Pipeline and Vector Processing

Outline

- Introduction
- Arithmetic pipelines
- Instruction pipelines
- Vector processing
- Memory interleaving
- SIMD example
- Summary

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.

Multiple function units

See figure 9.1, p. 300 of the textbook.

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.

Flynn's classification

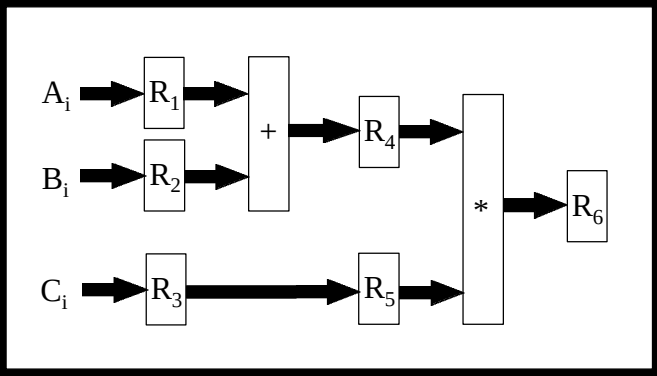
- Single Instruction Single Data (SISD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Single Data (MISD)
- Multiple Instruction Multiple Data (MIMD)

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.

Arithmetic pipelining example

For i = 1 to 1000 DO (A[i] + B[i]) * C[i]



To illustrate arithmetic pipelining, consider a loop which performs the computation (A[i] + B[i]) * 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.

Example: data flow

Cycle	R_1	R_2	R_3	R_4	R_5	R_6
1	A_1	B_1	C_1	---	---	---
2	A_2	B_2	C_2	$A_1 + B_1$	C_1	---
3	A_3	B_3	C_3	$A_2 + B_2$	C_2	X_1
4	A_4	B_4	C_4	$A_3 + B_3$	C_3	X_2
5	A_5	B_5	C_5	$A_4 + B_4$	C_4	X_3

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).

Example: space-time diagram

Time:	1	2	3	4	5	6
Stage						
1	1	2	3	4	5	6
2	---	1	2	3	4	5
3	---	---	1	2	3	4

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

- **The reduction in time using a pipeline rather than a single-stage processor**

$$S_n = (n*t_n)/((k+n-1)*t_p)$$

$$S_\infty = t_n/t_p$$

$$S_{MAX} <= k$$

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, tn 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. tp 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 tn/tp. The maximum speedup is k since, in the best possible case, the processing time tn is partitioned exactly evenly among the k stages of the pipeline, and tp = tn/k. In reality, this does not occur due to the delay of the latches.

Example: speedup

$t_+ = 20$ ns; $t_* = 25$ ns; $t_R = 5$ ns

$t_n = 20 \text{ ns} + 25 \text{ ns} + 5 \text{ ns} = 50 \text{ ns}$

$t_p = 25 \text{ ns} + 5 \text{ ns} = 50 \text{ ns}$

$S_n = (n*t_n)/((k+n-1)*t_p)$

$= (1000*50)/((3+1000-1)*30) = 1.66$

$S_\infty = t_n/t_p = 50/30 = 1.67$

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.

Floating point addition

$X = A * 10^a \qquad Y = B * 10^b$

1. Compare the exponents
2. Align the mantissas
3. Add or subtract the mantissas
4. Normalize the result

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 * 10^a, where A is the mantissa and 0.1 <= A < 1, i.e. A does not have a leading zero. (X=0 is a special case; we assume X <> 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 * 10^2 to .001 * 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.

Floating point pipeline

See figure 9.6, p. 309 of the textbook.

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

$X = .97 * 10^2$ $Y = .81 * 10^1$

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.

Example: floating point addition

$X = .97 * 10^2$ $Y = .81 * 10^1$

Stage 2: Align the mantissas

$Y = .081 * 10^2$

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 * 10^1 as Y = .081 * 10^2. Note that this does not change the value of Y, only its representation.

Example: floating point addition

$X = .97 * 10^2$ $Y = .81 * 10^1 = .081 * 10^2$

Stage 3: Add the mantissas

Mantissa = .97 + .081 = 1.051

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

$X = .97 * 10^2 \quad Y = .81 * 10^1 = .081 * 10^2$

Stage 4: Normalize the result

$$X + Y = 1.051 * 10^2 = .1051 * 10^3$$

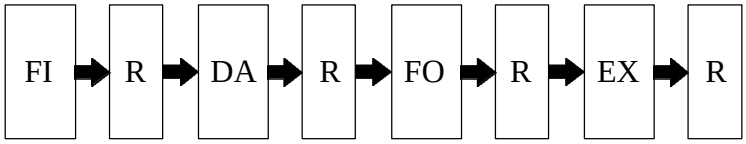
The result of the addition, 1.051 * 10^2, is not in normal form, which requires the mantissa to be at least 0.1 but less than one. In this stage we shift the mantissa one position to the right and increment the exponent by one. This converts 1.051 * 10^2 to .1051 * 10^3, or 97 + 8.1 = 105.1.

Instruction pipelines

1. Fetch instruction (FI)
2. Decode instruction/calculate effective address (DA)
3. Fetch operands (FO)
4. Execute instruction (EX)

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.

Instruction pipeline



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.

Space-time diagram

Time:	1	2	3	4	5	6	7
Instr.							
1	FI	DA	FO	EX	---	---	---
2	---	FI	DA	FO	EX	---	---
3	---	---	FI	DA	FO	EX	---
4	---	---	---	FI	DA	FO	EX

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.

Pipeline conflicts

- Resource conflicts
- Data dependency
- Branch difficulties

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.

Data dependency

1: A = B + C
2: D = A + E

Time:	1	2	3	4	5
Instr.					
1	FI	DA	FO	EX	---
2	---	FI	DA	FO	EX

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.

Resolving data dependency

- Hardware interlocks
- Operand forwarding
- Delayed load

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.

Delayed load

1: A = B + C
2: NOP
3: D = A + E

Time:	1	2	3	4	5	6
Instr.						
1	FI	DA	FO	EX	---	---
2	---	FI	DA	FO	EX	---
3	---	---	FI	DA	FO	EX

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.

Branch difficulties

1: A = B + C
2: JUMP X
3: D = A + E

Time:	1	2	3	4	5	6
Instr.						
1	FI	DA	FO	EX	---	---
2	---	FI	DA	FO	EX	---
3	---	---	FI	DA	FO	EX

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.

Resolving branch difficulties

- Prefetch target instructions
- Branch target buffer
- Loop buffer
- Branch prediction
- Delayed branch

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.

Example: Delayed Branch

Time:	1	2	3	4	5	6	7	8
LOAD	FI	DA	FO	EX	---	---	---	---
INCR	---	FI	DA	FO	EX	---	---	---
ADD	---	---	FI	DA	FO	EX	---	---
JUMP X	---	---	---	FI	DA	FO	EX	---
NOP	---	---	---	---	FI	DA	FO	EX
NOP	---	---	---	---	---	FI	DA	FO
NOP	---	---	---	---	---	---	FI	DA
X: SUB	---	---	---	---	---	---	---	FI

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

Time:	1	2	3	4	5	6	7	8
JUMP X	FI	DA	FO	EX	---	---	---	---
LOAD	---	FI	DA	FO	EX	---	---	---
INCR	---	---	FI	DA	FO	EX	---	---
ADD	---	---	---	FI	DA	FO	EX	---
X: SUB	---	---	---	---	FI	DA	FO	EX

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.)

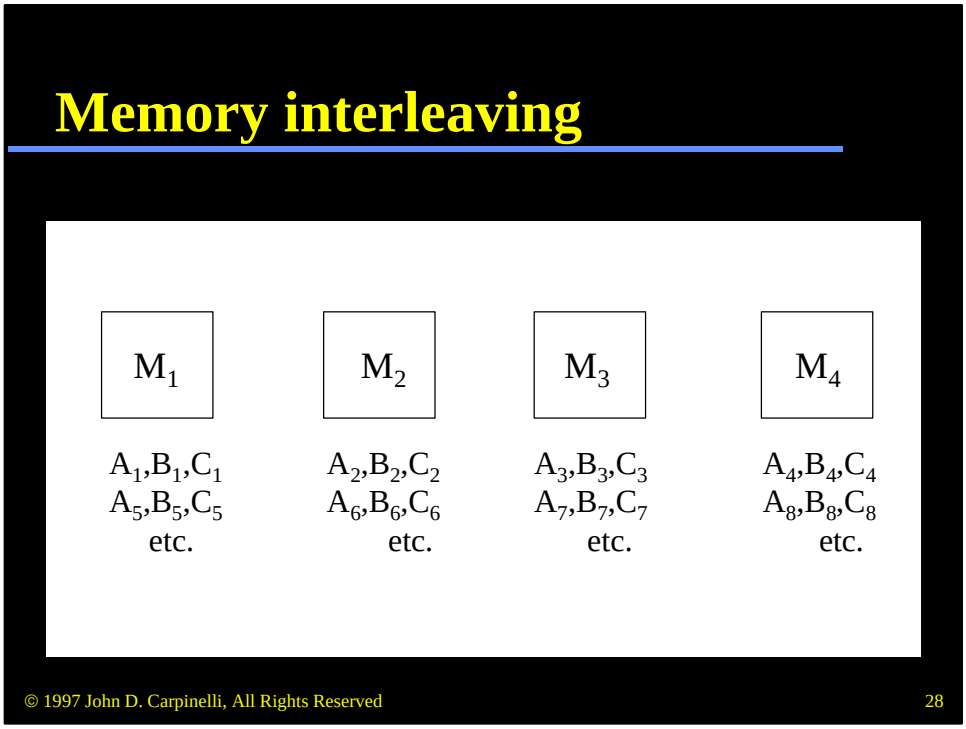
Note that the SUB instruction at location X must still wait until the JUMP instruction has moved X into the program counter before it can be fetched.

Vector processing

For i = 1 to 1000 DO (A[i] + B[i]) * C[i]

- **Execute the code in parallel for different values of i.**

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 sumultaneously.



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.

Example: SIMD matrix multiplication

$$\begin{bmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{bmatrix} * \begin{bmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{bmatrix} = \begin{bmatrix} c11 & c12 & c13 \\ c21 & c22 & c23 \\ c31 & c32 & c33 \end{bmatrix}$$

$$c_{i,j} = \sum_{k=1}^{3} a_{i,k} b_{k,j}$$

Matrix multiplication is a tyical 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.

Example: Sequential code

```
FOR i = 1 TO n
  FOR 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]}
     }
```


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 (Initially C = 0):

$c_{1,1} = c_{1,1} + a_{1,1} * b_{1,1}$ $\quad$ $c_{2,1} = c_{2,1} + a_{2,1} * b_{1,1}$

$c_{1,2} = c_{1,2} + a_{1,1} * b_{1,2}$ $\quad$ $c_{2,2} = c_{2,2} + a_{2,1} * b_{1,2}$

$c_{1,3} = c_{1,3} + a_{1,1} * b_{1,3}$ $\quad$ $c_{2,3} = c_{2,3} + a_{2,1} * b_{1,3}$

$c_{3,1} = c_{3,1} + a_{3,1} * b_{1,1}$ $\quad$ $c_{3,2} = c_{3,2} + a_{3,1} * b_{1,2}$

$c_{3,3} = c_{3,3} + a_{3,1} * b_{1,3}$

Here is the result of the first iteration of the k loop.

Example: Parallel code execution

k = 2:

$c_{1,1} = c_{1,1} + a_{1,2} * b_{2,1}$ $\quad$ $c_{2,1} = c_{2,1} + a_{2,2} * b_{2,1}$

$c_{1,2} = c_{1,2} + a_{1,2} * b_{2,2}$ $\quad$ $c_{2,2} = c_{2,2} + a_{2,2} * b_{2,2}$

$c_{1,3} = c_{1,3} + a_{1,2} * b_{2,3}$ $\quad$ $c_{2,3} = c_{2,3} + a_{2,2} * b_{2,3}$

$c_{3,1} = c_{3,1} + a_{3,2} * b_{2,1}$ $\quad$ $c_{3,2} = c_{3,2} + a_{3,2} * b_{2,2}$

$c_{3,3} = c_{3,3} + a_{3,2} * b_{2,3}$

These operations occur when k = 2.

Example: Parallel code execution

k = 3:

$c_{1,1} = c_{1,1} + a_{1,3} * b_{3,1}$ $\quad$ $c_{2,1} = c_{2,1} + a_{2,3} * b_{3,1}$

$c_{1,2} = c_{1,2} + a_{1,3} * b_{3,2}$ $\quad$ $c_{2,2} = c_{2,2} + a_{2,3} * b_{3,2}$

$c_{1,3} = c_{1,3} + a_{1,3} * b_{3,3}$ $\quad$ $c_{2,3} = c_{2,3} + a_{2,3} * b_{3,3}$

$c_{3,1} = c_{3,1} + a_{3,3} * b_{3,1}$ $\quad$ $c_{3,2} = c_{3,2} + a_{3,3} * b_{3,2}$

$c_{3,3} = c_{3,3} + a_{3,3} * b_{3,3}$

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.

Example: Alternate 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,x * b_x,j}
     }
```

Note: x = (i + j + k) mod n + 1

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.

Example: Alternate code execution

k = 1 (Initially C = 0):

$c_{1,1} = c_{1,1} + a_{1,1} * b_{1,1}$ $\quad$ $c_{2,1} = c_{2,1} + a_{2,2} * b_{2,1}$

$c_{1,2} = c_{1,2} + a_{1,2} * b_{2,2}$ $\quad$ $c_{2,2} = c_{2,2} + a_{2,3} * b_{3,2}$

$c_{1,3} = c_{1,3} + a_{1,3} * b_{3,3}$ $\quad$ $c_{2,3} = c_{2,3} + a_{2,1} * b_{1,3}$

$c_{3,1} = c_{3,1} + a_{3,3} * b_{3,1}$ $\quad$ $c_{3,2} = c_{3,2} + a_{3,1} * b_{1,2}$

$c_{3,3} = c_{3,3} + a_{3,2} * b_{2,3}$

As can be seen here, no two products use the same value of a or b in any two computations.

Example: Alternate code execution

k = 2:

$c_{1,1} = c_{1,1} + a_{1,2} * b_{2,1}$

$c_{2,1} = c_{2,1} + a_{2,3} * b_{3,1}$

$c_{1,2} = c_{1,2} + a_{1,3} * b_{3,2}$

$c_{2,2} = c_{2,2} + a_{2,1} * b_{1,2}$

$c_{1,3} = c_{1,3} + a_{1,1} * b_{1,3}$

$c_{2,3} = c_{2,3} + a_{2,2} * b_{2,3}$

$c_{3,1} = c_{3,1} + a_{3,1} * b_{1,1}$

$c_{3,2} = c_{3,2} + a_{3,2} * b_{2,2}$

$c_{3,3} = c_{3,3} + a_{3,3} * b_{3,3}$

These operations occur when k = 2.

Example: Alternate code execution

k = 3:

$c_{1,1} = c_{1,1} + a_{1,3} * b_{3,1}$

$c_{2,1} = c_{2,1} + a_{2,1} * b_{1,1}$

$c_{1,2} = c_{1,2} + a_{1,1} * b_{1,2}$

$c_{2,2} = c_{2,2} + a_{2,2} * b_{2,2}$

$c_{1,3} = c_{1,3} + a_{1,2} * b_{2,3}$

$c_{2,3} = c_{2,3} + a_{2,3} * b_{3,3}$

$c_{3,1} = c_{3,1} + a_{3,2} * b_{2,1}$

$c_{3,2} = c_{3,2} + a_{3,3} * b_{3,2}$

$c_{3,3} = c_{3,3} + a_{3,1} * b_{1,3}$

When k = 3, the final results are generated.

Summary

- ☑ Arithmetic pipelines
- ☑ Instruction pipelines
- ☑ Vector processing
- ☑ Memory interleaving
- ☑ SIMD example
- ☑ *Next module: Computer Arithmetic*

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.