Hochschule
München
University of
Applied Sciences

Identifying Code Size Optimization Opportunities in Embedded Systems

FPGA Ignite 2024

Andreas Hager-Clukas, Stefan Wallentowitz

AEMY @ Hochschule München

AEMY Lab at Hochschule München

- Research group at Hochschule München University of Applied Sciences
- Vision: Build s**A**fe s**E**cure s**M**art s**Y**stems for the future
- All open source

RISC-V ISA Extensions	Open Source Chip Design Tools
WebAssembly for Embedded	Formal Methods for Embedded Systems

Open Positions @ AEMY

PhD Students

- RISC-V Instruction Set Extensions for Bytecode Virtualization
- RISC-V Instruction Set Extensions for Deeply Embedded DSP Algorithms
- Open Source Tooling for Automated Hardware-Software Co-Exploration of Fault Robustness
- Improving Inter-Tool Compatibility of Open Source EDA Tools (CIRCT, Verilator, Yosys)
- LTL Verification Support in CIRCT (LLVM framework) and Automation for ISA Extensions
- later this year: Open Source Formally Verified WebAssembly Interpreter in Rust

PostDoc

Looking for someone to help me around RISC-V instruction set extensions and open source EDA tools

Follow-up from yesterday

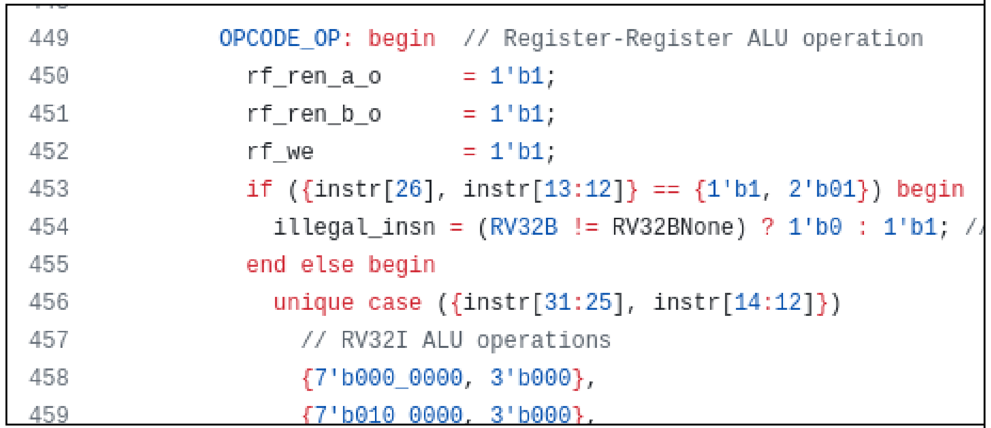
- Completed complex calculation unit (add and conjugation)
- cocotb test for complex calculation unit: randomization was missing
- Started implementation of the coprocessor unit: not tested
- cocotb tests started
- Now: brief review of complete project
- Find on Github: `https://github.com/hm-aemy/cvxif-tutorial`

Extending Cores

Just a very rough overview about alternatives to integrate new instructions

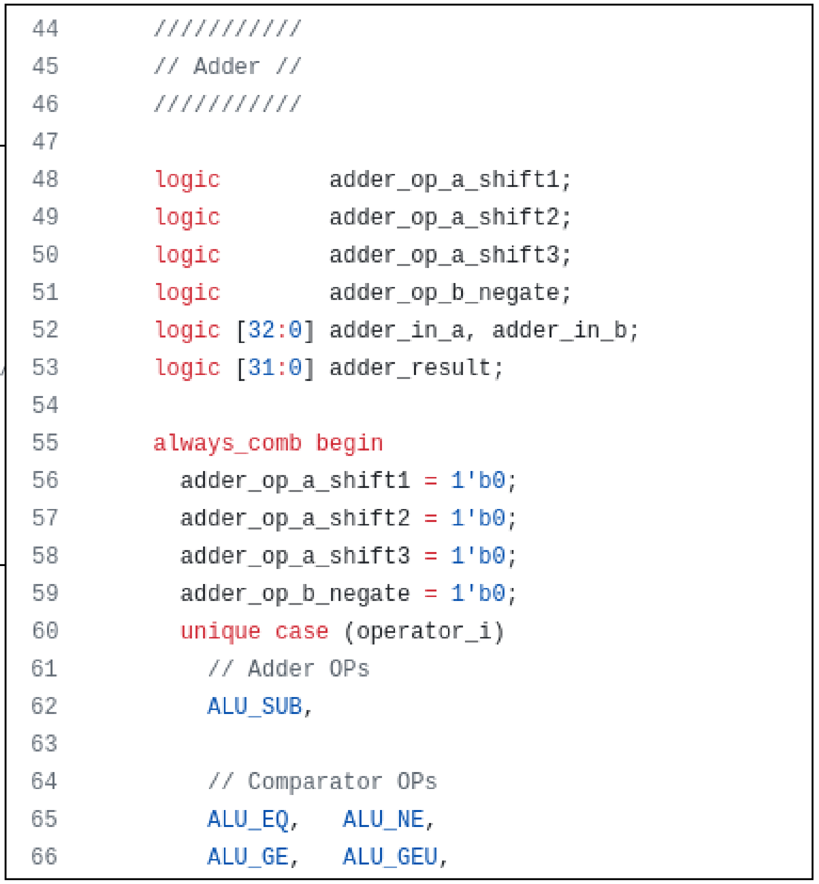
- ibex: The *standard* way of extending a core
- VexRiscv: Using Scala to design cores
- CVA6: Implements the core-v extension interface

Extending Cores: ibex



```
OPCODE_OP: begin  // Register-Register ALU operation
  rf_ren_a_o      = 1'b1;
  rf_ren_b_o      = 1'b1;
  rf_we           = 1'b1;
  if ({instr[26], instr[13:12]} == {1'b1, 2'b01}) begin
    illegal_insn = (RV32B != RV32BNone) ? 1'b0 : 1'b1; //
  end else begin
    unique case ({instr[31:25], instr[14:12]})
      // RV32I ALU operations
      {7'b000_0000, 3'b000},
      {7'b010_0000, 3'b000},
```

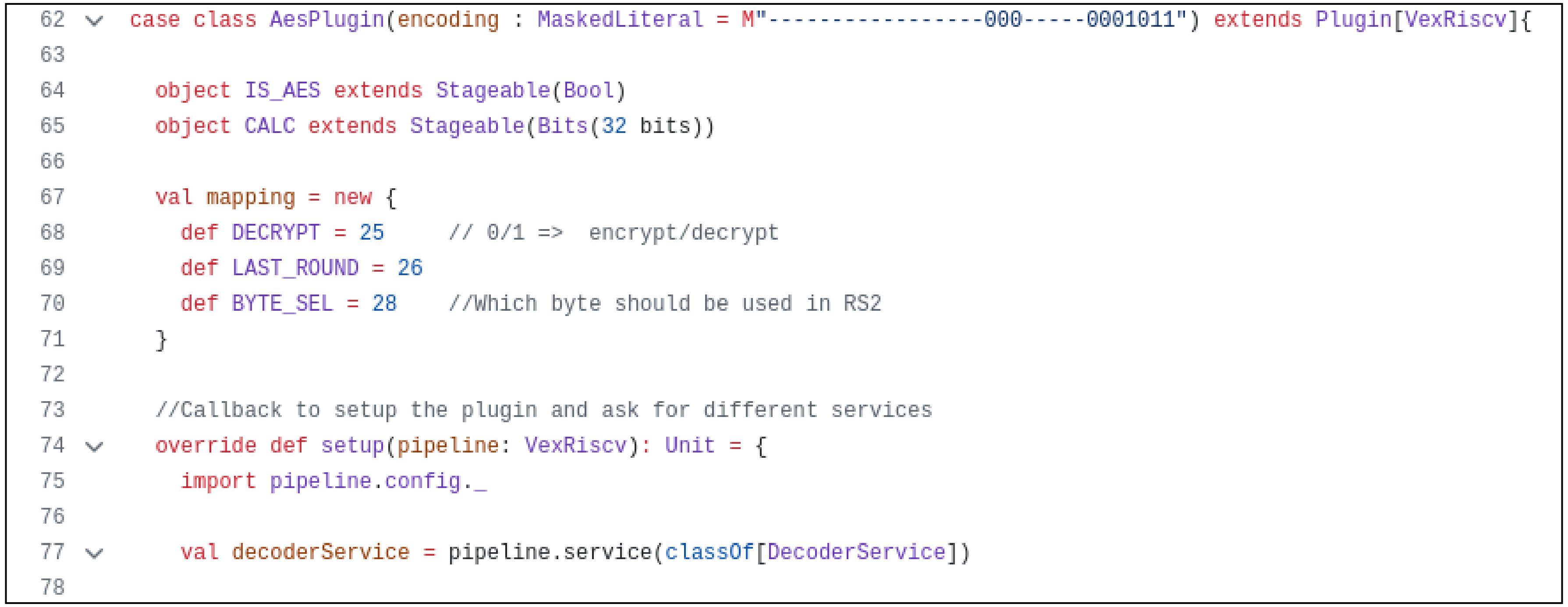
ibex_decoder.sv



```
///////////
// Adder //
///////////

logic        adder_op_a_shift1;
logic        adder_op_a_shift2;
logic        adder_op_a_shift3;
logic        adder_op_b_negate;
logic [32:0] adder_in_a, adder_in_b;
logic [31:0] adder_result;

always_comb begin
  adder_op_a_shift1 = 1'b0;
  adder_op_a_shift2 = 1'b0;
  adder_op_a_shift3 = 1'b0;
  adder_op_b_negate = 1'b0;
  unique case (operator_i)
    // Adder OPs
    ALU_SUB,

    // Comparator OPs
    ALU_EQ,   ALU_NE,
    ALU_GE,   ALU_GEU,
```

ibex_alu.sv

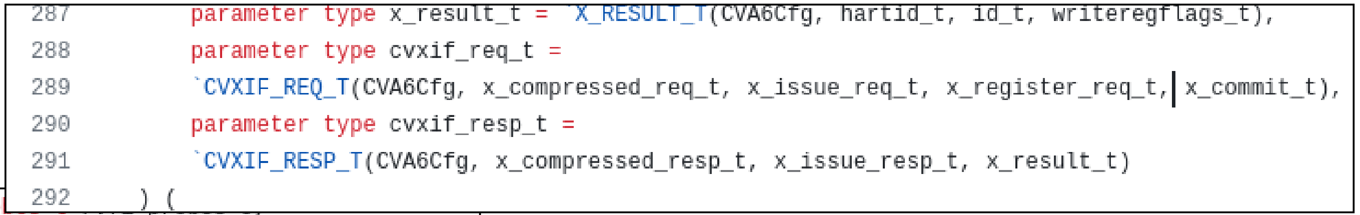
Extending Cores: VexRiscv



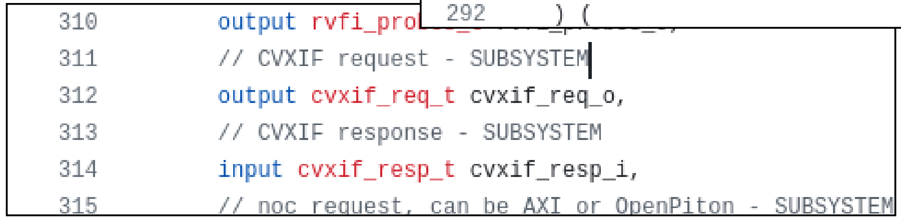
```scala
case class AesPlugin(encoding : MaskedLiteral = M"-----------------000-----0001011") extends Plugin[VexRiscv]{

  object IS_AES extends Stageable(Bool)
  object CALC extends Stageable(Bits(32 bits))

  val mapping = new {
    def DECRYPT = 25     // 0/1 =>  encrypt/decrypt
    def LAST_ROUND = 26
    def BYTE_SEL = 28    //Which byte should be used in RS2
  }

  //Callback to setup the plugin and ask for different services
  override def setup(pipeline: VexRiscv): Unit = {
    import pipeline.config._

    val decoderService = pipeline.service(classOf[DecoderService])

```

AesPlugin.scala

Extending Cores: CVA6



```
        parameter type x_result_t = `X_RESULT_T(CVA6Cfg, hartid_t, id_t, writeregflags_t),
        parameter type cvxif_req_t =
        `CVXIF_REQ_T(CVA6Cfg, x_compressed_req_t, x_issue_req_t, x_register_req_t, x_commit_t),
        parameter type cvxif_resp_t =
        `CVXIF_RESP_T(CVA6Cfg, x_compressed_resp_t, x_issue_resp_t, x_result_t)
    ) (
```



```
        output rvfi_pro
        // CVXIF request - SUBSYSTEM
        output cvxif_req_t cvxif_req_o,
        // CVXIF response - SUBSYSTEM
        input cvxif_resp_t cvxif_resp_i,
        // noc request, can be AXI or OpenPiton - SUBSYSTEM
```

cva6.sv

But which instructions should we add?

Potential goals of instruction set extensions:

- Improve application performance
- Reduce code size
- Reduce power consumption

Computer architecture is the sciene of trade-offs

Key driver is an application or set of applications

- Maybe you want to focus on a single application
- In most cases you want to look at a *domain* (identified by set of applications)
- Synthetic benchmarks may not be representative

Methods to Identify new Instructions

- Goal: Identify patterns in the code that can be accelerated
- Quick evaluation of the potential benefit
- Ideally: (semi-)automated exploration

Systematic Algorithm Analysis

- Manully inspect specific algorithms (not their implementation)
- Understand how algorithm transforms into low level operations
- Identify operations in algorithms that can benefit
- Requires deep understanding of algorithms and hardware implications
- A lot of manual work, discussions; compilers are good too..

Static Analysis

Peephole optimization

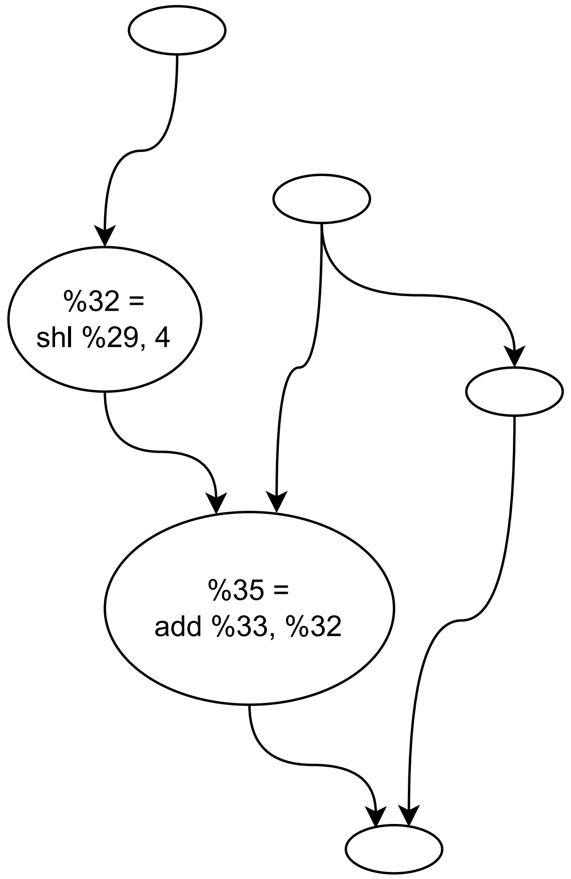
- Input: Candidate patterns of instructions
- Look at small sequences of generated instructions
- Pattern matching of instruction sequences
- The more code instructions you cover, the better the optimization can be

Static Program Analysis

- How to even find candidate patterns?
- Static analysis of the code to identify patterns
- Important to define and restrict pattern search

Example Data Flow Pattern

- Each node is an operation
- Data flows between operations
- If common subpattern: candidate
- Here: `rd = rs1 << 4 + rs2`



Dynamic Analysis

Static analysis can find patterns in a systematic way, but if and how often instructions are executed is unknown!

Use hardware, simulation, or (often) virtual prototype

-
- Dynamically executing the program exposes this information
- Need to execute on realistic data set
- Need to execute for a reasonable amount of time

Insights from dynamic analysis

- Annotate and weigh instructions with their dynamic occurences and durations
- Potential to find patterns that are not visible statically

Turning Patterns into Instructions

Implement into hardware

- Often: Start with virtual prototype
- Ultimately: Extend core, might bring surprises

Make use in software

- Most brutal way: inline code `asm volatile (".byte 0x00, 0xC7, 0x07, 0x7B;");`
  Actually this is very suboptimal, see this stackoverflow discussion for a better workable solution
- Extend compiler and assembler with intrinsics: `__riscv_complex_add(a, b);`
  This needs you to provide libraries or the user to hand-craft their code
- Ideally: Compiler generates the instructions: `complex_t c = a + b;`

Adding instructions to LLVM

LLVM adds the **tablegen** language to define instructions (and more)

```
// This defines patterns for assembler (simple add):
class ALU_rr<bits<7> funct7, bits<3> funct3, string opcodestr>
    : RVInstR<funct7, funct3, OPC_OP, (outs GPR:$rd), (ins GPR:$rs1, GPR:$rs2),
    opcodestr, "$rd, $rs1, $rs2">;

def ADD  : ALU_rr<0b0000000, 0b000, "add">;

// Actual instruction selection
Pat<(add GPR:$rs1, GPR:$rs2), (ADD GPR:$rs1, GPR:$rs2)>;
```

With more complex instructions, more sophisticated instruction selection (ISel) must be implemented
→ makes automation hard

Generating Hardware Implementation

Ideally, you would want to also generate hardware

Commonly summarized as *Application-Specific Instruction Set Processors (ASIPs)*

- Automatically generate RTL from instruction set description
- Automatically generate compiler backend
- Commercial examples: Codasip Codal, Synopsys ASIP Designer
- Open source example: CoreDSL

Example from our research: Sizalizer

Scale4Edge

- Goal: Reduce memory footprint of programs
- Identify further compressed instructions, in particular 48-bit
- Build a tool flow on established open source tools, each best in their domain
- Use case: DSP algorithms in the embedded space

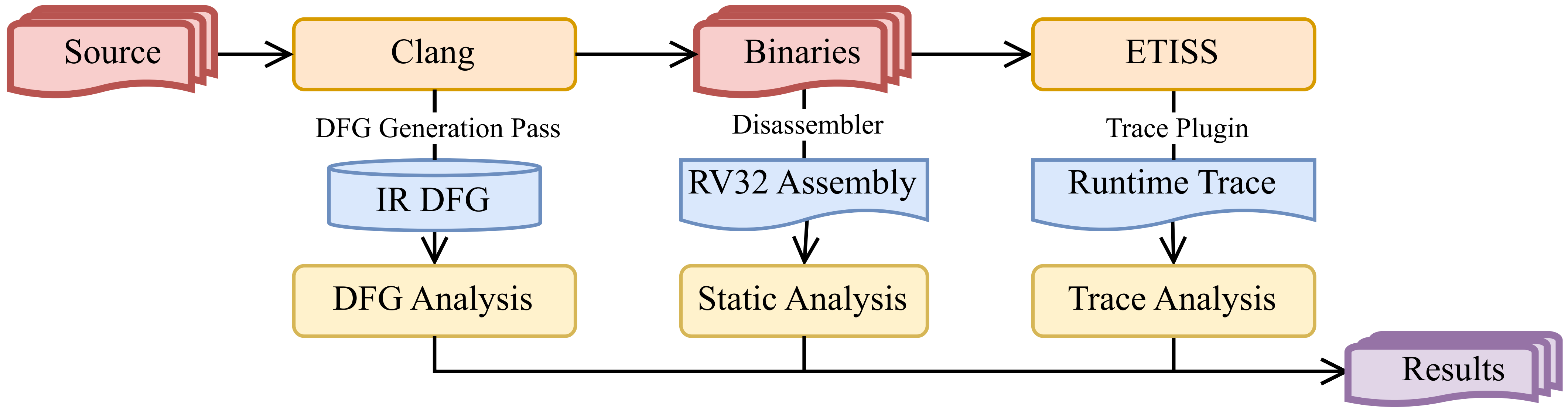
The Problem

- Tight resource constraints for embedded systems developer
- Necessity for development frameworks to achieve optimized memory footprints and power profiles
- Possible reduction of the memory footprint via an optimized instruction set architecture (ISA) with improved code density
- Revived research on domain-specific ISA optimization and the need for optimized platforms due to RISC-V
- Lack of fully integrated, easily accessible, holistic frameworks

The Vision

Sizalizer, which is an open-source framework suitable for the automated analysis of C/C++ applications targeting ISA based code size optimizations.

Analysis Framework: Architecture



Foundations

- Three levels analysis: Intermediate Representation (IR), binary executable code, and executed instructions
- Leveraging existing open-source tools to perform analysis
- LLVM IR used for structural evaluation, employing Control Flow Graph (CFG) and Data Flow Graph (DFG) metrics.
- Higher-level analysis via the Executable and Linkable Format (ELF) binary information
- Converting binary code back into assembly language mnemonics using tools like ‘objdump‘ for evaluation

Control-Data Flow Graph

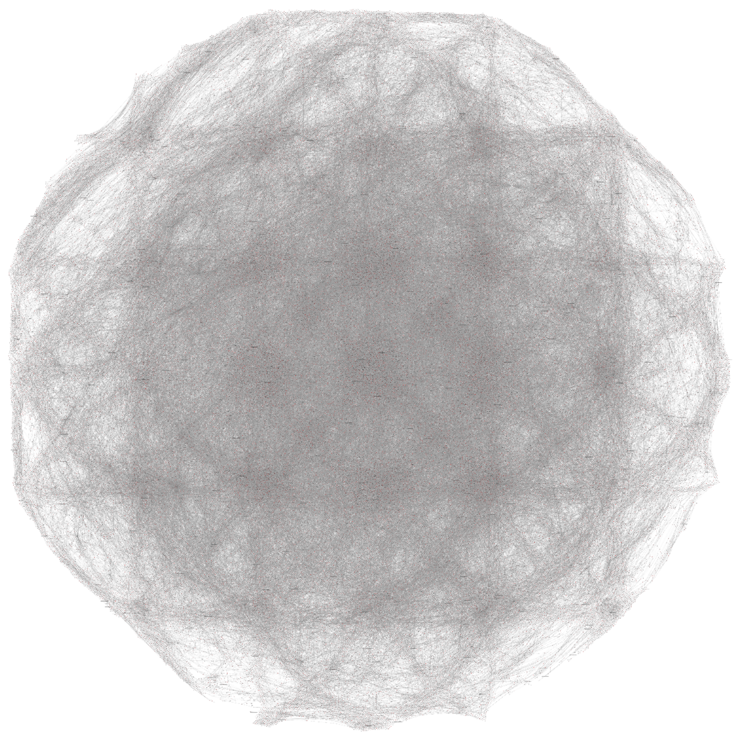
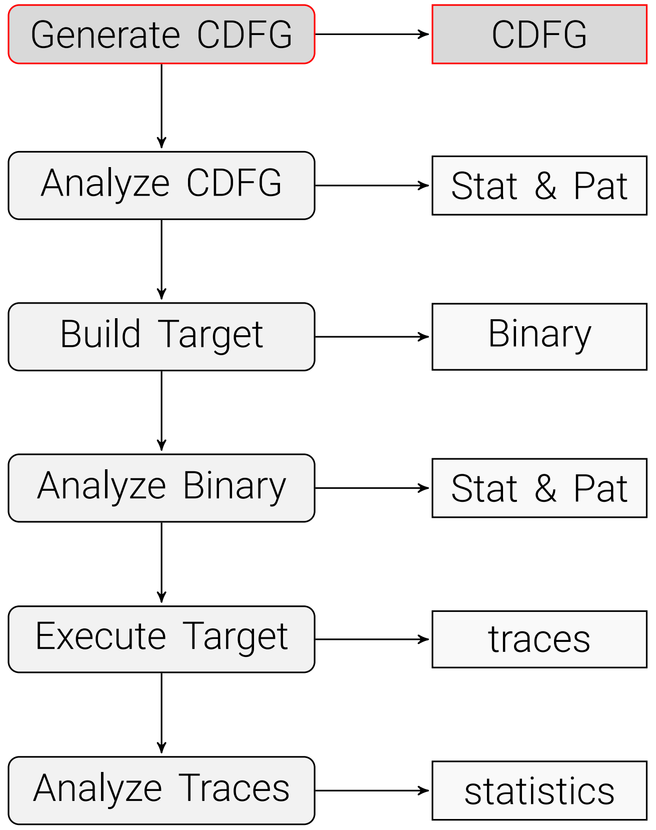


Figure: CDFG of CMSIS-DSP Example.



Control Flow Generation

```
void riscv_add_f64(
 const float64_t * pA,
 const float64_t * pB,
 float64_t * p,
 uint32_t blockSize
) { // Start of BB % 4
 uint32_t blkCnt;
 blkCnt = blockSize;
 while (blkCnt > 0U /*BB 11*/) {
  // Start of BB % 14
  *p++ = (*pA++) + (*pB++);
  blkCnt--;
} /*BB 16*/}
```

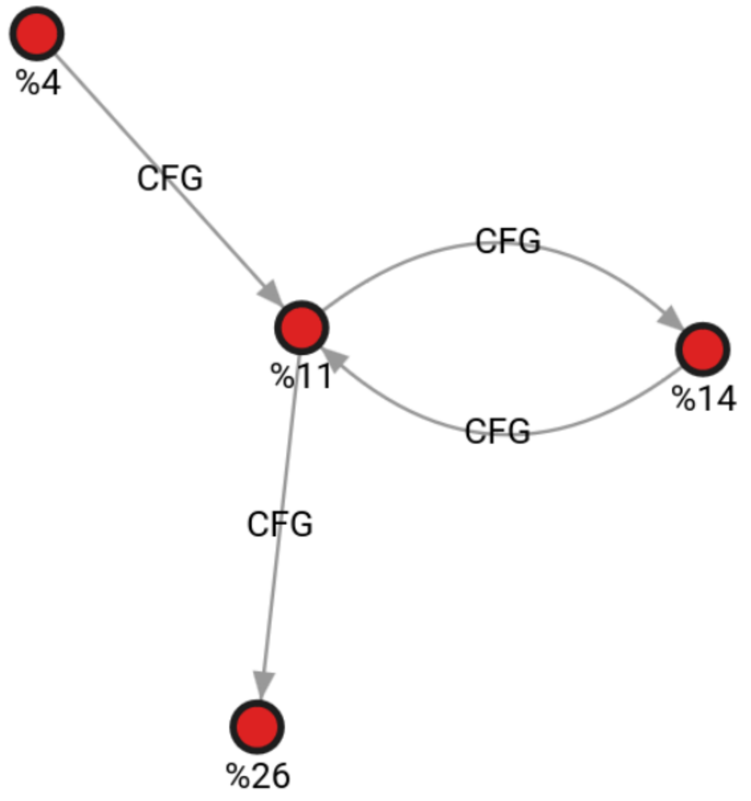
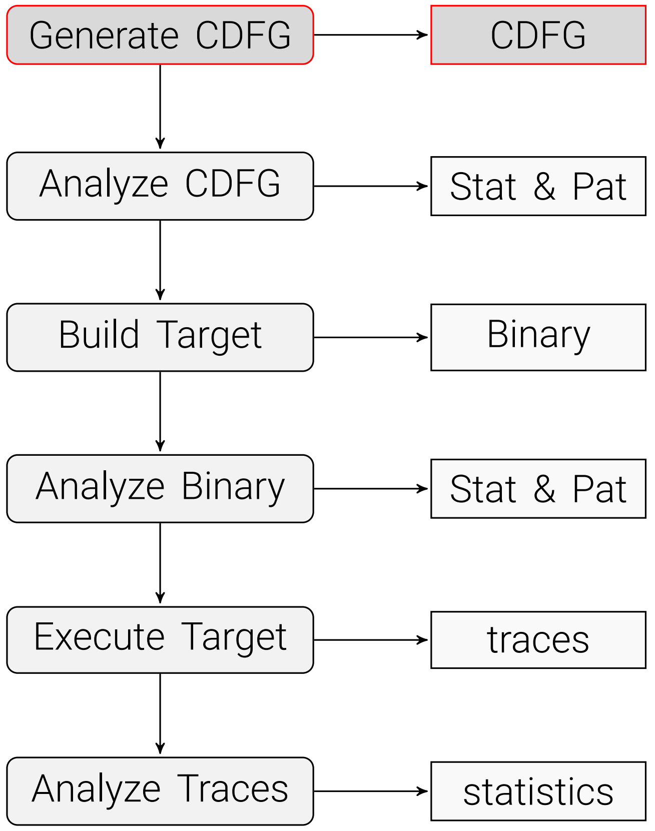


Figure: CFG of riscv_add_f64



Data Flow Generation

```
void riscv_add_f64(
 const float64_t * pA,
 const float64_t * pB,
 float64_t * p,
 uint32_t blockSize
) { // Start of BB % 4
 uint32_t blkCnt;
 blkCnt = blockSize;
 while (blkCnt > 0U /*BB 11*/) {
  // Start of BB % 14
  *p++ = (*pA++) + (*pB++);
  blkCnt--;
} /*BB 16*/}
```

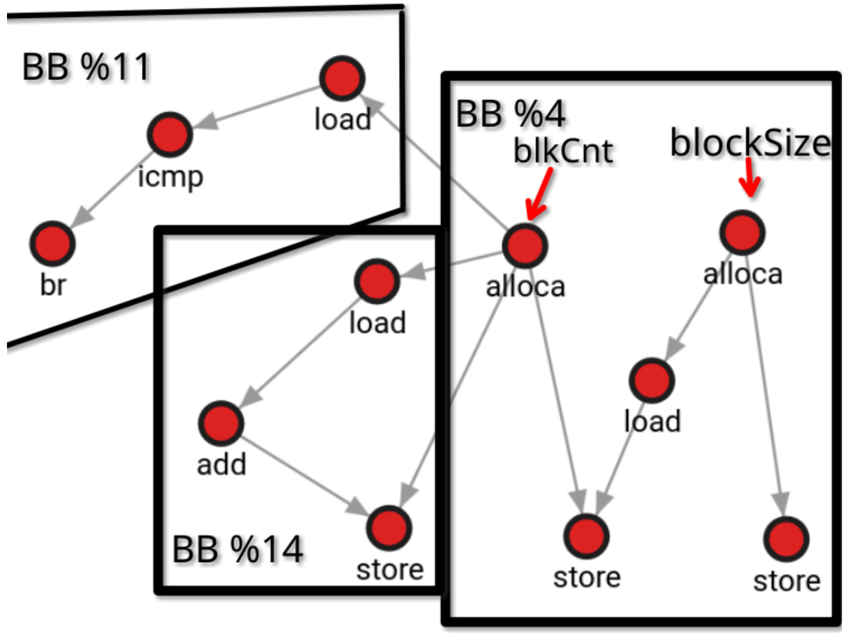
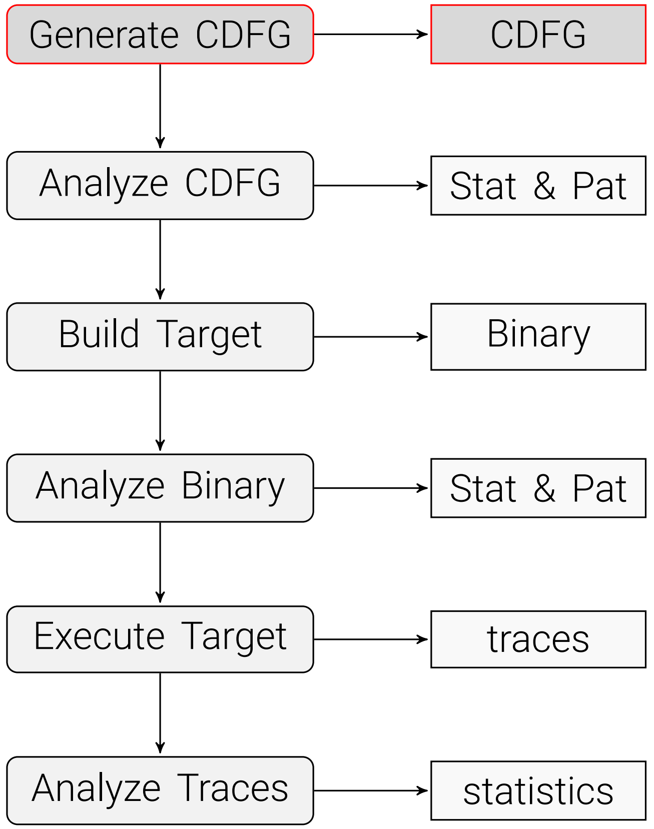


Figure: DFG of the blkCnt Variable

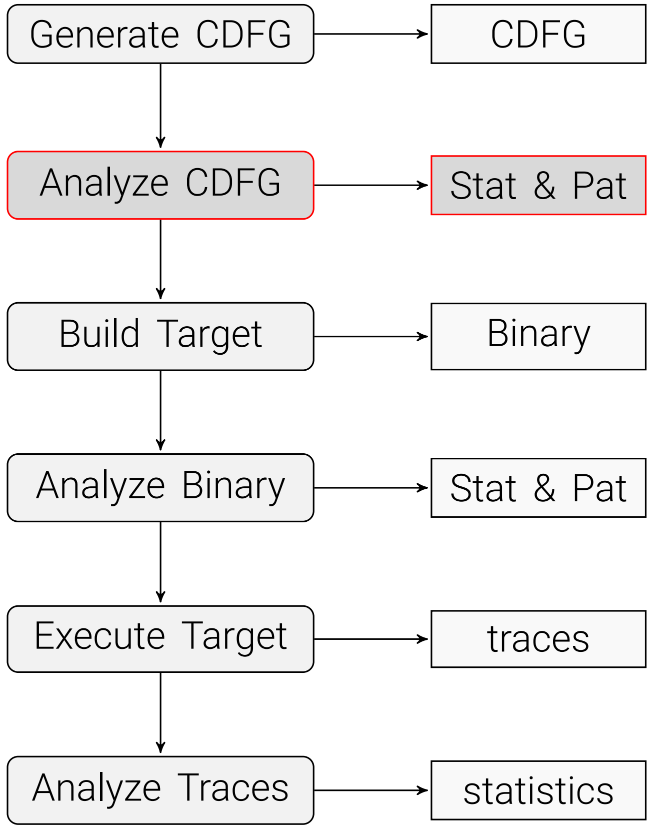


DFG Analysis

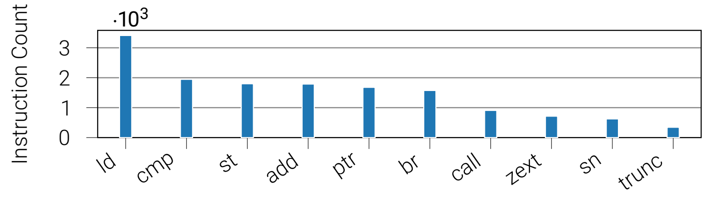
- Generate DFG patterns
  - N: width
  - M: depth
  - SpecialCond: func describing node relations (e.g. equality)

```
MATCH p1=($x_{11}$)-[:DFG]->...-[:DFG]-> ($x_{1N}$)
MATCH ...
MATCH pM=($x_{M1}$)-[:DFG]->...-[:DFG]-> ($x_{MN}$)
WHERE SpecialCond($x_{11}$, ..., $x_{1N}$) AND ...  AND SC($x_{M1}$, ..., $x_{MN}$)
RETURN p;
```

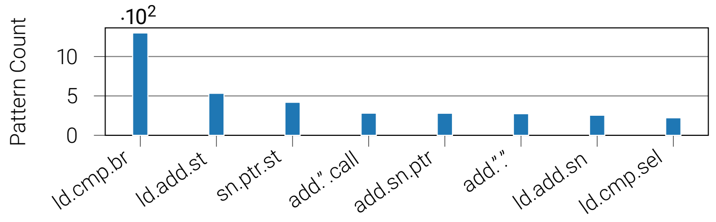
- query_builder(depth, width, special_cond, pat_to_ignore)



DFG Analysis

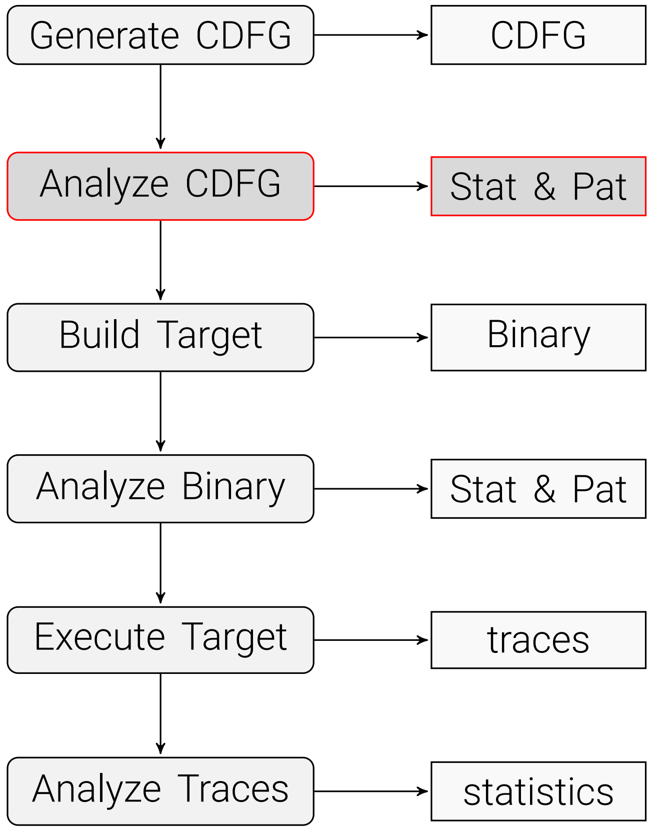


(a) Instruction Count



(b) Chain instruction Pattern

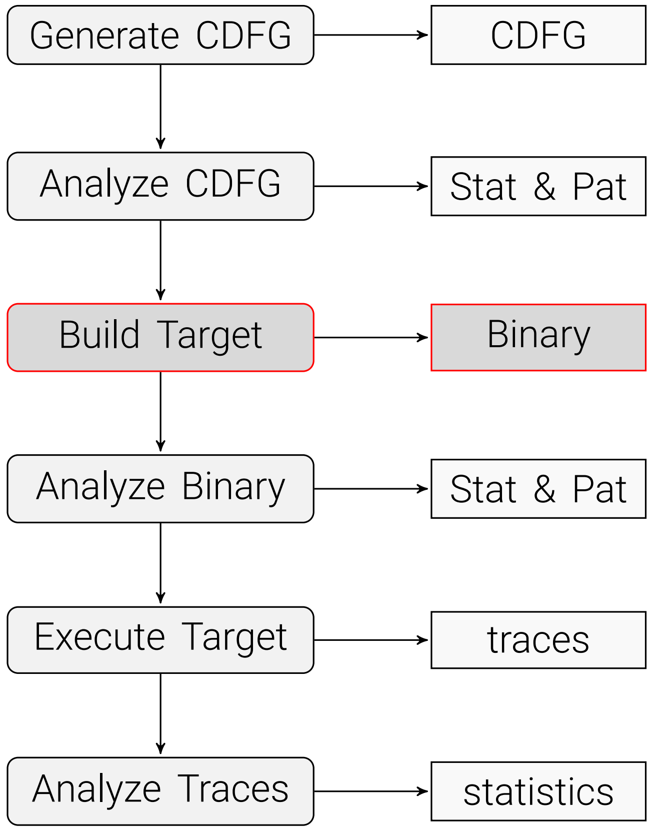
Figure: DFG Analysis Example



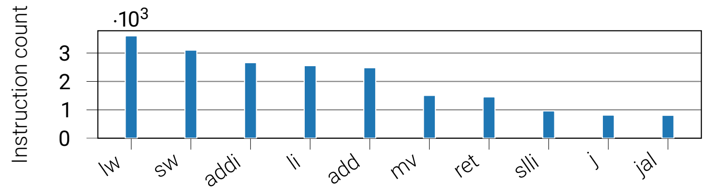
Binary Analysis

Address	Opcode	Mnemonic	Params
100f2	098e	c.sub	a2, a0
100f4	8145	c.li	a1, 0x0
100f6	ef003051	jal	ra, 0x10e08 ¡memset¿
100fa	17150000	auipc	a0, 0x1
100fe	130545e8	addi	a0, a0, -0x17c
10102	19c5	c.beqz	a0, 0x10110 ¡start+0x2e¿

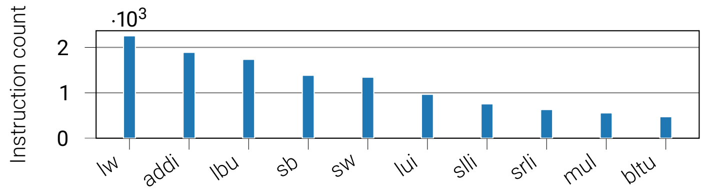
Table: Instruction Set



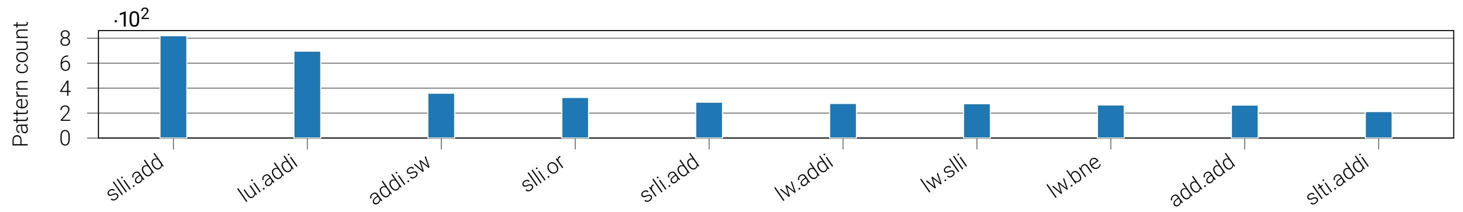
Binary Analysis



(a) 16 Bit Instructions

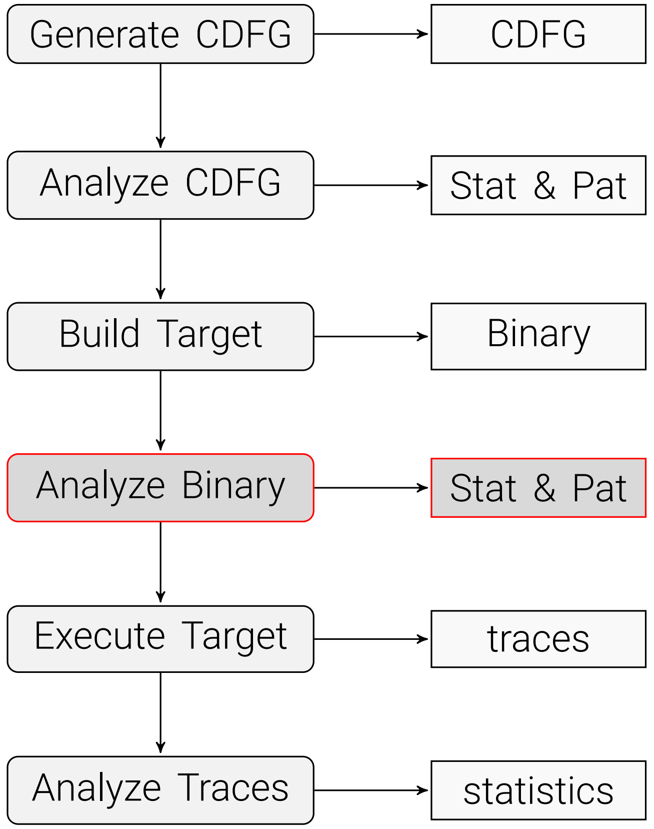


(b) 32 Bit Instructions



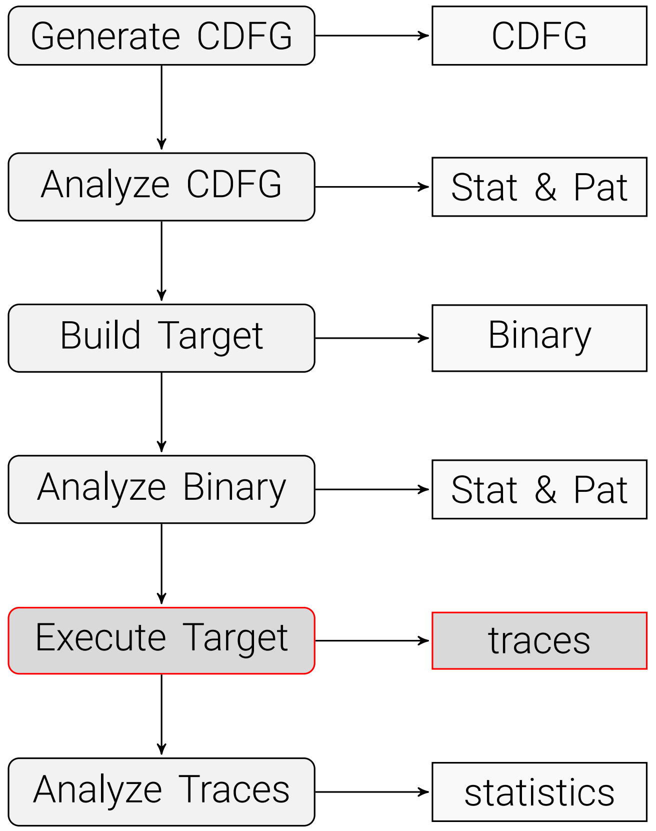
(c) Frequent instruction pairs

Figure: RISC-V Instruction Distribution Bitwith comparison.

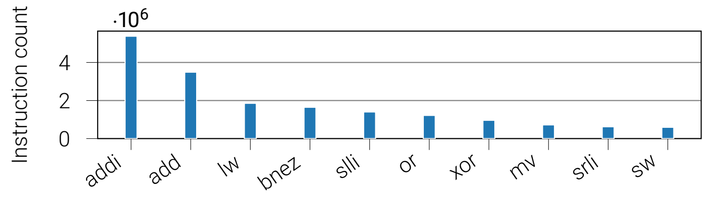


Trace Generation

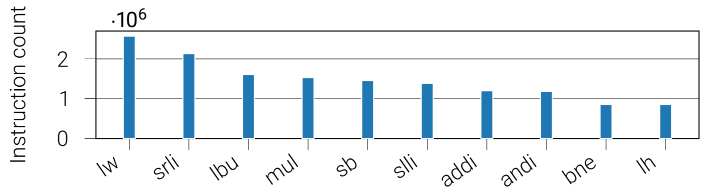
```
0x100036b8  auipc  00000000000000000000001010010111  [rd=5 | imm=0]
0x100036bc  addi   00000011100000101000001010010011  [rd=5 | rs1=5 | imm=56]
0x100036c0  csrrw  00110000010100101001000001110011  [rd=0 | rs1=5 | csr=773]
0x100036c4  auipc  00001111111111111101000110010111  [rd=3 | imm=268423168]
0x100036c8  addi   00010011110000011000000110010011  [rd=3 | rs1=3 | imm=316]
0x100036cc  auipc  00010000000011111101000100010111  [rd=2 | imm=269471744]
0x100036d0  addi   10010011010000010000000100010011  [rd=2 | rs1=2 | imm=2356]
0x100036d4  addi   10000100000000011000010100010011  [rd=10 | rs1=3 | imm=2112]
0x100036d8  cli    0100010110000001                  [imm=0 | rd=11]
0x100036da  addi   01001110010000011000011000010011  [rd=12 | rs1=3 | imm=1252]
0x100036de  csub   1000111000001001                  [rs2=2 | rd=4]
0x100036e0  cjal   0010111000001101                  [imm=818]
0x10003a12  cbeqz  1100011000011001                  [imm=14 | rs1=4]
```



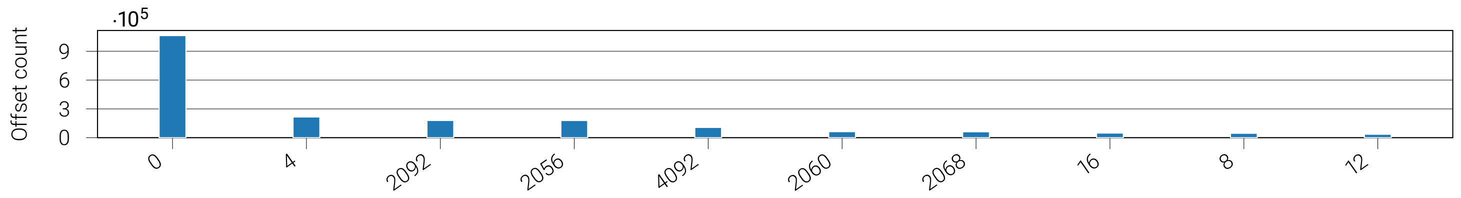
Analysis Framework: Trace Analysis



(a) Executed 16 Bit Instructions

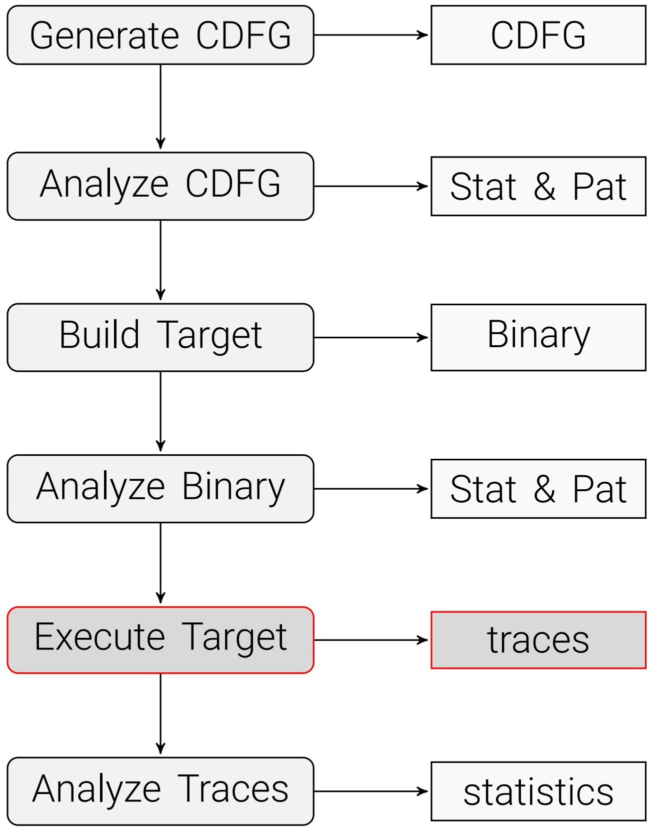


(b) Executed 32 Bit Instructions



(c) LW Immediate value Distribution

Figure: Dynamic Results



Suggested Instructions and Size Improvement

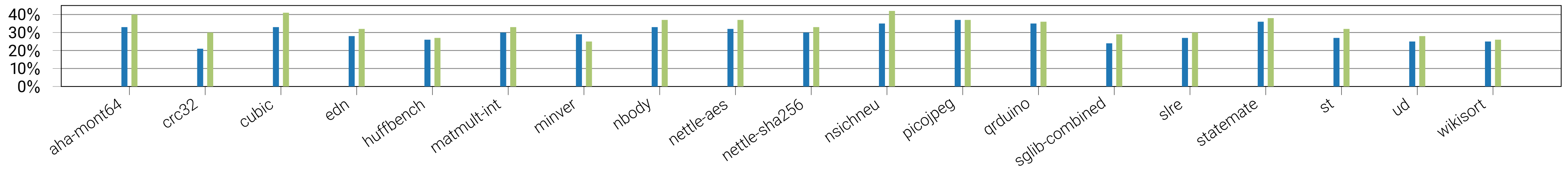


Figure: Calculated Static and Dynamic Improvement (blue: static improvement, green: dynamic improvement)

Discussion

- Sizalizer provides logical and functional instruction suggestions, but no real implementations
- Sizalizer can not provide hardware overhead of additional instructions
- Framework strenght lies in assisting the hardware architect in selecting new instructions
- Clear definition of the analysis target, beacue predefined forms or user-defined forms can be required.
- Static and dynamic size optimizations are calculated upper bounds

Using Code Analysis in Hackathon

- Sizalizer is available open source, but currently hard to use
- I have a readymade devcontainer, I can share if you are interested, but needs assistance
- Setup uses CMSIS-DSP as target for optimization
- Get in touch if this is something you would consider for Hackathon
- Alternatively: Join Jan Zielasko with his dynamic analysis tool

Contacts

Hager-Clukas, Andreas: andreas.hager-clukas@hm.edu

Wallentowitz, Stefan: stefan.wallentowitz@hm.edu

Website: https://aemy.cs.hm.edu