Chapter 9 – Pipeline and Vector Processing

Section 9.1 – Parallel Processing

- A parallel processing system is able to perform concurrent data processing to achieve faster execution time
- The system may have two or more ALUs and be able to execute two or more instructions at the same time
- Also, the system may have two or more processors operating concurrently
- Goal is to increase the throughput – the amount of processing that can be accomplished during a given interval of time
- Parallel processing increases the amount of hardware required

- Example: the ALU can be separated into three units and the operands diverted to each unit under the supervision of a control unit
- All units are independent of each other
- A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components
- Parallel processing can be classified from:
  - The internal organization of the processors
  - The interconnection structure between processors
  - The flow of information through the system
  - The number of instructions and data items that are manipulated simultaneously
- The sequence of instructions read from memory is the instruction stream
- The operations performed on the data in the processor is the data stream
• Parallel processing may occur in the instruction stream, the data stream, or both

• Computer classification:
  o Single instruction stream, single data stream – SISD
  o Single instruction stream, multiple data stream – SIMD
  o Multiple instruction stream, single data stream – MISD
  o Multiple instruction stream, multiple data stream – MIMD

• SISD – Instructions are executed sequentially. Parallel processing may be achieved by means of multiple functional units or by pipeline processing

• SIMD – Includes multiple processing units with a single control unit. All processors receive the same instruction, but operate on different data.

• MIMD – A computer system capable of processing several programs at the same time.

• We will consider parallel processing under the following main topics:
  o Pipeline processing
  o Vector processing
  o Array processors

Section 9.2 -- Pipelining

• Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments

• Each segment performs partial processing dictated by the way the task is partitioned

• The result obtained from the computation in each segment is transferred to the next segment in the pipeline

• The final result is obtained after the data have passed through all segments

• Can imagine that each segment consists of an input register followed by a combinational circuit

• A clock is applied to all registers after enough time has elapsed to perform all segment activity

• The information flows through the pipeline one step at a time

• Example: \( A_i \times B_i + C_i \) for \( i = 1, 2, 3, \ldots, 7 \)

• The suboperations performed in each segment are:

\[
\begin{align*}
R1 & \leftarrow A_i, \quad R2 \leftarrow B_i \\
R3 & \leftarrow R1 \times R2, \quad R4 \leftarrow C_i \\
R5 & \leftarrow R3 + R4
\end{align*}
\]
Figure 9-2  Example of pipeline processing.

![Diagram of pipeline processing]

<table>
<thead>
<tr>
<th>Clock Pulse Number</th>
<th>Segment 1</th>
<th>Segment 2</th>
<th>Segment 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
</tr>
<tr>
<td>1</td>
<td>$A_1$</td>
<td>$B_1$</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>$A_2$</td>
<td>$B_2$</td>
<td>$A_1 \cdot B_1$</td>
</tr>
<tr>
<td>3</td>
<td>$A_3$</td>
<td>$B_3$</td>
<td>$A_2 \cdot B_2$</td>
</tr>
<tr>
<td>4</td>
<td>$A_4$</td>
<td>$B_4$</td>
<td>$A_3 \cdot B_3$</td>
</tr>
<tr>
<td>5</td>
<td>$A_5$</td>
<td>$B_5$</td>
<td>$A_4 \cdot B_4$</td>
</tr>
<tr>
<td>6</td>
<td>$A_6$</td>
<td>$B_6$</td>
<td>$A_5 \cdot B_5$</td>
</tr>
<tr>
<td>7</td>
<td>$A_7$</td>
<td>$B_7$</td>
<td>$A_6 \cdot B_6$</td>
</tr>
<tr>
<td>8</td>
<td>—</td>
<td>—</td>
<td>$A_7 \cdot B_7$</td>
</tr>
<tr>
<td>9</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**TABLE 9-1 Content of Registers in Pipeline Example**
Any operation that can be decomposed into a sequence of suboperations of about the same complexity can be implemented by a pipeline processor.

The technique is efficient for those applications that need to repeat the same task many times with different sets of data.

A task is the total operation performed going through all segments of a pipeline.

The behavior of a pipeline can be illustrated with a space-time diagram.

This shows the segment utilization as a function of time.

Once the pipeline is full, it takes only one clock period to obtain an output.

Consider a \( k \)-segment pipeline with a clock cycle time \( t_p \) to execute \( n \) tasks.

The first task \( T_1 \) requires time \( kt_p \) to complete.

The remaining \( n - 1 \) tasks finish at the rate of one task per clock cycle and will be completed after time \( (n - 1)t_p \).

The total time to complete the \( n \) tasks is \( [k + n - 1]t_p \).

The example of Figure 9-4 requires \( [4 + 6 - 1] \) clock cycles to finish.

Consider a nonpipeline unit that performs the same operation and takes \( t_n \) time to complete each task.

The total time to complete \( n \) tasks would be \( nt_n \).

The speedup of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio

\[
S = \frac{nt_n}{(k + n - 1)t_p}
\]

As the number of tasks increase, the speedup becomes

\[
S = \frac{t_n}{t_p}
\]
• If we assume that the time to process a task is the same in both circuits, \( t_n = k t_p \)
  \[
  S = \frac{kt_n}{t_p} = k
  \]

• Therefore, the theoretical maximum speedup that a pipeline can provide is \( k \)

• Example:
  
  o Cycle time = \( t_p = 20 \) ns
  o # of segments = \( k = 4 \)
  o # of tasks = \( n = 100 \)

  The pipeline system will take \((k + n - 1)t_p = (4 + 100 -1)20\text{ns} = 2060\text{ ns}\)

  Assuming that \( t_n = kt_p = 4 \times 20 = 80\text{ ns}, \)
  A nonpipeline system requires \( nkt_p = 100 \times 80 = 8000\text{ ns} \)
  The speedup ratio = \( 8000/2060 = 3.88 \)

• The pipeline cannot operate at its maximum theoretical rate
• One reason is that the clock cycle must be chosen to equal the time delay of the segment with the maximum propagation time
• Pipeline organization is applicable for arithmetic operations and fetching instructions

Section 9.3 – Arithmetic Pipeline

• Pipeline arithmetic units are usually found in very high speed computers
• They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems

• Example for floating-point addition and subtraction
• Inputs are two normalized floating-point binary numbers
  
  \[ \begin{align*}
  X &= A \times 2^a \\
  Y &= B \times 2^b
  \end{align*} \]

  A and B are two fractions that represent the mantissas
  a and b are the exponents
• Four segments are used to perform the following:
  o Compare the exponents
  o Align the mantissas
  o Add or subtract the mantissas
  o Normalize the result
Figure 9-6  Pipeline for floating-point addition and subtraction.
• $X = 0.9504 \times 10^3$ and $Y = 0.8200 \times 10^2$
• The two exponents are subtracted in the first segment to obtain $3-2=1$
• The larger exponent 3 is chosen as the exponent of the result
• Segment 2 shifts the mantissa of Y to the right to obtain $Y = 0.0820 \times 10^3$
• The mantissas are now aligned
• Segment 3 produces the sum $Z = 1.0324 \times 10^3$
• Segment 4 normalizes the result by shifting the mantissa once to the right and incrementing the exponent by one to obtain $Z = 0.10324 \times 10^4$

Section 9.4 – Instruction Pipeline

• An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments
• This causes the instruction fetch and execute phases to overlap and perform simultaneous operations
• If a branch out of sequence occurs, the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded

• Consider a computer with an instruction fetch unit and an instruction execution unit forming a two segment pipeline
• A FIFO buffer can be used for the fetch segment
• Thus, an instruction stream can be placed in a queue, waiting for decoding and processing by the execution segment
• This reduces the average access time to memory for reading instructions
• Whenever there is space in the buffer, the control unit initiates the next instruction fetch phase
• The following steps are needed to process each instruction:
  o Fetch the instruction from memory
  o Decode the instruction
  o Calculate the effective address
  o Fetch the operands from memory
  o Execute the instruction
  o Store the result in the proper place
• The pipeline may not perform at its maximum rate due to:
  o Different segments taking different times to operate
  o Some segment being skipped for certain operations
  o Memory access conflicts
• Example: Four-segment instruction pipeline
• Assume that the decoding can be combined with calculating the EA in one segment
- Assume that most of the instructions store the result in a register so that the execution and storing of the result can be combined in one segment

- Up to four suboperations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time
- It is assumed that the processor has separate instruction and data memories

- Reasons for the pipeline to deviate from its normal operation are:
  - *Resource conflicts* caused by access to memory by two segments at the same time.
  - *Data dependency* conflicts arise when an instruction depends on the result of a previous instruction, but his result is not yet available

*Figure 9-7* Four-segment CPU pipeline.
Branch difficulties arise from program control instructions that may change the value of PC

- Methods to handle data dependency:
  - Hardware interlocks are circuits that detect instructions whose source operands are destinations of prior instructions. Detection causes the hardware to insert the required delays without altering the program sequence.
  - Operand forwarding uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments. This requires additional hardware paths through multiplexers as well as the circuit to detect the conflict.
  - Delayed load is a procedure that gives the responsibility for solving data conflicts to the compiler. The compiler is designed to detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no-operation instructions.

- Methods to handle branch instructions:
  - Prefetching the target instruction in addition to the next instruction allows either instruction to be available.
  - A branch target buffer is an associative memory included in the fetch segment of the branch instruction that stores the target instruction for a previously executed branch. It also stores the next few instructions after the branch target instruction. This way, the branch instructions that have occurred previously are readily available in the pipeline without interruption.
  - The loop buffer is a variation of the BTB. It is a small very high speed register file maintained by the instruction fetch segment of the pipeline. Stores all branches within a loop segment.
  - Branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed. The pipeline then begins prefetching instructions from the predicted path.
  - Delayed branch is used in most RISC processors so that the compiler rearranges the instructions to delay the branch.