We Are Almost Done With This…

- Single-cycle Microarchitectures
- Multi-cycle Microarchitectures
- Pipelining
  - Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Other Execution Paradigms
Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Fine-Grained Multithreading
- Systolic Arrays
- Decoupled Access Execute
- **SIMD Processing (Vector and array processors, GPUs)**
Readings for this Week

**Required**

**Recommended**
Announcement

- **Late submission of lab reports in Moodle**
  - Open until June 14, 2019, 11:59pm (cutoff date -- hard deadline)
  - You can submit any past lab report, which you have not submitted before its deadline
  - It is NOT allowed to re-submit anything (lab reports, extra assignments, etc.) that you had already submitted via other Moodle assignments
  - We will grade your reports, but late submission has a **penalization of 1 point**, that is, the highest possible score per lab report will be 2 points
Exploiting Data Parallelism: SIMD Processors and GPUs
SIMD Processing:
Exploiting Regular (Data) Parallelism
Flynn’s Taxonomy of Computers


- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- **MISD**: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor
Data Parallelism

- Concurrency arises from performing the **same operation on different pieces of data**
  - Single instruction multiple data (SIMD)
  - E.g., dot product of two vectors

- Contrast with data flow
  - Concurrency arises from executing different operations in parallel (in a data driven manner)

- Contrast with thread (“control”) parallelism
  - Concurrency arises from executing different threads of control in parallel

- SIMD exploits operation-level parallelism on different data
  - Same operation concurrently applied to different pieces of data
  - A form of ILP where instruction happens to be the same across data
SIMD Processing

- Single instruction operates on multiple data elements
  - In time or in space
- Multiple processing elements

- Time-space duality
  - **Array processor**: Instruction operates on multiple data elements at the same time using different spaces
  - **Vector processor**: Instruction operates on multiple data elements in consecutive time steps using the same space
Array vs. Vector Processors

Instruction Stream

**ARRAY PROCESSOR**

<table>
<thead>
<tr>
<th>PE0</th>
<th>PE1</th>
<th>PE2</th>
<th>PE3</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD0</td>
<td>LD1</td>
<td>LD2</td>
<td>LD3</td>
</tr>
<tr>
<td>AD0</td>
<td>AD1</td>
<td>AD2</td>
<td>AD3</td>
</tr>
<tr>
<td>MU0</td>
<td>MU1</td>
<td>MU2</td>
<td>MU3</td>
</tr>
<tr>
<td>ST0</td>
<td>ST1</td>
<td>ST2</td>
<td>ST3</td>
</tr>
</tbody>
</table>

**VECTOR PROCESSOR**

<table>
<thead>
<tr>
<th>LD</th>
<th>ADD</th>
<th>MUL</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD1</td>
<td>AD0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD2</td>
<td>AD1</td>
<td>MU0</td>
<td></td>
</tr>
<tr>
<td>LD3</td>
<td>AD2</td>
<td>MU1</td>
<td>ST0</td>
</tr>
</tbody>
</table>

- **Same op @ same time**
- **Different ops @ same space**
- **Different ops @ time**
- **Same op @ space**
SIMD Array Processing vs. VLIW

- VLIW: Multiple independent operations packed together by the compiler
SIMD Array Processing vs. VLIW

- Array processor: Single operation on multiple (different) data elements
A vector is a one-dimensional array of numbers

Many scientific/commercial programs use vectors

for (i = 0; i <= 49; i++)
    C[i] = (A[i] + B[i]) / 2

A vector processor is one whose instructions operate on vectors rather than scalar (single data) values

Basic requirements

- Need to load/store vectors \( \rightarrow \) vector registers (contain vectors)
- Need to operate on vectors of different lengths \( \rightarrow \) vector length register (VLEN)
- Elements of a vector might be stored apart from each other in memory \( \rightarrow \) vector stride register (VSTR)
  - Stride: distance in memory between two elements of a vector
Vector Processors (II)

- A vector instruction performs an operation on each element in consecutive cycles
  - Vector functional units are pipelined
  - Each pipeline stage operates on a different data element

- Vector instructions allow deeper pipelines
  - **No intra-vector dependencies** → no hardware interlocking needed within a vector
  - **No control flow within a vector**
  - **Known stride allows easy address calculation** for all vector elements
    - Enables *prefetching* of vectors into registers/cache/memory
Vector Processor Advantages

+ No dependencies within a vector
  - Pipelining & parallelization work really well
  - Can have very deep pipelines, no dependencies!

+ Each instruction generates a lot of work
  - Reduces instruction fetch bandwidth requirements

+ Highly regular memory access pattern

+ No need to explicitly code loops
  - Fewer branches in the instruction sequence
Vector Processor Disadvantages

-- Works (only) if parallelism is regular (data/SIMD parallelism)
  ++ Vector operations
-- Very inefficient if parallelism is irregular
  -- How about searching for a key in a linked list?

To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is

Vector Processor Limitations

-- Memory (bandwidth) can easily become a bottleneck, especially if

1. compute/memory operation balance is not maintained
2. data is not mapped appropriately to memory banks
Vector Processing in More Depth
Vector Registers

- Each vector data register holds N M-bit values
- Vector control registers: VLEN, VSTR, VMASK
- Maximum VLEN can be N
  - Maximum number of elements stored in a vector register
- Vector Mask Register (VMASK)
  - Indicates which elements of vector to operate on
  - Set by vector test instructions
    - e.g., VMASK[i] = (V_k[i] == 0)
Vector Functional Units

- Use a deep pipeline to execute element operations → fast clock cycle

- Control of deep pipeline is simple because elements in vector are independent

\[
V_1 \times V_2 \rightarrow V_3
\]

*Six stage multiply pipeline*
Vector Machine Organization (CRAY-1)

- CRAY-1
- Scalar and vector modes
- 8 64-element vector registers
- 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers
CRAY X-MP-28 @ ETH (CAB, E Floor)

Cray X-MP-28

Der von Seymour Cray entwickelte Supercomputer Cray X-MP
beachtet durch seinen kleinen, eleganten Aufbau. Von 1982
bis 1986 hat der 5,5 Tonnen schwere und bis zu 15 Millionen
Dollar teure Walter-Decoder als schnellsten Computer der
Welt.

Die Anschaffung des Cray X-MP28 im Jahr 1986 markiert
den Ausgangspunkt für ein Engagement der ETH, auch um
Betrieb des Rechensystems mit einem solchen Computer.

Beim ausgeklügelten System handelt es sich lediglich um die
Prozessorenleistung. Zusätzlich war noch ein 1,6-Bit von
Anschluss von Reihenrechnern und Anschluss Rechners
Rechners.

Für den Betrieb waren an der ETH etwa vier Angestellte von
Cray Research vor Ort. Zwei für die Pflege der Hardware,
neun für die Programmierung und Administration.

Seit 1991 sind es Supercomuter der ETH Zürich im Swiss
National Supercomputing Center (CSCS) im Betrieb ablaufender
Aktualität ist es wieder ein Cray, der dort für Simulationsleistungen
vorhanden ist. Der ETH-Supercomputer mit Ende 2013 als schnellster
energieeffizienteste Rechner

Raspberry Pi 1 Model B

Die gängige 400-Megaflops werden bereit zum Beispiel von
der weltbekannten Raspberry Pi ausgeliefert, ausgestattet mit
einem ARM11-Processor (125 MHz System).
CRAY X-MP System Organization


CRAY X-MP Design Detail

CRAY X-MP design detail

Mainframe

CRAY X-MP single- and multiprocessor systems are designed to offer users outstanding performance on large-scale, compute-intensive and I/O-bound jobs.

CRAY X-MP mainframes consist of six (X-MP/1), eight (X-MP/2) or twelve (X-MP/4) vertical columns arranged in an arc. Power supplies and cooling are clustered around the base and extend outward.

<table>
<thead>
<tr>
<th>Model</th>
<th>Number of CPUs</th>
<th>Memory size (millions of 64-bit words)</th>
<th>Number of banks</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRAY X-MP/4/16</td>
<td>4</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>CRAY X-MP/48</td>
<td>4</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>CRAY X-MP/2/16</td>
<td>2</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>CRAY X-MP/2/24</td>
<td>2</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>CRAY X-MP/2/4</td>
<td>2</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>CRAY X-MP/2/8</td>
<td>1</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>CRAY X-MP/2/14</td>
<td>1</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>CRAY X-MP/2/12</td>
<td>1</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>CRAY X-MP/2/11</td>
<td>1</td>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

A description of the major system components and their functions follows.

CPU computation section

Within the computation section of each CPU are operating registers, functional units and an instruction control network — hardware elements that cooperate in executing sequences of instructions. The instruction control network makes all decisions related to instruction issue as well as coordinating the three types of processing within each CPU: vector, scalar and address. Each of the processing modes has its associated registers and functional units.

The block diagram of a CRAY X-MP/4 (opposite page) illustrates the relationship of the registers to the functional units, instruction buffers, I/O channel control registers, interprocessor communications section and memory. For multiple-processor CRAY X-MP models, the interprocessor communications section coordinates processing between CPUs, and central memory is shared.

Registers

The basic set of programmable registers is composed of:

- Eight 24-bit address (A) registers
- Sixty-four 24-bit intermediate address (B) registers
- Eight 64-bit scalar (S) registers
- Sixty-four 64-bit scalar-save (T) registers
- Eight 64-element (4096-bit) vector (V) registers with 64 bits per element

The 24-bit A registers are generally used for addressing and counting operations. Associated with them are 64 B registers, also 24 bits wide. Since the transfer between an A and a B register takes only one clock period, the B registers assume the role of data cache, storing information for fast access without tying up the A registers for relatively long periods.

Hardware features:

- 9.5 nsec clock
- One, two or four CPUs, each with its own computation and control sections
- Large multiport central memory
- Memory bank cycle time of 38 nsec on X-MP/4 systems, 76 nsec on X-MP/1 and X-MP/2 models
- Memory bandwidth of 25-100 gigaibits, depending on model
- I/O section
- Proven cooling and packaging technologies
### CRAY X-MP CPU Functional Units

<table>
<thead>
<tr>
<th>CRAY X-MP CPU functional units</th>
<th>Register usage</th>
<th>Time in clock periods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Address functional units</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addition</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>Multiplication</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td><strong>Scalar functional units</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addition</td>
<td>S</td>
<td>3</td>
</tr>
<tr>
<td>Shift-single</td>
<td>S</td>
<td>2</td>
</tr>
<tr>
<td>Shift-double</td>
<td>S</td>
<td>3</td>
</tr>
<tr>
<td>Logical</td>
<td>S</td>
<td>1</td>
</tr>
<tr>
<td>Population, parity and leading zero</td>
<td>S</td>
<td>3 or 4</td>
</tr>
<tr>
<td><strong>Vector functional units</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addition</td>
<td>V</td>
<td>3</td>
</tr>
<tr>
<td>Shift</td>
<td>V</td>
<td>3 or 4</td>
</tr>
<tr>
<td>Full vector logical</td>
<td>V</td>
<td>2</td>
</tr>
<tr>
<td>Second vector logical</td>
<td>V</td>
<td>4</td>
</tr>
<tr>
<td>Population, parity</td>
<td>V</td>
<td>5</td>
</tr>
<tr>
<td><strong>Floating-point functional units</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addition</td>
<td>S and V</td>
<td>6</td>
</tr>
<tr>
<td>Multiplication</td>
<td>S and V</td>
<td>7</td>
</tr>
<tr>
<td>Reciprocal approximation</td>
<td>S and V</td>
<td>14</td>
</tr>
</tbody>
</table>

CRAY X-MP System Configuration

<table>
<thead>
<tr>
<th>System configuration options</th>
<th>X-MP/1</th>
<th>X-MP/2</th>
<th>X-MP/4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mainframe</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPUs</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Bipolar memory (64-bit words)</td>
<td>N/A</td>
<td>N/A</td>
<td>8 or 16M</td>
</tr>
<tr>
<td>MOS memory (64-bit words)</td>
<td>1, 2, 4 or 8M</td>
<td>4, 8 or 16M</td>
<td>N/A</td>
</tr>
<tr>
<td>6-Mbyte channels</td>
<td>2 or 4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>100-Mbyte channels</td>
<td>1 or 2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1000-Mbyte channels</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>I/O Subsystem</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I/O processors</td>
<td>2, 3 or 4</td>
<td>2, 3 or 4</td>
<td>4</td>
</tr>
<tr>
<td>Disk storage units</td>
<td>2-32</td>
<td>2-32</td>
<td>2-32</td>
</tr>
<tr>
<td>Magnetic tape channels</td>
<td>1-8</td>
<td>1-8</td>
<td>1-8</td>
</tr>
<tr>
<td>Front-end interfaces</td>
<td>1-7</td>
<td>1-7</td>
<td>1-7</td>
</tr>
<tr>
<td>Buffer memory (Mbytes)</td>
<td>8, 32 or 64</td>
<td>8, 32 or 64</td>
<td>64</td>
</tr>
<tr>
<td><strong>Solid-state Storage Device</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory size (Mbytes)</td>
<td>256, 512 or 1024</td>
<td>256, 512 or 1024</td>
<td>256, 512 or 1024</td>
</tr>
</tbody>
</table>

N/A signifies option is not available on the model

Seymour Cray, the Father of Supercomputers

"If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?"

© amityrebecca / Pinterest. https://www.pinterest.ch/pin/473018767088408061/

Vector Machine Organization (CRAY-1)

- CRAY-1

- Scalar and vector modes
- 8 64-element vector registers
- 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers
Loading/Storing Vectors from/to Memory

- Requires loading/storing multiple elements

- Elements separated from each other by a constant distance (stride)
  - Assume stride = 1 for now

- Elements can be loaded in consecutive cycles if we can start the load of one element per cycle
  - Can sustain a throughput of one element per cycle

- Question: How do we achieve this with a memory that takes more than 1 cycle to access?

- Answer: Bank the memory; interleave the elements across banks
Memory Banking

- Memory is divided into banks that can be accessed independently; banks share address and data buses (to minimize pin cost)
- Can start and complete one bank access per cycle
- Can sustain $N$ parallel accesses if all $N$ go to different banks

![Diagram of memory banking]

Picture credit: Derek Chiou
Vector Memory System

- Next address = Previous address + Stride
- If (stride == 1) && (consecutive elements interleaved across banks) && (number of banks >= bank latency), then
  - we can sustain 1 element/cycle throughput

- For I = 0 to 49
  - C[i] = (A[i] + B[i]) / 2

- Scalar code (instruction and its latency)
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVI R0 = 50</td>
<td>1</td>
</tr>
<tr>
<td>MOVA R1 = A</td>
<td>1</td>
</tr>
<tr>
<td>MOVA R2 = B</td>
<td>1</td>
</tr>
<tr>
<td>MOVA R3 = C</td>
<td>1</td>
</tr>
<tr>
<td>LD R4 = MEM[R1++]</td>
<td>11; autoincrement addressing</td>
</tr>
<tr>
<td>LD R5 = MEM[R2++]</td>
<td>11</td>
</tr>
<tr>
<td>ADD R6 = R4 + R5</td>
<td>4</td>
</tr>
<tr>
<td>SHFR R7 = R6 &gt;&gt; 1</td>
<td>1</td>
</tr>
<tr>
<td>ST MEM[R3++] = R7</td>
<td>11</td>
</tr>
<tr>
<td>DECBNZ R0, X</td>
<td>2</td>
</tr>
</tbody>
</table>

304 dynamic instructions
Scalar Code Execution Time (In Order)

- Scalar execution time on an in-order processor with 1 bank
  - First two loads in the loop cannot be pipelined: 2*11 cycles
  - 4 + 50*40 = 2004 cycles

- Scalar execution time on an in-order processor with 16 banks (word-interleaved: consecutive words are stored in consecutive banks)
  - First two loads in the loop can be pipelined
  - 4 + 50*30 = 1504 cycles

- Why 16 banks?
  - 11-cycle memory access latency
  - Having 16 (>11) banks ensures there are enough banks to overlap enough memory operations to cover memory latency
Vectorizable Loops

- A loop is **vectorizable** if each iteration is independent of any other.

- For $I = 0$ to $49$
  - $C[i] = (A[i] + B[i]) / 2$

- Vectorized loop (each instruction and its latency):
  - MOV1 VLEN = 50
  - MOV1 VSTR = 1
  - VLD V0 = A
  - VLD V1 = B
  - VADD V2 = V0 + V1
  - VSHFR V3 = V2 >> 1
  - VST C = V3

  7 dynamic instructions
Basic Vector Code Performance

- Assume **no chaining** (no vector data forwarding)
  - i.e., output of a vector functional unit cannot be used as the direct input of another
  - The entire vector register needs to be ready before any element of it can be used as part of another operation

- One memory port (one address generator)

- 16 memory banks (word-interleaved)

- 285 cycles
Vector Chaining

- **Vector chaining**: Data forwarding from one vector functional unit to another

Slide credit: Krste Asanovic
Vector Code Performance - Chaining

- **Vector chaining**: Data forwarding from one vector functional unit to another

These two VLDs cannot be pipelined. WHY?

- 182 cycles

VLD and VST cannot be pipelined. WHY?

Strict assumption: Each memory bank has a single port (memory bandwidth bottleneck)
Vector Code Performance – Multiple Memory Ports

- Chaining and 2 load ports, 1 store port in each bank

- 79 cycles
- 19X perf. improvement!
What if # data elements > # elements in a vector register?

- Idea: Break loops so that each iteration operates on # elements in a vector register
  - E.g., 527 data elements, 64-element VREGs
  - 8 iterations where VLEN = 64
  - 1 iteration where VLEN = 15 (need to change value of VLEN)

- Called vector stripmining
Surface mining, including strip mining, open-pit mining and mountaintop removal mining, is a broad category of mining in which soil and rock overlying the mineral deposit (the overburden) are removed, in contrast to underground mining, in which the overlying rock is left in place, and the mineral removed through shafts or tunnels.

Surface mining began in the mid-sixteenth century[1] and is practiced throughout the world, although the majority of surface coal mining occurs in North America.[2] It gained
Questions (II)

- What if vector data is not stored in a strided fashion in memory? (irregular memory access to a vector)
  - Idea: Use indirection to combine/pack elements into vector registers
  - Called scatter/gather operations
Gather/Scatter Operations

Want to vectorize loops with indirect accesses:

```c
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

Indexed load instruction (Gather)

```assembly
LV vD, rD       # Load indices in D vector
LVI vC, rC, vD  # Load indirect from rC base
LV vB, rB       # Load B vector
ADDV.D vA,vB,vC # Do add
SV vA, rA       # Store result
```
Gather/Scatter Operations

- Gather/scatter operations often implemented in hardware to handle sparse vectors (matrices)
- Vector loads and stores use an index vector which is added to the base register to generate the addresses

**Scatter example**

<table>
<thead>
<tr>
<th>Index Vector</th>
<th>Data Vector (to Store)</th>
<th>Stored Vector (in Memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.14</td>
<td>Base+0 3.14</td>
</tr>
<tr>
<td>2</td>
<td>6.5</td>
<td>Base+1 X</td>
</tr>
<tr>
<td>6</td>
<td>71.2</td>
<td>Base+2 6.5</td>
</tr>
<tr>
<td>7</td>
<td>2.71</td>
<td>Base+3 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+4 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+5 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+6 71.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+7 2.71</td>
</tr>
</tbody>
</table>
Conditional Operations in a Loop

What if some operations should not be executed on a vector (based on a dynamically-determined condition)?

```
for (i=0; i<N; i++)
    if (a[i] != 0) then b[i]=a[i]*b[i]
```

Idea: Masked operations

- VMASK register is a bit mask determining which data element should not be acted upon
  
  VLD V0 = A  
  VLD V1 = B  
  VMASK = (V0 != 0)  
  VMUL V1 = V0 * V1  
  VST B = V1

- This is _predicated execution_. Execution is _predicated_ on mask bit.
Another Example with Masking

for (i = 0; i < 64; ++i)
    if (a[i] >= b[i])
        c[i] = a[i]
    else
        c[i] = b[i]

Steps to execute the loop in SIMD code

1. Compare A, B to get VMASK
2. Masked store of A into C
3. Complement VMASK
4. Masked store of B into C

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>VMASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>-5</td>
<td>-4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>-7</td>
<td>-8</td>
<td>1</td>
</tr>
</tbody>
</table>
Masked Vector Instructions

Simple Implementation
- execute all N operations, turn off result writeback according to mask

Density-Time Implementation
- scan mask vector and only execute elements with non-zero masks

Which one is better? Tradeoffs?

Slide credit: Krste Asanovic
Some Issues

- Stride and banking
  - As long as they are *relatively prime* to each other and there are enough banks to cover bank access latency, we can sustain 1 element/cycle throughput

- Storage of a matrix
  - **Row major**: Consecutive elements in a row are laid out consecutively in memory
  - **Column major**: Consecutive elements in a column are laid out consecutively in memory
  - You need to change the stride when accessing a row versus column
Matrix Multiplication

- **A and B, both in row-major order**

  A and B, both in row-major order, for the matrix multiplication of A and B.

  **A:** Load $A_0$ into vector register $V_1$
  - Each time, increment address by one to access the next column
  - Accesses have a **stride of 1**

  **B:** Load $B_0$ into vector register $V_2$
  - Each time, increment address by 10
  - Accesses have a **stride of 10**

  Different strides can lead to bank conflicts.

  How do we minimize them?

- **Dot products of rows and columns of A and B**

  Dot products of rows and columns of A and B.
Minimizing Bank Conflicts

- More banks

- Better data layout to match the access pattern
  - Is this always possible?

- Better mapping of address to bank
  - E.g., randomized mapping
Array vs. Vector Processors, Revisited

- Array vs. vector processor distinction is a “purist’s” distinction

- Most “modern” SIMD processors are a combination of both
  - They exploit data parallelism in both time and space
  - GPUs are a prime example we will cover in a bit more detail
Recall: Array vs. Vector Processors

**ARRAY PROCESSOR**

- LD0
- AD0
- MU0
- ST0

**VECTOR PROCESSOR**

- LD
- ADD
- MUL
- ST

### Instruction Stream

- **LD**: VR ← A[3:0]
- **ADD**: VR ← VR, 1
- **MUL**: VR ← VR, 2
- **ST**: A[3:0] ← VR

#### Time

- **LD0**: AD0
- **AD0**: MU0
- **MU0**: ST0
- **ST0**: LD0

#### Space

- **Same op @ same time**
- **Different ops @ time**
- **Different ops @ same space**
- **Same op @ space**
Vector Instruction Execution

![Diagram showing vector instruction execution using one and four pipelined functional units.]

- **VADD A,B → C**
- **Execution using one pipelined functional unit**
- **Execution using four pipelined functional units**

- Time
- Space

Slide credit: Krste Asanovic
Vector Unit Structure

Partitioned Vector Registers

Lane

Memory Subsystem

Elements 0, 4, 8, ...

Elements 1, 5, 9, ...

Elements 2, 6, 10, ...

Elements 3, 7, 11, ...

Functional Unit

Slide credit: Krste Asanovic
Vector Instruction Level Parallelism

Can overlap execution of multiple vector instructions

- Example machine has 32 elements per vector register and 8 lanes
- Completes 24 operations/cycle while issuing 1 vector instruction/cycle

Slide credit: Krste Asanovic
We did not cover the following slides. They are for your preparation for the next lecture.
Automatic Code Vectorization

Scalar Sequential Code

Vectorized Code

Vectorization is a compile-time reordering of operation sequencing
⇒ requires extensive loop dependence analysis

Slide credit: Krste Asanovic
Vector/SIMD Processing Summary

- Vector/SIMD machines are good at exploiting regular data-level parallelism
  - Same operation performed on many data elements
  - Improve performance, simplify design (no intra-vector dependencies)

- Performance improvement limited by vectorizability of code
  - Scalar operations limit vector machine performance
  - Remember Amdahl’s Law
  - CRAY-1 was the fastest SCALAR machine at its time!

- Many existing ISAs include (vector-like) SIMD operations
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD
SIMD Operations in Modern ISAs
SIMD ISA Extensions

- Single Instruction Multiple Data (SIMD) extension instructions
  - Single instruction acts on multiple pieces of data at once
  - Common application: graphics
  - Perform short arithmetic operations (also called *packed arithmetic*)
- For example: add four 8-bit numbers
- Must modify ALU to eliminate carries between 8-bit values

```
padd8 $s2, $s0, $s1
```

<table>
<thead>
<tr>
<th>32</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
<th>Bit position</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_3</td>
<td>a_2</td>
<td>a_1</td>
<td>a_0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$s0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
+-----+-----+-----+-----+-----+-----+-----+-----+--------------|
| b_3 | b_2 | b_1 | b_0 |     |     |     |     | $s1          |
|     |     |     |     |     |     |     |     |              |
| a_3 + b_3 | a_2 + b_2 | a_1 + b_1 | a_0 + b_0 | $s2 |
Intel Pentium MMX Operations

- Idea: One instruction operates on multiple data elements simultaneously
  - *À la* array processing (yet much more limited)
  - Designed with multimedia (graphics) operations in mind

No VLEN register

Opcode determines data type:
- 8 8-bit bytes
- 4 16-bit words
- 2 32-bit doublewords
- 1 64-bit quadword

Stride is always equal to 1.

MMX Example: Image Overlaying (I)

- Goal: Overlay the human in image 1 on top of the background in image 2

![Image overlay diagram]

**Figure 8. Chroma keying: image overlay using a background color.**

```c
for (i=0; i<image_size; i++) {
    if (x[i] == Blue) new_image[i] = y[i];
    else new_image[i] = x[i];
}
```

**Figure 9. Generating the selection bit mask.**

Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

Movq mm3, mem1  /* Load eight pixels from woman’s image
Movq mm4, mem2  /* Load eight pixels from the blossom image
Pcmpeqb mm1, mm3
Pand mm4, mm1
Pandn mm1, mm3
Por mm4, mm1

Figure 11. MMX code sequence for performing a conditional select.

Design of Digital Circuits
Lecture 20: SIMD Processors

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zurich
Spring 2019
9 May 2019