Pipeline and Vector Processing
Pipelining and vector processing are two methods of speeding up processing by performing multiple operations in parallel. We don’t speed up individual computations; rather, we perform more computations simultaneously in order to generate more results in less time.

A pipeline is very much like an assembly line. Each stage in the pipeline performs its operation on data and passes the data to the next stage. Each stage works on different data simultaneously. Each computation takes the same amount of time, but, since you process data simultaneously in the different stages of the pipeline, results are generated more quickly.

After a brief introduction, we begin by examining two types of pipelines. Arithmetic pipelines are used to speed up computations, and instruction pipelines are used to speed up the fetch/decode/execute process.

Next, we review vector processing, in which entire computations are performed in parallel on different data. We examine how to interleave memory in order to avoid data conflicts and examine an algorithm using a SIMD architecture. Finally, concluding remarks are presented.
One method to achieve parallelism within a CPU is to have multiple function units. Instead of having one unit within the ALU which performs all possible computations, there are several sub-units, each of which can perform one operation or type of operation. It is possible to have more than one unit active at a given time, thus achieving parallelism and reducing overall computation time.

See figure 9.1, p. 300 of the textbook.
Flynn classifies computers based on how they handle instructions and data. Single instruction single data (SISD) computers read in a single instruction, decode and execute that instruction, and then go to the next instruction. A low level microprocessor is a SISD CPU. Single instruction multiple data (SIMD) computers are popular for operations which manipulate data by performing the same operation on all data. Applications which manipulate matrices, such as graphics rendering, perform fast Fourier transformations, and other such operations, are well suited for SIMD architectures.

Multiple instruction single data (MISD) computers are not used; this is included in the classification solely for completeness. Multiple instruction multiple data (MIMD) computers are classic supercomputers, which are beyond the scope of this course.
To illustrate arithmetic pipelining, consider a loop which performs the computation \((A[i] + B[i]) \times C[i]\). Here we create a pipeline to perform this operation. The first stage loads \(A[i]\), \(B[i]\) and \(C[i]\) into registers \(R1\), \(R2\) and \(R3\), respectively. The second stage adds \(A[i]\) and \(B[i]\) and stores the result in \(R4\). It also passes \(C[i]\) directly from \(R3\) to \(R5\). Finally, the third stage multiplies these two values and stored them in \(R6\).

This process doesn’t speed up individual computation. In fact, due to the time needed to latch the data, this function actually takes more time to complete. However, data can be fed into the pipeline continuously. While the data for \(i=1\) is in the second stage, the data for \(i=2\) is loaded into the first stage.
Here is the data flow for this pipeline. Note that new data is introduced during every clock cycle; therefore, a result is produced every cycle (excluding the initial latency to fill the pipeline).
The space-time diagram gives an overview of where data is within the pipeline during a given cycle. It is a condensed version of the table shown in the previous slide.
Speedup is a measure of the performance of a pipeline. It is the ratio of the time needed to process \( n \) pieces of data using a non-pipelined CPU to the time needed using a pipelined CPU. Here, \( t_n \) is the sequential processing time. It is equal to the sum of the times needed to carry out each operation plus the time needed to latch the final result. \( t_p \) is the clock period of the pipeline. It is equal to the time of the slowest stage plus the time needed to latch the results. As \( n \) approaches infinity, \( n/(k+n-1) \) approaches 1 and the speedup approaches \( t_n/t_p \). The maximum speedup is \( k \) since, in the best possible case, the processing time \( t_n \) is partitioned exactly evenly among the \( k \) stages of the pipeline, and \( t_p = t_n/k \). In reality, this does not occur due to the delay of the latches.
For our example, assume the adder takes 20 ns to do its task, the multiplier takes 25 ns and each register takes 5 ns to latch its inputs. The sequential time is 50 ns (20 + 25 + 5). Since the slowest stage takes 25 ns to multiply data and 5 ns to latch the result, tp = 30 ns. The results of the speedup calculations are shown above.
Arithmetic processes which can be broken into sequential sub-processes are good candidates for pipelining. Floating point addition is one such operation. A floating point number is stored as two values: the mantissa and the exponent. Assume that $X$ is stored in floating point format as $A \times 10^a$, where $A$ is the mantissa and $0.1 \leq A < 1$, i.e. $A$ does not have a leading zero. ($X=0$ is a special case; we assume $X \neq 0$ in this example.)

We break this process into four steps. First, we compare the exponents to see which is greater. Then we align the mantissas. This simply converts the format of the numbers so that they have the same exponent. For example, we may convert $.1 \times 10^2$ to $.001 \times 10^4$. In the third step, we add or subtract the mantissas. (We may need to subtract if one of the numbers is negative.) Finally, we normalize the result in the case where the mantissa is greater than or equal to one, or is less than 0.1.
This pipeline implements the 4-step process for floating point addition. Notice that I have added latches at the end of stages 1 and 2 which is not included in the figure in the text. Not having this latch will cause problems in the pipelined implementation because otherwise the value would be overwritten if data is introduced in two consecutive cycles.
Example: floating point addition

\[ X = .97 \times 10^2 \quad Y = .81 \times 10^1 \]

Stage 1: Compare the exponents

2>1, so align Y

Consider the example shown here, where we wish to calculate \( X + Y \). In the first stage, we compare the exponents and determine that the exponent of \( X \) is one greater than the exponent of \( Y \).
In stage 2, we align the mantissas by shifting the mantissa of \( Y \) one position to the right. This has the effect of expressing \( Y = .81 \times 10^1 \) as \( Y = .081 \times 10^2 \). Note that this does not change the value of \( Y \), only its representation.
Now that the mantissas are aligned, i.e. both numbers have the same exponent, we can add the mantissas. Here the result is greater than one, but the next stage will take care of that.
Example: floating point addition

\[ X = .97 \times 10^2 \quad Y = .81 \times 10^1 = .081 \times 10^2 \]

Stage 4: Normalize the result

\[ X + Y = 1.051 \times 10^2 = .1051 \times 10^3 \]

The result of the addition, \(1.051 \times 10^2\), is not in normal form, which requires the mantissa to be at least 0.1 but less than one. In this stage we shift the mantissa one position to the right and increment the exponent by one. This converts \(1.051 \times 10^2\) to \(.1051 \times 10^3\), or \(97 + 8.1 = 105.1\).
Processing instructions is another operation that lends itself well to pipelining. With a few variations, instruction pipelines follow this four-step process. First, the instruction is fetched from memory. Then, it is decoded and any effective addresses are calculated. The effective address may be supplied by the instruction directly or it may need to be computed, for example, if the address is indexed. In the third stage, operands are fetched from memory, and in the fourth stage the instruction is executed.
We can implement this process as a 4-stage pipeline, as shown here. Note that, just as with the arithmetic pipeline, registers separate the stages.
Here is a space-time diagram showing how this process works. Notice that this is similar to the space-time diagram for the arithmetic pipeline.
Pipelining offers the opportunity to speed up the processing of instructions, but it introduces several problems of its own. Resource conflicts occur when two stages try to access the same memory module at the same time. For example, this may occur when one instruction is being fetched and another is fetching operands from the same memory module simultaneously. This can be avoided by properly allocating data to memory modules which don’t contain instructions.

When an instruction needs results which are calculated by a previous instruction which may not be ready, we have a data dependency conflict. We will examine this shortly.

Branch difficulties occur when you process jump instructions, because other instructions will be in the pipeline. We’ll also examine this in more detail shortly.
Consider this example. During time = 4, we execute the instruction which performs $A = B + C$. Simultaneously, we fetch operands $A$ and $E$ for the second instruction. This will be the old value of $A$, not the new value calculated in instruction 1. This is the data dependency problem.
There are several methods of resolving data dependency. Hardware interlocking uses hardware to detect when a data dependency will occur and to insert delays into the pipeline so that the data conflict is avoided.

Operand forwarding is another hardware method. It also senses when a data dependency will occur, but it routes the operands from within the pipeline to the part of the pipeline which needs them. In our example, it forwards the operand A from the execute stage back to the operand fetch stage, where the second instruction needs it.

Finally, a compiler can resolve data dependency by inserting no-ops to effect a delayed load. This is similar to hardware interlocking, but it moves the task out of hardware and into software.
Here is the same example using delayed load. By inserting a NOP between the two original instructions, the value $A$ is calculated and stored before it is fetched by the last instruction.
In this example, we see that equation 3 has entered the pipeline and is being processed even though it should not be executed. Also, it will be necessary to flush the pipeline with no-ops before taking the jump. Neither effect is desirable.
There are several possible ways to resolve branch difficulties. One possibility is to create two instruction streams into the pipeline and to fill one with the instructions that will be executed if the branch is taken and the other with the instructions that will be executed if it is not taken. When the branch is executed, the correct stream is accessed for future instructions. Another is to use a branch target buffer, an associative memory into which potential instructions are fetched. If the branch is taken, this provides a quicker location for the instructions to be executed.

The loop buffer is a variation of the branch target buffer. This is a cache memory which stores loops in their entirety, thus avoiding repeated memory accesses to fetch the instructions. Branch prediction uses probability to “guess” whether or not a branch will be taken. Depending on the odds, it fetches either the following instruction or the instruction branched to as the next instruction in the pipeline. When the guess is correct, performance improves. This is particularly useful for loops, which always branch back except for the last iteration.

Finally, there is the delayed branch, in which the compiler rearranges instructions to enter the pipeline in order to perform useful work while the branch is being processed, without modifying the function of the program.
Consider the program shown here. Once the JUMP instruction is taken, three no-ops are inserted to clear the pipeline. (The address X is not entered into the program counter until time=7, so the instruction at X cannot be fetched until time=8.) We seek to improve performance by getting rid of the no-ops.

### Example: Delayed Branch

<table>
<thead>
<tr>
<th>Time:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>FI</td>
<td>DA</td>
<td>FO</td>
<td>EX</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>INCR</td>
<td>---</td>
<td>FI</td>
<td>DA</td>
<td>FO</td>
<td>EX</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>ADD</td>
<td>---</td>
<td>---</td>
<td>FI</td>
<td>DA</td>
<td>FO</td>
<td>EX</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>JUMP X</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>FI</td>
<td>DA</td>
<td>FO</td>
<td>EX</td>
<td>---</td>
</tr>
<tr>
<td>NOP</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>FI</td>
<td>DA</td>
<td>FO</td>
<td>EX</td>
</tr>
<tr>
<td>NOP</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>FI</td>
<td>DA</td>
<td>FO</td>
</tr>
<tr>
<td>NOP</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>FI</td>
<td>DA</td>
</tr>
<tr>
<td>X: SUB</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>FI</td>
</tr>
</tbody>
</table>
We implement a delayed branch by reordering the instructions. The LOAD, INCR and ADD are moved after the JUMP and enter the pipeline while the JUMP is being processed. In essence, we replace the no-ops with instructions which must be executed anyway. As long as the overall result is not changed, this is acceptable. (For example, if the JUMP was indexed and one of these instructions changed the contents of the index register, we could not do this.)

Note that the SUB instruction at location X must still wait until the JUMP instruction has moved X into the program counter before it can be fetched.
Unlike pipelining, vector processing achieves parallelism by having multiple processing units each handle individual computations. For example, for the code shown, we might have four processors, each of which computes the result for a different value of $i$ simultaneously.
One bottleneck in this system is memory access. You can have as many processors as you want. However, if they all have to get data from the same memory module, you won’t reduce computing time, since the processors will have to wait until the memory module is free in order to get the data they need to perform their computations.

To resolve these problems, we use memory interleaving. This is the process of distributing memory locations and data among different memory modules in order to maximize memory access. In this example, the four processors can access the data for $i = 1, 2, 3$ and $4$ in parallel. They can then do so for the next four values of $i$, $i = 5, 6, 7$ and $8$, and so on.
Matrix multiplication is a typical application which can be processed using SIMD architectures. Each element C[i,j] is the sum of the element-wise product of elements of row i of the first matrix and their corresponding elements in column j of the second matrix.
This code implements the matrix multiplication. The i loop processes the rows of the product matrix c. The j loop processes the individual elements of row i. The k loop processes the individual products which comprise entry c[i,j] of the matrix.
Example: Parallel code

PARFOR i = 1 TO n
    PARFOR j = 1 TO n
        { c_{i,j} = 0; 
            FOR k = 1 TO n
                { c_{i,j} = c_{i,j} + a_{i,k} * b_{k,j} }
            }
    }

Since the elements of c can be computed independent of each other, this algorithm can be parallelized by calculating every element of c simultaneously. This is done by parallelizing the i and j loops. The k loop generates a running total and cannot be parallelized.
Example: Parallel code execution

\[ k = 1 \text{ (Initially } C = 0): \]

\[\begin{align*}
c_{1,1} &= c_{1,1} + a_{1,1} \times b_{1,1} \\
c_{1,2} &= c_{1,2} + a_{1,1} \times b_{1,2} \\
c_{1,3} &= c_{1,3} + a_{1,1} \times b_{1,3} \\
c_{3,1} &= c_{3,1} + a_{3,1} \times b_{1,1} \\
c_{3,2} &= c_{3,2} + a_{3,1} \times b_{1,2} \\
c_{3,3} &= c_{3,3} + a_{3,1} \times b_{1,3}
\end{align*}\]

Here is the result of the first iteration of the \( k \) loop.
Example: Parallel code execution

$k = 2$

\[
\begin{align*}
c_{1,1} &= c_{1,1} + a_{1,2} \times b_{2,1} \\
c_{1,2} &= c_{1,2} + a_{1,2} \times b_{2,2} \\
c_{1,3} &= c_{1,3} + a_{1,2} \times b_{2,3} \\
c_{2,1} &= c_{2,1} + a_{2,2} \times b_{2,1} \\
c_{2,2} &= c_{2,2} + a_{2,2} \times b_{2,2} \\
c_{2,3} &= c_{2,3} + a_{2,2} \times b_{2,3} \\
c_{3,1} &= c_{3,1} + a_{3,2} \times b_{2,1} \\
c_{3,2} &= c_{3,2} + a_{3,2} \times b_{2,2} \\
c_{3,3} &= c_{3,3} + a_{3,2} \times b_{2,3}
\end{align*}
\]

These operations occur when $k = 2$. 
When \( k = 3 \), the final results are generated.

There is one major problem with this algorithm. As can be seen from the individual computations, we use the same three values of \( a \) and the same three values of \( b \) in all nine operations. This will cause a data access conflict. Ideally, we would like to get rid of this altogether.
Since each value in c is the sum of three distinct products, we may juggle the order in which these products are calculated and added to the running total in order to avoid memory access conflicts, as long as the same products are generated. We do this by varying the product generated for each value of k for each element in a row of c. Setting $x = (i + j + k) \mod n + 1$ does this for us.
As can be seen here, no two products use the same value of a or b in any two computations.

**Example: Alternate code execution**

\[
\begin{align*}
\text{k} = 1 \text{ (Initially C = 0):} \\
\text{c}_{1,1} &= \text{c}_{1,1} + a_{1,1} \times \text{b}_{1,1} \\
\text{c}_{1,2} &= \text{c}_{1,2} + a_{1,2} \times \text{b}_{2,2} \\
\text{c}_{1,3} &= \text{c}_{1,3} + a_{1,3} \times \text{b}_{3,3} \\
\text{c}_{3,1} &= \text{c}_{3,1} + a_{3,1} \times \text{b}_{3,1} \\
\text{c}_{3,2} &= \text{c}_{3,2} + a_{3,2} \times \text{b}_{2,3} \\
\text{c}_{2,1} &= \text{c}_{2,1} + a_{2,2} \times \text{b}_{2,1} \\
\text{c}_{2,2} &= \text{c}_{2,2} + a_{2,3} \times \text{b}_{3,2} \\
\text{c}_{2,3} &= \text{c}_{2,3} + a_{2,1} \times \text{b}_{1,3} \\
\text{c}_{3,3} &= \text{c}_{3,3} + a_{3,2} \times \text{b}_{2,3} 
\end{align*}
\]
Example: Alternate code execution

\[ k = 2: \]

\[ c_{1,1} = c_{1,1} + a_{1,2} \cdot b_{2,1} \]
\[ c_{2,1} = c_{2,1} + a_{2,3} \cdot b_{3,1} \]
\[ c_{1,2} = c_{1,2} + a_{1,3} \cdot b_{3,2} \]
\[ c_{2,2} = c_{2,2} + a_{2,1} \cdot b_{1,2} \]
\[ c_{1,3} = c_{1,3} + a_{1,1} \cdot b_{1,3} \]
\[ c_{2,3} = c_{2,3} + a_{2,2} \cdot b_{2,3} \]
\[ c_{3,1} = c_{3,1} + a_{3,1} \cdot b_{1,1} \]
\[ c_{3,2} = c_{3,2} + a_{3,2} \cdot b_{2,2} \]
\[ c_{3,3} = c_{3,3} + a_{3,3} \cdot b_{3,3} \]

These operations occur when \( k = 2 \).
When $k = 3$, the final results are generated.

### Example: Alternate code execution

$k = 3$:

- $c_{1,1} = c_{1,1} + a_{1,3} \times b_{3,1}$
- $c_{1,2} = c_{1,2} + a_{1,1} \times b_{1,2}$
- $c_{1,3} = c_{1,3} + a_{1,2} \times b_{2,3}$
- $c_{3,1} = c_{3,1} + a_{3,2} \times b_{2,1}$
- $c_{3,2} = c_{3,2} + a_{3,3} \times b_{3,2}$
- $c_{3,3} = c_{3,3} + a_{3,1} \times b_{1,3}$

$c_{2,1} = c_{2,1} + a_{2,1} \times b_{1,1}$
- $c_{2,2} = c_{2,2} + a_{2,2} \times b_{2,2}$
- $c_{2,3} = c_{2,3} + a_{2,3} \times b_{3,3}$
- $c_{3,2} = c_{3,2} + a_{3,3} \times b_{3,2}$
This module introduced pipelining and vector processing, two methods of achieving parallelism in computer design. We examined the application of pipelining both to processing data and to processing instructions. We examined vector processing and the effect of memory interleaving on system performance. The matrix multiplication example implements vector processing using a SIMD architecture.

In the next module we will study computer arithmetic. We will examine both the algorithms to perform computations on numbers in a variety of formats, and the hardware to implement these algorithms.