Pipelining and Vector Processing

Chapter 8
S. Dandamudi
Outline

- Basic concepts
- Handling resource conflicts
- Data hazards
- Handling branches
- Performance enhancements
- Example implementations  
  * Pentium  
  * PowerPC  
  * SPARC  
  * MIPS

- Vector processors  
  * Architecture  
  * Advantages  
  * Cray X-MP  
  * Vector length  
  * Vector stride  
  * Chaining

- Performance  
  * Pipeline  
  * Vector processing
Basic Concepts

• Pipelining allows overlapped execution to improve throughput
  * Introduction given in Chapter 1
  * Pipelining can be applied to various functions
    » Instruction pipeline
      – Five stages
      – Fetch, decode, operand fetch, execute, write-back
    » FP add pipeline
      – Unpack: into three fields
      – Align: binary point
      – Add: aligned mantissas
      – Normalize: pack three fields after normalization
### Basic Concepts (cont’d)

#### Execution cycle

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>OF</th>
<th>IE</th>
<th>EB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>Instruction decode</td>
<td>Operand fetch</td>
<td>Instruction execution</td>
<td>Result write back</td>
</tr>
</tbody>
</table>

**Instruction execution phase**

(a) Instruction pipeline stages

- Unpack
- Align
- Add
- Normalize

(b) Floating-point add pipeline stages
Basic Concepts (cont’d)

(a) Serial execution

(b) Pipelined execution

Serial execution: 20 cycles

Pipelined execution: 8 cycles
Basic Concepts (cont’d)

• Pipelining requires buffers
  * Each buffer holds a single value
  * Uses just-in-time principle
    » Any delay in one stage affects the entire pipeline flow
  * Ideal scenario: equal work for each stage
    » Sometimes it is not possible
    » Slowest stage determines the flow rate in the entire pipeline
Basic Concepts (cont’d)

• Some reasons for unequal work stages
  * A complex step cannot be subdivided conveniently
  * An operation takes variable amount of time to execute
    » EX: Operand fetch time depends on where the operands are located
      – Registers
      – Cache
      – Memory
  * Complexity of operation depends on the type of operation
    » Add: may take one cycle
    » Multiply: may take several cycles
Basic Concepts (cont’d)

- Operand fetch of I2 takes three cycles
  * Pipeline *stalls* for two cycles
    » Caused by hazards
  * Pipeline stalls reduce overall throughput

<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>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>IF</td>
<td>ID</td>
<td>OF</td>
<td>IE</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td>IF</td>
<td>ID</td>
<td>OF</td>
<td></td>
<td>IE</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td>OF</td>
<td>IE</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td>IF</td>
<td></td>
<td>ID</td>
<td>OF</td>
<td>IE</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Basic Concepts (cont’d)

• Three types of hazards
  * Resource hazards
    » Occurs when two or more instructions use the same resource
    » Also called *structural hazards*
  * Data hazards
    » Caused by data dependencies between instructions
      – Example: Result produced by I1 is read by I2
  * Control hazards
    » Default: sequential execution suits pipelining
    » Altering control flow (e.g., branching) causes problems
      – Introduce control dependencies
Handling Resource Conflicts

• Example

  * Conflict for memory in clock cycle 3
    » I1 fetches operand
    » I3 delays its instruction fetch from the same memory

<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>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>IF</td>
<td>ID</td>
<td>OF</td>
<td>IE</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td>IF</td>
<td>ID</td>
<td>OF</td>
<td>IE</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td></td>
<td>Idle</td>
<td>IF</td>
<td>ID</td>
<td>OF</td>
<td>IE</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td>IF</td>
<td>ID</td>
<td>OF</td>
<td>IE</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Handling Resource Conflicts (cont’d)

- Minimizing the impact of resource conflicts
  * Increase available resources
  * Prefetch
    » Relaxes just-in-time principle
    » Example: Instruction queue
Data Hazards

• Example
  
  \[ \text{I1: add R2, R3, R4} \quad */ R2 = R3 + R4 */ \]
  
  \[ \text{I2: sub R5, R6, R2} \quad */ R5 = R6 - R2 */ \]

• Introduces data dependency between I1 and I2
Data Hazards (cont’d)

• Three types of data dependencies require attention
  * Read-After-Write (RAW)
    » One instruction writes that is later read by the other instruction
  * Write-After-Read (WAR)
    » One instruction reads from register/memory that is later written by the other instruction
  * Write-After-Write (WAW)
    » One instruction writes into register/memory that is later written by the other instruction

* Read-After-Read (RAR)
  » No conflict
Data Hazards (cont’d)

- Data dependencies have two implications
  - Correctness issue
    » Detect dependency and stall
      - We have to stall the SUB instruction
  - Efficiency issue
    » Try to minimize pipeline stalls

- Two techniques to handle data dependencies
  - Register interlocking
    » Also called *bypassing*
  - Register forwarding
    » General technique
Data Hazards (cont’d)

- Register interlocking
  - Provide output result as soon as possible

- An Example
  - Forward 1 scheme
    » Output of I1 is given to I2 as we write the result into destination register of I1
    » Reduces pipeline stall by one cycle
  - Forward 2 scheme
    » Output of I1 is given to I2 during the IE stage of I1
    » Reduces pipeline stall by two cycles
Data Hazards (cont’d)

(a) Forward scheme 1

(b) Forward scheme 2
Data Hazards (cont’d)

- Implementation of forwarding in hardware
  - Forward 1 scheme
    » Result is given as input from the bus
      – Not from A
  - Forward 2 scheme
    » Result is given as input from the ALU output
Data Hazards (cont’d)

• Register interlocking
  * Associate a bit with each register
    » Indicates whether the contents are correct
      – 0 : contents can be used
      – 1 : do not use contents
  * Instructions lock the register when using
  * Example
    » Intel Itanium uses a similar bit
      – Called NaT (Not-a-Thing)
      – Uses this bit to support speculative execution
      – Discussed in Chapter 14
Data Hazards (cont’d)

- Example

  \textbf{I1:} \texttt{add} \ R2,R3,R4 \ /* \ R2 = R3 + R4 */
  \textbf{I2:} \texttt{sub} \ R5,R6,R2 \ /* \ R5 = R6 – R2 */

- \textbf{I1} locks \texttt{R2} for clock cycles 3, 4, 5
Data Hazards (cont’d)

- Register forwarding vs. Interlocking
  - Forwarding works only when the required values are in the pipeline
  - Interlocking can handle data dependencies of a general nature
  - Example
    ```
    load   R3,count ; R3 = count
    add    R1,R2,R3 ; R1 = R2 + R3
    ```
    » add cannot use R3 value until load has placed the count
    » Register forwarding is not useful in this scenario
Handling Branches

- Braches alter control flow
  - Require special attention in pipelining
  - Need to throw away some instructions in the pipeline
    » Depends on when we know the branch is taken
    » First example (next slide)
      - Discards three instructions I2, I3 and I4
    » Pipeline wastes three clock cycles
      - Called \textit{branch penalty}
  - Reducing branch penalty
    » Determine branch decision early
      - Next example: penalty of one clock cycle
Handling Branches (cont’d)

(a) Branch decision is known during the IE stage

(b) Branch decision is known during the ID stage
Handling Branches (cont’d)

• Delayed branch execution
  * Effectively reduces the branch penalty
  * We always fetch the instruction following the branch
    » Why throw it away?
    » Place a useful instruction to execute
    » This is called delay slot

```plaintext
add     R2,R3,R4
branch  target
sub     R5,R6,R7
. . .
```
Branch Prediction

• Three prediction strategies
  * Fixed
    » Prediction is fixed
      – Example: `branch-never-taken`
        ↳ Not proper for loop structures
  * Static
    » Strategy depends on the branch type
      – Conditional branch: always not taken
      – Loop: always taken
  * Dynamic
    » Takes run-time history to make more accurate predictions
Branch Prediction (cont’d)

- **Static prediction**
  - Improves prediction accuracy over Fixed

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Instruction Distribution (%)</th>
<th>Prediction: Branch taken?</th>
<th>Correct prediction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconditional branch</td>
<td>70*0.4 = 28</td>
<td>Yes</td>
<td>28</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>70*0.6 = 42</td>
<td>No</td>
<td>42*0.6 = 25.2</td>
</tr>
<tr>
<td>Loop</td>
<td>10</td>
<td>Yes</td>
<td>10*0.9 = 9</td>
</tr>
<tr>
<td>Call/return</td>
<td>20</td>
<td>Yes</td>
<td>20</td>
</tr>
</tbody>
</table>

Overall prediction accuracy = 82.2%
Branch Prediction (cont’d)

• Dynamic branch prediction
  * Uses runtime history
    » Takes the past \( n \) branch executions of the branch type and makes the prediction
  * Simple strategy
    » Prediction of the next branch is the majority of the previous \( n \) branch executions
    » Example: \( n = 3 \)
      – If two or more of the last three branches were taken, the prediction is “branch taken”
    » Depending on the type of mix, we get more than 90% prediction accuracy
Branch Prediction (cont’d)

- Impact of past $n$ branches on prediction accuracy

<table>
<thead>
<tr>
<th>$n$</th>
<th>Compiler</th>
<th>Business</th>
<th>Scientific</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64.1</td>
<td>64.4</td>
<td>70.4</td>
</tr>
<tr>
<td>1</td>
<td>91.9</td>
<td>95.2</td>
<td>86.6</td>
</tr>
<tr>
<td>2</td>
<td>93.3</td>
<td>96.5</td>
<td>90.8</td>
</tr>
<tr>
<td>3</td>
<td>93.7</td>
<td>96.6</td>
<td>91.0</td>
</tr>
<tr>
<td>4</td>
<td>94.5</td>
<td>96.8</td>
<td>91.8</td>
</tr>
<tr>
<td>5</td>
<td>94.7</td>
<td>97.0</td>
<td>92.0</td>
</tr>
</tbody>
</table>
Branch Prediction (cont’d)

No branch

00
Predict no branch

Branch

01
Predict no branch

No branch

10
Predict branch

Branch

11
Predict branch

No branch

Branch
Branch Prediction (cont’d)

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Branch instruction address</th>
<th>Prediction bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Branch instruction address</th>
<th>Target address</th>
<th>Prediction bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) vs (b)
Performance Enhancements

• Several techniques to improve performance of a pipelined system
  * Superscalar
    » Replicates the pipeline hardware
  * Superpipelined
    » Increases the pipeline depth
  * Very long instruction word (VLIW)
    » Encodes multiple operations into a long instruction word
    » Hardware schedules these instructions on multiple functional units
      – No run-time analysis
Performance Enhancements

• Superscalar
  * Dual pipeline design
    » Instruction fetch unit gets two instructions per cycle
Performance Enhancements (cont’d)

• Dual pipeline design assumes that instruction execution takes the same time
  * In practice, instruction execution takes variable amount of time
    » Depends on the instruction
  * Provide multiple execution units
    » Linked to a single pipeline
    » Example (next slide)
      – Two integer units
      – Two FP units

• These designs are called superscalar designs
Performance Enhancements (cont’d)
Performance Enhancements (cont’d)

- Superpipelined processors
  - Increases pipeline depth
    - Ex: Divide each processor cycle into two or more subcycles
  - Example: MIPS R40000
    - Eight-stage instruction pipeline
    - Each stage takes half the master clock cycle

**IF1 & IF2**: instruction fetch, first half & second half

**RF** : decode/fetch operands

**EX** : execute

**DF1 & DF2**: data fetch (load/store): first half and second half

**TC** : load/store check

**WB** : write back
### Performance Enhancements (cont’d)

**Diagram (a): Pipelined Execution**

- Clock cycle 1
- **I1**: IF, ID, OF, IE, WB
- **I2**: IF, ID, OF, IE, WB
- **I3**: IF, ID, OF, IE, WB
- **I4**: IF, ID, OF, IE, WB
- **I5**: IF, ID, OF, IE, WB

**Diagram (b): Superpipelined Execution**

- Clock cycle 1
- **I1**: IF1, IF2, ID1, ID2, OF1, OF2, IE1, IE2, WB1, WB2
- **I2**: IF1, IF2, ID1, ID2, OF1, OF2, IE1, IE2, WB1, WB2
- **I3**: IF1, IF2, ID1, ID2, OF1, OF2, IE1, IE2, WB1, WB2
- **I4**: IF1, IF2, ID1, ID2, OF1, OF2, IE1, IE2, WB1, WB2
- **I5**: IF1, IF2, ID1, ID2, OF1, OF2, IE1, IE2, WB1, WB2

---

Performance Enhancements (cont’d)

- Very long instruction word (VLIW)
  * With multiple resources, instruction scheduling is important to keep these units busy
  * In most processors, instruction scheduling is done at run-time by looking at instructions in the instructions queue
    » VLIW architectures move the job of instruction scheduling from run-time to compile-time
      – Implies moving from hardware to software
      – Implies moving from online to offline analysis
        ➔ More complex analysis can be done
        ➔ Results in simpler hardware
Performance Enhancements (cont’d)

• Out-of-order execution

\[
\begin{align*}
\text{add} & \quad R1, R2, R3 & ; R1 = R2 + R3 \\
\text{sub} & \quad R5, R6, R7 & ; R5 = R6 - R7 \\
\text{and} & \quad R4, R1, R5 & ; R4 = R1 \text{ AND } R5 \\
\text{xor} & \quad R9, R9, R9 & ; R9 = R9 \text{ XOR } R9 \\
\end{align*}
\]

* Out-of-order execution allows executing XOR before AND

  » Cycle 1: \texttt{add, sub, xor}
  » Cycle 2: \texttt{and}

* More on this in Chapter 14
Performance Enhancements (cont’d)

• Each VLIW instruction consists of several primitive operations that can be executed in parallel
  * Each word can be tens of bytes wide
  * Multiflow TRACE system:
    » Uses 256-bit instruction words
    » Packs 7 different operations
    » A more powerful TRACE system
      – Uses 1024-bit instruction words
      – Packs as many as 28 operations
  * Itanium uses 128-bit instruction bundles
    » Each consists of three 41-bit instructions
Example Implementations

- We look at instruction pipeline details of four processors
  * Cover both RISC and CISC
  * CISC
    » Pentium
  * RISC
    » PowerPC
    » SPARC
    » MIPS
Pentium Pipeline

• Pentium
  * Uses dual pipeline design to achieve superscalar execution
    » U-pipe
      – Main pipeline
      – Can execute any Pentium instruction
    » V-pipe
      – Can execute only simple instructions
  * Floating-point pipeline
  * Uses the dynamic branch prediction strategy
Pentium Pipeline (cont’d)

- System bus
- Instruction cache
- Branch prediction unit
- Instruction prefetch buffers
- Integer ALU (V-pipe)
- Integer ALU (U-pipe)
- Integer register file
- Data cache
- Floating-point unit
- Floating-point register file
- FP adder
- FP multiplier
- FP divider
Pentium Pipeline (cont’d)

• Algorithm used to schedule the U- and V-pipes
  * Decode two consecutive instructions I1 and I2
    IF (I1 and I2 are simple instructions) AND
      (I1 is not a branch instruction) AND
      (destination of I1 ≠ source of I2) AND
      (destination of I1 ≠ destination of I2)
    THEN
      Issue I1 to U-pipe and I2 to V-pipe
    ELSE
      Issue I1 to U-pipe
Pentium Pipeline (cont’d)

- Integer pipeline
  * 5-stages
- FP pipeline
  * 8-stages
  * First 3 stages are common

(a) Integer pipeline

(b) Floating-point pipeline
Pentium Pipeline (cont’d)

• Integer pipeline
  * Prefetch (PF)
    » Prefetches instructions and stores in the instruction buffer
  * First decode (D1)
    » Decodes instructions and generates
      – Single control word (for simple operations)
        ➔ Can be executed directly
      – Sequence of control words (for complex operations)
        ➔ Generated by a microprogrammed control unit
  * Second decode (D2)
    » Control words generated in D1 are decoded
    » Generates necessary operand addresses
Pentium Pipeline (cont’d)

* Execute (E)
  » Depends on the type of instruction
    – Accesses either operands from the data cache, or
    – Executes instructions in the ALU or other functional units
  » For register operands
    – Operation is performed during E stage and results are written back to registers
  » For memory operands
    – D2 calculates the operand address
    – E stage fetches the operands
    – Another E stage is added to execute in case of cache hit

* Write back (WB)
  » Writes the result back
Pentium Pipeline (cont’d)

• 8-stage FP Pipeline
  * First three stages are the same as in the integer pipeline
  * Operand fetch (OF)
    » Fetches necessary operands from data cache and FP registers
  * First execute (X1)
    » Initial operation is done
    » If data fetched from cache, they are written to FP registers
Pentium Pipeline (cont’d)

* Second execute (X2)
  » Continues FP operation initiated in X1

* Write float (WF)
  » Completes the FP operation
  » Writes the result to FP register file

* Error reporting (ER)
  » Used for error detection and reporting
  » Additional processing may be required to complete execution
PowerPC Pipeline

- PowerPC 604 processor
  - 32 general-purpose registers (GPRs)
  - 32 floating-point registers (FPRs)
  - Three basic execution units
    - Integer
    - Floating-point
    - Load/store
  - A branch processing unit
  - A completion unit
  - Superscalar
    - Issues up to 4 instructions/clock
PowerPC Pipeline (cont’d)
PowerPC Pipeline (cont’d)

• Integer unit
  * Two single-cycle units (SCIU)
    » Execute most integer instructions
    » Take only one cycle to execute
  * One multicycle unit (MCIU)
    » Executes multiplication and division
    » Multiplication of two 32-bit integers takes 4 cycles
    » Division takes 20 cycles

• Floating-point unit (FPU)
  * Handles both single- and double precision FP operations
PowerPC Pipeline (cont’d)
PowerPC Pipeline (cont’d)

- Load/store unit (LSU)
  - Single-cycle, pipelined access to cache
  - Dedicated hardware to perform effective address calculations
  - Performs alignment and precision conversion for FP numbers
  - Performs alignment and sign-extension for integers
  - Uses
    » a 4-entry load miss buffer
    » 6-entry store buffer
PowerPC Pipeline (cont’d)

- **Branch processing unit (BPU)**
  - Uses dynamic branch prediction
  - Maintains a 512-entry branch history table with two prediction bits
  - Keeps a 64-entry branch target address cache

- **Instruction pipeline**
  - 6-stage
  - Maintains 8-entry instruction buffer between the fetch and dispatch units
    - 4-entry decode buffer
    - 4-entry dispatch buffer
PowerPC Pipeline (cont’d)

• Fetch (IF)
  * Instruction fetch

• Decode (ID)
  * Performs instruction decode
  * Moves instructions from decode buffer to dispatch buffer as space becomes available

• Dispatch (DS)
  * Determines which instructions can be scheduled
  * Also fetches operands from registers
PowerPC Pipeline (cont’d)

• **Execute (E)**
  * Time in the execution stage depends on the operation
  * Up to 7 instructions can be in execution

• **Complete (C)**
  * Responsible for correct instruction order of execution

• **Write back (WB)**
  * Writes back data from the rename buffers
SPARC Processor

- UltraSPARC
  - Superscalar
    - Executes up to 4 instructions/cycle
  - Implements 64-bit SPARC-V9 architecture
- Prefetch and dispatch unit (PDU)
  - Performs standard prefetch and dispatch functions
  - Instruction buffer can store up to 12 instructions
  - Branch prediction logic implements dynamic branch prediction
    - Uses 2-bit history
SPARC Processor (cont’d)

Prefetch and dispatch unit (PDU)

| Instruction buffer | Instruction cache |

Grouping logic

| Integer registers and annex |

Integer execution unit (IEU)

Floating-point unit (FPU)

| FP registers |
| FP multiply |
| FP divide |
| FP add |

Graphics unit (GRU)

Memory management unit (MMU)

| iTLB | dTLB |

Load/store unit (LSU)

| Data cache |
| Load buffer |
| Store buffer |

External cache unit (ECU)

Memory interface unit (MIU)

System bus

External cache
SPARC Processor (cont’d)

• Integer execution
  * Has two ALUs
  * A multicycle integer multiplier
  * A multicycle divider

• Floating-point unit
  * Add, multiply, and divide/square root subunits
  * Can issue two FP instructions/cycle
  * Divide and square root operations are not pipelined
    » Single precision takes 12 cycles
    » Double precision takes 22 cycles
SPARC Processor (cont’d)

- 9-stage instruction pipeline
  * 3 stages are added to the integer pipeline to synchronize with FP pipeline

<table>
<thead>
<tr>
<th>Integer pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Floating-point and graphics pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
</tr>
</tbody>
</table>
SPARC Processor (cont’d)

• Fetch and Decode
  * Standard fetch and decode operations

• Group
  * Groups and dispatches up to 4 instructions per cycle
  * Grouping stage is also responsible for
    » Integer data forwarding
    » Handling pipeline stalls due to interlocks

• Cache
  * Used by load/store operations to get data from the data cache
  * FP and graphics instructions start their execution
SPARC Processor (cont’d)

- **N1 and N2**
  * Used to complete load and store operations
- **X2 and X3**
  * FP operations continue their execution initiated in X1 stage
- **N3**
  * Used to resolve traps
- **Write**
  * Write the results to the integer and FP registers
MIPS Processor

- MIPS R4000 processor
  - Superpipelined design
    » Instruction pipeline runs at twice the processor clock
      - Details discussed before
  - Like SPARC, uses 8-stage instruction pipeline for both integer and FP instructions
  - FP unit has three functional units
    » Adder, multiplier, and divider
    » Divider unit is not pipelined
      - Allows only one operation at a time
    » Multiplier unit is pipelined
      - Allows up to two instructions
MIPS Processor

Data cache

Instruction cache

CP0
- Exception/control registers
- Memory management registers
- Translation lookaside buffers

CPU
- CPU registers
  - ALU
  - Load/store unit
  - Integer multiplier/divider
  - Address unit
  - PC incremenet

FPU
- FPU registers
  - Pipeline bypass
  - FP multiplier
  - FP divider
  - FP adder/square root

Pipeline control
Vector Processors

• Vector systems provide instructions that operate at the vector level
  * A vector instruction can replace a loop
    » Example: Adding vectors A and B and storing the result in C
      – n elements in each vector
    » We need a loop that iterates n times
      \[ \text{for}(i=0; \ i<n; \ i++) \]
      \[ C[i] = A[i] + B[i] \]
    » This can be done by a single vector instruction
      \[ \text{V3 V2} + \text{V1} \]
      ✔ Assumes that A is in V2 and B in V1
Vector Processors (cont’d)

• **Architecture**
  *
  * Two types
    » Memory-memory
      – Input operands are in memory
        ➔ Results are also written back to memory
        – First vector machines are of this type
          ➔ CDC Star 100
    » Vector-register
      – Similar to RISC
      – Load/store architecture
      – Input operands are taken from registers
        ➔ Result go into registers as well
      – Modern machines use this architecture
Vector Processors (cont’d)

• Vector-register architecture
  * Five components
    » Vector registers
      – Each can hold a small vector
    » Scalar registers
      – Provide scalar input to vector operations
    » Vector functional units
      – For integer, FP, and logical operations
    » Vector load/store unit
      – Responsible for movement of data between vector registers and memory
    » Main memory
Vector Processors (cont’d)

Based on Cray 1

Vector load/store unit

Main memory

Scalar registers

Vector functional units

FP add

FP multiply

FP reciprocal

Integer add

Logical

Shift

Vector length register

64 elements of 64 bits each

v0  v1  v2  v3  v4  v5  v6  v7

0  1  2  ...  63
Vector Processors (cont’d)

• Advantages of vector processing
  * Flynn’s bottleneck can be reduced
    » Due to vector-level instructions
  * Data hazards can be eliminated
    » Due to structured nature of data
  * Memory latency can be reduced
    » Due to pipelined load and store operations
  * Control hazards can be reduced
    » Due to specification of large number of iterations in one operation
  * Pipelining can be exploited
    » At all levels
Cray X-MP

• Supports up to 4 processors
  * Similar to RISC architecture  
    » Uses load/store architecture
  * Instructions are encoded into a 16- or 32-bit format  
    » 16-bit encoding is called *one parcel*  
    » 32-bit encoding is called *two parcels*

• Has three types of registers
  * Address
  * Scalar
  * Vector
Cray X-MP (cont’d)

• Address registers
  * Eight 24-bit addresses (A0 – A7)
    » Hold memory address for load and store operations
  * Two functional units to perform address arithmetic operations

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>24-bit integer ADD</td>
<td>2 stages</td>
</tr>
<tr>
<td>24-bit integer MULTIPLY</td>
<td>4 stages</td>
</tr>
</tbody>
</table>

* Cray assembly language format

\[
\begin{align*}
Ai & \quad Aj+Ak & (Ai = Aj+Ak) \\
Ai & \quad Aj\cdot Ak & (Ai = Aj\cdot Ak)
\end{align*}
\]
Cray X-MP (cont’d)

- Scalar registers
  - Eight 64-bit scalar registers (S0 – S7)
  - Four types of functional units

<table>
<thead>
<tr>
<th>Scalar functional unit</th>
<th># of stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer add (64-bit)</td>
<td>3</td>
</tr>
<tr>
<td>64-bit shift</td>
<td>2</td>
</tr>
<tr>
<td>128-bit shift</td>
<td>3</td>
</tr>
<tr>
<td>64-bit logical</td>
<td>1</td>
</tr>
<tr>
<td>POP/Parity (population/parity)</td>
<td>4</td>
</tr>
<tr>
<td>POP/Parity (leading zero count)</td>
<td>3</td>
</tr>
</tbody>
</table>
Cray X-MP (cont’d)

• Vector registers
  * Eight 64-element vector registers
    » Each holds 64 bits
  * Each vector instruction works on the first VL elements
    » VL is in the vector length register
  * Vector functional units
    » Integer ADD
    » SHIFT
    » Logical
    » POP/Parity
    » FP ADD
    » FP MULTIPLY
    » Reciprocal
### Cray X-MP (cont’d)

#### Vector functional units

<table>
<thead>
<tr>
<th>Vector functional unit</th>
<th>#stages</th>
<th>Avail. to chain</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit integer ADD</td>
<td>3</td>
<td>8</td>
<td>VL + 8</td>
</tr>
<tr>
<td>64-bit SHIFT</td>
<td>3</td>
<td>8</td>
<td>VL + 8</td>
</tr>
<tr>
<td>128-bit SHIFT</td>
<td>4</td>
<td>9</td>
<td>VL + 9</td>
</tr>
<tr>
<td>Full vector LOGICAL</td>
<td>2</td>
<td>7</td>
<td>VL + 7</td>
</tr>
<tr>
<td>Second vector LOGICAL</td>
<td>4</td>
<td>9</td>
<td>VL + 9</td>
</tr>
<tr>
<td>POP/Parity</td>
<td>5</td>
<td>10</td>
<td>VL + 10</td>
</tr>
<tr>
<td>Floating ADD</td>
<td>6</td>
<td>11</td>
<td>VL + 11</td>
</tr>
<tr>
<td>Floating MULTIPLY</td>
<td>7</td>
<td>12</td>
<td>VL + 12</td>
</tr>
<tr>
<td>Reciprocal approximation</td>
<td>14</td>
<td>19</td>
<td>VL + 19</td>
</tr>
</tbody>
</table>
Cray X-MP (cont’d)

• Sample instructions

1. \( Vi \ Vj+Vk \) ; \( Vi = Vj+Vk \) integer add
2. \( Vi \ Sj+Vk \) ; \( Vi = Sj+Vk \) integer add
3. \( Vi \ Vj+FVk \) ; \( Vi = Vj+Vk \) FP add
4. \( Vi \ Sj+FVk \) ; \( Vi = Vj+Vk \) FP add
5. \( Vi \ ,A0,Ak \) ; \( Vi = M(A0;Ak) \)
   Vector load with stride Ak
6. \( ,A0,Ak \ Vi \) ; \( M(A0;Ak) = Vi \)
   Vector store with stride Ak
Vector Length

- If the vector length we are dealing with is equal to VL, no problem
  - What if vector length < VL
    - Simple case
    - Store the actual length of the vector in the VL register
      
      | A1 | 40 |
      | VL | A1 |
      | V2 | V3+FV4 |

    - We use two instructions to load VL as
      
      | VL | 40 |

      is not allowed
Vector Length

* What if vector length > VL
  » Use strip mining technique
  » Partition the vector into strips of VL elements
  » Process each strip, including the odd sized one, in a loop
  » Example: Vector registers are 64 elements long
    – Odd size strip size = N mod 64
    – Number of strips = (N/64) + 1
    – If N = 200
      → Four strips: 64, 64, 64, 8 elements
      → In one iteration, we set VL = 8
      → Other three iterations VL = 64
Vector Stride

• Refers to the difference between elements accessed

• 1-D array
  * Accessing successive elements
    » Stride = 1

• Multidimensional arrays are stored in
  * Row-major
  * Column-major
  * Accessing a column or a row needs a non-unit stride
### Vector Stride (cont’d)

<table>
<thead>
<tr>
<th>Row 0</th>
<th>Row 1</th>
<th>Row 2</th>
<th>Row 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>21</td>
<td>31</td>
<td>41</td>
</tr>
<tr>
<td>12</td>
<td>22</td>
<td>32</td>
<td>42</td>
</tr>
<tr>
<td>13</td>
<td>23</td>
<td>33</td>
<td>43</td>
</tr>
<tr>
<td>14</td>
<td>24</td>
<td>34</td>
<td>44</td>
</tr>
</tbody>
</table>

(a) Row-major order

<table>
<thead>
<tr>
<th>Column 0</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
</tr>
</tbody>
</table>

(b) Column-major order

**Stride = 4 to access a column, 1 to access a row**

**Stride = 4 to access a row, 1 to access a column**
Vector Stride (cont’d)

• Cray X-MP provides instructions to load and store vectors with non-unit stride
  * Example 1: non-unit stride load
    $$\text{Vi} , A0, Ak$$
    Loads vector register Vi with stride Ak
  
  * Example 2: unit stride load
    $$\text{Vi} , A0, 1$$
    Loads vector register Vi with stride 1
Vector Operations on X-MP

• Simple vector ADD
  * Setup phase takes 3 clocks
  * Shut down phase takes 3 clocks

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|-------------|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| A1 5        | I | E |   |   |   |   |   |   |   |     |     |     |     |     |     |     |     |     |
| VL A1       | I | E |   |   |   |   |   |   |   |     |     |     |     |     |     |     |     |     |
| V1 V2 + FV3 | I | S | S | S | F | F | F | F | F | R1 | R2 | R3 | R4 | R5 | D  | D  | D  | D  |

Setup phase

Shutdown phase
Vector Operations on X-MP (cont’d)

- Two independent vector operations
  - FP add
  - FP multiply
  * Overlapped execution is possible

<table>
<thead>
<tr>
<th>Instruction</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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 5</td>
<td>I</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VL A1</td>
<td>I</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V1 V2 + FV3</td>
<td>I</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R4</td>
<td>R5</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V4 V5 * FV6</td>
<td>I</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R4</td>
<td>R5</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector Operations on X-MP (cont’d)

- Chaining example
  - Dependency from FP add to FP multiply
    - Multiply unit is kept on hold
    - X-MP allows using the first result after 2 clocks

<table>
<thead>
<tr>
<th>Instruction</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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 5</td>
<td>I</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VL A1</td>
<td>I</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V1 V2+FV3</td>
<td>I</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R4</td>
<td>R5</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V4 V5*FV1</td>
<td>I</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td></td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R4</td>
<td>R5</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Multiply unit on hold
Performance

• Pipeline performance

\[
\text{Speedup} = \frac{\text{non-pipelined execution time}}{\text{pipelined execution time}}
\]

• Ideal speedup:
  * $n$ stage pipeline should give a speedup of $n$

• Two factors affect pipeline performance
  * Pipeline fill
  * Pipeline drain
Performance (cont’d)

• N computations on a n-stage pipeline
  * Non-pipelined: \((N \times n \times T)\) time units
  * Pipelined: \((n + N - 1) T\) time units

\[
\text{Speedup} = \frac{N \times n}{n + N - 1}
\]

Rewriting

\[
\text{Speedup} = \frac{1}{1/N + 1/n - 1/(n \times N)}
\]

Speedup reaches the ideal value of \(n\) as \(N \to \infty\)
Performance (cont’d)

Number of elements, $N$

Speedup
Performance (cont’d)

$\text{Speedup}$ vs $\text{Number of stages, } n$
Performance (cont’d)

• Vector processing performance
  * Impact of vector register length
    » Exhibits saw-tooth shaped performance
      – Speedup increases as the vector size increases to VL
        ‣ Due to amortization of pipeline fill cost
      – Speedup drops as we increase the vector length to VL+1
        ‣ We need one more strip to process the vector
        ‣ Speedup increases as we increase the vector length beyond
      – Speedup peaks at vector lengths that are a multiple of the vector register length