write (RAW) dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

```
1  irmovq $50, %rax
2  addq %rax, %rbx
3  mrmovq 100(%rbx), %rdx
```
SEQ Hardware

- Stages occur in sequence
- One operation in process at a time
SEQ+ Hardware

- Still sequential implementation
  - Reorder PC stage to put at beginning

- PC Stage
  - Task is to select PC for current instruction
  - Based on results computed by previous instruction

- Processor State
  - PC is no longer stored in register
  - Can determine PC based on other stored information
Pipeline Stages

- Fetch
  - Select current PC
  - Read instruction
  - Compute incremented PC
- Decode
  - Read program registers
- Execute
  - Operate ALU
- Memory
  - Read or write data memory
- Write Back
  - Update register file
Pipeline Demonstration

irmovq $1,%rax  #I1
irmovq $2,%rcx  #I2
irmovq $3,%rdx  #I3
irmovq $4,%rbx  #I4
halt  #I5

Cycle 5
PIPE- Hardware

- Pipeline registers hold intermediate values from instruction execution
- Forward (Upward) Paths
  - Values passed from one stage to next
  - Cannot jump past stages
    - e.g., valC passes through decode
Signal Naming Conventions

- **S_Field**
  - Value of Field held in stage S pipeline register
  - E.g., M_stat, M_Cnd

- **s_Field**
  - Value of Field computed in stage S
  - E.g., m_stat, e_Cnd
Feedback Paths

• Predicted PC
  • Guess value of next PC

• Branch information
  • Jump taken/not-taken
  • Fall-through or target address

• Return point
  • Read from memory

• Register updates
  • Register file write ports
Predicting the PC

• Start fetch of new instruction after current one has completed fetch stage
  • Not enough time to reliably determine next instruction (e.g., Jxx instructions)

• Guess which instruction will follow
  • Recover if prediction was incorrect
Our Prediction Strategy

- Instructions that do not transfer control
  - Predict next PC to be valP
  - Always reliable

- Call and unconditional jumps (jmp)
  - Predict next PC to be valC (destination)
  - Always reliable

- Conditional jumps
  - Predict next PC to be valC (destination)
  - Only correct if branch is taken
    - Typically, right 60% of time

- Return instruction
  - Do not try to predict
Branches and Returns

• Mispredicted Branches
  • Will see branch condition flag once instruction is in memory stage
  • Can get fall-through PC from \( \text{valA} \) (value \( \text{M_valA} \))

• Return Instruction
  • Will get return PC when \( \text{ret} \) reaches write-back stage (\( \text{W_valM} \))

• More on this later when we talk about Control Hazards
Feedback Paths

• Predicted PC
  • Guess value of next PC

• Branch information
  • Jump taken/not-taken
  • Fall-through or target address

• Return point
  • Read from memory

• Register updates
  • Register file write ports
Data Dependencies

0x000: `irmovq $10, %rdx`
0x00a: `irmovq $3, %rax`
0x014: `addq %rdx, %rax`
0x016: `halt`

Cycle 4

M

\[ M_{\text{valE}} = 10 \]
\[ M_{\text{dstE}} = %rdx \]

E

\[ e_{\text{valE}} \leftarrow 0 + 3 = 3 \]
\[ E_{\text{dstE}} = %rax \]

D

\[ \text{valA} \leftarrow R[\%rdx] = 0 \]
\[ \text{valB} \leftarrow R[\%rax] = 0 \]
Data Dependencies: 1 Nop

0x000: `irmovq $10, %rdx`
0x00a: `irmovq $3, %rax`
0x014: `nop`
0x015: `addq %rdx, %rax`
0x017: `halt`

Cycle 5

\[ W \]
\[ R[\%rdx] \leftarrow 10 \]
\[ M \]
\[ M_{\text{valE}} = 3 \]
\[ M_{\text{dstE}} = \%rax \]
\[ D \]
\[ \text{valA} \leftarrow R[\%rdx] = 0 \]
\[ \text{valB} \leftarrow R[\%rax] = 0 \]
Data Dependencies: 2 Nop's

0x000: ` irmovq $10, %rdx`
0x00a: ` irmovq $3, %rax`
0x014: `nop`
0x015: `nop`
0x016: `addq %rdx, %rax`
0x018: `halt`

Cycle 6

W

R[ %rax] \(\leftarrow 3\)

D

valA \(\leftarrow R[ %rdx] = 10\)

valB \(\leftarrow R[ %rax] = 0\)

Error
Data Dependencies: 3 Nop’s

0x000: irmovq $10, %rdx
0x00a: irmovq $3, %rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq %rdx, %rax
0x019: halt

Cycle 6

W
R[%rax] ← 3

Cycle 7

D
valA ← R[%rdx] = 10
valB ← R[%rax] = 3

4/11/2023
Data Dependencies: NOPs

• The problem is a RAW (Read After Write) dependency
• Read values from the register in the decode stage
• Registers are not updated until the write-back stage
  • 3 cycles later
• One solution
  • Make sure registers that are written are not read until after 3 cycles has passed
  • Can insert NOPs between the instructions to insure correct behavior
  • This can be very error-prone
• Have the processor detect and correct this situation
  • We will do this next lecture
Pipelined Processor Summary

- Concept
  - Break instruction execution into 5 stages
  - Run instructions through in pipelined mode

- Limitations
  - Data dependencies
    - One instruction writes register, later one reads it
  - Control dependency
    - Mispredicted branch and return
    - Instruction sets PC in way that pipeline did not predict correctly

- Fixing the pipeline
  - We’ll do that next time