01 Background

Changes in conventional wisdom:

|  |  |
| --- | --- |
| **Old** | **New** |
| Power = cheap vs. transistors = expensive | Power = expensive vs. transistors = cheap |
| Increase ILP (Instruction-Level Parallelism) via compilers & hardware innovation. | Law of diminishing returns on more hardware for ILP. |
| Multiplies = slow, Memory = fast | Multiplies = fast, Memory = slow |
| 2x Processor performance / 1.5 years | 2x Processor performance / 5 years |

**Multiple**, more **simple** processors are more **power efficient**.

Performance issues:

1. Efficient algorithm & implementation
2. Data locality
3. Parallelism - multiple threads / vector parallelism

### The Principle of Locality

A program can access a relatively small portion of the address space at any instant in time.

1. **Temporal Locality**: (Locality in time)  
   If an item is referenced, it will tend to be referenced again soon. (i.e. loops)
2. **Spatial Locality**: (Locality in space)  
   If an item is referenced, items whose addresses are close tend to be referenced soon. (i.e. array accesses)

### Cache Misses

1. **Compulsory**:  
   Occurs the first time a piece of data is accessed.
2. **Capacity**:  
   The size of the cache is limited ⇒ data must be removed.
3. **Conflict**:  
   Cache lines get swapped out even if the cache is not full.
4. **Coherency**:  
   Cache invalidation due to cache coherency protocol.

**Reducing the miss rate**:

1. Increase block size (compulsory)
2. Increase cache size (capacity)
3. Increase associativity (conflict)

**Reducing miss penalty**: Multi-level caches

**Reducing hit time**: Prioritise reads over writes in the write buffer.

### Programming with Locality in Mind

Reducing cache misses:

1. Make data structures more **compact**.
2. Separate frequently and infrequently accessed data.

Linked lists have terribly locality, because each link can be anywhere in memory.

**Hybrid** data structures - e.g. linked lists where each element is a short array.

Collapse other linked structures - e.g. replace binary trees with B-trees.

Hash tables > binary trees.

In C, declare the **largest** component in structs first.

02 Efficiency

Improving code efficiency:

1. Code motion out of loops.
2. Combining tests.
3. Loop unrolling.
4. Unrolling to remove copies.
5. Software pipelining.
6. Unconditional branch removal.
7. Loop peeling.
8. Loop fusion.
9. Exploiting algebraic identities.
10. Short-circuiting monotone functions.
11. Long circuiting.
12. Arithmetic w/ flags.
13. Reordering tests.
14. Precompute functions.
15. Boolean / state variable elimination.
16. Collapsing procedure hierarchies.
    * Inlining.
    * Specialisation.
17. Exploit common cases.
    * Memoisation.
    * Pre-computed tables.
18. Coroutines.
19. Transformations on recursive procedures.
    * Tail call optimisation.
    * Inlining.
    * Recursion instead of iteration for automatic cache blocking.
20. Parallelism.
21. Compile-Time Initialisation.
    * Initialise tables at compile time instead of at runtime.
    * CPU time vs. disk read speed.
22. Strength reduction / incremental algorithms.
23. Common subexpression elimination.
24. Pairing computation.
25. Exploit word parallelism / SIMD.
26. Data structure augmentation.
27. Lazy evaluation.
28. Packing - data compression.
29. Interpreters / factoring.
30. Compiler flags.

### Faster C code

Use static keyword if function is used in only **one** file.

Use inline declaration where you want a function **inlined**.

Use restrict pointers to tell the compiler about **pointer aliasing** (The same memory location being accessible by different names).

Use a register declaration on variables to prevent it being unavailable for register allocation.

### Parallelism

Parallelism between:

* CPUs = **multithreading**
* CPU and disk = **prefetching** / write buffering
* CPU and GPU = **triple buffering**
* Machine instructions = instruction scheduling

03 OpenMP

**Open Multi-Processing (OpenMP)** is comprised of:

1. Compiler directives
2. Runtime library routines
3. Environment variables

### Shared Memory Model

|  |  |
| --- | --- |
| **Uniform Memory Access** | **Non-Uniform Memory Access** |
|  |  |

### Threads

OpenMP uses **threads** to accomplish parallelism:

* A thread of execution is the **smallest unit of processing** that can be scheduled by an operating system.
* Threads exist within the resources of a **single process**.
* Usually, the number of threads = the number of machine processors / cores.  
  The use of threads is up to the application.

### OpenMP

The **master thread** does most of the **sequential work** of the program. Other threads are activated for parallel sections.

|  |
| --- |
| int x = 5; #pragma omp parallel {  x++; // This will be executed by all threads. } |

Three basic constructs for dividing work in OpenMP:

1. Parallel for
2. Parallel sections
3. Parallel task

Each iteration of the **for** loop is divided between the threads:

|  |
| --- |
| #pragma omp parallel for for (int i = 0; i < n; i++) {  a[i] = b[i] \* c[i]; } |

Each **section** is executed in parallel by different threads:

|  |
| --- |
| #pragma omp parallel sections {  #pragma omp section  min = getMin(array);  #pragma omp section  max = getMax(array); } |

**Tasks** in the pool of tasks are executed by available threads:

|  |
| --- |
| #pragma omp parallel {  #pragma omp single // Just one thread adds these tasks into the pool.  {  #pragma omp task  printf("Hello ");   #pragma omp task  printf("World\n");  } } |

### Scope of Data

By default in OpenMP, **all data is shared** between threads.

|  |
| --- |
| #pragma omp parallel {  #pragma omp atomic  x++; // Update the variable in a single, unbreakable step. } |

|  |
| --- |
| #pragma omp parallel {  #pragma omp critical  x++; // Only one thread can execute this section at once. } |

**Named critical sections** allow us to specify which sections cannot be executed in parallel.

Variables and can both be updated in parallel, since their critical sections are named differently:

|  |
| --- |
| #pragma omp parallel {  #pragma omp critical (updateX)  x++; // Only one thread can execute this section at once.  #pragma omp critical (updateY)  y++; // Only one thread can execute this section at once. } |

Atomic updates are **faster** than critical sections, and are less **error-prone** in general.

### Private Variables

|  |
| --- |
| int x = 0, y = 0;  #pragma omp parallel private(y) {  y = 0; // Local copy is uninitialised.   for (int i = 0; i < omp\_get\_thread\_num(); i++) {  y++; // Increment local copy.  }   #pragma omp atomic  x++; // Increment shared variable. } |

Using firstprivate() allows private variables to maintain the value from outside their scope:

|  |
| --- |
| int x = 0, y = 0;  #pragma omp parallel firstprivate(y) {  for (int i = 0; i < omp\_get\_thread\_num(); i++) {  y++; // Increment local copy.  }   #pragma omp atomic  x++; // Increment shared variable. } |

If using variables in **tasks**, a private copy must be created. This prevents the value from changing in between the task being added to the pool and the task being executed:

|  |
| --- |
| int x = 0;  #pragma omp parallel {  #pragma omp single  {  #pragma omp task firstprivate(x)  doFunction(x);  } } |

The default(none) declaration requires all variables to be declared as shared or private.

### Reductions

A reduction involves combining **multiple** values into a **single** value.

A private copy of sum is created for each thread, which OpenMP combines into a final value at the **end** of the parallel section:

|  |
| --- |
| int sum = 0;  #pragma omp parallel for reduction(+:sum) for (int i = 0; i < n; i++) {  sum += a[i]; } |

### Scheduling Parallel For Loops

OpenMP has three scheduling options:

1. **Static**: (default)  
   Iterations are **divided evenly** between threads.
2. **Dynamic**:  
   Iterations are put onto a **work queue** from which idle threads take. A **chunk size** can be specified to allocate more than one iteration to each thread.
3. **Guided**:  
   Similar to dynamic, but chunk sizes **decrease** as the loop progresses.

|  |
| --- |
| #pragma omp parallel for schedule(dynamic, 100) |

### Conditional Parallelism

OpenMP directives can be made conditional on **runtime conditions**:

|  |
| --- |
| #define DEBUGGING 1 #pragma omp parallel for if (!DEBUGGING) |

|  |
| --- |
| #pragma omp parallel for if (n > 128) |

### Cost of OpenMP Constructs

There is an implicit barrier at the end of each for loop. The nowait clause eliminates this barrier:

|  |
| --- |
| #pragma omp parallel {  #pragma omp for nowait  for (int i = 0; i < n; i++)  a[i] = b[i] + c[i];   // There is no dependency between these two loops.  #pragma omp for  for (int i = 0; i < m; i++)  x[i] = b[i] + c[i]; } |

### Vector SIMD

The SIMD (Single Instruction, Multiple Data) directive requires the compiler to **vectorise** the loop, even though the vectorisation may be unsafe or wrong:

|  |
| --- |
| #pragma omp simd for (int i = 0; i < n; i++) {  a[i] = a[i] + s\*b[i]; } |

Loops cannot be easily vectorised if there are **dependencies** between iterations. This can be done however, if there is a large dependence distance:

|  |
| --- |
| #pragma omp simd safelen(10) for (int i = 10; i < n; i++) {  a[i] = sqrt(a[i-10] + b[i]); } |

The safelen keyword tells the compiler that it should not generate vector code that operates on more than the specified number of iterations.

Loops containing function calls **cannot** be easily vectorised.

We can declare that the compiler should create a vectorised version of a function:

|  |
| --- |
| #pragma omp declare simd float myFunction(float number) {  // Body of function. } |

The compiler will generate a version of the code that takes an entire vector of floats as a parameter, and computes a whole vector of results.

04 More OpenMP

### Barriers

An explicit barrier can be declared using the barrier directive:

|  |
| --- |
| #pragma omp barrier |

### Locks

Creating / destroying locks:

|  |
| --- |
| void omp\_init\_lock(omp\_lock\_t \*lock); void omp\_destroy\_lock(omp\_lock\_t \*lock); |

Setting / releasing locks:

|  |
| --- |
| void omp\_set\_lock(omp\_lock\_t \*lock); void omp\_unset\_lock(omp\_lock\_t \*lock); |

Unsetting a lock needs to be done by the thread that set it.

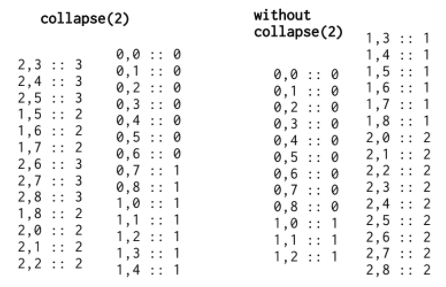
Since the omp\_set\_lock() call is blocking, you can test a lock:

|  |
| --- |
| omp\_test\_lock(); |

### Collapsing Nested For Loops

Nested for loops can be collapsed using the collapse directive:

|  |
| --- |
| #pragma omp parallel for collapse(2) for (int i = 0; i < 3; i++)   for (int j = 0; j < 9; j++)  printf("%d, %d :: %d\n", i, j, omp\_get\_thread\_num()); |



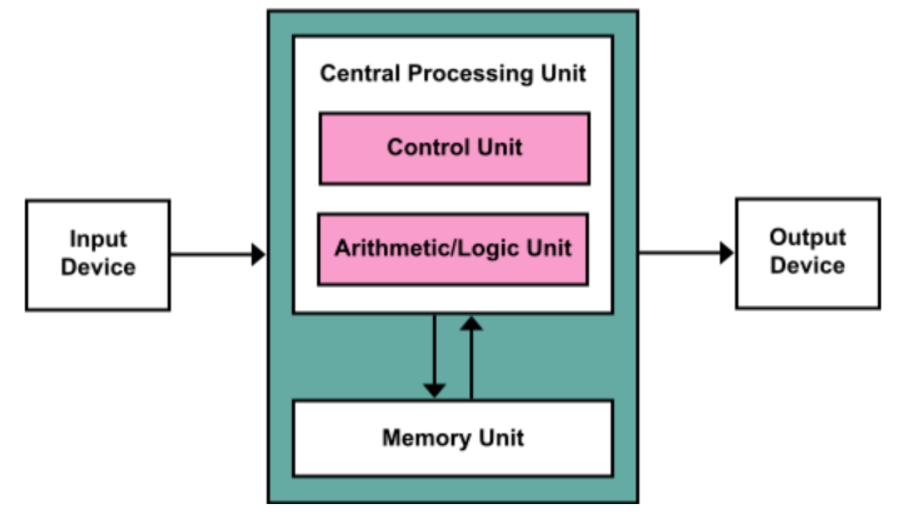
05 Flynn’s Taxonomy

Flynn’s Taxonomy classifies computer architectures into **four types**: SISD, SIMD, MISD, MIMD.

How many instructions vs. how much data can be processed simultaneously?

|  |  |
| --- | --- |
| **SISD** - Single Instruction, Single Data  Exploits **no parallelism**, either on the instruction or the data level.  e.g. Uniprocessor machines. |  |
| **SIMD** - Single Instruction, Multiple Data  **Same instruction** runs on **all** processors.  **Data-level** parallelism, not concurrency.  Applications:   * Image editing * Multimedia processing   e.g. Vector computers, GPUs. |  |
| **MISD** - Multiple Instructions, Single Data  **Different instructions** run on the **same data**.  Pipelining.  Applications:   * Convolution * Matrix operations * Data sorting   Uncommon architecture in reality. |  |
| **MIMD** - Multiple Instructions, Multiple Data  **Different instructions** can run on **different data**.  **Processing**: Asynch, independent  **Memory**: Shared or distributed  Applications:   * Simulation * Emulation (VM) * CAD / CAM * Modelling   e.g. Multi-core superscalar processors. |  |

**Von-Neumann** Architecture:



### Types of MIMD

|  |  |
| --- | --- |
| **SPMD** - Single Program, Multiple Data | **MPMD** - Multiple Programs, Multiple Data |
| Program is executed at independent execution points.  Most common style of parallel programming. | At least two programs.  One program is the master / controller.  Other nodes receive program from master. |

Flynn’s Taxonomy:

* Quite difficult to fit **parallel** architectures.
* Old (1966).
* Where do these fit?
  + ILP (Instruction-Level Parallelism)
  + Fine-grain speculative multithreading
* Most important distinction is between **SIMD** and **MIMD**.

06 Parallel Architectures

### Memory Access

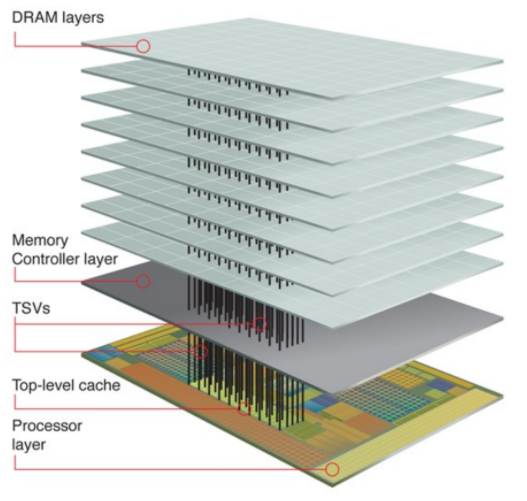
|  |  |
| --- | --- |
| **Uniform Memory Access**: (UMA) | **Non-Uniform Memory Access**: (NUMA) |
|  |  |

|  |  |  |
| --- | --- | --- |
|  | **UMA** | **NUMA** |
| **Latency** | Same | Different |
| **Bandwidth** | Same | Different |
| **Memory** | Different | Distributed |

### Heterogeneous Uniform Memory Access (hUMA)

|  |  |
| --- | --- |
| **Without hUMA**: | **With hUMA**: |
| **1)** CPU copies data to GPU memory.  **2)** GPU completes computation.  **3)** GPU copies result back to CPU memory. | Both CPU and GPU can access and allocate any location in the system’s virtual memory space. |

### 3D Processors



Stacked DRAM - HBM (High-Bandwidth Memory):

* Offers **more bandwidth**.
* Uses **less energy** usage per bit.
* Occupies **less space**.

### Symmetric vs. Asymmetric Multiprocessing

|  |  |
| --- | --- |
| **SMP (Symmetric Multiprocessing)** | **ASMP (Asymmetric Multiprocessing)** |
| Two or more **identical** processors connected to a **single** shared memory (UMA).  Most multiprocessors use SMP.  For the OS, all processors are treated the same.  Processors are **tightly-coupled** (Connected at the bus level). | Processors are **not** treated the same.  ASMP is **expensive**, hence is **rarer** than SMP. |

### Variable SMP (vSMP)

|  |  |
| --- | --- |
|  | Used for **background** tasks, audio, video, email syncs, social media syncs etc.  Single-core performance for email, 2D games, basic web browsing, maps etc.  Dual-core performance for Flash-enabled browsing, multitasking, video chat etc.  Quad-core performance for console-class gaming, faster browsing, media processing etc. |

**Multicore Processors**:

* May or may not share a cache.
* May implement message passing or IPC (interprocess communication).
* Cores can be connected in:
  + Bus
  + Ring
  + 2D mesh
  + Crossbar
* Can be **homogeneous** (identical cores) or **heterogeneous** (non-identical cores).

### ARM Heterogeneous Multicore Architectures

|  |
| --- |
| **big.LITTLE**:  Offers finer-grained control of workloads by combining powerful (big) and weak (LITTLE) processing cores. Typically one “side” will be active at a time.  Since all cores have access to the same memory regions, workloads can be **swapped** between big and LITTLE cores on the fly.  This multicore processor can better adjust to **dynamic** computing needs and uses **less power** than clock scaling alone (up to 75%). |
| **DynamIQ**:  Combines big and LITTLE cores into a single, fully-integrated **cluster**.  Memory subsystem is **shared** across all cores for efficiency.  Has various configurations e.g. 1 big + 7 LITTLE CPUs. |

# Instruction Level Parallelism (ILP)

ILP measures how many instructions can be executed **simultaneously**.

**Hardware** (Dynamic Parallelism) - Decide at **runtime** what to execute.

**Software** (Static Parallelism) - **Compiler** decides what to execute.

### Instruction Pipelining

Occurs within a **single** processor.

Keeps every part of a processor busy by **dividing instructions** that execute in **parallel**.

Fetch → Decode → Execute cycle.

### Pipeline Branching

If a branch is **not** taken, resources are wasted. This causes a delay in execution.

Branch prediction:

* Algorithm to predict which branch might be taken to prevent bubbles.
* Complex to execute accurately.

### Superscalar Architectures

|  |  |
| --- | --- |
| **Scalar** | **Superscalar** |
| Each instruction manipulates 1 - 2 **data items** at one time. | Executes more than one **instruction** at one time.  Multiple simultaneous instructions to different execution units = more throughput per cycle. |

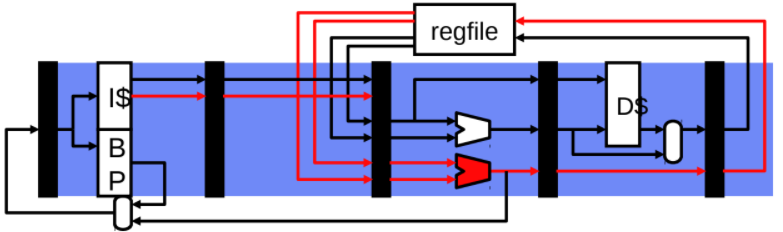
**Flynn’s Taxonomy**:

* SISD for single core / SIMD for vector operations.
* MIMD for multiple cores.

07 Superscalar Pipelines

### A Typical Dual-Issue Pipeline

1. **Fetch** an entire 16B / 32B cache block.
   * 4-8 instructions - 32-bit instruction length
   * Predict a single branch per cycle
2. Parallel **decode** instructions.
   * Check for **conflicting** instructions i.e. data dependencies.
   * Check for **stalls** e.g. load-use delay.



**Multi-ported register file**:

* Larger area, latency, power requirements, cost, complexity.

**Multiple execution units**:

* Simpler adders are easy, but bypass paths are expensive.

**Memory unit**:

* Single load per cycle (stall at decode) is fine for dual-issue pipelines.
* *Alternative*: Add a **read port** to data cache.
  + Larger area, latency, power, cost, complexity.

### Superscalar Challenges

|  |
| --- |
| **Instruction Fetch**:  *Modest* - Fetch **multiple** instructions per cycle.  *Aggressive* - **Buffer** instructions and / or predict **multiple** branches. |
| **Instruction Decode**:  Replicate decoders. |
| **Instruction Issue**:  Determine when instructions can proceed **in parallel**.  More complex stall logic - O(N2) for N-wide machine. |
| **Register Read**:  Port for each register read (4-wide superscalar = 8 read ports)  Each port needs its own set of addresses and data wires - latency & area #ports2 |
| **Instruction Execution**:  Replicate arithmetic units.  Perhaps multiple **cache ports** (slower access, higher energy) - Only for 4-wide or larger. |
| **Register Bypass Paths**:  More possible sources for data values.  O(N2) for N-wide machine. |
| **Instruction Register Writeback**:  One write port per instruction that writes a register i.e. 4-wide = 4 write ports. |
| **Fundamental Challenge**:  Amount of ILP in the program. |

### Register Bypass

Register bypassing is a **hardware mechanism** that allows data to flow **directly** from the output of one instruction into the input of another. This requires a hardware interconnection network between the outputs of functional units (e.g. adders / multipliers) and the inputs of other units.

### Superscalar Register Bypass

|  |
| --- |
| **N2 Bypass Network**:   * input muxes at each ALU input. * point-to-point connections. * Routing lengthens wires. * Heavy capacitive load.   Even more is required for deeper pipelines. This is one of the big superscalar problems. |

### Solutions for Superscalar Register Bypass

|  |
| --- |
| **Clustering**: (Mitigates bypass)   * Group ALUs into clusters.   + Full bypassing **within** a cluster.   + Limited bypassing **between** clusters - 1-2 cycle delay which can hurt IPC. * inputs at each mux. * bypass paths in each cluster. |
| **Steering**: (Key to performance)   * Steer **dependent** instructions to the same cluster. |
| **Cluster Register File**:   * Replicate a register file **per cluster**. * All register file writes update **all** replicas. * Fewer read ports - only for clusters. |

### Challenge: Superscalar Fetch

If the multiple instructions being fetched are within the **same** cache block, then there is no issue.

If the next instruction is the **last** instruction in a block, then only **one** instruction will be fetched that cycle. Some processors, however, *may* allow fetching from two **consecutive** blocks.

**Branches**:

For taken branches, the average number of instructions per taken branch = 10 instructions

(20% branches, 50% taken).

Without smarter fetch, for a 5 instruction 4-issue processor ILP is limited to 2.5, instead of 4.

### Multiple-Issue Implementations

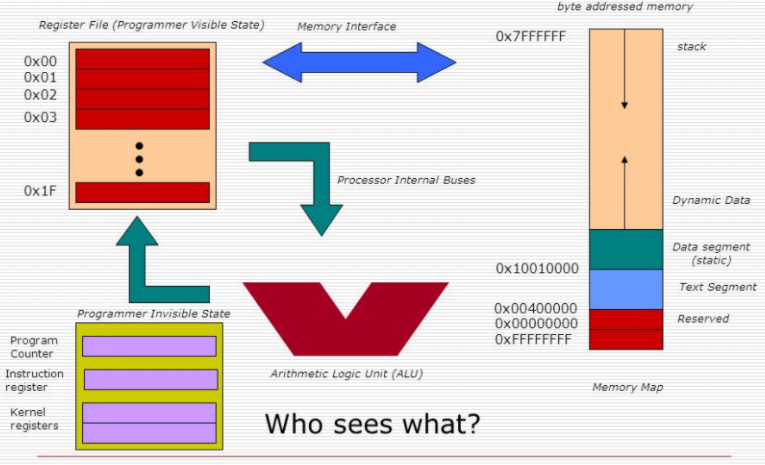
|  |
| --- |
| **Statically-Scheduled (In-Order) Superscalar**:   * Executes unmodified sequential programs. * **Hardware** must figure out what can be done in parallel. |
| **Very Long Instruction Word (VLIW)**:   * **Compiler** identifies independent instructions. * Hardware can be **simple** and perhaps lower power. |
| **Dynamically-Scheduled (Out-of-Order) Superscalar**:   * Hardware extracts more ILP by on-the-fly **reordering** of instructions. |

Issue width has saturated at **4-6** for high performance cores = not enough ILP to justify wider.

For high performance per watt cores (smart phones), 2-wide superscalar. (Increasing each gen.)

08 Instruction Set Architecture (ISA)

An ISA allows software to direct hardware, and defines machine language.



An ISA defines:

* Data types
* Memory (registers)
* Addressing modes
* Instruction set
* I/O

x86 is an ISA. Processors with different designs / microarchitectures can share the same ISA.

### Classifying ISA

Based on complexity:

* **CISC** - Complex Instruction Set Computer
* **RISC** - Reduced Instruction Set Computer

Parallelism / Word size:

* **VLIW** - Very Long Instruction Word
* **LIV** - Long Instruction Word
* **EPIC** - Explicitly Parallel Instruction Computing

### CISC

Single instructions can execute **multiple operations**.

Designed to implement programming constructs into a single instruction such as:

* Procedure calls
* Loops
* Array access
* Address lookups

However, **less complex** instructions can be performed better.

### RISC

RISC vs CISC:

* Reduced means **less complex** than CISC, not reduced in size.
* Requires **fewer cycles** to execute.
* Instructions are **simpler**, smaller and more general.
* Has **more instructions** than CISC.

Has **fixed-length** instructions - mostly 32-bit

One drawback is code density.

Mainly used in **smartphone** CPUs.

### Massively Parallel Processing (MPP)

MPP uses a large number of processors which simultaneously process different parts of a program.

Can use different computers to create MPPAs (Massively Parallel Processing Arrays). This is **grid computing**.

Used in supercomputers (this is **not** cluster computing).

### Grid Computing

Grid computing utilises a **distributed** system.

Useful for **non-interactive workloads** (tasks where little interaction is needed between nodes).

Each node can perform a **different task**.

* Nodes *can* be heterogeneous (non-identical cores).
* Nodes are **not** physically coupled.

### GPU / GPGPU

GPGPU - General Purpose computing on Graphical Processing Unit

GPUs are great at **parallelisation**.

Multiple GPUs can be used in a **pipeline** formation e.g. Nvidia CUDA.

### Vector Processing

Instructions operate on 1D vectors (SIMD).

Vector processing is very useful for numerical tasks.

GPUs are *similar* to vector processors.

### Gustafson’s & Amdahl’s Law

A task executed by a system whose resources are improved compared to an initial similar system, can be split into two parts:

1. A part that does **not** benefit from the improvement.
2. A part that **does** benefit from the improvement.

09 Vector Processors

### SIMD - Single Instruction, Multiple Data

Useful for applying **identical** computations across many data elements.

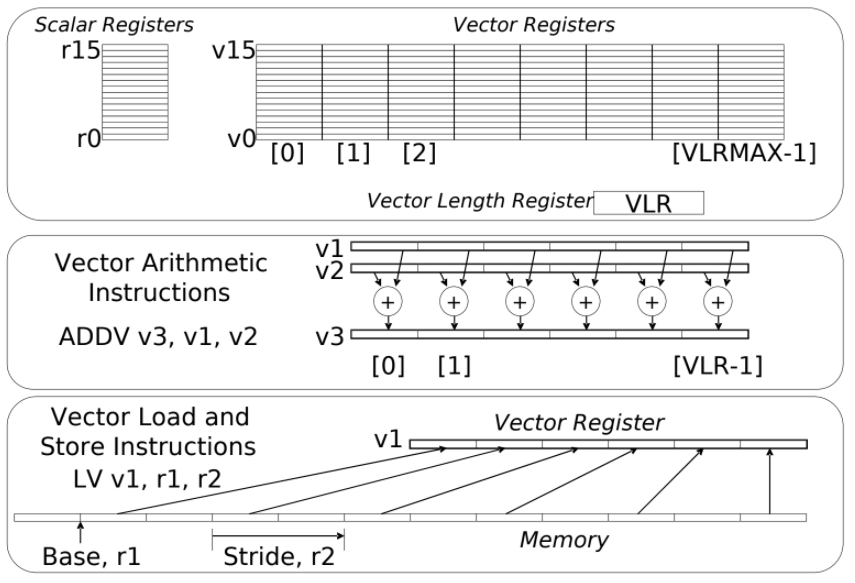
Energy efficient:

* Less control logic per functional unit,
* Less instruction fetch / decode energy.

Bandwidth efficient & latency tolerant:

* Memory systems are good for **sequential** scans through arrays.

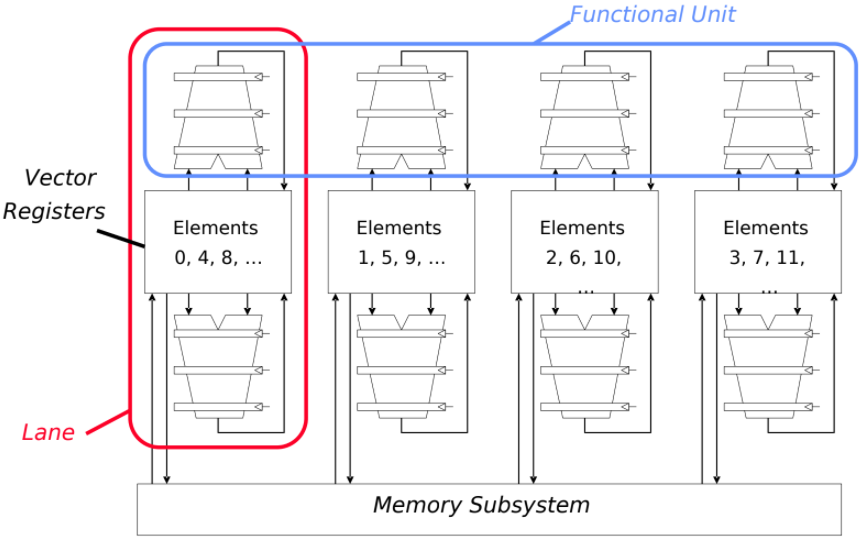
### Vector Programming Model



### Vector Memory-Memory vs. Vector Register Machines

Vector Memory-Memory instructions hold all vector operands in **main memory**, not in vector registers.

### Vector Unit Structure



### Multimedia Extensions (SIMD Extensions)

Uses very **short vectors** added to **existing ISAs** for microprocessors.

Splits 64-bit registers into 2\*32-bit or 8\*8-bit

A single instruction operates on **all elements** within a register.

**Multimedia Extensions vs. Vectors**:

* Limited instruction set:
  + No vector length control.
  + No strided load/store or scatter/gather.
  + Unit-stride loads must be aligned to 64 / 128-bit boundary.
* Limited vector register length:
  + Requires superscalar dispatch to keep multiply / add / load units busy.
  + Loop unrolling to hide latencies increases register pressure.
* Trend towards fuller vector support in microprocessors:
  + Better support for misaligned memory accesses.
  + Support of double precision.

### Modern vs. Old Vector Computers

|  |  |
| --- | --- |
| **Modern Vector Computers** | **Old Vector Computers** |
| Relatively **short vectors** (16B / 32B), but the trend is for them to grow longer. | Used to have much **longer vectors** (256B). |
| Use **vector registers** for everything - short vector registers don’t require much space on the chip. | Took data directly from **memory**. |
| Use a **separate** arithmetic / floating point unit per lane. | Used as little as one arithmetic / floating point unit to implement vector instructions, but were very **deeply pipelined**. |

Vector architectures are becoming important again, especially for **low-energy computation**.

### Supercomputers

A supercomputer is:

* The fastest machine in the world at a given task.
* A device to turn a compute-bound problem into an I/O problem.

All supercomputers involve computations on large data sets.

### Vector Chaining

Vector chaining is a vector version of **register bypassing**.

With chaining, a dependent instruction can be executed as soon as the result is calculated.

### Vector Instruction Parallelism

Multiple vector instructions can be overlapped.

### Vector Startup

Two components of **vector startup penalty**:

1. Functional unit latency - time through pipeline.
2. Dead time / recovery time - time before another vector instruction can start down the pipeline.

### Vector Scatter / Gather

**Gather**:

A sparsely-populated vector holding non-empty elements can be represented by two densely-populated vectors of length :

1. containing the non-empty elements of .
2. giving the index in where 's element is located.

**Scatter:**

The sparse scatter is the **reverse** operation. It copies the values of into the corresponding locations in the sparsely-populated vector .

### Vector Conditional Execution

In order to vectorise loops with conditional code, vector mask / vector flag registers can be added using 1 bit per element.

These must be implemented with maskable vector instructions which produces NOPs where the mask bit is clear.

10 Intel SSE

### Intrinsics

Intrinsics are special functions for which the compiler generally has a specific optimisation path. They generally encode to a specific small list of sequential instructions.

They offer results nearly as good as assembly language, and code readability is not as compromised.

We do not have to worry about register management, as this is handled by the compiler. The SSE intrinsics and types are defined in the xmmintrin.h header file. Simply #include this header.

### Types (128-bit wide)

|  |
| --- |
| \_\_m128 v1 // 4x 32-bit single-precision floats. \_\_m128i v2 // 4x 32-bit integers. \_\_m128d v3 // 2x 64-bit double precision floats. |

### Load Instructions

|  |
| --- |
| // Load 4 floats from a 16B aligned address. \_\_m128 \_\_mm\_load\_ps(float \*src);  // Load from an unaligned address (4x slower). \_\_m128 \_\_mm\_loadu\_ps(float \*src);  // Load 1 float into all 4 fields of an \_\_m128. \_\_m128 \_\_mm\_load1\_ps(float \*src);  // Load 4 floats from parameters into an \_\_m128. \_\_m128 \_\_mm\_setr\_ps(float a, float b, float c, float d);  // Load 1 float into all 4 fields of an \_\_m128. \_\_m128 \_mm\_set1\_ps(float w); |

### Store Instructions

|  |
| --- |
| // Store 4 floats to an aligned address. void \_mm\_store\_ps(float \*dest, \_\_m128 src);  // Store 4 floats to an unaligned address. void \_mm\_storeu\_ps(float \*dest, \_\_m128 src); |

Aligned stores / loads must operate on a 16B aligned address. Otherwise, a segfault will occur.

If you must use an unaligned address, use the unaligned intrinsics. These are much slower however than aligned.

### Arithmetic Instructions

|  |
| --- |
| \_\_m128 \_mm\_add\_ps(\_\_m128 a, \_\_m128 b); // Addition of two floats. \_\_m128 \_mm\_sub\_ps(\_\_m128 a, \_\_m128 b); // Subtraction of two floats. \_\_m128 \_mm\_mult\_ps(\_\_m128 a, \_\_m128 b); // Multiplication of two floats. \_\_m128 \_mm\_div\_ps(\_\_m128 a, \_\_m128 b); // Division of two floats. \_\_m128 \_mm\_min\_ps(\_\_m128 a, \_\_m128 b); // Minimum of two floats. \_\_m128 \_mm\_max\_ps(\_\_m128 a, \_\_m128 b); // Maximum of two floats. |

### Other Instructions

|  |
| --- |
| \_\_m128 \_mm\_sqrt\_ps(\_\_m128 a); // Square root of four floats. \_\_m128 \_mm\_rcp\_ps(\_\_m128 a); // Rough (12-bit) reciprocal of four floats. \_\_m128 \_mm\_rsqrt\_ps(\_\_m128 a); // Rough square root of four floats. |

### Precision and Speed

Some of the SSE instructions have reduced accuracy and **low precision**. Because of this, a reciprocal executes as fast as an ADD instruction.

For full precision, a Newton-Rhapson Iteration can be performed.

11 More Intel SSE

### Newton-Rhapson Method

Newton’s Method is a root-finding algorithm that uses the first few terms of the Taylor series of a function in the vicinity of a suspected root.

**Newton’s iteration** is reserved to the application of Newton’s method for computing **square roots**.

### Newton-Rhapson Reciprocal

|  |
| --- |
| \_\_m128 rcp\_nr(const \_\_m128 &x) {  const \_\_m128 rcp = \_mm\_rcp\_ps(x); // 1/x  const \_\_m128 lhs = \_mm\_add\_ps(rcp, rcp); // 2 \* 1/x  const \_\_m128 tmp = \_mm\_mul\_ps(x, rcp); // x \* 1/x  const \_\_m128 rhs = \_mm\_mul\_ps(rcp, tmp); // 1/x \* (x \* 1/x)   return \_mm\_sub\_ps(lhs, rhs); } |

### Newton-Rhapson Reciprocal Square Root

|  |
| --- |
| \_\_m128 rsqrt\_nr(const \_\_128 &x) {  const \_\_m128 half = \_mm\_set1\_ps(0.5f);  const \_\_m128 three = \_mm\_set1\_ps(3.0f);  const \_\_m128 rcp = \_mm\_sqrt\_ps(x); // rsqrtps(x)   const \_\_m128 tmp1 = \_mm\_mul\_ps(rcp, rcp); // rsqrtps(x) \* rsqrtps(x)  const \_\_m128 tmp2 = \_mm\_mul\_ps(x, tmp1); // x \* (rsqrtps(x) \* ...)  const \_\_m128 tmp3 = \_mm\_sub\_ps(three, tmp2); // 3 - (x \* ...)  const \_\_m128 tmp4 = \_mm\_mul\_ps(rcp, tmp3); // rsqrtps(x) \* (3 \* ...)  return \_mm\_mul\_ps(half, tmp4); // 1/2 \* (rsqrtps(x) \* ...) } |

### Operating on Bits

|  |
| --- |
| \_\_m128 \_mm\_and\_ps(\_\_m128 a, \_\_m128 b); // ANDs two vectors.  \_\_m128 \_mm\_or\_ps(\_\_m128 a, \_\_m128 b); // ORRs two vectors.  \_\_m128 \_mm\_andnot\_ps(\_\_m128 a, \_\_m128 b); // NANDs two vectors.  \_\_m128 \_mm\_xor\_ps(\_\_m128 a, \_\_m128 b); // XORs two vectors. |

### Comparisons

SSE allows us to compare four values at a time.

|  |
| --- |
| \_\_m128 \_mm\_cmpeq\_ps(\_\_m128 a, \_\_m128 b); // a == b  \_\_m128 \_mm\_cmplt\_ps(\_\_m128 a, \_\_m128 b); // a < b  \_\_m128 \_mm\_cmple\_ps(\_\_m128 a, \_\_m128 b); // a <= b  \_\_m128 \_mm\_cmpgt\_ps(\_\_m128 a, \_\_m128 b); // a > b  \_\_m128 \_mm\_cmpge\_ps(\_\_m128 a, \_\_m128 b); // a >= b  \_\_m128 \_mm\_cmpneq\_ps(\_\_m128 a, \_\_m128 b); // a != b |

Comparison instructions return a **bitmask** indicating which of the constituent parts of the register passed / failed.

SSE provides the ability to convert the \_\_m128 mask into a 4-bit integer using the \_mm\_movemask\_ps intrinsic.

|  |  |
| --- | --- |
| 1111  0000  1100 | True for all four floats.  False for all four floats.  True for the first two floats |

### Shuffle

Vector word values can be reordered using the shuffle instruction. It takes two separate vector words and takes two 32-bit values from each:

|  |
| --- |
| \_\_m128 a = \_mm\_set\_ps(0.0, 1.0, 2.0, 3.0); \_\_m128 b = \_mm\_set\_ps(4.0, 5.0, 6.0, 7.0); \_\_m128 c = \_mm\_shuffle\_ps(a, b, \_MM\_SHUFFLE(1, 0, 3, 2); // c = {2.0, 3.0, 4.0, 5.0} |

### Horizontal Operations

|  |
| --- |
| \_\_m128 \_mm\_hadd\_ps(\_\_m128 a, \_\_m128 b); |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| a = | a0 | a1 | a2 | a3 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| b = | b0 | b1 | b2 | b3 |

c = a + b

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| c = | b0+b1 | b2+b3 | a0+a1 | a2+a3 |

### Scalar Operations

Scalar operations act on only the **least significant** portion of the vector:

|  |
| --- |
| \_\_m128 \_mm\_add\_ss(\_\_m128 a, \_\_m128 b); |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| a = | a0 | a1 | a2 | a3 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| b = | b0 | b1 | b2 | b3 |

c = a + b

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| c = | a0 | a1 | a2 | a3+b3 |

SSE instructions suffixed with \_ps are **packed** instructions.

**Scalar** instructions are suffixed with \_ss instead.

13 Multithreading

### Thread-Level Parallelism (TLP)

A thread is a process with its own instructions and data. It may be a subpart of a parallel program (thread), or it may be an independent program (process).

Each thread has all the state (instructions, data, PC, register state) necessary to allow it to execute.

### Data-Level Parallelism

Performing identical operations on data.

### Continuum of Granularity

|  |  |
| --- | --- |
| **Coarse** | **Fine** |
| Each processor is **more powerful**. | Each processor is **less powerful**. |
| Usually **fewer** processors. | Usually **more** processors. |
| Communication is more expensive between processors. | Communication is cheaper between processors. |
| Processors are more **loosely-coupled**. | Processors are more **tightly-coupled**. |
| Tend towards **MIMD**. | Tend toward **SIMD**. |

### Thread-Level Parallelism (TLP)

ILP exploits **implicit** parallel operations within a loop or straight-line code segment.

TLP instead is **explicitly** represented by the use of multiple threads of execution that are inherently parallel. Code **must** be written to be thread-parallel.

Multiple instruction streams improve:

* Throughput of computers that run many programs.
* Execution time of multithreaded programs.

TLP could be more cost-effective to exploit than ILP.

### Multithreading

Multithreading uses multiple threads to share the functional unit of **one** processor via **overlapping**.

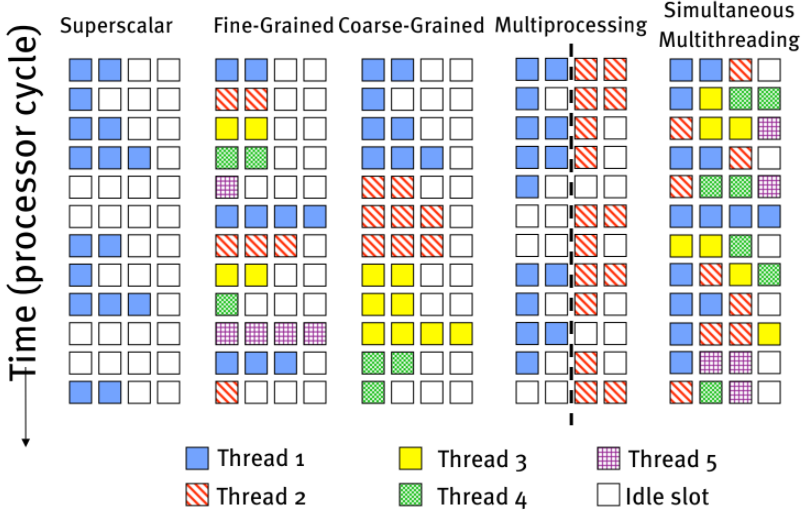
The processor must **duplicate** independent state of each thread (e.g. register file, PC, page table).

Memory is shared through the **virtual memory** mechanisms, which already support multiple processes.

Hardware for fast thread switch is much faster than full process switches.

|  |  |
| --- | --- |
| **Single-Threaded CPU**:  Executes instructions from one thread at a time. |  |
| **Symmetric Multiprocessing (SMP)**:  Multiple threads execute on multiple processors at the same time. |  |
| **Superthreading**:  Executes instructions from multiple threads simultaneously on one core. |  |
| **Simultaneous Multithreading (SMT) / Hyperthreading**:  Instructions issued on the same clock need **not** be from the **same** thread. |  |

### Multithreaded Categories



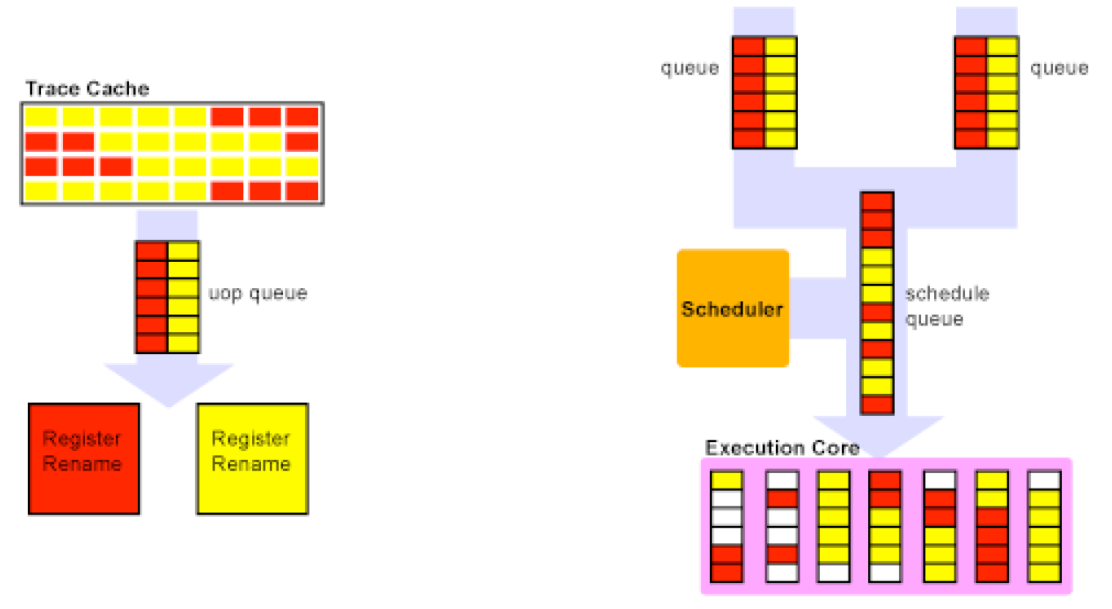
### Switching Between Threads

When do we switch between threads?

1. **Fine-grain** - Alternate instruction per thread.
2. **Coarse-grain** - When a thread is stalled, another thread can be executed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Fine-Grained Multithreading**:  Switches between threads on each instruction, causing the execution of multiple threads to be interleaved. This is usually done in a **round-robin** fashion, skipping any stalled threads.   |  |  | | --- | --- | | **Advantages**: | **Disadvantages**: | | **✓** It can hide both short and long stalls, since instructions from other threads are executed when one thread stalls. | **✗** It **slows down** the execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads. |   The CPU must be able to switch threads **every** clock cycle. |
| **Coarse-Grained Multithreading**:  Switches threads only on costly stalls, such as L2 cache misses.   |  |  | | --- | --- | | **Advantages**: | **Disadvantages**: | | **✓** Relieves the need to have fast thread switching.  **✓** Doesn’t slow down the thread, since instructions from other threads are only issued when the thread encounters a costly stall. | **✗** Hard to overcome throughput losses from shorter stalls due to the pipeline startup costs.   * When a stall occurs, the pipeline must be emptied / frozen. * New thread must fill the pipeline before instructions can complete. |   Because of the startup overhead, coarse-grained multithreading is better for reducing the penalty of high-cost stalls, where pipeline refill time < stall time. |

### Static vs. Dynamic Partitioning



### Design Challenges in SMT

Since simultaneous multithreading makes sense only with **fine-grained** implementation, what is the impact of this scheduling on single-thread performance?

* With a **preferred thread** approach, the processor is likely to sacrifice some throughput when the preferred thread **stalls**.
* A larger **register file** is needed to hold **multiple contexts**.
* Challenges in not affecting clock cycle time, especially in:
  + **Instruction issue** - more candidates need to be considered.
  + **Instruction completion** - choosing which instructions to commit may be challenging.
* Ensuring that cache and TLB **conflicts** do not degrade performance.

### Problems with SMT

* One thread **monopolises resources**:
  + One thread can tie up the functional unit with long-latency instructions, while the other thread is tied up in the scheduler.
* Cache effects:
  + Caches are unaware of SMT and can’t make warring threads cooperate.
  + If both warring threads access different memory and have cache conflicts, the constant swapping occurs.

### Olukotun’s View

“With the exhaustion of essentially all performance gains that can be achieved for ‘free’ with technologies such as **superscalar dispatch** and **pipelining**, we are now entering an era where programmers must switch to more **parallel programming models** in order to exploit multi-processors effectively, if they desire improved single-program performance.”

“This is because there are only three real ‘dimensions’ to processor performance increases beyond Moore’s law: clock **frequency**, **superscalar instruction** issue, and **multiprocessing**. We have pushed the first two to their logical limits and must now embrace multiprocessing, even if it means that programmers will be forced to change to a parallel programming model to achieve the highest possible performance.”

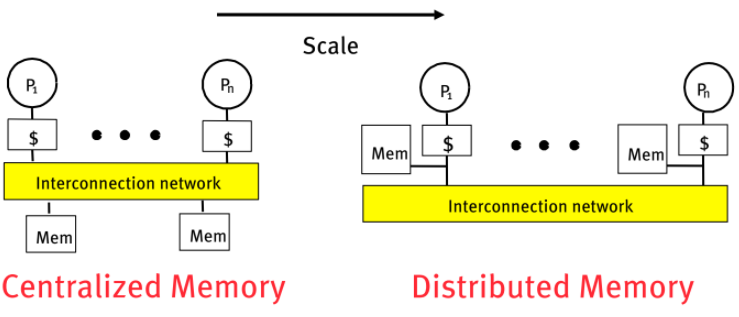
### Chip Multiprocessing (CMP)

CMP utilises two or more conventional superscalar processors on the **same die**.

Benefits:

1. **Volume**: Two processors where one was before.
2. **Power**: Processors on the same die share a single connection to the rest of the system.

### Centralised vs. Distributed Memory



# Programming Models

A programming model is made up of the languages and libraries that create an abstract view of the machine.

**Control**:

* How is **parallelism** created?
* What **orderings** exist between operations?
* How do different threads of control **synchronise**?

**Data**:

* What data is private vs. shared?
* How is logically shared data accessed / communicated with?

**Synchronisation**:

* What operations can be used to coordinate parallelism?
* What are the atomic operations?

### Programming Models

|  |
| --- |
| **Model 1A - Shared Memory**:  Processors are all connected to a large shared memory - SMPs.    Difficulty scaling to a large number of processors (≥32).  **Advantage**: Uniform Memory Access (UMA).  **Cost**: Cheaper to access data in **cache** than in memory. |

|  |
| --- |
| **Model 1B - Multithreaded Processor**:  Multiple thread contexts without full processors.  Memory and some other state is shared. |
| **Model 1C - Distributed Shared Memory**:  Memory is **logically shared**, but physically distributed.   * Any processor can access any address in memory. * Cache lines / pages are passed around the machine.     **Limitation**: Cache coherency protocols - how to keep cached copies of the same address consistent? |

|  |
| --- |
| **Model 2 - Message Passing**:  Program consists of a collection of named processes.   * Fixed at program startup time. * Thread of control plus local address space - No shared data. * Logically shared data is partitioned over local processes.   Processes communicate by explicit send / receive pairs.   * Coordination is implicit in every communication event. * MPI (Message Passing Interface) is the most commonly used software. |
| **Model 2A - Distributed Memory**:  Each processor has its own memory and cache but cannot directly access another processor’s memory.  Each node has a network interface (NI) for all communication and synchronisation. |
| **Model 2B - Internet / Grid Computing**: |

|  |
| --- |
| **Model 2C - Global Address Space**:  Program consists of a collection of named threads.   * Fixed at program startup time. * Local and shared data, as in the shared memory model.   + Shared data is partitioned over local processes - remote data is expensive.   Intermediate point between message passing and shared memory.    Network interface supports **RDMA (Remote Direct Memory Access)**:   * NI can directly access memory without interrupting the CPU. * One processor can read / write memory with one-sided operations (put / get). * Not just a load / store as on a shared memory machine - can **continue computing** while waiting for a memory operation to complete. * Remote data is typically **not** cached locally. |
| **Model 3 - Data Parallel**:  Single thread of control consisting of parallel operations.   * Communication and coordination is implicit in parallel operations. * Elegant and easy to understand.   Disadvantages:   * Not all problems fit this model. * Difficult to map onto coarse-grained machines. |
| **Model 4 - Hybrids**:  Programming models can be mixed:   * Message passing (MPI) at the top level with shared memory within a node is common. * Can mix data parallel and threads in a global address space. * Global address space models can call message passing libraries or vice-versa. * Global address models can be used with:   + Shared memory when it exists in hardware.   + Communication (done by the runtime system). |

|  |
| --- |
| **Model 4 - Clusters of SMPs**:  Symmetric multiprocessors are the fastest commodity machine and can be used for a larger machine with a network.  Common names:   * CLUMP = Cluster of SMPs * Hierarchical machines * Constellations   Shared memory **within** one SMP, but message passing outside. |

### MPI - Message Passing Interface

MPI has become the standard for parallel computing using message passing.

Pros and cons of standards:

|  |  |
| --- | --- |
| **Advantages**: | **Disadvantages**: |
| **✓** Created a **standard** for application development in the HPC (High-Performance Computing) community - portability. | **✗** MPI is a least-common denominator building on mid-80s technology, which may discourage innovation. |

### Race Conditions

A race condition / data race occurs when:

* Two processors or threads access the same data, and at least one is a **write** operation.
* The accesses are **concurrent**.

### Problems with Scaling Shared Memory Hardware

Why not put more processors on with larger memory?

1. The memory bus becomes a bottleneck.
2. Caches need to be kept coherent.

# Parallel Processing

### Challenges of Parallel Processing

**Application parallelism**: New algorithms that have better parallel performance.

Reduce frequency of remote accesses by:

1. Caching shared data (hardware).
2. Restructuring the data to make more accesses local (software).

### Symmetric Shared-Memory Architectures

Caches:

* Private data is used by a single processor.
* Shared data is used by multiple processors.

Caching shared data:

* Reduces latency to shared data, memory bandwidth for shared data and interconnect bandwidth.
* Introduces a cache-coherence problem.

### 

# Cache Coherence

**Coherence** defines behaviour for the **same** processor.

**Consistency** defines behaviours for **other** processors.

**Write serialisation**: Two writes to the same location by any two processors are seen in the **same order** by all processors.

SMPs use a **hardware protocol** to maintain coherent caches. **Migration** and **replication** are key to **performance** of shared data

**Migration**:

Data can be **moved to a local cache** and used there in a transparent fashion. This reduces both **latency** to access shared data that is allocated remotely and **bandwidth demand** on the shared memory.

**Replication**:

Used for reading shared data simultaneously since caches make a copy of data in the local cache. It reduces both **latency** of access and **contention** for reading shared data.

### Cache-Coherence Protocols

**Directory based**:

The sharing status of a block of physical memory is kept in just one location; the **directory**.

**Snooping**:

Every cache with a copy of data also has a copy of the sharing status of the block, but **no centralised state** is kept.

* All caches are accessible via a **bus** or switch.
* All cache controllers monitor / snoop on the medium to determine whether or not they have a copy of a block that is requested on the bus.

### Snoopy Cache-Coherence Protocols

The cache controller snoops **all** transactions on the shared medium.

If a transaction concerns data in its cache, **invalidate** the value and **update** it, or instead **supply** the value (depending on the state of the block and protocol).

Either get **exclusive access** before writing via **write invalidate**, or **update all copies** on write.

### Cache Performance

Cache performance is a combination of:

* Uniprocessor cache miss traffic.
* Traffic caused by communication - results in invalidations and subsequent misses.

### Coherency Misses

**True sharing misses** arise from the communication of data through the cache coherence mechanism.

* Invalidates due to **first write** to shared block.
* Reads by another CPU of modified block in a different cache.

**False sharing** misses occur when a block is **invalidated** because some word in the block, **other** than the one being read, is written to.

* Invalidation does **not** cause a new value to be communicated, but only causes an extra cache miss.
* Block is shared, but no word in block is actually shared.

### Bus-Based Coherence

The faulting processor must send out a request, which others respond to and take necessary action.

It could be done in a scalable network - conceptually simple, but **doesn’t scale well**. On a scalable network, each fault leads to at least network transactions.

**Scalable coherence**:

* Can have the same cache states and state transition diagram.
* Different mechanisms to manage protocol.

### Scalable Approach - Directories

Every memory block has associated directory information:

* Keeps track of copies of cached blocks and their states.
* On a **miss**, finds the directory entry, looks it up, and communicates only with the nodes that have copies if necessary.
* In scalable networks, communication is through **network transactions**.

### Operation of Directories

**Read** from main memory by processor i:

* If dirty bit is off then:
  + Read from main memory
  + Turn p[i] on
* If dirty bit is on then:
  + Recall line from dirty processor (cache state to shared)
  + Update memory
  + Turn dirty bit off
  + Turn p[i] on
  + Supply recalled data to i

**Write** to main memory by processor i:

* If dirty bit is off then:
  + Supply data to i
  + Send invalidations to all caches that have the block
  + Turn dirty bit on
  + Turn p[i] on

### Memory Consistency Models

**Sequential consistency**: The **result** of any execution is the same as if the accesses of each processor were kept **in order** and the accesses among different processors were interleaved.

Delay all memory accesses until all **invalidations** are done.

Schemes faster execution to sequential consistency.

Not an issue for most programs as they are synchronised.

Only those programs willing to be non-deterministic are not synchronised.

### Relaxed Consistency Models

Allow read and writes to complete **out of order**, but to use synchronisation operations to enforce ordering so that a synchronised program behaves *as if* the processor were sequentially consistent.

* By relaxing orderings, it may obtain **performance advantages**.
* Also specifies range of **legal compiler optimisations** on shared data.
* Unless **synchronisation points** are clearly defined and programs are synchronised, the compiler could not interchange read and write of two shared data items because it might affect the **semantics** of the program.

There are three major sets of relaxed orderings:

1. **W→R ordering** (**all** writes completed before the next read).  
   Because it retains ordering among writes, many programs that operate under sequential consistency operate under this model *without* additional synchronisation. This is called **processor consistency**.
2. **W→W ordering** (all writes completed before the next write).
3. **R→W and R→R orderings**  
   A variety models depending on ordering restrictions and how synchronisation operations enforce ordering.

There are many complexities in relaxed consistency models:

* Defining *precisely* what it means for a write to complete.
* Deciding **when** processors can see values that it has written.

### Mark Hill Observation

Use **speculation** to hide latency from the strict consistency model. If a processor receives invalidation for a memory reference before it is committed, the processor uses **speculation recovery** to back out of a computation and restart it with an **invalidated** memory reference.

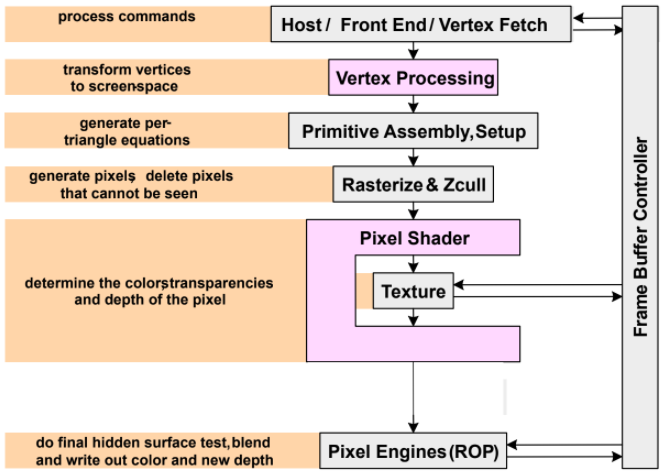
Advantages:

1. Aggressive implementation of sequential consistency or processor consistency gains *most* of the advantages of more relaxed models.
2. Implementation adds little to the implementation cost of a speculative processor.
3. It allows the programmer to reason using the simpler programming models.

14 GPU Computing Architecture

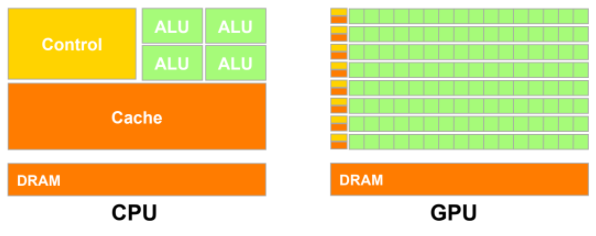
A GPU is a **Graphics Processing Unit**:

* Accelerator for raster-based graphics (OpenGL, DirectX).
* Highly programmable.
* 100s of ALUs & 10000s of concurrent threads.



### GPUs for Computing

A GPU uses a larger fraction of silicon for computation than a CPU.



At peak performance, a GPU uses an order of magnitude **less energy** per operation than a CPU.

### GPGPUs vs. Vector Processors

There are similarities on the **hardware level** between GPUs and vector processors.

### GPGPU Programming Model

The CPU offloads parallel **kernels** to the GPU:

1. Transfers data to GPU memory.
2. GPU hardware spawns **threads**.
3. Transfers result back to CPU main memory.

**Kernels** / **compute shaders** are routines compiled for high throughput accelerators (e.g. GPUs). They roughly correspond to inner loops when implementing algorithms in traditional languages.

### CUDA / OpenCL Threading Model

The CPU spawns a **fork-join** style **grid** of parallel threads. The CPU spawns more threads than the GPU can run. The excess threads may wait.

Threads are organised into **blocks** of up to 1024 threads:

* Threads can communicate with other threads in the block.
* Threads and blocks have an identifier, and can be 1 / 2 / 3 dimensional.

Each **kernel** spawns a **grid** containing one or more thread blocks.

### SIMT Execution Model

Programmers see MIMD threads (scalar).

The GPU bundles threads into **warps** (wavefronts):

* The most basic unit of scheduling for a GPU.
* NVIDIA warps consist of 32 threads, AMD contains 64 threads.
* They are run in **lockstep** (parallel)on **SIMD** hardware.

How to handle branch operations when different threads in a warp follow different paths through a program? **Serialise** different paths.

### GPU Memory Address Spaces

A GPU has three **address spaces** to support increasing the visibility of data between threads:

1. Local
2. Shared
3. Global

There are two more **read-only** address spaces:

1. Constant
2. Texture

|  |
| --- |
| **Local (Private) Address Space**:  Each thread has its own local / private memory.  It contains **local variables** private to that thread. |
| **Shared (Local) Address Space**:  Each thread in the same thread block can access a memory region called shared memory.  Shared memory is limited in size (16 - 48KB).  Used as a software-managed cache to avoid off-chip memory accesses.  Threads in a thread block are synchronised using \_\_syncthreads(); |
| **Global Address Space**:  Each thread in the different thread blocks can access a region called global memory.  GPGPU workloads can write their **own portion** of global memory. This avoids the need for synchronisation, which is slow, and unpredictable thread block scheduling.  **Coalescing Global Addresses**:  Not the same as CPU write combining / buffering:   * Aligned accesses request a single 128B cache block. * Memory divergence. |

### Bank Conflicts

In order to increase bandwidth, it is common to organise memory into multiple **banks**.

**Independent** accesses to different banks can proceed in **parallel**.

### CUDA Streams

CUDA and OpenCL provide the capability to overlap **computation on GPU** with **memory transfers** using **streams**.

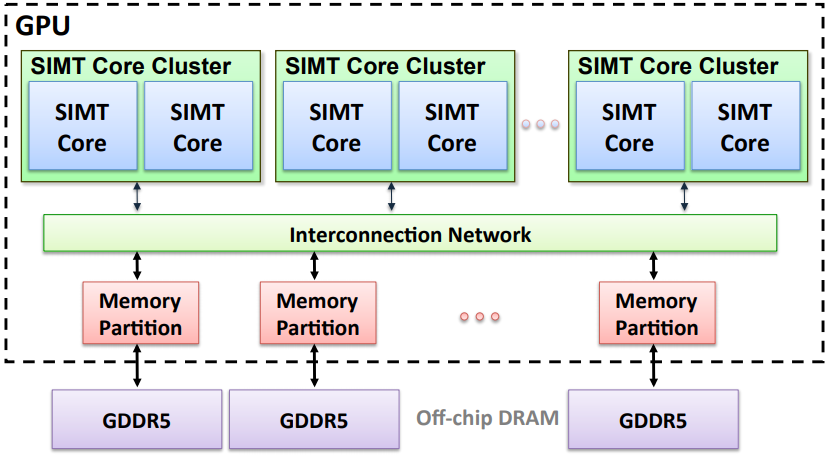
A stream orders a sequence of **kernels** and **memory copy operations**. Operations in one stream can overlap with operations in a different stream.

### GPU Instruction Set Architecture (ISA)

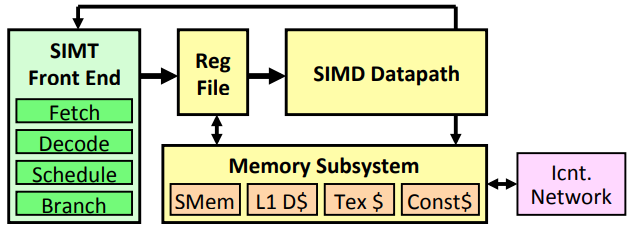
PTX (Parallel Thread eXecution) is a virtual ISA by NVIDIA, which is a reduced instruction set architecture with an **infinite** set of registers.

PTX is translated to **hardware ISA** by a backend compiler either at compile time, or at runtime with the GPU driver.

# GPU Microarchitecture Overview



### SIMT Cores



SIMT (Single Instruction, Multiple Threads) frontend / SIMD backend.

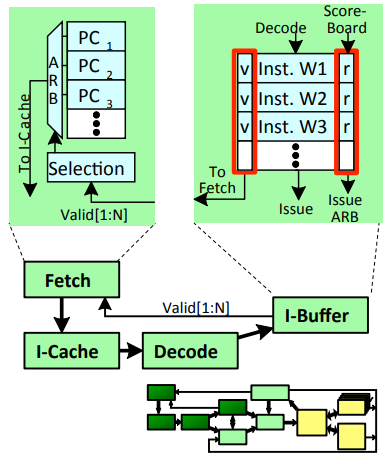
SIMT = SIMD execution of scalar threads.

Uses **fine-grained** multithreading:

* **Interleaves** warp execution to hide latency.
* Register values of all threads stays in the core.

### Fetch & Decode

1. Arbitrate the l-Cache among warps. Cache misses are handled by fetching again later.
2. The fetched instruction is **decoded** and then stored in the **I-Buffer**.
   * One or more entries per warp.
   * Only warps with **vacant** entries are considered during fetch.



### Instruction Issue

1. Select a warp, and issue an instruction from its I-Buffer for execution.
   * Scheduling uses **Greedy-Then-Oldest (GTO)**.
   * To avoid stalling the pipeline, instruction may be kept in the I-Buffer until it knows it can complete (replay).

### SIMT Using a Hardware Stack

An **execution mask stack** is implemented with special instructions to push / pop.



In practice, augment stack with **prediction** to lower overhead.

### Register File

* 32 warps
* 32 threads per warp
* 16x 32-bit registers per thread = 64KB register file

Needs four ports, which greatly increases its area.

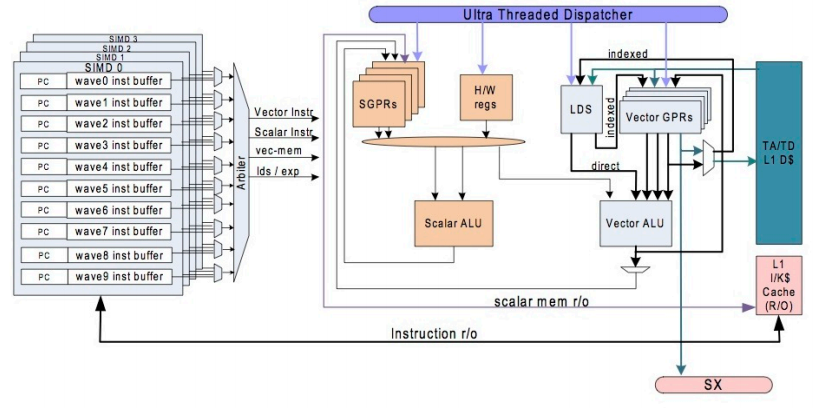
*Alternative*: **Banked** single-ported register file - reduced area, access time and power consumption. How do we avoid bank conflicts?

### Banked Register File

|  |  |
| --- | --- |
|  |  |

### AMD Southern Islands SIMT Core

This is an ISA visible scalar unit that executes computation identical across SIMT threads in a wavefront.



15 Multimedia Instruction Sets

General-purpose processors and ISAs are **not** sufficient for multimedia:

1. Performance
2. Power consumption
3. Cost

Characteristics of multimedia applications:

* Requirement for **real-time** response:
  + **Incorrect** result is preferable to a slow result.
  + Unpredictability can be bad (e.g. dynamic execution).
* **Narrow** data types:
  + Typical width of data in memory = 8 - 16-bit
  + Typical width of data during computation = 16 - 32-bit
  + 64-bit data types are **rarely required**.
  + **Fixed-point arithmetic** often replaces floating point.
* Fine-grain (data) parallelism:
  + **Identical operation** applied to streams of input data:
    - Branches have **high predictability**.
    - High **instruction locality** in small loops or kernels.
* Coarse-grain parallelism:
  + Most apps are organised as a pipeline of functions.
  + Multiple threads of execution can be used.
* Memory requirements:
  + **High bandwidth** requirements, but can tolerate **high latency**.
  + **High spatial locality** (predictable pattern), but **low temporal locality**.
  + Cache bypassing and prefetching can be crucial.

# SIMD Extensions for GPP

Motivation:

* Low media-processing performance of GPPs.
* Cost and lack of flexibility of specialised ASICs for graphics / video.
* Underutilised datapaths and registers.

Sub-word parallelism:

* Treat a 64-bit register as a vector of 2x 32-bit / 4x 16-bit / 8x 8-bit values.
* Partition 64-bit datapaths to handle multiple narrow operations in parallel.

Initial constraints:

* No additional architecture state (registers).
* No additional exceptions.
* Minimum area overhead.

### SIMD Operations

Integer arithmetic:

* Addition and subtraction with saturation.
* Fixed-point rounding modes for multiply and shift.
* Sum of absolute differences.
* Multiply-add, multiplication with reduction.
* Mix, max.

Floating-point arithmetic:

* Packed floating-point operations.
* Square root, reciprocal.
* Exception masks.

Data communication:

* Merge, insert, extract.
* Pack, unpack (width conversion).
* Permute, shuffle.

Comparisons:

* Integer and floating-point packed comparison.
* Compare absolute values.
* Element masks and bit vectors.

Memory:

* No new load-store instructions for short vector - no support for strides or indexing.
* Short vectors are handled with 64-bit load and store instructions.
* Pack, unpack, shift, rotate, shuffle to handle alignment of narrow data types within a wider one.
* Prefetch instructions for utilising temporal locality.

### Programming with SIMD Extensions

Optimised shared libraries:

* Written in **assembly**, distributed by a vendor.
* Need well defined APIs for data format and use.

Language macros for variables and operations:

* C / C++ wrappers for short vector variables and function calls.
* Allow instruction scheduling and register allocation optimisations for specific processors.
* Lack of portability, and is non-standard.

Compilers for SIMD extensions:

* No commercially available compiler so far.
* Problems:
  + Language support for expressing fixed-point arithmetic and SIMD parallelism.
  + Complicated model for loading / storing vectors.
  + Frequent updates.
* Assembly coding.

# Vector Processors

Initially developed for supercomputing applications, vector processors have high-level operations that work on **linear arrays** of numbers (vectors).

**Properties**:

1. Single-vector instruction implies lots of work (loop) - **fewer instruction fetches**.
2. Each result is **independent** of the previous result.
   * Compiler ensures **no dependencies**.
   * Multiple **operations** can be executed in **parallel**.
   * Simpler design, high clock rate.
3. Reduces branches and branch problems in pipelines.
4. Vector instructions access memory with a **known pattern**:
   * Effective **prefetching**.
   * Amortise memory latency over large number of elements.
   * Can exploit a high-bandwidth memory system.
   * No data caches are required.

### Styles of Vector Architectures

|  |
| --- |
| **Memory-Memory Vector Processors**:  All vector operations are memory to memory. |
| **Vector-Register Processors**:  All vector operations are between vector registers, **except for load and store**.  This is the vector equivalent of load-store architectures. |

### Components of a Vector Processor

|  |
| --- |
| **Scalar CPU**:   * Registers * Datapaths * Instruction fetch logic |
| **Vector Registers**:   * Fixed-length memory bank holding a single vector. * Has at least two read and one write port. * Typically 8 - 32 vector registers, each holding 1 - 8 Kb. * Can be viewed as an **array** of 64 / 32 / 16 / 8-bit elements. |
| **Vector Functional Units (FUs)**:   * Fully pipelined - starts a new operation every clock cycle. * Typically 2 - 8 FUs (integer and floating-point). * Multiple datapaths (pipelines) used for each unit to process multiple elements per cycle. |
| **Vector Load-Store Units (LSUs)**:   * Fully pipelined unit to load / store a vector. * Multiple elements fetched / stored per clock cycle. * May have multiple LSUs. |
| **Crossbar**:   * Connects FUs, LSUs and registers. |

### Vector Memory Operations

Load / store operations move groups of data between registers and memory.

Three types of addressing:

1. Unit stride - fastest
2. Non-unit (constant) stride
3. Indexed (**gather-scatter**)
   * Vector equivalent of register indirect.
   * Good for sparse arrays of data.
   * Increases number of programs that vectorise.

Support for various combinations of data widths in memory and registers.

The **Vector Length (VL) register** controls the length of any vector operation (0 - MVL [Maximum Vector Length]).

### Choosing the Data Type Width

1. **Separate instructions** for each width:
   * Popular with SIMD extensions for GPPs.
   * Uses too many opcodes.
2. Specify it in a control register:
   * Virtual Processor Width (VPW).
   * Updated only on **width changes**.

MVL increases when VPW gets narrower. Always pick the **narrowest VPW** needed by the application.

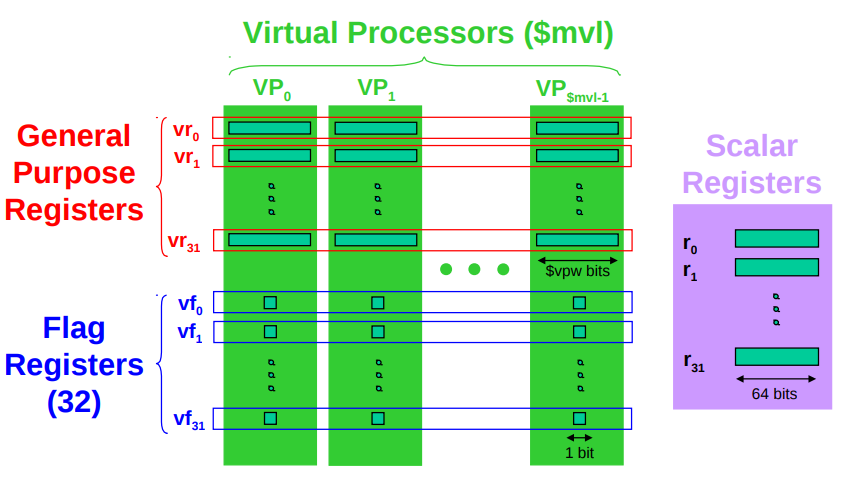
### Other Features for Multimedia

|  |
| --- |
| **Support for Fixed-Point Arithmetic**:  Saturation / rounding modes. |
| **Permutation Instructions of Vector Registers**:  For reductions and FFTs, not general permutations as it is too expensive. |

### Optimisations

|  |
| --- |
| **Chaining**:  A vector register is not a single entity, but a group of individual registers.  Pipeline forwarding can work on **individual** vector elements.  **Flexible chaining** allows a vector to chain to any other active vector operations ⇒ more read / write ports. |
| **Multi-Lane Implementation**:  Elements for vector registers are interleaved across the lanes.   * Each lane receives identical control. * Multiple element operations are executed each cycle. * Modular & scalable design. * No need for interlane communication for most vector instructions. |
| **Conditional Execution**:  Add vector flag registers with single-bit elements, and a vector compare to set a flag register.  Use the flag register as a mask control for the vector - instruction is executed only for vector elements with the corresponding element set. |

### Vector Architecture State



### Methods of Vectorisation

|  |
| --- |
| **Inner Loop Vectorisation**:  Good for vectorising single-dimension arrays or regular kernels. |
| **Outer Loop Vectorisation**:  Good for irregular kernels or kernels with loop-carried dependencies in the inner loop. |

These are compiler perspectives - the **hardware is identical** for both.

### Designing a Vector Processor

Design decisions to make:

* Changes to the **scalar core**.
* Picking the **maximum vector length (MVL)**.
* Picking the number of **vector registers**.
* Overhead of **context switches**.
* **Exception handling**.
* Masking and flag instructions.

|  |
| --- |
| **Changes to the Scalar Core**:   * Decode **vector** instructions. * Send **scalar registers** to the vector unit (vector-scalar operations). * Synchronisation for results back from vector register, including **exceptions**. * Things that don’t run in vector don’t have high ILP, so make the scalar CPU **simple**. |
| **Context Switch Overhead**:  The vector register file holds a huge amount of architectural state. This is too expensive to save and restore on each context switch.   * Extra **dirty bit** per processor - if vector registers are **not** written, **don’t save**. * Extra **valid bit** per vector register, which is cleared on process start.   Save / restore vector state only if the context needs to issue vector instructions. |

# Exception Handling

### Arithmetic

Arithmetic traps are hard.

Precise interrupts cause a large performance loss. Multimedia applications don’t care much about arithmetic traps anyway.

Alternative model:

* Store exception information in the vector flag registers.
* A “set flag” bit indicates that the corresponding operation caused an exception.
* Software inserts trap barrier instructions from software to check the flag bits if needed.
* IEEE floating point requires 5 flag registers (5 types of traps).

### Page Faults

Page faults must be precise. Instruction page faults are not a problem, data page faults are harder.

|  |
| --- |
| **Option 1**:  Save / restore internal vector unit state:   1. Freeze pipelines (dumping all vector state). 2. Fix fault. 3. Restore state. 4. Continue vector pipeline. |
| **Option 2**:  Expand memory pipeline to check all addresses before sending to memory:   * Requires address and instruction **buffers** to avoid stalls during address checks. * On a page fault, only needs to save the state in those buffers. * Instructions that have cleared the buffer can be allowed to complete. |

### Interrupts

Interrupts that occur due to external sources (e.g. I/O, timers).

These are handled by the **scalar core**.

Should the vector unit be interrupted?

* Not immediately (no context switch).
* Only if it **causes an exception**, or if the interrupt handler needs to execute a vector instruction.

### Why Use Vectors for Multimedia?

1. Natural match to **parallelism** in multimedia:
   * Vector operations with vector length of the image or frame width.
   * Easy to efficiently support vectors of **narrow** data types.
2. High performance at low cost:
   * Multiple operations per cycle while issuing one instruction per cycle.
   * Multiple operations per cycle at low power consumption.
   * Structured access pattern for registers and memory.
3. Scalable:
   * Higher performance by adding lanes - no architecture modifications.
4. Compact code size:
   * Describe N operations with one short instruction (VLIW).
5. Predictable performance:
   * No need for caches or dynamic execution.
6. Mature, developed compiler technology.

### Comparison with SIMD

1. More **scalable**:
   * Can use double the hardware (datapaths / registers) without modifying the architecture or increasing instruction issue bandwidth.
2. Simpler hardware:
   * A simpler scalar core is enough.
   * Multiple operations per instruction.
3. Full support for vector loads and stores.
   * No overhead for alignment or data width mismatch.
4. Mature compiler technology:
   * Although language problems are similar.

**Disadvantages**:

* Complexity of exception model.
* Out of fashion.

16 Instruction Scheduling

**Scheduling** is the act of finding **independent** instructions:

* **Static**: Done at compile time by the compiler (software).
* **Dynamic**: Done at runtime by the processor (hardware).

Why schedule code?

|  |
| --- |
| **Scalar Pipelines**:  Fill in load-to-use **delay slots** to improve CPI (Clock cycles per instruction). |
| **Superscalar**:  Place **independent** instructions together.  Again, fill load-to-use delay slots.  Allow multiple-issue decode logic to let them execute at the same time. |

### Dynamically-Scheduled Processors

Also called **out-of-order** processors, has **hardware** reschedule instructions within a **sliding window** of instructions. As with pipelining and superscalar, ISA is unchanged.

This increases scheduling scope:

* Loop unrolling is done transparently.
* Uses branch prediction to unroll branches.

Each cycle, the next **ready** instructions are selected from the window of instructions.

In order to enable **out-of-order** execution:

1. **Register renaming** to avoid false dependencies.
2. **Dynamically schedule** to enforce true dependencies.

### Dependence Types

1. **RAW** (Read After Write) - true dependence (true)
2. **WAW** (Write After Write) - output dependence (false)
3. **WAR** (Write After Read) - anti-dependence (false)

### Register Renaming

Done to eliminate **register conflicts** / hazards.

Architected vs. physical registers - level of indirection.

* Names: r1, r2, r3, ...
* Locations: p1, p2, p3, …
* Original mapping: r1 → p1, r2 → p2, …
  + p4 - p7 are available

Renaming - Conceptually write each register once

* Removes false dependencies
* Leaves **true dependencies** intact.

When to reuse a physical register? After the **overwriting instruction** is complete.

### Dynamic Scheduling

Instructions are fetched / decoded / renamed into the instruction buffer (instruction window / instruction scheduler).

Instructions conceptually check the ready bits every cycle. Execute the oldest ready instruction, set output as “ready”.

### Dynamic Scheduling / Issue Algorithm

|  |
| --- |
| for (i in instructions) {  if (table[i.physInput1] == "ready" && table[i.physInput2] == "ready") {  i.status = "ready";  }  selectOldestReadyInstruction();  table[i.physOutput] = "ready"; } |

For an instruction with latency of , set the “ready” bit cycles in the future.

### Register Renaming

At decode stage for each instruction, rewrite the instruction with physical registers instead of architectural registers.

|  |
| --- |
| i.physInput1 = mapTable[i.archInput1]; i.physInput2 = mapTable[i.archlInput2];  newReg = newPhysicalReg();  mapTable[i.archOutput] = newReg; i.physOutput = newReg; |

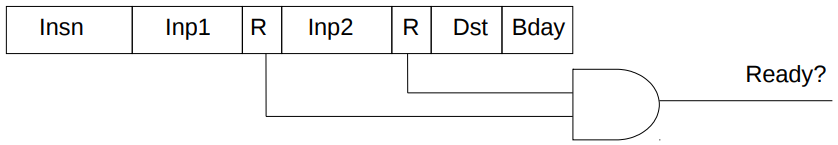
### Dispatch

Puts renamed instructions into out-of-order structures.

A **reorder buffer** (ROB) holds instructions from fetch through commit.

The issue queue:

* Holds instructions from dispatch through issue.
* Tracks ready inputs:
  + Physical register names
  + Ready bit
  + ANDs these bits to tell if the instruction is ready.



Dispatch steps:

1. Allocate issue queue (IQ) slot. If full, **stall**.
2. Read ready bits of inputs - 1 bit per physical register.
3. Clear ready bit of output in table - instruction has not produced a value yet.
4. Write data into the IQ slot.

### Out-of-Order Pipeline

Selects **ready** instructions and sends them for execution.

Wakes up dependents.

### Issue

Issue = Select + Wakeup

Selects the oldest of the ready instructions.

Wake up dependent instructions:

1. Search for destination in inputs, and set “ready” bit.
   * Implemented with a special memory array circuit called **Content-Addressable Memory (CAM)**.
2. Update the ready-bit table for future instructions.

For multi-cycle operations (loads, floating point):

* Wakeup is deferred for a few cycles.
* Includes checks to avoid structural hazards.

### Content-Addressable Memory (CAM)

CAM is indexed by the content of each location, not by address. This is sometimes known as an **associative memory**.

It compares an input key against a table of keys, and returns the location of the key in the table.

* In software, this might be a hash table.
* In hardware, it is possible but potentially slow.

To search all locations in a single cycle, a lot of hardware is required. Fast CAMs are expensive.

### Issue

Select + Wakeup in one cycle.

Dependent instructions are executed on back-to-back cycles.

Issued instructions are removed from the IQ (issue queue). This frees up space for subsequent instructions.

### Out-of-Order Benefits and Challenges

|  |  |
| --- | --- |
| **Benefits** | **Challenges** |
| Allows **speculative reordering**:   * Loads / stores * Branch prediction.   Done by **hardware**:   * Compiler may want a different schedule for different hardware. * Hardware has only its own configuration to deal with.   Schedule can change due to cache misses.  Memory-level parallelism:   * Executes around cache misses to find independent instructions. * Finds and initiates independent misses, reducing memory latency - especially good at hiding L2 hits. | Design complexity:   * More **complicated** than in-order. * However, we have managed to overcome the design complexity.   Clock frequency:   * A high ILP machine at high frequency can be built with additional pipe stages and clever design.   Limits to efficiently scaling the window and ILP:   * Large physical register file. * Fast renaming / wakeup / select / load queue / store queue - active areas of research. * Branch & memory dependency prediction (limits effective window size). * Power efficiency - mobile phone chips use Out-of-Order cores. |

### Hardware vs. Software Scheduling

|  |  |
| --- | --- |
| **Software** | **Hardware** |
| Static scheduling is done by the **compiler**. It is limited in several ways. | Dynamic scheduling is done by **hardware** in order to overcome the limitations. |
| **Static Limitations**: | **Dynamic Mitigations**: |
| **✗** Number of registers in the ISA | **✓** Register renaming |
| **✗** Scheduling scope | **✓** Branch prediction & speculation |
| **✗** Inexact memory aliasing information | **✓** Speculative memory operations |
| **✗** Unknown latencies of cache misses | **✓** Execute when ready |

The compiler does what it can, hardware does the rest:

* Dynamic scheduling is needed to sustain more than two-way issue.
* Helps with hiding memory latency (execute around misses).

17 Asynchronous Programming

**Concurrency** is when two or more tasks can start, run and complete in overlapping time periods. It doesn’t necessarily mean they’ll ever both be running at the same instant.

**Parallelism** is when tasks are running at the **same time**.

Parallelism can only take place when **hardware** allows it. The number of threads running at the same time is limited by the **number of cores**.

Concurrency can take place regardless of hardware capability as it is a **model of execution**.

**Time slicing**: Break down the execution of one thread into units of time and execute slices in some scheduling pattern. The OS can decided which threads should have a higher priority.

**Virtual threads** are a software construct, and do not directly map to hardware threads. The OS / kernel maintains control of a virtual thread.

**Concurrency in Programming Constructs**:

If a task is taking a long time, set its virtual thread to a low priority. Sleep the thread if necessary.

**Issues with threads**:

* Too much complexity.
* Debugging is difficult.
* Hardware dependant.
* Compilers cannot optimise well.
* OS has to provide decent constructs for threading.

### Coroutines

* Multiple entry points.
* Suspend and resume.
* **Non-preemptive**:
  + Voluntarily give up control.
  + When idle - waiting for something / finished a task.
* Cooperative multitasking - lets other tasks run for a while.

### Generators

Generators are co-routines for iterators:

* Returns a list of items, one at a time.
* Gives up control after returning an item.
* Raise an error if control is passed after last value has been returned.

### Asynchronous

Task A and task B:

* Do task B while task A is still ongoing.
* It is possible that A is dependent on an event - A is waiting.
* A is seen to be executing in the background, B in the foreground.
* Periodically check if A’s event has happened
  + If not, keep executing B.
  + If so, stop executing B and continue A.
* A **must** be a coroutine, B *can* be a coroutine.

We need at least **one thread**.

Saving state:

* Function call stack.
* Point of execution / interruption.
* Function state: variables.

We do not need hardware support because it is a software construct.

**Async** is used to define a function (routine) as fit for asynchronous execution (coroutine).

**Await** is used to indicate that something is waiting for an event. Control break can occur there.

### Threads vs. Coroutines

|  |  |
| --- | --- |
| **Threads** | **Coroutines** |
| Better for:   * **Parallel** execution. * Not worrying about execution control. * Performing the **same task** repeatedly. * Setting priority to repetitive tasks. * Handling **different events simultaneously**. | Better for:   * **Event-based** programming. * Having multiple points of execution and interruption. * Concurrent execution even on single threads. * Saving state of execution and resuming it later. |



### Combining Threads and Coroutines

Coroutines piggybacking on threads:

* Multiple threads means that multiple coroutines can execute simultaneously.
* Create a **pool of threads** to execute a **pool of coroutines**.
  + Each thread gets some coroutine and executes it.
  + When interrupted, it executes another.

