Chapter 4

The Processor

Contents

- 4.1 Introduction
- 4.2 Logic Design Conventions
- 4.3 Building a Datapath
- 4.4 A Simple Implementation Scheme
- 4.5 An Overview of Pipelining
- 4.6 Pipelined Datapath and Control
4.5 An Overview of Pipelining

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution.
- All stages in pipelining are operating concurrently.

Laundry Analogy for Pipelining (1/2)

1. Washer
2. Dryer
3. Fold
4. Put away
Laundry Analogy for Pipelining (2/2)

- Four stages in pipeline (Figure 4.25)
  - non-pipelining approach: 16 units in execution time
  - pipelining approach: 7 units
- Pipelining improves throughput of our laundry system without improving the time to complete a single task
- The speedup due to pipelining is likely equal to the number of stages in the pipeline

MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access (data) memory operand
  5. WB: Write result back to register

Fig 27 (a) Single-cycle, nonpipelined
Chapter 4 — The Processor

1-Cycle vs Pipelined Performance

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

Fig. 4.26

Figure 4.27

(b) Single-cycle, pipelined

Clock cycle length = 200 ps
### ISA Design for Pipelining

- **MIPS ISA designed for pipelining**
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

### MCU Application

- **[video] 德州儀器-(TI)-OMAP5-系統單晶片-(SoC)-介紹** (4:10)
- **[video] TI OMAP-5 demo** (4:15)
  - 新的使用者介面：gesture (手勢) recognition
  - DLP projection
  - Wireless video streaming
  - Mobile wallet (錢包)
  - M-shield (盾) transaction security
  - 3D video capture
  - Glass free 3D display
  - 支援4 screens
  - Augmented reality
Case Study: gesture control

- [video] Interaction-with-a-Quadrotor-via-the-Kinect (1:10)
- [video] Microsoft-Kinect-Motion-Sensing-Debuts-at-E3 (3:30)

Pipeline Hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle.

These events are called hazards

- Structure hazards
  - Conflict for use of a resource

- Data hazard
  - Need to wait for previous instruction to complete its data read/write

- Control hazard
  - Deciding on control action depends on previous instruction
**Structural Hazards**

- **Example**
  - If the pipeline in Figure 4.27(b) had a fourth instruction
    - In the same clock cycle that the first instruction is accessing data from memory while the fourth instruction is fetching an instruction from the same memory
  - Without two memories, our pipeline could have a structure hazard

![Diagram of pipeline with instructions and memory references](image)

![Diagram of processor structure](image)
Data Hazards

- Data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline

- Example
  
  \[
  \begin{align*}
  \text{add} & \quad \text{\texttt{$s0, \ t0, \ t1$}} \\
  \text{sub} & \quad \text{\texttt{$t2, \ s0, \ t3$}}
  \end{align*}
  \]

  The \texttt{add} instruction does not write its result until the fifth stage, meaning that we would have to add three \textit{bubbles} to the pipeline (next page)

How to Solve Data Hazards?

- Waiting 2 clock cycles
  - Correct, but too pessimistic

[Diagram showing the pipeline with bubbles added to resolve data hazards]
Exercise

- Solve three data hazards
  - add $3, $4, $6
  - sub $5, $3, $2
  - lw $7, 100($5)
  - add $8, $7, $2

Forwarding

- Use result ($s0) when it is computed
- Don’t wait for it to be stored in a register
- Requires extra connections in the datapath

Fig. 4.29
Case Study: NFC

- [video] NFC Film - Nokia (4:45)
- [video] NFC tags-Function-Demo (2:20)
- [video] Xperia™-SmartTags,讓生活簡單有智慧 (1:18)
- [video] Amazing-Smart-Mobile-Life-with-NFC (3:55)

Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!

Fig. 4.30
Exercise 1

Show the forwarding paths needed to execute the following four instructions:

- add $3, $4, $6
- sub $5, $3, $2
- lw $7, 100($5)
- add $8, $7, $2

Answer

Addition and subtraction operations are shown with forwarding paths in the pipeline stages.

The diagram illustrates the execution order and forwarding paths for the given instructions.
Exercise 2

- Identify all of the data dependencies in the following code.
  - Which dependencies are data hazards that will be resolved via forwarding?
  - Which dependencies are data hazards that will cause a stall?

```
add $3, $4, $2
sub $5, $3, $1
lw  $6, 200($3)
add $7, $3, $6
```

Answer

```
add $3, $4, $2
sub $5, $3, $1
lw  $6, 200($3)
add $7, $3, $6
```

- Answer
  - The data dependency between the load and the last add instruction cannot be resolved by using forwarding.
Reordering Code to Avoid Stalls

Example.
Consider the following code segment in C:
\[
A = B + E;
\]
\[
C = B + F;
\]
Here is the generated MIPS code for this segment:

\[
\begin{align*}
&\text{lw } \$t1, 0($t0) \quad \text{#load } B \\
&\text{lw } \$t2, 4($t0) \quad \text{#load } E \\
&\text{add } \$t3, \$t1, \$t2 \quad \text{ #$t3 = B+E} \\
&\text{sw } \$t3, 12($t0) \quad \text{#store } A = \$t3 \\
&\text{lw } \$t4, 8($t0) \quad \text{#load } F \\
&\text{add } \$t5, \$t1, \$t4 \quad \text{ #$t5 = B+F} \\
&\text{sw } \$t5, 16($t0) \quad \text{#store } C = \$t5
\end{align*}
\]

Reorder the instructions to avoid any pipeline stalls.

Answer.

Both \texttt{add} instructions have a hazard

On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.
### Reordering Code to Avoid Stalls

- **Answer (cont’d).**
  - Moving up the 3rd `lw` instruction eliminates both hazards:
    - `lw $t1, 0($t0)`
    - `lw $t2, 4($t0)`
    - `lw $t4, 8($t0)`
    - `add $t3, $t1, $t2`
    - `sw $t3, 12($t0)`
    - `add $t5, $t1, $t4`
    - `sw $t5, 16($t0)`

### Control Hazards

- **Control hazard arises from the need to make a decision based on the results of one instruction while others are executing**
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch
Example:

```
beq $1, $2, 40
xor $7, $8, $9
```

We must begin fetching the or instruction on the next clock cycle. But the pipeline cannot possibly know what the next instruction should be.

---

**Stall on Branch**

- `beq $1, $2, 40`
- `xor $7, $8, $9`

We must begin fetching the or instruction on the next clock cycle. But the pipeline cannot possibly know what the next instruction should be.
Stall on Branch

- Assume that we put in extra hardware so that we can test register, calculate the branch address, and update the PC during the 2nd stage of the pipeline.

![Diagram showing the pipeline stages (IF, ID, EXE, MEM, WB) with an example of a branch instruction `beq $1,$2,40` and `xor $7, $8, $9`]

Branch Prediction

- One simple approach is to always predict that branches will untaken
  - When you are right, the pipeline proceeds at full speed (see Figure 4.32 (a))
  - Only when branches are taken does the pipeline stall (see Figure 4.32 (b))
- A more sophisticated version of branch prediction would have some branches predicted as taken and some as untaken
MIPS with Predict Not Taken

![Diagram showing program execution order and time for correct and incorrect prediction scenarios.]

More-Realistic Branch Prediction

- **Static branch prediction**
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken

- **Dynamic branch prediction**
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history