Rapid Prototyping of Complex Micro-architectures Through High-Level Synthesis

Sara Sadat Hoseininasab, Caroline Collange, Steven Derrien

Rapid Prototyping of Complex Micro-architectures Through High-Level Synthesis

Sara Sadat Hoseininasab(✉), Caroline Collange, and Steven Derrien

Inria, Univ Rennes, CNRS, IRISA, Rennes, France
{sara-sadat.hoseininasab,caroline.collange}@inria.fr,
steven.derrien@irisa.fr

Abstract. Register-Transfer Level (RTL) design has been a traditional approach in hardware design for several decades. However, with the growing complexity of designs and the need for fast time-to-market, the design and verification process at the RTL level can become impractical. This has motivated for raising the abstraction level in hardware design. High-Level Synthesis (HLS) provides higher-level abstraction by automatically transforming a behavioral specification of a circuit into a low-level RTL, making it easier to design, simulate and verify complex digital systems. HLS relies on static scheduled data paths which can limit its effectiveness. This limitation makes it difficult to design the micro-architectural features of processors from an Instruction Set Architecture described in high-level languages. This work aims to demonstrate how the available features of HLS can be deployed in designing various pipelined processors micro-architecture. Our approach takes advantage of the capabilities of HLS and employs multi-threading and dynamic scheduling techniques to overcome the limitation of HLS in pipelining a processor from an Instruction Set Simulator written in C.

Keywords: High-Level Synthesis · Pipelined Micro-architecture · Multi-threading

1 Introduction

Field-programmable gate arrays (FPGAs) are flexible devices that offer numerous advantages for fast prototyping and evaluating complex designs, including CPUs and multi-cores [15]. During the hardware design process, prototyping serves as a crucial preliminary step to explore and evaluate designs before committing to the costly and time-consuming process of application-specific integrated circuit (ASIC) design. This approach enables designers to swiftly evaluate their designs on physical hardware, pinpoint any issues in the design process, and address them early on, prior to finalizing them for ASIC implementation.

Designing an FPGA-based soft-core processor using Register-Transfer Level (RTL) spans both software and hardware designs. First, the designer usually

F. Palumbo et al. (Eds.): ARC 2023, LNCS 14251, pp. 19–34, 2023.
https://doi.org/10.1007/978-3-031-42921-7_2

expresses the Instruction Set Architecture (ISA) execution model using programming languages such as C/C++ to verify the functional correctness of the processor. Then they develop the processor with its micro-architectural features in Hardware Description Languages (HDLs) such as Verilog and VHDL for synthesis which requires detailed digital design knowledge at a low abstraction level. These two design steps are performed sequentially, resulting in a time-consuming and challenging design flow and verification process, particularly for individuals lacking a background in hardware engineering. However, with the advent of High-Level Synthesis (HLS), the design and verification steps can now be conducted in parallel, thereby streamlining the process and enhancing its efficiency. By increasing the abstraction level from RTL to the behavioral level in HLS, software developers can also program FPGAs by focusing on their algorithm rather than individual registers and cycle-to-cycle operations. HLS automatically translates a design written in high-level languages (e.g., C/C++) into a hardware description, making FPGA programming easier and accessible for all developers, reducing the design time, facilitating design exploration and evaluation, and simplifying debugging compared to the manual RTL [12].

The process of synthesizing a processor from a high-level programming language involves expressing the behavioral description of the ISA as an Instruction Set Simulator (ISS) model in C/C++ and utilizing the HLS tool to obtain the hardware implementation. Although HLS works well with straightforward control flows, it encounters challenges when dealing with data-dependent control flows [7]. As we will show in Sect. 2, current HLS tools cannot infer a fully pipelined micro-architecture from this ISS. Since HLS relies on static scheduled data paths, it conservatively considers dependencies on the program counter (`pc`) and register file for all instructions and limits the performance to the worse-case schedule data path. We will show in Sect. 4 how this work tackles the challenge of designing pipelined micro-architecture of a processor.

This study serves as a use case to demonstrate the potential of HLS for inferring the micro-architectural characteristics of processors from an ISS implemented in C, with a particular focus on the RISC-V ISA. Our contributions are outlined as follows:

- Exploiting automatic scheduling in HLS: We showcase the effective utilization of the automatic scheduling feature offered by HLS, enabling the design of a CPU without delving into the RTL implementation details.
- Designing various micro-architectures: We propose and implement various classes of micro-architectures, including dynamic single-threaded, static multi-threaded, dynamic multi-threaded, and multi-core designs.
- Performance and area evaluation: We thoroughly evaluate our designed micro-architectures in terms of both performance and area metrics. Performance assessment involves analyzing factors such as maximum clock speed (F_{max}), and Million Instruction Per Second (MIPS), while area evaluation includes the examination of resource utilization on an FPGA.

The rest of the paper is organized as follows. Section 2 provides the necessary background and the motivation of our work. We discuss related works in Sect. 3.

In Sect. 4, we dive into the details of our designs and implementations. The experimental results and concluding remarks are presented in Sects. 5 and 6, respectively.

2 Background and Motivation

A single-threaded in-order pipelined processor is a type of CPU that leverages a pipeline to improve its performance by processing multiple instructions concurrently. The pipeline is single-threaded because it can only process instructions from a single hardware thread (*hart* in RISC-V terminology). In case of two consecutive instructions are dependent or require the same hardware resource at the same time an hazard occurs, resulting in a delay or stall in the execution of subsequent instructions. Designing the micro-architecture of this processor at RTL level can be complex and challenging and requires careful attention to ensure that all pipeline stages are well-coordinated.

2.1 Different Micro-architecture Design Tools

There are various methodologies for designing digital circuits, including HDL, HLS and Hardware Construction Language (HCL). Using HDL, designers must accurately describe the behavior of digital circuits using registers, logic gates and other basic building blocks that operate on the data stored in the registers. In addition, designers must introduce many of the performance and timing constraints in the design and carefully balance the trade-offs between performance, area and power consumption. Designing process using HDL is time-consuming and error-prone as it requires extensive manual coding and testing which can lead to mistakes and delays.

On the other hand, HLS and HCL allows designers to describe the hardware design at a higher level of abstraction than HDL. HLS achieves this by enabling designers to describe hardware designs in C, C++, or SystemC, raising the level of abstraction and coping with design complexity [18]. HLS tools then automatically translate this high-level code into low-level hardware description languages, resulting in faster design time, easier verification and simplified DSE. In contrast to HLS which infers hardware from high-level software description, HCL allows designers to build complex hardware designs in a higher-level programming paradigm [3], while operating at the same level as RTL. Chisel as an HCL candidate allows designers to compose reusable components using high-level constructs like object-oriented programming for describing the functionality of a hardware [1]. However, prototyping and exploring different design alternatives using Chisel is not straightforward and can be time-consuming.

Figure 1 illustrates the anticipated pipelined schedule derived from a vector addition example. Consider two arrays, namely A and B, both containing three elements. The objective is to perform element-wise addition of Array A with the corresponding elements of Array B, storing the results in Array D. The pipeline approach enables concurrent execution of addition operations on different elements. This figure also presents the code snippets in HLS, and RTL level

necessary to achieve this scheduling. To achieve the desired scheduling using HLS, the loop pipelining directive can be applied over a loop that iterates on the size of the array, performing addition on the elements of arrays A and B. The automatic pipeline scheduling feature provided by HLS facilitates the generation of the desired schedule. On the other hand, when working at RTL level, it is necessary to carefully divide the various stages of the pipeline and describe the operations that must be performed at each individual stage.

The complexity of designing vector addition at the RTL level, as shown in Fig. 1, highlights the challenges and difficulties faced in designing the micro-architecture of a processor at this level of abstraction. Although HCL provides a higher level of design methodology compared to HDL, designers still need to focus on the explicit description of micro-architecture and manage the pipeline stages in their design, resulting in complicated hazard detection and resolution.

In summary, HLS technology presents several advantages over HDL and HCL by increasing the level of abstraction. First, HLS can significantly reduce design time by eliminating the need for manual coding of hardware descriptions, resulting in rapid prototyping and simulation. Second, HLS can enable designers to explore multiple design options quickly and efficiently, allowing them to identify the optimal design solution. Third, HLS can provide a more intuitive and easier-to-understand design flow, allowing designers to focus on system-level behavior rather than implementation details [14].

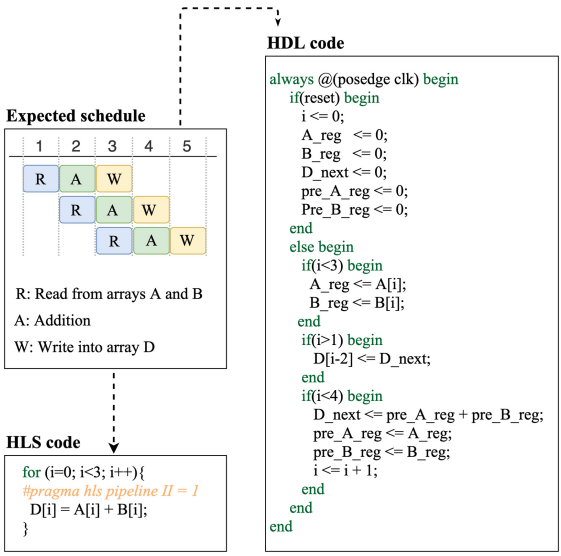

Fig. 1. The expected 3-stage pipeline schedule of vector addition.

2.2 HLS Limitation in Pipelining an Instruction Set Simulator

The code snippet displayed in Fig. 2 shows the ISS of an in-order single-threaded processor. To boost the processor's performance, a pipelined architecture can

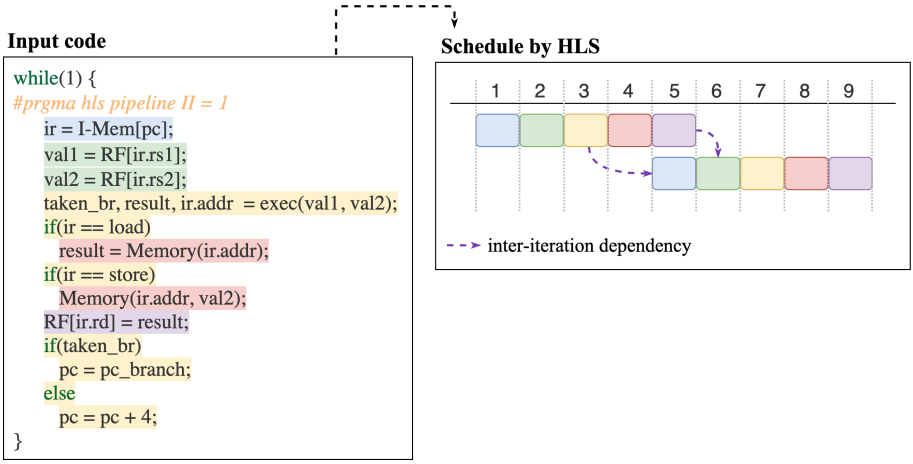

Fig. 2. Static pipelining of a single-threaded processor by HLS

be constructed by utilizing the loop pipelining directive in HLS. The schedule provided by HLS from this kernel as an input is depicted in this figure, where each stage corresponds to the line(s) in the code with the same color.

Loop pipelining is a technique that allows for the execution of different instructions to overlap, resulting in better throughput and ideally leading to a new instruction commencing execution every cycle. However, the read-after-write (RAW) dependency over the `pc` and the content of the register file presents significant obstacles for generating a pipelined micro-architecture by HLS tools. As demonstrated in the figure, HLS is unable to generate a fully pipelined micro-architecture from a single-threaded ISS because of its reliance on static scheduling and consideration of worst-case scenarios, where dependencies exist between all two instructions in a row. Therefore, HLS has to increase the Initiation Interval (II) - the number of cycles between the execution of two consecutive iterations of a loop - to 4 in order to respect these dependencies.

Over the years, pipeline hazards have emerged as a major challenge in micro-architecture design using HLS, significantly limiting the achievable throughput. This work takes advantage of dynamic scheduling and hardware multi-threading to handle the data dependencies in the pipeline.

3 Related Work

This section examines some of the approaches to tackle the challenges posed by static pipelining in HLS. Furthermore, this section also explores relevant works in dynamic multi-threading micro-architecture designs.

3.1 Deploying Speculative and Dynamic Techniques in HLS

Several recent researches focus on developing scheduling methods to address the conservatism in static scheduling of HLS. Two such methods proposed by All et

al. [2] and Dai et al. [6] involve introducing a mechanism for resolving pipeline hazards at run-time, thus enabling pipelining of loops with dynamic data dependencies. Josipović et al. [9] propose generating elastic data flow circuits that allow for dynamic scheduled pipeline, which leads to increased throughput in scenarios involving variable-latency operations or dynamic data dependencies. On the other hand, [7] and [10] present works on speculative scheduling. Derrien et al. [7] propose a mechanism for supporting control and memory speculation in static loop pipelining. Their approach involves tracking all speculated data, which can be discarded if a misspeculation occurs. Josipović et al. [10] incorporate speculation into data flow circuits by allowing parts of the circuit to execute with speculated data, while also employing a rollback mechanism to switch back to correct data when needed.

3.2 Pipelined CPU Designs Using HLS

In recent years, there have been several endeavors to generate pipelined processors using HLS. Researches such as those discussed in [16] and [13] concentrate on exposing the pipeline and hazard detection unit at the C level to enhance performance, necessitating coding that directly presents the pipeline stages. Additionally, a pipelined multi-threaded processor introduced in [8] using a similar approach but it partitioned all arrays in design, such as register file, into the registers in order to achieve $\mathrm{II} = 1$. Moreover the design's generality is limited, as it can accommodate a maximum of 8 harts, and scaling it to support a larger number of hart is a tedious task as it needs to adapt the hart scheduling unit at every stage of the pipeline. All these approaches described the pipeline stages at a programming level and forced HLS tool to adhere to this schedule, rather than relying on its automatic scheduling. Despite their use of high-level languages, they can be seen as RTL designs in disguise, with a level of complexity similar to RTL, as they require identification and handling of pipeline stages and pipeline hazards.

3.3 Dynamic Hart Scheduling in Multi-threaded CPU and GPU

Many studies focus on dynamic hart scheduling in multi-threaded CPUs. Notably, the research presented in [11] and [4] implement a dynamic hart scheduler that effectively switches to the next hart in case of long latency events, such as a cache miss.

Simty [5] and Vortex [19] are two RTL-based GPU cores that employ a dynamic scheduling methodology to effectively select a warp - a group of harts - to proceed within their architectures. The adaptive warp scheduling enables the GPU core to dynamically prioritize and schedule warps based on their readiness for execution and available resources.

4 Proposed Approach

In this section, the aim is to demonstrate the feasibility and challenges of designing an in-order pipelined processor at a higher level of abstraction by HLS.

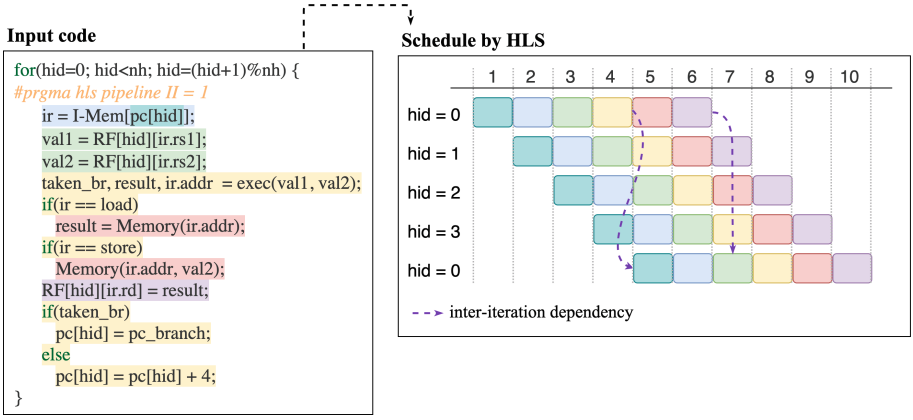

Fig. 3. Instruction Set Simulator of a static multi-threaded processor with four harts

4.1 Static Multi-threaded RISC-V Core

Figure 3 represents the high-level description of a multi-threaded processor where several harts (nh) are interleaved in a round-robin fashion at left and its corresponding schedule provided by HLS tool at right. This processor architecture allows each hart to have a unique identifier (hid), as well as a private `pc` and register file, which can support hardware multi-threading.

The round-robin scheduling of the harts means that switching to a different hart at each cycle, eliminating the need for dependency checking on the register file and branch prediction logic. Additionally, since the number of harts within the core is greater or equal the pipeline's depth, the processor does not fetch an instruction from the same hart until all control and data dependencies are resolved. This method offers a significant benefit by enabling the processor to mask the latency involved in accessing off-chip memory, a potential bottleneck that can limit the performance of high-computing applications [17]. By using a round-robin scheduling approach for multiple harts, the processor can ensure that each hart can execute its instructions without delay, even if another hart is engaged in a memory operation with high latency.

4.2 Dynamic Single-Threaded RISC-V Core

To ensure the effectiveness of static multi-threading, it is important to have an adequate number of harts within the processor. This number depends on the specific architecture of the design and the latency of the off-chip memory, and it is predetermined during the design phase. If the number of harts falls below this threshold, the performance of the processor will diminish as it is not possible to achieve II = 1 anymore. Hence, it is crucial to use dynamic scheduling to mitigate the impact of the dependencies and memory latency by effectively leveraging as many harts as available within a processor.

To pipeline a processor with a single hart, a specific sequence of actions must be implemented to ensure efficient instruction fetching and execution. First, the processor must be forced to initiate the instruction fetching process when its `pc` is ready, and then execute it whenever its source registers are available. In cases where an instruction is not a conditional branch, the next `pc` will be available after the instruction is fetched. On the other hand, in cases where the instruction is a conditional branch, the branch `pc` will only be available during the execution stage.

To address the dependency on `pc`, a scoreboard is utilized to determine when the `pc` will become available. During each cycle, the processor first checks the scoreboard to verify if the `pc` is ready. If the `pc` is not yet ready, the processor must wait before proceeding with instruction fetching. On the other hand, if the `pc` is available, the processor can proceed with fetching the instruction. The processor utilizes a similar mechanism to tackle the dependency on the register file: when an instruction depends on the result of a load instruction, the processor must wait until the write back stage is completed. However, if the dependency is on a non-load instruction, the forwarding path comes into play, and the value from the previous execution stages is forwarded to the current execute stage.

4.3 Dynamic Multi-threaded RISC-V Core

In a dynamic single-threaded environment, CPU experiences idle cycles while awaiting the valid `pc` or write back of the previous instructions in case of the dependency on register file. Therefore, it is important to find ways to maximize CPU usage and minimize idle cycles. One effective approach is to engage the CPU in productive tasks rather than waiting for dependency resolution. This can be achieved by initiating execution from another hart that is ready at a given cycle. By making the most of idle cycles and leveraging multiple harts, it's possible to achieve significant improvements in overall efficiency and processing speed.

The Input code in Fig. 4 provides an illustration of an efficient approach to pipeline the execution of instructions from two harts by dynamically interleaving them. This algorithm involves a few key steps that enable the processor to operate efficiently. The first step is to select a single hart from a pool of available harts. In this case, the pool contains two harts. Once a hart is selected, the processor can proceed with the fetch of an instruction from the selected hart and subsequently execute it. This allows for the processor to take full advantage of the available harts and execute instructions in a parallel and efficient manner. However, in the event that no hart is chosen (*idle_state*), the processor will encounter a stall in its operation.

Each individual hart in the processor is equipped with a dedicated scoreboard that displays the readiness of the private `pc` to initiate the retrieval of an instruction, as well as the anticipated readiness of the source registers. This allows for efficient execution of instructions in a parallel fashion. In cases where the previous load instruction has yet to commit its result to the register file, this scoreboard serves as an indicator of potential dependencies. The scoreboard

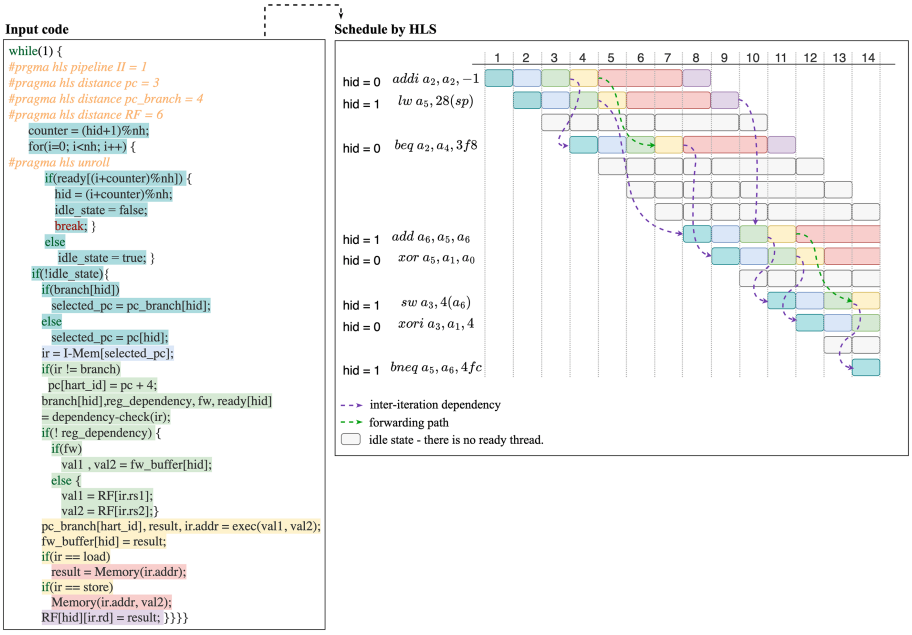

Fig. 4. Instruction Set Simulator of a dynamic multi-threaded processor with two harts

helps to identify potential data hazards that may occur during the execution of instructions. By examining the scoreboard, the processor can determine whether the instruction can proceed or if it needs to wait until the dependent instruction commits its result (*reg_dependency*). By detecting these hazards early, the processor can take corrective actions to ensure that the execution of instructions proceeds smoothly without any stalls. Additionally, each hart features a forwarding buffer that retains the outcomes of executed ALU instructions. The length of this buffer is correlated with the maximum number of preceding instructions on which the current instruction has dependency on based on the register file content and influenced by off-chip memory latency. The forwarding buffer is crucial in reducing the latency of dependent instructions. In cases where an instruction requires a value that has not yet been written to the register file, the processor executes a thorough examination of the forwarding buffer. Subsequently, the required value is forwarded to the instruction, thereby circumventing the need to wait for the instructions to write their results into the register file. This reduces the delay caused by the need to access the register file and improves the overall performance of the processor.

4.4 Thread Synchronization

In the previous sections, we have explored the possibility of hiding the dependency between two instructions from the same hart by either statically or

dynamically interleaving harts. However, when it comes to memory dependencies between harts, we need to consider a different approach. Assuming that all harts access distinct memory regions, as done in [8], is not realistic since harts require sharing information at certain points. In modern parallel computing systems, it is crucial to ensure that multiple processors or harts operate in a synchronized and coordinated manner to avoid race conditions and guarantee correct program behavior. In parallel computing, data races can occur when multiple threads attempt to access the same memory location simultaneously. This can lead to unexpected program behavior, such as incorrect values being read or written, or even crashes. To prevent such issues, it is important to use synchronization mechanisms to ensure that different harts access shared memory locations in a coordinated way and they are operating on consistent memory states.

One approach to synchronize harts, as proposed in this work, is to introduce a new instruction called *Barrier* to the processor. When a hart reaches the *Barrier* instruction, it will enter a sleep state, waiting for the other harts to arrive. The idea is to pause execution until all harts have reached the *Barrier* instruction. Once all harts have reached the *Barrier*, they can then be woken up one by one in sequence, ensuring that all harts proceed from the synchronization point together in a consistent manner. It is important to note that during this synchronization process, any hart in a sleep mode must not be included in the scheduling process. This ensures that the harts have already hit the *Barrier* are waiting for other harts to arrive, preventing any potential data races or incorrect program behavior. The *Barrier* instruction provides a way for harts to coordinate their actions and ensure that they are operating on consistent memory states. This is particularly important in parallel computing, where harts can execute in a non-deterministic order, leading to potential data races and incorrect program behavior.

From the software perspective, developers can easily integrate the *Barrier* instruction into their code using inline assembly where it is needed. Doing so, all executing threads are forced to pause and wait until all other threads reach the same point in the program before continuing.

4.5 Shared-Memory RISC-V Multi-core

The multi-threaded multi-core processor offers an efficient solution for handling different program instructions simultaneously. Each core in this processor has its own Instruction memory (*I-Mem*) and Data Scratchpad Memory (*D-SPM*), while sharing a single Data memory (*D-Mem*) with all other cores. The ability to perform multiple tasks at the same time significantly boosts the overall processing power of the system, making it a desirable option for various applications.

Designing such a multi-core processor in HLS can be achieved by using two nested loops, as described in Input code in Fig. 5. The outer loop iterates on the number of harts interleaved within a core, while the inner loop iterates on the number of cores (*nc*). Unrolling the inner loops by the factor of the number of cores will instantiate *nc* cores, and by pipelining the outer loop, one hart of

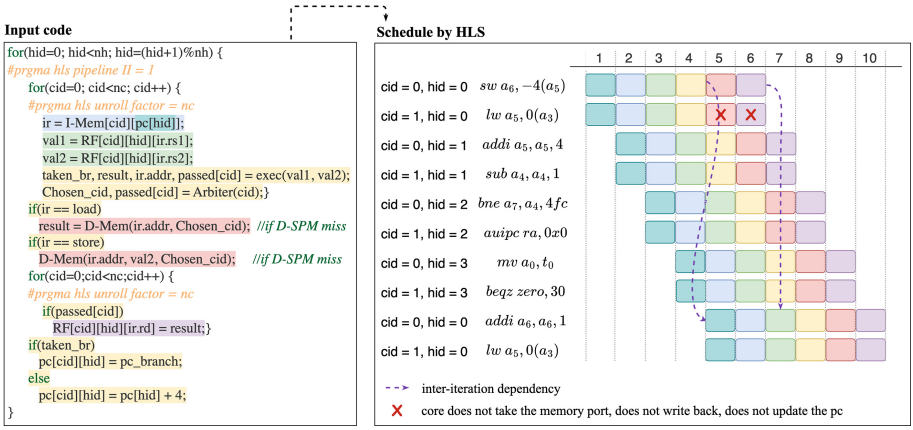

Fig. 5. Instruction Set Simulator of a Shared-memory multi-core system with two cores, each core has four harts

all cores starts execution at each cycle. However, designing such a processor in HLS can be challenging, because the HLS tool has no prior knowledge about the different instructions from different cores until run-time, it expects *nc* memory instructions at the same cycle. This leads to resource conflicts on the shared memory, II bigger than one, and poor performance.

To overcome the challenge of multiple cores accessing a shared port, an arbiter unit has been designed. The arbiter unit employs a round-robin algorithm to ensure that each core has an equal opportunity to access the port at each cycle. In addition, the pipeline is decomposed to separate the memory and write back stages from other stages, thereby ensuring that there is no more than one memory transaction at any given cycle. If multiple memory requests are received at the same cycle, the core that is granted access to the port (*Chosen_cid*) can proceed with its execution, while the other cores will have to wait until their turn comes up. Ultimately, only the cores that execute non-memory instructions or manage to acquire the port for the memory transaction (*passed*) will write back the results to their register file and update their `pc`. This approach allows designing the multi-threaded multi-core processor using HLS tools to handle multiple program instructions simultaneously in a highly efficient and effective manner.

5 Experimental Validation

In this section, we present an experimental study aimed at evaluating the performance and area utilization of our proposed micro-architecture designs. To conduct these experiments, we utilize the Vitis HLS 2022.2 tool, targeting a Kintex7 XQ7K410TRF9002L FPGA board. It is important to note that the data memory is considered off-chip and is accessed through an M-AXI port with a latency of 8 cycles. To assess the performance of our designs, we employ the matrix

multiplication benchmark, which offers a comprehensive workload for evaluating computational efficiency. The entire evaluation process, encompassing C simulation, C synthesis, RTL simulation, and RTL synthesis, is seamlessly executed using a single tool, Vitis HLS, streamlining the workflow. To measure the performance of the proposed micro-architectures, we record the maximum frequency achieved during RTL synthesis and the total number of clock cycles obtained from RTL simulation. Additionally, we evaluate the area utilization based on RTL synthesis, providing a comprehensive assessment of the micro-architectures' efficiency. Furthermore, as a baseline for comparison, we synthesize the RV32I configuration of the Comet processor [16].

Table 1. Area and performance evaluation: Comparison of proposed multi-threaded micro-architectures and Comet using matrix multiplication benchmark

Approach	nh	II	F_{max} (MHz)	MIPS	LUTs	FFs	SRLs	BRAMs
Comet [16]	1	1	134	48	$2k$	$1.1k$	0	2
Fully Dynamic	2	1	113	52	$10k$	$5.8k$	132	2
	4	1	98	92	$16k$	$7.2k$	133	2
	8	1	78	75	$21.4k$	$8.6k$	134	10
Hybrid	4	1	109	103	$11.1k$	$2.2k$	131	2
	8	1	103	85	$15.3k$	$3k$	142	3
Fully Static	16	1	203	186	$2.2k$	$1.4k$	297	6

Table 1 presents the experimental results for our multi-threaded processor with different numbers of harts, indicated by the *nh* column. We consider various approaches, including Fully Dynamic, Hybrid, and Fully Static. When using 2 harts in the processor, dynamic scheduling is the only option to hide dependencies. However, when the processor has at least 4 harts, we have the choice to schedule them either dynamically or statically. In the static scheduling approach, we can hide the dependency on `pc`, leaving only the dependency on the register file. This allows us to dynamically resolve the register file dependency. In this approach which is referred as Hybrid in the table, harts are scheduled using a round-robin scheduling technique, which ensures fairness and balanced utilization. Before the execution of each hart, a thorough verification of the scoreboard is performed to ascertain the readiness of the source registers. In the event of identifying a dependency, the scheduled hart is required to forfeit its turn, resulting in a temporary halt in processor activity. Upon the subsequent arrival of the hart's turn, the same instruction must be reissued.

The data presented in Table 1 clearly demonstrates that the proposed multi-threaded micro-architectures exhibit superior performance in terms of MIPS when compared to the Comet architecture. By employing multiple harts in the processor, we are able to achieve a remarkable increase in MIPS, surpassing Comet by at least 8%. Notably, the Fully static micro-architecture with 16 harts

demonstrates significantly better performance while occupying a comparable area to Comet.

In our experimental evaluation, we observe that in the Fully Dynamic and Hybrid approaches, the forwarding unit is in the critical path of the design and becomes a limiting factor in terms of performance. This characteristic explains why increasing the number of harts from 4 to 8 does not yield performance benefits, despite increasing the utilized area. In the Fully Dynamic approach, the scheduling unit is essential for resolving dependencies and ensuring proper execution order, resulting in increased area usage. However, by adopting the Hybrid approach, we are able to eliminate the scheduling unit, thereby reducing the area overhead while achieving better performance compared to the Fully Dynamic approach. To visualize the trade-offs between performance and area utilization, a scatter plot is generated (see Fig. 6). The scatter plot represents the Hybrid and Fully Dynamic approaches with 4 and 8 harts. The Pareto front showcases the designs that offer the best compromise between performance and area utilization. Upon examining the plot, it becomes evident that the Hybrid approach with 4 harts (4H) occupies a position on the Pareto front. This positioning indicates that the Hybrid approach with 4 harts is a superior design choice, as it achieves a desirable balance between performance and area utilization compared to the other approaches.

In our investigation, we examine the impact of increasing the number of harts to 16 on our micro-architecture. This expansion enables us to adopt a Fully Static execution scheme, eliminating the need for scheduling, hazard detection, and forwarding units. By statically interleaving multiple harts, we successfully conceal dependencies on both `pc` and register file. Through coordination of instruction execution in a predetermined order, we achieve static scheduling and hazard resolution. This approach significantly reduces the complexity of the design and the required area for these units, leading to increased performance. While the configuration with 8 harts using Fully Dynamic approach is found to be inefficient, we can utilize either 4 harts with a Hybrid approach or 16 harts with a Fully Static approach instead. Our study demonstrates how different micro-architectures and configurations, considering various numbers of harts and approaches, can be designed to determine the optimal design based on silicon budget and desired performance.

Figure 7 illustrates the scatter plot for proposed multi-core processor, where each data point is labeled with the corresponding number of cores ranging from 1 to 8. It is important to note that each core in our design supports 16 harts, implemented using the Fully Static approach. This configuration allows for increased parallelism and enhanced performance within each core. The plot provides a visual representation of the performance and area trade-offs associated with different core counts. It enables us to observe how the number of cores impacts both performance and area utilization. A noticeable trend observed in the plot is the linear increase in the used area as the number of cores is increased. This observation aligns with our expectations, as each additional core contributes to an incremental area overhead. However, it is essential to highlight that the

performance does not scale linearly with the number of cores, as anticipated. Notably, by utilizing our design, researchers and developers have the opportunity to rapidly prototype various multi-core micro-architectures with different core counts, tailored to their specific parallelism requirements and workloads. The linear increase in the used area as the number of cores expands demonstrates the flexibility of our design in accommodating diverse multi-core configurations. By swiftly iterating through different core counts, designers can explore and evaluate the performance and area trade-offs for their particular applications, allowing for quick experimentation and optimization of the micro-architecture. This capability to rapidly prototype different multi-core configurations empowers designers to fine-tune their designs according to the specific demands of their workloads, harnessing the potential benefits of parallel processing while considering the limitations imposed by sequential portions of the program.

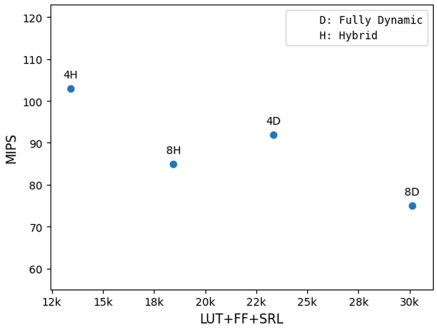

Fig. 6. Relationship between performance and area utilization for different approaches in multi-threaded processor with 4 and 8 harts

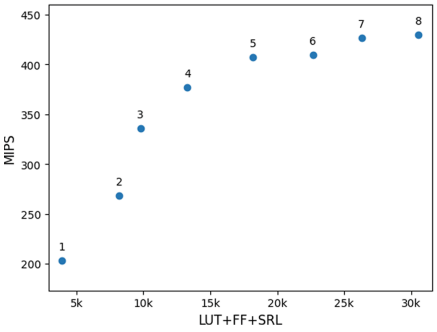

Fig. 7. Relationship between performance and area utilization for different number of cores in multi-core processor

6 Conclusion

This study demonstrates the benefits of using automatic scheduling in HLS for rapid prototyping of CPU architectures. We identify limitations of existing HLS tools in inferring fully pipelined micro-architectures with dynamic data dependency. Additionally, we showcase the successful design of complex micro-architectures, such as static multi-threaded CPU, dynamic multi-threaded CPU, and multi-core systems. These results emphasize the effectiveness of leveraging HLS for advanced architectures.

Future work involves exploring GPU micro-architecture design using HLS to uncover potential challenges and limitations.

Acknowledgements. This study is partially funded by the French National Research Agency (ANR) as part of the project DYVE (ANR-19-CE25-0004-01).

References

1. Chisel Homepage. https://www.chisel-lang.org
2. Alle, M., Morvan, A., Derrien, S.: Runtime dependency analysis for loop pipelining in high-level synthesis. In: Proceedings of the 50th Annual Design Automation Conference, pp. 1–10 (2013)
3. Bachrach, J., et al.: Chisel: constructing hardware in a scala embedded language. In: Proceedings of the 49th Annual Design Automation Conference, pp. 1216–1225 (2012)
4. Borkenhagen, J.M., Eickemeyer, R.J., Kalla, R.N., Kunkel, S.R.: A multithreaded powerPC processor for commercial servers. IBM J. Res. Dev. **44**(6), 885–898 (2000)
5. Collange, C.: Simty: generalized SIMT execution on RISC-V. In: CARRV 2017–1st Workshop on Computer Architecture Research with RISC-V, vol. 6, p. 6 (2017)
6. Dai, S., et al.: Dynamic hazard resolution for pipelining irregular loops in high-level synthesis. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 189–194 (2017)
7. Derrien, S., Marty, T., Rokicki, S., Yuki, T.: Toward speculative loop pipelining for high-level synthesis. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **39**(11), 4229–4239 (2020)
8. Goossens, B.: Guide to Computer Processor Architecture: A RISC-V Approach, with High-level Synthesis. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-18023-1
9. Josipović, L., Ghosal, R., Ienne, P.: Dynamically scheduled high-level synthesis. In: Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 127–136 (2018)
10. Josipovic, L., Guerrieri, A., Ienne, P.: Speculative dataflow circuits. In: Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 162–171 (2019)
11. Kvatinsky, S., Nacson, Y.H., Etsion, Y., Friedman, E.G., Kolodny, A., Weiser, U.C.: Memristor-based multithreading. IEEE Comput. Archit. Lett. **13**(1), 41–44 (2013)
12. Liu, S., Lau, F.C., Schafer, B.C.: Accelerating FPGA prototyping through predictive model-based HLS design space exploration. In: Proceedings of the 56th Annual Design Automation Conference 2019, pp. 1–6 (2019)
13. Mantovani, P., Margelli, R., Giri, D., Carloni, L.P.: HL5: a 32-bit RISC-V processor designed with high-level synthesis. In: 2020 IEEE Custom Integrated Circuits Conference (CICC), pp. 1–8. IEEE (2020)
14. Meeus, W., Van Beeck, K., Goedemé, T., Meel, J., Stroobandt, D.: An overview of today's high-level synthesis tools. Des. Autom. Embed. Syst. **16**, 31–51 (2012)
15. Ravindran, K., Satish, N., Jin, Y., Keutzer, K.: An FPGA-based soft multiprocessor system for IPv4 packet forwarding. In: International Conference on Field Programmable Logic and Applications, pp. 487–492. IEEE (2005)
16. Rokicki, S., Pala, D., Paturel, J., Sentieys, O.: What you simulate is what you synthesize: designing a processor core from C++ specifications. In: 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1–8. IEEE (2019)

17. Smith, B.J.: Architecture and applications of the HEP multiprocessor computer system. In: Real-Time Signal Processing IV, vol. 298, pp. 241–248. SPIE (1982)
18. Takach, A.: High-level synthesis: Status, trends, and future directions. IEEE Des. Test **33**(3), 116–124 (2016)
19. Tine, B., Yalamarthy, K.P., Elsabbagh, F., Hyesoon, K.: Vortex: extending the RISC-V ISA for GPGPU and 3D-graphics. In: MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 754–766 (2021)