HL5: A 32-bit RISC-V Processor Designed with High-Level Synthesis

Paolo Mantovani, Robert Margelli, Davide Giri and Luca P. Carloni
Department of Computer Science · Columbia University, New York, NY 10027
[paolo, margelli, davide_giri, luca]@cs.columbia.edu

***Abstract*—The growing complexity of system-on-chip fuels the adoption of high-level synthesis (HLS) to reduce the design time of application-specific accelerators. General-purpose processors, however, are still designed using RTL and logic synthesis. Yet they are the most complex components of most systems-on-chip. We show that HLS can simplify the design of processors while enhancing their customization and reusability. We present HL5 as the first 32-bit RISC-V microprocessor designed with SystemC and optimized with a commercial HLS tool. We evalute HL5 through the execution of software programs on an experimental infrastructure that combines FPGA emulation with a standard RTL synthesis flow for a commercial 32 nm CMOS technology. By describing the challenges and opportunities of applying HLS to processor design, our paper aims also at sparking a renewed interest in HLS research.**

***Index Terms*—High Level Synthesis; RISC-V; Processor Pipeline.**

I. Introduction

The worldwide market for semiconductor *intellectual property (IP) blocks* is expected to approach 8 billion by 2019 [1]. A growing variety of these IP blocks populates any system-on-chip (SoC) for embedded systems or Internet-of-Things (IoT) because SoC architects must find the right mix of components for the target application domain, while being pressured by stringent time-to-market constraints.

To cope with design complexity it is necessary to raise the level of abstraction by embracing *system-level design* methods [2], [3]. These include the use of high-level programming languages, like C/C++ and SystemC, for design specification and the application of *high-level synthesis (HLS)* for design optimization [4]. The benefits include: (1) the ability to complete a richer design-space exploration (DSE); (2) larger portability to various technology platforms, including FPGAs; (3) and a broader reusability across different target systems, as each IP-block implementation offers a different trade-off option between performance and cost (i.e. power or area).

Over the past decade, HLS has been used for thousands of ASIC tapeouts and FPGA designs [4], particularly to optimize IP blocks for wireless communication [5] and video decoding/encoding [6]. Meanwhile, researchers have demonstrated the application of HLS to many other IP blocks such as: accelerators for database analytics [7] and machine learning [8], the on-chip memory subsystems [9], [10], the memory hierarchy [11] and the on-chip interconnect, including networks-on-chip and interfaces for standard bus protocols [12], [13]. Moreover, HLS has been proposed as a key ingredient for new methodologies that target the problem of heterogeneous component integration in SoCs [12], [14]–[16].

In this landscape, one particular IP block stands out as a major exception: the processor core.

As discussed in more detail in Section V, the literature offers very few examples of papers investigating HLS for processor design. Commercial HLS tools are very efficient in synthesizing computationally intensive datapaths, when the amount of control logic is limited. However, despite recent advances [17], [18], they still struggle to produce high-quality RTL for control-dominated branching logic [19]. These limitations may have prevented researchers from investing time and resources in applying HLS to processor design. Yet processor cores are the most complex components of most SoCs, with the biggest impact on design and verification costs [20]. Furthermore, as technology trends push designers towards even more heterogeneity and customization [21], [22], processor designs must become more configurable so that they can be optimized for each particular target SoC where they can be instanced. These considerations motivated our work.

Contributions. We present HL5 as the *first 32-bit processor fully designed and implemented with HLS*. HL5 is a pipelined processor that implements the full `RV32IM` subset of the RISC-V instruction set architecture (ISA). The RISC-V ISA, originally developed at UC Berkeley [23] and now supported by the RISC-V Foundation [24], has raised broad interest across academia and industry thanks to the appealing promise of representing a free and open ISA that could become an industry standard for a variety of systems, from IoT devices to datacenter servers [25].

We designed HL5 with the goal of leveraging HLS to maximize its configurability and minimize design costs. We derived a concurrent specification for the HL5 design that is entirely based on the synthesizable subset of SystemC. This required us to circumvent some limitations of current HLS tools through clever design choices applied to the high-level specification, as explained in Section II.

In Section III, we dive into the details of our design. The latency of the functional units of HL5 can be varied based on the HLS configuration parameters (aka *HLS-knob settings* or *knobs*). By using latency-insensitive channels [3], [26] as the main communication mechanism across the processor stages, the pipeline of HL5 can tolerate latency variations of the caches, the register file, the functional units and the branch

Code snippet 1:

```
typedef struct instruction {
  raddr_t rs1, rs2, rd;
  func_t func;
  ctrl_t ctrl;
} instruction_t;

// Body of the main SystemC SC_CTHREAD
while(true)
{
  HLS_PIPELINE_LOOP;

  wait(); // Wait on virtual clock edge

  pc = npc;
  instruction_t inst = decode(imem[pc]);
  op1 = regfile[rs1]; op2 = regfile[rs2];
  npc = next_pc(pc, inst, op1, op2);

  if (check_for_hazards(inst)) { // Stall
    npc = pc; continue;
  }
  if (inst.ctrl.jump) {
    if (npc == pc) break; // End of program
    else continue; // Jump or branch taken
  }
  out = execute(pc, inst, op1, op2);

  if (inst.ctrl.store)          dmem[out]   = op2;
  else if (inst.ctrl.load)      regfile[rd] = dmem[out];
  else if (inst.ctrl.writereg) regfile[rd] = out;
}
```

Figure 1. Naive SystemC description of a pipeline.

logic. These variations can be the results of both customization decisions at design time (through HLS knobs) and their impact on the data/instruction flow when executing a given program at run time. We used Cadence Stratus, a commercial HLS tool, to process the SystemC specification and synthesize 12 RTL implementations, with clock frequencies ranging from 700 MHz up to 2 GHz.

In Section IV, we describe our experimental infrastructure to evaluate HL5 with the execution of actual software programs: this combines FPGA emulation with a standard RTL synthesis flow for a commercial 32 nm CMOS technology. We compare our synthesized RTL implementations of HL5 against ZERO-RISCY, a processor core for IoT that is part of the PULP platform [27] and that was carefully optimized for area occupation through manual RTL design [28]. Through the design of HL5, we illustrate a set of guidelines to design processor pipelines in SystemC. Furthermore, we consider the limitations of this approach, which can drive future improvements of HLS tools.

II. HLS for Processor Design

In this section, we discuss how to address the current limitations of HLS for processor design. We show that HLS can actually reduce the effort to design the pipeline of a processor compared to hand-written RTL, while still achieving good quality of results.

First, let us consider the snippet of code in Fig. 1, which shows the body of a SystemC process of type `SC_CTHREAD` that specifies a simple processor pipeline. In the SystemC simulation engine, the `SC_CTHREAD` processes are concurrent threads that are triggered for execution at every edge of a clock signal. This is often called a *virtual* clock because a state transition in SystemC, controlled by its virtual clock, may correspond to many state transitions in the RTL circuit. For example, an invocation of function `execute()` in Fig. 1 does not cause the SystemC simulation time to advance: the loop body processes one full instruction consuming zero simulation time, then suspends the execution of the `SC_CTHREAD` until the next virtual-clock edge by calling the `wait()` function. The RTL implementation, instead, will consume at least one physical-clock cycle to execute the logic synthesized for the `execute()` function, even in the case of simple instructions. In general, the virtual-clock abstraction allows a designer to simply specify the hardware for complex computations in a loosely-timed fashion, while letting HLS schedule these computations across physical-clock cycles and infer the necessary finite-state machines (FSM) to control them.

Wouldn't it be great if we could synthesize the pipeline of a complete processor from this simple SystemC *specifications?*

Indeed, we can! State-of-the-art HLS tools can generate a working RTL implementation from the SystemC specification of Fig. 1. To the best of our knowledge, however, from this naive specification no HLS tool is currently capable of achieving a throughput of one *instruction-per-cycle (IPC)*, which is the ideal IPC for single-issue processors in absence of dependencies across instructions. Despite its simplicity, this code snippet captures most of the key challenges that HLS faces when the target design is a complex pipeline such as a processor pipeline.

Loop-Carried Dependencies. The ISA is an abstraction layer that exposes the *processor state* (i.e., the content of the program counter, the register file, and the memory) to the software while hiding the details of the hardware implementation (e.g., the content of the pipeline registers in a pipelined implementation). Indeed, the ISA is like a "contract" that allows many different pipelined hardware implementations as long as they can run software programs *as if* each instruction is executed atomically before the next one starts. The specification of Fig. 1 reflects this abstraction as each loop iteration may change the value of the processor state in a way that impacts the next iterations. Since a pipeline implementation overlaps the partial execution of instructions, it requires a careful handing of control and data dependencies to avoid possible *hazards*. These dependencies may translate into *loop-carried dependencies (LCDs)* that pose significant challenge to any HLS tool.

Control Feedback. For example, with no control hazard the computation of the next value `npc` of the program counter requires just a simple addition that has typically a delay smaller than the physical-clock period. However, if a control hazard is present because the current instruction is a branch, then its resolution typically takes multiple clock cycles in any pipelined implementation. Hence, without any additional information from the designer, a valid scheduling cannot allow an initiation interval equal to one for a pipeline that implements the specification of Fig. 1.

Similar considerations apply for data dependencies and, indeed, for any modification of a state variable in the body

of a SystemC loop. For example, since the instruction flow cannot be known at design time, the HLS tool must expect that an entry in the register file (or in the data memory) that is written by an instruction at a given iteration could be read by another as early as in the next iteration. Hence, to synthesize a classic 5-stage pipeline without any data hazard, the HLS tool may force the execution of just one instruction every 5 cycles, thus yielding a poor $IPC = 0.2$. Note that checking for hazards in the body of the loop prevents that an instruction with invalid operands commits data into the register file and the data memory. When the HLS tool analyzes the code, however, this check does not eliminate intrinsic dependencies across loop iterations due to array accesses.

Memory Access. In SystemC, memories are naturally specified with arrays. To map an array to a given memory block, a HLS tools transforms a simple array indexing into a bundle of logic for address generation and read/write enable signals. While accessing an array in SYSTEMC incurs a zero-time penalty, most memories have latency of one or more clock cycles, each imposing the injection of at least one state into the synthesized RTL to schedule the memory access operation. Combining such latency constraints with LCDs further complicates the synthesis of pipelined implementations.

Given the above limitations, we now explain how to write a HLS-friendly specification of a pipeline in SYSTEMC.

Breaking Dependencies with Concurrency. First, we must prevent HLS from implicitly handling data and control dependencies. Many hazards are *read-after-write (RAW)* hazards, which correspond to real data dependencies. Hence, forcing the HLS tool to ignore them during scheduling requires to model the pipeline stages with multiple `SC_CTHREAD` processes, such that no `SC_CTHREAD` is both reading from and writing to the register file (or the data memory). Fig. 2 shows a partial specification based on this idea: e.g., Stage 3 may only access the register file with a write operation, whereas Stage 1 always accesses it with a read operation and Stage 2 operates without accessing it. In this way, dependencies only exist across concurrent `SC_CTHREAD` processes so that the HLS tool does not have to abide by the C++ semantics to handle them conservatively. Instead, they are handled explicitly by the `check_for_hazards()` function.

Distributed Pipeline Control. The next step is to introduce a flow-control mechanism ensuring that each stage only processes and retires valid data and instructions. We do so through synchronization primitives across the stages that may block the caller whenever there is no valid data to process. For instance, Stage 2 in Fig. 2 calls `wait_for_stage_1()`, which suspends the execution of the corresponding `SC_CTHREAD` until Stage 1 calls `signal_stage_2()`. The latter is called when a valid instruction is decoded and all operands are available. These synchronization functions use latency-insensitive protocols [3], [26], [29] and apply the principles of *transaction-level modeling (TLM)* [30], which advocates a separation between computation and communication. As a result, the composition of the three processes is correct-by-construction, independently from the latency and the throughput of the logic

Code snippet 2:

```
// Stage 1
while(true)
{
  HLS_CONSTRAINT_LATENCY;
  pc = npc;
  instruction_t inst = decode(imem[pc]);

  if (check_for_hazards(inst)) { // Stall
    wait_for_stage_3(); // May block.
  }

  op1 = regfile[rs1]; op2 = regfile[rs2];
  npc = next_pc(pc, inst, op1, op2);

  if (inst.ctrl.jump) {
    if (npc == pc) break; // End of program
    else continue; // Jump or branch taken
  }
  signal_stage_2();
}

// Stage 2
while (true) {
  HLS_PIPELINE_LOOP;
  wait_for_stage_1(); // May block
  out = execute(pc, inst, op1, op2);
  signal_stage_3();
}


// Stage 3
while (true) {
  HLS_CONSTRAINT_LATENCY;
  wait_for_stage_2(); // May block
  if (inst.ctrl.store)          dmem[out]    = op2;
  else if (inst.ctrl.load)      regfile[rd] = dmem[out];
  else if (inst.ctrl.writereg) regfile[rd] = out;
  signal_stage_1();
}
```

Figure 2. HLS-friendly specification of a pipeline.

synthesized from each `SC_CTHREAD`. This enables a richer DSE by allowing an independent optimization of each pipeline stage with HLS.

TLM Channels. While the partial SystemC specification in Fig. 2 shows generic synchronization primitives across the processes, HLS tools offer libraries of *point-to-point (p2p) channels* based on TLM. Usually these channels can be customized to be blocking, non-blocking, or conditionally-blocking. In order to implement a pipeline across processes, the best choice is to use conditionally-blocking channels, which guarantee maximum throughput when both the sending and the receiving processes call the synchronization primitive. A conditionally-blocking channel, on the other hand, enforces correctness by preventing the receiving process to advance if no valid data is present on the channel. Note that non-blocking channels would achieve similar performance but require that the designer implements all the necessary checks on the presence of valid data. While these checks complicate the design, non-blocking channels are useful when implementing portions of the pipeline that may execute "out-of-order". In fact, in order to prevent deadlock when the order of execution is not predetermined, the designer must be able to check the state of each channel without blocking the pipeline control logic. This behavior can be obtained only with non-blocking

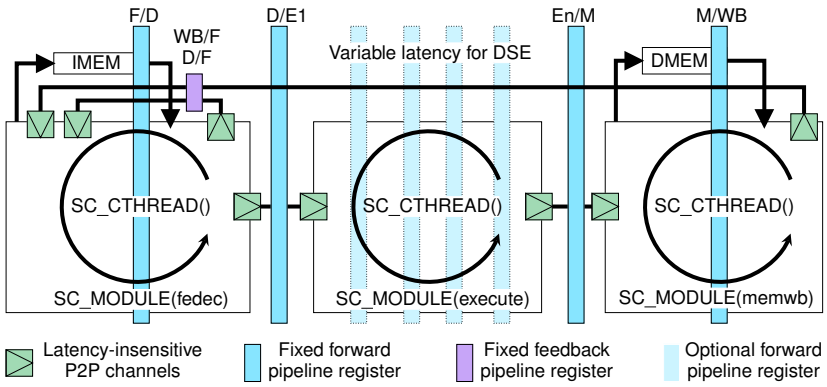

Figure 3. HL5 pipeline: SystemC specification.

channels.

The implementation of conditionally-blocking channels depends on the particular HLS tool. The most common behavior consists in letting the sending process advance for one or a few iterations and then suspend it until the receiving process is ready to consume the data. The number of iterations allowed before blocking determines the amount of storage necessary to save the information on the channel and can be considered a HLS knob to explore trade-offs between area and throughput.

Remarks. While the specification of Fig. 1 cannot be synthesized with good quality-of-results, Fig. 2 shows how minor code changes can circumvent the limitations of HLS. With these guidelines, in the span of two months one master student was able to complete the initial design of HL5 and use HLS to synthesize several pipelined implementations, each characterized by a particular trade-off point between area and performance. In just two more hours another student managed to introduce *data forwarding* across stages to improve the overall IPC. The experience of designing HL5 allowed us to appreciate the advantage of debugging a complex IP block at a level where the ISA simulator corresponds to the design entry point for synthesis, thus eliminating the risk of injecting bugs while manually translating the initial specification into RTL. Furthermore, since the control logic of each stage is abstracted away from the specification, extending the ISA of HL5 with any standard or custom instruction can be done without changing the baseline design. The latency-insensitive p2p channels across the stages, in fact, create a flexible design, that is tolerant to variations in latency and throughput both at design time and run time in every stage of the pipeline.

III. HL5 Processors Design

Fig. 3 illustrates the structure of the synthesizable SystemC specification of the HL5 pipeline, which consists of three main stages: `fedec`, `execute`, and `memwb`. Each stage is modeled with an `SC_CTHREAD` process as part of an `SC_MODULE` and communication is implemented through latency-insensitive p2p channels and read or write initiators (i.e. `get()` and `put()`). The `execute` stage is the target of most DSE procedures to optimize the structure of the operations it implements: e.g. addition, subtraction, multiplication, division. As shown in Fig. 4, each process is structured with two main sections: a reset section where initialization and configuration steps are performed, and an infinite loop where communication and computation occur. Within this loop, the stage acquires new data from the previous stage, performs some computation, and transfers the processed information to the next stage.

Each `wait()` statement corresponds to a virtual-clock boundary. The code in a region delimited by two consecutive `wait()` statements is typically specified as *untimed* logic, i.e. no constraints are imposed on the HLS tool with respect to the actual timing of the hardware that must be synthesized to implement this logic. In general, the HLS tools can be left free to decide how many physical-clock cycles to use for implementing this logic (which can be seen as implicitly adding/removing `wait()` statements during synthesis until the physical-clock boundaries coincide with the virtual-clock ones.) These synthesis decisions can be influenced by setting a variety of HLS knobs, such as loop unrolling or pipelining, to guide the HLS tools towards synthesizing a particular microarchitecture with a specific trade-off in terms of performance versus area occupation. In general, depending on the specified settings of the HLS knobs, the synthesized pipeline may have a different number of stages, which doesn't necessarily correspond to the three `SC_MODULE`s.

Some HLS directives can be used to constrain more the HLS tool on the desired timing characteristics of a circuit. Among these, the `PROTOCOL_REGION()` directive forces a code region to be interpreted as *cycle accurate*: this means that the HLS tool does not add or prune any `wait()` statement while synthesizing hardware for this region, but, instead, interprets those that are present as physical-clock boundaries. In particular, the p2p-channel primitives provide a transparent abstraction for optimized low-level communication protocols that are implemented with protocol regions. *In summary, the combination of latency-insensitive p2p channels and* `SC_CTHREAD` *processes provides a clear separation of computation and communication in a compositional way that makes DSE more effective.*

As an example, the listing in Fig. 5 presents the data structures and latency-insensitive p2p interfaces for one module.

Code snippet 3:

```
void pipeline_stage_cthread() {
  {
    PROTOCOL_REGION("reset");
    from_previous_stage_if.reset_get();
    to_next_stage_if.reset_put();
    // ... state initialization.
    wait();
  }
  while(true) {
    {
      PROTOCOL_REGION("input");
      din = from_previous_stage_if.get();
    }
    dout = compute(din); // Relax latency for DSE
    {
      PROTOCOL_REGION("output");
      to_next_stage_if.put(dout);
      wait();
    }
}
```

Figure 4. Structure of a pipeline stage process.

Code snippet 4:

```
/* hl5_datatypes.h */
typedef struct exe2memwb_s {
  // ... } exe2memwb_t;

typedef struct memwb2fedec_s{
  sc_bv <1> regwrite;
  sc_bv <5> regfile_address;
  sc_bv <32> regfile_data;
} memwb2fedec_t;

/* memwb.h */

  // LIC interfaces
  LIC_get_if<exe2memwb_t> memwb_get_if;
  LIC_put_if<memwb2fedec_t> memwb_put_if;

  // LIC data structures
  exe2memwb_t memwb_din;
  memwbMa2fedec_t memwb_dout;
```

Figure 5. Definition of the data structure and p2p interfaces for the `memwb` stage.

The members of each `struct` are of type `sc_bv`, which models a single wire or a bundle (the fields of the `exe2memwb_t` structure are omitted in the reported listing). Whenever the values on these wires must be used for computation, it is possible to cast them to other types (such as `sc_int`) that support the C++ arithmetic and logic operators.

The use of multiple `SC_MODULE`s was initially intended for simply keeping the design modular and thus easier to manage. However, it has an additional advantage over specifying the three threads within a single module: HLS preserves the hierarchy, including signals across the stages in this case, thus these can still be observed when running RTL simulation.

Fedec Stage. Fetch and decode are the first two stages of many processor pipelines and are typically separated by pipeline registers. For the design of HL5, however, we combined them into a single stage due to the HLS limitations discussed in Section II. In particular, since scheduling an instruction-memory access requires at least one cycle and the communication across two `SC_CTHREAD` processes consumes another cycle, splitting fetch and decode would cause an undesirable lower-bound of three cycles for just fetch and decode.

While the HLS tools automatically map the instruction memory to a static RAM, the register file is modeled in SystemC as a simple non-shared array of type `sc_bv`. The operations of reading/writing its content both occur at the beginning of `fedec`, thus eliminating the problem of loop-carried dependencies, and allowing for an initiation interval of one. Besides instruction fetching, `fedec` may also get input data from `memwb` through a feedback path. Hence, at any given iteration of its main loop it decodes the instruction while accessing the register file for reading and/or writing operands, before propagating its output to the `execute` stage.

Execute Stage. The `execute` stage is the core stage of the pipeline, where arithmetic and logical operations are performed. The main loop of its SystemC specification contains a large `switch` statement to select the operation that must be performed on the operands. Generally, the first operand is statically mapped to the first register `RS1`, while the second operand may be mapped to register `RS2` or consists of the immediate field of the instruction. All operations, except for the C++ operators `/` (division) and `%` (modulo), are synthesizable by the HLS tool we used. We implemented an optimized algorithm for the division that we encapsulated in a separate function, which is called within the `switch` statement. The 32-bit division algorithm supports the execution of the `div`, `divu`, `rem` and `remu` instructions from the RISC-V ISA. Note that any ISA extension can be similarly implemented with a simple function call.

The main loop is specified without using `PROTOCOL_REGION()` to give maximum freedom to the HLS tool while we performed our DSE to obtain many alternative microarchitectural implementations by applying HLS knobs. In particular:

(1) loop unrolling is applied to increase hardware parallelism. For instance, the hardware resources necessary to implement the divisor can be replicated multiple times in order to reduce the division latency from 32 clock cycles down to 16, 8, or 4. In traditional RTL synthesis, there is no control of such kind and loops are always completely unrolled. In this case, a division which by the definition of the algorithm takes 32 clock cycles (CC), may be transformed into different implementations which may take as little as a few clock cycles to perform the operation. On the down-side the replication of hardware yields a larger area occupation.

(2) loop pipelining is applied to raise throughput while keeping the possibility of sharing most resources of multi-cycle units, thus reducing area occupation. While loop-carried dependencies prevent this option to be applied to the division, it can be used to improve a multi-cycle version of the multiplier and to automatically implement multiple pipeline stages within the `execute` phase.

(3) tool-specific synthesis directives enable fine tuning of the scheduling by requiring more aggressive synthesis approaches, such as scheduling operations as-soon-as-possible, or extract-

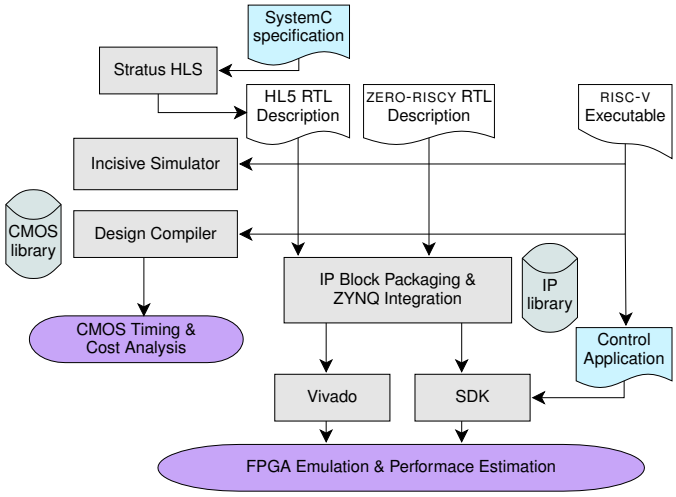

Figure 6. Implementation and evaluation flow.

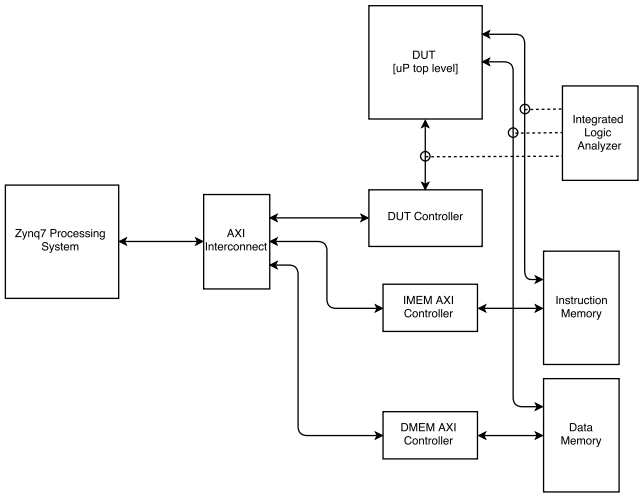

Figure 7. IP block system for FPGA emulation.

ing portions of the logic into separate FSMs that are concurrent with respect to the rest of the circuit.

Memwb Stage. To avoid the same issue discussed for the `fedec` stage, we combined memory and write-back into a single `memwb` stage. This accesses data memory for load/store operations and retires completed instructions while marking accordingly those that update an entry in the register file.

IV. Design Evaluation

We synthesized the SystemC specification of HL5 with the commercial STRATUS HLS tool from Cadence. Through various combinations of the HLS knobs, we obtained 12 different RTL implementations of the HL5 pipeline, each corresponding to a different microarchitecture. We compared these implementations with ZERO-RISCY, a processor core for IoT that is part of the PULP platform [27], [28]. Both processors implement the full `RV32IM` subset of the RISC-V ISA [24].

Fig. 6 shows the CAD flow we used to implement and evaluate both HL5 and ZERO-RISCY: the only difference is in the initial steps for HL5 that consist in simulating the SystemC program to check functional correctness with respect to the ISA and in synthesizing the RTL with Stratus HLS. The RTL implementation is validated again via RTL simulation (for ZERO-RISCY this is the starting point) before going through logic synthesis for FPGA. Estimates on area occupation and maximum clock frequency are also obtained for each implementation with Synopsys Design Compiler using a commercial 32 nm CMOS technology. The performance of each implementation is then evaluated by running on the FPGA bare-metal applications compiled from C software. In particular, each implementation is packaged as an IP block within the Xilinx Vivado environment and integrated "as a client processor" with the dual-core ARM processor on the ZYNQ SoC FPGA. Fig. 7 illustrates the ZYNQ system instantiating either HL5 or ZERO-RISCY as the device-under-test. Through the AXI interconnect, we added two SRAM modules that serve as instruction and data memories for both HL5 and ZERO-RISCY. The ARM core executes a control application to load these memories, start the target processor, and monitor its execution. A custom core controller, also implemented with HLS, interfaces the control application with the processor under test. Additionally, this module raises an interrupt to the ARM processor when the execution on the target core is completed and returns the value of a performance counter, corresponding to the number of clock cycles taken by the program execution. Notice that every HL5 implementation can be seamlessly integrated into the ZYNQ processing system without any edits to the SYSTEMC code, or to the control application.

Area-Performance Analysis. Fig. 8 reports the performance of four Pareto-optimal implementations of HL5 normalized against ZERO-RISCY, when executing eight popular benchmarks: DHRYSTONE, the HISTOGRAM-EQUALIZATION, AES256, MATRIX MULTIPLICATION, a division-intensive synthetic benchmark, fixed-point FFT, CONVOLUTION, and 2D-CONVOLUTION. The four HL5 implementations are labeled based on the chosen HLS-knob settings. For the "Basic" implementation, we set only default HLS knobs; for the "ASAP" implementation, we force the HLS tool to schedule operations as soon as possible, thus trading off some opportunities for resource sharing; finally, for the implementations labeled "DIV2" and "DIV4", we leverage loop unrolling to speed up sequential units, and in particular the divider. Each bar with a value above one corresponds to a speedup, while each of the others corresponds to a slow-down. The speedup is evaluated as the ratio between the effective latency of ZERO-RISCY and the effective latency of HL5. This metric is computed as the product of the cycle count, measured through the FPGA emulation, and the achievable clock period. When considering performance, HL5 implementations are on average comparable with respect to ZERO-RISCY with a speedup that varies across benchmarks [1].

[1] HL5 does not support the custom instructions of ZERO-RISCY.

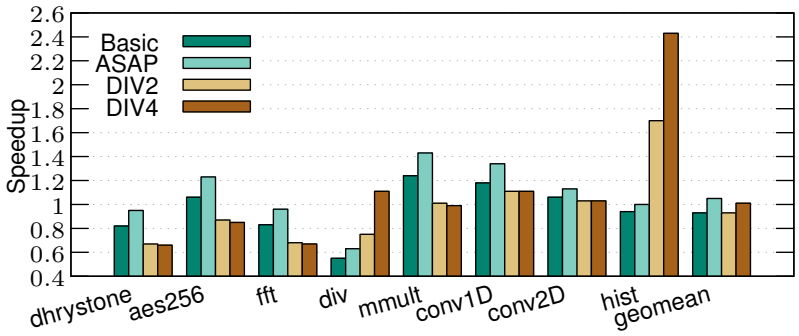

Figure 8. Performance analysis: HL5 vs ZERO-RISCY.

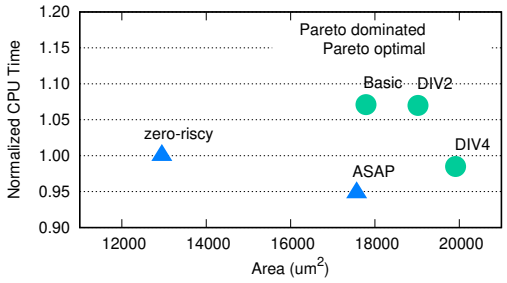

Figure 9. Normalized CPU time vs. area.

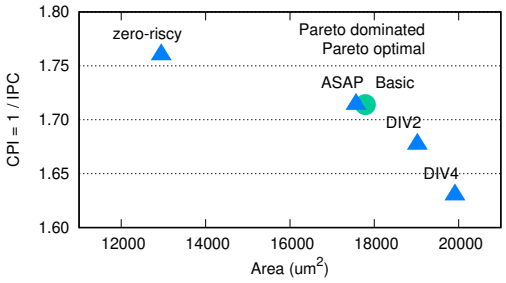

Figure 10. Clock-per-instruction (1/IPC) vs. area.

HISTOGRAM-EQUALIZATION shows most of the benefits of applying HLS to automatically trade off resources and clock period for spatial computation through loop unrolling. Even though this advantage is clearly application-dependent, in the context of embedded systems and IoT, the mix of applications is typically well characterized. Therefore, designers can select a microarchitecture that performs best with the target workload without editing the source code of the processor.

Fig. 9 shows the area-speedup trade offs in a bi-objective space where normalized latency is the y axis and area occupation is the x axis. Fig. 10, instead, shows a different bi-objective space where latency is replaced with clock-per-instruction (1/IPC). This metric is relevant because HLS tools are currently too pessimistic when calculating the critical path, thus penalizing some microarchitectural choices which fail the scheduling step if the target clock period is too stringent. These results show that HL5 footprint is between 35% and 50% larger than ZERO-RISCY, which was carefully optimized for area occupation through manual RTL design. On the other hand, HLS allows us to automatically generate multiple implementations of HL5 which have comparable performance and improved IPC with respect to ZERO-RISCY.

Lines of code (LOC). It should be noted that the effort and time required by the proposed design activity is notably less than that involved into a traditional RTL design flow. These aspects are usually hard to measure. We report the number of lines of code (LOC), which is a commonly used metric for estimating design effort. The SystemC LOC for HL5 is about 2k, while the RTL of ZERO-RISCY consists of more than 6k LOC.

V. Related Work

SystemC has been used for modeling processors, but there are no papers on its application to design them with HLS and synthesize an implementation for FPGA or ASIC flows.

The work by Huang and Despain [31] is the first in a series of papers that propose processor-specific HLS tools for both pipeline optimization and compiler generation [32], [33]. In contrast, we use a commercial general-purpose HLS tool, since specialized ones have never reached a market large enough to enable hardware development beyond the research stage.

Skalicky et al. have proposed using a commercial HLS to improve the performance of a customizable MIPS-like processor [34]. While they only comment about how pipeline hazards reduce performance, their approach severely hinders the achievable throughput, as it is based on a purely sequential specification. In contrast, we start from a concurrent model of the processor and achieve better performance because pipeline hazards are handled up front.

Along different research lines, some researchers started from a domain-specific model of a processor pipeline and automatically performed transformations such as bypass and speculation to increase its throughput [35]–[37]. Some of these transformations (e.g. speculation) can be automatically performed by the commercial HLS tool we used, while others (e.g. register-file or memory bypass) could represent interesting enhancements to any HLS tool.

Finally, several implementations of RISC-V have been made in Chisel [38], a Scala-embedded language that allows functional and object-oriented descriptions of hardware circuits. Chisel, however, *"more closely resembles traditional hardware description languages like Verilog than high-level synthesis systems"*, as recognized by some of its developers [39].

VI. Concluding Remarks

We presented HL5, which supports the `RV32IM` subset of the RISC-V ISA, as the first 32-bit processor fully designed and implemented using SystemC and HLS. We showed how the HLS flow can be applied to realize processor pipelines with performance comparable to that of a manually-optimized RTL implementation. Despite the limitations of current HLS tools, the effort to design and optimize HL5 was significantly smaller compared to traditional RTL flows. Addressing these limitations represents a research driver for future HLS tools. We plan to to release HL5 in public domain to serve as a initial template for the design of future, more complex processors with HLS.

Acknowledgments. This work is supported in part by the National Science Foundation `A#: 1219001 and 1527821` and DARPA DECADES `C#: FA8650-18-2-7862`. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, eitherexpressed or implied, of Air Force Research Laboratory and DARPA or the U.S. Government.

References

[1] S. R. Corp., "Licensing, royalty and service revenues for 3rdParty SIP: Market analysis and forecast for 2015."

[2] A. Sangiovanni-Vincentelli, "Quo vadis SLD: Reasoning about trends and challenges of system-level design," *Proc. of the IEEE*, vol. 95, no. 3, pp. 467–506, 2007.

[3] L. P. Carloni, "From latency-insensitive design to communication-based system-level design," *Proc. of the IEEE*, vol. 103, no. 11, pp. 2133–2151, Nov. 2015.

[4] A. Takach, "High-level synthesis: Status, trends, and future directions," *IEEE Design & Test of Comp.*, vol. 33, no. 3, pp. 116–124, 2016.

[5] S. Mhaske, H. Kee, T. Ly, and P. Spasojevic, "FPGA-accelerated simulation of a hybrid-ARQ system using high level synthesis," in *IEEE 37th Sarnoff Symposium*, Sep. 2016, pp. 19–21.

[6] P. Sjovall, J. Virtanen, J. Vanne, and T. D. Hamalainen, "High-level synthesis design flow for HEVC intra encoder on SoC-FPGA," in *Euromicro Conf. on Digital System Design*, Aug. 2015, pp. 49–56.

[7] G. A. Malazgirt, N. Sonmez, A. Yurdakul, A. Cristal, and O. Unsal, "High level synthesis based hardware accelerator design for processing SQL queries," in *Proc. of the 12th FPGAworld Conference*, Sep. 2015, pp. 27–32.

[8] E. D. Sozzo, A. Solazzo, A. Miele, and M. D. Santambrogio, "On the automation of high level synthesis of convolutional neural networks," in *Proc. of International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 217–224.

[9] C. Pilato, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "System-level optimization of accelerator local memory for heterogeneous systems-on-chip," *IEEE Trans. on CAD*, vol. 36, no. 3, pp. 435–448, Mar. 2017.

[10] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based Data Reuse Optimization for Configurable Computing," in *Proc. of Symp. on FPGA*, Jan. 2013, pp. 29–38.

[11] J. Cong, P. Zhang, and Y. Zou, "Optimizing memory hierarchy allocation with loop transformations for high-level synthesis," in *Proc. of DAC*, Jun. 2012, pp. 1233–1238.

[12] V. Kathail, J. Hwang, W. Sun, Y. Chobe, T. Shui, and J. Carrillo, "SDSoC: A higher-level programming environment for Zynq SoC and Ultrascale+ MPSoC," in *Proc. of Symp. on FPGA*, 2016, pp. 4–4.

[13] P. Parakh, D. Mullassery, A. Chandrashekar, H. Koc, D. Dal, and N. Mansouri, "Interconnect-centric high level synthesis for enhanced layouts with reduced wire length," in *Intl. Midwest Symp. on Circuits and Systems*, Aug 2006, pp. 595–600.

[14] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski, S. Brown, and J. Anderson, "From software to accelerators with LegUp high-level synthesis," in *Proc. of CASES*, 2013, pp. 18:1–18:9.

[15] L. P. Carloni, "Invited - the case for embedded scalable platforms," in *Proc. of DAC*, 2016, pp. 17:1–17:6.

[16] Y.-T. Chen *et al.*, "Accelerator-rich CMPs: From concept to real hardware," in *Proc. of ICCD*, Oct. 2013, pp. 169–176.

[17] D. Pursley and T. H. Yeh, "High-level low-power system design optimization," in *Proc. of VLSI-DAT*, April 2017, pp. 1–4.

[18] A. Kondratyev, L. Lavagno, M. Meyer, and Y. Watanabe, "Realistic performance-constrained pipelining in high-level synthesis," in *Proc. of DATE*, Mar. 2011, pp. 1–6.

[19] 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.

[20] T. Kam, S. Rawat, D. Kirkpatrick, R. Roy, G. Spirakis, N. Sherwani, and C. Peterson, "EDA challenges facing future microprocessor design," *IEEE Trans. on CAD*, vol. 19, no. 12, pp. 1498–1506, Dec. 2000.

[21] S. Borkar and A. A. Chien, "The future of microprocessors," *Communication of the ACM*, vol. 54, pp. 67–77, May 2011.

[22] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *ISSCC*, Feb. 2014, pp. 10–14.

[23] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," EECS Dept., UC Berkeley, Tech. Rep. UCB/EECS-2014-146, Aug. 2014.

[24] "The RISC-V foundation," https://riscv.org/.

[25] D. Kanter, "RISC-V offers simple, modular ISA," in *Microprocessor Report*, Mar. 2016.

[26] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli, "A methodology for "correct-by-construction" latency insensitive design," in *Proc. of ICCAD*, Nov. 1999, pp. 309–315.

[27] "The PULP platform," https://pulp-platform.org.

[28] P. D. Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race?" in *Proc. of PATMOS*, Sept 2017, pp. 1–8.

[29] L. P. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Trans. on CAD*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001.

[30] F. Ghenassia, *Transaction-Level Modeling with SystemC*. Springer, 2010.

[31] I. J. Huang and A. M. Despain, "High level synthesis of pipelined instruction set processors and back-end compilers," in *Proc. of DAC*, Jun. 1992, pp. 135–140.

[32] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai, "PEAS-III: an ASIP design environment," in *Proc. of ICCD*, Sep. 2000, pp. 430–436.

[33] O. Schliebusch, A. Chattopadhyay, R. Leupers, G. Ascheid, H. Meyr, M. Steinert, G. Braun, and A. Nohl, "RTL processor synthesis for architecture exploration and implementation," in *Proc. of DATE*, Feb. 2004, pp. 156–160.

[34] S. Skalicky, T. Ananthanarayana, S. Lopez, and M. Lukowiak, "Designing customized ISA processors using high level synthesis," in *Proc. of ReConFig*, Dec. 2015.

[35] S. Gupta, T. Kam, M. Kishinevsky, S. Rotem, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau, "Coordinated transformations for high-level synthesis of high performance microprocessor blocks," in *Proc. of DAC*, 2002, pp. 898–903.

[36] T. Kam, M. Kishinevsky, J. Cortadella, and M. Galceran-Oms, "Correct-by-construction microarchitectural pipelining," in *Proc. of ICCAD*, Nov. 2008, pp. 434–441.

[37] E. Nurvitadhi, J. C. Hoe, T. Kam, and S. L. L. Lu, "Automatic pipelining from transactional datapath specifications," in *Proc. of DATE*, Mar. 2010.

[38] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *Proc. of DAC*, June 2012, pp. 1212–1221.

[39] K. Asanovic *et al.*, "The Rocket chip generator," EECS Dept., UC Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016.