6 ENHANCING PERFORMANCE WITH PIPELINING

6.1 An Overview of Pipelining

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, pipelining is key to making processor fast. Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction, but instruction throughput is the important metric because real programs execute billions of instructions. Many authors separate the pipeline into two categories:

- Instructional pipeline, where different stages of an instruction fetch and execution are handled in a pipeline.
- Arithmetic pipeline, where different stages of an arithmetic operation are handled along the stages of a pipeline.

Instructional pipeline is used in advanced microprocessors where the microprocessor begins executing a second instruction before the first has been completed. That is, several instructions are in the pipeline simultaneously, each at a different processing stage. The pipeline is divided into stages and each stage can execute its operation concurrently with the other stages. When a stage completes an operation, it passes the result to the next stage in the pipeline and fetches the next operation from the preceding stages. The final results of each instruction emerge at the end of the pipeline in rapid succession.

A RISC processor pipeline operates in much the same way, although the stages in the pipeline are different. While different processors have different numbers of stages, they are basically variations of these five, used in the MIPS R3000 processor:

1. fetch instructions from memory (F);
2. read registers and decode the instruction (D);
3. execute the instruction or calculate an address (E);
4. access an operand in data memory (M);
5. write the result into a register (W).
Each stage is separated by a register which stores the results of the previous stage so that the correct values are used in the next stage. The clock cycle length of the pipelined processor is the time it takes for the slowest stage of the pipeline to complete its function.

**Stage 1 - Instruction Fetch**

The Instruction Fetch (F) stage is responsible for obtaining the requested instruction from memory. The main components of the IF stage can be seen in the figure below. The instruction and the program counter (which is incremented to the next instruction) are stored in the IF/ID pipeline register as temporary storage so that they may be used in the next stage at the start of the next clock cycle.

![Figure 6.1 - Main Components of Stage 1](image1)

**Stage 2 - Instruction Decode**

The Instruction Decode (D) stage is responsible for decoding the instruction and sending out the various control lines to the other parts of the processor. Figure 6.2 shows the structure of the ID stage. The instruction is sent to the control unit where it is decoded and the read registers are fetched from the register file. The ID stage is also where branching occurs.

![Figure 6.2 - Main Components of Stage 2](image2)
Stage 3 - Execution
The Execution (E) stage is where any calculations are performed. The main component in this stage is the ALU. The ALU is made up of arithmetic, logic and shifting capabilities.

![Figure 6.3 - Main Components of Stage 3](image)

Stage 4 - Memory
The Memory (M) stage is responsible for storing and loading values to and from memory. It is also responsible for any input or output from the processor. If the current instruction is not of Memory type than the result from the ALU is passed through to the write back stage.

![Figure 6.4 - Main Components of Stage 4](image)

Stage 5 - Write Back
The Write Back (W) stage is responsible for writing the result of a calculation, memory access or input into the register file.

![Figure 6.5 - Main Components of Stage 5](image)
6.2 Single-Cycle versus Pipelined Performance

Ideally, each of the stages in a RISC processor pipeline should take 1 clock cycle so that the processor finishes an instruction each clock cycle and averages one cycle per instruction (CPI). Let's consider only 8 instructions: lw, sw, add, sub, and, or, slt, beq. The operation times for the major functional units are 2 ns for memory access, 2 ns for ALU operation, and 1 ns for register file read or write. As we said in Chapter 5, in the single-cycle model every instruction takes exactly 1 clock cycle, so the clock cycle period must be stretched to accommodate the slowest instruction.

Total time (ns) for 8 instructions calculated in Table 6.1.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instr. fetch</th>
<th>Register read</th>
<th>ALU operation</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>sw</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>R-format</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>beq</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

We can see that the longest instruction is lw – so the time required for every instruction is 8 ns. Figure 6.6 compares no pipelined and pipelined execution of three lw instructions. Thus, the time between the first and forth instructions in the no pipelined design is 3x8 = 24 ns.

All pipeline stages take a single clock cycle, so the clock cycle must be long enough to accommodate the slowest operation. Just as the single-clock design must take the worst-case clock cycle of 8 ns even though some instructions can be as fast as 5 ns, the pipeline execution clock cycle must have the worst-case clock cycle of 2 ns even though some stages take only 1 ns. Pipelining still offers a fourfold performance improvement: the time between the first and the forth instructions is 3x2 = 6 ns. So, we can write a formula (for ideal conditions):

\[
\text{Time between instructions (pipelined)} = \frac{\text{Time between instructions (no pipelined)}}{\text{Number of pipe stages}}
\]
The pipeline designer's goal is to balance the length of each pipeline stage. If the stages are perfectly balanced, then the time per instruction on the pipelined machine is equal to

\[
\text{Time per instruction on no-pipelined machine} \div \text{Number of pipe stages}
\]

Under these conditions, the speedup from pipelining equals the number of pipe stages. Usually, however, the stages will not be perfectly balanced; besides, the pipelining itself involves some overhead.

There are two disadvantages of pipeline architecture. The first is complexity. The second is the inability to continuously run the pipeline at full speed, i.e. the pipeline stalls.

### 6.3 Pipeline hazards

Let us examine why the pipeline cannot run at full speed. There are phenomena called pipeline hazards which disrupt the smooth execution of the pipeline. The resulting delays in the pipeline flow are called bubbles. These pipeline hazards include:

- data hazards arising from data dependencies;
- structural hazards from hardware conflicts;
- control hazards that come about from branch, jump, and other control flow changes.
Data Hazards

Data Hazards occur when an instruction attempts to use a register whose value depends on the result of previous instructions that have not yet finished.

In figure 6.7 result of the stage W of the addition must be used in stage D of the subtraction.

<table>
<thead>
<tr>
<th>Add $1, $2, $3</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>W↓</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub $4, $1, $5</td>
<td>F</td>
<td>↓D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>

Figure 6.7 - Example of a Data Hazard

There are two main ways of dealing with such hazards: stalling and forwarding.

Stalling

Stalling involves halting the flow of instructions until the required result is ready to be used. It is the simplest way to resolve a data hazard. However, as can be seen in Figure 6.8 stalling wastes processor time by doing nothing while waiting for the result.

<table>
<thead>
<tr>
<th>Add $1, $2, $3</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>W↓</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>Stall</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>Stall</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>Sub $4, $1, $5</td>
<td>F</td>
<td>↓D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>

Figure 6.8 - Stalling the Pipeline

Forwarding

The forwarding method is best described through the use of an example. Figure 6.9 shows two instructions in the pipeline, you can see that the SUB instruction needs the result of the ADD instruction in the SUB's E stage but the ADD instruction does not write the result until the ADD's W stage. However, you can also see that the result for the ADD instruction is actually computed before the SUB instruction needs it so the result is forwarded from the E/M stage back to the E stage of the SUB instruction.

<table>
<thead>
<tr>
<th>Add $1, $2, $3</th>
<th>F</th>
<th>D</th>
<th>E↓</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub $4, $1, $5</td>
<td>F</td>
<td>D↓</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>

Figure 6.9 - Forwarding
Reordering Code to Avoid Pipeline Stalls

Example: Find the hazard in this code from the body of the swap procedure:

```plaintext
# reg $5 has the address of v[k]

sll $2,$5,2 # $2 ← k × 4
add $2, $4, $2 # $2 = v+(k × 4)
lw $15, 0($2) # $15 ← v[k]
lw $16, 4($2) # $16 = v[k+1]
sw $16, 0($2) # v[k] gets $16 which is v[k+1]
sw $15, 4($2) # v[k+1] gets $15 (temp, which is v[k])
```

The hazard occurs on register $16 between the second lw and the first sw. Swapping the two sw instructions removes this hazard:

```plaintext
sll $2,$5,2 # $2 ← k × 4
add $2, $4, $2 # $2 = v+(k × 4)
lw $15, 0($2) # $15 ← v[k]
lw $16, 4($2) # $16 = v[k+1]
sw $15, 4($2) # v[k+1] gets $15 which is v[k]
sw $16, 0($2) # v[k] gets $16 (temp, which is v[k+1])
```

Note that we do not create a new hazard because there is still one instruction between the write of register $15 by the load and the read of register $15 in the store. Thus, on a machine with forwarding, the reordered sequence takes 4 clock cycles.

Branch prediction

Branch instructions are those that tell the processor to make a decision about what the next instruction to be executed should be based on the results of another instruction. Branch instructions can be troublesome in a pipeline if a branch is conditional on the results of an instruction which has not yet finished its path through the pipeline.

For example:

```plaintext
Loop : add $r3, $r2, $r1
       sub $r6, $r5, $r4
       beq $r3, $r6, Loop
```
The example above instructs the processor to add r1 and r2 and put the result in r3, then subtract r4 from r5, storing the difference in r6. In the third instruction, beq stands for branch if equal. If the contents of r3 and r6 are equal, the processor should execute the instruction labeled "Loop." Otherwise, it should continue to the next instruction. In this example, the processor cannot make a decision about which branch to take because neither the value of r3 or r6 has been written into the registers yet.

The processor could stall, but a more sophisticated method of dealing with branch instructions is branch prediction. The processor makes a guess about which path to take - if the guess is wrong, anything written into the registers must be cleared, and the pipeline must be started again with the correct instruction. Some methods of branch prediction depend on stereotypical behavior. Branches pointing backward are taken about 90% of the time since backward-pointing branches are often found at the bottom of loops. On the other hand, branches pointing forward, are only taken approximately 50% of the time. Thus, it would be logical for processors to always follow the branch when it points backward, but not when it points forward. Other methods of branch prediction are less static: processors that use dynamic prediction keep a history for each branch and uses it to predict future branches. These processors are correct in their predictions 90% of the time.

Still other processors forgo the entire branch prediction ordeal. The RISC System/6000 fetches and starts decoding instructions from both sides of the branch. When it determines which branch should be followed, it then sends the correct instructions down the pipeline to be executed.

Control Hazard
A control hazard occurs whenever there is a change in the normal execution flow of the program. Events such as branches, interrupts, exceptions and return from interrupts. A hazard occurs because branches, interrupts etc are not caught until the instruction is decoded in the second stage. By the time it is decoded the following instruction is already entered into the pipeline and left unchecked an unwanted instruction would remain in the pipeline. There is really only one solution to this type of hazard. That is, to implement a hardware stall. The hardware stall simply flushes the offending instruction from the pipeline.

Structural Hazard
A structural hazard occurs when the hardware is unable to handle certain combinations of instruction simultaneously. For example, the IF stage under normal conditions will be accessing the memory on every clock cycle. When a load or store
word instruction is used the MEM stage tries to access the memory and because of the single memory architecture a conflict occurs. There are a couple of ways of dealing with such conflicts-stalling and prefetching.

**Stalling**

This method works in the same way as stalling for data hazards. Instead of accessing memory in the IF stage the load/store instruction is allowed to use memory and the processor is simply stalled until the load/store instruction is finished. The problem with this method like with data hazards is that it can take a long time if there are multiple load/store instructions in a row.

**Prefetching**

Prefetching involves fetching two instructions in the F stage and storing them in a small buffer. The buffer size that we have used is 4 instructions to save on the hardware required. Due to the fact that we fetch two instruction in the F stage, when a load/store instruction is used it is allowed to access memory and the instruction for the F stage is fetched from the buffer and no instructions are fetched from memory. This prefetching method is used to solve this particular hazard.

**NOTE:** Prefetching works better than stalling only if the memory used is fast enough to do two accesses in one clock cycle.

### 6.4 Pipeline Control

Figure 6.25 shows the control lines on the pipelined data-path. We borrow as much as we can from the control for the simple data-path in Chapter 5. To specify control for the pipeline, we need only set the control values during each pipeline stage. Since pipelining the data-path leaves the meaning of the control lines unchanged, we can use the same control values as before. Figure 6.10 has the same values as in Chapter 5, but now the nine control lines are grouped by pipeline stage. They have been shuffled into three groups corresponding to the last three pipeline stages.

<table>
<thead>
<tr>
<th>Instruc.</th>
<th>Ex/Addr. calculation stage</th>
<th>Mem. stage</th>
<th>WR stage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ex/Addr. calculation stage</td>
<td>Mem. stage</td>
<td>WR stage</td>
</tr>
<tr>
<td>Instruc.</td>
<td>RegDst</td>
<td>ALUOp1</td>
<td>ALUOp0</td>
</tr>
<tr>
<td>R-type</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Implementing control means setting the nine control lines to these values in each stage for each instruction. The simplest way to do it is to extend the pipeline registers to include control information. Since the control lines start with the EX stage, we can create the control information during instruction decode.

Figure 6.29 shows that these control signals are then used in appropriate pipeline stage as the instruction moves down the pipeline. Figure 6.30 shows the full data-path with the extended pipeline registers and with the control lines connected to the proper stage.
FIGURE 6.25 The pipelined datapath of Figure 6.18 with the control signals identified. This datapath borrows the control logic for PC source, register destination number, and ALU control from Chapter 5. Note that we now need the 6-bit funct field (function code) of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register. Recall that these 6 bits are also the 6 least significant bits of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves these bits unchanged.
FIGURE 6.30 The pipelined datapath of Figure 6.26, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.