What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications

Simon Rokicki, Davide Pala, Joseph Paturel, Olivier Sentieys

What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications

Invited Paper

Simon Rokicki, Davide Pala, Joseph Paturel, Olivier Sentieys
Univ Rennes, Inria, CNRS, IRISA

***Abstract*—Designing the hardware of a processor core as well as its verification flow from a single high-level specification would provide great advantages in terms of productivity and maintainability. In this work, we highlight the gain of starting from a unique high-level synthesis and simulation C++ model to design a processor core implementing the RISC-V Instruction Set Architecture (ISA). The specification code is used to generate both the hardware target design through High-Level Synthesis as well as a fast and cycle-accurate bit-accurate simulator of the latter through software compilation. The object oriented nature of C++ greatly improves the readability and flexibility of the design description compared to classical HDL-based implementations. Therefore, the processor model can easily be modified, expanded and verified using standard software development methodologies. The main challenge is to deal with C++ based synthesizable specifications of core and uncore components, cache memory hierarchy, and synchronization. In particular, the research question is how to specify such parallel computing pipelines with high-level synthesis technology and to demonstrate that there is a potential high gain in design time without jeopardizing performance and cost. Our experiments demonstrate that the core frequency and area of the generated hardware are comparable to existing RTL implementations.**

I. Introduction

Since decades software and hardware have shared a symbiotic relationship - a situation where both elements support and need each other to thrive and progress. Innovations in hardware architecture are leading to better processors which can support the complex software applications being developed today. Similarly, software and programming languages have made it easier to prototype, simulate and even synthesize hardware, for example the creation of High-Level Synthesis (HLS) tools. HLS is a hardware design technique where an algorithm written in a high-level language like C or C++ is interpreted to create digital hardware which implements the same functionality. HLS tools are very useful for designing complex controllers or accelerators, as they allow engineers to focus simply on functionality, without worrying about architectural details too much.

Standard development flows for processor architecture are based around the development and maintenance of a hardware model for synthesis, and a software model (i.e. an instruction-set simulator) to validate the applications that will run on the design. Those two models have to be verified independently. High-level synthesis would also be considerably useful to cut down the complexity of the hardware development in a processor design methodology. As shown in Figure 1, HLS enables the use of a single behavioral model of the simulator, which is compiled into a cycle-accurate bit-accurate simulator by standard compilers and also synthesized into a hardware component through HLS. It is also possible to debug the behavioral model at the C/C++ level and to simply validate the behavior of the generated hardware through co-simulation and standard software development tools.

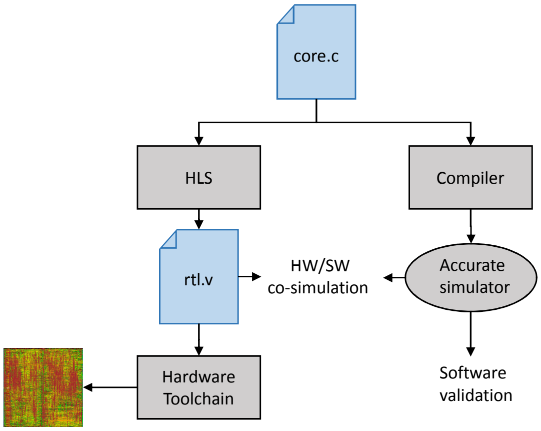

Fig. 1. Hardware processor development flow based on High-Level Synthesis.

Although high-level synthesis is making huge progress in dealing with complex structures, how far can these tools go? Can they be used to design something as complex as a microprocessor? In particular how to specify such parallel computing pipelines (e.g., core pipeline stages, cache hierarchy, communications with uncore components) with the HLS technology and to demonstrate that there is a potential high gain in design time, without jeopardizing performance and cost.

In this work, we present an attempt to answer these questions with the design of Comet, a five-stage pipelined processor implementing the RISC-V Instruction Set Architecture (ISA). Comet is fully synthesized from a unique C/C++ model using state-of-the-art HLS tools. To measure the quality of the generated hardware, we compare it against other implementations based on hardware description languages. Results show that, if the C/C++ model is written using the methodology we describe in this paper, the generated hardware is comparable or even better than other implementations when targeting an ASIC 28 nm technology node. The resource footprint on FPGA targets is slightly higher but still comparable with hand-optimized designs. Moreover, a fast, cycle-accurate and bit-accurate simulator of the processor can be compiled from the same C/C++ model. The resulting maximum performance of the simulator reaches 24 million cycles per second (Mcps) on integer workloads and around 15 Mcps on average, when executing floating-point and integer workloads. The object oriented nature of C++ also greatly improves the readability and flexibility of the design description compared to classical HDL-based implementations. Therefore, the processor model can easily be modified, expanded and verified using standard software development methodologies.

The rest of this paper is organized as follows. Section II provides the background on HLS needed to understand the challenges involved in designing a processor core using HLS. In Section III, we describe how the C/C++ models have to be designed in order to be correctly synthesized by the HLS tool. In Section IV and Section V, we present the experimental results and their discussion before to draw some conclusions and perspectives in the last section.

II. Background on High-Level Synthesis

The principle of high-level synthesis is to generate hardware components from functional representations using high-level languages (typically C/C++). While HLS could be seen as a compilation of software to generate hardware, it has to face several different challenges. For example, the tool can allocate computing units and hardware resources to exploit operator-level parallelism. HLS also needs to meet designer constraints such as target operating frequency and maximum allowed area or resource usage. The first behavioral synthesis methodologies and tools date back to the 1990s [1]. However, until the 2000s, those tools were commercial failures due to the quality of the generated hardware [2] not being high enough to compete with handcrafted designs. However, more recent commercial tools like Catapult HLS from Mentor Graphics [3], [4], [5] and Vivado HLS from Xilinx [6] generate efficient hardware for a broad set of applications. Nevertheless, HLS is still aimed at generating hardware accelerators for compute-intensive applications and is less capable when dealing with control-dominated tasks.

HLS flows are organized around two main steps. First, the tool allocates hardware components to the different operations found in the functional representation. Then, during the second step, the tool schedules the operations. The allocation and the scheduling problems are joined: allocating fewer resources to save area results may lead to a longer schedule, thus reducing the speed or the throughput of the synthesized hardware. Consequently, those two problems are often solved at the same time [7].

HLS tools also have to efficiently handle loops. This can be done through loop pipelining. The idea is to begin a new iteration of the loop before the completion of the current one. Loop pipelining is characterized by the Initiation Interval (II) achieved, which is the time between two iteration starts. Ideally, a loop is pipelined with an II of one, which means that a new iteration starts every cycle. However an II of one can be difficult to obtain as the II is constrained by both available resources and inter-iteration dependencies.

III. Designing a Processor with HLS

In this section, we present how to efficiently synthesize a processor using a behavioral description (i.e., a simulator). We first explain why HLS tools cannot generate efficient hardware from an Instruction Set Simulator. Then, we present our explicitly pipelined simulator and how to integrate multi-cycle operators (e.g. integer division, floating-point unit).

A. Synthesis from an Instruction Set Simulator

The most intuitive way to synthesize a processor from a C/C++ description is to express the ISA execution as in an Instruction Set Simulator (ISS). An ISS is a simple way to express the functionalities of the processor organized around an infinite loop implementing the ISA. At each iteration of the loop, an instruction is fetched, its fields are decoded and a switch/case statement modifies the simulator's state function of the current instruction definition. Finally, the Program Counter (PC) is incremented and the next instruction can be executed at the next iteration.

Algorithm 1 is a simple ISS example: we can see the main parts of the simulator (the infinite while loop and the switch/case). If we consider the case corresponding to the addition (ADD), the simulator simply reads the two operand registers, executes the addition and writes the result in the register file. Similarly, the case corresponding to the load instruction (LD) computes the address, accesses the memory and writes the result on the register file.

```
while true do
    instr = mem[pc];
    switch opcode do
        /* -- Jump register --          */
        case JR do
            pc = reg[rs1];
        end
        /* -- Load word --              */
        case LD do
            reg[rd] = mem[reg[rs1] + imm];
        end
        /* -- Addition --               */
        case ADD do
            reg[rd] = reg[rs1] + reg[rs2]
        end
    end
end
```

Algorithm 1: Example of an Instruction Set Simulator specification.

Thanks to their simple operating logic, Instruction Set Simulators are often used to provide the first simulator of an ISA. Moreover, studies from Rohou *et al.* demonstrated that thanks to modern branch predictors, such simulators deliver good simulation performance without needing complex optimizations [8]. Among RISC-V simulators, SWERV-ISS [9] from Western Digital is based on this idea. The Spike simulator [10] is more complex and provides higher performance, but remains based on a similar switch/case structure.

To synthesize an efficient hardware component from an ISS, some transformations need to be applied to the functional description. As Algorithm 1 shows, an addition operator is used in different *cases* of the *switch* statement (the `LD` and `ADD` cases in this example), since a single opcode is evaluated at each loop iteration, The two operators are never solicited simultaneously and a unique addition operator instance and some multiplexers can be shared between the different execution paths. This transformation is known as datapath-merging [11]. If the considered operator's inputs/outputs are different in different datapaths, it might be more or less interesting resource-wise to apply the merging transformation. Similarly, all the read and write operations to the register file could be merged into one or two accesses. This transformation allows for the design size to be reduced.

To gain on the performance front, the infinite *while* loop of the ISS can be pipelined, hence reducing the length of the critical path and thus allowing for higher throughput to be achieved. Unfortunately, current HLS tools cannot achieve an II of 1 with such an ISS-like processor description. Indeed, several dependencies would be detected by the compiler:

- The Read After Write (RAW) dependency on the register file: when executing a *load* instruction, the value to write in the register file has to be fetched from memory at an address that is stored in the register file itself. If the previously executed instruction modified the register that contains the target address, its complete execution is required before the current instruction can be evaluated and the memory can be accessed. As HLS tools always schedule for the worst case, they will not be able to reach an Initiation Interval of 1 on their own.
- The next PC value: the PC value for the next loop iteration can come from different paths. If the current instruction is a conditional branch, the condition has to be evaluated (which often needs register file accesses and integer comparisons). Therefore, depending on this condition, an immediate value may be added to the current PC to obtain the next value. Also, when executing an indirect branch, the next PC value comes from the register file. Because of this dependency on the PC value, HLS tools determine the following PC value at the end of the longest path, which is the conditional branch. This leads to an Initiation Interval greater or equal to 3, depending on the number of pipeline stages of the processor.

The first dependency could be handled by HLS tools if they were capable of scheduling for the most probable path, stalling the pipeline when necessary. The dependency related to the next PC value requires to generate a speculative pipeline: starting an iteration with the most probable PC value (which is often an increment of the current value, pointing to the next instruction in memory) and canceling the execution if the prediction turned out to be wrong. These two transformations could be handled by the HLS tools and represent interesting research directions. It is also noteworthy that, if the tools were capable of generating such schedules, we would get a pipelined processor that resembles the ones that are manually designed.

To summarize, synthesizing a processor core from an Instruction Set Simulator would lead to very low performance as current HLS tools are not capable of efficiently pipelining and breaking dependency in such specifications. Consequently, to develop a processor core using HLS, we need to write a simulator where the pipeline structure is explicitly encoded at the specification level.

B. Explicitly Pipelined Simulator

As explained in the previous section, current HLS tools are not capable of generating efficient pipelines for processors, that is to generate pipelines which speculate on the next value of the PC and capable of handling stall and forwarding mechanisms automatically. As we want to design a core with a performance level close to what is achievable with an HDL, we designed an *explicitly pipelined simulator*. In this simulator, the structure of the pipeline, as well as all the control mechanisms (stall and forward), are specified at the C/C++ level. As described in Figure 2, our simulator uses a 5-stage pipeline (*Fetch*, *Decode*, *Execute*, *Memory* and *Write Back*), with a forwarding mechanism from the output of the *Execute* and *Memory* stages to the input of the *Execute* stage. Figure 2 also represents data and instruction caches, as well as multi-cycle operators. The design of these latter components is discussed in Section III-C.

Algorithm 2 represents the high-level description of Comet, our processor core with the explicit pipeline structure. The different pipeline registers are defined as `struct` variables (e.g., `ftodc` represents the pipeline register between the *Fetch* and *Decode* stages), these variables are declared outside the scope of the main execution loop and will hence retain their values between iterations. Each iteration of the infinite loop executes the functions of all the pipeline stages on their respective input registers. Each pipeline stage function writes its output in a temporary register (that is not synthesized) that represents the data present at the input of the following physical pipeline registers. Note that five different instructions are alive in the simulator at the same time, one for each pipeline stage.

After executing all the pipeline stages, the simulator handles the stall and forward logic. First, a stall signal is computed for each pipeline stage. If a stage is not stalled, the temporary values that represent its output will be copied over to the physical registers. Stall signals can be activated if an external global stall signal is set, when a cache miss occurs or when a Read After Write dependency that cannot be solved with forwarding, is detected. With the latter stall source, only the two first pipeline stages are inhibited while the *Execute*, *Memory* and *Write-Back* stages keep committing their results. Forward mechanisms simply compare source and destination registers and trigger a forward bit. If this bit is set, one of the input values of the *Execute* stage is modified with the result of the *Execute* or the *Memory* stage to avoid stalling the pipeline and wait for the value to be written in the register file.

The simulator described in Algorithm 2 has therefore no inter-loop dependencies and can hence start the execution of a new instruction at every clock cycle, leading to an II of one. It is also interesting to note that the explicit pipelining

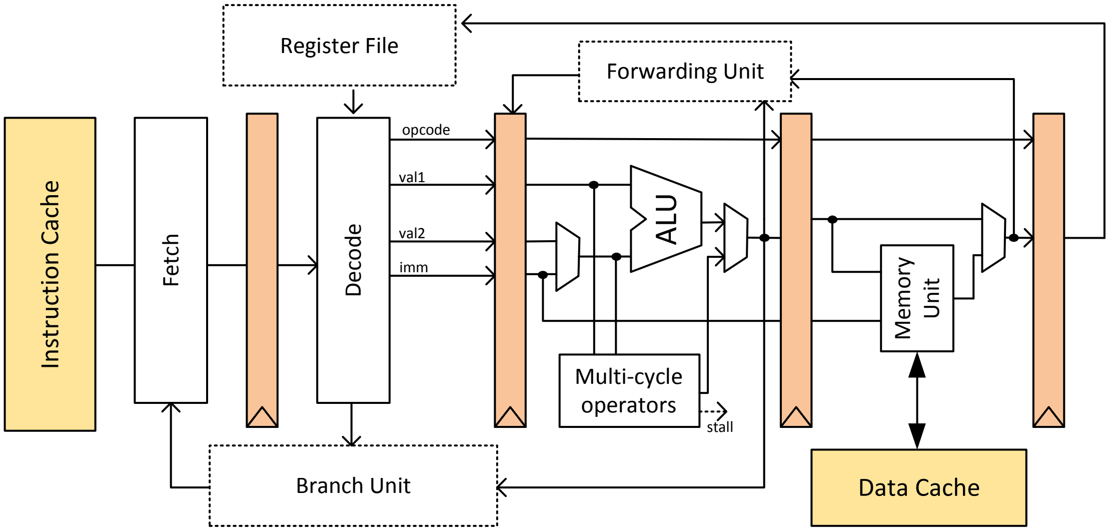

Fig. 2. Internal organization of a processor with a standard RISC five-stage pipeline, forward mechanisms, multi-cycle operators, and data and instruction caches.

```
struct FtoDC ftodc;
struct DCtoEx dctoex;
struct ExtoMem extomem;
struct MemtoWB memtowb;
while true do
    ftodc_temp = fetch();
    dctoex_temp = decode(ftodc);
    extomem_temp = execute(dctoex);
    memtowb_temp = memory(extomem);
    writeback(memtowb);
    /* -- Handling stalls --              */
    bool stall[5] = stallLogic();
    if !stall[0] then
        ftodc = ftodc_temp;
    end
    if !stall[1] then
        dctoex = dctoex_temp;
    end
    ...
    /* -- Handling forwarding --          */
    bool forward = forwardLogic();
    if forward then
        dctoex.value1 = extomem.result;
    end
end
```

Algorithm 2: High-level specification of an explicitly pipelined simulator.

transformation (effectively removing all the timing flexibility of the HLS compiler) makes the simulator cycle-accurate bit-accurate, unlike a standard solution based on an ISS.

C. Handling Multi-Cycle Operators and Caches

The simulator described in the previous section has only one pipeline stage dedicated to execution. While this is enough for most of the instruction of the base RISC-V ISA, some extensions introduce operations requiring a longer execution time (e.g., the integer division of the M extension of the ISA). If the instruction is simply added into the simulator described on Algorithm 2 in a new case statement using a normal division operator, the synthesized design will see its critical paths greatly increased and therefore its operating frequency lowered accordingly. To counter this side effect, the division needs to be broken down into small control steps. An implementation based on a `for` loop is not suitable, as HLS compilers always schedule for the worst case and will hence lower the II of the pipeline to the number of iterations needed to fully execute the division algorithm.

To solve this issue, multi-cycle operations are expressed as Finite State Machines (FSM) and the execution stage is stalled when a multi-cycle opcode is detected. Each state of the FSM corresponds to a computation cycle. In the case of the division, there is a state for each bit in the operand (groups of bits can be processed together using loop unrolling techniques to increase performance). When a multi-cycle opcode enters the execution stage, the pipeline will be stalled until the opcode FSM has reached its final state and a result has been produced. All the commit and forwarding logic is kept from the original pipeline.

As multi-cycle operators are synthesized alongside the processor, HLS tools can perform some resource sharing between the normal mono-cycle operations and the multi-cycle ones. For example, the division is implemented using successive subtractions. The HLS tools allow for the subtractor circuit of the Arithmetic and Logic Unit (ALU) to be re-used in the execution of the division. The drawback of this approach is that the computation of the multi-cycle operators has to be split into atomic pieces that can be executed in one cycle and the designer has to encode a state machine that guides the execution of the algorithm and generates the stall signal. This solution has been used to implement the division instruction as well as the Floating-Point Unit (FPU) of the Comet processor.

Less intuitively, we used a similar mechanism to handle associative data and instruction caches. Caches load the tags associated to the requested address. If there is a match, they return the corresponding value. If no match is detected, the caches stall the core while a line is freed and populated with values from the next memory level in the hierarchy. These two match/miss possibilities, and the processing associated to each, are sequenced using an explicit state-machine. Because of the different memory accesses, synthesizing this part is challenging. We used HLS directives to ignore some of the dependencies detected by the tool when we knew that conflicts never occur.

To illustrate the usage of the previously exposed optimization techniques, we present in the next section Comet, a RISC-V compatible 32-bit CPU based on the micro-architectural features illustrated in Figure 2.

IV. Design of the Comet Processor Core

In this section, we describe the experimental study conducted and the main results obtained. This study aims to: i) evaluate the quality of the generated hardware and compare it with existing RISC-V cores; ii) measure the performance of the simulator; iii) demonstrate how HLS improves the maintainability and the extensibility of a processor core design.

A. Quality of the Generated Hardware

The first part of the study is focused on measuring the quality of the generated hardware. Indeed, current HLS tools are not built for control-flow dominated C++ code such as a processor core simulator. To ensure that the synthesized core is competitive, we compare its characteristics against other similar cores.

The Comet core has been synthesized with different configurations: i) based on the `rv32i` that only supports basic instructions; ii) based on `rv32im` that has additional support for 32×32 integer multiplication and division, and iii) based on `rv32imf` that supports all the 32-bit single-precision floating-point instructions. Note that this last configuration also adds 32 32-bit floating-point registers. All those three different versions of Comet are synthesized with Mentor Graphics Catapult HLS (V10.3a), and then synthesized with Synopsys Design Compiler (P-2018.06-SP5). Each configuration targets an operating frequency of 700 MHz and the 28 nm FDSOI technology node from ST Microelectronics, using a V_{dd}=1V, 25°C corner.

As baselines, we used several other cores written with more standard hardware description languages. The Rocket core [12] is a 5-stage pipelined core developed at Berkeley using the Chisel HDL [13]. Its micro-architecture is close to the one of Comet but it also supports additional RISC-V extensions (compressed and atomic instructions), which may slightly impact its area results. As the Rocket core is fully configurable, we generated three versions comparable to the three Comet configurations. PicoRV is a size-optimized RISC-V core: it has no barrel shifter and only supports the `rv32i` ISA. PicoRV is written using Verilog HDL. Those different cores were synthesized using the same synthesis flow and technology parameters.

Table I contains the synthesis results for the different cores. For the `rv32i` configurations, our implementation is larger than the size-optimized PicoRV but smaller than the Rocket core. Once support for the M-extension added, Comet becomes smaller than PicoRV and Rocket, even though the difference is less significant. The addition of the floating-point extension changes this tendency, the Rocket core for `rv32imf` being slightly smaller than its equivalent for Comet. However, the important conclusion in this study is that the use of HLS does not significantly impact the area and frequency of the synthesized core, even when compared to an optimized version designed at the RT level.

TABLE I
Area and frequency results for different RISC-V cores synthesized using a 28 nm technology node.

Core	ISA	freq. (MHz)	Area (μm^2)	Lang.
Comet [14]	rv32i	700	8 168	C++
	rv32im		11 099	
	rv32imf		26 760	
Rocket [12]	rv32i		11 114	Chisel
	rv32im		12 606	
	rv32imf		26 550	
PicoRV [15]	rv32i		7 747	Verilog
	rv32im		11 176	

To investigate the impact of the floating-point extension, sub-parts of the Comet core have been synthesized separately: the FPU and the core alone (note that the core still contains the 32 floating-point registers). The results of this experiment are given in Table II. Intuitively, synthesizing sub-parts of the design separately should lead to a larger design, as the HLS tool would no longer perform resource-sharing optimizations between those different parts. However, the results in Table II show the opposite: synthesizing the FPU and the core separately leads to better results. This counter-intuitive result seems to point out that we are reaching the limits of what current HLS tools can perform. As the control-flow grows in complexity with the addition of an FPU, the optimization algorithms of the HLS tool are not capable of optimizing the design as much as if it was split into smaller and simpler sub-designs. This hypothesis also holds if we measure the time spent by the HLS tool to synthesize the design. For the full core with the FPU, the compilation time reaches 23 minutes, whereas the synthesis of the core and FPU alone took two minutes each.

TABLE II
Area results for Comet floating point unit synthesized using a 28 nm technology node.

Design	Area (μm^2)
FPU	8 147
Core w/o FPU	15 299
Core w/ FPU	26 760

Another advantage of HLS is its ability to synthesize a hardware design specialized for various technology targets. As an example, we synthesized Comet as a soft core embedded into a Xilinx Artix 7 FPGA (XC7A12T). The core itself is still synthesized using Catapult HLS (V10.3a) and the RTL code is then synthesized and placed and routed using Vivado 2018.3. Table III summarizes results in terms of area and maximal frequency for different RISC-V cores. Area is expressed as the number of Look-Up Tables (LUT), Flip-Flops (FF), Multiplexers (Mux), and arithmetic (DSP) blocks used in the FPGA fabric. Even if the comparison between Rocket and Comet is more difficult on FPGA, the area and frequency of the synthesized designs are comparable. We can notice that Comet uses more multiplexers and arithmetic blocks than Rocket for each configuration. On the other side, PicoRV is far smaller than Comet and operates at a higher frequency. The manually optimized FPGA design of PicoRV is much more efficient than both Rocket and Comet designs. It is however important to notice that the micro-architecture of PicoRV is completely different and the performance level of Comet is higher, PicoRV being optimized for frequency and area. Indeed, when executing a Dhrystone benchmark, Comet exhibits a Cycles Per Instruction (CPI) of 1.9, whereas the best performing configuration of PicoRV only reaches a CPI of 4.1. Note that the CPI of Comet and Rocket is similar.

TABLE III
RESOURCE USAGE (LUT, FF, MUX, DSP BLOCKS) AND MAXIMAL FREQUENCY FOR DIFFERENT RISC-V CORES SYNTHESIZED TARGETING A XILINX ARTIX 7 FPGA USING CATAPULT HLS AND VIVADO 2018.3.

Core	ISA	freq.	Area			
		(MHz)	LUT	FF	Mux	DSP
Comet	rv32i	80	2 032	1 503	260	0
	rv32im	70	2 910	2 244	227	3
	rv32imf	74	6 460	3 527	448	5
Rocket	rv32i	76	2 253	1 154	41	0
	rv32im		2 570	1 275	43	2
	rv32imf		8 132	3 094	586	4
PicoRV	rv32i	140	880	583	0	0
	rv32im	110	1 977	1 085	0	0

B. Simulator Performance

As already stated, Comet is also a cycle-accurate and bit-accurate simulator of the processor. The simulator contains code that does not aim to be synthesized, which constitutes a wrapper around the core. It allows for the execution of standard elf files, solves systems calls (without the need of additional code within the executed binary), and implements code instrumentation facilities. Consequently, a broad range of bare metal and OS supported applications can be executed by the simulator.

To measure simulation performance, several applications from the MiBench benchmark suite have been selected (the set of workloads has been picked from all the categories of the suite). Simulator performance has been evaluated using different configurations of the target system: support for the base ISA and/or the M and F extensions as well as the presence or not of L1 caches. Applications from the MiBench suite have been executed on the simulator on a host workstation equipped with an 8th-generation Intel Core i7 CPU running at 3.9GHz.

In Table IV, we can observe a peak performance at 23.6 million cycles per second (Mcps) average when the core is set up to only support the base ISA and no caches are used. As expected the performance numbers dip as the number of supported features and system complexity increase. The lowest performance figure of 11.6 Mcps is obtained when the core supports both the M and F extensions and when L1 caches are in use. The average performance of the simulator is around 15 Mcps, when executing floating-point and integer workloads.

Comparatively, the verilator-based [16] simulator of the Rocket chip executes three orders of magnitude slower, with approximately 0.023 Mcps. The widely used Gem5 simulator [17] can simulate around 0.2 Mcps. However, due to the extended control flow introduced by the explicit pipelining needed for HLS use, Comet is not as fast as an ISS. For example, SWERV-ISS [9] can simulate up to 140 million instructions per second. Finally, the fastest way to simulate a RISC-V application is to use dynamic binary translation. For example, Qemu [18] can simulate around 1.3 billion instructions per second, which is close to native performance. Note that ISS and DBT-based simulators are no longer cycle accurate. The performance of Comet (around 10 million instructions per second in average) coupled to the accurate nature of the simulator make it a good candidate for fast micro-architectural design-space exploration.

TABLE IV
SIMULATOR PERFORMANCE AS A FUNCTION OF THE CORE CONFIGURATION. RESULTS ARE GIVEN IN MCPS WHICH STANDS FOR MILLIONS OF SIMULATED CYCLES PER SECOND.

Supported extension	rv32i	rv32im	rv32imf
Simulator perf. w/o L1 (Mcps)	23.6	18.2	15.1
Simulator perf. w/ L1 (Mcps)	16.7	13.0	11.6

C. Example of Core Specialization

Using HLS for processor design offers as another advantage the ease of feature addition. As a demonstration of this extensibility, we have developed a custom instruction performing a Fast Fourier Transform (FFT) butterfly operation to speed-up 64-point, radix 16, FFT computations. To simplify the implementation, the custom instruction can only read two 32-bit registers and write in one hence avoiding the need for a second register file write port and additional forwarding mechanisms. The new instruction can be integrated into the pipeline by simply adding a `case` statement in the ALU of the *Execute*-stage and in the *Decode* stage function.

The custom butterfly instruction receives two 16-bit fixed-point complex numbers represented with two 32-bit values. It performs the butterfly computation using a local array containing the twiddle factors and writes the result in the destination register. This single custom instruction performs four 16-bit multiplications and four 32-bit additions in a single clock cycle.

The custom instruction is added on top of the configuration supporting the `rv32im` ISA. The modification in the C++ model only represents 60 new lines added to the code. The new core is synthesized using a 28 nm technology for an operating frequency of 700 MHz. With the custom instruction, the core is 31% larger than the original Comet, which represents an area of 14 831 μm^2. This area increase is mainly due to the added flipflops and multipliers required by our implementation. An FFT program was also modified to make use of the new custom instruction using the `.instr` assembly directive. The performance measurements demonstrate that the version exploiting the custom instruction executes 14× faster than the straight `rv32im` one. This also shows the ability of our C++ processor model to be easily and fastly extended with new functionalities.

V. Discussion and Related Work

Several attempts at generating processors using HLS have already been made. The objective is usually to generate an application-specific processor where some instructions are removed. In the work of Ahmed *et al.* [19], the processor is not pipelined at all. In the work of Skalicky *et al.* [20], the processor is pipelined using the automatic pipelining capabilities of their HLS tool, which limits the Initiation Interval to a minimum of three or four.

Experimental results show that the Comet core synthesized through HLS presents similar area and performance to other popular HDL-based solutions. In this section, we will discuss some of the advantages and drawbacks of HLS usage to design complex, control-oriented digital circuits such as processor cores.

A. Advantages of the Proposed Design Flow

As previously mentioned, the main advantage of HLS is that a single C++ model is used to generate both a high performance and accurate simulator and a functional hardware component. As C++ is a high-level language, the source is easy to read and is also small, offering great educational and maintainability advantages over standard HDLs. HLS also enables simplified feature additions. As an example, the M ISA extension has been added to the sources and synthesized in a single day. Working with a single model for hardware and software also simplifies the debugging/testing part of the development process. Indeed, all the debugging is done at the C++ level, using dedicated tools like GDB or Valgrind. Once the simulator is thoroughly tested, the hardware is ready to be synthesized and co-simulation can be performed to ensure that the RTL representation behaves the same way than the software model. This design flow is more resilient and less error-prone than conventional hardware development flows.

Thanks to these properties, the Comet core is used to ease architectural exploration of processor core in several research projects:

- Design-space exploration of fault-tolerant CPU designs. The core wrapper code has been modified to include fault injection facilities allowing the injection of errors in any memory point of the design (core and pipeline registers, caches and memories) at any time during the execution of a workload. Since the core code is kept the same, it is still synthesizable and each pipeline stage can be analyzed separately at the RT level or gate level using conventional fault-injection tools. Errors extracted at this low-level can then be re-injected in the simulator with the guidance of some data acquired during synthesis (area of the different pipeline stages, combinational/sequential logic ratios), hence allowing fast and accurate vulnerability analysis. Comet is used as a testbed to characterize several fault-mitigation techniques at different core locations and granularity levels.
- Building a non-volatile processor. The cycle accurate and flexible nature of the Comet simulator easily allows the creation of tracing capabilities and the implementation of uncore peripherals. In the context of normally-off or intermittently-powered computing systems, memory backups need to be precisely scheduled to avoid data loss without having an important impact on power consumption. Based on the memory access traces provided by Comet, the estimation of several characteristics, e.g., the traffic generated by cache misses and fetch operations, is made easier. From these traces, an accelerator for the management of the backup into non-volatile memory can be rapidly designed and simulated.
- The Hybrid-DBT project also made extensive use of HLS based processors [21]. Hybrid-DBT is a system based on Dynamic Binary Translation (DBT), where RISC-V binaries are executed on a VLIW processor. Thanks to a DBT layer, RISC-V binaries are translated into the VLIW explicitly parallel ISA, before being executed. The Hybrid-DBT system is composed of two different types of cores: a small in-order core that translates and optimizes RISC-V binaries and a wider VLIW core that is used to execute the translated binaries. While Comet is used as the DBT processor, the VLIW core is also synthesized using HLS. The same principles are used to design the VLIW: the pipeline is explicitly described in the simulator. However, the VLIW is less complex as it does not use any stall or forward mechanisms. As the whole system is built around C++ models, it is straightforward to build a C++ simulator of the Hybrid-DBT system.

B. Designing a Processor with HLS has Some Limitations

If describing processors using HLS has several advantages it also has some drawbacks, they are mainly introduced because HLS tools are not designed to generate circuits from control-flow dominated specifications. As explained in Section III-A, HLS tools are not capable of generating an efficient pipeline from an ISS-like description of the core, and do not generate hardware below three cycles per instructions. This limitation is due to inter-loop dependencies on the register file and the PC value. In order to get around this limitation, Comet is synthesized from an explicitly-pipelined simulator specification. This type of simulators can be complex to read and write and over-constrain the final design. An ideal solution would be to let the HLS tool generate the pipeline, taking into account the targeted operating frequency and adjusting the pipeline depth accordingly. If the user wants the design to operate at low frequencies, the HLS tool could generate fewer pipeline stages and improve the number of cycles needed to execute a single instruction.

In a hand-made core, pipelining related dependencies are removed using simple but clever mechanisms: stall and forward mechanisms remove the inter-loop dependency on the register file; speculation and pipeline flush mechanisms remove the dependency on the next PC value. As future work, we plan to generalize those mechanisms and work on source-to-source transformations which automatically insert those mechanisms at the C++ level. With these dependencies removed, the HLS tool should be able to pipeline the ISS with an initiation interval of one.

It is also worth noting that standard ASIC fixing techniques, like metal and gate fixes, are hardly applicable to HLS-based designs. HLS tools are not able to ingest a modified gate-level netlist and to reflect the changes back to a C/C++ based representation. It is also hard to tell the HLS compiler to make provisions of cells to create sewing kits as the timing and behavioral information of the kits need to be explicit. If this could be an issue when designing processor chips, it is however a general limitation of any HLS design.

VI. Conclusion

The work done around Comet highlights the benefits of using HLS to develop CPU cores since it significantly reduces development and debugging time. Moreover, this also provides a cycle-accurate and bit-accurate simulator which is, by construction, equivalent to the hardware core. This work demonstrates that HLS tools are mature enough to handle complex, control-dominated code such as pipelined cores. The resulting hardware presents a similar area and operating frequency to HDL written cores.

However, we also observed some limitations with state-of-the-art HLS tools. Indeed, we had to explicitly describe the organization of the pipeline, forwarding logic as well as the multi-cycle operators integration mechanics. We believe that this kind of optimizations could be done automatically by source-to-source transformations. As a future work on this project, we plan to develop more complex micro-architectures (ISA extensions, interrupt controller for OS support, out-of-order processor, shared-memory multiprocessor) to evaluate how far the limits of HLS can be pushed.

Comet is fully open-source and can be found at https://gitlab.inria.fr/srokicki/Comet.

Acknowledgements

The authors thank Valentin Egloff, Edwin Mascarenhas, Lauric Desauw and Gurav Datta for their technical involvement and their contributions in the Comet project. This work was partly funded by the Rapid-DGA Flodam project.

References

[1] E. Martin, O. Sentieys, H. Dubois, and J. L. Philippe, "Gaut: An architectural synthesis tool for dedicated signal processors," in *Proceedings of European Design Automation Conference (EURO-DAC)*, pp. 14–19, Sep. 1993.

[2] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, 2009.

[3] M. Fingeroff, *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010.

[4] T. Bollaert, "Catapult Synthesis: A Practical Introduction to Interactive C Synthesis," in *High-Level Synthesis: From Algorithm to Digital Circuit* (P. Coussy and A. Morawiec, eds.), pp. 29–52, Dordrecht: Springer Netherlands, 2008.

[5] "Catapult C synthesis." https://www.mentor.com/hls-lp/catapult-high-level-synthesis, 2008. Mentor Graphics.

[6] T. M. Feist, "Vivado Design Suite," 2012.

[7] P. Kollig and B. M. Al-Hashimi, "Simultaneous scheduling, allocation and binding in high level synthesis," *Electronics Letters*, vol. 33, pp. 1516–1518, Aug. 1997.

[8] E. Rohou, B. N. Swamy, and A. Seznec, "Branch Prediction and the Performance of Interpreters: Don'T Trust Folklore," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 103–114, IEEE Computer Society, 2015.

[9] J. Rahmeh, "Western Digital's Open Source RISC-V SweRV Instruction Set Simulator." https://github.com/westerndigitalcorporation/swerv-ISS.

[10] A. Waterman and Y. Lee, "Spike, a RISC-V ISA Simulator." https://github.com/riscv/riscv-isa-sim.

[11] N. Moreano, E. Borin, C. de Souza, and G. Araujo, "Efficient datapath merging for partially reconfigurable architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 7, pp. 969–980, 2005.

[12] K. Asanović *et al.*, "The Rocket Chip Generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016.

[13] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in a Scala Embedded Language," in *Proceedings of the IEEE/ACM 49th Annual Design Automation Conference (DAC)*, (New York, NY, USA), pp. 1216–1225, 2012.

[14] S. Rokicki, J. Paturel, D. Pala, E. Mascarenhas, V. Egloff, and O. Sentieys, "Comet: A pipelined RISC-V Processor generated with HLS." https://gitlab.inria.fr/srokicki/Comet.

[15] C. Wolf., "PicoRV32: A Size-Optimized RISC-V CPU." https://github.com/cliffordwolf/picorv32.

[16] W. Snyder, "Verilator and systemperl," in *North American SystemC Users' Group, Design Automation Conference*, 2004.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[18] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), p. 41, USENIX Association, 2005.

[19] T. Ahmed, N. Sakamoto, J. Anderson, and Y. Hara-Azumi, "Synthesizable-from-C Embedded Processor Based on MIPS-ISA and OISC," in *Proceedings of the IEEE 13th International Conference on Embedded and Ubiquitous Computing*, pp. 114–123, Oct 2015.

[20] S. Skalicky, T. Ananthanarayana, S. Lopez, and M. Lukowiak, "Designing customized isa processors using high level synthesis," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–6, Dec 2015.

[21] S. Rokicki, E. Rohou, and S. Derrien, "Hybrid-DBT: Hardware/Software Dynamic Binary Translation Targeting VLIW," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–14, 2018.