PIPELINING AND VECTOR PROCESSING

- Parallel Processing
- Pipelining
- Arithmetic Pipeline
- Instruction Pipeline
- RISC Pipeline
- Vector Processing
- Array Processors
PARALLEL PROCESSING

Execution of *Concurrent Events* in the computing process to achieve faster *Computational Speed*.

Levels of Parallel Processing
- Job or Program level
- Task or Procedure level
- Inter-Instruction level
- Intra-Instruction level
**PARALLEL COMPUTERS**

Architectural Classification

* Flynn's classification
  - Based on the multiplicity of *Instruction Streams* and *Data Streams*
  - Instruction Stream
    Sequence of Instructions read from memory
  - Data Stream
    Operations performed on the data in the processor

<table>
<thead>
<tr>
<th>Number of Instruction Streams</th>
<th>Number of Data Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>Single</td>
</tr>
<tr>
<td>Single</td>
<td>SISD</td>
</tr>
<tr>
<td>Multiple</td>
<td>MISD</td>
</tr>
<tr>
<td>Multiple</td>
<td>SIMD</td>
</tr>
<tr>
<td>Multiple</td>
<td>MIMD</td>
</tr>
</tbody>
</table>
COMPUTER ARCHITECTURES FOR PARALLEL PROCESSING

- **Von-Neuman based**
  - SISD
    - Superscalar processors
    - Superpipelined processors
    - VLIW
  - MISD
    - Nonexistence
  - SIMD
    - Array processors
    - Systolic arrays
    - Associative processors
  - MIMD
    - Shared-memory multiprocessors
      - Bus based
      - Crossbar switch based
      - Multistage IN based
      - Message-passing multicomputers
        - Hypercube
        - Mesh
        - Reconfigurable

- **Dataflow**

- **Reduction**
SISD COMPUTER SYSTEMS

![Diagram of SISD Computer System]

Characteristics

- Standard von Neumann machine
- Instructions and data are stored in memory
- One operation at a time

Limitations

- **Von Neumann bottleneck**

  Maximum speed of the system is limited by the *Memory Bandwidth* (bits/sec or bytes/sec)

  - Limitation on *Memory Bandwidth*
  - Memory is shared by CPU and I/O
PERFORMANCE IMPROVEMENTS

- Multiprogramming
- Spooling
- Multifunction processor
- Pipelining
- Exploiting instruction-level parallelism
  - Superscalar
  - Superpipelining
  - VLIW (Very Long Instruction Word)
Characteristics

- There is no computer at present that can be classified as MISD
**SIMD COMPUTER SYSTEMS**

Characteristics

- Only one copy of the program exists
- A single controller executes one instruction at a time
TYPES OF SIMD COMPUTERS

Array Processors

- The control unit broadcasts instructions to all PEs, and all active PEs execute the same instructions
- ILLIAC IV, GF-11, Connection Machine, DAP, MPP

Systolic Arrays

- Regular arrangement of a large number of very simple processors constructed on VLSI circuits
- CMU Warp, Purdue CHiP

Associative Processors

- Content addressing
- Data transformation operations over many sets of arguments with a single instruction
- STARAN, PEPE
MIMD COMPUTER SYSTEMS

Characteristics
- Multiple processing units
- Execution of multiple instructions on multiple data

Types of MIMD computer systems
- Shared memory multiprocessors
- Message-passing multicomputers
SHARED MEMORY MULTIPROCESSORS

Characteristics
All processors have equally direct access to one large memory address space

Example systems
- Bus and cache-based systems
  - Sequent Balance, Encore Multimax
- Multistage IN-based systems
  - Ultracomputer, Butterfly, RP3, HEP
- Crossbar switch-based systems
  - C.mmp, Alliant FX/8

Limitations
- Memory access latency
- Hot spot problem
MESSAGE-PASSING MULTICOMPUTER

Message-Passing Network

Point-to-point connections

Characteristics

- Interconnected computers
- Each processor has its own memory, and communicate via message-passing

Example systems

- Tree structure: Teradata, DADO
- Mesh-connected: Rediflow, Series 2010, J-Machine
- Hypercube: Cosmic Cube, iPSC, NCUBE, FPS T Series, Mark III

Limitations

- Communication overhead
- Hard to programming
Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a partial dedicated segment that operates concurrently with all other segments.

For $i = 1, 2, 3, \ldots, 7$:

1. Load $A_i$ and $B_i$
2. Multiply $R1 * R2$ and load $C_i$
3. Add $R3 + R4$

Diagram:
- Segment 1: $A_i \leftarrow R1$
- Segment 2: $B_i \leftarrow R2$
- Multiplier: $R3 \leftarrow R1 * R2, R4 \leftarrow C_i$
- Segment 3: $R5 \leftarrow R3 + R4$

Load $A_i$ and $B_i$, Multiply and load $C_i$, Add
## OPERATIONS IN EACH PIPELINE STAGE

<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>A1</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A2</td>
<td>B2</td>
<td>A1 * B1</td>
</tr>
<tr>
<td>3</td>
<td>A3</td>
<td>B3</td>
<td>A2 * B2</td>
</tr>
<tr>
<td>4</td>
<td>A4</td>
<td>B4</td>
<td>A3 * B3</td>
</tr>
<tr>
<td>5</td>
<td>A5</td>
<td>B5</td>
<td>A4 * B4</td>
</tr>
<tr>
<td>6</td>
<td>A6</td>
<td>B6</td>
<td>A5 * B5</td>
</tr>
<tr>
<td>7</td>
<td>A7</td>
<td>B7</td>
<td>A6 * B6</td>
</tr>
<tr>
<td>8</td>
<td>A7</td>
<td>B7</td>
<td>A7 * B7</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GENERAL PIPELINE

General Structure of a 4-Segment Pipeline

Clock cycles
Segment 1 2 3 4 5 6 7 8 9
1 T1 T2 T3 T4 T5 T6
2 T1 T2 T3 T4 T5 T6
3 T1 T2 T3 T4 T5 T6
4 T1 T2 T3 T4 T5 T6

Space-Time Diagram

Computer Organization
Pipelining and Vector Processing

PIPELINE SPEEDUP

n:  Number of tasks to be performed

Conventional Machine (Non-Pipelined)
- \( t_n \): Clock cycle
- \( \tau_1 \): Time required to complete the n tasks
  \[ \tau_1 = n \times t_n \]

Pipelined Machine (k stages)
- \( t_p \): Clock cycle (time to complete each suboperation)
- \( \tau_k \): Time required to complete the n tasks
  \[ \tau_k = (k + n - 1) \times t_p \]

Speedup
- \( S_k \): Speedup
  \[ S_k = \frac{n \times t_n}{(k + n - 1) \times t_p} \]

\[
\lim_{n \to \infty} S_k = \frac{t_n}{t_p} \quad (= k, \text{ if } t_n = k \times t_p)
\]
Example
- 4-stage pipeline
- suboperation in each stage; \( t_p = 20\text{nS} \)
- 100 tasks to be executed
- 1 task in non-pipelined system; \( 20 \times 4 = 80\text{nS} \)

Pipelined System
\[(k + n - 1) \times t_p = (4 + 99) \times 20 = 2060\text{nS} \]

Non-Pipelined System
\[n \times k \times t_p = 100 \times 80 = 8000\text{nS} \]

Speedup
\[S_k = 8000 / 2060 = 3.88 \]

4-Stage Pipeline is basically identical to the system with 4 identical function units
Floating-point adder

\[ X = A \times 2^a \]
\[ Y = B \times 2^b \]

1. Compare the exponents
2. Align the mantissa
3. Add/sub the mantissa
4. Normalize the result

Exponents

\[ a \downarrow \]
\[ b \downarrow \]

Mantissas

\[ A \downarrow \]
\[ B \downarrow \]

Segment 1:

Compare exponents
by subtraction

Segment 2:

Choose exponent

Segment 3:

Add or subtract mantissas

Segment 4:

Adjust exponent

Normalize result
**4-STAGE FLOATING POINT ADDER**

A = a \times 2^p

B = b \times 2^q

\[ C = A + B = c \times 2^r = d \times 2^s \]

\( r = \max(p, q) \), \( 0.5 \leq d < 1 \)
INSTRUCTION CYCLE

Six Phases* in an Instruction Cycle

1. Fetch an instruction from memory
2. Decode the instruction
3. Calculate the effective address of the operand
4. Fetch the operands from memory
5. Execute the operation
6. Store the result in the proper place

* Some instructions skip some phases
* Effective address calculation can be done in the part of the decoding phase
* Storage of the operation result into a register is done automatically in the execution phase

=> 4-Stage Pipeline

1. FI: Fetch an instruction from memory
2. DA: Decode the instruction and calculate the effective address of the operand
3. FO: Fetch the operand
4. EX: Execute the operation
Execution of Three Instructions in a 4-Stage Pipeline

Conventional

Pipelined
INSTRUCTION EXECUTION IN A 4-Stage Pipeline

Segment 1: Fetch instruction from memory

Segment 2: Decode instruction and calculate effective address

Branch?:
- yes
- no

Segment 3: Fetch operand from memory

Segment 4: Execute instruction

Interrupt handling:
- yes
- no

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

Instruction Pipeline:
- FI
- DA
- FO
- EX

Update PC
Empty pipe

Interrupt handling:
- yes
- no

Instruction Pipeline:
- FI
- DA
- FO
- EX

Update PC
Empty pipe
MAJOR HAZARDS IN PIPELINED EXECUTION

Structural hazards (Resource Conflicts)

Hardware Resources required by the instructions in simultaneous overlapped execution cannot be met.

Data hazards (Data Dependency Conflicts)

An instruction scheduled to be executed in the pipeline requires the result of a previous instruction, which is not yet available.

Control hazards

Branches and other instructions that change the PC make the fetch of the next instruction to be delayed.

Hazards in pipelines may make it necessary to **stall** the pipeline.

Pipeline Interlock:
Detect Hazards Stall until it is cleared.

R1 <- B + C
R1 <- R1 + 1

Data dependency:

Branch address dependency:

ADD DA B,C +
INC DA bubble R1 +1

JMP ID PC + PC

IF ID OF OE OS
Structural Hazards

Occur when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute.

Example: With one memory-port, a data and an instruction fetch cannot be initiated in the same clock.

The Pipeline is stalled for a structural hazard:

<- Two Loads with one port memory

-> Two-port memory will serve without stall.
DATA HAZARDS

Data Hazards

Occurs when the execution of an instruction depends on the results of a previous instruction

ADD R1, R2, R3
SUB R4, R1, R5

Data hazard can be dealt with either hardware techniques or software technique

Hardware Technique

Interlock
- hardware detects the data dependencies and delays the scheduling of the dependent instruction by stalling enough clock cycles

Forwarding (bypassing, short-circuiting)
- Accomplished by a data path that routes a value from a source (usually an ALU) to a user, bypassing a designated register. This allows the value to be produced to be used at an earlier stage in the pipeline than would otherwise be possible

Software Technique

Instruction Scheduling (compiler) for delayed load
Example:

ADD  R1, R2, R3
SUB  R4, R1, R5

3-stage Pipeline

I: Instruction Fetch
A: Decode, Read Registers, ALU Operations
E: Write the result to the destination register

ADD  I  A  E  
SUB  I  A  E  Without Bypassing
SUB  I  A  E  With Bypassing
## INSTRUCTION SCHEDULING

\[
\begin{align*}
  a &= b + c; \\
  d &= e - f;
\end{align*}
\]

<table>
<thead>
<tr>
<th>Unscheduled code</th>
<th>Scheduled Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{LW} \ Rb, b)</td>
<td>(\text{LW} \ Rb, b)</td>
</tr>
<tr>
<td>(\text{LW} \ Rc, c)</td>
<td>(\text{LW} \ Rc, c)</td>
</tr>
<tr>
<td>(\rightarrow \text{ADD} \ Ra, Rb, Rc)</td>
<td>(\text{ADD} \ Ra, Rb, Rc)</td>
</tr>
<tr>
<td>(\rightarrow \text{SW} \ a, Ra)</td>
<td>(\text{SW} \ a, Ra)</td>
</tr>
<tr>
<td>(\text{LW} \ Re, e)</td>
<td>(\text{LW} \ Re, e)</td>
</tr>
<tr>
<td>(\text{LW} \ Rf, f)</td>
<td>(\text{SW} \ a, Ra)</td>
</tr>
<tr>
<td>(\rightarrow \text{SUB} \ Rd, Re, Rf)</td>
<td>(\text{SUB} \ Rd, Re, Rf)</td>
</tr>
<tr>
<td>(\rightarrow \text{SW} \ d, Rd)</td>
<td>(\rightarrow \text{SW} \ d, Rd)</td>
</tr>
</tbody>
</table>

**Delayed Load**

A load requiring that the following instruction not use its result
**CONTROL HAZARDS**

**Branch Instructions**

- Branch target address is not known until the branch instruction is completed

```
Branch Instruction
FI  DA  FO  EX
```

```
Next Instruction
FI  DA  FO  EX
```

- Stall -> waste of cycle times

**Dealing with Control Hazards**

* Prefetch Target Instruction
* Branch Target Buffer
* Loop Buffer
* Branch Prediction
* Delayed Branch
CONTROL HAZARDS

Prefetch Target Instruction
- Fetch instructions in both streams, branch not taken and branch taken
- Both are saved until branch branch is executed. Then, select the right instruction stream and discard the wrong stream

Branch Target Buffer (BTB; Associative Memory)
- Entry: Addr of previously executed branches; Target instruction and the next few instructions
- When fetching an instruction, search BTB.
- If found, fetch the instruction stream in BTB;
- If not, new stream is fetched and update BTB

Loop Buffer (High Speed Register file)
- Storage of entire loop that allows to execute a loop without accessing memory

Branch Prediction
- Guessing the branch condition, and fetch an instruction stream based on the guess. Correct guess eliminates the branch penalty

Delayed Branch
- Compiler detects the branch and rearranges the instruction sequence by inserting useful instructions that keep the pipeline busy in the presence of a branch instruction
RISC PIPELINE

RISC
- Machine with a very fast clock cycle that executes at the rate of one instruction per cycle
<- Simple Instruction Set
  Fixed Length Instruction Format
  Register-to-Register Operations

Instruction Cycles of Three-Stage Instruction Pipeline

Data Manipulation Instructions
I: Instruction Fetch
A: Decode, Read Registers, ALU Operations
E: Write a Register

Load and Store Instructions
I: Instruction Fetch
A: Decode, Evaluate Effective Address
E: Register-to-Memory or Memory-to-Register

Program Control Instructions
I: Instruction Fetch
A: Decode, Evaluate Branch Address
E: Write Register(PC)
### DELAYED LOAD

**LOAD:** R1 ← M[address 1]  
**LOAD:** R2 ← M[address 2]  
**ADD:** R3 ← R1 + R2  
**STORE:** M[address 3] ← R3

**Three-segment pipeline timing**

**Pipeline timing with data conflict**

<table>
<thead>
<tr>
<th>clock cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load R1</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load R2</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R1+R2</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store R3</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pipeline timing with delayed load**

<table>
<thead>
<tr>
<th>clock cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load R1</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load R2</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R1+R2</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store R3</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The data dependency is taken care by the compiler rather than the hardware.
Compiler analyzes the instructions before and after the branch and rearranges the program sequence by inserting useful instructions in the delay steps.

Using no-operation instructions

<table>
<thead>
<tr>
<th>Clock cycles:</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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Load</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Increment</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Add</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Subtract</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Branch to X</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. NOP</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. NOP</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. Instr. in X</td>
<td></td>
<td></td>
<td></td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rearranging the instructions

<table>
<thead>
<tr>
<th>Clock cycles:</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>1. Load</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Increment</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Branch to X</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Add</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Subtract</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Instr. in X</td>
<td>I</td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
VECTOR PROCESSING

Vector Processing Applications
- Problems that can be efficiently formulated in terms of vectors
  - Long-range weather forecasting
  - Petroleum explorations
  - Seismic data analysis
  - Medical diagnosis
  - Aerodynamics and space flight simulations
  - Artificial intelligence and expert systems
  - Mapping the human genome
  - Image processing

Vector Processor (computer)
- Ability to process vectors, and related data structures such as matrices and multi-dimensional arrays, much faster than conventional computers

Vector Processors may also be pipelined
## VECTOR PROGRAMMING

**DO 20 I = 1, 100**

**20** \( C(I) = B(I) + A(I) \)

### Conventional computer

- Initialize \( I = 0 \)
- **DO 20**
  - Read \( A(I) \)
  - Read \( B(I) \)
  - Store \( C(I) = A(I) + B(I) \)
  - Increment \( I = I + 1 \)
  - If \( I \leq 100 \) goto 20

### Vector computer

- \( C(1:100) = A(1:100) + B(1:100) \)
## VECTOR INSTRUCTIONS

<table>
<thead>
<tr>
<th>Type</th>
<th>Mnemonic</th>
<th>Description (I = 1, ..., n)</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>VSQR</td>
<td>Vector square root</td>
<td>B(I) * SQR(A(I))</td>
</tr>
<tr>
<td></td>
<td>VSIN</td>
<td>Vector sine</td>
<td>B(I) * sin(A(I))</td>
</tr>
<tr>
<td></td>
<td>VCOM</td>
<td>Vector complement</td>
<td>A(I) * \overline{A(I)}</td>
</tr>
<tr>
<td>f2</td>
<td>VSUM</td>
<td>Vector summation</td>
<td>S * Σ A(I)</td>
</tr>
<tr>
<td></td>
<td>VMAX</td>
<td>Vector maximum</td>
<td>S * max{A(I)}</td>
</tr>
<tr>
<td>f3</td>
<td>VADD</td>
<td>Vector add</td>
<td>C(I) * A(I) + B(I)</td>
</tr>
<tr>
<td></td>
<td>VMPY</td>
<td>Vector multiply</td>
<td>C(I) * A(I) * B(I)</td>
</tr>
<tr>
<td></td>
<td>VAND</td>
<td>Vector AND</td>
<td>C(I) * A(I) . B(I)</td>
</tr>
<tr>
<td></td>
<td>VLAR</td>
<td>Vector larger</td>
<td>C(I) * max(A(I),B(I))</td>
</tr>
<tr>
<td></td>
<td>VTGE</td>
<td>Vector test &gt;</td>
<td>C(I) * 0 if A(I) &lt; B(I)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C(I) * 1 if A(I) &gt; B(I)</td>
</tr>
<tr>
<td>f4</td>
<td>SADD</td>
<td>Vector-scalar add</td>
<td>B(I) * S + A(I)</td>
</tr>
<tr>
<td></td>
<td>SDIV</td>
<td>Vector-scalar divide</td>
<td>B(I) * A(I) / S</td>
</tr>
</tbody>
</table>
### Vector Instruction Format

<table>
<thead>
<tr>
<th>Operation code</th>
<th>Base address source 1</th>
<th>Base address source 2</th>
<th>Base address destination</th>
<th>Vector length</th>
</tr>
</thead>
</table>

### Pipeline for Inner Product

- Source A
- Source B
- Multiplier pipeline
- Adder pipeline
Multiple Module Memory

Address Interleaving

Different sets of addresses are assigned to different memory modules