Check out my sabbatical project

Jonathan Balkind
University of California, Santa Barbara
USA

Introduction

This paper introduces a compiler from ISA specifications to processor implementations. We extract from per-instruction specifications to a multi-level IR known as a resource graph, which, crucially, can represent both ISA-level constructs and microarchitectural implementations. Key to our approach is the decoupling of the "what", "where", and "how" of microarchitectural optimisation. Leveraging our resource graph IR, we provide a suite of optimisations as algorithms which can automatically transform graphs when provided with the "where" by the designer. The designer thus engages in a directed compilation process where they iteratively refine their implementation until it meets their needs, as shown in Figure 1.

Our tool, `isacomp`, provides a complete processor design methodology which starts from ISA-level specifications and produces RTL. At each step, the designer can observe a resource graph and iterate or revert back to a previous iteration. While a general algorithm cannot be given for many highly-specific microarchitectural optimisations, we provide a library of common ones.

We provide frontends for our .isa DSL plus ILA, SAIL, and annotated Verilog/VHDL RTL [1, 3–5, 11, 13, 14]. To provide confidence in our optimisations' correctness, we provide an SMT verification flow to prove equality of behaviour between two resource graphs, e.g. representing the "before" and "after" of an optimisation. This can also verify that an instruction is correctly implemented in a design. For egress, `isacomp` has backends for Verilog, VHDL, CIRCT, PyRTL, Chisel, Hardcaml, RTLIL, Clash, and C++ simulation.

The .isa DSL

We define the new .isa DSL, in which the user writes ISA specifications as simple, imperative programs. This approach resembles ILA and SAIL but focuses on per-instruction specifications. We have specified the whole or large subsets of (300K+ SLOC) ISAs including: RISC-V32IMAFDCV_Zicsr_Zicond_Zb*, RISC-V64IMAFDC, 8086, i486, 6502, M68K, z80, SPARCv8, Alpha64, MIPS32, OpenRISC 32 ORBIS, SH-2/J-2, SH-4, PDP-11/70, MicroVAX, PA-RISC 1.1, PA-RISC 2.0, MMIX, ARM7TDMI, subleq, and Zarf. Figure 2 shows the .isa DSL specification for the RISCV-32 add instruction.

We distinguish ISA state with the `isa_state` keyword. In principle, ISA state is any state that is persistent between instructions, which may not be programmer-visible. Ultimately, all state has a stepwise transfer function in the specification, whether it is the program counter, register file, or something else. This enables us to apply some of our optimisations more generically and to handle more ISAs. It also means that some of our optimisations require arguments that one may consider extraneous if one only cares about a particular class of ISAs (e.g. RISC), such as explicitly providing the PC node as an input to the branch prediction optimisation.

`assume` statements are used to represent decode conditions and differentiate instructions. This is used for instruction merging, where it will create the decode logic for each instruction in the merged design. It is also used for SMT property verification to check the case entailed by a specific instruction.

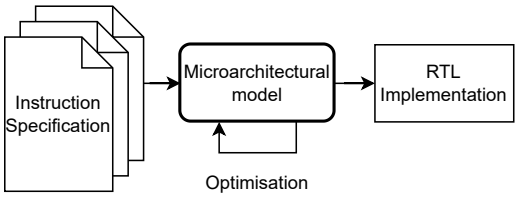

Figure 1: Expected process for an ISA compiler

```
isa_state pc[31:0], rf[31:0][4:0];
mem umem[12:0];
var instr[31:0], rs1[4:0], rs2[4:0], rd[4:0];
var r1[31:0], r2[31:0], res[31:0];

instr = umem[pc];
assume instr[6:0] == 0b0110011;
assume instr[14:12] == 0x0;
assume instr[31:25] == 0x00;
pc = pc + 4;
rs1 = instr[19:15];
rs2 = instr[24:20];
rd = instr[11:7];
r1 = rf[rs1];
r2 = rf[rs2];
res = r1 + r2;
rf[rd] when (rd != 0) = res;
```

Figure 2: The .isa code for the RISC-V 32 add instruction.

Our `mem` construct infers memory access width per-instruction, then uses it at instruction merging time to create byte enables. The "memory port width setting" optimisation can insert either multiple memory ports to handle unaligned accesses or serialisation logic.

The `aspect` statement is used to handle ISA features that span across instructions. An aspect can take priority in updating ISA state with the `gates` keyword, e.g. for exceptions whose PC update should take precedence (as seen in Listing 1 and Listing 2).

```
isa_state pc[31:0];
isa_state rf[31:0][4:0];
aspect interrupts.isa gates pc;
// Normal instruction execution
pc = pc + 4;
```

Listing 1: Main file which includes the aspect

```
ext isa_state interrupt_vector[31:0];
sensitive isa_state taking_interrupt;
pc when (taking_interrupt) = interrupt_vector;
```

Listing 2: Aspect file (interrupts.isa)

Resource Graphs

Our resource graph IR describes microarchitectural models and includes ISA-level details to make optimisations feasible. Each node represents a functional component (with or without state). Directed edges indicate the output of one node flowing into the other. Nodes can have associated latency in multi-cycle designs.

Extracting a Single-Cycle Processor. Architectural specifications must be implementable as microarchitectural models and HDL-level implementations or else we lack machines at all. We show that ISA specifications can thus be straightforwardly "unrolled" into single-cycle implementations in resource graph form. These lack real world performance and may use many resources, but they provide a semantics preserving source from which one can optimise, particularly since ISA state information is known. Importantly, from here, many optimisations entail reducing intra-instruction parallelism. For example, a SIMD add would be completely unrolled here, and many optimisations would instead re-serialise it.

The resource graph represents the original .isa code as a flattened dataflow graph. Loops are bounded in the DSL and unrolled in the resource graph. Multiple assignments to a variable or isa_state are split into separate nodes and later ones override earlier ones (intermediate values/nodes persist). ISA state reads and writes are split into separate nodes.

`isacomp` is able to extract a conforming single-cycle implementation from several languages. However, we are clearly not the first to have made progress on this problem; other precursors exist [2, 7, 9].

Merging Resource Graphs. Our per-instruction approach means that we first extract each instruction into its own resource graph. We thus require a means to merge together resource graphs for each instruction, with ISA state considered global. The naive approach is to make all nodes instruction-specific and simply mux between them e.g. if merging AND with OR, the logical operations are instruction-specific. Conflict resolution then inserts muxes to choose between the instructions where they output to the same node. The per-instruction mux select signals (decode logic) come from each instruction's `assume` statements. This naive approach is massively redundant: the common 37-instruction subset of rv32i can be around 1000 nodes. Our optimised approach performs structure-aware merging and enables "partial sharing" in groups where subsets of the instructions have the same structure. For the same rv32i subset, we reduce the resource graph size to around 240 nodes.

Optimisations as Algorithms

Note that "microarchitectural optimisation" does not refer to optimising for a single criterion. Optimisations with directly opposing outcomes (e.g. serialising vs parallelising) are both optimisations.

Serialisation and Parallelisation. Serialization is a common microarchitectural strategy to save area. A bit-serial processor completes its ALU operations one bit at a time, but less aggressive serialisation is used even for high performance use cases, for example AMD Zen 4's AVX-512 with its 256 bit datapath [8]. Our optimisation enables the designer to select an operation node inside the resource graph to be serialised and pick the new operation width. `isacomp` will then share a narrower operation node, insert a microarchitectural counter, select sub-operands and aggregate the sub-results. The designer can then select this logic and iteratively narrow, widen, or even eliminate it by fully parallelizing.

Resource Sharing and Duplication. Our implementation enables the designer to select a set of operation nodes and transforms the resource graph to share the underlying resource over multiple cycles. It will insert ISA state write enable logic to only update ISA state nodes once per instruction. The optimisation also enables the user to re-duplicate resources that were previously shared. Memory ports can be shared through a similar optimisation.

Pipelining is an early exercise given to novice computer architects. Tasked with the **what**, they must determine **where** to insert their pipeline registers, and must work through **how** to handle the resulting issues that arise from their decision. Our multi-stage manual pipelining algorithm enables the designer to choose the edges in the resource graph at which to insert pipeline registers. We then identify the ISA state read and write locations and algorithmically identify the inter-instruction dependencies that could result in hazards. We then expose a second phase of **what** and **where** to the designer: which hazard resolution strategy (stalling or forwarding) to apply for which hazards. This automation enabled us to write a single script that allows the user to generate a 2/3/4/5 stage pipeline with either hazard resolution strategy. We have such scripts for RV32I, RV64I, OpenPOWER SFS, and MIPS32 from which the pipelined implementations can pass test suites and benchmarks (these are used for CI).

Other Optimisations we have implemented in the framework include operation latency setting (static or dynamic), branch prediction, dual-issue superscalar in-order pipelining, cache insertion, memory externalisation, retiming, and TLB+PTW creation. We summarise the proof-of-concept optimisations currently available in `isacomp` in Table 1. Each has been exercised on at least one ISA; however, we do not claim completeness or correctness guarantees for all of them. The SMT verification framework can be used to check equivalence before and after an optimisation is applied.

Table 1: Proof-of-concept optimisations in `isacomp`. "Designer input" describes the "where" provided by the designer.

Optimisation	Designer Input
Serialisation	Node(s), target width
Parallelisation	Node(s), target width
Resource sharing	Set of nodes to share
Resource duplication	Shared node to split
Multi-stage pipelining	Edges to cut
Hazard resolution	Strategy (stall/bypass)
Unpipelining	(automatic)
Branch prediction	PC node, mux, strategy
Branch target buffer	PC node, entries, associativity
Return address stack	Call/return nodes
Operation latency setting	Node, latency (or variable)
Memory port width	Memory, width
Memory port sharing	Memory, port count
Memory externalisation	Memory node
Cache insertion	Memory, cache params
TLB + page table walker	Memory, TLB params
Dual-issue superscalar	Pair of resource graphs
Architectural retiming	Node to pre-calculate
Retiming	Register to move, direction
Register renaming	Register file, physical register count
Instruction buffer	Memory, PC, instr. width nodes, buf. size

Designer Guidance

The designer directs compilation through Python scripts or the `isacomp` GUI. In either case, the workflow follows the same pattern: ingress specifications, iteratively apply optimisations by selecting the "what" and "where" (with `isacomp` handling the "how"), then egress to RTL. Listing 3 shows a condensed version of a script used to create a pipelined RV32I processor.

```
# Ingress: parse and merge .isa files
merger = ResourceGraphMerger()
for name, path in rv32i_instructions.items():
    merger.add_instruction(name, path)
graph = merger.merge_graphs()

# Designer selects edges to cut
cuts['Decode']  = [find_edge('umem[pc]_rd','instr')]
cuts['Execute'] = [find_edge('rf[rs1]_rd', 'r1'),
                   find_edge('rf[rs2]_rd', 'r2')]
# ... (memory, writeback cuts similar)

# Framework inserts pipeline registers, auto-detects
# hazards, resolves them, adds prediction
transform = PipelineTransform(graph)
transform.add_multiple_pipeline_stages(
    [{'edges': cuts[s], 'stage_index': i}
     for i, s in enumerate(stages)])
hazards = HazardAnalyzer(graph).analyze_hazards()
HazardResolver(graph).resolve_all_hazards(hazards)
apply_branch_prediction_not_taken(graph,
  pc_node_id='pc_wr', not_taken_input_id='pc_add_4')

# Egress
VerilogGenerator(graph).generate("rv32i_5stage.v")
```

Listing 3: Condensed designer workflow for a RV32I pipeline.

The designer's edge selections (the "where") determine the pipeline structure. The framework then identifies inter-instruction dependencies that could cause hazards and inserts resolution logic (the "how"). Changing from a 5-stage to a 3-stage pipeline requires only changing which edges are cut. Hazard analysis and resolution re-run automatically.

A simpler example is operation serialisation, where the designer selects a specific operation node and a target width:

```
spec = SerializationSpec(operation_id='(r1 + r2)', # where
                         granularity_bits=8)
apply_operation_serialization(graph, [spec])
```

Listing 4: Serialising a 32-bit add to 8-bit granularity.

The framework replaces the 32-bit adder with an 8-bit adder, a microarchitectural counter, sub-operand selection muxes, and a result accumulator, completing the operation over 4 cycles (Figure 3). ISA state write enables are gated so that architectural state updates only on the final cycle. The designer can later re-widen or fully parallelise the operation by changing the granularity.

Validation

We have validated our implementation in a variety of ways. We have leveraged riscv-tests and created similar test suites for a number of other ISAs to check basic functionality. With an optimisation to externalise memory outside of the processor, we can place the processor into a larger SoC and perform further end-to-end validation. At present, nine of our single-cycle machines (ISA-compatible specs

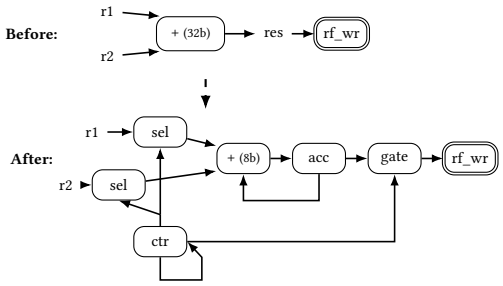

Figure 3: Serialising a 32-bit add to 8-bit granularity. The resource graph transformation replaces the wide adder (top) with a narrow adder, sub-operand selectors, a cycle counter, and a result accumulator (bottom). The ISA state write (double border) is gated to fire only when the counter signals completion.

for SuperH-2/J-2, SuperH-4, RV32IMA, RV64IMA, OpenPOWER LCS, Alpha EV56, PA-RISC 1.1 (32b), PA-RISC 2.0 (64b), OpenRISC ORBIS32) extracted to Verilog successfully boot Linux in Verilator. Several more are advancing along similar lines, with significant Linux progress shown for CISC ISAs M68K and i486.

Related Work and Discussion

We draw parallels with Halide and Exo [6, 10], which separate the *algorithm* (what to compute) from the *schedule* (how to compute it), letting programmers explore implementation strategies without rewriting their program. Similarly, `isacomp` separates the ISA specification from the schedule of microarchitectural optimisations. Like PDL [12], our targets include pipelined processors, but where PDL requires the designer to rewrite their program to change the pipeline structure, `isacomp`'s optimisations are parameterised transformations on the resource graph: Changing from a 3-stage to a 5-stage pipeline means changing which edges to cut, not rewriting the design. The designer's optimisation scripts can be parameterised across stage counts and ISAs; our CI reuses a single per-ISA script for 2- to 5-stage pipelines across four ISAs, each using the same pipelining and hazard resolution infrastructure. Adding ISA extensions (e.g. new custom instructions) to an already-optimised design requires re-running the merge and existing script, though more complex additions may require further attention.

Acknowledgements

As noted, this project started in earnest as a sabbatical project during Fall 2025. To within rounding error, all code for the project (200K+ SLOC of Python, 50K+ SLOC of Rust, 300K+ SLOC of .isa DSL, 40K+ SLOC of assembly tests) was written by Claude at the direction of the author. Claude Max subscription was generously provided by Anthropic with particular thanks to Siddharth Mishra-Sharma. Major thanks go to Zachary D. Sisco, Harlan Kringen, Jingtao Xia, Sijie Kong, Naomi Rehman, Brian Li, Sarkis Ter-Martirosyan, Luke Herbelin, Joshua Gray, and Katie Lim for providing crucial input throughout and letting me flip the meeting by having me presenting results to you instead of vice versa.

References

[1] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*. doi:10.1145/3290384 Proc. ACM Program. Lang. 3, POPL, Article 71.

[2] Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Pratap Singh, Andy Wright, and Adam Chlipala. 2023. Flexible Instruction-Set Semantics via Abstract Monads (Experience Report). 7, ICFP, Article 192 (Aug. 2023), 17 pages. doi:10.1145/3607833

[3] Bo-Yuan Huang, Sayak Ray, Aarti Gupta, Jason M. Fung, and Sharad Malik. 2018. Formal Security Verification of Concurrent Firmware in SoCs using Instruction-Level Abstraction for Hardware. In *Proc. Design Automation Conference*. 91.

[4] Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik. 2019. ILAng: A Modeling and Verification Platform for SoCs using Instruction-Level Abstractions. In *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*. 351–357. doi:10.1007/978-3-030-17462-0{_}21

[5] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *ACM Trans. Des. Autom. Electron. Syst.* 24, 1, Article 10 (dec 2018), 24 pages. doi:10.1145/3282444

[6] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. doi:10.1145/3519939.3523446

[7] Harlan Kringen, Zachary Sisco, Timothy Sherwood Jonathan Balkind, and Ben Hardekopf. 2023. Semi-Automated Translation of a Formal ISA Specification to Hardware. (2023). https://pldi23.sigplan.org/details/plarch-2023-papers/17/Semi-Automated-Translation-of-a-Formal-ISA-Specification-to-Hardware PLDI 2023 (PLARCH Series).

[8] Michael Larabel. 2022. *AMD Zen 4 AVX-512 Performance Analysis On The Ryzen 9 7950X Review*. https://www.phoronix.com/review/amd-zen4-avx512

[9] Louis-Emile Ploix, Alasdair Armstrong, Tom Melham, Ray Lin, Haolong Wang, and Anastasia Courtney. 2025. Comprehensive Formal Verification of Observational Correctness for the CHERIoT-Ibex Processor. arXiv:2502.04738 [cs.AR] https://arxiv.org/abs/2502.04738

[10] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. doi:10.1145/2491956.2462176

[11] Pramod Subramanyan, Yakir Vizel, Sayak Ray, and Sharad Malik. 2017. Template-based Synthesis of Instruction-Level Abstractions for SoC Verification. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design*. 160–167. doi:10.1109/FMCAD.2015.7542266

[12] Drew Zagieboylo, Charles Sherk, Gookwon Edward Suh, and Andrew C. Myers. 2022. PDL: a high-level hardware design language for pipelined processors. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 719–732. doi:10.1145/3519939.3523455

[13] Hongce Zhang, Caroline Trippel, Yatin A Manerkar, Aarti Gupta, Margaret Martonosi, and Sharad Malik. 2018. ILA-MCM: Integrating Memory Consistency Models with Instruction-Level Abstractions for Heterogeneous System-on-Chip Verification. In *Proc. Conf. Formal Methods in Computer-Aided Design*. 1–10.

[14] Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, and Sharad Malik. 2020. Synthesizing Environment Invariants for Modular Hardware Verification. In *Verification, Model Checking, and Abstract Interpretation*, Dirk Beyer and Damien Zufferey (Eds.). Springer International Publishing, Cham, 202–225.