A MLIR-Based Hardware Synthesis Framework

Ruifan Xu
Peking University
xuruifan@pku.edu.cn

Youwei Xiao
Peking University
shallwe@pku.edu.cn

Jin Luo
Peking University
luo-jin@pku.edu.cn

Yun Liang
Peking University
ericlyun@pku.edu.cn

***Abstract*—Hardware synthesis adopts a higher abstraction to improve the productivity of hardware design. High-level synthesis tools can automatically transform a high-level description into hardware design, while hardware generators adopt domain-specific languages and synthesis flows for specific applications. However, the implementation of these tools generally requires substantial engineering efforts due to RTL's weak expressivity and low level of abstraction. To lower the engineering cost and get competitive hardware design rapidly, we build Hector, a two-level IR providing a unified intermediate representation for hardware synthesis methodologies. The high-level IR binds computation with a control graph annotated with timing information, while the low-level IR provides a concise way to describe hardware modules and elastic interconnections among them. Implemented based on the multi-level compiler infrastructure (MLIR), Hector's IRs can be converted to synthesizable RTL designs. Different hardware synthesis approaches can adopt suitable levels of intermediate representations (IR) and are well supported in Hector with minimal engineering effort. The multi-level representation also enables optimizations like pipeline, which is hard to support at RTL.**

***Index Terms*—intermediate representation, hardware synthesis**

I. Introduction

Traditional hardware description languages (HDLs) including Verilog [3] and VHDL [4] adopt a low level of abstraction, which seriously hampers the productivity of hardware design. Hardware synthesis tools including high-level synthesis (HLS) and hardware generators provide a higher abstraction to get hardware design quickly. These synthesis tools often adopt different synthesis methodologies. HLS compilers automatically convert the high-level description into a hardware implementation. HLS tools often take high-level languages as inputs and reuse their compiler infrastructure for transformations and optimizations. For example, Vitis HLS [9] adopts LLVM IR [6] as its internal representation, which is a widely used software compilation IR. After that, additional passes are needed to convert to hardware semantics. Hardware generators perform sophisticated architectural transformations to improve performance and resource consumption. In addition to domain-specific optimizations, hardware generators are often guided with specific hardware templates to generate hardware implementation for different applications. Domain-specific languages (DSL) and optimizations are often used for description and optimization.

Both synthesis methodologies require substantial engineering efforts due to RTL's weak expressivity and low level of abstraction. The semantic gap between RTL and higher abstraction further improves the difficulty of engineering, because the lowering process should handle different levels of information at the same time. However, these different methodologies share some similarities in the intermediate representations like control logic generation. The scheduling step in HLS is implemented using different algorithms including static, dynamic, and hybrid scheduling [8]. Different hardware behaviors including pipeline and streaming are also adopted in hardware generators. Furthermore, a multi-level description also simplifies the generation process and enables optimizations at different levels. Therefore, a unified and multi-level IR makes it easier to design new hardware synthesis techniques based on the same infrastructure and explore different methodologies.

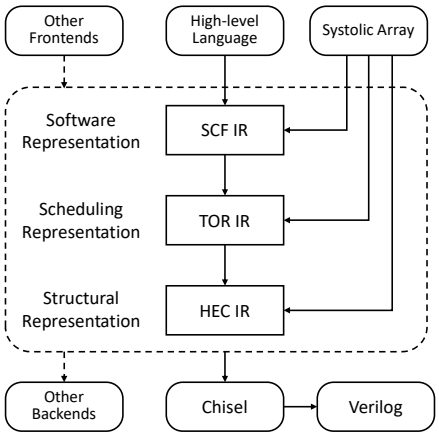


Fig. 1: Hardware synthesis flow built on the Hector.

We propose Hector [5][1], a two-level IR providing a unified description for different hardware synthesis methodologies with expressivity and flexibility. The high-level IR (TOpological Representation) binds computations with a control graph annotated with timing information, while the low-level IR (Hierarchical Elastic Component) provides a concise way to describe various hardware components and elastic interconnections among them using customizable primitives. Both IRs provide a uniform representation of the control logic in various manners but at different abstraction levels. The IRs in Hector are converted to synthesizable RTL programs through a series of transformations including time graph transformation, lowering pass, and RTL generation. The two-level IR and

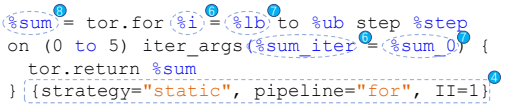
[1]Hector is open source at (https://github.com/pku-liang/Hector)

all the transformations are built on the multi-level compiler infrastructure (MLIR) [1] (section II), which simplifies the implementation and provides the opportunity of sharing optimizations at different levels. As shown in Figure 1, the proposed IR (section III) together with a built-in dialect provides the ability of describing hardware at software, scheduling and structural levels. This multi-level description makes it easier to design with multiple abstractions and get competitive hardware design quickly. In addition, Hector provides optimizations like pipeline (section IV) that can be easily reused in hardware synthesis.

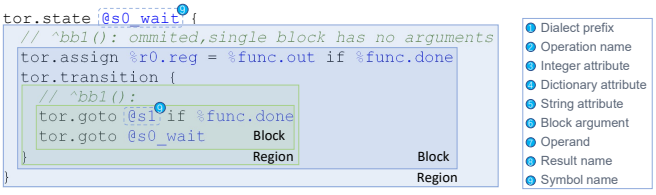
II. MLIR Infrastructure

MLIR is a novel compiler infrastructure that provides powerful scalability and modularity and aims at providing multi-level IR. All the IRs in MLIR obey Static Single Assignment (SSA) form and explicit type system. IR in MLIR is called *Dialect*, which is a hierarchical structure template that is shown as Figure 2c. This hierarchy supplies an expressive representation, which makes it easy to implement a flexible IR. The feature that *region* can also be nested in a *operation* provides enough semantic features including functional programming and nested operation. *Dialect* can be seen as a collection of user-defined operations and types, which makes it easy to define a new language and implement a multi-level IR. MLIR also provides several built-in Dialects, such as SCF and Standard. SCF Dialect describes static control flow in a higher level abstraction than jumping between different blocks. This dialect prevents if and loop structure, which helps analyze and optimize. Standard Dialect is a collection of basic operations such as addition, comparison and function call. These dialects can be used to describe a program at the same time, which is also the feature provided by MLIR, called progressive lowering.

(a) ToR operation in custom printing/parsing format.



(b) The "tor.for" operation in internal representation.



(c) HEC operations in MLIR hierarchy structure.

Fig. 2: Example HEC operations that make use of in MLIR

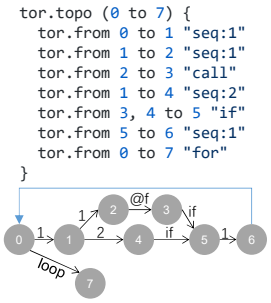
Figure 2 shows the detailed implementation of Hector IR, which can be easily built on MLIR due to flexible representation. Each operation is composed of a prefix and operation name, which is shown in Figure 2a. Figure 2b shows the representation of a loop. This loop operation takes `%lb`, `%ub` and `%step` as operands, and returns the value of `%sum`. The iterative variable `%i` and accumulative result `%sum_iter` are the block arguments of this operation, and the initial value `%sum_0` is also an operand. Besides basic operands and results, operations may have *attributes* and *regions*. Figure 3b shows the functional syntax of operations in ToR. There are two *regions* in `tor.if` representing two branches separately. The nested region is naturally supported in MLIR, which enables the definition of `state` in Figure 2c. The timing information, the binding of operations and the behavior of a module are all implemented as *attributes* including integer, string and directory.

III. Hector Representation

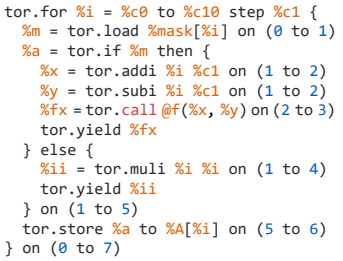
Hector contains a two-level IR system, where ToR is the high-level IR and HEC IR is the low-level IR. ToR IR combines a software-like control flow with the schedule information of each operation. HEC IR proposes an allocate-assign mechanism to explicitly describe the relationship between computation and compute units. Both IRs provide a uniform representation of the control logic with various scheduling manners such as static and dynamic. The main difference between the two IRs is that ToR describes when the operation begins, while HEC describes where it takes place. The two-level representation makes it easy to implement different synthesis methods. In this section, we present the details of the IRs, ToR (TOpological Representation) at the high level and HEC (Hierarchical Elastic Component) at the low level.

A. ToR IR

The software IR such as LLVM IR [6] lacks hardware semantics. The idea of the high-level IR is to make it closer to hardware by providing a directed graph that carries control flow and timing information and binding software operations to elements of the graph. ToR is composed of two parts, topology and functional operations.



(a) ToR Topology



(b) Functional Operation

Fig. 3: The design of ToR IR. ToR IR consists of topology and functional operations. In (a), topology describes the time graph with supplementary information on edges, where all edges and nodes are set "static" as default. In (b), functional operations are bound on the graph.

Topology describes a time graph, which is a directed graph describing control flow and timing information. Topology includes a `tor.topo (x to y)` operation, which indicates the source node `x` and sink node `y`, respectively. The `tor.from` operations inside `tor.topo` specify edges of the time graph. Attributes add supplementary information such as latency and scheduling manners to the time graph. There are four types of nodes in the time graph. **Normal** node for a sequential edge, **call** node for a function call, **if** node with two edges, and loop node with a loop body and loop back edge. The type of each node on the time graph is determined by the operation binding. The combination of these four node types is capable of describing the schedule at high-level.

Three scheduling manners: **static**, **pipeline**, and **dynamic**, are supported in `ToR`. Pipelining is a key optimization technique to improve throughput. `ToR` supports pipelining by aligning branches of all `if` operations and adding `pipeline` and `II` attributes to modules. Topology also supports dynamic behavior that resolves conflicts at run-time. Stalling occurs only when the conflict occurs, avoiding the conservative assumption of static behaviors. This unified representation makes it easier to transform among different behaviors.

Functional operations present the algorithmic specification with high-level control flow semantics (e.g., if, for, and while). It binds each operation to some element of the time graph, either a node or an edge. To be specific, general operations (computation, memory access, function call) are bounded on edges, while if/loop operations are bounded on nodes. Figure 3 illustrates an example of `ToR` IR. The time graph in (a) contains a loop, which is composed of two branches $1 \rightarrow 2 \rightarrow 3$ and $1 \rightarrow 4$. There is also a function call on the edge $2 \rightarrow 3$, and the loop exits at the edge $0 \rightarrow 7$. Figure 3b shows the functional operations which are bounded on the time graph. For example, the `tor.for` operation is bounded on node `0`, and the `tor.muli` is bounded on edge `(1 to 2)`.

B. HEC IR

`HEC` IR describes hardware with different manners in a unified allocate-assign mechanism. Allocation explicitly defines all function units and sub-modules on the datapath, and the signals of these units are determined through assignments. The allocate-assign mechanism omits the insertion of the multiplexer, simplifying the assignment of signals.

```
//Allocations
%m.lhs, %m.rhs, %m.res =
   hec.primitive "m" is "muli" : i32, i32, i32
%i = hec.wire "i" :i32
//Assignments
hec.assign %m.lhs = %0
hec.assign %i = %m.res if %valid
```

Compared with `ToR`, `HEC` works at a level much closer to hardware. It explicitly describes the resource usage (including registers, memory, and compute units). Corresponding to the different behaviors in `ToR`: **static**, **pipeline** and **dynamic**, a `HEC` design is composed of three types of components matching their manners.

```
hec.component @STG(%c, %d, %f) {
   ... // Allocations
   stateset {
      state @s0{
         assign %addf.op0 = %c
         assign %reg0 = %addf.result
         transition {
            goto @s1 if %d
         }
      } //other states
   }
} {"STG"}
```

`HEC` describes a static module in a state transition graph (STG) style. The `hec.stateset` operation defines a set of states. Inside each state `@sx`, `hec.assign` operations specify the signal delivery among the allocated resources. Such representation naturally supports fine-grained parallelism. There is also a `tor.transition` operation in every state, specifying which state the control is transferred to, either unconditionally or based on guard signals. Based on the allocate-assign mechanism, it is convenient to describe resource sharing in an STG-style component by simply feeding signals into the shared resources (either registers or compute units) inside different states. Other two style components: pipeline and handshake have similar representations except the control logic.

IV. Pipeline Optimization

Transformation and optimization are implemented as passes in the MLIR infrastructure. The multi-level representation simplifies the implementation of optimizations like pipelining. Pipeline optimization constructs a cycle-sensitive multiple-stage component, where each node of the time graph in the original `ToR` function is allocated to a definite stage according to its distance from the source node. The pipeline structure requires that every path from the source node to a specific time node must have the same length, which serves as the prerequisite of the pipeline's stability. As shown in the Figure 4, pipeline generation pass has four steps:

a) Stage division: The generator traverses the time graph of the operated `ToR` function and calculates the "distance" of each node from the source node. The "distance" is indicated by "#cycle" attributes of every time edge composing the path from the source node to the current node. The time nodes with the same "distance" will be placed in one stage. Each stage responds to a clock cycle, on which the allocated time node starts the execution of the operations whose start time is set as the current node. As shown in the left figure of Figure 5a, the stage number and the distance from the source node strictly match for every node on the time graph.

b) Register Allocation and Binding: Each SSA-formed value in the `ToR` function has one def and several uses, and values with no use are optimized by dead code elimination pass. The value must be stored for pipelining usage after its def until the last use, so assuming that the value is defined at the sth stage, and used at the tth stage for the last time, $t - s$ registers have to be allocated, each of which is bound to

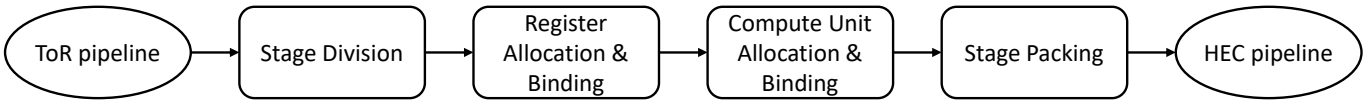
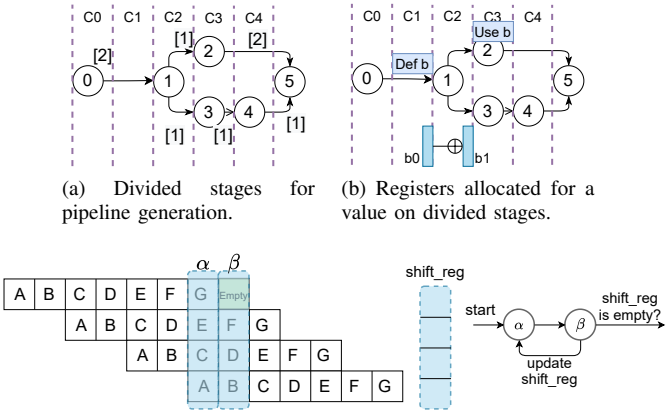


Fig. 4: Pipeline generation flow.



(a) Divided stages for pipeline generation.

(b) Registers allocated for a value on divided stages.

(c) Pipeline states composed of stacked stages.

Fig. 5: Stages and states for pipelining.

a value edition in a certain middle stage. As the right figure of Figure 5b shows, value b is defined at stage C1 and lastly used at stage C3, and two registers are allocated and store b's data between C1, C2 and C2, C3 individually.

c) Compute Unit Allocation and Binding: Each compute operand in `ToR` dialect, including integer addition, integer multiplication, etc., corresponds to a specific compute unit in `HEC` primitives and hardware implementation. The divided stages are overlapped due to the profiling initiation interval, and the compute units on any overlapped stage conflict with each other. Resource sharing happens among compute units of the same kind with no conflicts. A compute unit that is shared by several uses will be allocated only once, then it will be bound to those conflict-free uses by appropriate signal assignment operation in `HEC` dialect.

d) Stage Packing: Each pipeline state is composed of stacked pipeline stages due to various initiation interval (II). For a pipelining loop or function with initiation interval II, II stages are constructed. For example, as shown in Figure 5c, two states, marked as $\alpha =< A, C, E, G >$ and $\beta =< B, D, F, \text{Empty} >$, are generated for a pipelining loop with $II = 2$, and a shift register of size $\lceil \frac{\#\text{stages}}{II} \rceil$ is allocated for control of the execution of each stage stacked in a state. Finally, the partial state transition solution is generated.

Due to the unified description of `HEC`, the generation of pipeline only considers about all the stages and resources for each operation. The implementations of control logic like a finite state machine and shift register are omitted in the representation.

V. Future Work

We aim at improving the infrastructure of Hector at present. First, we plan to build necessary tools including simulation and verification at different levels, which can significantly improve the robustness and convenience of Hector. We also plan to integrate Hector with CIRCT [2] infrastructure, an open-source hardware compiler infrastructure. Because CIRCT is also built on the MLIR, the implementation of Hector IR and internal passes can be simply shared by CIRCT. We will insert conversions between Hector and CIRCT for extensibility and reusability. To avoid extra compilation of Chisel programs, we should replace Chisel generation with a translation pass to the FIRRTL [7] IR which is a CIRCT dialect of IR in Chisel. Finally, we plan to implement some real-world accelerators like sparse matrix-vector multiplication (SPMV) to prove the effectiveness and convenience of Hector.

VI. Conclusion

Hector is a two-level IR providing a unified description for different synthesis methodologies. Through a series of transformations and optimizations based on the MLIR infrastructure, Hector's IRs are finally converted to synthesizable RTL programs. The multi-level description enables optimization passes like pipelining that can be easily reused by hardware synthesis methodologies. Moreover, the open-source framework provides enough flexibility to customize synthesis approaches and allows users to explore advanced techniques.

References

[1] C. Lattner *et al.* "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).
[2] The CIRCT authors, "CIRCT: Circuit IR compilers and tools," 2022, https://github.com/llvm/circt.
[3] IEEE. 1364-2005. Standard for Verilog Hardware Description Language.
[4] IEEE. 1076-2008. VHDL Language Reference Manual.
[5] Ruifan Xu *et al.* "HECTOR: A Multi-level Intermediate Representation for Hardware Synthesis Methodologies", to appear in the proceeding of the International Conference on Computer Aided Design (ICCAD), Nov. 2022.
[6] C. Lattner *et al.* "LLVM: a compilation framework for lifelong program analysis & transformation." In: 2004 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).
[7] A. M. Izraelevitz *et al.* "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," in IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2017
[8] Jianyi Cheng *et al.* 2020. Combining Dynamic & Static Scheduling in High-Level Synthesis. In Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (Seaside, CA, USA) (FPGA '20).
[9] Xilinx. 2021. Vitis High-Level Synthesis. Retrieved March 7, 2021 from https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html