Lab 0

0.1 Introduction

0.1.1 Why parallel processing ?

The main aim of the development and implementation of computer architecture is the **increase of processing speed**. The simplest way seems to just **increase clock frequency**.
Effectively, in the **1990s–early 2000s**, performance improvements came mainly from cranking up clock speeds (e.g. Pentium 4 up to ~3.8 GHz).
When engineers tried to keep improving processors simply by **raising clock speed**, they ran into several **hard limits**. Increasing the operating frequency or voltage makes **power consumption rise dramatically**, because dynamic power scales with both frequency and the square of voltage.

$$\textbf{power} \approx C \times V^2 \times f$$

As a result, chips running above about 4–5 GHz on conventional silicon generate so much heat that they become impossible to cool with standard methods. At the same time, advances in manufacturing that shrank transistors to smaller process nodes introduced a new challenge often called the "**energy wall.**"
Leakage currents, small but constant flows of electricity even when a transistor is supposed to be off, became significant, and running at higher speeds meant wasting far more energy per instruction.
Another limitation comes from signal **propagation delay**. Within a chip, every signal must travel from one part of the die to another within a single clock cycle. While transistors scale down well, long interconnect wires do not, so simply pushing the clock higher means signals can no longer reach their destination in time.

Finally, there are diminishing returns from **deep pipelines**. Designers extended pipelines to achieve higher frequencies, as Intel famously did with the Pentium 4's **31 stages**.
But this strategy backfired: a mispredicted branch meant discarding dozens of in-flight stages, leading to poor real-world performance per clock even if the advertised frequency was high.

Together, these barriers explain why raw clock speed stopped being the main route to faster processors and why architects turned to **multicore (MIMD) and vector parallelism (SIMD) instead**.

0.1.2 Why RISC architectures are good for parallel processing ?

RISC (Reduced Instruction Set Computer) designs were originally created with simplicity, efficiency, and regularity in mind and they are well suited to multicore and vector-oriented designs.
They rely on a small, regular set of instructions, which reduces the complexity of decode logic, minimizes irregular corner cases, and keeps pipelines clean, an essential feature when the same core design is replicated many times across a chip.

Their uniform instruction format and load/store architecture also make it easier to build **efficient pipelines and parallelize execution**, ensuring that multiple cores can run with predictable behavior without wasting power on unnecessary complexity.

The straightforward design of RISC **instructions, with fixed lengths** and simple semantics, further makes it easy to integrate SIMD and vector extensions: operations that apply the same action to many pieces of data can be added naturally, without conflicting with a bulky legacy instruction set.

Finally, RISC architectures tend to deliver better energy efficiency, achieving higher performance per watt than more complex **CISC counterparts** (**eg. x86** – Intel, AMD). This efficiency is critical in multicore systems, since it allows designers to fit more cores within a given power budget while maintaining strong overall performance.

0.1.3 Why RISC-V in particular is attractive ?

RISC-V builds on the RISC philosophy with a clean, **minimal base instruction set of only around fifty instructions**, while **additional features** such as integer **multiplication**, **atomics**, **floating-point arithmetic**, **bit manipulation**, or the **powerful vector extension (RVV)** are provided as modular add-ons.

This **modularity** makes it easy to **tailor a chip to its purpose:** one design may be a tiny embedded controller, while another can be a high-performance processor with wide vector capabilities.

The RVV extension itself was designed from scratch for **scalability**. Unlike traditional fixed-width SIMD models such as Intel's SSE (128 bits) or AVX (256 bits), **RVV is length-agnostic**, meaning that the same program can run efficiently whether the hardware vector registers are 128, 256, or even 1024 bits wide. This flexibility is particularly valuable for systems-on-chip (SoCs) targeting different performance and power envelopes.

Because the instruction set is simple and modular, cores are smaller and consume less power, which allows designers to integrate many of them on a single die, from tiny microcontrollers to large-scale server processors with dozens of cores.

Another major advantage is that RISC-V comes **without the legacy baggage** of architectures like x86 or even ARM. It was created recently with modern parallelism in mind, so it avoids having to preserve outdated quirks purely for backward compatibility.

Finally, as an **open standard** with no licensing restrictions, RISC-V fosters a **vibrant ecosystem** where anyone can build cores, design accelerators, or experiment with custom extensions. This openness encourages innovation in areas such as **vector processing**, **AI acceleration**, and **massively multicore systems**, all without the legal or financial barriers of proprietary ISAs.

0.2 SIMD versus MIMD

There are two major modes of parallel processing commonly discussed in computer architecture: SIMD and MIMD.

SIMD, or **Single Instruction, Multiple Data**, refers to the execution model where one instruction operates on many data elements at the same time.

This approach is particularly effective for **vectorized workloads** such as graphics rendering, image and audio processing, or scientific calculations where the same operation must be applied repeatedly across large datasets.

Well-known examples include **ARM's NEON** instruction set and the **RISC-V Vector extension**, both of which allow a single instruction to handle multiple data items simultaneously.

MIMD, or **Multiple Instruction, Multiple Data,** takes a different approach. In this model, multiple instructions are executed in parallel, each working on its own set of data. Every core or thread can follow its own control flow independently of the others, which makes MIMD ideal for general-purpose multicore CPUs or GPU threads that may be running different kernels.

The following figure illustrates the information flows in these four types of processing architectures. Note that the last architecture type refers to multi-core CPUs that also include integrated vector instructions.

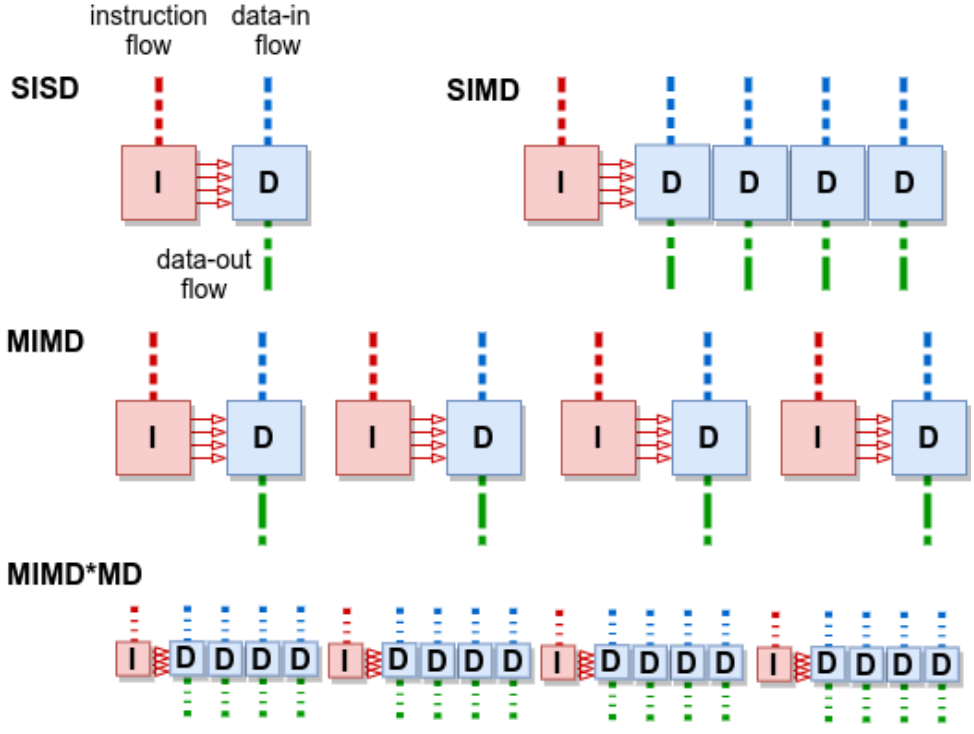

Fig 0.1 instruction flow(s) and data flow(s) in **SISD**, **SIMD**, **MIMD** , and **MIMD*MD** processing architectures

Let us explain in more details these architectures and operational modes.

0.2.1 SIMD operational mode

In the SIMD operational mode, a single control unit issues one instruction, and that **instruction is broadcast to multiple processing elements** (**PEs**). Each PE then executes the very same operation, but on its own individual data element. The data to be processed is usually drawn from **wide vector registers** or from memory blocks that **are accessed in parallel**, ensuring that multiple data items can be operated on simultaneously.

A simple example is the **addition of two arrays**: instead of performing one addition after another in sequence, each PE adds a different pair of elements at the same time under a single “add” instruction type **`vadd`**, thereby accelerating the computation dramatically.

To make this possible, several essential hardware components are required. At the center is the Control Unit (CU), which maintains a single control flow that is shared across all the PEs. Feeding the data to these PEs are **Vector Registers or Lanes**, which are wide registers (commonly 128, **256**, or 512 bits, and in RISC-V even scalable up to 1024 bits or more) that can hold multiple elements side by side.

Each lane of the vector register connects to its own Arithmetic Logic Unit (ALU), so that every lane can perform the same operation on its respective portion of data simultaneously. Supporting this setup are Interconnects and Data Paths, which deliver the same instruction stream but distribute different data elements to each ALU lane.

Finally, modern SIMD designs also include **Masking and Predication mechanisms**, which allow certain lanes to be enabled or disabled dynamically. This means that even under a uniform instruction, not every element needs to be processed, an important feature for handling edge cases, conditional operations, or irregular data sizes.

Altogether, SIMD combines one control flow with many parallel data paths, making it an efficient hardware model for vectorized computation such as multimedia, graphics, scientific workloads, and **modern AI processing**.

0.2.2 MIMD operational mode

In the MIMD operational mode, **each processor core or thread operates independently**, fetching and executing its own instruction stream. Unlike SIMD, where all units follow a single control flow, in MIMD **each core has its own control logic** and can work on a separate data set without being tied to what the other cores are doing.
This independence allows the system to run **different programs at the same** time or to divide a large application into distinct tasks that are processed concurrently, a principle known as task-level parallelism.
For example, in a modern game engine one core might handle **rendering graphics**, another core might process **artificial intelligence logic**, and a third core could **simulate physics** , all progressing simultaneously. Importantly, this does not exclude cooperation between cores: they can still process shared data structures such as large matrices, but they do so while running independent instruction streams.

To support this level of autonomy, MIMD architectures require a number of essential hardware components. Each core contains its own control unit responsible for fetching and decoding instructions, as well as its own register set for holding intermediate values.

Every core is also equipped with one or more ALUs or FPUs, giving it an independent execution pipeline. Around these pipelines sits a cache hierarchy, typically with **private L1 and L2 caches for each core** and often a **larger shared L3 cache** to allow communication and **reduce memory latency**.

The cores are tied together by an interconnect and memory system, which may rely on cache-coherence protocols or message-passing mechanisms to keep data consistent and enable collaboration. At a higher level, schedulers and operating system support are necessary to assign programs and threads to the available cores and manage their execution efficiently.

The hardware philosophy behind MIMD differs fundamentally from that of SIMD. In SIMD, the design replicates many execution units but keeps a single shared control unit, gaining efficiency from applying the same operation to multiple data elements. In MIMD, by contrast, the design replicates whole processors, including control units, registers, and pipelines , which provides far greater flexibility. This flexibility allows the system to exploit task-level parallelism, **running different instructions on different data simultaneously**, making MIMD ideal for general-purpose multicore CPUs and heterogeneous systems.

0.2.3 Lanes and Threads

The concepts of **lanes** in SIMD and **threads** in MIMD provide a useful way to understand how these two architectures organize their parallel work. In SIMD, computation is carried out through *lanes*, which are slices of a wide vector unit.
For example, **a 256-bit RISC-V vector register operating on 32-bit integers can be thought of as eight lanes**, each lane responsible for one element of the vector. All lanes are driven by a single instruction stream under the control of one central unit, meaning they execute in lockstep: when the instruction `vadd` is issued, every lane performs an addition at the same time.
Each lane processes a different data element, so **lane 0** might handle **element[0]**, **lane 1 element[1]**, and so on.
Because the control is unified, lanes cannot branch off independently. If a subset of elements should not participate in a given operation, masking or predication is used to selectively disable them while the rest continue.
Lanes are considered **lightweight execution units**: they are not full processors but rather duplicated **slices of arithmetic logic** that share the same register file and control infrastructure, allowing efficient parallelism across large blocks of data.

In contrast, threads in a MIMD system represent independent flows of execution with their own control, registers, and execution pipelines. Where SIMD lanes excel at **data-level parallelism**, threads enable **task-level parallelism**. Together, the distinction between **SIMD lanes and MIMD threads** highlights the different philosophies of parallel computing: tightly coupled lockstep execution versus independent, concurrent instruction streams.

The following figure illustrates the lane organization of 256-bit vector registers for different element sizes. Depending on the chosen data width (e.g., 8-bit, 16-bit, 32-bit, or 64-bit), the 256-bit vector is partitioned into a corresponding number of lanes. Each lane is processed independently and in parallel by a dedicated arithmetic logic unit (ALU) within the vector processing unit.

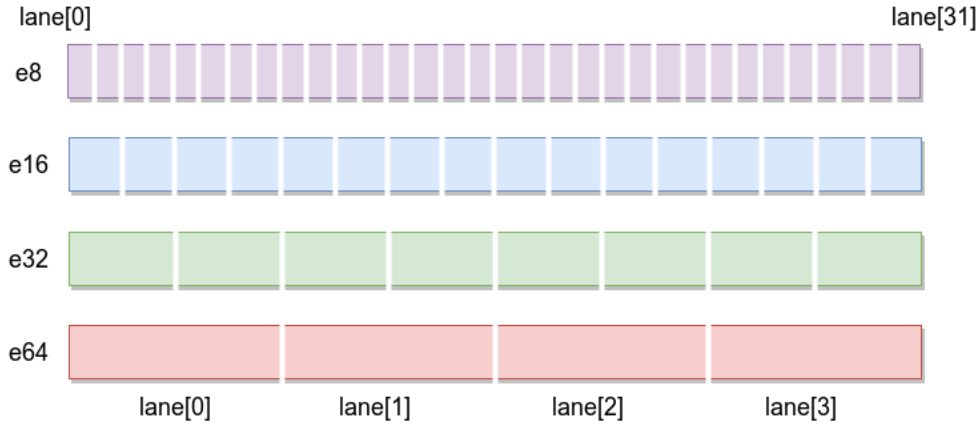

Fig 0.2 RISC-V vector register formats (256-bit) and SIMD processing **lanes** depending on element size

The following figure illustrates the organization of multi-threaded processing. Each thread is executed independently, using its own control unit to decode instructions and generate the corresponding control signals. A thread operates on its **own program counter**, **register file** and maintains a **private stack** and **heap memory space**, supported by a **local L1 cache**. Threads may share higher-level caches, such as the L2 cache.

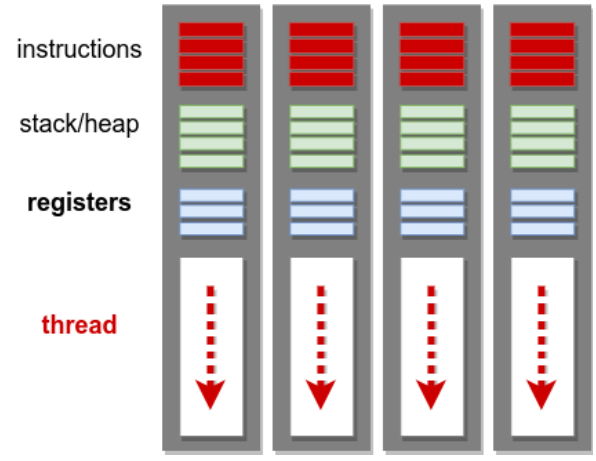

Fig 0.3 MIMD processing context (multi-core) with registers, stack/heap for independent threads

0.3 Data caches and parallel processing performance

Modern processors rely heavily on caches because of the fundamental **memory gap problem**. While processor clock cycles operate at **nanosecond speeds**, accessing **main DRAM can take hundreds of cycles**. If the CPU had to wait for every instruction or piece of data to arrive directly from memory, it would spend most of its time idle, with execution units underutilized.
To overcome this **gap**, designers introduced caches, which serve as **small but extremely fast storage** layers positioned close to the processor.

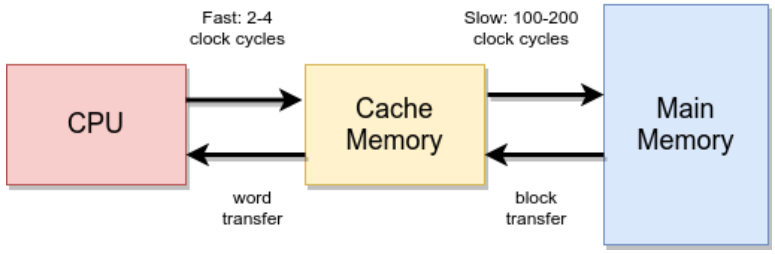

Fig 0.4 Acces time to cache versus main (RAM) memory

The effectiveness of caches comes from the principle of **locality of reference**. Programs generally reuse the same instructions and data repeatedly, known as **temporal locality**, and also tend to access memory locations that are physically close to one another, known as **spatial locality**.

Caches exploit **both patterns by holding recently used blocks of instructions and data**, so that when the CPU requests them again, they can be supplied almost instantly without a long trip to main memory. To handle these two types of reuse effectively, modern CPUs employ separate caches for instructions and data.

The **instruction cache (I-cache)** keeps recently fetched instructions, allowing the pipeline and branch predictor to operate smoothly without stalling for memory.
The **data cache (D-cache)** stores recently accessed operands so that load and store operations can complete quickly, keeping arithmetic units continuously busy.
Together, these caches form the essential bridge between the extremely fast cores and the much slower main memory, reducing latency, keeping pipelines filled, and ultimately making high-performance processing feasible.

0.3.1 How data caches affect SIMD (vector processing)

SIMD lanes are designed to execute the same instruction on multiple data items simultaneously, but this parallelism only delivers its full benefit if wide vectors can be kept constantly supplied with data. Caches play a crucial role in making this possible. When data arrays fit within the cache and are accessed sequentially, SIMD reaches very high throughput, since each cache line brings in several elements at once and every lane of the vector unit can remain active.

The situation changes when the **dataset is larger than the available cache**. In that case, c**ache misses occur more frequently**, forcing the processor to fetch data from main memory.
Because all SIMD lanes depend on this **shared flow of data**, a single miss can stall the entire vector unit, leaving all lanes idle while waiting for DRAM.

As a result, **SIMD performance is highly sensitive to both cache size and memory bandwidth**. When the **working set does not fit into cache**, the advantages of vectorization diminish sharply, as wide parallel hardware sits unused during memory stalls.

0.3.2 How data caches affect MIMD (multicore processing)

MIMD systems achieve parallelism by replicating entire processor cores, each with its own independent instruction stream. To support this independence, every core is typically equipped with its own **private caches**, such as **L1** and **L2** instruction and **data caches**, which allow it to fetch and store frequently used information without interfering with other cores.
Beyond these private layers, the cores eventually converge on **shared resources**, such as an **L3** cache and the main memory system.

Caches are highly **beneficial** in this setup: when each thread's **working set fits within its private caches**, the cores can operate largely in isolation, with little need for external communication or memory access. Under these conditions, parallel efficiency is high. However, as the number of active cores increases or as **individual workloads expand beyond the siz**e of their caches, **problems emerge.**

Capacity pressure arises when data for a thread no longer fits in its private caches, forcing **frequent fetches from shared memory**. At the same time, **bandwidth contention** develops as multiple cores try to access the shared cache or memory bus simultaneously, leading to stalls.
If threads also share data, the system **must maintain consistency through** a **cache coherence protocol**, which invalidates and updates cache lines across cores. This constant synchronization introduces **further overhead** and reduces effective performance.

The overall result is that MIMD efficiency declines gradually as more cores oversubscribe the cache and memory hierarchy.
Unlike SIMD systems, where **a single cache miss can stall every lane simultaneously**, in MIMD **some cores may continue to make progress** while others are delayed. Nevertheless, the system **as a whole slows down** due to contention, bandwidth saturation, and coherence traffic in the shared memory system.

0.4 Amdahl's Law

In parallel processing, **execution speedup** is a measure of performance improvement obtained when a program is executed on multiple processing units (e.g., vector units or multiple cores) compared to execution on a single processing unit. It is formally defined as:

$$S = \frac{T_1}{T_p}$$

where:

- **T1** is the execution time using a single processing unit (scalar or single-core execution),
- **Tp** is the execution time using p processing units (e.g., cores or vector lanes).

A speedup value **S>1** indicates a performance gain due to parallelization. Ideally, one might expect **linear speedup**, where **S=p**.
However, in practice, speedup is often sub-linear due to sequential portions of code, synchronization overheads, and hardware limitations.

This limitation is captured by **Amdahl's Law**, which expresses the theoretical maximum speedup achievable by parallelization. If a fraction α of a program is inherently sequential and cannot be parallelized, while the remaining fraction (**1–*f***) can be executed in parallel, then the maximum speedup on **N** processors is:

$$S(N) = \frac{1}{f + \frac{1-f}{N}}$$

Key implications:

- If ***f*>0**, the **speedup** is **bounded**, no matter how many processors are added.
- The larger the parallelizable portion (**1–*f***), the closer the speedup approaches the ideal linear case.
- Amdahl's Law highlights the importance of minimizing the sequential fraction of a program to achieve significant performance gains in parallel systems.

We will see that the resulting **speedup also depends on several factors**, including element size, vector length, and the functional characteristics of the processed example.

Example for **eight** processing cores and ***f*=0.1**:

$$S(8) = \frac{1}{0.1 + \frac{0.9}{8}} = \frac{1}{0.1 + 0.1125} = \frac{1}{0.2125} \approx 4.71$$

0.5 RISC-V paralel programming platforms (K1/X1 SoC - X60)

In this book, we use the modern multi-core **RV64GCV** architecture for parallel programming at both the vector (SIMD) and multi-core (MIMD) levels.

0.5.1 RISC-V K1 SoC architecture

RISC-V CPUs such as the **X60**, integrated into the **K1 SoC**, are based on standard modern architectures that support vector instructions with a vector register length of 256 bits.
This means we can execute vector instructions operating in parallel on either eight 32-bit integer or floating-point values, or on sixty-four 8-bit elements.
The K1 SoC integrates **eight X60 CPUs**, enabling the parallel execution of eight vector threads. This provides an effective parallel processing capacity equivalent to 8 × 8 = 64 integer or floating-point operations.

The boards run a Debian-based operating system, Bianbu OS, or Ubuntu OS.
The following figure shows the architectural block diagram of the X60 CPU .

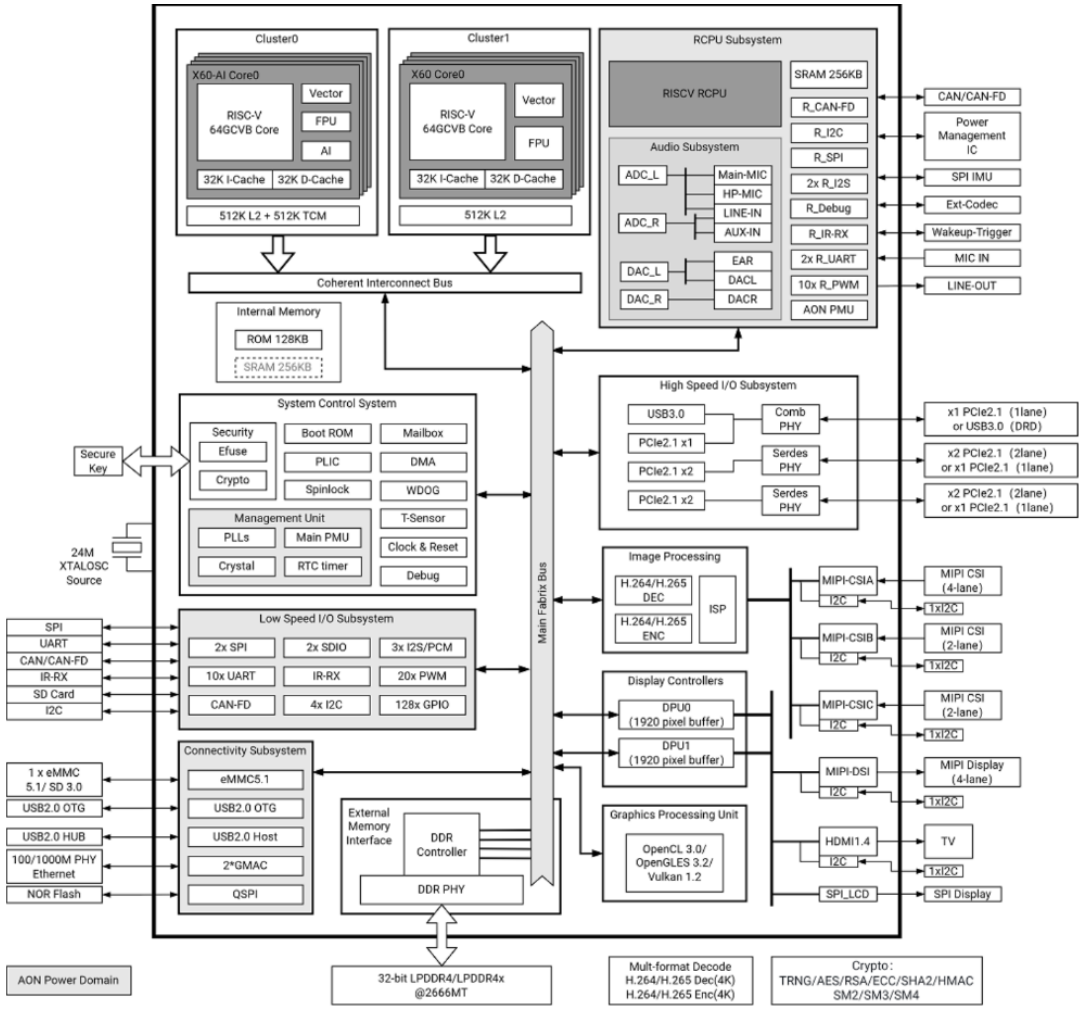

Fig. 0.5 SpacemiT K1/X1 SoC block architecture with eight X60 cores in two clusters.

Compliance with RISC-V **64GCVB** standard ISA extensions and **RVA22** profile

- Each core has 32KB L1-I cache and 32KB L1-D cache
- Each cluster contains 512KB L2 cache
- Cluster 0 integrates 512KB TCM (Tight-Coupled Memory) for AI extension
- L1 cache supports MESI consistency protocol, instead L2 cache supports MOESI consistency protocol
- Vector extension: RVV1.0 with VLEN 256/128-bit and x2 execution width
- AI customized instructions explored and implemented in Cluster 0
- Support for CLINT and PLIC with a total of 256 interrupts
- Support for RISC-V performance PMU
- Support for SV39 virtual memory
- Support for 32 PMP entries adhering to RISC-V security framework
- Support for RISC-V debug framework
- Support for the following extensions:
- RV64I M A F D C V and many Binary extension plus AI customized instructions

0.5.2 X60 memory and registers

Each processing core (**X60**) communicates with local cache memory. The processing units operate on 3 sets of registers : general purpose registers (`x0-x31`), floating point registers (`f0-f32`) and the blcok of vvector registers (`v0-v31`).

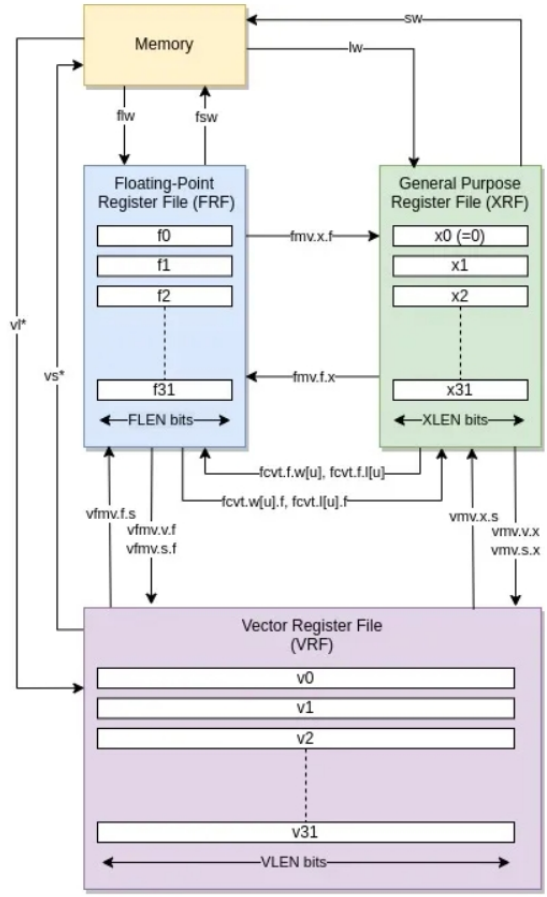

Fig. 0.6 The use of general-purpose registers (**XRF**) and floating-point registers (**FRF**) is straightforward. However, before using the vector registers (**VRF**), it is necessary to configure the data format, including the element size and type. This configuration specifies how the data will be processed by subsequent vector instructions.

0.5.3 Spacemit Muse Pi Pro

Muse Pi Pro shown in the following picture is a feature-packed, credit card-sized SBC powered by the SpacemIT M1 octa-core 64-bit RISC-V AI SoC with a **2 TOPS NPU** and equipped with up to 16GB LPDDR4x and 128GB eMMC flash.
The single board computer features gigabit Ethernet and a WiFi 6 + Bluetooth 5.3 module for connectivity, HDMI and MIPI DSI display interfaces, two MIPI CSI interfaces, a 3.5mm audio jack, four USB 3.0 ports, an M.2 socket for an NVMe SSD or wireless module, a mini PCIE socket for WiFi and 4G LTE/5G cellular connector, and a 40-pin GPIO header for expansion. That's quite a lot of features for such a compact board.

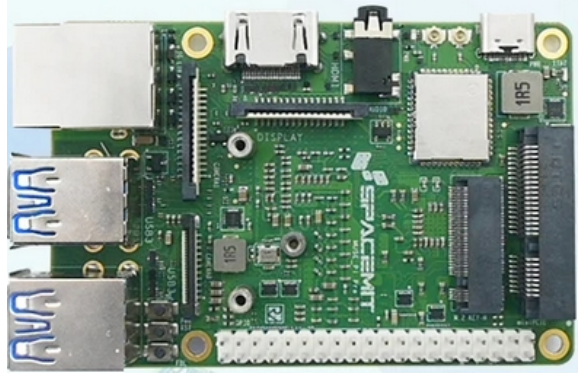

Fig 0.7 MUSE Pi Pro board from **SpacemiT** integrating **X60 SoC**

0.5.4 Other boards with SpacemiT X60 SoC

For our laboratory exercises and examples, we used the MUSE Pi Pro development board. However, the same experiments can also be carried out on other compatible hardware platforms, including:

- **Banana Pi BPI-F3**
- **Orange Pi RV2**

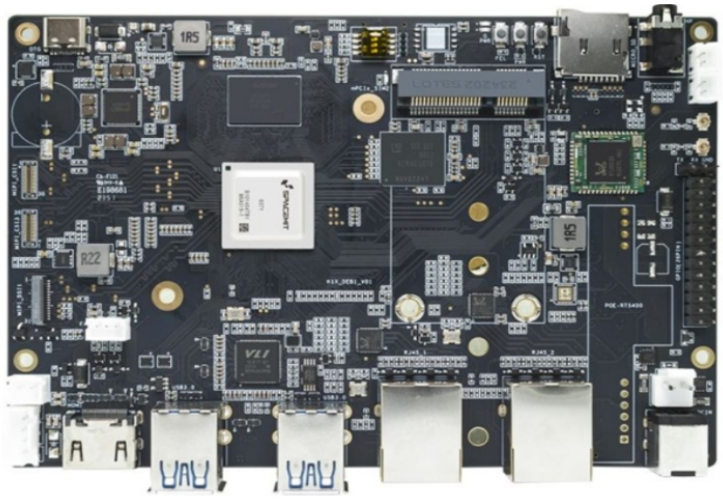

Fig 0.8 Banana Pi F3 board from integrating X60 SoC

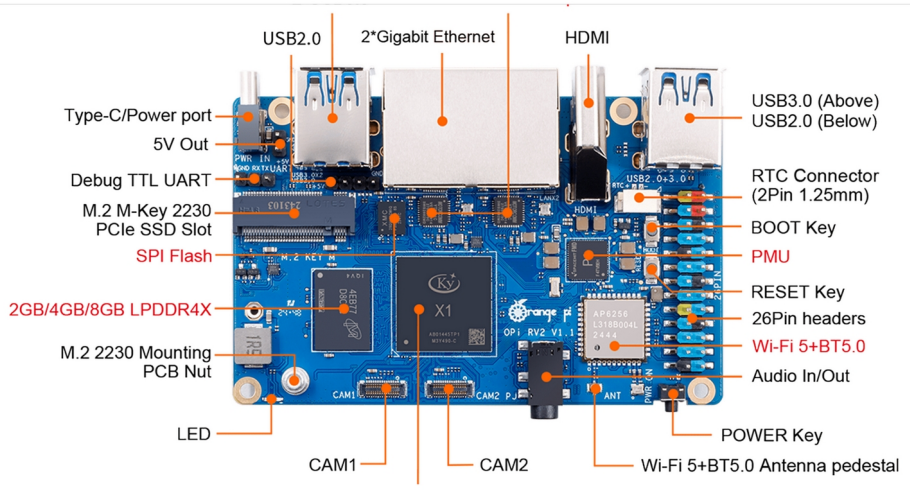

Fig 0.9 Orange PI RV2 with K1 – SoC

Note: The Orange Pi RV2 is available at entry-level pricing of approximately $30–40 (or €30–40), depending on the RAM configuration.

0.6 Experimenting with Parallel Programming (SIMD and MIMD)

The programming languages and tools for parallel programming on RISC-V are based on C/C++ and assembly.
Vector programming (SIMD) requires access to vector instructions, which can be achieved either through **assembly-level programming**, by using **C intrinsics** that directly map to the corresponding assembly instructions, or by forcing the modern `gcc` compiler (version 14) to exploit the vector instructions (**options** and **march** parameters) during C language program compilation.

Our choice is to use **both assembly and C programming with (and without) compiler options.** This requires a brief introduction to assembly programming before working with vector instructions.

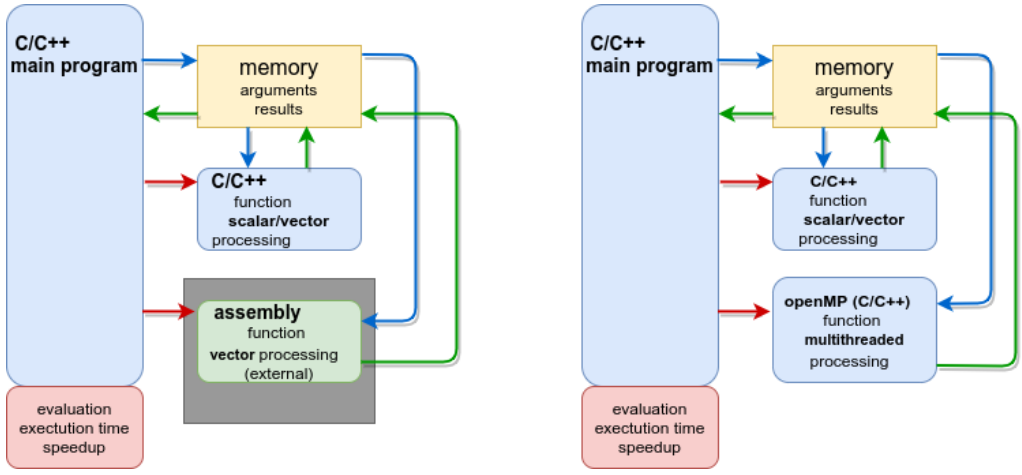

Fig 0.10 Using C/C++ for **scalar**/**vector** programming and
(a) **assembly** for **vector** programming (external function)
(b) **openMP** C/C++ for multithreaded programming

0.7 GCC Optimization Levels and RVV

0.7.1 Using compilation options:

- `-O2`
  - The “safe” optimization level.
  - Enables most optimizations that **don’t increase code size too much** or risk breaking strict standards compliance.
  - **Vectorization**:
    - Enables **basic auto-vectorization** (**loop vectorization using RVV when available**).
    - Focuses on **conservative heuristics**: loops must be simple, with clear bounds and no tricky dependencies.
    - May leave some scalar code unchanged if the compiler thinks vectorization won’t help.
  - Goal: balance **speed and binary size**.
- `-O3`
  - The “aggressive” optimization level.
  - Enables **all of `-O2` plus extra optimizations** that can **increase code size** or sometimes **reduce performance** on certain workloads.
  - **Vectorization**:
    - Enables **more aggressive auto-vectorization**, including loops with more complex dependencies or memory patterns.
    - Turns on **loop unrolling** and **vectorization of outer loops** where possible.
    - Tries to generate **wider RVV instructions** and may use **masked vector ops** for non-trivial loops.
- Goal: **maximize speed**, even if the binary grows and compile time is longer.

0.7.2 Example (RVV impact)

Consider a function with a simple loop – **`vec.c`** :

```
1 #include <stddef.h>
2
3 void addf(float * __restrict a,
4           const float * __restrict b,
5           const float * __restrict c,
6           size_t n) {
7   for (size_t i = 0; i < n; ++i)
8     a[i] = b[i] + c[i];
9 }
```

- `-O2`:
  - GCC will vectorize if it can **prove alignment and safety**.
  - Uses RVV intrinsics like **`vle32.v`**, **`vadd.vv`**, **`vse32.v`**.
- `-O3`:
  - GCC will try to vectorize even if safety is less obvious.
  - Might **unroll the loop**, issuing multiple RVV vector ops per iteration.
  - Could choose **wider vector length assumptions** (**VLEN**) and rely on masking for leftovers.
  - May inline small helper functions into vectorized code.

0.7.3 Practical Differences for RISC-V RVV

- `-O2` → reliable, portable, conservative RVV usage.
- `-O3` → pushes GCC to **emit more RVV instructions**, especially for complex loops, at the cost of larger code and sometimes worse cache performance.
- In practice, on current RISC-V GCC backends:
    - **`-O2`** already gives you RVV vectorization on "obvious" loops.
    - **`-O3`** gives you more **outer-loop and unrolled vectorization**, which can boost throughput on **large arrays** but may bloat binaries.

0.7.4 GCC switches

Below are the **exact GCC switches** you can use to see which loops got vectorized (and which didn't), plus a tiny example to try on your RV2.
These **print vectorizer diagnostics** during compile.

```
# Show only successful vectorizations on stderr
$ gcc -O2 -march=rv64gcv -fopt-info-vec vec.c -c -S
vec.c:7:24: optimized: loop vectorized using variable length vectors


# Show only missed opportunities (and why)
gcc -O2 -march=rv64gcv -fopt-info-vec-missed vec.c -c

# Show everything about vectorization (both optimized + missed)
gcc -O2 -march=rv64gcv -fopt-info-vec vec.c -c
```

Send them to files (cleaner):

```
gcc -O2 -march=rv64gcv -fopt-info-vec-optimized=vec.opt \
         -fopt-info-vec-missed=vec.missed \
         vec.c -c
```

0.8 Objectives and Methodology

Our **first objective** is to demonstrate **how to program** with mixed languages, combining C/C++ and assembly, in order to implement **simple parallelizable functions** such as computation, image processing, and image generation. To support this goal, we provide a series of examples organized as programming laboratory exercises.

The **second objective** is to **evaluate the execution time** (measured in seconds) of the selected examples and compare the results to calculate the corresponding speedup. For each example introduced, we measure the scalar execution time (a plain C function without optimization) and compare it with the vectorized execution time to determine the performance gain.

The vectorized versions are produced both by compiling C/C++ code with optimization options (`-O2` and `-O3`) and by writing direct assembly programs. This methodology is further extended to compare single-core and multi-core execution times, allowing us to analyze and quantify the speedup achieved through parallelism.

Finally, to reinforce understanding of the worked examples, each laboratory session concludes with a short set of follow-up exercises. (**To do:**)

0.9 Lab's organization

The book covers the following subjects:

- Basic assembly programming for RISC-V
- Introduction to vector programming and processing with RISC-V vector (**RVV**) instructions
- Vector programming for simple numerical calculations
- Vector programming for basic image processing (using **OpenCV** and **OpenGL**)
- Fundamentals of multi-core (multi-threaded) programming with **OpenMP** in C/C++
- Multi-threaded programming for simple image processing tasks
- Combined multi-threaded and vector programming

The above subjects are organized as a series of programming and processing laboratories:

- **Lab 1: Introduction to RISC-V Assembly Programming**
  Covers the basics of RISC-V assembly language, register usage, and simple arithmetic/logic operations.
- **Lab 2: Introduction to RISC-V Vector Assembly Programming I**
  Introduces vector registers and the fundamentals of SIMD operations on small datasets.
- **Lab 3: Introduction to RISC-V Vector Assembly Programming II**
  Extends vector programming concepts to more advanced operations, including loops and mixed data types.
- **Lab 4: Vector Programming for Image Processing I**
  Demonstrates the use of vector instructions for basic image manipulation tasks such as pixel transformations.
- **Lab 5: Vector Programming for Image Processing II**
  Explores more complex image processing operations, including filtering and convolution, using vectorized code.
- **Lab 6: Parallel Multi-Core Programming with OpenMP for Image Processing**
  Introduces OpenMP in C/C++ for parallel execution across multiple cores, applied to image-processing workloads.
- **Lab 7: Combined Vector and Multi-Core Programming for Image Processing**
  Integrates vectorization and multi-core parallelism to maximize performance in image-processing applications.
- **Lab 8: Vector and Multi-core processing for AI applications**

All programs presented in the book are available for download at our **smartcomputerlab github** repository:

https://github.com/smartcomputerlab/RISC-V-Parallel-SIMD-and-MIMD-Programming-and-Processing-Book/tree/main

Attention:

Before starting the assembly programming labs, we introduce the essential elements of assembly language.

This section may be skipped and revisited later during the programming and processing labs, allowing you to proceed directly to **Lab 1** if preferred

0.10 RISC-V assembly language

0.10.1 Introduction

Programming RISC-V with assembly language offers several key advantages, especially in scenarios where control, optimization, and hardware awareness are critical. Below are some of the primary benefits:

1. **Fine-Grained Control of Hardware**
    - **Direct Access to CPU Features**: RISC-V assembly allows developers to directly interact with processor instructions, registers, memory, and I/O devices. This level of control is essential for hardware-level tasks such as interrupt handling, device drivers, or manipulating specific hardware peripherals.
    - **Custom Instruction Set Extensions**: RISC-V allows for user-defined custom instructions, so writing in assembly helps exploit these extensions effectively when needed for specialized tasks.
2. **Performance Optimization**
    - **Manual Optimization**: Assembly language gives developers the ability to optimize their code for speed, size, or power efficiency by manually tuning instructions, avoiding unnecessary overhead, and making decisions about which operations are faster for a given processor.
    - **Instruction-Level Parallelism**: Developers can control how instructions are scheduled, potentially reducing instruction stalls, pipeline hazards, and maximizing the use of the CPU's pipelines.
    - **Efficient Use of Memory**: Assembly allows developers to minimize memory usage, a critical factor for embedded systems or resource-constrained environments like micro-controllers.

3 **Small Code Size**

- **Minimal Overhead**: Writing in assembly produces minimal overhead since high-level language constructs like loops, conditionals, and function calls are replaced with direct machine instructions. This is particularly useful in systems with limited memory (e.g., embedded systems).
- **Precise Control of Memory Layout**: In assembly, the programmer has direct control over how data and code are laid out in memory, allowing for optimized and compact memory usage.

4 **Embedded Systems and Real-Time Applications**

- **Low-Level Access**: Assembly language is often used in embedded and real-time systems where low-level control is essential, such as controlling specific peripherals, real-time performance tuning, and interrupt handling.
- **Deterministic Execution**: In real-time systems, knowing the exact execution time of instructions is important. Assembly provides a clear understanding of how long each instruction will take, ensuring real-time constraints are met.

Writing in assembly helps **developers gain a deep understanding of the underlying RISC-V architecture**, including how **memory** is accessed, how **instructions are executed**, and how **control flow** is managed.

By learning to write in assembly, programmers also develop insights into what compilers do behind the scenes, allowing for better high-level code optimization and debugging.

0.10.2 RISC-V: base assembly instruction set

The RISC-V base integer instruction set, commonly referred to as the **"I" (Integer) instruction set**, provides a small yet complete collection of instructions required for general-purpose computing. It is defined in both the **RV32I** (32-bit) and **RV64I** (64-bit) variants. This instruction set includes fundamental arithmetic, logical, control-flow, memory-access, and system instructions.
In our laboratories, we use a **K1** (**SpacemiT**) single-board computer (SBC) as the development platform. The K1 integrates two quad-core clusters, each based on X60 processors implementing the **RV64GV** architecture.
Below is an overview of the basic instructions in the RISC-V **"I" instruction set**, organized by purpose.
Note the typical RISC-style architectural separation into:

- **Arithmetic** and logical instructions
- Memory **load** and **store** instructions
- **Control** transfer instructions (jumps and branches)

1. Arithmetic Instructions

These instructions perform integer arithmetic operations.

- **`add rd, rs1, rs2`** — Add two registers (**`rd = rs1 + rs2`**).
- **`addi rd, rs1, imm`** — Add immediate (**`rd = rs1 + imm`**).
- **`sub rd, rs1, rs2`** — Subtract (**`rd = rs1 - rs2`**).
- **`lui rd, imm`** — Load upper immediate (**`rd = imm << 12`**).
- **`auipc rd, imm`** — Add upper immediate to PC (**`rd = PC + (imm << 12)`**).

2. Logical Instructions

These instructions perform **bitwise logical operations**.

- **`and rd, rs1, rs2`** — Bitwise AND (**`rd = rs1 & rs2`**).
- **`andi rd, rs1, imm`** — Bitwise AND with immediate (**`rd = rs1 & imm`**).
- **`or rd, rs1, rs2`** — Bitwise OR (**`rd = rs1 | rs2`**).
- **`ori rd, rs1, imm`** — Bitwise OR with immediate (**`rd = rs1 | imm`**).
- **`xor rd, rs1, rs2`** — Bitwise XOR (**`rd = rs1 ^ rs2`**).
- **`xori rd, rs1, imm`** — Bitwise XOR with immediate (**`rd = rs1 ^ imm`**).

3. Shift Instructions

These instructions perform **left or right shifts**.

- **`sll rd, rs1, rs2`** — Shift left logical (**`rd = rs1 << rs2`**).
- **`slli rd, rs1, imm`** — Shift left logical immediate (**`rd = rs1 << imm`**).
- **`srl rd, rs1, rs2`** — Shift right logical (**`rd = rs1 >> rs2`**).
- **`srli rd, rs1, imm`** — Shift right logical immediate (**`rd = rs1 >> imm`**).
- **`sra rd, rs1, rs2`** — Shift right arithmetic (**`rd = rs1 >> rs2`**).
- **`srai rd, rs1, imm`** — Shift right arithmetic immediate (**`rd = rs1 >> imm`**).

4. Comparison Instructions

These **instructions compare values** in registers and set the **destination register to 1** if the comparison is **`true`**, **otherwise set it to 0**.

- **`slt rd, rs1, rs2`** — Set if less than (**`rd = (rs1 < rs2))`**.
- **`slti rd, rs1, imm`** — Set if less than immediate (**`rd = (rs1 < imm)`**).
- **`sltu rd, rs1, rs2`** — Set if less than (unsigned) (**`rd = (rs1 < rs2) unsigned`**).
- **`sltiu rd, rs1, imm`** — Set if less than immediate (unsigned) (**`rd = (rs1 < imm)`** unsigned).
-

5. Memory Access Instructions

These instructions **load** data from **memory into registers** or **store** data from registers into memory.

- **`lw rd, imm(rs1)`** — Load word (**`rd = Mem[rs1 + imm]`**).
- **`lh rd, imm(rs1)`** — Load halfword.
- **`lb rd, imm(rs1)`** — Load byte.
- **`lbu rd, imm(rs1)`** — Load byte unsigned.
- **`lhu rd, imm(rs1)`** — Load halfword unsigned.
- **`sw rs2, imm(rs1)`** — Store word (**`Mem[rs1 + imm] = rs2`**).
- **`sh rs2, imm(rs1)`** — Store halfword.
- **`sb rs2, imm(rs1)`** — Store byte.
-

6. Control Transfer Instructions

These instructions control the flow of execution, including conditional branches and unconditional jumps.

- **`beq rs1, rs2, offset`** — Branch if equal.
- **`bne rs1, rs2, offset`** — Branch if not equal.
- **`blt rs1, rs2, offset`** — Branch if less than (signed).
- **`bge rs1, rs2, offset`** — Branch if greater than or equal (signed).
- **`bltu rs1, rs2, offset`** — Branch if less than (unsigned).
- **`bgeu rs1, rs2, offset`** — Branch if greater than or equal (unsigned).
- **`jal rd, offset`** — Jump and link (used for function calls).
- **`jalr rd, offset(rs1)`** — Jump and link register.
-

7. System Instructions

These instructions provide system-level control, including **traps and environment calls** (for example, for operating system services).

- **`ecall`** — Environment call (used to invoke system services, e.g., syscalls).
- **`ebreak`** — Environment break (used for debugging or breakpoints).
-

8. No-Operation Instruction

This instruction does nothing and is often used for padding.

- **`nop`** — No operation (**`addi x0, x0, 0`** is commonly used as **`nop`**).

0.10.3 Example program: Sum of two numbers

Here is a simple RISC-V assembly program that **adds two numbers and stores the result in a register**.

```
    .text
    .globl _start

_start:
    # Load two numbers into registers
    li a0, 10          # Load immediate value 10 into register a0
    li a1, 20          # Load immediate value 20 into register a1
    # Perform addition
    add a2, a0, a1     # a2 = a0 + a1 (10 + 20 = 30)
    # Exit the program using ecall
    li a7, 93          # Syscall number for exit
    ecall              # Make system call
```

Assembly (`as`) and load (`ld`) of the above program:

```
$as add_simple.s -o add_simple.o
$ld add_simple.o -o add_simple
$./add_simple
$
```

Register Name	ABI Name	Description
x0	zero	Hard-Wired Zero
x1	ra	Retun Address
x2	sp	Stack Pointer
x3	gp	Global Pointer
x4	tp	Thread Pointer
x6-x7	t1-t2	Temporary Registers
x8	s0/fp	Saved Register / Frame Pointer
x9	s1	Saved Register
x10-x11	a0-a1	Function Argument / Return Value Registers
x12-x17	a2-a7	Function Argument Registers
x18-x27	s2-s11	Saved Registers
x28-x31	t3-t6	Temporary Registers

Fig 0.8 RISC-V 32/64 register file: `t1,t2,..,t3–t6` and `a0–a7` - user data registers; `a0–a7` – function argument registers, `a0,a1` - return value registers

0.10.4 RISC-V (V) assembly vector instruction set

Below is a compact, simplified list of commonly used **RISC-V Vector Extension (RVV) instructions** enough to follow most basic examples like the vector add program.

1. Setup & Configuration

- `vsetvli rd, rs1, eX, mY, ta|tu, ma|mu` – Set **VL** (**Vector Length**) based on `rs1` elements, element width (`e8/e16/e32/e64`), **LMUL** (`m1/m2/m4/m8`), and **tail**/**mask** policy.
- `vsetvl rd, rs1, rs2` – Set **VL** based on registers (dynamic settings).

2. Vector Load / **Store**

- `vleX.v vd, (rs1)` – Load **X-bit elements** from memory into vector register `vd`.
- `vseX.v vd, (rs1)` – Store **X-bit elements** from `vd` to memory.
  (`X` can be `8,16,32,64`)

3. Integer Arithmetic

- `vadd.vv vd, vs2, vs1` – Add vectors.
- `vadd.vx vd, vs2, rs1` – Add vector and scalar.
- `vsub.vv / vsub.vx` – Subtract.
- `vmul.vv / vmul.vx` – Multiply.
- `vdiv.vv / vdiv.vx` – Divide (signed).
- `vrem.vv / vrem.vx` – Remainder (signed).

4. Logical / **Bitwise**

- `vand.vv / vand.vx` – AND.
- `vor.vv / vor.vx` – OR.
- `vxor.vv / vxor.vx` – XOR.

5. Comparison (Sets mask register)

- `vmseq.vv / vmseq.vx` – Equal.
- `vmsne.vv / vmsne.vx` – Not equal.
- `vmslt.vv / vmslt.vx` – Less than (signed).
- `vmsle.vv / vmsle.vx` – Less or equal (signed).

6. Mask Operations

- `vmerge.vvm vd, vs2, vs1, v0` – Merge based on mask `v0`.
- `vmv.v.i / vmv.v.x / vmv.v.v` – Move immediate, scalar, or vector to vector.

7.Floating-Point (if F/D extension present)

- `vfadd.vv / vfadd.vf` – FP add.
- `vfsub.vv / vfsub.vf` – FP subtract.
- `vfmul.vv / vfmul.vf` – FP multiply.
- `vfdiv.vv / vfdiv.vf` – FP divide.

Other Useful Ops

- `vmv.x.s rd, vs1` – Move first element of vector to scalar register.
- `vmv.s.x vd, rs1` – Move scalar to first element of vector.
- `vslideup.vx / vslidedown.vx` – Shift elements within a vector.

0.10.4.1 Initial vector instruction - `vsetvli`

`vsetvli` sets **VL** (**vector length**) and **VTYPE** (**vector element configuration**) before executing vector instructions.

It tells the hardware:

- **How many vector elements** will be processed in this loop iteration.
- **What size and grouping** of elements to use.
- How to handle leftover elements ("**tail**") and **masked** lanes.

General format

```
vsetvli rd, rs1, e<E>, m<LMUL>, <tail-policy>, <mask-policy>
```

`rd` – destination register to hold the actual VL chosen by hardware.
`rs1` – requested number of elements (remaining elements to process).
`e<E>` – element width in bits: e8, e16, e32, e64 (and larger if supported).
`m<LMUL>` – LMUL (vector register group multiplier): m1, m2, m4, m8, etc.

Tail policy – what to do with unused lanes at the end of a vector:
`ta` (tail **agnostic**) → hardware can leave unused lanes undefined.
`tu` (tail **undisturbed**) → unused lanes keep their old values.
Mask policy – what to do with lanes where the mask bit is 0:
`ma` (mask **agnostic**) → inactive lanes can be undefined.
`mu` (mask **undisturbed**) → inactive lanes keep their old values.

Below we give a compact reference table for common **`vsetvli`** configurations. It shows how element size (SEW), group multiplier (LMUL), and tail/mask policies (TA/TU, MA/MU) are typically combined:

	Instruction	SEW (Element Size)	LMUL (Group Multiplier)	Tail Policy	Mask Policy	Notes
1	`vsetvli x10, x11, e32,m1,tu,mu`	32-bit	1	TU (tail undisturbed)	MU (mask undisturbed)	Standard 32-bit vector ops, safe for partial vectors
2	`vsetvli x5, x6, e16,m2,ta,ma`	16-bit	2	TA (tail agnostic)	MA (mask agnostic)	Faster, but tail/masked values may be overwritten
3	`vsetvli x7, x8, e64,m4,tu,ma`	64-bit	4	TU	MA	High throughput with large vectors, tails preserved
4	`vsetvli x12, x0, e8,mf2,tu,mu`	8-bit	1/2 (mf2)	TU	MU	Efficient for byte-level operations, tails/masks preserved
5	`vsetvli x3, x0, e32,m8,ta,ma`	32-bit	8	TA	MA	Very wide vector grouping, maximum parallelism but not tail-safe

0.10.4.2 How to read and understand RVV assembly

1. **Identify the ABI / arguments**

- Scalar regs: **`a0–a7`** carry **args**/**returns**; **`t0–t6`** temps; **`s0–s11`** callee-**saved**.
- Typical pointer args: **`a0=srcA`**, **`a1=srcB`**, **`a2=dst`**, **`a3=count/bytes`**.

2. **Find the outer loop structure**

- Look for labels like **`.Lloop`**, **`.Ldone`**.
- Counters are often in **`a3`** or **`t?`**; decremented by **`VL`** each iteration.

3. **Decode `vsetvli / vsetvl`**

- **`vsetvli rd, rs1, e32, m1, ta, ma`**
  - **`SEW`** (**`e8/e16/e32/e64`**) = element width.
  - **`LMUL`** (**`m1/m2/…`**) = register grouping (vector register usage).
  - **`rd`** gets **VL** (lanes processed this pass).
  - **`ta/ma`** **tail**/**mask** policy: **`ta,ma`** = don't preserve inactive lanes.
- **RS1** is usually "remaining elements".

4. **Understand memory access**

- **Unit-stride** loads/stores: **`vleX.v / vseX.v`**.
- **Strided** loads: **`vlseX.v vd,(rs1),rs2`** (**stride in bytes** in *register*).
- **Segmented** loads/stores (if used): **`vlsegN*`** (de-interleave), but many tools don't support every variant.
- Scalar pointer bumps: **`add a0,a0, t0<<log2(bytes/elt)`**.

5. **Core vector ops**

- Integer: **`vadd/vsub/vmul/vdiv`**, logical **`vand/vor/vxor`**.
- Widen/narrow: **`vzext/vsext.vf2`** (widen x2), **`vnsrl.wi`** etc (narrow—often missing on older envs).
- Reductions: **`vredsum.vs, vredmax.vs`**, etc. Seed is in the second source vector.

6. **Reductions & widening**

- Sum of 8/16/32-bit products often widens to avoid overflow:
  - widen inputs (e.g., e16→e32) then **`vmul`** at e32, or
  - if **`vwmul.*`** exists, use it, else emulate via widen+`vmul`.
- After reduction, move scalar out: **`vmv.x.s rd, v?`**.

7. **Tail handling**

- Length-agnostic loops rely on `VL` possibly < vector width at the last iteration; no scalar epilogue needed.

8. **Common pitfalls**

- **Illegal operands**: strided load stride must be in a register, not an immediate (e.g., **`vlse8.v vd,(rs1), a3, not ..., 3`**).
- **Unsupported instructions**: narrowing ops (**`vnclip*, vnsrl*`**) or widening multiplies (**`vwmul.*`**) may be missing in reduced RVV subsets—replace with widen+shift+strided-pack patterns.
- **BGR vs RGB order** mistakes.
- **Signed vs unsigned** widening (**`vsext`** vs **`vzext`**).

0.10.4.3 Mini example + explanation

Code

```
# c[i] = a[i] + b[i] for n elements (int32)
# a0=a, a1=b, a2=c, a3=n
.Lloop:
  vsetvli t0, a3, e32, m1, ta, ma   # choose VL for remaining int32s
  vle32.v  v0, (a0)                 # load A chunk
  vle32.v  v1, (a1)                 # load B chunk
  vadd.vv  v2, v0, v1               # v2 = v0 + v1
  vse32.v  v2, (a2)                 # store C chunk
  slli     t1, t0, 2                # bytes = VL*4
  add      a0, a0, t1               # bump pointers
  add      a1, a1, t1
  add      a2, a2, t1
  sub      a3, a3, t0               # remaining -= VL
  bnez     a3, .Lloop
  ret
```

What each part means

- **`vsetvli`**: configures SEW=32, LMUL=1, sets VL (how many elems this pass).
- **`vle32.v`**: loads VL 32-bit ints from the current pointer.
- **`vadd.vv`**: lane-wise add.
- **`vse32.v`**: stores VL results.
- Pointer bump uses VL * **`sizeof(int32)`**; loop repeats until **`a3==0`**.
