Digital Design & Computer Arch.
Lecture 19: SIMD Architectures

Prof. Onur Mutlu

ETH Zürich
Spring 2023
5 May 2023
Other Execution Paradigms

- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Systolic Arrays
- Decoupled Access Execute

- **SIMD Architectures (Vector and Array processors)**

- Graphics Processing Units (GPUs)
Readings for this Week

- **Required**

- **Recommended**
Optional Lecture: Decoupled Access/Execute

Digital Design & Computer Arch.
Lecture 18c: Decoupled Access-Execute

Prof. Onur Mutlu
ETH Zürich
Spring 2023
4 May 2023

https://www.youtube.com/watch?v=6gh7tWf3xgM&list=PL5Q2soXY2Zi-ElmKxYY1SZuGiOAOBKaf&index=23
Wrap Up: Systolic Arrays
Lecture 18b: Systolic Array Architectures

Systolic Computation Example: Convolution (II)

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

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.
An Example Modern Systolic Array: TPU (I)

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: TPU (II)

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.

Recall: Example 2D Systolic Array Computation

- Multiply two 3x3 matrices (inputs)
  - Keep the final result in PE accumulators

\[
\begin{bmatrix}
c_{00} & c_{01} & c_{02} \\
c_{10} & c_{11} & c_{12} \\
c_{20} & c_{21} & c_{22}
\end{bmatrix}
= \begin{bmatrix}
a_{00} & a_{01} & a_{02} \\
a_{10} & a_{11} & a_{12} \\
a_{20} & a_{21} & a_{22}
\end{bmatrix}
\times \begin{bmatrix}
b_{00} & b_{01} & b_{02} \\
b_{10} & b_{11} & b_{12} \\
b_{20} & b_{21} & b_{22}
\end{bmatrix}
\]

Figure 1: A systolic array processing element

\[P = M\]
\[Q = N\]
\[R = R + M \times N\]
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.
An Example Modern Systolic Array: TPU2

4 TPU chips vs 1 chip in TPU1

High Bandwidth Memory vs DDR3

Floating point operations vs FP16

45 TFLOPS per chip vs 23 TOPS

Designed for training and inference vs only inference

An Example Modern Systolic Array: TPU3

- **32GB HBM per chip** vs **16GB HBM in TPU2**
- **4 Matrix Units per chip** vs **2 Matrix Units in TPU2**
- **90 TFLOPS per chip** vs **45 TFLOPS in TPU2**
An Example Modern Systolic Array: TPU4

New ML applications (vs. TPU3):
- Computer vision
- Natural Language Processing (NLP)
- Recommender system
- Reinforcement learning that plays Go

250 TFLOPS per chip in 2021 vs 90 TFLOPS in TPU3

1 ExaFLOPS per board

https://spectrum.ieee.org/tech-talk/computing/hardware/heres-how-googles-tpu-v4-ai-chip-stacked-up-in-training-tests
Ten Lessons From Three Generations Shaped Google’s TPUv4i

Industrial Product

Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson, Google LLC
A Recent Reading on the TPUv4

TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings

Industrial Product*

Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson

Google, Mountain View, CA

An Analysis of the Google Edge TPU

- Amirali Boroumand, Saugata Ghose, Berkin Akin, Ravi Narayanaswami, Geraldo F. Oliveira, Xiaoyu Ma, Eric Shiu, and Onur Mutlu,
  "Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks"
  Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques (PACT), Virtual, September 2021.
  [Slides (pptx) (pdf)]
  [Talk Video (14 minutes)]

> 90% of the total system energy is spent on **memory** in large ML models
Other Machine Learning Accelerators
Cerebras’s Wafer Scale Engine (2019)

- The largest ML accelerator chip
- 400,000 cores

Cerebras WSE
1.2 Trillion transistors
46,225 mm²

Largest GPU
21.1 Billion transistors
815 mm²

NVIDIA TITAN V

https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning
https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/
Cerebras’s Wafer Scale Engine-2 (2021)

- The largest ML accelerator chip
- 850,000 cores

**Cerebras WSE-2**
- 2.6 Trillion transistors
- 46,225 mm²

**Largest GPU**
- 54.2 Billion transistors
- 826 mm²

[https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/](https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/)
Huge Demand for Performance & Efficiency

Exponential Growth of Neural Networks

1800x more compute
In just 2 years

Tomorrow, multi-trillion parameter models

https://www.youtube.com/watch?v=x2-qB0J7KHW
More on the Cerebras WSE

https://www.youtube.com/watch?v=x2-qB0J7KHz
Microsoft Brainwave FPGA Acceleration

Serving DNNs in Real Time at Datacenter Scale with Project Brainwave

To meet the computational demands required of deep learning, cloud operators are turning toward specialized hardware for improved efficiency and performance. Project Brainwave, Microsoft’s principal infrastructure for AI serving in real time, accelerates deep neural network (DNN) inferencing in major services such as Bing’s intelligent search features and Azure. Exploiting distributed model parallelism and pinning over low-latency hardware microservices, Project Brainwave serves state-of-the-art, pre-trained DNN models with high efficiencies at low batch sizes. A high-performance, precision-adaptable FPGA soft processor is at the heart of the system, achieving up to 39.5 teraflops (Tflops) of effective performance at Batch 1 on a state-of-the-art Intel Stratix 10 FPGA.

The recent successes of machine learning, enabled largely by the rise of DNNs, have fueled a growing demand for ubiquitous AI, ranging from conversational agents to object recognition to intelligent search. While state-of-the-art DNNs continue to deliver major breakthroughs in challenging AI domains such as computer vision and natural language processing, their computational demands have steadily outpaced the performance growth rate of standard CPUs. These trends have spurred a

Figure 3. The framework-neutral Brainwave tool flow accepts models from different DNN toolchains and exports them into a common intermediate graph representation. Tool flow optimizes the intermediate representation and partitions it into sub-graphs assigned to different CPUs and FPGAs. Device-specific backends generate device assembly and are linked together by a federated runtime that gets deployed into a live FPGA hardware microservice.

Figure 5. An independent SRAM memory port is dedicated to every lane of a multi-lane vector dot product unit within the MVU, allowing up to 80,000 MACs on a Stratix 10 280 to be fed with independent weights. As a result, FPGA can achieve near-peak utilization on Batch 1-oriented matrix-vector multiplication.
Exploiting Data Parallelism: SIMD Architectures 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
Flynn’s Taxonomy of Computers


Very High-Speed Computing Systems

MICHAEL J. FLYNN, MEMBER, IEEE

Abstract—Very high-speed computers may be classified as follows:

1) Single Instruction Stream—Single Data Stream (SISD)
2) Single Instruction Stream—Multiple Data Stream (SIMD)
3) Multiple Instruction Stream—Single Data Stream (MISD)
4) Multiple Instruction Stream—Multiple Data Stream (MIMD).

“Stream,” as used here, refers to the sequence of data or instructions as seen by the machine during the execution of a program.

The constituents of a system: storage, execution, and instruction handling (branching) are discussed with regard to recent developments and/or systems limitations. The constituents are discussed in terms of concurrent SISD systems (CDC 6600 series and, in particular, IBM Model 90 series), since multiple stream organizations usually do not require any more elaborate components.

Representative organizations are selected from each class and the arrangement of the constituents is shown.

Introduction

Many significant scientific problems require the use of prodigious amounts of computing time. In order to handle these problems adequately, the large-scale scientific computer has been developed. This computer addresses itself to a class of problems characterized by having a high ratio of computing requirement to input/output requirements (a partially de facto situation...
MISD Example from Flynn

Similar to a “generalized” systolic array

Systolic Computation Example: Convolution (II)

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

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.
SIMD Example from Flynn

Similar to an “array processor”

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 Paradigm

- Single instruction operates on multiple data elements
  - In time or in space
- Multiple processing elements (PEs), i.e., execution units

- Time-space duality

  - **Array processor**: Instruction operates on multiple data elements at the same time using different spaces (PEs)

  - **Vector processor**: Instruction operates on multiple data elements in consecutive time steps using the same space (PE)
Storing Multiple Data Elements: Vector Registers

- Each vector data register holds $N$ $M$-bit values
  - Each register stores a vector
  - Not a (single) scalar value as we saw before
Array vs. Vector Processors

Instruction Stream

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

Time

ARRAY PROCESSOR

Same op @ same time

VECTOR PROCESSOR

Different ops @ time

Different ops @ same space

Same op @ space

Space
SIMD Array Processing vs. VLIW

- VLIW: **Multiple** independent **operations** packed together into a “long inst.”

![Diagram of VLIW execution](image-url)
SIMD Array Processing vs. VLIW

- Array processor: **Single operation** on multiple (different) data elements

```
Program Counter

add VR, VR, 1
```

- VLEN = 4

```
add VR[0],VR[0],1  add VR[1],VR[1],1  add VR[2],VR[2],1  add VR[3],VR[3],1
```

Instruction Execution

```
PE
PE
PE
PE
```
Lecture 18a: VLIW Architectures

VLIW Concept

- ELI: Enormously longword instructions (512 bits)
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 in memory between two elements of a vector
Vector Stride Example: Matrix Multiply

- A and B matrices, both stored in memory in row-major order.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>B₀</td>
<td>C₀</td>
</tr>
<tr>
<td>A₁</td>
<td>B₁</td>
<td>C₁</td>
</tr>
<tr>
<td>A₂</td>
<td>B₂</td>
<td>C₂</td>
</tr>
<tr>
<td>A₃</td>
<td>B₃</td>
<td>C₃</td>
</tr>
<tr>
<td>A₄</td>
<td>B₄</td>
<td>C₄</td>
</tr>
<tr>
<td>A₅</td>
<td>B₅</td>
<td>C₅</td>
</tr>
</tbody>
</table>

A₄×6  B₆×10 → C₄×10

Dot product of each row vector of A with each column vector of B.

- Load A’s row 0 (A0 through A5) into vector register V₁
  - Each time, increment address by 1 to access the next column
  - Accesses have a stride of 1

- Load B’s column 0 (B0 through B50) into vector register V₂
  - Each time, increment address by 10 to access the next row
  - Accesses have a stride of 10
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 $\rightarrow$ no hardware interlocking needed within a vector
  - No control flow within a vector
  - Known stride allows easy address calculation for all vector elements
    - Enables easy loading (or even early loading, i.e., 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 (without the penalty of deep pipelines)

+ Each instruction generates a lot of work (i.e., operations)
  - Reduces instruction fetch bandwidth requirements
  - Amortizes instruction fetch and control overhead over many data
    --> Leads to high energy efficiency per operation

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

+ Highly regular memory access pattern
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

**Recommended Paper: VLIW**

**VERY LONG INSTRUCTION WORD ARCHITECTURES AND THE ELI-512**

JOSEPH A. FISHER
YALE UNIVERSITY
NEW HAVEN, CONNECTICUT 06520

**ABSTRACT**

By compiling ordinary scientific applications programs with a radical technique called trace scheduling, we are generating code for a parallel machine that will run these programs faster than an equivalent sequential machine — we expect 10 to 30 times faster.

Trace scheduling generates code for machines called Very Long Instruction Word architectures. In Very Long Instruction Word machines, many statically scheduled, tightly coupled, fine-grained operations execute in parallel within a single instruction stream. VLIWs are more parallel extensions of several current architectures.

These current architectures have never cracked a fundamental barrier. The speedup they get from parallelism is never more than a factor of 2 to 3. Not that we couldn't build more parallel machines of this type; but until trace scheduling we didn't know how to generate code for them. Trace scheduling finds sufficient parallelism in ordinary code to justify thinking about a highly parallel VLIW.

At Yale we are actually building one. Our machine, the ELI-512, has a horizontal instruction word of over 500 bits and will be a 42 RISC level instruction word. (It has 321.

are presented in this paper. How do we put enough tests in each cycle without making the machine too big? How do we put enough memory references in each cycle without making the machine too slow?

**WHAT IS A VLIW?**

Everyone wants to use cheap hardware in parallel to speed up computation. One obvious approach would be to take your favorite Reduced Instruction Set Computer, let it be capable of executing 10 to 30 RISC-level operations per cycle controlled by a very long instruction word. (In fact, call it a VLIW.) A VLIW looks like very parallel horizontal microcode.

More formally, VLIW architectures have the following properties:

- There is one central control unit issuing a single long instruction per cycle.
- Each long instruction consists of many tightly coupled independent operations.
- Each operation requires a small, statically predictable number of cycles to execute.
- Operations can be pipelined. These properties distinguish
Amdahl’s Law

- Amdahl’s Law
  - $f$: Parallelizable fraction of a program
  - $N$: Number of processors

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


- Maximum speedup limited by serial portion: Serial bottleneck

- All parallel machines “suffer from” the serial bottleneck
Validity of the single processor approach to achieving large scale computing capabilities

by DR. GENE M. AMDAHL
International Business Machines Corporation
Sunnyvale, California

INTRODUCTION
For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution. Various the proper direction has been pointed out as general purpose computers with a generalized interconnection of memories, or as specialized computers with geometrically related memory interconnections and controlled by one or more instruction streams.

Demonstration is made of the continued validity of the single processor approach and of the weaknesses of the multiple processor approach in terms of application to real problems and their attendant irregularities.

The arguments presented are based on statistical characteristics of computation on computers over the last decade and upon the operational requirements within problems of physical interest. An additional processing rate, even if the housekeeping were done in a separate processor. The non-housekeeping part of the problem could exploit at most a processor of performance three to four times the performance of the housekeeping processor. A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.

Data management housekeeping is not the only problem to plague oversimplified approaches to high speed computation. The physical problems which are of practical interest tend to have rather significant complications. Examples of these complications are as follows: boundaries are likely to be irregular; interiors are likely to be inhomogeneous; computations required may be dependent on the states of the variables at each point; propagation rates of different physical effects may be quite different; the
Caveats of Parallelism

- **Amdahl’s Law**
  - f: Parallelizable fraction of a program
  - N: Number of processors
  
  \[
  \text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}
  \]


- Maximum speedup limited by serial portion: **Serial bottleneck**
- Parallel portion is usually not perfectly parallel
  - **Synchronization** overhead (e.g., updates to shared data)
  - **Load imbalance** overhead (imperfect parallelization)
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., \( \text{VMASK}[i] = (V_k[i] == 0) \)
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

Six stage multiply pipeline

\[ V_1 \times V_2 \rightarrow V_3 \]
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
Recommended Paper


The CRAY-1 Computer System

Richard M. Russell
Cray Research, Inc.

This paper describes the CRAY-1, discusses the evolution of its architecture, and gives an account of some of the problems that were overcome during its manufacture.

The CRAY-1 is the only computer to have been built to date that satisfies ERDA's Class VI requirement (a computer capable of processing from 20 to 60 million floating point operations per second) [1].

The CRAY-1's Fortran compiler (CFT) is designed to give the scientific user immediate access to the benefits of the CRAY-1's vector processing architecture. An optimizing compiler, CFT, "vectorizes" innermost DO loops. Compatible with the ANSI 1966 Fortran Standard and with many commonly supported Fortran extensions, CFT does not require any source program modifications or the use of additional nonstandard Fortran statements to achieve vectorization. Thus the user's investment of hundreds of man months of effort to develop Fortran programs for other contemporary computers is protected.

Key Words and Phrases: architecture, computer systems
CR Categories: 1.2, 6.2, 6.3

Introduction

Vector processors are not yet commonplace machines in the larger-scale computer market. At the time of this writing we know of only 12 non-CRAY-1 vector processor installations worldwide. Of these 12, the most powerful processor is the ILLIAC IV (1 installation), the most populous is the Texas Instruments Advanced Scientific Computer (7 installations) and the most publicized is Control Data's STAR 100
Cray X-MP-28 @ ETH (CAB, E Floor)

Cray X-MP-28


Für den Betrieb war Cray Research verantwortlich. Zwei der Mitarbeiter waren Corvette, zwei für die Programmierung und Administration.


Miniaturisierung und explodierende Leistung


Raspberry Pi 1 model B

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.

Model Number of CPUs Memory size (millions of 64-bit words) Number of banks
--- --- --- --- ---
CRAY X-MP/416 4 16 64
CRAY X-MP/48 4 8 32
CRAY X-MP/216 2 16 32
CRAY X-MP/28 2 8 32
CRAY X-MP/24 2 4 16
CRAY X-MP/18 1 4 16
CRAY X-MP/14 1 2 16
CRAY X-MP/12 1 1 16
CRAY X-MP/11 1 1 16

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 multiprotocol 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 gigabits, 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>

## 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, Leader in Supercomputer Design

"If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?"
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 reduce memory chip pins)
- Can start and complete one bank access per cycle
- Can sustain N concurrent accesses if all N go to different banks

```
Bank 0  Bank 1  Bank 2  Bank 15
M DR  M AR  M DR  M AR  M DR  M AR
```

```
Address bus
Data bus
CPU
```
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 in clock cycles)

```plaintext
MOVI R0 = 50 1
MOVA R1 = A 1
MOVA R2 = B 1
MOVA 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
```

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 1 bank with 2 memory ports (two different memory accesses can be serviced concurrently) OR 2 banks (where arrays B and C are stored in different banks)
  - First two loads in the loop can be pipelined: 1 + 11 cycles
  - $4 + 50*30 = 1504$ cycles
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):

- MOVI VLEN = 50
- MOVI VSTR = 1
- VLD V0 = A
- VLD V1 = B
- VADD V2 = V0 + V1
- VSHFR V3 = V2 >> 1
- VST C = V3

1 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

- 1 memory port (one address generator) per bank

- 16 memory banks (word-interleaved: consecutive elements of an array are stored in consecutive banks)

- 285 cycles
Basic Vector Code Performance (II)

- 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

- The above assumes a unit stride (i.e., stride = 1)
  - Correct for our example program

- What if stride > 1?
  - How do you ensure we can access 1 element per cycle when memory latency is 11 cycles?
Recall: 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
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

- **182 cycles**

---

These two VLDs cannot be pipelined. **WHY?**

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**
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

Source: https://en.wikipedia.org/wiki/Surface_mining
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

- Doing so also helps with avoiding useless computation on sparse vectors (i.e., vectors where many elements are 0)
Want to vectorize loops with indirect accesses:

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

Indexed load instruction (\textit{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) or indirect indexing
- 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)?

```
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

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

- **Storage format 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.
Bank Conflicts in Matrix Multiplication

- A and B matrices, both stored in memory in row-major order

A_4x6 \times B_{6x10} \rightarrow C_{4x10}

Dot product of each row vector of A with each column vector of B

- Load A’s row 0 into vector register V_1
  - Each time, increment address by 1 to access the next column
  - Accesses have a stride of 1

- Load B’s column 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?
Minimizing Bank Conflicts

- More banks
- More ports in each bank
- Better data layout to match the access pattern
  - Is this always possible?
- Better mapping of address to bank
  - E.g., randomized mapping
Recommended Reading: Minimizing Bank Conflicts

PSEUDO-RANDOMLY INTERLEAVED MEMORY

B. Ramakrishna Rau
Hewlett Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303

ABSTRACT

Interleaved memories are often used to provide the high bandwidth needed by multiprocessors and high performance uniprocessors such as vector and VLIW processors. The manner in which memory locations are distributed across the memory modules has a significant influence on whether, and for which types of reference patterns, the full bandwidth of the memory system is achieved. The most common interleaved memory architecture is the sequentially interleaved memory in which successive memory locations are assigned to successive memory modules. Although such an architecture is the simplest to implement and provides good performance with strides that are odd integers, it can degrade badly in the face of even strides, especially strides that are a power of two.

In a pseudo-randomly interleaved memory architecture, memory locations are assigned to the memory modules in some pseudo-random fashion in the hope that those sequences of references, which are likely to occur in practice, will end up being evenly distributed across the memory modules. The notion of polynomial interleaving modulo an irreducible polynomial is introduced as a way of achieving pseudo-random interleaving with certain attractive and provable properties. The theory behind this scheme is developed and the results of simulations are presented.

Keywords: supercomputer memory, parallel memory, interleaved memory, hashed memory, pseudo-random interleaving, memory buffering.

The conventional solution is to provide each processor with a data cache constructed out of SRAM. The problem is maintaining cache coherency, at high request rates, across multiple private caches in a multiprocessor system. The alternative is to use a shared cache if the additional delay incurred in going through the processor-cache interconnect is acceptable. The problem here is that the bandwidth, even with SRAM chips, is inadequate unless some form of interleaving is employed in the cache. So once again, the interleaving scheme used is an issue. Furthermore, data caches are susceptible to problems arising out of the lack of spatial and/or data locality in the data reference pattern of many applications. This phenomenon has been studied and reported elsewhere, e.g., in [4,5]. Since data caches are essential to achieving good performance on scalar computations with little parallelism, the right compromise is to provide a data cache that can be bypassed when referencing data structures with poor locality. This is the solution employed in various recent products such as the Convex C-1 and Intel's i860.

Interleaved memory systems. Whether or not a data cache is present, it is important to provide a memory system with bandwidth to match the processors. This is done by organizing the memory system as multiple memory modules which can operate in parallel. The manner in which memory locations are distributed across the memory modules has a significant influence on whether, and for which types of reference patterns, the full bandwidth of the memory system is achieved.

Engineering and scientific applications include

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 more detail
Recall: Array vs. Vector Processors

Instruction Stream

**ARRAY PROCESSOR**

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

**VECTOR PROCESSOR**

- **LD**
- **ADD**
- **MUL**
- **ST**

**Time**

**Space**

Same op @ same time

Different ops @ time

Different ops @ same space

Same op @ space
Vector Instruction Execution

Execution using one pipelined functional unit:

Execution using four pipelined functional units:
- A[27] B[27]
- A[22] B[22]

Time

Space

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, ...

- **Lane**

- **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
- Example with 24 operations/cycle (steady state) while issuing 1 vector instruction/cycle

Slide credit: Krste Asanovic
Automatic Code Vectorization

Scalar Sequential Code

Vectorized Code

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

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

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
Recall: Amdahl’s Law

- Amdahl’s Law
  - $f$: Parallelizable fraction of a program
  - $N$: Number of processors

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


- Maximum speedup limited by serial portion: Serial bottleneck

- All parallel machines “suffer from” the serial bottleneck
We Covered Until This Point in Lecture
Slides for Future Lectures
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$s0</td>
</tr>
<tr>
<td>a₃</td>
<td>a₂</td>
<td>a₁</td>
<td>a₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b₃</td>
<td>b₂</td>
<td>b₁</td>
<td>b₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$s2</td>
</tr>
<tr>
<td>a₃ + b₃</td>
<td>a₂ + b₂</td>
<td>a₁ + b₁</td>
<td>a₀ + b₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

Peleg and Weiser, “MMX Technology Extension to the Intel Architecture,”
**MMX Example: Image Overlaying (I)**

- **Goal:** Overlay the human in image $x$ on top of the background in image $y$.

![Image overlay](image.png)

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

**PCMPEQW MM1, MM3**

<table>
<thead>
<tr>
<th>MM1</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM3</td>
<td>X7=blue</td>
<td>X6=blue</td>
<td>X5=blue</td>
<td>X4=blue</td>
<td>X3=blue</td>
<td>X2=blue</td>
<td>X1=blue</td>
<td>X0=blue</td>
</tr>
</tbody>
</table>

**Image $x[\ ]$**

**Bit mask**

| MM1 | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF |

Figure 9. Generating the selection bit mask.

MMX Example: Image Overlaying (II)

<table>
<thead>
<tr>
<th>PAND MM4, MM1</th>
<th>Y = Blossom image</th>
<th>PANDN MM1, MM3</th>
<th>X = Woman's image</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM4</td>
<td>( Y_7 ) ( Y_6 ) ( Y_5 ) ( Y_4 ) ( Y_3 ) ( Y_2 ) ( Y_1 ) ( Y_0 )</td>
<td>MM1</td>
<td>( X_7 ) ( X_6 ) ( X_5 ) ( X_4 ) ( X_3 ) ( X_2 ) ( X_1 ) ( X_0 )</td>
</tr>
<tr>
<td>MM1</td>
<td>( 0\times0000 ) ( 0\times0000 ) ( 0\timesFFFF ) ( 0\timesFFFF ) ( 0\times0000 ) ( 0\timesFFFF ) ( 0\timesFFFF ) ( 0\timesFFFF )</td>
<td>MM3</td>
<td>( X_7 ) ( X_6 ) ( X_5 ) ( X_4 ) ( X_3 ) ( X_2 ) ( X_1 ) ( X_0 )</td>
</tr>
<tr>
<td>MM4</td>
<td>( 0\times0000 ) ( 0\times0000 ) ( 0\timesFFFF ) ( 0\timesFFFF ) ( 0\times0000 ) ( 0\timesFFFF ) ( 0\timesFFFF ) ( 0\timesFFFF )</td>
<td>MM1</td>
<td>( X_7 ) ( X_6 ) ( 0\times0000 ) ( 0\times0000 ) ( X_3 ) ( X_2 ) ( 0\times0000 ) ( 0\times0000 )</td>
</tr>
</tbody>
</table>

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

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

**Figure 11.** MMX code sequence for performing 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**
From MMX to AMX in x86 ISA

- **MMX**
  - 64-bit MMX registers for integers

- **SSE (Streaming SIMD Extensions)**
  - SSE-1: 128-bit XMM registers for integers and single-precision floating point
  - SSE-2: Double-precision floating point
  - SSE-3, SSSE-3 (supplemental): New instructions
  - SSE-4: New instructions (not multimedia specific), shuffle operations

- **AVX (Advanced Vector Extensions)**
  - AVX: 256-bit floating point
  - AVX2: 256-bit floating point with FMA (Fused Multiply Add)
  - AVX-512: 512-bit

- **AMX (Advanced Matrix Extensions)**
  - Designed for AI/ML workloads
  - 2-dimensional registers
  - Tiled matrix multiply unit (TMUL)
SIMD Operations in Modern (Machine Learning) Accelerators
Cerebrars’s Wafer Scale Engine (2019)

- The largest ML accelerator chip (2019)
- 400,000 cores

Cerebras WSE
1.2 Trillion transistors
46,225 mm²

Largest GPU
21.1 Billion transistors
815 mm²

https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning
https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/
Cerebras’s Wafer Scale Engine-2 (2021)

- The largest ML accelerator chip (2021)
- 850,000 cores

Cerebras WSE-2
2.6 Trillion transistors
46,225 mm²

Largest GPU
54.2 Billion transistors
826 mm²
NVIDIA Ampere GA100

https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning
https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/
Size, Place, and Route in Cerebras’s WSE

- Neural network mapping onto the whole wafer is a challenge

Multiple possible mappings

Different dies of the wafer work on different layers of the neural network: MIMD machine

James et al., “ISPD 2020 Physical Mapping of Neural Networks on a Wafer-Scale Deep Learning Accelerator.”
Recall: 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
A MIMD Machine with SIMD Processors (I)

- **MIMD** machine
  - Distributed memory (no shared memory)
  - 2D-mesh interconnection fabric

![Diagram of a MIMD machine with SIMD processors](image)

**Fig. 2. CS-1 Wafer Scale Engine (WSE).** A single wafer (rightmost) contains one CS-1 processor. Each processor is a collection of dies arranged in a 2D fashion (middle). Dies are then further subdivided into a grid of tiles. One die hosts thousands of computational cores, memory and routers (leftmost). There is no logical discontinuity between adjacent dies and there is no additional bandwidth penalty for crossing the die-die barrier. In total, there are 1.2 trillion transistors in an area of 462.25 cm$^2$.

- Tensors, making use of tensor address generation hardware to efficiently access tensor data in memory. These play the role of nested loops and eliminate any loop overhead. There are enough memory banks to provide the bandwidth needed to fetch eight 16-bit words from memory and store four such words per cycle, enough to support SIMD-4, AXPY operations $y = y + ax$, where the operand $a$ is a scalar held in a register and $x$ and $y$ are tensors that stream to and from memory. Such an operation can be launched with a single instruction.

  - In mixed precision with multiplications in fp16 and additions performed in fp32, the throughput is two FMACs per core per cycle. Purely 32-bit floating point computations run one FMAC per core per cycle. The theoretical peak performance of the system varies depending on the number of cores configured on the wafer, clock rate and power settings.

- The core connects to a local router that has five bidirectional links, one to each of its four nearest neighbors and one to its own core. The router can move data into and out of these five links, in parallel, on every cycle. Even with scalar granularity, communication is efficient. The router has hardware queues for its connection to the core and for each of a set of virtual channels, avoiding deadlock. Communication between potentially distant processors occurs along predetermined routes. Routing is configured offline, as part of compilation; data travel along virtual channels that can be programmatically reconfigured at runtime. The fanout of data to multiple destinations is done through the routing; the router can forward an input word to any subset of its five output ports. There is no runtime software involved with communication. Arriving data are deposited by the hardware directly into memory or registers or routed to functional units as specified by the program.

- An instruction with tensor operands can run synchronously or, at the discretion of the programmer, as a background thread that shares the datapath with other threads including the main one. A background thread runs a single tensor operation, as a single asynchronously running instruction. There is no context switch overhead. The registers and memory used by an asynchronous thread are those assigned by the programmer or compiler in the instruction, and these may not be overwritten until the thread terminates. Subsequent computation can be delayed until the thread terminates. The core supports nine concurrent threads of execution.

- A stream of data to or from the fabric may be used as an input to a tensor operation, or as the destination for one. The hardware directly implements scheduling activities that would normally be performed by an operating system. This allows compact and efficient software implementations. For example, one core can be sending data from its local memory to another core; simultaneously it can receive data from another core while adding it to values stored in its local memory. All of this is accomplished using only two machine instructions that run as independent threads.

- Code consists of tasks that react to events. Tasks are triggered by other tasks, or by arriving data words. The channel of the arriving word determines the code that is triggered. There is little delay between the completion of a task and the start of a subsequent task, as this is handled in hardware. Together with the SIMD operations, Rocki et al., “Fast stencil-code computation on a wafer-scale processor.” SC 2020.
A MIMD Machine with SIMD Processors (II)

- **SIMD processors**
  - 4-way SIMD for 16-bit floating point operands
  - 48 KB of local SRAM

---

More on the Cerebras WSE

https://www.youtube.com/watch?v=x2-qB0J7KHz
Serving DNNs in Real Time at Datacenter Scale with Project Brainwave

To meet the computational demands required of deep learning, cloud operators are turning toward specialized hardware for improved efficiency and performance. Project Brainwave, Microsoft's principal infrastructure for AI serving in real time, accelerates deep neural network (DNN) inferencing in major services such as Bing's intelligent search features and Azure. Exploiting distributed model parallelism and pinning over low-latency hardware microservices, Project Brainwave serves state-of-the-art, pre-trained DNN models with high efficiencies at low batch sizes. A high-performance, precision-adaptable FPGA soft processor is at the heart of the system, achieving up to 39.5 teraflops (Tflops) of effective performance at Batch 1 on a state-of-the-art Intel Stratix 10 FPGA.

The recent successes of machine learning, enabled largely by the rise of DNNs, have fueled a growing demand for ubiquitous AI, ranging from conversational agents to object recognition to intelligent search. While state-of-the-art DNNs continue to deliver major breakthroughs in challenging AI domains such as computer vision and natural language processing, their computational demands have steadily outpaced the performance growth rate of standard CPUs. These trends have spurred a

https://web.eecs.umich.edu/~mosharaf/Readings/Brainwave.pdf
Figure 4. The Brainwave NPU is a “mega-SIMD” vector processor architecture. A sequentially programmed control processor asynchronously controls the neighboring NFU, optimized for fast DNN operations. The heart of the NFU is a dense matrix-vector multiplication unit (MVU) capable of processing single DNN requests at low batch with high utilization. The MVU is joined to secondary multifunctional units (MFUs) that perform element-wise vector-vector operations and activation functions.
Fine-Grained Multithreading
Fine-Grained Multithreading

- Idea: Fetch from a different thread every cycle such that no two instructions from a thread are in the pipeline concurrently
  - Hardware has multiple thread contexts (PC+registers per thread)
  - Threads are completely independent
  - No instruction is fetched from the same thread until the prior branch/instruction from the thread completes

+ No logic needed for handling control and data dependences within a thread
+ High thread-level throughput

-- Single thread performance suffers
-- Extra logic for keeping thread contexts
-- Throughput loss when there are not enough threads to keep the pipeline full

Each pipeline stage has an instruction from a different, completely-independent thread.
Each pipeline stage has an instruction from a different, completely-independent thread

We need a PC and a register file for each thread + muxes and control
Fine-Grained Multithreading (II)

- **Idea:** Fetch from a different thread every cycle such that no two instructions from a thread are in the pipeline concurrently.

- **Tolerates control and data dependence resolution latencies** by overlapping the latency with useful work from other threads.

- **Improves pipeline utilization** by taking advantage of multiple threads.

- **Improves thread-level throughput** but sacrifices per-thread throughput & latency.

Multithreaded Pipeline Example

Slide credit: Joel Emer
Sun Niagara Multithreaded Pipeline

Fine-Grained Multithreading

- **Advantages**
  + No need for dependence checking between instructions
    (only one instruction in pipeline from a single thread)
  + No need for branch prediction logic
  + Otherwise-bubble cycles used for executing useful instructions from different threads
  + Improved system throughput, latency tolerance, pipeline utilization

- **Disadvantages**
  - Extra hardware complexity: multiple hardware contexts (PCs, register files, ...), thread selection logic
  - Reduced single thread performance (one instruction fetched every N cycles from the same thread)
  - Resource contention between threads in caches and memory
  - Dependence checking logic between threads may be needed (load/store)
Lecture on Fine-Grained Multithreading

- Idea: Fetch from a different thread every cycle such that no two instructions from a thread are in the pipeline concurrently
- Hardware has multiple thread contexts (PC+registers per thread)
- Threads are completely independent
- No instruction is fetched from the same thread until the prior branch/instruction from the thread completes

- No logic needed for handling control and data dependences within a thread
- High thread-level throughput
  -- Single thread performance suffers
  -- Extra logic for keeping thread contexts
  -- Throughput loss when there are not enough threads to keep the pipeline full

Each pipeline stage has an instruction from a different, completely-independent thread

Digital Design & Computer Architecture - Lecture 14: Pipelined Processor Design (Spring 2022)

Onur Mutlu Lectures
24.5K subscribers

Digital Design and Computer Architecture, ETH Zürich, Spring 2022 (https://safari.ethz.ch/digitaltechnik...)

Lecture 14: Pipelined Processor Design
Lecturer: Professor Onur Mutlu (https://people.inf.ethz.ch/omutlu/)
Date: April 8, 2022

https://youtu.be/XaW_O9nKPe0?t=5070
Lectures on Fine-Grained Multithreading

- Digital Design & Computer Architecture, Spring 2022, Lecture 14
  - Pipelined Processor Design (ETH, Spring 2022)
  - https://youtu.be/XaW_O9nKPe0?t=5070

- Digital Design & Computer Architecture, Spring 2020, Lecture 18c
  - Fine-Grained Multithreading (ETH, Spring 2020)
  - https://www.youtube.com/watch?v=bu5dxKTvQVs&list=PL5Q2soXY2Zi_FRrIoMa2fUYWPgiZUBQo2&index=26

https://www.youtube.com/onurmutlulectures