Iron: selectively turn RISC-V binaries into hardware co-processors.

Sylvain Lefebvre
Université de Lorraine, CNRS, Inria, LORIA

Abstract

We explore whether already compiled RISC-V binaries can be selectively turned into hardware. From a list of function symbols selected by the designer, Iron *automatically generates a hardware design consisting of a SOC and RISC-V CPU, together with co-processors synthesized from the selected functions machine code. Whenever the CPU reaches the address of one of these functions, the corresponding co-processor executes instead. This makes the change from the software to the hardware versions completely seamless, enabling client-side hardware acceleration, possibly depending on available hardware resources (e.g. various FPGAs).*

RISC-V is particularly well suited to this endeavor thanks to its wide adoption, reduced instruction set, many registers and clean open-source ISA. We discuss a proof of concept, the encountered challenges and the exciting venues for future work. Iron *is an open-source software.*

Introduction

Opening the door of hardware design to software developers has become a key area of focus. This stems from the need to make hardware design more accessible, allowing the much larger pool of software developers to tap in the potential of FPGA and ASIC design [1].

To this effect, huge efforts have been devoted to design and implement High Level Synthesis tools [2], capable of taking source code in e.g. the C [3, 4, 5] or OpenCL languages [6] – to name only a couple – and turn it into an efficient design. The advantage of working from the high level language description is that many compiler-level optimizations are possible, as language constructs are explicitly available during synthesis, facilitating loop unrolling, pipelining and register allocation. For instance, *LegUp* [3] tightly integrates within the LLVM framework, to leverage its intermediate representation for analysis and synthesis.

There are however downsides to working from high level languages. First and foremost, designs have to be rebuilt from source, and thus the approach can only apply where source code is available. The tools are intrinsically linked to the original language and would not easily translate to other languages, even though the LLVM framework does mitigate this. Finally, the advantage of being able to enrich the original language with novel keywords also turns into less portable code that may no longer compile as pure software.

In this work we explore a somewhat unusual point of view and propose to generate hardware co-processors out of already compiled RISC-V binaries. At first sight this seems like a huge penalty, since a compiled binary carries little information in terms of the high level language constructs that produced it.

Yet, we seek after a very specific advantage: Working from a compiled binary means that the software to hardware step can work on *any* ELF executable, regardless of source code availability and regardless of the initial choice of language. Binaries can be specially re-optimized to benefit from local hardware resources – client side – possibly adjusting the mobilized hardware resources based on the actual usage. Furthermore, improvements to the tool can be immediately re-applied to existing binaries in the field.

The core of our approach is a compiler that takes as input an ELF binary, analyzes its control flow, register usage and stack usage, and generates HDL code (Verilog through Silice [7]). The output is a SOC embedding a RISC-V CPU and all generated co-processors, with the CPU seamlessly executing the hardware counterparts of the functions upon calls – an execution model overall similar to *LegUp*.

Methodology

Our approach starts from a RISC-V binary (ELF format) for a RV32I target. We assume the SOC is built solely around BRAM. The main steps of our approach are to first *deconstruct* the binary, reverting it in a form that facilitates optimizations on hardware, in particular re-scheduling and register optimization. During this **analysis step**, we perform:

Control flow graph analysis. We extract of control flow graph (CFG) from the binary, gathering successive instructions in blocks such that 1) a new block starts if it is a target of a jump and 2) a block ends on a jumping instruction (JAL, JALR, conditional branches). Under this form, each block can trivially be turned into a stateless block of combinational logic, with a global finite state machine (FSM) implementing the control flow. However, this FSM still has the many states of the binary program.

De-branching. We perform a *de-branch* step, which objective is to reduce the number of node (states) in

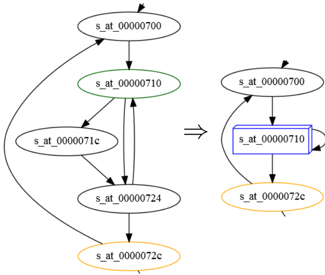

Figure 1: *Collapse performed during de-branch, where nodes 0710, 071c, 724 are collapsed into a pipelined loop. Green indicates a load is present, orange indicates both a store and load.*

the CFG, collapsing them into larger combinational blocks. Intuitively, a conditional branch is a if/else construct where the *if* jumps and *else* goes to next by program counter. In many places we can collapse the states being reached directly into a combinational *if/else*. A non-trivial collapse is illustrated in Figure 1. This step effectively flattens the CFG into combinational logic as much as possible. *De-branch* is an iterative process greedily applied until no further collapses can be performed.

Re-reg. Given the flattened CFG we perform a register separation pass. In each instruction `rd`, `rs1` and `rs2` are replaced by temporary registers within a block, based on a data dependency analysis, such that no combinational cycle is produced. The original registers `x0-x31` are updated only when no longer needed within the combinational block, before jumping to any successor block. This results in per-instruction *register aliasing*, which is a list of register assignments to perform before and after each instruction.

De-stack. On non-recursive functions, we replace `s0-s11` stack push/pop by writes/reads to registers.

We are now ready to enter to **design step**. To generate the Verilog we traverse the CFG blocks, producing the global state machine and writing the instructions and register aliasing assignments. Instructions are turned into Verilog expressions, with trivial optimizations whenever possible. This step has two refinements:

Memory scheduling. We assume execution on BRAM. We have to ensure that loads/stores are scheduled in non-conflicting states. We insert additional states before any load/store following a preceding one in the same block. Note that no additional registers need to be introduced: temporary registers added during *re-reg* are automatically promoted into actual flip-flops as required by the state split.

Pipelining. We identify simple opportunities for pipelining, in particular blocks looping on themselves that contain a single load (Figure 1, blue box). These are turned into two-stages pipelines, such that the entire loop executes in one cycle.

Results and limitations

We compare execution in simulation using *verilator*, in terms of equivalent instruction per cycles (eqIPC).

case	eq IPC	fmax MHz	LUTs	description
bubble sort	5.0	61	4484	A simple bubble sort.
factorial	4.9	64	3767	Computing factorial n.
rotating xor	7.9	64	4683	Rotating xor pattern on screen.
sqrt(int)	4.2	54	7101	Per pixel to draw circles.
F.Bellards Pi	5.0	(*)	(*)	Decimals of Pi (n=5).

Table 1: *CPU only version:* 3151 *LUTs at* 63 *MHz. Compiled for RV32I: mul, div and fp ops emulated.*
(*) simulation only.

We compile with `gcc -O3`. We first run the CPU-only version to count the number of retired instructions. We synthesize onto hardware on an `ULX3S` 85F board with `yosys` [8] and `nextpnr` [9]. We report LUT and fmax between software/hardware. Results are in Table 1.

Limitations. Clearly there are many more opportunities in *de-branch*, most importantly allowing nodes to be duplicated. Our pipelining is preliminary, and exciting opportunities exist to relax fmax and improve throughput with auto-pipelining [10]. We partially support `JALR` through heuristic detection of jump tables (this is used in the software floats, for instance). Arbitrary JALR would otherwise prevent generating hardware ; we did not observe any such case in our tests from `gcc`.

Source code. Iron and its source code can be found at `https://github.com/sylefeb/iron`.

References

[1] S. Lahti et al. "Are We There Yet? A Study on the State of High-Level Synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.5 (2019), pp. 898–911.

[2] D.F. Bacon, R. Rabbah, and S. Shukla. "FPGA Programming for the Masses". In: *Communications of the ACM* 56.4 (2013), pp. 56–63.

[3] A. Canis et al. "LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems". In: *ACM Transactions on Embedded Computer Systems* 13.2 (2013).

[4] J. Cong et al. "High-Level Synthesis for FPGAs: From Prototyping to Deployment". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.4 (2011), pp. 473–491.

[5] J. Villarreal et al. "Designing Modular Hardware Accelerators in C with ROCCC 2.0". In: *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. 2010, pp. 127–134.

[6] A. Baker. *Custom Hardware State-Machines and Datapaths - Using LLVM to Generate FPGA Accelerators*. 2014.

[7] S. Lefebvre. *Silice*. `https://github.com/sylefeb/silice/`.

[8] C. Wolf. *Yosys Open SYnthesis Suite*. `https://yosyshq.net/yosys/`.

[9] D. Shah et al. "Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs". In: *FCCM*. 2019, pp. 1–4.

[10] J. Kemmerer. *PipelineC*. `https://github.com/JulianKemmerer/PipelineC`.