There exist two basic techniques to increase the instruction execution rate of a processor. These are to increase the clock rate, thus decreasing the instruction execution time, or alternatively to increase the number of instructions that can be executed simultaneously. Pipelining and instruction-level parallelism are examples of the latter technique. Pipelining owes its origin to car assembly lines. The idea is to have more than one instruction being processed by the processor at the same time. Similar to the assembly line, the success of a pipeline depends upon dividing the execution of an instruction among a number of subunits (stages), each performing part of the required operations. A possible division is to consider instruction fetch ($F$), instruction decode ($D$), operand fetch ($F$), instruction execution ($E$), and store of results ($S$) as the subtasks needed for the execution of an instruction. In this case, it is possible to have up to five instructions in the pipeline at the same time, thus reducing instruction execution latency. In this Chapter, we discuss the basic concepts involved in designing instruction pipelines. Performance measures of a pipeline are introduced. The main issues contributing to instruction pipeline hazards are discussed and some possible solutions are introduced. In addition, we introduce the concept of arithmetic pipelining together with the problems involved in designing such a pipeline. Our coverage concludes with a review of a recent pipeline processor.

9.1. GENERAL CONCEPTS

Pipelining refers to the technique in which a given task is divided into a number of subtasks that need to be performed in sequence. Each subtask is performed by a given functional unit. The units are connected in a serial fashion and all of them operate simultaneously. The use of pipelining improves the performance compared to the traditional sequential execution of tasks. Figure 9.1 shows an illustration of the basic difference between executing four subtasks of a given instruction (in this case fetching $F$, decoding $D$, execution $E$, and writing the results $W$) using pipelining and sequential processing.
It is clear from the figure that the total time required to process three instructions \( (I_1, I_2, I_3) \) is only six time units if four-stage pipelining is used as compared to 12 time units if sequential processing is used. A possible saving of up to 50% in the execution time of these three instructions is obtained. In order to formulate some performance measures for the goodness of a pipeline in processing a series of tasks, a space-time chart (called the Gantt’s chart) is used. The chart shows the succession of the subtasks in the pipe with respect to time. Figure 9.2 shows a Gantt’s chart. In this chart, the vertical axis represents the subunits (four in this case) and the horizontal axis represents time (measured in terms of the time unit required for each unit to perform its task). In developing the Gantt’s chart, we assume that the time \( (T) \) taken by each subunit to perform its task is the same; we call this the unit time.

As can be seen from the figure, 13 time units are needed to finish executing 10 instructions \( (I_1 \text{ to } I_{10}) \). This is to be compared to 40 time units if sequential processing is used (ten instructions each requiring four time units).

In the following analysis, we provide three performance measures for the goodness of a pipeline. These are the Speed-up \( S(n) \), Throughput \( U(n) \), and Efficiency \( E(n) \). It should be noted that in this analysis we assume that the unit time \( T = t \) units.

1. **Speed-up \( S(n) \)** Consider the execution of \( m \) tasks (instructions) using \( n \)-stages (units) pipeline. As can be seen, \( n + m - 1 \) time units are required
to complete $m$ tasks.

$$\text{Speed-up } S(n) = \frac{\text{Time using sequential processing}}{\text{Time using pipeline processing}} = \frac{m \times n \times t}{(n + m - 1) \times t}$$

$$= \frac{m \times n}{n + m - 1}$$

$$\lim_{m \to \infty} S(n) = n \quad \text{(i.e., n-fold increase in speed is theoretically possible)}$$

2. Throughput $U(n)$

$$\text{Throughput } U(n) = \text{no. of tasks executed per unit time} = \frac{m}{(n + m - 1) \times t}$$

$$\lim_{m \to \infty} U(n) = 1 \text{ assuming that } t = 1 \text{ unit time}$$

3. Efficiency $E(n)$

$$\text{Efficiency } E(n) = \frac{\text{Ratio of the actual speed-up to the maximum speed-up}}{n} = \frac{m}{n + m - 1}$$

$$\lim_{m \to \infty} E(n) = 1$$

9.2. INSTRUCTION PIPELINE

The simple analysis made in Section 9.1 ignores an important aspect that can affect the performance of a pipeline, that is, pipeline stall. A pipeline operation is said to have been stalled if one unit (stage) requires more time to perform its function, thus forcing other stages to become idle. Consider, for example, the case of an instruction fetch that incurs a cache miss. Assume also that a cache miss requires three extra time units. Figure 9.3 illustrates the effect of having instruction $I_2$ incurring a cache miss (assuming the execution of ten instructions $I_1$ to $I_{10}$).

![Figure 9.3](https://mywbut.com)

Figure 9.3  Effect of a cache miss on the pipeline
The figure shows that due to the extra time units needed for instruction $I_2$ to be fetched, the pipeline stalls, that is, fetching of instruction $I_3$ and subsequent instructions are delayed. Such situations create what is known as pipeline bubble (or pipeline hazards). The creation of a pipeline bubble leads to wasted unit times, thus leading to an overall increase in the number of time units needed to finish executing a given number of instructions. The number of time units needed to execute the 10 instructions shown in Figure 9.3 is now 16 time units, compared to 13 time units if there were no cache misses.

Pipeline hazards can take place for a number of other reasons. Among these are instruction dependency and data dependency. These are explained below.

### 9.2.1. Pipeline “Stall” Due to Instruction Dependency

Correct operation of a pipeline requires that operation performed by a stage MUST NOT depend on the operation(s) performed by other stage(s). Instruction dependency refers to the case whereby fetching of an instruction depends on the results of executing a previous instruction. Instruction dependency manifests itself in the execution of a conditional branch instruction. Consider, for example, the case of a “branch if negative” instruction. In this case, the next instruction to fetch will not be known until the result of executing that “branch if negative” instruction is known. In the following discussion, we will assume that the instruction following a conditional branch instruction is not fetched until the result of executing the branch instruction is known (stored). The following example shows the effect of instruction dependency on a pipeline.

**Example 1** Consider the execution of ten instructions $I_1$ to $I_{10}$ on a pipeline consisting of four pipeline stages: $IF$ (instruction fetch), $ID$ (instruction decode), $IE$ (instruction execute), and $IS$ (instruction results store). Assume that the instruction $I_4$ is a conditional branch instruction and that when it is executed, the branch is not taken, that is, the branch condition(s) is(are) not satisfied. Assume also that when the branch instruction is fetched, the pipeline stalls until the result of executing the branch instruction is stored. Show the succession of instructions in the pipeline; that is, show the Gantt’s chart. Figure 9.4 shows the required Gantt’s chart. The bubble created due to the pipeline stall is clearly shown in the figure.

![Figure 9.4](mywbut.com)
9.2.2. Pipeline “Stall” Due to Data Dependency

Data dependency in a pipeline occurs when a source operand of instruction $I_i$ depends on the results of executing a preceding instruction, $I_j$, $i > j$. It should be noted that although instruction $I_i$ can be fetched, its operand(s) may not be available until the results of instruction $I_j$ are stored. The following example shows the effect of data dependency on a pipeline.

**Example 2** Consider the execution of the following piece of code:

- $ADD \ R_1, R_2, R_3; \ R_3 \leftarrow R_1 + R_2$
- $SL \ R_3; \ R_3 \leftarrow SL(R_3)$
- $SUB \ R_5, R_6, R_4; \ R_4 \leftarrow R_5 - R_6$

In this piece of code, the first instruction, call it $I_i$, adds the contents of two registers $R_1$ and $R_2$ and stores the result in register $R_3$. The second instruction, call it $I_{i+1}$, shifts the contents of $R_3$ one bit position to the left and stores the result back into $R_3$. The third instruction, call it $I_{i+2}$, stores the result of subtracting the content of $R_6$ from the content of $R_5$ in register $R_4$. In order to show the effect of such data dependency, we will assume that the pipeline consists of five stages, $IF, ID, OF, IE$, and $IS$. In this case, the $OF$ stage represents the operand fetch stage. The functions of the remaining four stages remain the same as explained before. Figure 9.5 shows the Gantt’s chart for this piece of code. As shown in the figure, although instruction $I_{i+1}$ has been successfully decoded during time unit $k + 2$, this instruction cannot proceed to the $OF$ unit during time unit $k + 3$. This is because the operand to be fetched by $I_{i+1}$ during time unit $k+3$ should be the content of register $R_3$, which has been modified by execution of instruction $I_i$. However, the modified value of $R_3$ will not be available until the end of time unit $k + 4$. This will require instruction $I_{i+1}$ to wait (at the output of the $ID$ unit) until $k + 5$. Notice that instruction $I_{i+2}$ will

![Figure 9.5](image-url)
have also to wait (at the output of the IF unit) until such time that instruction $I_{i+1}$ proceeds to the ID. The net result is that pipeline stall takes place due to the data dependency that exists between instruction $I_i$ and instruction $I_{i+1}$.

The data dependency presented in the above example resulted because register $R_3$ is the destination for both instructions $I_i$ and $I_{i+1}$. This is called a write-after-write data dependency. Taking into consideration that any register can be written into (or read from), then a total of four different possibilities exist, including the write-after-write case. The other three cases are read-after-write, write-after-read, and read-after-read. Among the four cases, the read-after-read case should not lead to pipeline stall. This is because a register read operation does not change the content of the register. Among the remaining three cases, the write-after-write (see the above example) and the read-after-write lead to pipeline stall. The following piece of code illustrates the read-after-write case:

$$\begin{align*}
\text{ADD} & \quad R_1, R_2, R_3; \quad R_3 \leftarrow R_1 + R_2 \\
\text{SUB} & \quad R_3, 1, R_4; \quad R_4 \leftarrow R_3 - 1
\end{align*}$$

In this case, the first instruction modifies the content of register $R_3$ (through a write operation) while the second instruction uses the modified contents of $R_3$ (through a read operation) to load a value into register $R_4$. While these two instructions are proceeding within a pipeline, care should be taken so that the value of register $R_3$ read in the second instruction is the updated value resulting from execution of the previous instruction. Figure 9.6 shows the Gantt’s chart for this case assuming that the first instruction is called $I_i$ and the second instruction is called $I_{i+1}$.

It is clear that the operand of the second instruction cannot be fetched during time unit $k+3$ and that it has to be delayed until time unit $k + 5$. This is because the modified value of the content of register $R_3$ will not be available until time slot $k + 5$.

![Figure 9.6](mywbut.com) The read after write data dependency
Fetching the operand of the second instruction during time slot $k + 3$ will lead to incorrect results.

**Example 3** Consider the execution of the following sequence of instructions on a five-stage pipeline consisting of $IF$, $ID$, $OF$, $IE$, and $IS$. It is required to show the succession of these instructions in the pipeline.

$I_1 \rightarrow \text{Load } 1, R1; \quad R1 \leftarrow 1;$
$I_2 \rightarrow \text{Load } 5, R2; \quad R2 \leftarrow 5;$
$I_3 \rightarrow \text{Sub } R2, 1, R2 \quad R2 \leftarrow R2 - 1;$
$I_4 \rightarrow \text{Add } R1, R2, R3; \quad R3 \leftarrow R1 + R2;$
$I_5 \rightarrow \text{Add } R4, R5, R6; \quad R6 \leftarrow R4 + R5;$
$I_6 \rightarrow \text{SL } R3 \quad R3 \leftarrow \text{SL} (R3)$
$I_7 \rightarrow \text{Add } R6, R4, R7; \quad R7 \leftarrow R4 + R6;$

In this example, the following data dependencies are observed:

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Type of data dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_3$ and $I_2$</td>
<td>Read after write and write after write (W W)</td>
</tr>
<tr>
<td>$I_4$ and $I_1$</td>
<td>Read after write (R W)</td>
</tr>
<tr>
<td>$I_4$ and $I_3$</td>
<td>Read after write (R W)</td>
</tr>
<tr>
<td>$I_6$ and $I_4$</td>
<td>Read after write and write after write (W W)</td>
</tr>
<tr>
<td>$I_7$ and $I_5$</td>
<td>Read after write (R W)</td>
</tr>
</tbody>
</table>

Figure 9.7 illustrates the progression of these instructions in the pipeline taking into consideration the data dependencies mentioned above. The assumption made in constructing the Gantt’s chart in Figure 9.7 is that fetching an operand by an instruction that depends on the results of a previous instruction execution is delayed until such operand is available, that is, the result is stored. A total of 16 time units are required to execute the given seven instructions taking into consideration the data dependencies among the different instructions.
Based on the results obtained above, we can compute the speed-up and the throughput for executing the piece of code given in Example 3 as:

\[
\text{Speed-up } S(5) = \frac{\text{Time using sequential processing}}{\text{Time using pipeline processing}} = \frac{7 \times 5}{16} = 2.19
\]

\[
\text{Throughput } U(5) = \text{No. of tasks executed per unit time} = \frac{7}{16} = 0.44
\]

The discussion on pipeline stall due to instruction and data dependencies should reveal three main points about the problems associated with having such dependencies. These are:

1. Both instruction and data dependencies lead to added delay in the pipeline.
2. Instruction dependency can lead to the fetching of the wrong instruction.
3. Data dependency can lead to the fetching of the wrong operand.

There exist a number of methods to deal with the problems resulting from instruction and data dependencies. Some of these methods try to prevent the fetching of the wrong instruction or the wrong operand while others try to reduce the delay incurred in the pipeline due to the existence of instruction or data dependency. A number of these methods are introduced below.

**Methods Used to Prevent Fetching the Wrong Instruction or Operand**

**Use of NOP (No Operation)** This method can be used in order to prevent the fetching of the wrong instruction, in case of instruction dependency, or fetching the wrong operand, in case of data dependency. Recall Example 1. In that example, the execution of a sequence of ten instructions \( I_1 \) to \( I_{10} \) on a pipeline consisting of four pipeline stages: IF, ID, IE, and IS were considered. In order to show the execution of these instructions in the pipeline, we have assumed that when the branch instruction is fetched, the pipeline stalls until the result of executing the branch instruction is stored. This assumption was needed in order to prevent fetching the wrong instruction after fetching the branch instruction. In real-life situations, a mechanism is needed to guarantee fetching the appropriate instruction at the appropriate time. Insertion of “NOP” instructions will help carrying out this task. A “NOP” is an instruction that has no effect on the status of the processor.

**Example 4** Consider the execution of ten instructions \( I_1 \) to \( I_{10} \) on a pipeline consisting of four pipeline stages: IF, ID, IE, and IS. Assume that instruction \( I_4 \) is a conditional branch instruction and that when it is executed, the branch is not taken; that is, the branch condition is not satisfied.
In order to execute this set of instructions while preventing the fetching of the wrong instruction, we assume that a specified number of NOP instructions have been inserted such that they follow instruction $I_4$ in the sequence and they precede instruction $I_5$. Figure 9.8 shows the Gantt’s chart illustrating the execution of the new sequence of instructions (after inserting the NOP instructions). The figure shows that the insertion of THREE NOP instructions after instruction $I_4$ will guarantee that the correct instruction to fetch after $I_4$, in this case $I_5$, will only be fetched during time slot number 8 at which the result of executing $I_4$ would have been stored and the condition for the branch would have been known.

It should be noted that the number of NOP instructions needed is equal to $(n - 1)$, where $n$ is the number of pipeline stages.

Example 4 illustrates the use of NOP instructions to prevent fetching the wrong instruction in the case of instruction dependency. A similar approach can be used to prevent fetching the wrong operand in the case of data dependency. Consider the execution of the following piece of code on a five-stage pipeline ($IF$, $ID$, $OF$, $IE$, $IS$).

```
ADD R_1, R_2, R_3;     R_3 ← R_1 + R_2
SUB R_3, 1, R_4;       R_4 ← R_3 - 1
MOV R_5, R_6;          R_6 ← R_5
```

Note the data dependency in the form of read-after-write (R-W) between the first two instructions. Fetching the operand for the second instruction, that is, fetching the content of $R_3$, cannot proceed until the result of the first instruction has been stored. In order to achieve that, NOP instructions can be inserted between the first two instructions as shown below.

```
ADD R_1, R_2, R_3;     R_3 ← R_1 + R_2
NOP
NOP
SUB R_3, 1, R_4;       R_4 ← R_3 - 1
MOV R_5, R_6;          R_6 ← R_5
```

Execution of the modified sequence of instructions is shown in Figure 9.9. The figure shows that the use of NOP guarantees that during time unit #6 instruction...
I_2 will fetch the correct value of R_3. This is the value stored as a result of executing instruction I_1 during time unit #5.

Methods Used to Reduce Pipeline Stall Due to Instruction Dependency

Unconditional Branch Instructions

In order to be able to reduce the pipeline stall due to unconditional branches, it is necessary to identify the unconditional branches as early as possible and before fetching the wrong instruction. It may also be possible to reduce the stall by reordering the instruction sequence. These methods are explained below.

Reordering of Instructions

In this case, the sequence of instructions are reordered such that correct instructions are brought to the pipeline while guaranteeing the correctness of the final results produced by the reordered set of instructions. Consider, for example, the execution of the following group of instructions I_1, I_2, I_3, I_4, I_5, ..., I_j, I_{j+1}, ... on a pipeline consisting of three pipeline stages: IF, IE, and IS. In this group of instructions, I_4 is an unconditional branch instruction whereby the target instruction is I_j. Execution of this group of instructions in the same sequence as given will lead to the incorrect fetching of instruction I_5 after fetching instruction I_4. However, consider execution of the reordered sequence I_1, I_4, I_2, I_3, I_5, ..., I_j, I_{j+1}, ... Execution of this reordered sequence using the three-stage pipeline is shown in Figure 9.10.

The figure shows that the reordering of the instructions causes instruction I_j to be fetched during time unit #5, that is, after instruction I_4 has been executed. Reordering of instructions can be done using a “smart” compiler that can scan the sequence of code and decide on the appropriate reordering of instructions that will lead to

![Figure 9.9](https://www.mywbut.com) Use of NOP in data dependency

![Figure 9.10](https://www.mywbut.com) Instruction reordering
producing the correct final results while minimizing the number of time units lost due to the instruction dependency. One important condition that must be satisfied in order for the reordering of the instruction method to produce correct results is that the set of instructions that are swapped with the branch instruction hold no data and/or instruction dependency relationship among them.

USE OF DEDICATED HARDWARE IN THE FETCH UNIT In this case, the fetch unit is assumed to have associated with it a dedicated hardware unit capable of recognizing unconditional branch instructions and computing the branch target address as quickly as possible. Consider, for example, the execution of the same sequence of instructions as illustrated above. Assume also that the fetch unit has a dedicated hardware unit capable of recognizing unconditional branch instructions and computing the branch address using no additional time units. Figure 9.11 shows the Gantt’s chart for this sequence of instructions. The figure shows that the correct sequence of instructions is executed while incurring no extra unit times.

The assumption of needing no additional time units to recognize branch instructions and computing the target branch address is unrealistic. In typical cases, the added hardware unit to the fetch unit will require additional time unit(s) to carry out its task of recognizing branch instructions and computing target branch addresses. During the extra time units needed by the hardware unit, if other instructions can be executed, then the number of extra time units needed may be reduced and indeed may be eliminated altogether. This is the essence of the method shown below.

PRECOMPUTING OF BRANCHES AND REORDERING OF INSTRUCTIONS This method can be considered as a combination of the two methods discussed in the previous two sections above. In this case, the dedicated hardware (used to recognize branch instructions and computing the target branch address) executes its task concurrently with the execution of other instructions. Consider, for example, the same sequence of instructions given above. Assume also that the dedicated hardware unit requires one time unit to carry out its task. In this case, reordering of the instructions to become \( I_1, I_2, I_4, I_3, I_5, \ldots, I_j, I_{j+1}, \ldots \) should produce the correct results while causing no additional lost time units. This is illustrated using the Gantt’s chart in Figure 9.12. Notice that time unit #4 is used by the dedicated hardware unit.

![Figure 9.11 Use of additional hardware unit for branch instruction recognition](mywbut.com)
hardware unit to compute the target branch address concurrently with the fetching of instruction $I_3$.

It should be noted that the success of this method depends on the availability of instructions to be executed concurrently while the dedicated hardware unit is computing the target branch address. In the case presented above, it was assumed that reordering of instructions can provide those instructions that can be executed concurrently with the target branch computation. However, if such reordering is not possible, then the use of an instruction queue together with prefetching of instructions can help provide the needed conditions. This is explained below.

**INSTRUCTION PREFETCHING** This method requires that instructions can be fetched and stored in an instruction queue before they are needed. The method also calls for the fetch unit to have the required hardware needed to recognize branch instructions and compute the target branch address. If a pipeline stalls due to data dependency causing no new instructions to be fetched into the pipeline, then the fetch unit can use such time to continue fetching instructions and add them to the instruction queue. On the other hand, if a delay in the fetching of instructions occurs, for example, due to instruction dependency, then those prefetched instructions in the instruction queue can be used to provide the pipeline with new instructions, thus eliminating some of the otherwise lost time units due to instruction dependency. Providing the appropriate instruction from the instruction queue to the pipeline is usually done using what is called a “dispatch unit.” The technique of prefetching of instructions and executing them during a pipeline stall due to instruction dependency is called “branch folding.”

**Conditional Branch Instructions** The techniques discussed above in the context of unconditional branch instructions may not work in the case of conditional branch instructions. This is because in conditional branching the target branch address will not be known until the execution of the branch instruction has been completed. Therefore, a number of techniques can be used to minimize the number of lost time units due to instruction dependency represented by conditional branching.

**DELAYED BRANCH** Delayed branch refers to the case whereby it is possible to fill the location(s) following a conditional branch instruction, called the *branch delay slot(s)*, with useful instruction(s) that can be executed until the target
branch address is known. Consider, for example, the execution of the following program loop on a pipeline consisting of two stages: Fetch (F) and Execute (E).

\[
\begin{align*}
I_1 &\rightarrow Again: \ Load \ 5, \ R_1; \quad R_1 \leftarrow 5; \\
I_2 &\rightarrow Sub \ R_2; \quad R_2 \leftarrow R_2 - 1; \\
I_3 &\rightarrow Bnn \ Again; \quad \text{Branch to Again if result is Not Negative;} \\
I_4 &\rightarrow Add \ R_4, R_5, R_3; \quad R_3 \leftarrow R_4 + R_5;
\end{align*}
\]

It should be noted that at the end of the first loop, either instruction \(I_1\) or instruction \(I_4\) will have to be fetched depending on the result of executing instruction \(I_3\). The way with which such a situation has been dealt will delay fetching of the next instruction until the result of executing instruction \(I_3\) is known. This will lead to incurring extra delay in the pipeline. However, this extra delay may be avoided if the sequence of instructions has been reordered to become as follows.

\[
\begin{align*}
Again: \quad Sub \ R_2; \quad R_2 \leftarrow R_2 - 1; \\
Load \ 5, R_1; \quad R_1 \leftarrow 5; \\
Bnn \ Again; \quad \text{Branch to Again if result is Not Negative;} \\
Add \ R_4, R_5, R_3; \quad R_3 \leftarrow R_4 + R_5;
\end{align*}
\]

Figure 9.13 shows the Gantt’s chart for executing the modified piece of code for the case \(R_2 \geq 3\) before entering the loop. The figure indicates that branching takes place one instruction later than the actual place where the branch instruction appears in the original instruction sequence, hence the name “delayed branch.” It is also clear from Figure 9.13 that by reordering the sequence of instructions, it was possible to fill the branch delay time slot with a useful instruction, thus eliminating any extra delay in the pipeline. It has been shown in a number of studies that “smart” compilers were able to make use of one branch delay time slot in more than 80% of the cases. The use of branch delay time slots has led to the improvement of both the speed-up and the throughput of those processors using “smart” compilers.

**Prediction of the Next Instruction to Fetch** This method tries to reduce the time unit(s) that can potentially be lost due to instruction dependency by predicting the next instruction to fetch after fetching a conditional branch instruction. The basis is that if the branch outcomes are random, then it would be possible to save about 50% of the otherwise lost time. A simple way to carry out such a technique is to

![Figure 9.13 Delayed branch](mywbut.com)
assume that whenever a conditional branch is encountered, the system predicts that the branch will not be taken (or alternatively will be taken). In this way, fetching of instructions in sequential address order will continue (or fetching of instructions starting from the target branch instruction will continue). At the completion of the branch instruction execution, the results will be known and a decision will have to be made as to whether the instructions that were executed assuming that the branch will not be taken (or taken) were the intended correct instruction sequence or not. The outcome of this decision is one of two possibilities. If the prediction was correct, then execution can continue with no wasted time units. If, on the other hand, the wrong prediction has been made, then care must be taken such that the status of the machine, measured in terms of memory and register contents, should be restored as if no speculative execution took place.

Prediction based on the above scheme will lead to the same branch prediction decision every time a given instruction is encountered, hence the name static branch prediction. It is the simplest branch prediction scheme and is done during compilation time.

Another technique that can be used in branch prediction is dynamic branch prediction. In this case, prediction is done at run time, rather than at compile time. When a branch is encountered, then a record is checked to find out whether that same branch has been encountered before and if so, what was the decision made at that time; that is, was the branch taken or not taken. A run time decision is then made whether to take or not to take the branch. In making such a decision, a two-state algorithm, “likely to be taken” (LTK) or “likely not to be taken” (LNK), can be followed. If the current state is LTK and if the branch is taken, then the algorithm will maintain the LTK state; otherwise it will switch to the LNK. If, on the other hand, the current state is LNK and the branch is not taken, then the algorithm will maintain the LNK state; otherwise it will switch to the LTK state. This simple algorithm should work fine, particularly if the branch is going backwards, for example during the execution of a loop. It will, however, lead to misprediction when control reaches the last pass through the loop. A more robust algorithm that uses four states has been used by the ARM 11 microarchitecture (see below).

It is interesting to notice that a combination of dynamic and static branch prediction techniques can lead to performance improvement. An attempt to use a dynamic branch prediction is first made, and if it is not possible, then the system can resort to the static prediction technique.

Consider, for example, the ARM 11 microarchitecture (the first implementation of the ARMv6 instruction set architecture). This architecture uses a dynamic/static branch prediction combination. A record in the form of a 64-entry, four-state branch target address cache (BTAC) is used to help the dynamic branch prediction finding whether a given branch has been encountered before. If the branch has been encountered, the record will also show whether it was most frequently taken or most frequently not taken. If the BTAC shows that a branch has been encountered before, then a prediction is made based on the previous outcome. The four states are: strongly taken, weakly taken, strongly not taken, and weakly not taken.
In the case that a record cannot be found for a branch, then a static branch prediction procedure is used. The static branch prediction procedure investigates the branch to find out whether it is going backwards or forwards. A branch going backwards is assumed to be part of a loop and the branch is assumed to be taken. A branch going forwards is not taken.  The ARM 11 employs an eight-stage pipeline. Every correctly predicted branch is found to lead to a typical saving of five processor clock cycles. Around 80% of branches are found to be correctly predicted using the dynamic/static combination in the ARM 11 architecture. The pipeline features of the ARM 11 are introduced in the next subsection.

A branch prediction technique based on the use of a 16K-entry branch history record is employed in the UltraSPARC III RISC processor, a 14-stage pipeline. However, the impact of a misprediction, in terms of the number of cycles lost due to a branch misprediction is reduced by using the following approach. On predictions that a branch will be taken and while the branch target instructions are being fetched, the “fall-through” instructions are prepared for issue in parallel through the use of a four-entry branch miss queue (BMQ). This reduces the misprediction penalty to two cycles. The UltraSPARC III has achieved 95% success in branch prediction. The pipeline features of the UltraSPARC III are introduced in the next subsection.

**Methods Used to Reduce Pipeline Stall Due to Data Dependency**

**Hardware Operand Forwarding**  Hardware operand forwarding allows the result of one ALU operation to be available to another ALU operation in the cycle that immediately follows. Consider the following two instructions.

\[
\begin{align*}
\text{ADD} & \quad R_1, R_2, R_3; \quad R_3 \leftarrow R_1 + R_2 \\
\text{SUB} & \quad R_3, 1, R_4; \quad R_4 \leftarrow R_3 - 1
\end{align*}
\]

It is easy to notice that there exists a read-after-write data dependency between these two instructions. Correct execution of this sequence on a five-stage pipeline (IF, ID, OF, IE, IS) will cause a stall of the second instruction after decoding it and until the result of the first instruction is stored in \( R_3 \). Only at that time, the operand of the second instruction, that is, the new value stored in \( R_3 \), can be fetched by the second instruction. However, if it is possible to have the result of the first instruction forwarded to the ALU during the same time unit as it is being stored in \( R_3 \), then it will be possible to reduce the stall time. This is illustrated in Figure 9.14.

The assumption that the operand of the second instruction be forwarded immediately after it is available and while it is being stored in \( R_3 \) requires a modification in the data path such that an added feedback path is created to allow for such operand forwarding. This modification is shown using dotted lines in Figure 9.15. It should be noted that the needed modification to achieve hardware operand forwarding is expensive and requires careful issuing of control signals. It should also be noted that if it is possible to perform both instruction decoding and operand fetching during the same time unit, then there will be no lost time units.
Software Operand Forwarding

Operand forwarding can alternatively be performed in software by the compiler. In this case, the compiler should be “smart” enough to make the result(s) of performing some instructions quickly available, as operand(s), for subsequent instruction(s). This desirable feature requires the compiler to perform data dependency analysis in order to determine the operand(s) that can possibly be made available (forwarded) to subsequent instructions, thus reducing the stall time. This data dependency analysis requires the recognition of basically three forms. These are explained below using simple examples.

**STORE-FETCH** This case represents data dependency in which the result of an instruction is stored in memory followed by a request for a fetch of the same result by a subsequent instruction. Consider the following sequence of two instructions:

\[
\text{Store } R_2, (R_3); \quad M[R_3] \leftarrow R_2
\]
\[
\text{Load } (R_3), R_4; \quad R_4 \leftarrow M[R_3]
\]

In this sequence, the operand needed by the second instruction (the contents of memory location whose address is stored in register \( R_3 \)) is already available in register \( R_2 \) and therefore can be immediately (forwarded) moved into register \( R_4 \).
When it recognizes such data dependency, a “smart” compiler can replace the above sequence by the following sequence:

\[
\begin{align*}
\text{Store} & \quad R_2, (R_3); \quad M[R_3] \leftarrow R_2 \\
\text{Move} & \quad R_2, R_4; \quad R_4 \leftarrow R_2
\end{align*}
\]

**FETCH-FETCH**  This case represents data dependency in which the data stored by an instruction is also needed as an operand by a subsequent instruction. Consider the following instruction sequence:

\[
\begin{align*}
\text{Load} & \quad (R_3), R_2; \quad R_2 \leftarrow M[R_3] \\
\text{Load} & \quad (R_3), R_4; \quad R_4 \leftarrow M[R_3]
\end{align*}
\]

In this sequence, the operand needed by the first instruction (the contents of memory location whose address is stored in register \(R_3\)) is also needed as an operand for the second instruction. Therefore, this operand can be immediately (forwarded) moved into register \(R_4\). When it recognizes such data dependency, a “smart” compiler can replace the above sequence by the following sequence.

\[
\begin{align*}
\text{Load} & \quad (R_3), R_2; \quad R_2 \leftarrow M[R_3] \\
\text{Move} & \quad R_2, R_4; \quad R_4 \leftarrow R_2
\end{align*}
\]

**STORE-STORE**  This is the case in which the data stored by an instruction is overwritten by a subsequent instruction. Consider the following instruction sequence:

\[
\begin{align*}
\text{Store} & \quad R_2, (R_3); \quad M[R_3] \leftarrow R_2 \\
\text{Store} & \quad R_4, (R_3); \quad M[R_3] \leftarrow R_4
\end{align*}
\]

In this sequence, the results written during the first instruction (the content of register \(R_2\) is written into memory location whose address is stored in register \(R_3\)) is overwritten during the second instruction by the contents of register \(R_4\). Assuming that these two instructions are executed in sequence and that the result written by the first instruction will not be needed by an I/O operation, for example, a DMA, then the sequence of these two instructions can be replaced by the following single instruction.

\[
\begin{align*}
\text{Store} & \quad R_4, (R_3); \quad M[R_3] \leftarrow R_4
\end{align*}
\]
9.5. ARITHMETIC PIPELINE

The principles used in instruction pipelining can be used in order to improve the performance of computers in performing arithmetic operations such as add, subtract, and multiply. In this case, these principles will be used to realize the arithmetic circuits inside the ALU. In this section, we will elaborate on the use of arithmetic pipeline as a means to speed up arithmetic operations. We will start with fixed-point arithmetic operations and then discuss floating-point operations.

9.5.1. Fixed-Point Arithmetic Pipelines

The basic fixed point arithmetic operation performed inside the ALU is the addition of two \( n \)-bit operands \( A = a_n a_{n-1} a_{n-2} \cdots a_2 a_1 a_0 \) and \( B = b_n b_{n-1} b_{n-2} \cdots b_2 b_1 b_0 \). Addition of these two operands can be performed using a number of techniques. These techniques differ in basically two attributes: degree of complexity and achieved speed. These two attributes are somewhat contradictory; that is, a simple realization may lead to a slower circuit while a complex realization may lead to a faster circuit. Consider, for example, the carry ripple through (CRTA) and a carry look-ahead (CLAA) adders. The CRTA is simple, but slower, while the CLAA is complex, but faster.
It is possible to modify the CRTA in such a way that a number of pairs of operands are operated upon, that is, pipelined, inside the adder, thus improving the overall speed of addition in the CRTA. Figure 9.18 shows an example of a modified 4-bit CRTA. In this case, the two operands A and B are presented to the CRTA through the use of synchronizing elements, such as clocked latches. These latches will guarantee that the movement of the partial carry values within the CRTA are synchronized at the input of the subsequent stages of the adder with the higher order operand bits. For example, the arrival of the first carry out \( (c_0) \) and the second pair of bits \( (a_1 \) and \( b_1) \) is synchronized at the input of the second full adder (counting from low order bits to high order bits) using a latch.

Although the operation of the modified CRTA remains in principle the same; that is, the carry ripples through the adder, the provision of latches allows for the possibility of presenting multiple sets of pairs of operands to the adder at the same time. Consider, for example, the case of adding \( M \) pairs of operands, whereby the operands of each pair are \( n \)-bit. The time needed to perform the addition of these \( M \) pairs using a nonpipelined CRTA is given by \( T_{np} = M \times n \times T_a \), where \( T_a \) is the time needed to perform single bit addition. This is to be compared to the time needed to perform the same computation using a pipelined CTRA which is given by \( T_{pp} = (n + M - 1) \times T_a \). For example, if \( M = 16 \) and \( n = 64 \) bits, then we have \( T_{np} = 1024 \times T_a \) and \( T_{pp} = 79 \times T_a \), thus resulting in a speed-up of about 13. In the extreme case whereby it is possible to present unlimited number of pairs of operands \( (M) \) to the CRTA at the same time, the speed up will reach 64, the number of bits in each operand.

![Figure 9.18](image-url) A modified 4 bit CRTA
9.5.2. Floating-Point Arithmetic Pipelines

Using a similar approach, it is possible to pipeline floating-point (FP) addition/subtraction. In this case, the pipeline will have to be organized around the operations needed to perform FP addition. The main operations needed in FP addition are exponent comparison (EC), exponent alignment (EA), addition (AD), and normalization (NZ). Therefore, a possible pipeline organization is to have a four-stage pipeline,
each performing an operation from EC, EA, AD, and NZ. Figure 9.19 shows a schematic for a pipeline FP adder. It is possible to have multiple sets of FP operands proceeding inside the adder at the same time, thus reducing the overall time needed for FP addition. Synchronizing latches are needed, as before, in order to synchronize the operands at the input of a given stage in the FP adder.

9.5.3. Pipelined Multiplication Using Carry-Save Addition

As indicated before, one of the main problems with addition is the fact that the carry has to ripple through from one stage to the next. Carry rippling through stages can be eliminated using a method called carry-save addition. Consider the case of adding 44, 28, 32, and 79. A possible way to add these without having the carry ripple through is illustrated in Figure 9.20. The idea is to delay the addition of the carry resulting in the intermediate stages until the last step in the addition. Only at the last stage is a carry-ripple stage employed.

$$P = A \cdot B$$
$$= A \cdot (B_7 \cdot 2^7 + B_6 \cdot 2^6 + B_5 \cdot 2^5 + \ldots + B_0 \cdot 2^0)$$
$$= \sum_{i=0}^{7} A \cdot B_i \cdot 2^i$$
$$= \sum_{i=0}^{7} S_i$$

$S_i = A \cdot B_i \cdot 2^i$ represents a 16-bit partial product.

**Figure 9.21** A carry save based multiplication of two 8 bit operands $M$ and $Q$

**Figure 9.22** Carry save addition based multiplication scheme
Carry-save addition can be used to realize a pipelined multiplication building block. Consider, for example, the multiplication of two \( n \)-bit operands \( A \) and \( B \). The multiplication operation can be transformed into an addition as shown in Figure 9.21. The figure illustrates the case of multiplying two 8-bit operands \( A \) and \( B \). A carry-save based multiplication scheme using the principle shown in Figure 9.21 is shown in Figure 9.22. The scheme is based on the idea of producing the set of partial products needed and then adding them up using a carry-save addition scheme.
3. A computer system has a four-stage pipeline consisting of an instruction fetch unit (F), an instruction decode unit (D), an instruction execution unit (E), and a write unit (W). Compute the speed-up time $P(4)$, throughput $U(4)$, and the efficiency $\zeta(4)$ of the pipeline in executing a code segment consisting of 20 instructions, given that branch instructions occur as follows: $I_3, I_9, I_{10}, I_{15}, I_{20}$. Assume that when a branch instruction is fetched, the pipeline stalls until the next instruction to fetch is known. Determine the time required to execute those same 20 instructions using two-way interleaved memory if the functions performed by the F, E, and W units require the use of the memory. What is the average number of cycles per instruction in both cases? Use the following space time chart to compute the number of time units.

4. Consider the integer multiplication of two 16-bit numbers $M$ and $Q$ to produce a product $P$. Show that this operation can be represented as

$$P = \sum_{i=0}^{15} P_i$$

where $P_i = M \times Q_i \times 2^i$ represents a 32-bit partial product.

Design a pipeline unit to perform the above operation using the minimum number of carry-save adders and one carry-look-ahead adder. Show also the design of a pipeline for performing floating-point addition/subtraction. Give numerical examples to support your design.

5. A computer system has a three-stage pipeline consisting of a Fetch unit (F), a Decode unit (D), and an Execute (E) unit. Determine (using the space time chart) the time required to execute 20 sequential instructions using two-way interleaved memory if all three units require the use of the memory simultaneously.

6. What is the average instruction processing time of a five-stage instruction pipeline for 36 instructions if conditional branch instructions occur as follows: $I_5, I_7, I_{10}, I_{25}, I_{27}$. Use both the space time chart and the analytical model.

7. A computer has a five-stage instruction pipeline of one cycle each. The five stages are: Instruction Fetch (IF), Instruction Decode (ID), Operand Fetch (OF), Instruction Execution (IE), and Operand Store (OS). Consider the following code sequence, which is to be run on this computer.

```
Load 1, R1; R1 ← 1;
Load 5, R2; R2 ← 5;
Again: Sub R2, 1, R2; R2 ← R2 - 1;
Add R1, R2, R3; R3 ← R1 + R2;
Bnn Again; branch to Again if result is Not Negative;
Add R4, R5, R6; R6 ← R4 + R5;
Add R6, R4, R7; R7 ← R4 + R6;
```
a. Analyze the execution of the above piece of code in order to calculate the number of cycles needed to execute the above code without pipelining, assuming that each instruction requires exactly 5 cycles to execute.

b. Calculate (using the Gantt’s chart) the number of cycles needed to execute the above code if the pipeline described above is used. Assume that there is no forwarding hardware and that when branch instructions are fetched, the pipeline will “stall” until the target address is calculated and the branch decision is made. Ignore any data dependency.

c. Repeat (b) above if data dependency is considered with the remaining conditions the same.

d. Calculate the percentage of improvement due to the use of pipeline in each of the above cases (b) and (c).

REFERENCES AND FURTHER READING


