Quick Review: Multi Cycle + Pipelining

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE

CAMEL
Recall: Single Cycle Processor

Fetch

Instruction Memory
- Read Address
- Instruction

Read Addr 1
Read Addr 2
Write Addr
Write Data

Register File
- RegWrite
- ALUSrc
- ALUCtrl
- MemWrite
- MemtoReg

Data Memory
- Address Read Data
- Write Data

PC

Adder (ADD)
- 4

Fetch
- Decode
- Execute

Instruction

Memory Address

Reading Data 1

Reading Data 2

ALU
- ovf
- zero

ALUSrc

ALUCtrl

MemRead

Sign Extend
- 16
- 32

Execute

CAMELab

KAIST
Challenges of Single Cycle

- We can get a CPI of 1 (for most instructions)
- But clock cycle time is long
  - It must be long enough to complete the most time-consuming instruction

One cycle is determined to accommodate a long critical path (laggard)
Solution: Multicycle Datapath Approach

- **Single cycle implementation**

  - Clock
  - Load
  - Store (Waste)

- **Multi-cycle implementation**

  - Clock
  - Load
  - Store
  - ADD

This multicycle clock machine slower than $\frac{1}{5}$th of the single cycle one due to implementation issues (will discussed) But, it eventually removes the wasted time
Design Principles of Multicycle Datapath

Q1. How to let an instruction take more than 1 clock cycle to complete?
A. Break up an instruction into small steps

Q2. How to efficiently manage the datapath components? (Removes unnecessary resource duplication)
A. Allow functional units can be used more than once per instruction (but in a different cycle)

Q3. For the single cycle, control signals could be determined from opcode. How about multicycle?
A. Use a finite state machine (FSM) for control
Break up An Instruction into Multi-Steps

- Balance the amount of work to be done in each step
- Restrict each cycle to enable only one major functional unit

- **Ifetch**: Instruction Fetch and Update PC
- **Dec**: Instruction Decode, Register Read, Sign Extend Offset
- **Exec**: Execute R-type; Calculate Memory Address; Branch Comparison; Branch and Jump Completion
- **Mem**: Memory Read; Memory Write Completion; R-type Completion (RegFile write)
- **WB**: Memory Read Completion (RegFile write)
Design/Support for Multi-Step Instructions

[REMIND] Clock methodologies

1. Read contents of state elements
2. Send values through combinational logic
3. Write results to one or more state elements

State Element 1 → Combinational logic → State Element 2

At the end of a cycle, let’s store value(s), which will be needed in a next cycle into an internal register.
Design/Support for Multi-Step Instructions

- Functional units can be used on different clock cycles
  - Only need one memory – but only one memory access/cycle
  - Need only one ALU/adder – but only one ALU operation/cycle
- Additional internal state registers
  - IR (Instruction register), MDR (Memory Data Register)
  - A, B (regfile read data registers), ALUout (ALU output register)
Design/Support for Multi-Step Instructions

- Functional units can be used on different clock cycles
- Only need one memory – but only one memory access/cycle
- Need only one ALU/adder – but only one ALU operation/cycle
- Additional internal state registers
  - IR (Instruction register), MDR (Memory Data Register)
  - A, B (regfile read data registers), ALUout (ALU output register)
Finite State Machine for Control

- Opcode bits tell you on what operation the ALU should be doing, but *not* what instruction cycle is to be done in next

Finite state machine (FSM) for control

- A set of states (the current state stored in State Register)
- Next state function (determined by the current state and input)
- Output function (determined by the current state and input)
Finite State Machine for Control

- Opcode bits tell you on what operation the ALU should be doing, but not what instruction cycle is to be done next
- Finite state machine (FSM)
  - A set of states (current state stored in State Register)
  - Next state function (determined by current state and the input)
  - Output function (determined by current state and the input)

Example:

![Finite State Machine Diagram]

- Memory address computation
- (Op = LW) or (Op = SW)
- Memory access
- R-type completion
- Write-back step
- RegDst = 0
- RegWrite = 1
- MemtoReg = 0
- MemRead
- IorD = 1
- MemWrite
- IorD = 1
- ALUSrcA = 1
- ALUSrcB = 11
- ALUOp = 00
- PCWrite
- PCSource = 01
- Start
- Instruction fetch
- Instruction decode/register fetch
- ALUSrcA = 0
- ALUSrcB = 00
- ALUOp = 00
- PCWrite
- PCSource = 00
- Jump completion
- (Op = J)
- (Op = BEQ)
- Branch completion
- ALUSrcA = 1
- ALUSrcB = 00
- ALUOp = 01
- PCWriteCond
- PCSource = 01
- Execution
- ALUSrcA = 1
- ALUSrcB = 01
- ALUOp = 10
- Memory access
The Multicycle Datapath with Control Signals
**Multicycle Advantages & Disadvantages**

**+** Uses the clock cycle efficiently – each clock cycle considers the instruction “step” (not the longest instruction like what single cycle needs to do)

**+** Multicycle allows to use functional units more than once per instruction (across different cycles)

**-** But, it requires additional internal state registers, more muxes, and complicated (FSM) control
Pipelining

Still multicycle is slow..
Need for SPEED!
(Throughput viewpoint, not latency)
Oil Transport Analogy for Strategy

• Plan A: Move 1,000,000 gallons of oil at a time
  Buy a tanker ship, then repeat:
  Fill with oil, sail for 30 days, empty, go back

• Plan B: Move 100 gallons of oil at a time
  Buy a speedboat, then repeat:
  Take barrel of oil, sail for 2 days, unload, go back

• Plan C:

  Pipeline
Student Cafeteria, KAIMARU Analogy

- **Multi cycle processor**

- **Pipeline processor**
Design Principles of Pipeline Datapath

We want to overlap multiple instructions. What should be modified from multicycle Datapath?

Q1. In contrast to multicycle, 5 stages (IF, ID, EX, MEM, WB) require enabling some Datapath components at the same time

A. Like single cycle Datapath, duplicates several Datapath components again

Q2. How to execute multiple instructions at the same time? (Program counter is updated on every cycle)

A. Use additional intermediate registers to record the PC and Instruction

Q3. Similar to Q2, the functionality of control signals, which are required for each instruction, varies. How can we handle the signals?

A. Let’s propagate the control signals through the pipeline by passing them via the pipeline registers, along with the other data
Design 1: Duplicate Datapath Components

- Equivalent to the original single-cycle Datapath
- Separate memories for instructions and data
- Two adders for PC-based computations + one ALU
Design 1: Duplicate Datapath Components

- But, one register file is enough to support both the ID and WB
- Reads and writes go to separate *ports* on the register file
- Writes occur in the first half of the cycle, reads occur in the second half
Design 2: Pipeline Registers

- Any data values required in later stages must be propagated through the pipeline registers.
- Simplify diagrams consider the registers as just one big pipeline register between each stage (IF/ID, ID/EX, EX/MEM, MEM/WB).

### Diagram Description

**Instruction Memory Read:**
- Address

**Data Memory Read:**
- Address

**Instruction:**
- IR: Instruction Register
- PC: Program Counter

**Register:**
- A: RegData1
- B: RegData2
- Out: ALU output
- D: RegData2 (will be memory address)

**ALU:**
- Zero Result

**Mem:**
- Data Memory
- Address

**Write:**
- Data

**Pipe Stages:**
- IF/ID
- ID/EX
- EX/MEM
- MEM/WB
Aside: Example of Value Forwarding

- Destination register
  - The rd field of the instruction, retrieved in the IF, determines the destination register. However, such the register value isn’t propagated from the IF stage to the WB stage.
  - Thus, the rd field must be passed through all of the pipeline stages.
The control signals are generated in the same way as in the single-cycle processor. But, note that some of the control signals won’t be needed for some next-level stages and clock cycles.
An Example Execution Sequence

- Sample sequence of instructions to execute

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>lw $8, 4($29)</td>
</tr>
<tr>
<td>1004</td>
<td>sub $2, $4, $5</td>
</tr>
<tr>
<td>1008</td>
<td>and $9, $10, $11</td>
</tr>
<tr>
<td>1012</td>
<td>or $16, $17, $18</td>
</tr>
<tr>
<td>1016</td>
<td>add $13, $14, $0</td>
</tr>
</tbody>
</table>

- Pipeline terminology
  - The **pipeline depth**: number of stages – in this case, five
  - In the first 4 cycles, the pipeline is **filling**, since there are unused functional units
  - In cycle 5, the pipeline is **full**. All HW units are in use.
  - In cycle 6-9, the pipeline is **emptying**; if there are more instructions, **such pipeline can bring out an output at every single cycle**
Cycle 1 (Filling the Pipeline)

IF/ID
- Read address [31-0]
- Instruction memory

ID/EX
- IF
- EX/MEM
- MEM/WB
- Control
- Registers
- ALU
- ALUSrc
- ALUOp
- Zero
- Result
- Sign
- Extend
- RegDst
- RegWrite
- Shift left2
- ADD
- MemRead
- MemWrite
- MemtoReg

Instruction
Data
Read
Addr1
Data1
Read
Addr2
Read
Write
Addr
Write
Data
Instr [15 - 0]
Instr [20 - 16]
Instr [15 - 11]
Cycle 2 (Filling the Pipeline)

**Instruction Memory:**
- **Read Address:** [31-0]
- **Instruction Memory:**

**Control:**
- **RegWrite:**

**IF/ID:**
- **PC Src:**

**IF:**
- **Add:**

**ID/EX:**
- **Instr [15 - 0]:**
- **Instr [20 - 16]:**
- **Instr [15 - 11]:**
- **Shift Left2:**
- **ALUSrc:**
- **ALU Op:**
- **Zero Result:**

**EX/MEM:**
- **Write Address:**
- **Write Data:**
- **Mem Read:**
- **Mem Write:**
- **MemtoReg:**

**MEM/WB:**
- **Write:**
- **MemtoReg:**

**Registers:**
- **Read Addr1:**
- **Read Addr2:**
- **Write Addr:**
- **Write Data:**

**W/B:**
- **ALU Zero:**
- **Result:**
- **Reg Dst:**
- **Reg Dst:**
- **Reg Src:**

**Instruction:**
- **10:**

**PC:**
- **Add:**

**Registers:**
- **Read Addr1:**
- **Read Addr2:**
- **Write Addr:**
- **Write Data:**

**Control:**
- **RegWrite:**
Cycle 3 (Filling the Pipeline)

Instr2

IF/ID

Control

ID/EX

Instr [15 - 0]

RegWrite

RD

ALU Zero

Inst [20 - 16]

Instr [15 - 11]

Add

Shift left2

EX/MEM

PC

Instr [31 - 0]

Read

Instr memory

Add

Control

EX

RegDst

ALUSrc

MEM/WB

RegWrite

ALUOp

PCSrc

MemRead

Address

Write

Data

Read

Addr1

Data1

Read

Addr2

Data2

Write

Write

Addr

Read

Data

Write

Data

Write

MemtoReg

MEM

Write

Data

Read

data

Read

data

MemWrite

WB

Write

Addr

Read

Address

MemRead

Read

data

PC
Cycle 4 (Filling the Pipeline)

**Instr3**
- Read Inst. address [31-0]
- Instruction memory
- PCSrc
- ADD
- IF/ID
- Control
- RegWrite
- Read Addr1
- Read Addr2
- Write Addr
- Write Data
- Registers
- Instr [15-0]
- Sign Extend

**Instr2**
- ID/EX
- ALUSrc
- ALUOp
- ALU Zero
- Result
- ALU
- Zero
- Address
- Data
- memory
- Write
- Read data
- data
- MemWrite
- MemRead
- MemtoReg
- 0
- 1
- I
- D
- E
- X
- M
- W
- B

**Instr1**
- EX/MEM
- MemWrite
- MemRead
- MemtoReg
- 0
- 1
- I
- D
- E
- X
- M
- W
- B

Pipe stages:
- IF
- ID
- EX
- MEM
- WB

Instructions:
- Instr3
- Instr2
- Instr1
Cycle 5 (Full)

Instr4

PCSrc

1

0

4

IF/ID

EX/MEM

MEM/WB

Control

Instr [15 - 0]

Instr [20 - 16]

Instr [15 - 11]

RegWrite

Shift left2

ALU Zero

Result

ALUOp

MemWrite

MemtoReg

Data memory

Write data

Read data

Adresse

Data1

Data2

Instr [31-0]

Instruction memory

Read Inst. address [31-0]

Registers

Write Addr

Write Data

Read Addr

Read Data1

Read Data2

ALUSrc

RegDst

MemRead

PC

ADD

Registers

Instr [0]
The Pipelining Paradox

- Pipelining does not improve the any single instruction latency
- Each instruction actually takes longer to execute than in a single-cycle Datapath!

[REMIND] Meaning of Single Cycle in Lecture 3

T_s: Setup time that inputs to a register
T_h: Hold time that inputs to a register
Latch at the end of each stage adds latency!
Is it Faster?

- Instruction takes 5 cycles now!
- Instruction latency longer now
  - Latch at end of each stage adds latency
  - Longest stage determines clock cycle time
- Example:

<table>
<thead>
<tr>
<th></th>
<th>Design</th>
<th>Cycle time</th>
<th># of cycles</th>
<th>Inst Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>1-stage</td>
<td>4.1 ns (sum)</td>
<td>1</td>
<td>4.1 ns</td>
</tr>
<tr>
<td>ID</td>
<td>5-stage</td>
<td>1.2 ns (max)</td>
<td>5</td>
<td>6.0 ns</td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- But we are after instruction throughput, not latency!!!
Performance with Pipelining

- We finish one instruction per cycle
  - After the initial warm-up period

- Instruction throughput
  - Latch at end of each stage adds latency
  - Longest stage determines clock cycle time
  - Example:

<table>
<thead>
<tr>
<th>Design</th>
<th>Cycle time</th>
<th>Inst/Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-cycle</td>
<td>4.1 ns (sum)</td>
<td>1</td>
</tr>
<tr>
<td>Pipeline</td>
<td>1.2 ns (max)</td>
<td>1</td>
</tr>
</tbody>
</table>

  - Speedup due to pipelining: 3.42
  - Note: ideally it would be 5
**Ideal Speedup**

- Instead, pipelining increases the **throughput**; again, every cycle, it can deliver the output.
- As we can execute up to five instructions simultaneously, the **ideal speedup** equals the **pipeline depth**.

Q) Why does the pipelining speedup “less than” 5?  
A) The pipeline stages are **imbalanced**.
Let’s increase processor throughput by increasing pipeline depth

Pipeline depths and frequency at introduction

What happen here?

Source: Pipeline Architecture since 1985
Limits to Pipelining – Power Issue

- Too many things happening at once
  - Melt your chip!
- Must disable parts of the system that are not being used
  - Clock Gating, Asynchronous Design, Low Voltage Swings,...
Limits to Pipelining – Hazard Issue

- **Hazards** prevent next instruction from executing during its designated clock cycle
- 3 types of hazard: Structural hazards, Data hazards, Control hazards

Will be covered in next lecture (lecture 5)
Quick Review:
Multi Cycle + Pipelining

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory