Comet Core

Logan Fortune, Inria

Abstract

This paper should be used for personal use only. Comet is an open source general purpose central processing unit (CPU) core developed at Inria-Rennes [1]. The architecture is based upon RISCV RV32I base ISA. It is written in $C++$ for High Level Synthesis (HLS). This paper aims to describe with much accuracy this CPU and to present some micro-architectural choices for the multi-core version.

I. Introduction

Comet is a general purpose processor that is based upon the Harvard architecture as you can see in the figure 1. It means that the instruction memory and the data one communicate with the core on different buses. Thus, there are two memory interfaces implemented inside the design. The global stall is a signal that forces the core to freeze entirely. It is an external interrupt signal.

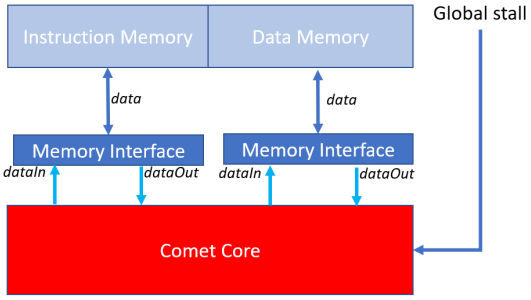

Figure 1: Comet Macroscopic Overview

The architecture core is a five-stage pipeline as you can see in figure 2: *fetch, decode, execute, memory* and *writeback*. This architecture has a forward and a branch unit making the CPU to run faster but also to stay consistent during execution.

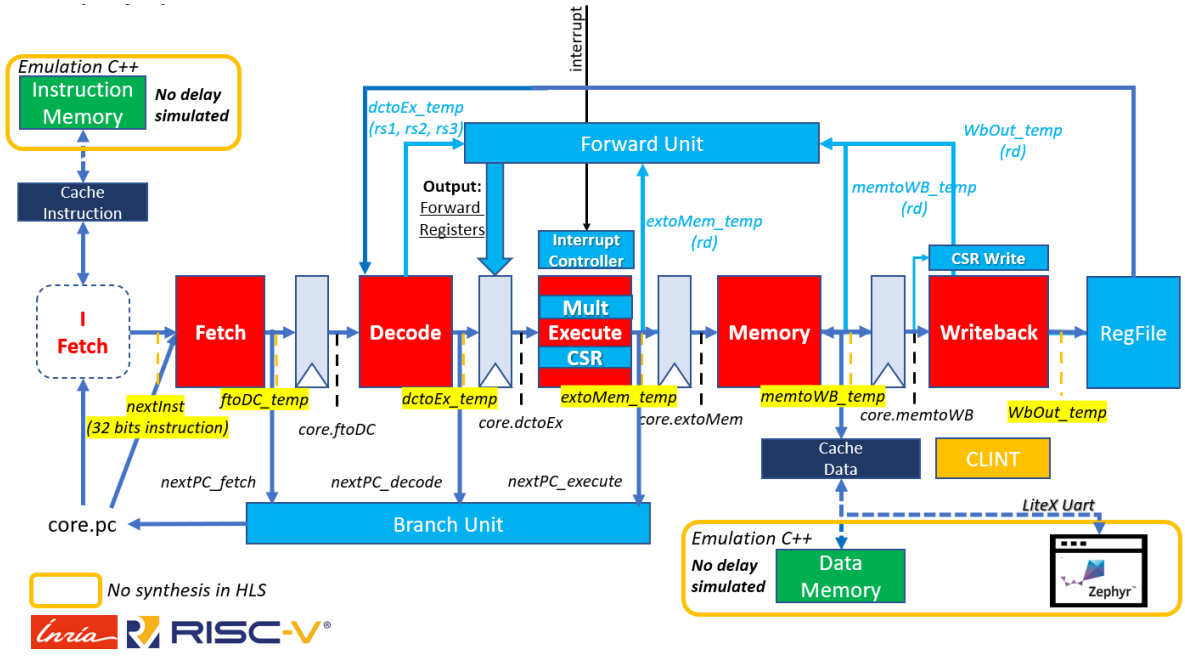

Figure 2: Core Architecture Overview

This CPU gathers standard design codes and does not have fancy state-of-the-art mechanisms because it mainly targets embedded applications. However, it is still a CPU that is competing in a good position against other RISCV CPUs (like Rocket and PicoRV) in its application range. The code is available at https://gitlab.inria.fr/srokicki/Comet/-/tree/zephyrOS and will be explained in the next sections.

II. Pipeline Stages

I. Fetch

The fetch stage is designed to monitor the *Program Counter* and the instructions dynamic during execution of the program. It takes the instruction that has been fetched to the instruction memory and send it to the decode stage at the next strike of clock. The main goal is to compute the next program instruction to fetch (see *nextPC_fetch* in the figure 2) which is the Program Counter plus four because four bytes is equal to 32 bits and instructions are word-aligned.

II. Decode

The decode stage goal is to prepare the instruction to the execution stage. It is strongly based on the RISCV ISA described in the figure 3 (not including CSR instructions). The decode stage decomposes the instruction into an *opcode* with the type of the operation (*funct7* and *funct3*), *source registers*, a *destination register* and finally an *immediate* value (unless it is a R-type instruction).

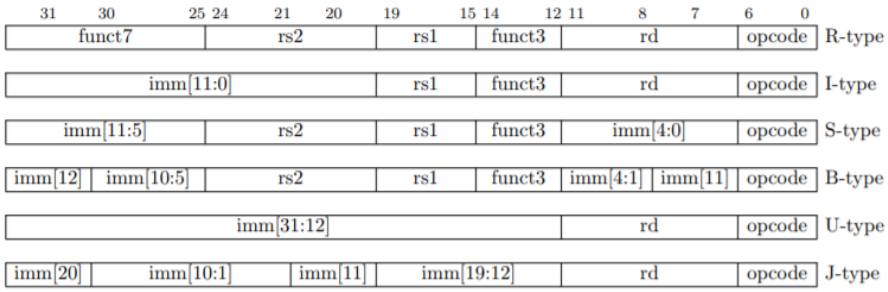

Figure 3: RISCV ISA Overview

In the decode stage, all possible immediate values (I, S, B, U, J types) are generated. And the correct values are correctly stored inside the *left hand side* (lhs), *right hand side* (rhs) and *datac* registers (see figure 4). It is also there that operand values are accessed in the registers file and store in rhs and lhs registers (see figure 4). These registers are used by the execution stage right after the forwarding mechanism processing (combinatorial part - multiplexer) to finally compute the operation requested in the execution stage.

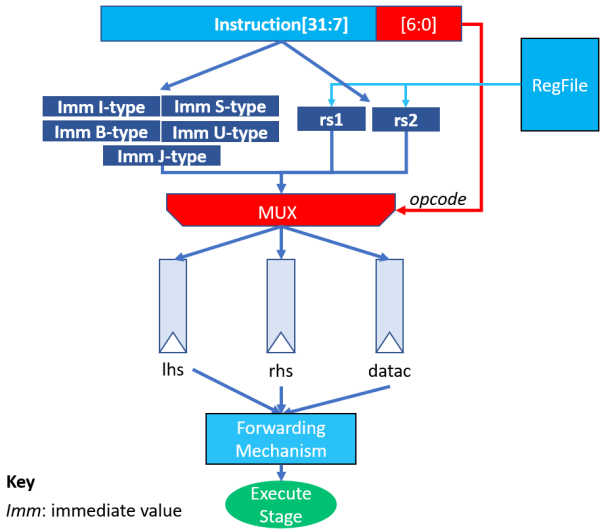

Figure 4: Comet Decode Architecture Overview

There is a special case in the decode stage: the JAL opcode. The JAL that stands for *Jump And Link* is an instruction that forces the CPU program counter to jump to the address computed (*nextPC_ decode* in figure 2) and to store the return address. In this case, the next Program Counter is affected and thus transmitted inside the branch unit with a control signal (*isBranch*) in order to change the program counter of the CPU (*core.pc*) accordingly.

III. Execute

The execution stage computes the operation requested by using lhs, rhs and datac registers according to the combination of the *opcode*, *funct3* and *funct7*. The result of the operation (extoMem.result) or the data (extoMem.datac) to be stored is sent to the memory stage.

IV. Memory

The memory stage prepares the result to be loaded or stored (RISCV_LD, RISCV_ST) inside the data memory thanks to the memory interface. After being processed by the memory stage, the destination register, the value to write and some control signals are sent to the Write Back stage.

V. Write Back

The write back stage aims to store the result inside the registers file that feeds the decode stage.

VI. Branch Unit

The branch unit aims to deliver a coherent branch execution by taking the different PCs computed inside the Fetch, the Decode or the Execution stage that have been triggered by a branch or a jump instruction. The different modifications of the core execution is represented in the table below. When a branch is taken, it is mandatory to flush some parts of the pipeline in order to keep the execution consistent.

Opcode	Branch Resolved in...	Control Signal	Flush Strategy
JAL	Decode	isBranch_decode	Fetch
BR, JALR	Execute	isBranch_execute	Fetch and Decode

VII. Forward Unit

The forward unit is a way to get faster execution by moving the execution stage result to the decode operands outputs (only if necessary) instead of waiting the result to be stored in the registers file. In fact, if the execute, the memory or the write back stage will write into a destination register (*executeUseRd*, *memoryUseRd*, *writebackUseRd*) the instruction that will be executed can already use the result to avoid Read After Write (RAW) issue. The forwarding mechanism is just after the decoder stage (see figure 4) and modifies the value of the lhs, rhs and datac registers as you can see in figure 5.

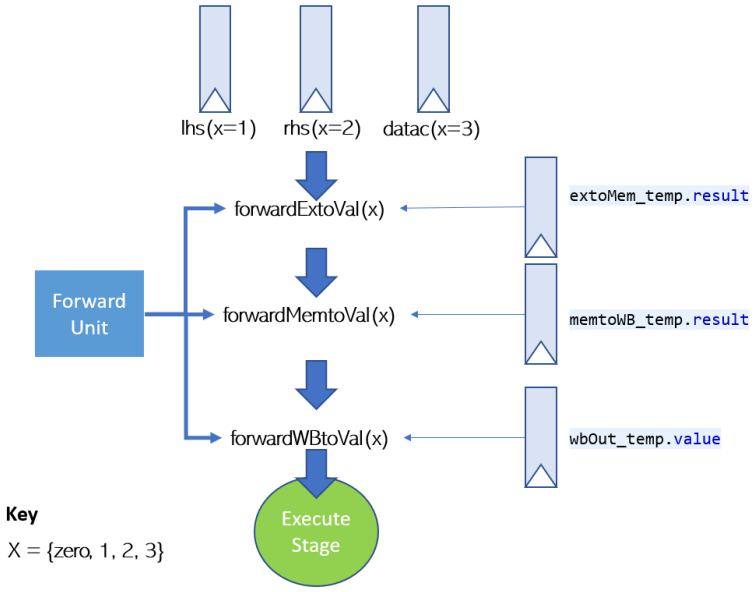

Figure 5: Comet Forward Unit Overview

VIII. Pipeline Stalls

VIII.1 Stall Verification Strategy Methodology

This section aims to elaborate a methodology to check that we get the best strategy for the different signals that stop execution of the core or a sub-section of it. The methodology is the following: *define all rules* for all different kind of control signals (global stall, invalidation, etc...) and *check that all rules* are accurately applied when a control signal is turned on inside the design. I have developed some global rules in order to optimize as much as possible the stalls used inside the Comet core. It is a personal verification list in order to check that we tend to an optimal strategy.

Definitions:

- A process inside the core is considered as a sub-system of the core. It can considered as a black-box. It is fully combinatorial or sequential. The granularity of the process precision is chosen as the one provided in the "core.cpp".
- A resource is a memory unit that can be used by the processes.
- A stall signal is a signal that is used to stop a subset of the core for an undefined amount of time.
- An invalidation signal ("*.we" in the cpp code) is a signal that informs the CPU that the targeted instruction should not be considered/processed.
- The *isLongInstruction* or *executeIsLongComputation* signals are used to stall the CPU for one cycle to anticipate that a load instruction cannot have its result after the execution stage but after the memory one. This bubble inside the pipeline is mandatory to solve all dependencies issues.

Rules:

- A process can create a stall signal. A stall must not block the process that releases it.
- The global stall stops all processes of the design.
- A stall must block the propagation of future instructions but not anterior instructions unless they consume the same resource than the process generator of the (un)stall signal.
- An invalidation must only invalidate the future instructions that are already in the pipeline. The invalidation signals must be kept along the wrong instructions during their whole propagation inside the pipeline.

The invalidation signal can be used to flush the pipeline in case of an interruption or a taken branch. The stall signal are mainly used to stop a subset of the core during long execution (multiplication, memory communication, flushing procedures).

VIII.2 Comet Control Signals

The core is mainly managed by control signals that ensure no hazards happen due to incorrect timing. That is why the use of *stall signals* are everywhere inside the core. The table below is a synthesis of the influence of all stall signals.

Core Process	stallIm*	stallDm	stallFetch**	stallDecode	stallExecute	stallMemory	stallMultAlu
IM Process							
DM Process	x					x	
Branch unit	x	x	x				x
Forward Unit							
Commit Fetch	x	x	x				x
Commit Decode	x	x		x			x
Commit Execute	x	x			x		x
Commit Memory	x	x				x	
Regfile Write Back	x	x					

* stallIm is a stall for instruction memory interface and stallDm is for the data memory interface.
** stallFetch, stallDecode, etc... is coded in the Comet HLS project as "core.stallSignals[STALL_FETCH]" and "core.stallSignals[STALL_DECODE]" respectively for instance.
*** A stall signal named *localstall* freezes entirely the pipeline. It does not appear in the table above.

The forward unit will not impact the core because the invalidation will make the signal "*.useRd" equal to zero which means that no forwarding is possible.

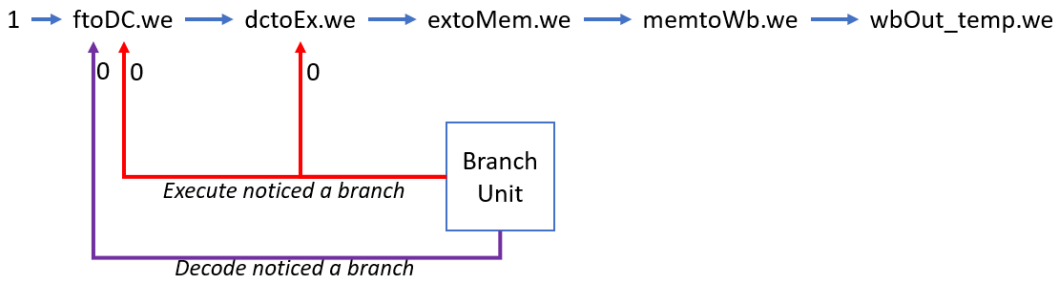

Figure 6: Comet Invalidation Signals

III. Memory and Cache

I. Memory Interface

The memory interface is coded in HLS as a table of 2^{24} integers. The address is in 32 bits while there is only 2^{24} data. That is why a mask is necessary (0xffffff) to convert the 32 bits address into a 24 bits address. It is interesting to notice that the two least significant bits are avoided because the instructions are word aligned.

II. Cache Technology

The Comet cache is a 4-way associative cache. It handles byte (unsigned or not), half (unsigned or not), word or long (maximum limited by the interface size) data size requests (see figure 8). The working principle of the cache can be explained with the figures below (figure 7 and 8):

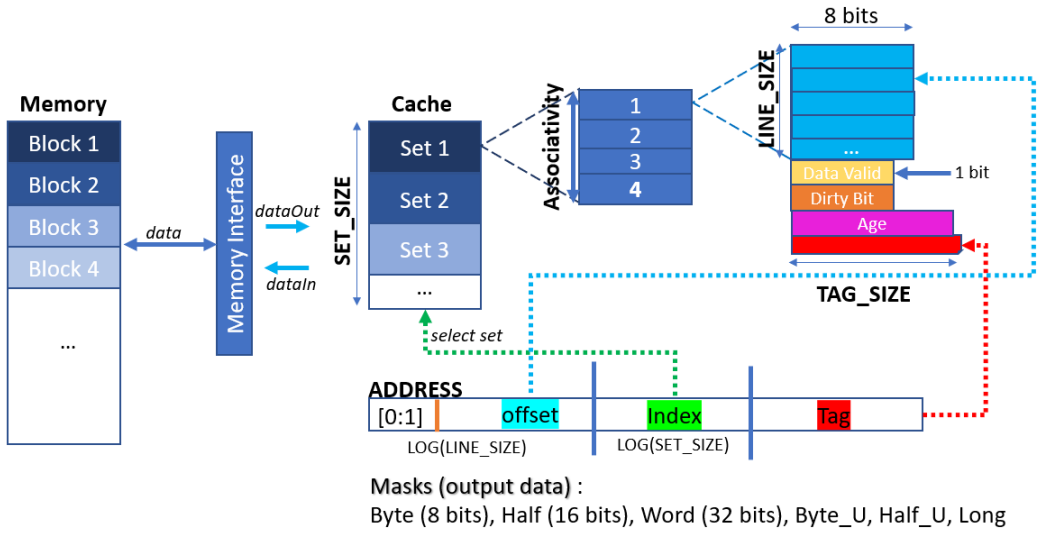

Figure 7: Comet Cache Architecture Overview

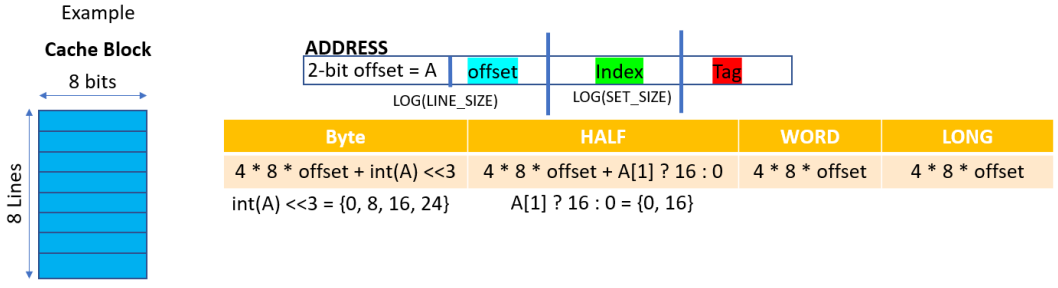

Figure 8: Comet Cache Possible Access

The cache is the closest level of memory inside the CPU. It is a memory sub-system that is less larger than the external memory. The cache is divided in sets (*set*-associative). And a set is decomposed into four blocks (*four-way* set associative) with each block decomposed also in lines which are byte memory units. Alongside a block of the set, there is a tag which can be compared with the tag address in order to resolve if there is a hit or a miss. There are also registers that store the *age* of the

block and a bit that says if the block is a valid memory block. The *age* register is used to update the cache by removing the *least recently used* blocks of the cache. The state machine of the cache can be explained thanks to the figures below (figure 9) . The interesting part of the cache state machine is when there is a miss. When a miss occurs, the most aged cache block of the set must be removed and new values should be inserted instead according to the address requested. The dirty bit of this block is checked because if the dirty bit is equal to zero, it means that the cache block has not been modified since it has been loaded from the main memory. In that case, there is no need to write back the data in the main memory. At contrary, if the dirty bit is equal to one, we need to correct the value in main memory. This mechanism is the origin of the *penalty miss*. While this mechanism could be seen as a burden for the memory interface, it can be proven that caching is efficient thanks to the locality principle which states that *things that will happen soon are likely to be closed to things that just happened*. As you can see in figure 9, this cache has a *write back* policy. It means that the store instruction writes effectively only in a cache block without updating the main memory (incoherence may occur) until a write back happens from the cache to the main memory.

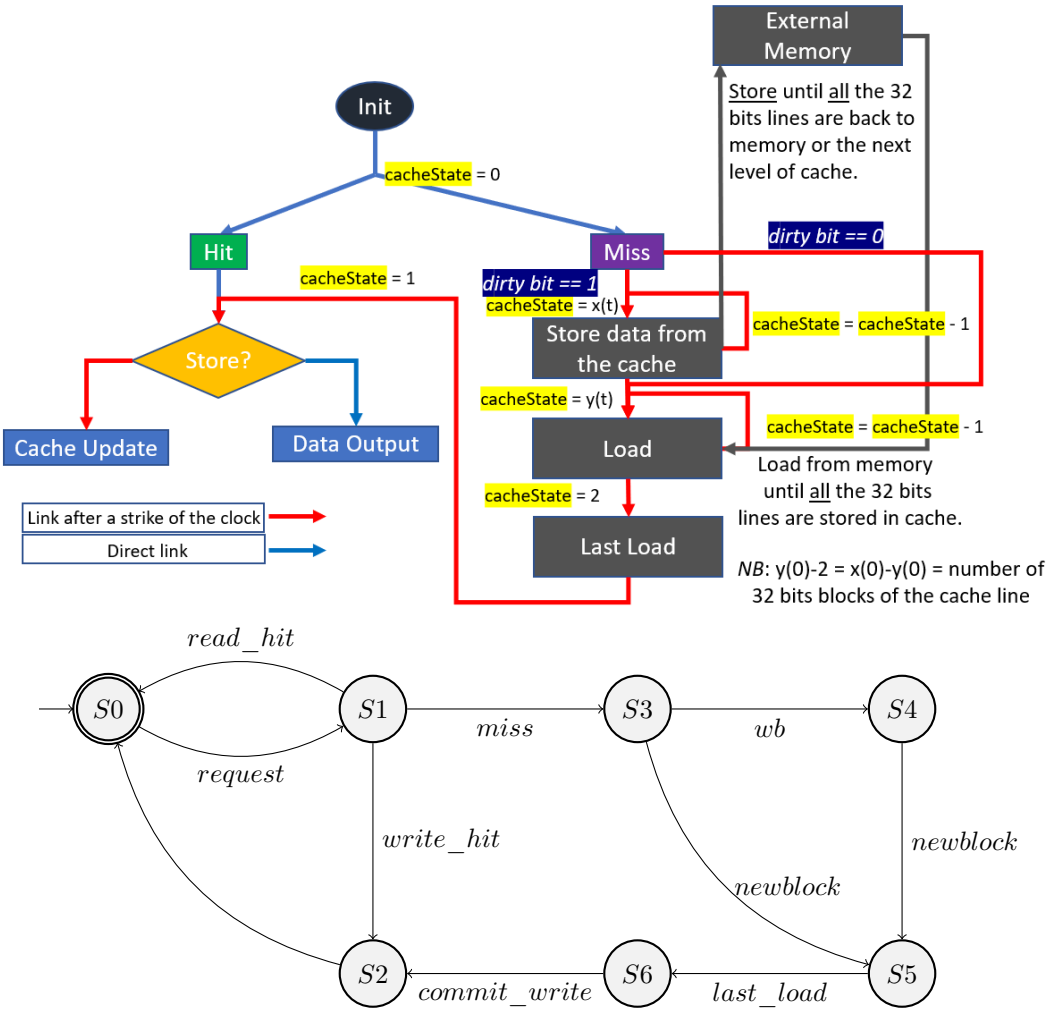

Figure 9: Cache Macroscopic Finite State Machine (*wb*: write-back, *newblock*: new cache block)

The Comet cache can be in different states :

- S0: *Neutral* State: the cache is waiting for new instructions.
- S1: Cache *request*: the cache is examining to know whether it is a hit or a miss.
- S2: Cache *write*: the cache writes into one of its sub-block (see figure 7).
- S3: The cache enters in the *miss* sub-states: the cache chooses the right cache block to evict. Then it sends the first memory request to elaborate a *the write-back* (Store) or to acquire the first chunk of the new cache block (Load).
- S4: In the this state the cache is doing write-backs (see figure 9).
- S5: In the this state the cache acquires cache blocks (see figure 9).
- S6: The cache leaves the *miss* sub-states, the cache is ready to write the new cache block. In this state, we do not yet write to the cache and we do not send a memory request. We just acquires the last memory chunk.

IV. Operating System Specific Architecture

An operating system (OS) is a software that is used to ease the management between user applications and hardware. Especially, the operating system schedules tasks and guarantees safe execution of the programs. We decided that **ZephyrOS** (*Linux Foundation*) will be the first OS to be booted with Comet. ZephyrOS is an operating system for embedded systems. This OS is close to Linux which can allow a smooth transition toward it. ZephyrOS is open-source with long-term support (LTS) with security updates.

I. Control State Registers (CSR)

The control state registers is used to gather statistics about the working core (like the number of instructions retired *minstret*, the counters *mhpmcounter3* or simply the number of cycles *mcycle*). It is also where the current core status is stored (inside *mstatus*). There are finally specific CSRs that are central in the making of exceptions and interruptions management.

II. Core-Local Interrupt Controller (CLINT)

The core-local interrupt controller is an element outside the core. We can access this element by making a memory call (memory-map). The goal of this block is to give a precise timer (*mtime*) independent of the execution of the program. It is also a way to prepare an interruption at a specific moment written inside the *mtimecmp* register.

III. Exceptions and Interruptions

Exceptions and interruptions are special triggers that change the normal execution of the program. An exception is raised by the program to elaborate a specific routine like a thread switch or a break (debug). The interruption is created outside the core (e.g CLINT). It changes the program execution the same way as the interruption but the handler of those calls will know if this is an interruption or an exception (see *mcause*). When returning from an exception or an interruption, we need to continue the program left thanks to a special register that store the previous program counter (*mepc*).

IV. Memory Map

In order to boot ZephyrOS with the core, the OS should have a memory map of all elements of the board described in the *device tree*. This memory map is the image of the environment allowing the core to communicate efficiently with all modules (see figure 10 - this figure is subject to change).

Base	Length	Attributes	Description
0x0000_0010	0x0000_0300		CLINT
0x0000_0310	0x0000_0480		(Early Console) Secure Enclave
0x0000_0790	0x0000_07A8		UART (LiteUART from LiteX)
0x0003_0000	ND	EX, NI, C	ROM + DRAM

Figure 10: Memory Map (EX: Executable, NI: Non-idempotent, C: Cached, ND: Not defined)

V. Uart - Console

The UART is a serial communication that aims to communicate information to external modules (included humans one). We used the UART to simulate a screen. There is also an early console that can be used to check the boot sequence. Indeed, the early console is a special zone in memory that aims to store specific data during boot sequence as you can see in figure 11. This part of the memory can also be used during execution for very specific purposes (security strategy, external data feed, etc...).

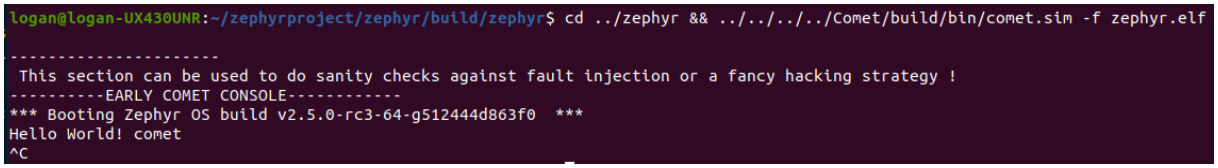

Figure 11: Zephyr Boot Sequence Demo

V. Testing Procedures And Benchmark

Comet core is coded in C++ which allows an easier way to describe the hardware. With the work done in Inria-Rennes [2], the HDL generated by the HLS tool should be equivalent to the C++ simulation. Thus, the designer should only be worried about the high level description of the core and all tests can be done with the C++ emulation. This workflow is a great opportunity for the CPU designer to evaluate quickly a micro-architecture but also to test it more easily by gathering the strengths of the C++ language and the community that comes with it. The workflow developed in Inria tries to be at best compliant with the RISC-V community last requirements.

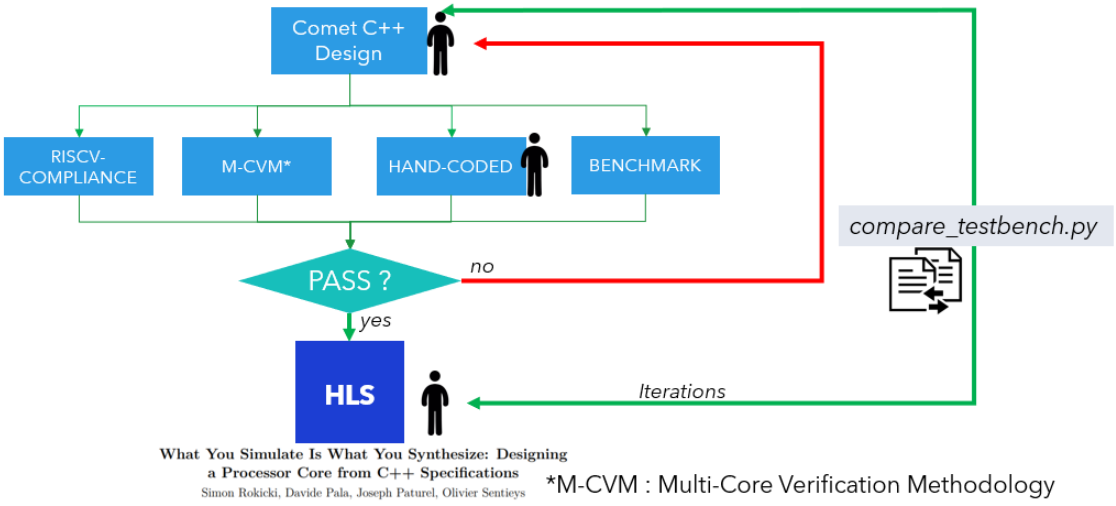

Figure 12: Comet Core Workflow

I. Compliance Suite

The compliance suite is a series of unitary tests that aims to evaluate if the hardware design is *RISC-V compliant* according to the ISA specification. RISC-V is an open-source standard ISA with lots of freedom given to the architects. That is why it is important to always ensure to pass the conformance testing preventing from fragmentation of the hardware developments (https://semiengineering.com/toward-risc-v-compliance/). These tests used all the instructions of the ISA. The tool feeds the core with specific instructions and checks each time the output signature.

II. Hand-Written Test

Hand-written tests are tests created at the beginning of the creation of comet. There are four tests that cover algorithms like djikstra, sorting methods, etc... These tests are more complex than the unitary ones. The reference model used to check the validity of the tests is the cpu of your computer.

III. Embench

In order to get an overview of the core performance, we added a benchmark named Embench. This benchmark targets embedded central processing units. It is a way to compare ourselves with other central processing units (https://antmicro.github.io/embench-tester/) but also to verify if any new implementation of Comet has not impacted the performance overall.

IV. M-CVM

M-CVM is a tool created especially to test comet with a multi-core configuration. This is a way to ensure that the multi-core system can work efficiently and without major deadlocks. Multi-core systems are an ensemble of central processing units. They can together execute unrelated programs or they share the same working data. When cores share their memories, there needs extra hardware to allow consistent memory communications and to force coherency between cores.The coherency protocol is hard to verify as the cardinal of the tests space skyrockets with the number of cores. Moreover, it is still difficult to find some bare-metal RISC-V tests for multi-core systems. Indeed, most of the time, the tests provided consider that the system can boot an operating system (see litmus tests).

VI. Core HLS

I. HLS in a nutshell

HLS stands for *High Level Synthesis*. The idea is to get a higher level of abstraction of hardware design. The C++ language is used to describe the architecture. And Catapult HLS tool chain transforms this description into a netlist format in vhdl (or verilog) that can be used inside your favorite FPGA or ASIC tool chains. Even if the level of abstraction is high, the hardware designer must still think about how the design elaborated in C++ could be synthesized in hardware. However, the burden of architecture optimization and simplification is entirely managed by the HLS tool.

II. Comet-Catapult workflow

This section aims to describe how the core is used by the Catapult HLS tool. The result of the HLS tool is a VHDL or Verilog netlist of the design. The HLS flow can be described as follow:

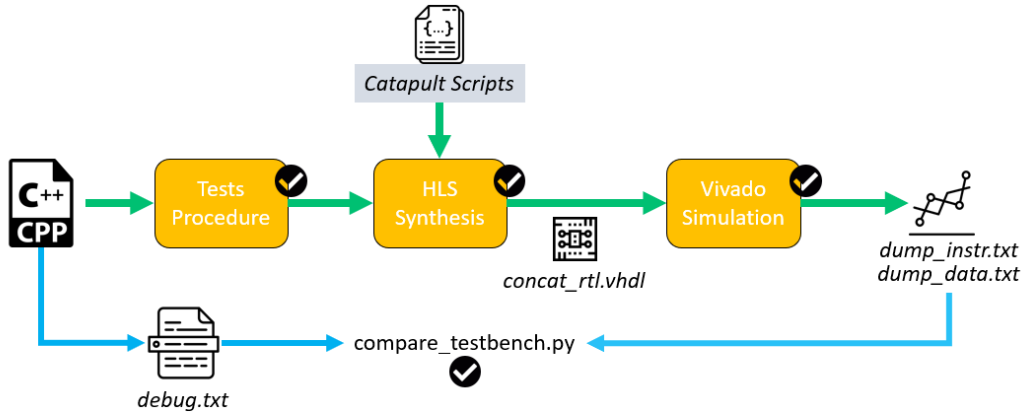

Figure 13: HLS Synthesis Flow

The HLS flow imposed by Catapult begins by giving the C++ design and useful scripts in order to synthesize correctly the design. The firsts step is the hierarchy organization description. During this step, the aim is to define what is the top design and the related hierarchical blocks. Just after that, we link the different hardware units with a production technology (ST Microelectronics, Intel, etc...). Then, one of the most important step is the architecture phase which defines memories types and memories organization. The memory type is the technology (RAM, ROM, etc...) while the memory organization is about data placement inside the memory. Finally, Catapult will elaborate the rtl design (vhdl/verilog output file - concat_rtl.vhdl) according to the previous constraints.

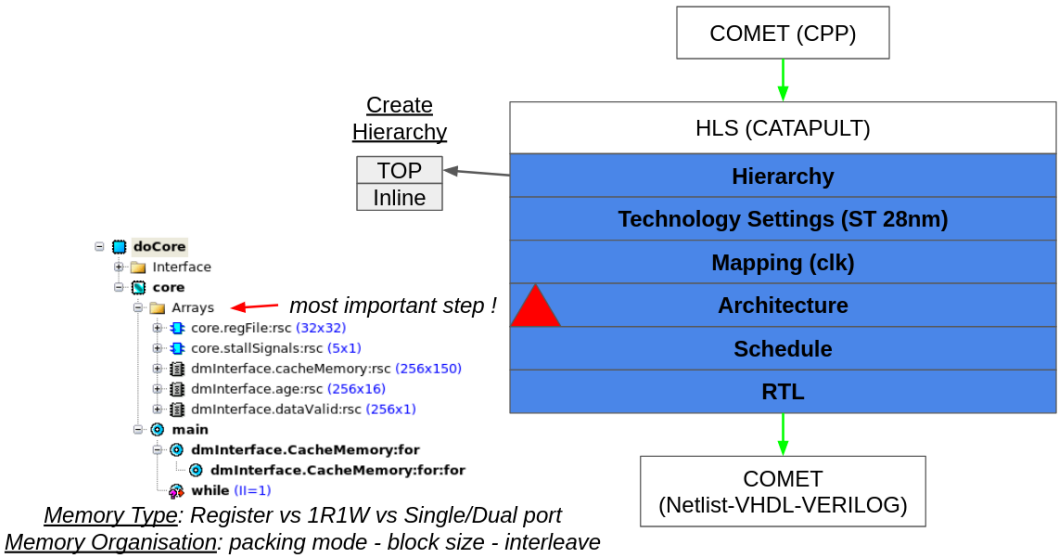

Figure 14: Catapult Flow

VII. Multi-Core Comet

I. Introduction to multi-core

The making of a multi-core architecture is a challenge that must involve both software and hardware coherency. The following section will separate the multi-core challenge in three parts: software, ISA and hardware architecture. The software part will focus on parallelism limits and the different challenges underlying such a programming strategy. The ISA part is the frontier between the software and the hardware design. We will see what kind of extensions must be provided to the Comet core in order to obtain a complete multi-core system. Then, the hardware part will focus mainly on the memory system and especially the caches.

II. Multi-processing Software

II.1 Multi-core applications

Multi-processing software is undeniably an efficient way to obtain faster results provided that we can set up parallel calculations. For instance, an embedded system that makes cryptography computations and at the same time does independently machine learning prediction would appreciate to have a dual core processor to compute faster these two threads. However, if the program is a tightly sequential algorithm, the burden created by separating the processing between two cores may not justify the cost (silicon, energy, computation time). There is not a linear progression between the speedup and the amount of processors you can use in parallel for a fixed problem size. Indeed, the gain in speedup follows the well-known Amdahl law. There is another law, the Gustafon's law, claiming that in practice we set the size of the problems to fully exploit the computing power so that we have a different performance prediction. The Gustafon'law "is based on the approximations that the parallel part scales linearly with the amount of resources, and that the serial part does not increase with respect to the size of the problem" [3].

- **Amdahl** : $speedup = \frac{1}{s+\frac{p}{N}}$
- **Gustafon**: $scaled\_speedup = s + p * N$

where s is the proportion of execution time spent on the serial part, p is the proportion of execution time spent on the part that can be parallelized, and N is the number of processors.

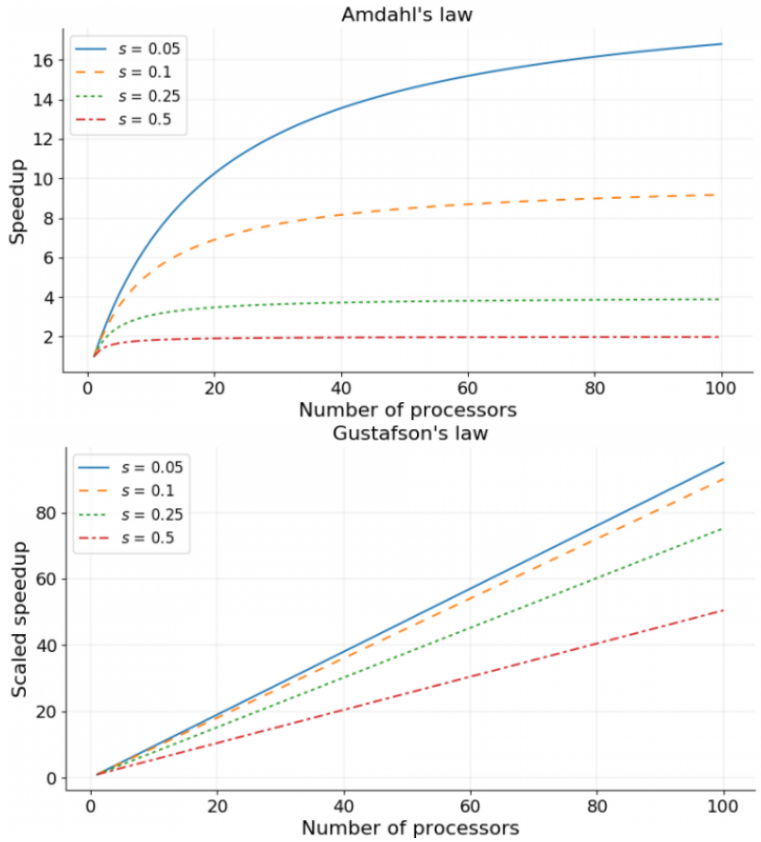

Figure 15: Amdahl's Law

II.2 Multi-processing Challenges

The first challenge to solve when considering multi-processing program is how the memory is seen by the programmer. Indeed, the hardware can provide two types of memories for multi-core architecture: *shared memory* or *distributed memory*. The two different memory architectures imply two different ways to program:

- *With the shared memory*, data and instructions are seen by all the cores without restrictions. The programmer does not have to think about the distribution of the data (or the instructions) among the cores.
- *With the distributed memory*, the programmer needs to explicitly determine how the data and the instructions memory should be split among the cores. This method allows a core to work independently without interfering too much with other cores. If a core needs to communicate with another one, there is a special protocol named "*Message Passing*" that transfer the shared data.

The advantages and drawbacks of the two strategies can be resumed with the following table:

	Message Passing	Shared Memory
Communication	Programmer	Automatic
Data Distribution	Manual	Automatic
Hardware Support	Simple	Extensive
Programming	Explicit - Specific compiler	
Correctness	difficult	less difficult
Performance	difficult	very difficult

The message passing strategy forces the programmer to think with much details how each core acts on the data. This kind of strategy imposes that the programmer will by himself do the data distribution among the cores. At contrary, the shared memory model will automatically do the distribution but the programmer has to carefully add some synchronization code snippets in order to have a coherent execution of the program. We will use the *shared memory* strategy because of the ease of programming for the core user. Moreover, there is lower overhead (latency) and a better use of bandwidth when communicating small items between cores. There is also less remote communications thanks to the automatic caching of the data.

The second challenge that comes with the shared memory strategy is the *synchronization*. To understand synchronization, an example is necessary. The following example uses two threads for counting occurrences of letters of a string. The first thread counts the first half of the string while the second one counts the second half.

```
 Core 0                  |      Core 1
1. LW L, 0(R1) # L="A"          LW L, 0(R1) # L="A"
2. LW R, Count[L] # 15          LW R, Count[L] # 15
3. ADD R, R, 1 # 16             ADD R, R, 1 # 16
4. SW R, Count[L] # 16          SW R, Count[L] # 16
```

In this example, because the two threads can simultaneously without synchronization between them, instead of counting A two times, it will maybe count A only once (line 3 is done the same time). In order to have a correct program execution, it needs to protect the critical section (between lines 2 and 4) with a lock (see the solution below): it is called *synchronization*.

```
 Core 0                  |      Core 1
1. LW L, 0(R1) # L="A"          LW L, 0(R1) # L="A"
LOCK                            LOCK
2. LW R, Count[L] # 15          LW R, Count[L] # 15
3. ADD R, R, 1 # 16             ADD R, R, 1 # 16
4. SW R, Count[L] # 16          SW R, Count[L] # 16
UNLOCK                          UNLOCK
```

The lock mechanism authorizes only one core to enter inside the part between the lock and the unlock. The lock can be implemented thanks to a simple variable and atomic instructions (see RISCV ISA part). This synchronization issue is fundamental to multi-processing programming. There is another type of synchronization called "*barrier synchronization*". The barrier is often used when a program needs to wait the results of multiple threads before going forward. The idea is that all threads must arrive to the barrier before any can leave it. A lot of programming challenges can be deduced with this simple example but they are mainly caused by the *memory consistency*.

The third challenge is the memory consistency model. While coherency defines order of accesses to the same address, consistency defines order accesses to different addresses. We need a memory model in order to define the rules. The model is the contract between the software and the implementation about the set of legal behaviors. A simple answer to what is a memory consistency model is that it specifies the values that can be returned by loads [5]. Again, an example will help to explain this challenge:

```
      Core 0    |  Core 1
      A = 0        B = 0
      A = 1        B = 1
L1:  if(B==0)     if(A==0)
```

The condition L1 can easily give the following answers (False, False), (True, False) or (False, True). But suppose the write invalidate is delayed, and the processor is allowed to continue during this delay; then it is possible that both Core 0 and Core 1 have not seen the invalidation for B and A (respectively) before they attempt to read the values. The question is, should this behavior be allowed, and if so, under what conditions ? The most straightforward model for memory consistency is called *sequential consistency*. Sequential consistency requires that the result of any execution be the same as if the accesses executed by each processor were kept in order and the accesses among different processors were interleaved. Sequential consistency ensures that all memory accesses execute as if we process them one at a time, each time selecting a core and letting it complete its next access in program order. Note that this does not require round-robin selection of cores – it even allows one core to be selected several times in a row. This allows for many possible interleaving of accesses from different cores, but it prevents accesses from one core from being reordered. But this consistency model is very bad for performance. That is why there are some relaxed models that give better performances but at the expense of a less predictable program execution.

For instance, consider the following situation:

```
Thread 0:                Thread 1:
D=1;                  printf("L:%d ", L);
L=1;                  printf("D:%d ", D);
```

Under *sequential consistency*, the possible outputs are L:0 D:0 (if Thread 0 does both of its accesses after Thread 1 prints both variables), L:1 D:1 (if Thread 0 does both accesses before Thread 1 prints them), or L:0 D:1 (if Thread 1 prints L, then Thread 1 modifies D, and then thread 1 prints D). The outcome of L:1 D:0 cannot happen because the printout of L:1 means that Thread 0 did its L=1 access, and sequential consistency ensues that by that time D=1 is also complete so D:0 cannot be printed after printing L:1. In contrast, *weak consistency* allows the two accesses in Thread 0 to be reordered, in which case it is possible to print out L:1 D:0 (print both L and D in Thread 1 after L=1 has executed but before D=1 is executed in thread 0). The different consistency model can be resumed with the following graphs (see figure 16):

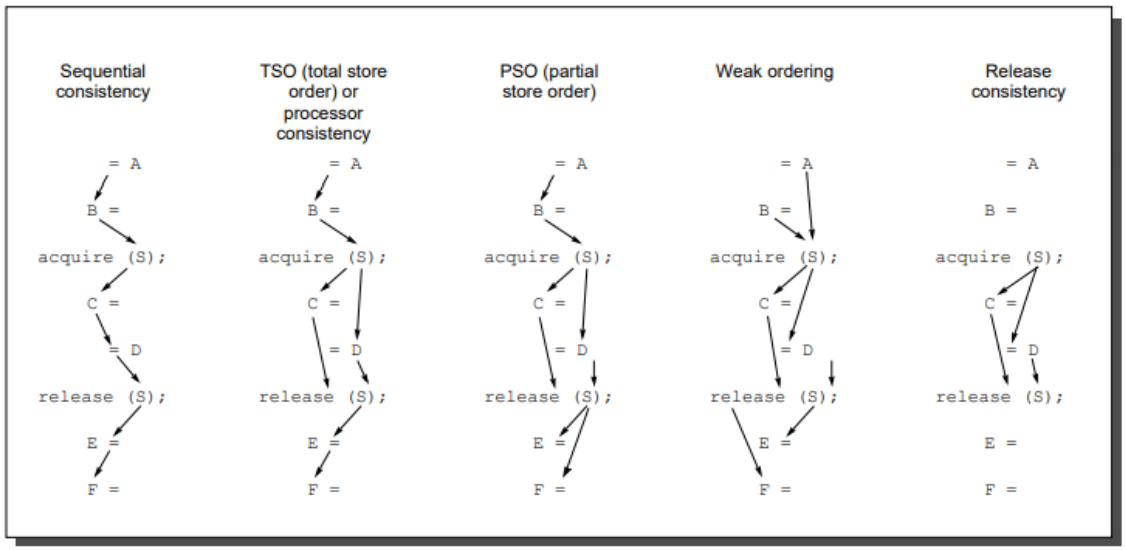

Figure 16: Consistency Models [4]

In order to solve issues that result from consistency models (like data races for instance), synchronization methods can help but there are also special instructions called FENCE that force ordering of instructions by saying that no other core can observe any operation (instructions) in the successor set following a FENCE before any operation in the predecessor set preceding the FENCE.

III. RISCV ISA

III.1 RISCV Memory Model (RVWMO) - Atomic Operations

The memory model of RISCV is named the RVWMO model. RVWMO stands for RISC-V Weak Memory Ordering. This memory model is defined thanks to axioms. A RISC-V hardware implementation is allowed as long as it does not fail to respect any axioms (see figure 24 in Annexes). The Comet core supports the 32-bits base Integer RISCV ISA. In order to have a multi-core system, the ISA must be extended to have atomic operations (A) and Zicsr extensions.

Atomic operations are mandatory to create a multi-core system. These operations prevent a core to modify a specific variable while another core has decided otherwise. There are three important atomic operations: *Load-Reserve, Store-Conditional, Swap-Write*. More atomic operations are possible but they can be re-created at a low-level software layer with only the three atomic operations aforesaid.

The *Load-Reserve* instruction is a basic load at a memory address given by the user. This load is special because it will save the address at a specific register named the *registration register* and will validate a bit (named *aq* bit here) that is associated with this register. The *Load-Reserve* instructions is always associated with the *Store-Conditional* to build lock-free data structures. "An SC instruction can never be observed by another RISC-V hart before the LR instruction that established the reservation." (RISC-V ISA specification) The *Store-Conditional* is a write that is effectively done *if and only if* the registration register is equal to the address of the store and if the *aq* bit is high. If a core X has written to the address saved in the registration register of the core Y before the Store-Conditional and after the Load-Reserve of the core Y, the *aq* bit of the core Y will be invalidated. Regardless of success or failure, executing a Store-Conditional instruction invalidates any reservation held by this hart. If the Store-Conditional is successful, the core will be notified by receiving an answer from the cache: a zero. At contrary, if unsuccessful, it will receive a one. The *Swap-Write* reads the data at the memory address specified as if it was a load and writes after, at the same address, a new value given by the instruction as if it was a store.

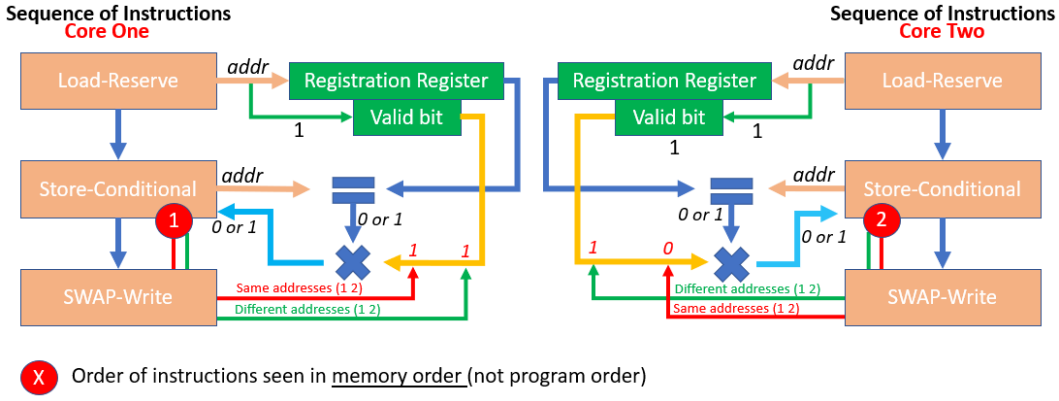

Figure 17: Atomic Operations

IV. Multi-core Architecture

The benefits of having multi-core hardware system should not be seen as obvious as we saw with the Amdahl's law. It is sometimes more interesting to have a better single core with less silicon area, better energy consumption and better *instruction per cycle* (IPC). But it can be proven that there are diminishing returns to optimize too much a single core. Moreover, one solution to get a better core is to increase the frequency and thus the voltage which is directly linked to the power (cubic relation). Thus a dual-core can have better benefits than a single very optimized processor (and vice versa).

Let us do a simple example to convince ourselves that this is not as simple. If we have a single core processor that takes $1cm^2$ silicon area with an IPC equal to 2.5 working at 50W for 2GHz, there are two strategies possible to improve the hardware system: a better single core or more cores. With the first strategy, we can obtain, for example, a new core with an IPC of 3.5 taking 2 cm^2 silicon area at 75W for 2.2 GHz. The second strategy can get a multi-core system taking obviously $2cm^2$ with single cores that work with an IPC of 2.5. They totally spent 100W at 2 GHz. The speedup of the first strategy is 1.1 (frequency optimization) * 1.4 (IPC optimization) = 1.5 while the second one is 2 *assuming that the program can simply be divided among cores.*

IV.1 Global Strategy

The first thing that must be think when considering multi-core hardware system is how to ensure memory coherence and especially cache coherence. Indeed, each core has its own cache which implies a coherency issue that can be illustrated with the following problem:

```
A = 0

Core 0      |  Core 1
.           |  .
Lw R0 <= A  |  .
R0++        |  .
Sw R0 => A  |  .
.           |  .
.           |  Lw R0 <= A
.           |  R0++
.           |  Sw R0 => A
```

We need to ensure that Core 1 will write 2 to A. This can be done if Core 0 informs to Core 1 the new value of A which is stored in its cache. This is called *cache coherency*. There are **three requirements** to get cache coherency:

- A Read address X on core 0 returns the value written by the most recent write to X on core 0 if no other core has written to X in between.
- If core 0 writes to X and core 1 reads after a sufficient time, and there are no other writes in between, core 1's read returns the value from core 0's write.
- Writes to the same location are serialized: any two writes to X must be seen to occur in the same order on all cores.

There are two steps for creating a hardware implementation with cache coherency. First, you need to decide how to broadcast information about cache blocks modifications: you can either broadcast writes in a shared bus (*snooping protocol*) or use a central directory (*directory-based*). Then, you need to choose how you update data in all caches: you can either update all values (*write-update*) or invalidate blocks that are no longer valid (*write-invalidate*). The differences between the write-update and the write-invalidate protocol can be resumed in the table below:

Application does...	Update	Invalidate
Burst of writes to one address	Each write sends an update (Bad)	First write invalidates (Good)
Write different words in the same block	Each write sends an update (Bad)	First write invalidates (Good)
Producer/consumer	Updates to consumers (Best)	Prod Inv Cons misses (Bad)

It appears that write-invalidate is slightly stronger, especially when there is an OS that transfer a thread to another core because the write-update could keep updating the old thread related to the old core's cache.

IV.2 Multi-core Cache Controller - MSI Protocol

The global multi-core architecture chosen is a write-invalidate protocol with a directory based shared memory. This strategy can be resumed with the following figure 18:

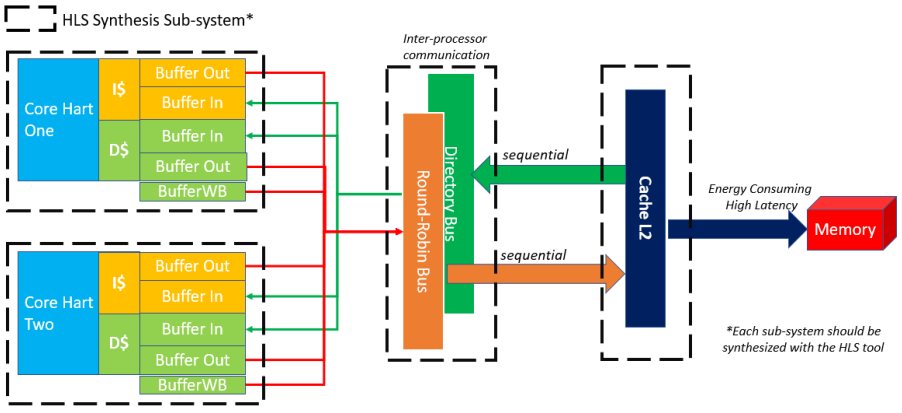

Figure 18: Memory Architecture

We have created a specific protocol that aims to communicate efficiently with the directory (see figure 18). The main challenge for the cache L1 is to be able to send an instruction to the cache L2 but also to be able to answer to specific requests sent by the cache L2.

```
/* Basic Interface Skeleton */
struct cacheDirectoryInterfaceBase {
  // in or out
  ac_int<32, false> addr;
  ac_int<1, false> interfaceAck;
  ac_int<1, false> releaseAck;
};

/* Buffer Out / WB Interface */
typedef struct cacheDirectoryInterfaceOut : public cacheDirectoryInterfaceBase {
  memOpType interfaceRequest;
  // data out
  ac_int<32, false> buffer;
  // data In
  ac_int<32, false> bufferIn;
} cacheDirectoryInterfaceOut;

/* Buffer In Interface */
typedef struct cacheDirectoryInterfaceIn : public cacheDirectoryInterfaceBase {
  ac_int<1, false> protocolMiss;
  directoryOpType interfaceRequest;
} cacheDirectoryInterfaceIn;

/* Cache L2 Interface --> Out */
typedef struct cacheL2DirectoryInterfaceOut : public cacheDirectoryInterfaceBase {
  // data out
  ac_int<32, false> buffer;
  ac_int<1, false> protocolMiss;
  directoryOpType  interfaceRequest;
  ac_int<4, false> hartid;
  ac_int<1, false> cache;
} cacheL2DirectoryInterfaceOut;

/* Cache L2 Interface --> In */
typedef struct cacheL2DirectoryInterfaceIn : public cacheDirectoryInterfaceBase {
  // in to cache
  memOpType interfaceRequest;
  ac_int<4, false> hartid;
  ac_int<1, false> cache;
  // out from cache
  ac_int<32, false> dataLoad;
  ac_int<1, false> protocolMiss;
  ac_int<1, false> rejection;
  // Special WB input
  ac_int<1, false> interfaceAckWB;
  ac_int<1, false> releaseAckWB;
  ac_int<32, false> addrWB;
  // data in
  ac_int<32, false> buffer;
  ac_int<32, false> bufferWB;
} cacheL2DirectoryInterfaceIn;
```

```
typedef struct directoryStruct {

  cacheStatus cacheStateDir = INVALID;              // cache status
  ac_int<log2const<NB_CORES>::value, false> owner; // Who is the owner ?
  ac_int<NB_CORES, false> sharers   = 0;            // Which cores are sharing the data ?
  ac_int<NB_CORES, false> coreToken = 0;            // Does a core use this directory line ?

} directoryStruct;
```

This protocol provides a way to manage caches dynamic by sending information to the directory or to the caches. There are two different types of request: directory requests and cache requests.

Requests	Goal
Directory	
REQ_INV	Request for invalidation of a cache block
REQ_DATATRANSFER_INV	Request to send a cache block and invalidate it after
REQ_DATATRANSFER_SHARED	Request to send a cache block and change cache status to SHARED
Cache	
LOAD	Load a data at a specific address.
STORE	Store a data at a specific address.
WRITEBACK	Send a cache line to the next level of caches.
WRITEHIT	Send a cache write hit. Other caches sharing this cache block must be invalidated.
AMO_(LR/SC/SWAPW)	Atomic operations requests.
MISS_(WRITE/READ)	Ask for permission to enter in a miss state.
END_MISS	Request to release the cache lock inside the directory.

The cache keeps its previous FSM but we have now added a specific sub-FSM (orange and blue links in figure 19) for atomic operations and directory requests. The C++ design of the cache tries to be at much as possible close a single core cache FSM in order to limit the silicon surface by re-using existing sub-FSM.

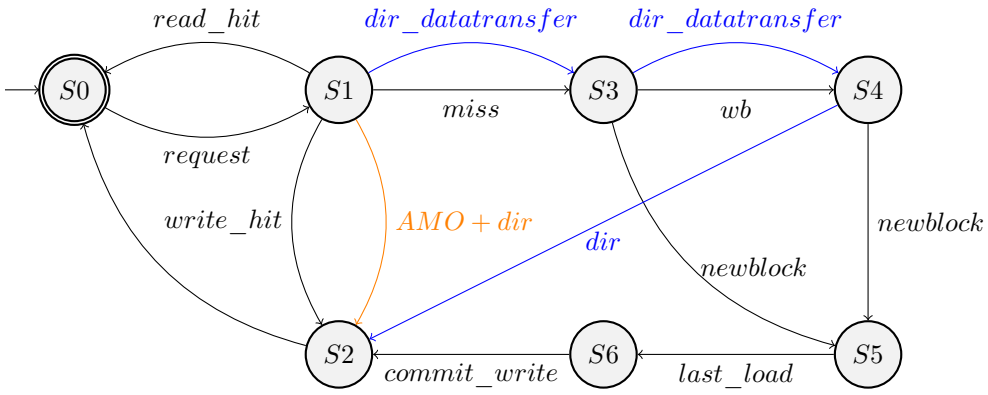

Figure 19: Cache Macroscopic *Incomplete* Finite State Machine (*wb*: write-back, *newblock*: new cache block, *dir*: directory request, *AMO*: atomic operations) for a multi-core system.

But this FSM is incomplete because there are multiple strategies possible that prioritize instructions from the directory given by the cache controller and the core given by the current instruction inside the core:

- Directory requests over core instructions.
- Core instructions over the directory requests.
- First in, first taken.

The only viable strategy is that directory requests are prioritized over core instructions. Indeed, if we let the cores instructions be prioritized, there will be possible deadlocks when processors need each other at the same time. The directory is the chief of the operations and should cleverly allocate priorities precisely to each core at the right moment as we will see in the next part.

The previous FSM must be augmented in order to get some in-between states that ensure the cache can go forward without creating some incoherence between the cores. In order to get an efficient multi-core design, we also must avoid to overwhelm the directory with requests. Indeed, a request to the directory costs at least one cycle of penalty. For instance, if we have a write hit, the cache with the hit must warn the others but not always as you can see in the table below:

Cache Status	Next Cache Status	Directory Need (Y/N)	Penalties
Exclusive	Modified	Y (change directory state)	1 cycle
Shared	Modified	Y (REQ_INV)	≥ 3 cycles
Modified	Modified	N	0 cycle

Table 1: Cache Status Dynamic - Write Hit

Cache Status	Write-Back (Y/N)
Invalid	N
Exclusive	N
Shared	N
Modified	Y (4 cycles penalties)

Table 2: Cache Status Dynamic - Write Back

The miss state is a special state of the cache. During a miss state, the cache will not receive a directory request (guarantee by the directory). This state will block the others cache that would like to use that specific cache for any reasons (Write Hit, Miss Write, Miss Read, etc.).

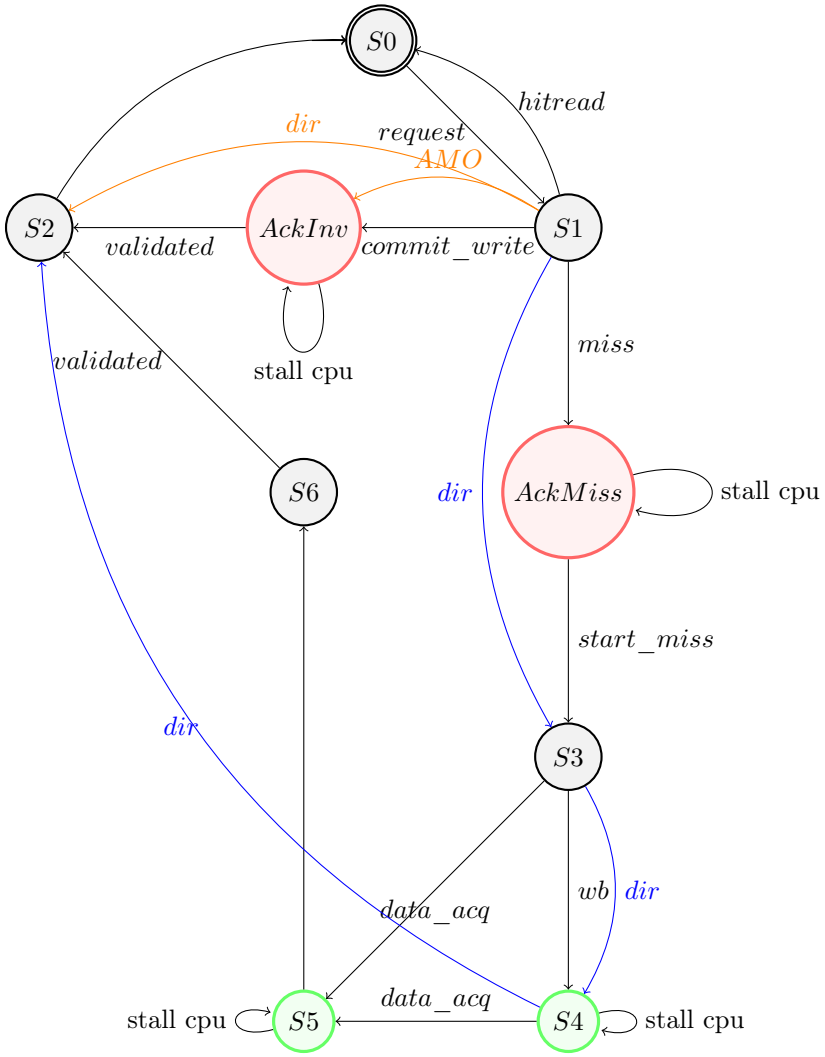

Figure 20: Cache Controller CacheToDirectory (1/2) Macroscopic FSM

- S0: There is no need to communicate something.
- S1:

Read Hit: Do nothing.
Write Hit: It must send a request to the directory to invalidate "sharers" (when in a Shared/Exclusive cache state).
Miss: It must send a request to the directory for a miss encountered at the specific address.

- S2:

Hit Write: I must write only if all other "*sharers*" have been invalidated (AckInv).
Miss: I must commit changes for the new cache block.
Directory: I must change the cache status of the cache block selected.

It must always correctly set the cache status (Exclusive vs Shared vs Modified vs Invalidate).

- S3/S4: We perform the write-back (WB) or data fetching if the directory has granted us (AckMiss).
- S5: We should get the data needed for the new cache block.
- S6: We end the communication with the directory.

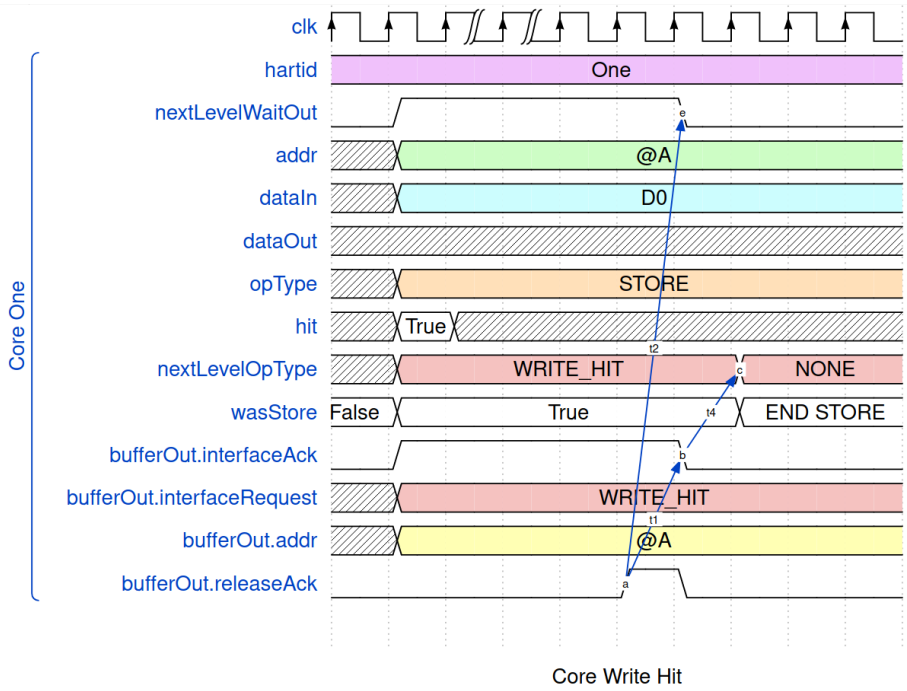

Figure 21: Write Hit Protocol Multi-core

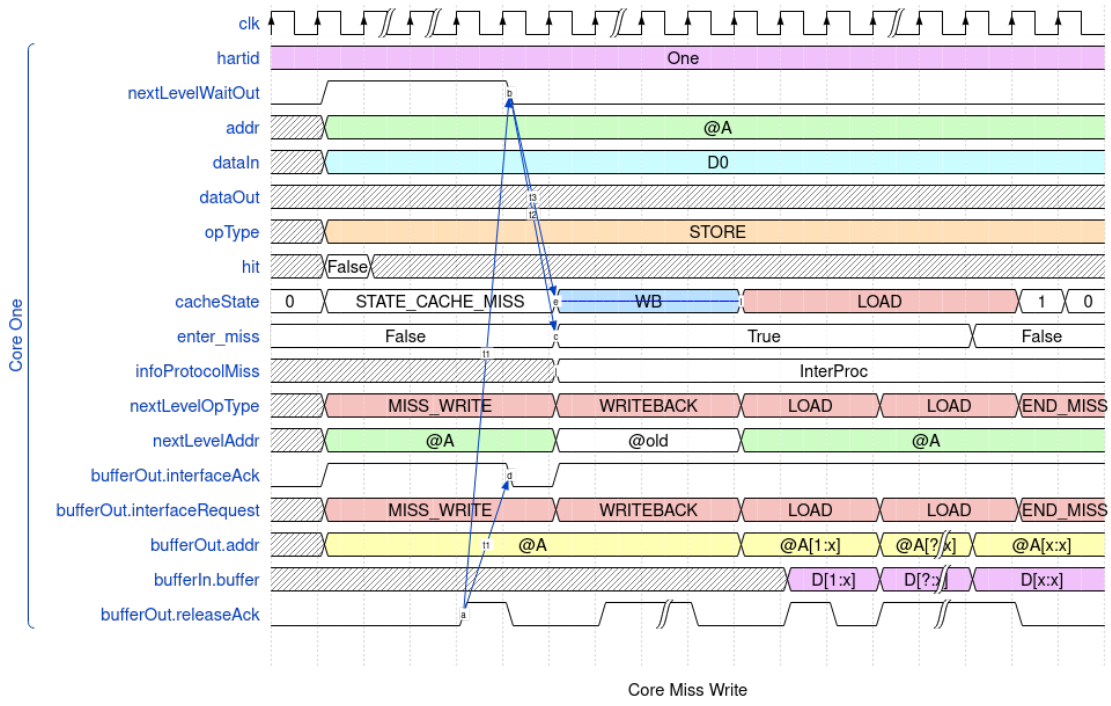

Figure 22: Write Miss Protocol Multi-core

We are going to enumerate all the possible necessary communications from a directory request to a cache. Directory requests have higher priorities to core requests. When a request is seen by the cache, all current procedures is halted. It is thus mandatory to be able to get back to the right execution after the directory request is done. There are three types of requests:

- Invalidation
- Data transfer with invalidation
- Data transfer with shared or owner status

During a directory request, the cache controller can be in different states:

- D0: no request.
- D1: seeking the cache block to modify.
- D2: modification of the cache block status (Invalid/Shared).
- D3: Send data to the directory for another cache using mechanism of the write-back.

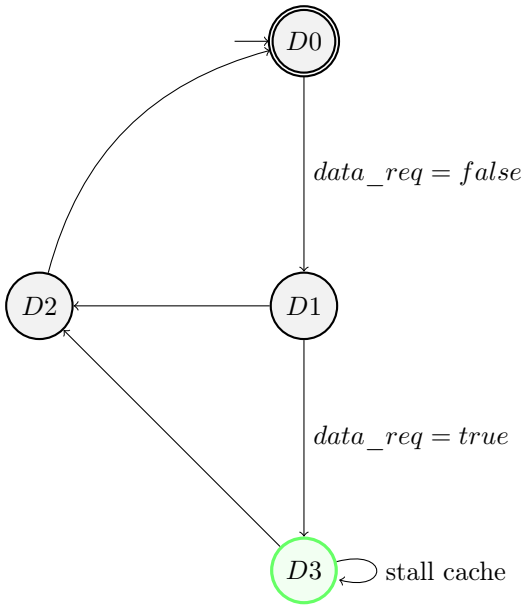

Figure 23: Cache Controller DirectoryToCache (2/2) Macroscopic FSM

VIII. Annexes

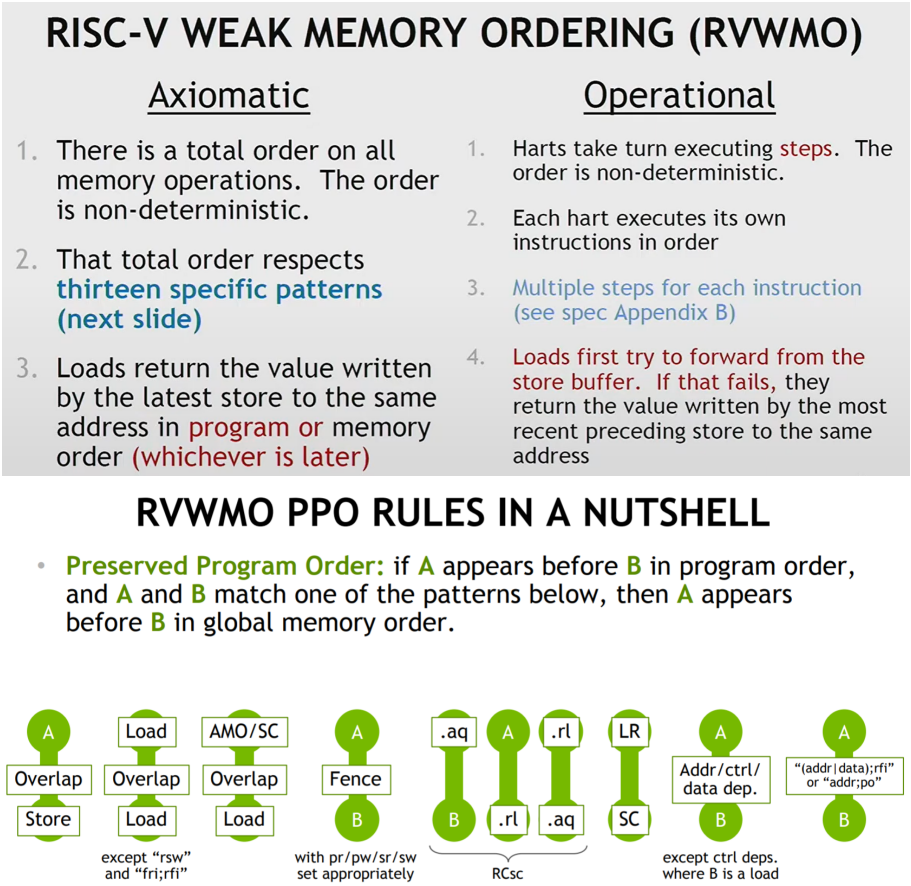

Figure 24: RVWMO - NVIDIA Slide [5]

References

[1] Comet CPU RISCV HLS https://gitlab.inria.fr/srokicki/Comet. Inria-Rennes, Simon Ronkicki

[2] Simon Rokicki, Davide Pala, Joseph Paturel, Olivier Sentieys. *What You Simulate Is What You Synthesize: Design of a RISC-V Core from C++ Specifications.* RISC-V Workshop 2019, Jun 2019,Zurich, Switzerland. pp.1-2.

[3] Xin Li *Scalability: strong and weak scaling* https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/, KTH blog.

[4] David A. Patterson, John L. Hennessy *Computer Organization and Design - The Hardware/Software Interface* Morgan Kaufmann Publishers

[5] Dan Lustig *Memory Model* https://www.youtube.com/watch?v=QkbWgCSAEoo, RISCV