Design of Digital Circuits
Lecture 20: SIMD Processors

Prof. Onur Mutlu
ETH Zurich
Spring 2018
11 May 2018
New Course: Bachelor’s Seminar in Comp Arch

- Fall 2018
- 2 credit units

- Rigorous seminar on fundamental and cutting-edge topics in computer architecture

- Critical presentation, review, and discussion of seminal works in computer architecture
  - We will cover many ideas, analyze their tradeoffs (strengths and weaknesses), perform critical thinking and brainstorming

- Participation, presentation, report and review writing
Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Other Execution Paradigms
Readings for Today


Other Approaches to Concurrency (or Instruction Level Parallelism)
Approaches to (Instruction-Level) Concurrency

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

**ARRAY PROCESSOR**

- LD0
- LD1
- LD2
- LD3
- AD0
- AD1
- AD2
- AD3
- MU0
- MU1
- MU2
- MU3
- ST0
- ST1
- ST2
- ST3

**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
- LD1
- AD0
- AD1
- MU0
- MU1
- LD2
- LD3
- AD2
- MU2
- ST0
- AD3
- MU3
- ST3

**Space**

- Same op @ same time
- Different ops @ time
- Different ops @ same space
- 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

![Diagram showing SIMD Array Processing]
Vector Processors (I)

- 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 → **vector registers (contain vectors)**
  - Need to operate on vectors of different lengths → **vector length register (VLEN)**
  - Elements of a vector might be stored apart from each other in memory → **vector stride register (VSTR)**
    - **Stride**: distance 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 deep pipeline to execute element operations → fast clock cycle
- Control of deep pipeline is simple because elements in vector are independent

![Six stage multiply pipeline](image)

V1 * V2 → V3
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 @ ETHz (CAB, E Floor)

Cray X-MP-28


Für den Betrieb waren an der ETH stets vier Angestellte von Cray Research vor Ort. Zwei für die Wartung der Hardware, zwei für die Programmierung und Administration.

Seit 1991 sind die Supercomputer der ETH Zürich im Swiss National Supercomputing Centre (CSCS) im Tessin abgelegt. Aktuell ist es wieder ein Cray, das dort für Simulationen sorgt. „Bei Cray“, so Cray, „ist es bei Cray Superrechner seit Ende 2013 als kleinster und energieeffizientester Rechner Europas.

Miniaturisierung und explodierende Leistung

Wenig später entscheidet sich die Leistungsfähigkeit der Hardware entscheidet, zeigt der Vergleich des gelben Reisern mit einem Mikrorechner von heute.

Cray X-MP/28


Blueberry Pi model B+

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/416</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/216</td>
<td>2</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>CRAY X-MP/28</td>
<td>2</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>CRAY X-MP/24</td>
<td>2</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>CRAY X-MP/18</td>
<td>1</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>CRAY X-MP/14</td>
<td>1</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>CRAY X-MP/12</td>
<td>1</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>CRAY X-MP/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.
## 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

## System configuration options

<table>
<thead>
<tr>
<th></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?"
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

![Memory Banking Diagram](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 can sustain 1 element/cycle throughput
Scalar Code Example

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

- Scalar code (instruction and its latency)
  MOV R0 = 50 1
  MOV R1 = A 1 304 dynamic instructions
  MOV R2 = B 1
  MOV R3 = C 1
  X: LD R4 = MEM[R1++] 11 ;autoincrement addressing
  LD R5 = MEM[R2++] 11
  ADD R6 = R4 + R5 4
  SHFR R7 = R6 >> 1 1
  ST MEM[R3++] = R7 11
  DECBNZ R0, X 2 ;decrement and branch if NZ
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):
  - MOV I VLEN = 50
  - MOV I 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

```
LV  v1
MULV v3,v1,v2
ADDV v5, v3, v4
```

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!
Questions (I)

- 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

- 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**

<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)?

  ```
  loop: 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

\[
\begin{align*}
M[0] &= 0 & C[0]
\end{align*}
\]

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

\[
\begin{align*}
M[7] &= 1 \\
M[6] &= 0 \\
M[5] &= 1 \\
M[4] &= 1 \\
M[3] &= 0 \\
M[2] &= 0 \\
M[1] &= 1 \\
M[0] &= 0 \\
\end{align*}
\]

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 & B, both m row major order

\[
\begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 \\
6 & 7 & 8 & 9 & 10 & 11
\end{bmatrix}
\quad \begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\
20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 \\
30 & 31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 \\
40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 & 49 \\
50 & 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59
\end{bmatrix}
\]

\[A_{6 \times 6} \cdot B_{6 \times 10} \rightarrow C_{4 \times 10}\] (dot products of rows & columns of A & B)

A:

Load Ao into a vector register \( V1 \)

- each time you need to increment the address by 1 to access the next column
- First matrix accesses have a \textit{stride of 1}

B:

Load Bo into a vector register \( V2 \)

- each time you need to increment by 10
- \textit{Stride of 10}

Different strides can lead to bank conflicts.

- How do you minimize them?
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
Remember: Array vs. Vector Processors

**ARRAY PROCESSOR**

Instruction Stream

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

**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>AD0</td>
<td>MU0</td>
<td>ST0</td>
</tr>
<tr>
<td>LD1</td>
<td>AD1</td>
<td>MU1</td>
<td>ST1</td>
</tr>
<tr>
<td>LD2</td>
<td>AD2</td>
<td>MU2</td>
<td>ST2</td>
</tr>
<tr>
<td>LD3</td>
<td>AD3</td>
<td>MU3</td>
<td>ST3</td>
</tr>
</tbody>
</table>

**Time**

- **Same op @ same time**
  - ARRAY: LD0, LD1, LD2, LD3
  - VECTOR: LD0

- **Different ops @ time**
  - ARRAY: AD0, AD1, AD2, AD3
  - VECTOR: AD0, AD1

- **Different ops @ same space**
  - ARRAY: MU0, MU1, MU2, MU3
  - VECTOR: MU0, MU1

- **Same op @ space**
  - ARRAY: ST0, ST1, ST2, ST3
  - VECTOR: ST0, ST1
Vector Instruction Execution

VADD A,B → C

Execution using one pipelined functional unit


Execution using four pipelined functional units


Slide credit: Krste Asanovic
Vector Unit Structure

**Partitioned Vector Registers**

- Elements 0, 4, 8, ...
- Elements 1, 5, 9, ...
- Elements 2, 6, 10, ...
- Elements 3, 7, 11, ...

**Functional Unit**

**Memory Subsystem**

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
Automatic Code Vectorization

Scalar Sequential Code

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

Vectorized Code

```
Vector Instruction
```

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

- Idea: One instruction operates on multiple data elements simultaneously
  - Ala 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.


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).
MMX Example: Image Overlaying (I)

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

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

Figure 9. Generating the selection bit mask.
MMX Example: Image Overlaying (II)

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

```
Movq    mm3, mem1   /* Load eight pixels from
Nomvq   mm4, mem2   /* Load eight pixels from the
Pcmpeqb mm1, mm3   blossom image
Pand    mm4, mm1
Pandn   mm1, mm3
Pmmulh  mm4, mm1
```

Figure 11. MMX code sequence for performing a conditional select.
GPUs (Graphics Processing Units)
GPUs are SIMD Engines Underneath

- The instruction pipeline operates like a SIMD pipeline (e.g., an array processor)

- However, the programming is done using threads, NOT SIMD instructions

- To understand this, let’s go back to our parallelizable code example

- But, before that, let’s distinguish between
  - Programming Model (Software)
  - Execution Model (Hardware)
Programming Model vs. Hardware Execution Model

- Programming Model refers to **how the programmer expresses the code**
  - E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), ...

- Execution Model refers to **how the hardware executes the code underneath**
  - E.g., Out-of-order execution, Vector processor, Array processor, Dataflow processor, Multiprocessor, Multithreaded processor, ...

- Execution Model can be very different from the Programming Model
  - E.g., von Neumann model implemented by an OoO processor
  - E.g., SPMD model implemented by a SIMD processor (a GPU)
How Can You Exploit Parallelism Here?

Scalar Sequential Code

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

Let’s examine three programming options to exploit instruction-level parallelism present in this sequential code:

1. Sequential (SISD)
2. Data-Parallel (SIMD)
3. Multithreaded (MIMD/SPMD)
Prog. Model 1: Sequential (SISD)

Scalar Sequential Code

- Can be executed on a:
  - Pipelined processor
  - Out-of-order execution processor
    - Independent instructions executed when ready
    - Different iterations are present in the instruction window and can execute in parallel in multiple functional units
    - In other words, the loop is dynamically unrolled by the hardware
  - Superscalar or VLIW processor
    - Can fetch and execute multiple instructions per cycle

```c
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```
**Prog. Model 2: Data Parallel (SIMD)**

For \( i = 0; i < N; i++ \)

\[ C[i] = A[i] + B[i]; \]

---

**Scalar Sequential Code**

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

---

**Vector Instruction**

```
VLD A \rightarrow V1
VLD B \rightarrow V2
VADD V1 + V2 \rightarrow V3
VST V3 \rightarrow C
```

---

**Vectorized Code**

**Realization:** Each iteration is independent

**Idea:** Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

**Best executed by a SIMD processor (vector, array)**
Prog. Model 3: Multithreaded

Scalar Sequential Code

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

Realization: Each iteration is independent

Idea: Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data)

Can be executed on a MIMD machine
Prog. Model 3: Multithreaded

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

**Idea:** Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data).

This particular model is also called:

**SPMD: Single Program Multiple Data**

Can be executed on a SIMT machine

**Single Instruction Multiple Thread**
A GPU is a SIMD (SIMT) Machine

- Except it is not programmed using SIMD instructions

- It is programmed using threads (SPMD programming model)
  - Each thread executes the same code but operates a different piece of data
  - Each thread has its own context (i.e., can be treated/restarted/executed independently)

- A set of threads executing the same instruction are dynamically grouped into a warp (wavefront) by the hardware
  - A warp is essentially a SIMD operation formed by hardware!
**SPMD on SIMT Machine**

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

Warp: A set of threads that execute the same instruction (i.e., at the same PC)

This particular model is also called:

**SPMD**: Single Program Multiple Data

A GPU executes it using the SIMT model:

**SIMT**: Single Instruction Multiple Thread
Graphics Processing Units
SIMD not Exposed to Programmer (SIMT)
**SIMD vs. SIMT Execution Model**

- **SIMD**: A single *sequential instruction stream* of **SIMD instructions** → each instruction specifies multiple data inputs
  - \([VLD, VLD, VADD, VST], VLEN\)

- **SIMT**: *Multiple instruction streams* of **scalar instructions** → threads grouped dynamically into warps
  - \([LD, LD, ADD, ST], \text{NumThreads}\)

**Two Major SIMT Advantages:**
- **Can treat each thread separately** → i.e., can execute each thread independently (on any type of scalar pipeline) → **MIMD processing**
- **Can group threads into warps flexibly** → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of **SIMD processing**
Multithreading of Warps

- Assume a warp consists of 32 threads
- If you have 32K iterations, and 1 iteration/thread → 1K warps
- Warps can be interleaved on the same pipeline → Fine grained multithreading of warps

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

```
load       load       load
Iter. 18*32 + 1  Iter. 20*32 + 2
```

```
Warp 0 at PC X
Warp 20 at PC X+2
```
Warp: A set of threads that execute the same instruction (on different data elements) \(\rightarrow\) SIMT (Nvidia-speak)

All threads run the same code

Warp: The threads that run lengthwise in a woven fabric ...
High-Level View of a GPU
Latency Hiding via Warp-Level FGMT

- **Warp**: A set of threads that execute the same instruction (on different data elements)

- **Fine-grained multithreading**
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies

- **Register values of all threads stay in register file**

- **FGMT enables long latency tolerance**
  - Millions of pixels

---

Slide credit: Tor Aamodt
Warp Execution (Recall the Slide)

32-thread warp executing ADD A[tid], B[tid] → C[tid]

Execution using one pipelined functional unit


C[2]
C[1]
C[0]

Execution using four pipelined functional units


A[22] B[22]

A[27] B[27]

C[8]
C[4]
C[0]

C[9]
C[5]

C[10]
C[6]

C[11]
C[7]

C[3]
SIMD Execution Unit Structure

- **Functional Unit**
- **Memory Subsystem**
- **Register for each Thread**
- **Lane**

- Registers for thread IDs 0, 4, 8, ...
- Registers for thread IDs 1, 5, 9, ...
- Registers for thread IDs 2, 6, 10, ...
- Registers for thread IDs 3, 7, 11, ...

Slide credit: Krste Asanovic
Warp Instruction Level Parallelism

Can overlap execution of multiple instructions
- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle while issuing 1 warp/cycle
Same instruction in different threads uses thread id to index and access different data elements

Let’s assume $N=16$, 4 threads per warp $\rightarrow$ 4 warps

Slide credit: Hyesoon Kim
Sample GPU SIMT Code (Simplified)

CPU code

```c
for (ii = 0; ii < 100000; ++ii) {
}
```

CUDA code

```c
// there are 100000 threads
__global__ void KernelFunction(...) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    int varA = aa[tid];
    int varB = bb[tid];
    C[tid] = varA + varB;
}
```

Slide credit: Hyesoon Kim
Sample GPU Program (Less Simplified)

**CPU Program**

```c
void add_matrix
  ( float *a, float* b, float *c, int N) {
    int index;
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j) {
        index = i + j*N;
        c[index] = a[index] + b[index];
      }
}

int main () {
  add_matrix (a, b, c, N);
}
```

**GPU Program**

```c
__global__ add_matrix
  ( float *a, float *b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if (i < N && j < N)
      c[index] = a[index]+b[index];
}

int main() {
  dim3 dimBlock( blocksize, blocksize) ;
  dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```
Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
  - Sequential instruction execution; lock-step operations in a SIMD instruction
  - Programming model is SIMD (no extra threads) → SW needs to know vector length
  - ISA contains vector/SIMD instructions

- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables multithreading and flexible dynamic grouping of threads
  - ISA is scalar → SIMD operations can be formed dynamically
  - Essentially, it is SPMD programming model implemented on SIMD hardware
**SPMD**

- Single procedure/program, multiple data
  - This is a programming model rather than computer organization

- Each processing element executes the same procedure, except on different data elements
  - Procedures can synchronize at certain points in program, e.g. barriers

- Essentially, *multiple instruction streams execute the same program*
  - Each program/procedure 1) works on different data, 2) can execute a different control-flow path, at run-time
  - Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD hardware
SIMD vs. SIMT Execution Model

- **SIMD**: A single *sequential instruction stream* of SIMD instructions → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

- **SIMT**: *Multiple instruction streams* of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

**Two Major SIMT Advantages:**
- Can treat each thread separately → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
- Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing
Threads Can Take Different Paths in Warp-based SIMD

- Each thread can have conditional control flow instructions
- Threads can execute different control flow paths
Control Flow Problem in GPUs/SIMT

- A GPU uses a SIMD pipeline to save area on control logic.
  - Groups scalar threads into warps

- Branch divergence occurs when threads inside warps branch to different execution paths.

This is the same as conditional/predicated/masked execution. Recall the Vector Mask and Masked Vector Operations?
Remember: Each Thread Is Independent

- Two Major SIMT Advantages:
  - Can treat each thread separately → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
  - Can group threads into warps flexibly → i.e., can group threads that are supposed to truly execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

- If we have many threads
  - We can find individual threads that are at the same PC
  - And, group them together into a single warp dynamically
  - This reduces “divergence” → improves SIMD utilization
    - SIMD utilization: fraction of SIMD lanes executing a useful operation (i.e., executing an active thread)
Dynamic Warp Formation/Merging

- Idea: **Dynamically merge threads executing the same instruction (after branch divergence)**
- Form new warps from warps that are waiting
  - Enough threads branching to each path enables the creation of full new warps
Dynamic Warp Formation/Merging

- **Idea:** Dynamically merge threads executing the same instruction (after branch divergence)

Dynamic Warp Formation Example

A new warp created from scalar threads of both Warp x and y executing at Basic Block D

Legend

Execution of Warp x at Basic Block A
Execution of Warp y at Basic Block A

A new warp created from scalar threads of both Warp x and y executing at Basic Block D

Baseline

Dynamic Warp Formation

Slide credit: Tor Aamodt
Hardware Constraints Limit Flexibility of Warp Grouping

Can you move any thread flexibly to any lane?
An Example GPU
NVIDIA GeForce GTX 285

- NVIDIA-speak:
  - 240 stream processors
  - “SIMT execution”

- Generic speak:
  - 30 cores
  - 8 SIMD functional units per core
NVIDIA GeForce GTX 285 “core”

- SIMD functional unit, control shared across 8 units
  - multiply-add
  - multiply

- Instruction stream decode

- Execution context storage

64 KB of storage for thread contexts (registers)

Slide credit: Kayvon Fatahalian
NVIDIA GeForce GTX 285 “core”

- Groups of 32 threads share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

Slide credit: Kayvon Fatahalian
NVIDIA GeForce GTX 285

30 cores on the GTX 285: 30,720 threads

Slide credit: Kayvon Fatahalian
Evolution of NVIDIA GPUs
NVIDIA V100

- NVIDIA-speak:
  - 5120 stream processors
  - “SIMT execution”

- Generic speak:
  - 80 cores
  - 64 SIMD functional units per core
  - Tensor cores for Machine Learning
NVIDIA V100 Block Diagram

80 cores on the V100
NVIDIA V100 Core

15.7 TFLOPS Single Precision
7.8 TFLOPS Double Precision
125 TFLOPS for Deep Learning (Tensor cores)

https://devblogs.nvidia.com/inside-volta/
Design of Digital Circuits
Lecture 20: SIMD Processors

Prof. Onur Mutlu
ETH Zurich
Spring 2018
11 May 2018
Approaches to (Instruction-Level) Concurrency

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

- Goal: design an accelerator that has
  - Simple, regular design (keep # unique parts small and regular)
  - High concurrency $\rightarrow$ high performance
  - Balanced computation and I/O (memory) bandwidth

- Idea: Replace a single processing element (PE) with a regular array of PEs and carefully orchestrate flow of data between the PEs
  - such that they collectively transform a piece of input data before outputting it to memory

- Benefit: Maximizes computation done on a single piece of data element brought from memory
Systolic Arrays

Why Systolic Architectures?

- **Idea:** Data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory.

- Similar to **blood flow:** heart → many cells → heart
  - Different cells “process” the blood
  - Many veins operate simultaneously
  - Can be many-dimensional

- **Why?** Special purpose accelerators/architectures need
  - Simple, regular design (keep # unique parts small and regular)
  - High concurrency → high performance
  - Balanced computation and I/O (memory) bandwidth
Systolic Architectures

- Basic principle: Replace a single PE with a regular array of PEs and carefully orchestrate flow of data between the PEs
  - Balance computation and memory bandwidth

- Differences from pipelining:
  - These are individual PEs
  - Array structure can be non-linear and multi-dimensional
  - PE connections can be multidirectional (and different speed)
  - PEs can have local memory and execute kernels (rather than a piece of the instruction)
Systolic Computation Example

- Convolution
  - Used in filtering, pattern matching, correlation, polynomial evaluation, etc ...
  - Many image processing tasks

Given the sequence of weights \( \{w_1, w_2, \ldots, w_k\} \)
and the input sequence \( \{x_1, x_2, \ldots, x_n\} \),

compute the result sequence \( \{y_1, y_2, \ldots, y_{n+1-k}\} \)
defined by

\[
y_i = w_1 x_i + w_2 x_{i+1} + \cdots + w_k x_{i+k-1}
\]
Systolic Computation Example: Convolution

- \( y_1 = w_1x_1 + w_2x_2 + w_3x_3 \)
- \( y_2 = w_1x_2 + w_2x_3 + w_3x_4 \)
- \( y_3 = w_1x_3 + w_2x_4 + w_3x_5 \)

Figure 8. Design W1: systolic convolution array (a) and cell (b) where \( w_i \)'s stay and \( x_i \)'s and \( y_i \)'s move systolically in opposite directions.
Systolic Computation Example: Convolution

- Worthwhile to implement adder and multiplier separately to allow overlapping of add/mul executions

Figure 10. Overlapping the executions of multiply and add in design W1.
Systolic Computation Example: Convolution

- One needs to carefully orchestrate when data elements are input to the array
- And when output is buffered

- This gets more involved when
  - Array dimensionality increases
  - PEs are less predictable in terms of latency
Systolic Arrays: Pros and Cons

- **Advantage:**
  - Specialized (computation needs to fit PE organization/functions)
    - improved efficiency, simple design, high concurrency/performance
    - good to do more with less memory bandwidth requirement

- **Downside:**
  - Specialized
    - not generally applicable because computation needs to fit the PE functions/organization
More Programmability in Systolic Arrays

- Each PE in a systolic array
  - Can store multiple “weights”
  - Weights can be selected on the fly
  - Eases implementation of, e.g., adaptive filtering

- Taken further
  - Each PE can have its own data and instruction memory
  - Data memory → to store partial/temporary results, constants
  - Leads to stream processing, pipeline parallelism
    - More generally, staged execution
Pipeline-Parallel (Pipelined) Programs

Figure 1. (a) The code of a loop. (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises Ai, Bi, Ci. (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.
Stages of Pipelined Programs

- Loop iterations are divided into code segments called *stages*
- Threads execute stages on different cores

```plaintext
loop {
    Compute1
    Compute2
    Compute3
}
```
Pipelined File Compression Example

Figure 3. File compression algorithm executed using pipeline parallelism
Systolic Array: Advantages & Disadvantages

- Advantages
  - Makes multiple uses of each data item → reduced need for fetching/refetching
  - High concurrency
  - Regular design (both data and control flow)

- Disadvantages
  - Not good at exploiting irregular parallelism
  - Relatively special purpose → need software, programmer support to be a general purpose model
Example Systolic Array: The WARP Computer

- HT Kung, CMU, 1984-1988

- Linear array of 10 cells, each cell a 10 Mflop programmable processor
- Attached to a general purpose host machine
- HLL and optimizing compiler to program the systolic array
- Used extensively to accelerate vision and robotics tasks

The WARP Computer

Figure 1: Warp system overview
Figure 2: Warp cell data path
An Example Modern Systolic Array

Figure 3. TPU Printed Circuit Board. It can be inserted in the slot for an SATA disk in a server, but the card uses PCIe Gen3 x16.

Figure 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.

An Example Modern Systolic Array

As reading a large SRAM uses much more power than arithmetic, the matrix unit uses systolic execution to save energy by reducing reads and writes of the Unified Buffer [Kun80][Ram91][Ovt15b]. Figure 4 shows that data flows in from the left, and the weights are loaded from the top. A given 256-element multiply-accumulate operation moves through the matrix as a diagonal wavefront. The weights are preloaded, and take effect with the advancing wave alongside the first data of a new block. Control and data are pipelined to give the illusion that the 256 inputs are read at once, and that they instantly update one location of each of 256 accumulators. From a correctness perspective, software is unaware of the systolic nature of the matrix unit, but for performance, it does worry about the latency of the unit.

An Example Modern Systolic Array

Figure 1. TPU Block Diagram. The main computation part is the yellow Matrix Multiply unit in the upper right hand corner. Its inputs are the blue Weight FIFO and the blue Unified Buffer (UB) and its output is the blue Accumulators (Acc). The yellow Activation Unit performs the nonlinear functions on the Acc, which go to the UB.
Decoupled Access/Execute (DAE)
Decoupled Access/Execute (DAE)

- Motivation: Tomasulo’s algorithm too complex to implement
  - 1980s before Pentium Pro

- Idea: Decouple operand access and execution via two separate instruction streams that communicate via ISA-visible queues.

Decoupled Access/Execute (II)

- Compiler generates two instruction streams (A and E)
  - Synchronizes the two upon control flow instructions (using branch queues)

\[ q = 0.0 \]
\[ \text{Do } k = 1, 400 \]
\[ x(k) = q + y(k) \times (r \times z(k+10) + t \times z(k+11)) \]

Fig. 2a. Lawrence Livermore Loop 1 (HYDRO EXCEPT)

\[
\begin{align*}
A7 &+ -400 & \text{. negative loop count} \\
A2 &+ 0 & \text{. initialize index} \\
A3 &+ 1 & \text{. index increment} \\
X2 &+ r & \text{. load loop invariants} \\
X5 &+ t & \text{. into registers} \\
\text{loop:} & \quad X3 &+ z + 10, A2 & \text{. load } z(k+10) \\
& & X7 &+ z + 11, A2 & \text{. load } z(k+11) \\
& & X4 &+ X2 \times f X3 & \text{. } r \times z(k+10) \text{-flt. mult.} \\
& & X3 &+ X5 \times f X7 & t \times z(k+11) \\
& & X7 &+ y, A2 & \text{. load } y(k) \\
& & X6 &+ X3 \times f X4 & \text{. } r \times z(x+10) + t \times z(k+11) \\
& & X4 &+ X7 \times f X6 & y(k) \times \text{(above)} \\
& & A7 &+ A7 + 1 & \text{. increment loop counter} \\
& & x, A2 &+ X4 & \text{. store into } x(k) \\
& & A2 &+ A2 + A3 & \text{. increment index} \\
& & \text{JAM loop} & \text{. Branch if } A7 < 0 \\
\end{align*}
\]

Access: \[ \begin{align*} 
& \text{AEQ + } z + 10, A2 \quad X4 + X2 \times f \text{ AEQ} \\
& \text{AEQ + } z + 11, A2 \quad X3 + X5 \times f \text{ AEQ} \\
& \text{AEQ + } y, A2 \quad X6 + X3 \times f X4 \\
& \text{A7 + A7 + 1} \quad \text{EAQ + AEQ } \times f X6 \\
& \text{x, A2 + EAQ} \quad \text{.} \\
& \text{A2 + A2 + A3} \quad \text{.} \\
& \text{.} \quad \text{.} \\
& \text{.} \quad \text{.} \\
\end{align*} \]

Execute:

Fig. 2b. Compilation onto CRAY-1-like architecture

Fig. 2c. Access and execute programs for straight-line section of loop
Decoupled Access/Execute (III)

- Advantages:
  + Execute stream can run ahead of the access stream and vice versa
    + If A takes a cache miss, E can perform useful work
    + If A hits in cache, it supplies data to lagging E
    + Queues reduce the number of required registers
  + Limited out-of-order execution without wakeup/select complexity

- Disadvantages:
  -- Compiler support to partition the program and manage queues
    -- Determines the amount of decoupling
  -- Branch instructions require synchronization between A and E
  -- Multiple instruction streams (can be done with a single one, though)
Astronautics ZS-1

- Single stream steered into A and X pipelines
- Each pipeline in-order


Loop Unrolling to Eliminate Branches

- **Idea:** Replicate loop body multiple times within an iteration
  - Reduces loop maintenance overhead
    - Induction variable increment or loop condition test
  - Enlarges basic block (and analysis scope)
    - Enables code optimization and scheduling opportunities
- What if iteration count not a multiple of unroll factor? (need extra code to detect this)
- Increases code size
A Modern DAE Example: Pentium 4

Intel Pentium 4 Simplified

Mutlu+, “Runahead Execution,”
HPCA 2003.
Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays
We Are Now Done With This…

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Other Execution Paradigms
An Example GPU Exercise

We define the *SIMD utilization* of a program run on a GPU as the fraction of SIMD lanes that are kept busy with *active threads* during the run of a program.

The following code segment is run on a GPU. Each thread executes a **single iteration** of the shown loop. Assume that the data values of the arrays A, B, and C are already in vector registers so there are no loads and stores in this program. (Hint: Notice that there are 4 instructions in each thread.) A warp in the GPU consists of 64 threads, and there are 64 SIMD lanes in the GPU.

```c
for (i=0; i<1,024,768; i++) {
    if (A[i] > 0) {
        A[i] = A[i] * C[i];
        B[i] = A[i] + B[i];
        C[i] = B[i] + 1;
    }
}
```

(a) How many warps does it take to execute this program?
(b) When we measure the SIMD utilization for this program with one input set, we find that it is 67/256. What can you say about arrays A, B, and C? Be precise.

A:

B:

C:
An Example GPU Exercise (III)

(c) Is it possible for this program to yield a SIMD utilization of 100% (circle one)?

YES                      NO

If YES, what should be true about arrays A, B, C for the SIMD utilization to be 100%? Be precise.

A:

B:

C:

If NO, explain why not.
An Example GPU Exercise (IV)

(d) Is it possible for this program to yield a SIMD utilization of 25% (circle one)?

YES        NO

If YES, what should be true about arrays A, B, and C for the SIMD utilization to be 25%? Be precise.

A:

B:

C:

If NO, explain why not.
Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Fine-Grained Multithreading
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays
We Are Now Done With This…

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Other Execution Paradigms