

WIP: A Flexible Intermediate Language for High-Level Synthesis

Anonymous Author(s)

Abstract

High-level synthesis (HLS) compilers transform high-level, untimed programs into synthesizable RTL designs and have the potential to drastically improve the productivity of accelerator design. However, research on HLS is hindered by existing tools' monolithic integration: existing toolchains tightly couple a specific input language with large, mandatory transformation passes and target-specific output. We posit that the fundamental problem is the lack of a self-contained intermediate language (IL) that can capture both computational semantics and hardware-level resource constraints.

Futil is an in-progress IL that can represent programs at every stage in the chain of transformations from high-level specification to the low-level implementation. The key idea in Futil is a dual representation that captures both the *structure*, consisting of physical hardware resources and their interconnection graph, and the *control*, which orchestrates the structure over time to run a computation. Futil includes a framework for implementing modular compiler passes that transform higher-level control constructs into hardware structure. Through composition of passes, Futil can offer flexible compilation strategies suited to different input languages and different reconfigurable hardware targets. Futil aims to provide a robust and expressive foundation for experimenting with HLS in the same way LLVM does for traditional software compilers.

1 Introduction

High-level synthesis (HLS) compilers transform high-level, untimed programs into synthesizable RTL designs. Widespread adoption and research into HLS tools is a crucial ingredient in the development of reconfigurable accelerators to counteract the stagnation resulting from the wane of Moore's Law. However, research in HLS is hindered by the monolithic integration of the compilers and tooling. Standard HLS tools are intertwined with the semantics of one or two input languages like C++. A given HLS compiler typically targets only a single vendor's FPGA or a single ASIC toolflow. Finally, monolithic HLS toolchains prevent the development of modular, reusable passes that manipulate or optimize accelerator programs.

A key innovation that enables the shared infrastructure of software tools today is the development of intermediate languages (ILs) such as as LLVM [13] that can concisely

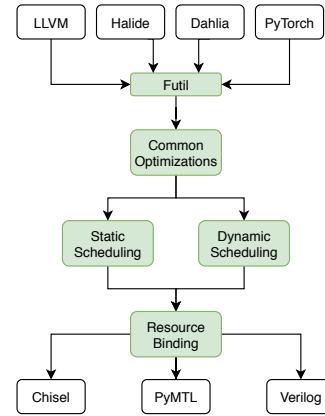


Figure 1. Futil (highlighted) separates microarchitectural decisions from source and target languages.

represent the constraints of target ISAs while also being a flexible frontend for many different languages. We posit that an IL for HLS compilation can enable similar reuse of tooling and infrastructure across many languages and backends.

We see the fundamental responsibilities of an HLS compiler as threefold: parallelization, resource binding, and cycle insertion. A traditional, monolithic HLS compiler intertwines all three responsibilities in a composite heuristic framework. *Parallelization* transforms a sequential program into a parallel schedule. HLS tools typically rely on standard conservative automatic parallelization techniques. During *resource binding*, the compiler maps logical instructions, such as add32, onto physical resources on the target fabric, such as adders. Finally, *cycle insertion* generates a finite state machine (FSM) that realizes the logical parallel schedule as physical, cycle-by-cycle timing. The challenging part of building a modular and extensible HLS compiler is cleanly separating these concerns without adversely affecting performance and area. For example, if the parallelization step attempts to maximize the throughput, the resource binding phase has to work harder to minimize resource conflicts.

Our main goal in the design of an IL for HLS is to enable modular passes to do parallelization, resource binding, and cycle insertion by transforming programs in the IL. We propose that such an IL should meet these criteria:

Self contained. An IL is a programming language that should capture the meaning of a program at any stage of compilation. Therefore, it should be possible to rigorously define a program's semantics without reference to the original input code, details of the passes, or *ad hoc* compiler data

111 structures. A self-contained IL allows for modular compiler
 112 design because the responsibility for a given pass can be com-
 113 pletely defined in terms of its input and output programs.

114 **Hardware aware.** An IL for HLS must represent the phys-
 115 ical constraints on timing and resources. This way, modular
 116 passes can optimize the timing behavior and resource usage
 117 of an accelerator design by transforming the IL program.
 118 When an IL is both self contained and hardware aware, the
 119 semantics of the IL can answer questions about the perfor-
 120 mance and area of a given program.

122 **Expressive.** An IL for HLS must be expressive enough to
 123 capture both the computational semantics of different high-
 124 level frontend languages as well as the resource and timing
 125 constraints of different targets, including commercial FPGAs
 126 and various flavors of CGRAs. An expressive IL can enable
 127 innovation in new high-level languages for accelerator de-
 128 sign that shed the legacy baggage of C [16, 6, 11], and it
 129 can facilitate novel reconfigurable hardware designs [17] by
 130 providing a starting point for their compiler toolchains.

132 Futil is an intermediate language for building extensible HLS
 133 compilers. Futil explicitly represents resource and timing
 134 decisions using a split representation that includes both static
 135 hardware *structure* and dynamic logical *control*. In addition
 136 to allowing common software optimizations used by HLS
 137 compilers, Futil toolchains can easily swap out hardware-
 138 focused cycle insertion and resource binding passes.

2 Related Work

141 Other HLS compilers also rely on intermediate representa-
 142 tions. The key difference in Futil is the explicit representation
 143 of hardware resources as a complement to imperative control
 144 flow. This section contrasts Futil with traditional software
 145 IRs and more recent languages that specifically target recon-
 146 figurable accelerators.

148 **LLVM and software IRs.** xPilot [4] and LegUp [2] are
 149 commercially successful HLS toolchains built on LLVM [13].
 150 Using LLVM allowed these tools to reuse complex software
 151 optimizations and generate timed RTL by writing monolithic
 152 compiler passes to perform cycle insertion and resource bind-
 153 ing. Some passes work by adding metadata to the LLVM
 154 program to encode hardware-level concerns like timing and
 155 resource binding. These metadata formats are undocumented
 156 internal data structures, however, and do not allow modu-
 157 lar passes to experiment with new mapping strategies. The
 158 goal of Futil is to expose these concerns in a language with
 159 self-contained semantics, making it easy to inspect and ma-
 160 nipulate the compiler’s hardware-level decisions.

161 **μ IR.** μ IR [19] is a recent proposal for a C-to-RTL com-
 162 piler that relies on a parallel extension to LLVM as an input.
 163 The μ IR compiler represents programs as a graph of

166 asynchronously communicating tasks which let it repre-
 167 sent forms of parallelism not manifest in traditional IRs like
 168 LLVM. Unlike Futil, it does not attempt to represent physi-
 169 cal resources. Frontends therefore cannot control resource
 170 mappings, and passes cannot manipulate the allocation of
 171 hardware resources to computations.

172 **HPVM.** The Heterogeneous Parallel Virtual Machine [12]
 173 is a new intermediate representation that targets a wide vari-
 174 ety of novel hardware targets, from multicores to GPUs
 175 and FPGAs. The key idea is to represent many forms of
 176 parallelism in the IR to enable efficient code generation on
 177 platforms that exploit parallelism in different ways. We see
 178 Futil as a potential *backend* for HPVM when targeting re-
 179 configurable hardware specifically. Unlike HPVM, Futil adds
 180 a mechanism for reasoning about the allocation of physical
 181 resources to exploit area-parallelism trade-offs.

182 **IRs for HDLs.** Modern hardware description languages
 183 such as Chisel [1, 9], PyMTL [15], and Magma [7, 5] include
 184 IRs for building pass-based hardware optimization frame-
 185 works, and LLHD [18] is a standalone IR designed to capture
 186 the semantics of traditional HDLs. These IRs are lower level
 187 than Futil and target optimization at the bit and wire level.
 188 We view these as target backends for Futil.

3 The Futil Language

191 This section introduces Futil, an IL that enables the design
 192 of extensible and modular HLS compilers. Futil programs
 193 are composed of *components*. Every component consists of a
 194 *structure* part and a *control* part. Structure instantiates sub-
 195 components and the data-flow connections between them,
 196 and control describes how the structure behaves over time.

197 The separation of structure and control is a key idea in
 198 Futil’s design. Structure lets Futil represent hardware-level
 199 concerns such as resource sharing and control enables rea-
 200 soning about a program’s computational semantics. Passes
 201 in a Futil-based compiler shift parts of the program from
 202 software-like control to hardware-like structure, eventually
 203 producing a mostly-structural program that closely corre-
 204 sponds to a hardware implementation.

205 We next describe the Futil language in more detail. Sec-
 206 tions 3.1 and 3.2 then describe how lowering and optimiza-
 207 tion passes work in a compiler based on Futil.

208 **Components.** A Futil program defines a component with
 209 the `define/component` syntax form. A component consists
 210 of a name, a list of named input ports and their bitwidths, a
 211 list of output ports, a structure list, and a control expression.
 212 The syntax looks like this:

```
(define/component component_name
  ([inputA 32] [inputB 1])
  ([output 32])
  ( /* structure */ )
  /* control */ )
```

221 Futil backends provide implementations for primitive com-
 222 ponents that other components can instantiate. Primitive
 223 components include adders, multiplexers, registers, mem-
 224 ories, etc. Backends also supply area, energy, and timing
 225 information for the primitives that can be used in passes.
 226

227 **Structure sub-language.** Futil components describe their
 228 static hardware structure as a graph where nodes are subcom-
 229 ponents and edges are wired connections. Subcomponents
 230 are declared with new. The -> statement connects ports be-
 231 tween component instances. The syntax (@ comp portA)
 232 references the port named portA on the component named
 233 comp. This example shows a structure with two subcom-
 234 ponents and two connections:

```
235 [new B (comp/memory 8)] // component instantiations
236 [new dot my/register]
237 [-> (@ B out) (@ dot in)] // port connections
238 [-> (@ dot out) (@ this out)]
```

239 The new statements can optionally provide parameters to
 240 components to specify properties like bitwidths or memory
 241 sizes. The keyword this in the last statement refers to the
 242 component currently being defined.

244 **Control sub-language.** The control sub-language in Fu-
 245 tilit orchestrates the behavior of the components instantiated
 246 in the structure. It resembles an ordinary imperative pro-
 247 gramming language augmented with parallelism.

248 The central statement that Futil control can execute is
 249 enable, which *activates* one or more structural components,
 250 running their respective computations:

```
251 (enable A reg0) // Execute A and allow writes to reg0
```

253 Futil provides two composition operators: par to execute
 254 components in parallel, and seq to execute components in
 255 sequence. A par or seq statement finishes executing when
 256 all sub-components are done.

```
257 (seq (enable A) (enable B) (enable C))
258 (par (enable A) (enable B) (enable C))
```

259 if and while statements allow expressing more complex
 260 control-flow.

```
261 (if (@ comp port)
262     (seq (enable A) ...))
263 (while (@ comp port)
264     (seq (enable A) ...))
```

266 The composition primitives (seq and par) in Futil give front-
 267 end compilers the ability to concisely express a rich class
 268 of *program schedules*, while the control-flow primitives (if
 269 and while) allow programmers to express high-level control
 270 in a similar fashion to high-level programming languages,
 271 making it easier to compile frontend languages into Futil.
 272 These high-level control statements are compiled away for
 273 Futil toolchain.

3.1 Compilation

273 Futil aims to enable a compiler to translate high-level pro-
 274 grams to low-level hardware implementations. High-level
 275

276 programs, early in the compiler pipeline, are control heavy
 277 while lower-level programs consist of more structure and
 278 less control. A purely structural program has Verilog-like
 279 semantics and admits straightforward translation to RTL. In
 280 this section, we demonstrate how Futil represents the tradi-
 281 tional scheduling and binding phases of an HLS compiler.

282 **Scheduling.** In an HLS compiler, the scheduling phase
 283 assigns each logical operation of a program to a specific
 284 clock cycle. The *control* language of each component in Futil
 285 represents a coarse-grained schedule; it describes a happens-
 286 before ordering of operations rather than a strict assignment
 287 of operations to clock cycles. Scheduling in Futil is the task
 288 of generating cycle-level timing for a component that im-
 289 plements its control description. Futil represents cycle-level
 290 timing with a *global schedule* with the following form:

```
(seq (enable A B) (enable C D) ...)
```

293 In a global schedule, operations in each enable correspond
 294 to actions for that clock cycle. Futil assigns each enable
 295 statement a precise latency by recursively computing the
 296 timing information of each sub-component.

297 While scheduling in traditional HLS compilers happens in
 298 a monolithic phase, Futil allows for the process of generating
 299 the global schedule to be broken up into several modular
 300 passes. For example, one pass could be responsible for re-
 301 placing while loops with equivalent structure and another
 302 pass could flatten nested seq / par constructs. This makes it
 303 easier to experiment with small changes to the scheduler.

305 **Binding.** The binding phase of an HLS compiler assigns
 306 physical resources to each logical resource, possibly reusing
 307 physical resources multiple times. Replacing this phase is
 308 challenging because it typically uses target-specific heuris-
 309 tics. The compiler implicitly maintains timing information
 310 and target specification to enable binding.

311 Since Futil directly represents resources and timing in-
 312 formation, binding is *just another optimization pass*. It can
 313 be implemented using small, modular passes that remove
 314 duplicate components, insert multiplexing logic, and mod-
 315 ify the control. For example, consider a program that uses
 316 multipliers A and B at two different times:

```
([new A (comp/mult 32)] // structure
 [new B (comp/mult 32)] ...)
 (seq (enable A) // control
      (enable B))
```

317 Since the multipliers execute in sequence, a compiler pass
 318 may decide to reuse the multiplier A and reduce the area of
 319 the final design by multiplexing the inputs and outputs of
 320 A:

```
([new A (comp/mult 32)] // structure
 [new M (comp/mux 32)] ...) // define new multiplexer
 (seq (enable A M) // control
      (enable A M))
```

331 Because resource binding is decoupled from the rest of the
332 compiler, experimenting with different binding strategies for
333 different targets is straightforward.

335 3.2 Optimization Passes

336 Through its explicit representation, Futil can represent both
337 traditional HLS optimizations such as loop unrolling as well
338 as timing and resource-directed optimizations.

Loop unrolling. Area-performance trade-offs such as loop unrolling are common in HLS programming. HLS loop unrolling (distinct from software loop unrolling) duplicates hardware to execute independent loop iterations in parallel, increasing throughput. Traditional HLS tools represent unrolling using `#pragma` annotations on C loops.

343 Futil can easily represent loop unrolling by explicitly making
344 copies of the loop structure and parallelizing the loop
345 control. Consider this Futil program:

```
349 ([new A (comp/memory 8)] // structure
350 [new m0 (comp/mult 32)]
351 [new i0 (comp/iter 0 1 8)])
352 (while (@ i0 stop)           // control
353   (enable A m0))
```

353 Unrolling the loop once results in code like this:
354

```
355 ([new A0 (comp/memory 4)]  [new A1 (comp/memory 4)]  
356  [new m0 (comp/mult 32)]  [new m1 (comp/mult 32)]  
357  [new i0 (comp/iter 0 2 8)] [new i1 (comp/iter 1 2 9)])  
358 (par (while (@ i0 stop) (enable A0 m0))  
      (while (@ i1 stop) (enable A1 m1)))
```

360 *Operator chaining.* *Operator chaining* is an optimization
361 that improves the overall latency of a design by scheduling
362 sequences of operations into a single clock cycle if the latency
363 of the sequence of operations is shorter than the estimated
364 cycle length. In Futil, this can be expressed as a control
365 transformation. Programs of the following form:

366 (seq (enable A) (enable B) (enable C) ...)

367 could be translated into:

368 (seq (enable A B) (enable C))

370 **371** **372** **373** **374** **375** **376** **377** *Software-style optimizations.* Futil enables classical compiler optimizations to be performed on the control language. Performing these optimizations in Futil, rather than in a software IR, allows these optimizations to compose cleanly with hardware optimizations. For example, classic loop-invariant code motion (LICM) lifts statements out of loop when their behavior is the same on every iteration, as in this Futil loop:

```
378  (while (@ io out)
379    (seq (enable A B mult c) // c = A * B
380          (enable x c))) // x = x * c
```

381 Here, c is recomputed every loop iteration but its value never
382 changes. A Futil LICM pass results in code like this:

```
383  (seq (enable A B mult c)  // c = A * B
384    (while (@ io out)
```

```
(seq (enable x c) /* x = x * c */ )))
```

4 Future Directions

Futil aims to make the development of HLS compilers flexible and modular. Rapid iteration of compiler technologies is a critical ingredient in widespread adoption of reconfigurable accelerators. We enumerate opportunities to build on Futil to enable future research.

Latency-insensitive design. Dynamic scheduling [10] is a scheduling strategy that leverages latency-insensitive interfaces to improve the execution time of designs that extensively use data-dependent control. Currently, Futil can represent static schedules but not dynamic ones. We plan to augment Futil with variants of `enable` that can wait asynchronously for a component to signal its completion.

Verified compilation. The last decade has produced breakthroughs in formal verification of software compilers. Futil's pass-based design will enable easier verification of HLS compilers. CompCert [14] and similar verified compilers use refinement to build up a proof of correctness for the compiler from modular proofs that individual passes preserve the semantics of the program. A self-contained IL semantics is a critical first step toward formulating a correctness theorem for individual compiler passes. Because Futil modularizes complex passes such as scheduling and binding, it will allow verification efforts to scale.

Hardware backends. Futil currently generates accelerators for commercial FPGAs. We want to extend Futil to target other hardware backends such as emerging coarse-grained reconfigurable arrays (CGRAs) and real silicon via ASIC toolchains. The challenge in targeting CGRAs is that, unlike FPGAs, their design and capabilities can vary wildly: some use static scheduling while some are purely dynamically scheduled [8]; each CGRA bakes different logic into its processing elements [3]; and each CGRA can distribute on-chip memories differently [17]. Meanwhile, ASIC design offers total flexibility in the instantiation of structural resources. We expect Futil’s modular pass framework to enable us to add ASIC- and CGRA-specific optimization and binding passes, making it easier to develop toolchains for these technologies.

New design languages. Traditional HLS relies on C and C-like input languages, but repurposing a legacy software language introduces a semantic gap between the programmer’s view and the compiler’s output. We see an opportunity to design new HLS languages that more faithfully reflect the constraints of accelerator implementation while still offering high-level algorithmic semantics. Futil’s representation of structural resources and hardware timing will let novel language frontends exert more control over the hardware they generate without resorting to generating Verilog.

441
442 **References**

443 [1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew
444 Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović.
445 2012. Chisel: constructing hardware in a Scala embedded language. In
446 *Design Automation Conference (DAC)*.

447 [2] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed
448 Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Cza-
449 jkowski. 2011. LegUp: high-level synthesis for FPGA-based pro-
450 cessor/accelerator systems. In *International Symposium on Field-
451 Programmable Gate Arrays (FPGA)*.

452 [3] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee
453 Kim, Yuko Hara-Azumi, and Jason Helge Anderson. 2017. CGRA-ME: A
454 unified framework for CGRA modelling and exploration. *International
455 Conference on Application-specific Systems, Architectures and Processors
456 (ASAP)* (2017).

457 [4] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. 2006. Platform-
458 Based Behavior-Level and System-Level Synthesis. In *International
459 SoC Conference*.

460 [5] Ross Daly, Lenny Truong, and Pat Hanrahan. 2018. Invoking and
461 Linking Generators from Multiple Hardware Languages using CoreIR.
462 In *Workshop on Open-Source EDA Technology (WOSET)*.

463 [6] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly,
464 Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat
465 Hanrahan. 2020. Type-Directed Scheduling of Streaming Accelerators.
466 In *ACM SIGPLAN Conference on Programming Language Design and
467 Implementation (PLDI)*.

468 [7] Pat Hanrahan. [n.d.]. Magma. <https://github.com/phanrahan/magma>.

469 [8] Yuanjie Huang, Paolo lenne, Olivier Temam, Yunji Chen, and Chengy-
470 ong Wu. 2013. Elastic CGRAs. In *International Symposium on Field-
471 Programmable Gate Arrays (FPGA)*.

472 [9] Adam M. Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angu Wang,
473 Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim
474 Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL ground:
475 Hardware construction languages, compiler frameworks, and trans-
476 formations. In *International Conference on Computer-Aided Design
477 (ICCAD)*.

478 [10] Lana Josipović, Radhika Ghosal, and Paolo lenne. 2018. Dy-
479 namically Scheduled High-Level Synthesis. In *International Symposium
480 on Field-Programmable Gate Arrays (FPGA)*.

481 [11] David Koepfinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang,
482 Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram,
483 Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: a language and
484 compiler for application accelerators. In *ACM SIGPLAN Conference on
485 Programming Language Design and Implementation (PLDI)*.

486 [12] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Ko-
487 muravelli, Vikram Adve, and Sarita Adve. 2018. HPVM: Heterogeneous
488 Parallel Virtual Machine. In *PPoPP*.

489 [13] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Frame-
490 work for Lifelong Program Analysis & Transformation. In *International
491 Symposium on Code Generation and Optimization (CGO)*.

492 [14] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Com-
493 munications of the ACM (CACM)* (July 2009), 107–115.

494 [15] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL:
495 A Unified Framework for Vertically Integrated Computer Architecture
496 Research. In *IEEE/ACM International Symposium on Microarchitecture
497 (MICRO)*.

498 [16] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore
499 Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang.
500 2020. Predictable Accelerator Design with Time-Sensitive Affine Types.
501 In *ACM SIGPLAN Conference on Programming Language Design and
502 Implementation (PLDI)*.

503 [17] Raghu Prabhakar, Yaqi Zhang, David Koepfinger, Matthew Feldman,
504 Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christoforos E. Kozyrakis,
505 and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for
506 parallel patterns. In *International Symposium on Computer Architecture
507 (ISCA)*.

508 [18] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020.
509 LLHD: A Multi-Level Intermediate Representation for Hardware De-
510 scription Languages. In *ACM SIGPLAN Conference on Programming
511 Language Design and Implementation (PLDI)*.

512 [19] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha,
513 Tony Nowatzki, and Arvindh Shriraman. 2019. μ IR: An Intermediate
514 Representation for Transforming and Optimizing the Microarchitec-
515 ture of Application Accelerators. In *IEEE/ACM International Sympo-
516 sium on Microarchitecture (MICRO)*.

517

518

519

520

521

522

523

524

525

526

527

528

529

530

531

532

533

534

535

536

537

538

539

540

541

542

543

544

545

546

547

548

549

550