Fungible Memories for Automated Technology Mapping and Retargeting

ZACHARY D. SISCO*†, The Chinese University of Hong Kong, Shenzhen, China
SIJIE KONG*, University of California, Santa Barbara, USA
DANIEL RUELAS-PETRISKO, University of Washington, USA
JINGTAO XIA, University of California, Santa Barbara, USA
JULIAN SPRINGER, Technische Universität Berlin, Germany
VARUN RAO, University of California, Berkeley, USA
SPENCER WANG, University of California, Santa Barbara, USA
GUS HENRY SMITH, Southmountain Research, USA
BEN HARDEKOPF, University of California, Santa Barbara, USA
JONATHAN BALKIND, University of California, Santa Barbara, USA

During chip development, engineers must target different technologies, such as simulation and various ASIC and FPGA technologies. Conventionally, they split parts of the code (e.g., memories) into separate technology-specialized blocks implementing the same high-level behavior. This leads to brittle code, with multiple but subtly different blocks describing the same semantic behavior, harming verification, agility, and extensibility.

We propose *fungible memories*, an HDL-level "write once, map anywhere" memory abstraction with rich enough semantics to automatically target all relevant technologies using a single generic interface. We incorporate fungible memories into a compiler called Memo. For designs without a specific technology mapping, we also present a memory *decompiler* which lifts memories from an existing gate-level design to Memo, enabling automated technology retargeting, which is a holy grail for digital designers. We present a *structure-aware equality saturation* technique which scales to netlists with millions of cells and identifies memories that the state of the art cannot. We demonstrate that Memo effectively targets backends across different technology platforms (simulation, ASIC, and FPGA) over a suite of representative designs, including a RISC-V multicore SoC.

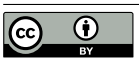
CCS Concepts: • **Hardware → Hardware description languages and compilation**; *Technology-mapping*.

Additional Key Words and Phrases: equality saturation, hardware decompilation, memory compilers

ACM Reference Format:
Zachary D. Sisco, Sijie Kong, Daniel Ruelas-Petrisko, Jingtao Xia, Julian Springer, Varun Rao, Spencer Wang, Gus Henry Smith, Ben Hardekopf, and Jonathan Balkind. 2026. Fungible Memories for Automated Technology

*Co-first author.
†Corresponding author.

Authors' Contact Information: Zachary D. Sisco, The Chinese University of Hong Kong, Shenzhen, China, zsisco@cuhk.edu.cn; Sijie Kong, University of California, Santa Barbara, USA, sijie_kong@ucsb.edu; Daniel Ruelas-Petrisko, University of Washington, Seattle, USA, petrisko@cs.washington.edu; Jingtao Xia, University of California, Santa Barbara, USA, jingtaoxia@ucsb.edu; Julian Springer, Technische Universität Berlin, Germany, springer@tu-berlin.de; Varun Rao, University of California, Berkeley, USA, vmrao@berkeley.edu; Spencer Wang, University of California, Santa Barbara, USA, spencer_wang@ucsb.edu; Gus Henry Smith, Southmountain Research, Seattle, USA, gus@southmountain.ai; Ben Hardekopf, University of California, Santa Barbara, USA, benh@cs.ucsb.edu; Jonathan Balkind, University of California, Santa Barbara, USA, jbalkind@ucsb.edu.



ACM 2475-1421/2026/6-ART216
https://doi.org/10.1145/3808294

Mapping and Retargeting. *Proc. ACM Program. Lang.* 10, PLDI, Article 216 (June 2026), 24 pages. https://doi.org/10.1145/3808294

1 Introduction

Hardware description languages (HDLs), such as Verilog, drive system-on-chip (SoC) development. When writing behavioral HDL code, engineers need to target different technologies to support different deployment platforms (for simulation, ASIC, FPGA, etc.). However, industry-standard HDLs are not equipped to cleanly target these different platforms from a single, generic implementation. As a result, it is standard for engineers to duplicate code into separate blocks, implementing the same functional behavior but targeting the specialized semantics of each specific technology. High-level source code is often littered with `ifdef` blocks corresponding with the diversity and divergence of the many ASIC and FPGA technologies that engineers need to target.

This practice is particularly problematic for memories, where technologies' structure and semantics vary widely. With the state-of-the-art solution, *memory inference*, the designer writes HDL code following syntactic templates dictated by vendor manuals to automate mapping [37]. However, the developer can only use memory inference to target one technology (FPGA), not a set of them. So while memory inference helps engineers in a restricted way with one aspect of the problem, per-technology work is still needed. This results in a brittle code base, with repeated but subtly different technology-specific blocks of code, harming verification, agility, and extensibility.

In this work, we focus on the problem of automated technology mapping for memories. Our insight is to incorporate a memory abstraction into the high-level HDL which is "write once, map anywhere," meaning the memory representation has a rich enough semantics to target all of the relevant technologies behind a single, generic interface. We call this memory abstraction a *fungible memory* to describe how it automatically transforms to satisfy a target technology's constraints (as illustrated in Figure 1). We implement fungible memories as an extension of PyRTL [9][1]. Our compiler, Memo, performs automated memory technology mapping across multiple platforms and technologies using a single, common HDL-level memory description.

While fungible memories are a useful abstraction for new hardware designs, there are many legacy designs that would benefit from them. As a second contribution, rather than requiring that existing code bases be ported manually, we present a hardware decompilation-based memory identification technique which automatically lifts memories from a gate-level netlist[2] enabling technology *retargeting*. We can take the netlist from an existing design that targeted a specific kind of memory and decompile it into HDL code that uses the fungible memory abstraction; the resulting code can then be retargeted to any type of memory covered by fungible memories, not just the original memory used by the design. Specific types of memory must be implemented for the fungible memory abstraction individually, but each type needs to be implemented only once and then all designs using fungible memories can benefit.

We introduce a rewrite-driven technique that transforms designs between high-level fungible memories and low-level logic in the same multi-level representation (Figure 1) that works in either direction (that is, for either mapping or decompilation). The core technology that drives the technique is *equality saturation* [32], a non-destructive rewriting technique. We describe a novel adaptation of equality saturation to scale to large real-world SoC designs and recover fungible memories in low-level logic through a combination of structure-aware scheduling and rewriting.

We show that Memo effectively targets five backends across simulation, ASIC, and FPGA, for a suite of representative designs. We evaluate our decompilation-based memory lifting technique

[1]Our abstraction can be added to any HDL. We choose PyRTL to make a concrete implementation for evaluation.
[2]A *netlist* is a graph of the wires and logic gates of a digital circuit and serves as a common, language-agnostic representation.

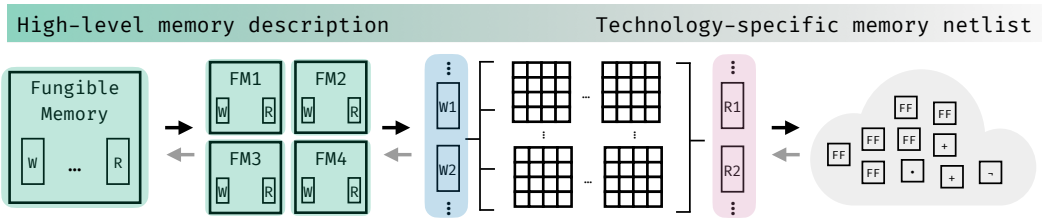


Fig. 1. Memo enables compilation of fungible memories via their decomposition into their constituent parts (left to right). It also enables abstraction and retargeting via hardware decompilation (right to left).

against state-of-the-art memory inference, demonstrating higher accuracy, inferring more semantics of the memories, and enabling automated retargeting. Our major contributions are:

- Fungible memories (Section 3), an abstraction and algebraic representation for manipulating the shape and configurations of memories in hardware designs.
- The Memo compiler (Section 4) for fungible memories which automatically maps memories according to technology constraints.
- The Memo decompiler (Section 5) which lifts memories in gate-level netlists to fungible memories. Our equality saturation technique (Section 6) scales to netlists with millions of cells and identifies memories that the state of the art cannot.
- An evaluation of Memo (Section 7) demonstrating that it enables automated memory retargeting, starting from a design mapped to one technology and retargeting it to another, using real-world open-source SoC designs.

2 Background

We provide background through a motivating example. Consider a developer implementing a core for an SoC. Such a core requires several memory blocks for instructions, data, caches, etc. As a small but non-trivial example, we focus on a register file, implemented as a memory block with 32 entries of 32-bit values. In this scenario, the target architecture specifies floating-point instructions which operate over three registers (e.g. as found in "fused" instructions in the RISC-V "RV32F" extension [13]), so the floating-point register file needs three read ports and one write port.

The developer needs to support three platforms for this SoC: software simulation, Xilinx UltraScale FPGA, and ASIC on TSMC 28 process node. Figure 2a shows the overall implementation in Verilog for this register file. The developer inserts `ifdef` blocks to support the three targets (an equivalent, common approach is to implement functionally equivalent modules with the same name and use tool-specific Verilog "file lists" to choose between them).

The behavioral Verilog code in Figure 2b intuitively captures the high-level functionality of the register file, and it suffices for simulation, e.g. with Verilator [29]. However, it is not guaranteed to correctly deploy to other technologies.

The FPGA only supports block RAM (BRAM) memories, which Verilog does not have built-in support for. Instead, the FPGA vendor provides a BRAM module as an intellectual property (IP) library, and through a detailed manual documents the module and how to instantiate it in Verilog.[3] The BRAMs for this FPGA are simple dual port, meaning they have two read–write ports (`2rw`). Unfortunately for the developer, it is not possible to directly map from a memory with three read ports and one write port to a single BRAM. They must write a custom mapping, illustrated in lines 7–9 of Figure 2a. Each line is an instantiation of a `2rw` BRAM module. The developer pores through the aforementioned manual and writes what they think is the correct way to instantiate BRAMs

[3]The UltraScale Architecture Memory Resources manual from AMD/Xilinx is 139 pages with 80 pages on BRAMs [39].

```
1  `ifdef VERILATOR
2  reg [31:0] regfile[31:0];
3    ...
4  `endif
5
6  `ifdef ULTRASCALE
7  bram_2rw_wrapper #(...) regfile1 (...);
8  bram_2rw_wrapper #(...) regfile2 (...);
9  bram_2rw_wrapper #(...) regfile3 (...);
10   ...
11 `endif
12
13 `ifdef TSMC28
14 tsmc28_1r1w_d32_w32_2_2rf mem1 (...);
15 tsmc28_1r1w_d32_w32_2_2rf mem2 (...);
16 tsmc28_1r1w_d32_w32_2_2rf mem3 (...);
17   ...
18 `endif
```

(a)

```
reg [31:0] regfile[31:0];
// Read logic
assign rs1_data_o = regfile[rs1];
assign rs2_data_o = regfile[rs2];
assign rs3_data_o = regfile[rs3];
// Write logic
always @(posedge clk)
  if (w_valid_i)
    regfile[rd] <= w_data_i;
```

(b)

```
bram_2rw_wrapper
  #(.D_WIDTH(32), .A_WIDTH(5))
regfile1 (.CLK(clk), .WEA(0),
.WEB(w_valid_i), .ADDRA(rs1), .ADDRB(rd),
.ENA(rs1_valid_i),  .ENB(w_valid_i),
.DOUTA(rs1_data_o), .DINB(w_data_i));
```

(c)

Fig. 2. **(a)** An implementation of the register file with three read ports and one write port, using `ifdef` blocks to support three technologies. **(b)** Behavioral implementation suitable for simulation which fills in lines 2–3 of **(a)**. **(c)** Module instantiation of a BRAM IP block from an FPGA vendor used in lines 7–9 of **(a)**.

for the register file. Figure 2c presents an instantiation of one of these; the developer instantiates two more for the `rs2` and `rs3` ports along with the necessary logic to connect them.

Next, the developer turns to ASIC. Again, Verilog does not explicitly support ASIC SRAM memories, and so the developer consults another vendor manual and instantiates another set of IP blocks. In lines 14–16 of Figure 2a, the developer splits the register file into three separate blocks with one read port and one write port (`1r1w`) to be able to map to the vendor's memory technology.

Overall, this implementation is brittle in three dimensions. **(1)** While the interface is the same, the developer must write three distinct implementations. Further, each `ifdef` block supports only one technology. The developer must add *another* `ifdef` to instantiate a new technology's memory and differentiate between the different, sometimes overlapping macros. **(2)** The implementation is specific to *one* memory block in the overall SoC; there are many more memories, each with their own particular semantics and configurations, thus the developer must repeat this process over and over. **(3)** The implementation is brittle to design changes, which may require updating all of the `ifdef` blocks across the whole code base. This issue arises with optimizations too. "Write-to-read forwarding" forwards the value of a write to the read port if they share the same address, bypassing a read. Some memory technologies support this optimization and can enable it through a parameter, while others require the developer to implement the logic manually.

FPGA vendors provide memory inference tools to alleviate this trouble, enabling developers to write behavioral Verilog code according to a template. Thus, some vendors may support this three-read port scenario if written in a conforming template. However, as our evaluation shows (Section 7), such tools are not always successful and they do not support SRAM technology mapping.

Memo ameliorates this convoluted situation. The developer only writes a single generic instantiation for each memory block, and the Memo compiler can automatically target all of these technologies without further developer effort.

3 Fungible Memories

In this section, we formalize the notion of a fungible memory. A fungible memory captures the shape, connections, and "portedness" of memory blocks, along with optimizations expressible abstractly or

$$
\begin{aligned}
\langle mem \rangle &::= \langle mem \rangle \oplus \langle mem \rangle \mid \mathsf{Mem}(\langle width \rangle, \langle height \rangle, \langle port \rangle^{+}) \circ \mathsf{Fwd}^{?} \\
\langle port \rangle &::= \langle r \rangle \mid \langle w \rangle \mid \langle rw \rangle \\
\langle r \rangle &::= \mathsf{Read}(\langle addr \rangle, \langle data_r \rangle) \circ \mathsf{Async}^{?} \\
\langle w \rangle &::= \mathsf{Write}(\langle en \rangle, \langle addr \rangle, \langle data_w \rangle, \langle mask \rangle^{?}) \\
\langle rw \rangle &::= \mathsf{ReadWrite}(\langle en \rangle, \langle addr \rangle, \langle data_r \rangle, \langle data_w \rangle, \langle mask \rangle^{?}) \\
\langle en \rangle, \langle addr \rangle, \langle data_r \rangle, \langle data_w \rangle, \langle mask \rangle &\in \mathbb{B}^k, \quad \langle width \rangle, \langle height \rangle, k \in \mathbb{N}
\end{aligned}
$$

Fig. 3. The grammar for fungible memories. The $\oplus$ operator denotes the parallel composition of two memories. Fwd and Async are supported features to denote a read–write forwarding optimization and asynchronous reads, respectively. $^{?}$ denotes an optional term.

in bit-level logic. The formal rewrite rules show how to transform fungible memories into different configurations while preserving behavioral equivalence. "Fungibility" enables designers to specify memories as high-level abstractions that map to technology-specific implementations.

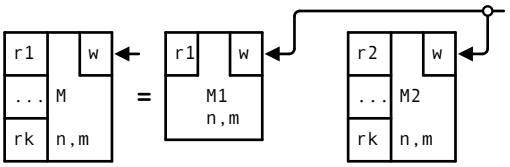
Figure 3 presents the grammar for fungible memories. A *fungible memory* denotes an $n \times m$ memory block with any combination of read, write, and read–write ports. Fungible memories compose in parallel with each other ($\oplus$), and their ports may be shared in the read/write logic of other fungible memories. Each port type has its own constructor for its specific inputs and outputs. Memories and ports may be applied with boolean functions that express various behaviors and optimizations commonly found in SoC memory blocks, such as write-to-read forwarding, asynchronous reads, write masks, and latch-last read.

3.1 Rewrite Rules

Figure 4 presents the formal algebraic rules for transforming fungible memories modulo equivalence. The rules cover transformations over memories and ports and can be read either forwards (left-to-right) or backwards (right-to-left). There are five kinds of fungible memory rules: split and merge by (1) read ports, (2) write ports, (3) width, and (4) height, and (5) function application (using write-to-read forward as an example). When reading the rules, "splitting" happens from the left-hand side of the equation to the right-hand side, while "merging" starts from the right-hand side to the left. Additionally, ports can be cast between read-only or write-only ports into shared read–write ports (by setting the relevant enable bit to zero).

The rules work at two levels: the level of memory blocks, and of ports connecting with memory blocks. While the symmetry of splitting and merging a memory appears obvious at the memory block level, there are intricate details at the port level, along with optimizations, that must be maintained to guarantee an equivalent memory system. To provide an intuition for the transformation each rule performs, we illustrate smaller examples of each rule type.

Read port rule. We start with the read port split/merge rule. Here, the left-hand side represents an `n`-by-`m` memory `M` with `k`-read ports (`r1`, ..., `rk`) and one write port `w`. The right-hand side shows `M` split into two memories, where `M1` has only one read port, `M2` has $(k-1)$ read ports, and they both share the interfaces from the same write port.



Read Port Split/Merge

$$\mathsf{Mem}(n, m, (r_i)_{i=1}^{k}, \vec{w}) = \mathsf{Mem}(n, m, (r_i)_{i=1}^{k-1}, \vec{w}) \oplus \mathsf{Mem}(n, m, r_k, \vec{w})$$

Write Port Split/Merge

$$\mathsf{Mem}(n, m, (r_i)_{i=1}^{k} \circ \mathsf{Async}, w_1, w_2) = \mathsf{Mem}(n, m, (r_i)_{i=1}^{k}, r_{k+1}, w_1') \oplus \mathsf{Mem}(n, m, (r_i)_{i=1}^{k}, r_{k+2}, w_2'),$$

$$\text{where } r_{k+1} = \mathsf{Read}(addr_{r_{k+1}}, data_{r_{k+1}}) \circ \mathsf{Async},\ r_{k+2} = \mathsf{Read}(addr_{r_{k+2}}, data_{r_{k+2}}) \circ \mathsf{Async},$$

$$w_1 = \mathsf{Write}(en_{w_1}, addr_{w_1}, data_{w_1}),\ w_1' = \mathsf{Write}(en_{w_1}, addr_{w_1}, \mathsf{Select}(data_{r_{k+2}}, data_{w_1})),$$

$$w_2 = \mathsf{Write}(en_{w_2}, addr_{w_2}, data_{w_2}),\ w_2' = \mathsf{Write}(en_{w_2}, addr_{w_2}, \mathsf{Select}(data_{r_{k+1}}, data_{w_2}))$$

Width Split/Merge

$$\mathsf{Mem}(n, m, (r_i)_{i=1}^{k}, (w_j)_{j=1}^{l}) = \mathsf{Mem}(n', m, (r_i')_{i=1}^{k}, (w_j')_{j=1}^{l}) \oplus \mathsf{Mem}(n - n', m, (r_i'')_{i=1}^{k}, (w_j'')_{j=1}^{l})$$

$$\text{where } r_i = \mathsf{Read}(addr_{r_i}, data_{r_i}) = \mathsf{Read}(addr_{r_i}, data'_{r_i} +\!\!+ data''_{r_i}),$$

$$r_i' = \mathsf{Read}(addr_{r_i}, data'_{r_i}) = \mathsf{Read}(addr_{r_i}, data_{r_i}[n' - 1 : 0]),$$

$$r_i'' = \mathsf{Read}(addr_{r_i}, data''_{r_i}) = \mathsf{Read}(addr_{r_i}, data_{r_i}[n - 1 : n']),$$

$$w_j = \mathsf{Write}(en_{w_j}, addr_{w_j}, data_{w_j}) = \mathsf{Write}(en_{w_j}, addr_{w_j}, data'_{w_j} +\!\!+ data''_{w_j}),$$

$$w_j' = \mathsf{Write}(en_{w_j}, addr_{w_j}, data'_{w_j}) = \mathsf{Write}(en_{w_j}, addr_{w_j}, data_{w_j}[n' - 1 : 0]),$$

$$w_j'' = \mathsf{Write}(en_{w_j}, addr_{w_j}, data''_{w_j}) = \mathsf{Write}(en_{w_j}, addr_{w_j}, data_{w_j}[n - 1 : n'])$$

Height Split/Merge

$$\mathsf{Mem}(n, 2^m, (r_i)_{i=1}^{k}, (w_j)_{j=1}^{l}) = \mathsf{Mem}(n, 2^{m-1}, (r_i')_{i=1}^{k}, (w_j')_{j=1}^{l}) \oplus \mathsf{Mem}(n, 2^{m-1}, (r_i'')_{i=1}^{k}, (w_j'')_{j=1}^{l})$$

$$\text{where } r_i = \mathsf{Read}(addr_{r_i}, data_{r_i}) = \mathsf{Read}(addr_{r_i}, \mathsf{Mux}(addr_{r_i}, data'_{r_i} +\!\!+ data''_{r_i})),$$

$$r_i' = \mathsf{Read}(addr'_{r_i}, data'_{r_i}) = \mathsf{Read}(addr_{r_i}[m - 2 : 0], data_{r_i}),$$

$$r_i'' = \mathsf{Read}(addr''_{r_i}, data''_{r_i}) = \mathsf{Read}(addr_{r_i}[m - 2 : 0], data_{r_i}),$$

$$w_j = \mathsf{Write}(en_{w_j}, addr_{w_j}, data_{w_j}),$$

$$w_j' = \mathsf{Write}(en'_{w_j}, addr'_{w_j}, data'_{w_j}) = \mathsf{Write}(en_{w_j} \wedge \neg addr_{w_j}[m - 1], addr_{w_j}[m - 2 : 0], data_{w_j}),$$

$$w_j'' = \mathsf{Write}(en''_{w_j}, addr''_{w_j}, data''_{w_j}) = \mathsf{Write}(en_{w_j} \wedge addr_{w_j}[m - 1], addr_{w_j}[m - 2 : 0], data_{w_j})$$

Forwarding (function application)

$$\mathsf{Mem}(n, m, (r_i)_{i=1}^{k}, \vec{w}) \circ \mathsf{Fwd} = \mathsf{Mem}(n, m, (r_i')_{i=1}^{k}, \vec{w})$$

$$\text{where } r_i = \mathsf{Read}(addr_{r_i}, data_{r_i}),\ r_i' = \mathsf{Read}(addr_{r_i},\ \mathsf{Fwd}(addr_{r_i}, data_{r_i}, \vec{w}))$$

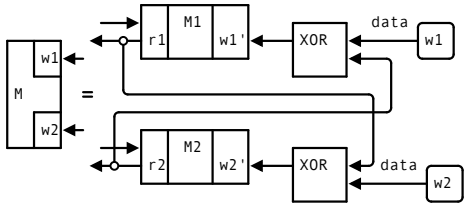
Cast Ports

$$\mathsf{Read}(addr, data_r) = \mathsf{ReadWrite}(0, addr, data_r, \_)$$

$$\mathsf{Write}(en, addr, data_w) = \mathsf{ReadWrite}(en, addr, \_, data_w)$$

$$\mathsf{ReadWrite}(en, addr, data_r, data_w) = (\mathsf{Read}(addr, data_r), \mathsf{Write}(en, addr, data_w))$$

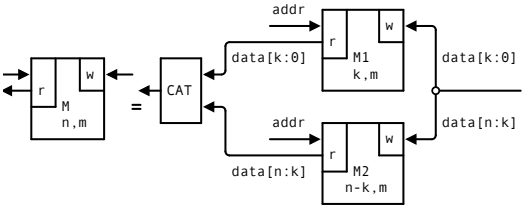
Fig. 4. Fungible memory rules, using the convention that r is a Read port, and w is a Write port. $\vec{p}$ is shorthand for a sequence of ports where the sequence length is not relevant to the rule.



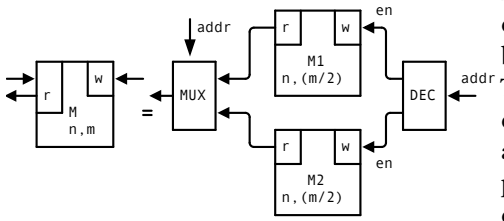
Write port rule. Write port splitting must also maintain data consistency. There are many strategies [2, 19, 20]; here we show one. For space, we only illustrate the two write ports `w1` and `w2` and ignore any existing read ports. The new split memories `M1` and `M2` each have a newly created write port `w1'` and `w2'`, respectively.

There are also two additional read ports `r1` and `r2` which forward a read data value to the selection function. The selection function is implemented as `XOR`, left generic in the formal rule in fig. 4. It selects the most up-to-date data value between the split memory blocks, leveraging the identity that x `XOR` y `XOR` $y = x$.

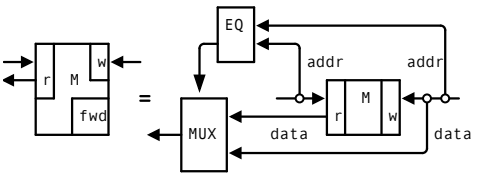
Width rule. Merging and splitting can also occur over the "width" and "height" dimensions (data and addresses). Rule `(3)` illustrates splitting a larger memory `M` by `k`, resulting in a `k`-width memory and `(n − k)`-width memory. The address and enable interfaces for the write port remain unchanged; we omit them for space. `CAT` stands for the *concatenation* of two values.



Height rule. Splitting over the height of the memory uses additional logic to choose which memory block to read from (write to) based on the address. This rule splits the height of fungible memories over powers of 2 because memory blocks are only addressable in powers of 2. The `MUX` cell multiplexes the read-data outputs of `M1` and `M2` based on the address. The `DEC` cell is a decoder that enables the write port of either `M1` or `M2`.



Function application. Fungible memories may be applied with functions that express behaviors and optimizations commonly found in SoC memory blocks, such as write-to-read forwarding, latch-last read, and asynchronous reads. We illustrate this via an example where a memory has write-to-read forwarding enabled (`fwd`), but the target technology does not support forwarding, so it is "pulled out" as separate logic.



3.2 Proof of Correctness

Formalization. We model the state of a fungible memory with word width w and address width k as a total function $\sigma : \mathbb{B}^k \to \mathbb{B}^w$, mapping each of the 2^k addresses to a w-bit data word. A write port $p = (en, addr, data)$ consists of a one-bit enable signal $en \in \mathbb{B}^1$, an address $addr \in \mathbb{B}^k$, and a data value $data \in \mathbb{B}^w$. The application of a single write port to a state is defined pointwise: [4]

$$\mathsf{applyWrite}(\sigma, p)(a) \;=\; \begin{cases} data & \text{if } en = 1 \text{ and } a = addr, \\ \sigma(a) & \text{otherwise.} \end{cases}$$

A list of write ports $(p_1, \ldots, p_n)$ is applied by left-folding applyWrite over the list. We define a fungible memory M inductively as either a *leaf* (σ, R, W)—carrying a state σ, read ports R, and write ports W—or a parallel composition $M_1 \oplus M_2$ of two fungible memories. A single step of evaluation applies each leaf's write-port list to its state; parallel-composed memories step independently.

[4]Without loss of generality, the formalism excludes write masks and read–write ports.

Behavioral Equivalence. We define equivalence by first *flattening* a memory into a list of leaf descriptors (σ_i, W_i) via a function flatten, which maps a leaf to a singleton list and parallel composition to list concatenation. Two leaf descriptors (σ_1, W_1) and (σ_2, W_2) are *leaf-equivalent* if, whenever $\sigma_1 = \sigma_2$, we have $\mathsf{applyWrites}(\sigma_1, W_1)(a) = \mathsf{applyWrites}(\sigma_2, W_2)(a)$ for every address a. Two memories M_1 and M_2 are then *behaviorally equivalent* when $\mathsf{flatten}(M_1)$ and $\mathsf{flatten}(M_2)$ are pointwise leaf-equivalent. Because flatten maps $\oplus$ to list concatenation, associativity of $\oplus$ reduces to associativity of concatenation. Note that read ports can be ignored since equivalence concerns only the write-induced state mutation.

Preservation of Equivalence. We verify each rewrite rule in Figure 4 by proving that the structural transformation preserves behavioral equivalence. *Read-port splitting* is immediate, since flatten ignores read ports. *Write-port splitting* shows that partitioning a write-port list across two parallel leaves, with a last-write-wins multiplexer selecting which half owns each write, preserves the folded semantics of applyWrites.[5] *Width splitting* decomposes a w-bit memory into high and low slices by showing that applyWrites commutes with bitvector extraction $(\cdot)[i{+}l{-}1 : i]$, then recombining the slices via $data = data[\, n{-}1 : n' \,] \mathbin{+\!\!+} data[\, n'{-}1 : 0 \,]$. *Height splitting* partitions the 2^k-row address space by the most significant bit (MSB) of the address: a write to the "upper half" has a MSB of 1 and targets only the upper sub-memory, disabling the write port of the "lower half." The correctness of this routing rests on the lemma that two k-bit addresses sharing the same MSB are equal if and only if their lower $(k{-}1)$ bits coincide.

We mechanically verified all rules in the Lean 4 proof assistant [12]. The proofs follow a uniform methodology: reduce each equivalence to a pointwise equality $\mathsf{applyWrites}(\sigma, W)(a) = \mathsf{applyWrites}(\sigma', W')(a)$ for an arbitrary address a, established by induction on the write-port list W with the state σ generalized. Case analysis on boolean conditions (the enable bit, MSB comparisons) and standard bitvector identities for slicing and reconstruction supply the remaining reasoning.

3.3 An Example

To see fungible memories in action, we return to the register file example from Section 2 and consider its formalization. When the developer writes the custom mapping from the original `3r1w` register file to the three-`1r1w` memories for the ASIC technology, they implicitly manipulate the configuration of the register file. Instead, if we use the rules from Figure 4, we *explicitly* transform the memory according to equality-preserving rewrites. For the register file, we formally express this transformation as two applications of the read-port splitting rule:

$$\begin{aligned}\mathsf{Mem}(32, 32, r_1, r_2, r_3, w) &= \mathsf{Mem}(32, 32, r_1, r_2, w) \oplus \mathsf{Mem}(32, 32, r_3, w) \\ &= \mathsf{Mem}(32, 32, r_1, w) \oplus \mathsf{Mem}(32, 32, r_2, w) \oplus \mathsf{Mem}(32, 32, r_3, w), \\ \text{where } r_1 &= \mathsf{Read}(\texttt{rs1}, \texttt{rs1\_data\_o}), r_2 = \mathsf{Read}(\texttt{rs2}, \texttt{rs2\_data\_o}), \\ r_3 &= \mathsf{Read}(\texttt{rs3}, \texttt{rs3\_data\_o}), w = \mathsf{Write}(\texttt{w\_valid\_i}, \texttt{rd}, \texttt{w\_data\_i})\end{aligned}$$

For the FPGA technology, we follow the same transformation, but then cast each `1r1w` memory to a `2rw`, following the first and second port casting rules, for the read and write port, respectively. The following demonstrates casting for (r_1, w):

$$\begin{aligned}\mathsf{Mem}(32, 32, r_1, w) &= \mathsf{Mem}(32, 32, rw_1, rw') \text{ where } r_1 = rw_1, \text{ and } w = rw', \\ rw_1 &= \mathsf{ReadWrite}(0, \texttt{rs1}, \texttt{rs1\_data\_o}, \_), \\ rw' &= \mathsf{ReadWrite}(\texttt{w\_valid\_i}, \texttt{rd}, \_, \texttt{w\_data\_i})\end{aligned}$$

[5] As noted in Section 3.1, different selection strategies can be used in the resulting implementation.

Parallel Composition $T[\![m_1 \oplus m_2]\!] ::= T[\![m_1]\!],\ T[\![m_2]\!]$

Memory Creation

$T[\![\text{Mem}(w, h, \vec{p})]\!] ::= \text{fresh } m \text{ where } m \texttt{ = MemBlock}(w, h),\ T[\![p_1]\!](m), \ldots, T[\![p_k]\!](m)$

$T[\![\text{Mem}(w, h, \vec{p}) \circ \text{Fwd}]\!] ::= \text{fresh } m \text{ where } m \texttt{ = MemBlock}(w, h),\ T_{\text{Fwd}}[\![p_1]\!](m), \ldots, T_{\text{Fwd}}[\![p_k]\!](m)$

Write Port

$T[\![\text{Write}(en, addr, data_w)]\!](m) ::= m\texttt{[}addr\texttt{]} \texttt{ <<= MemBlock.EnabledWrite(}data_w\texttt{, }en\texttt{)}$

$T[\![\text{Write}(en, addr, data_w, mask)]\!](m) ::= \text{fresh } v \text{ where } v = m\texttt{[}addr\texttt{]},$

$m\texttt{[}addr\texttt{] <<= MemBlock.EnabledWrite((}data_w \texttt{ \& } mask\texttt{) | (}v \texttt{ \& } \sim mask\texttt{), }en\texttt{)}$

Read Port

$T[\![\text{Read}(addr, data_r)]\!](m) ::= \text{fresh } r \text{ where } r \texttt{ = Register(bitwidth=len(}addr\texttt{))},$

$r\texttt{.next <<= }addr, \quad data_r \texttt{ <<= } m\texttt{[}r\texttt{]}$

$T[\![\text{Read}(addr, data_r) \circ \text{Async}]\!](m) ::= data_r \texttt{ <<= } m\texttt{[}addr\texttt{]}$

$T_{\text{Fwd}}[\![\text{Read}(addr, data_r)]\!](m) ::= \text{fresh } r \text{ where } r \texttt{ = Register(bitwidth=len(}addr\texttt{))},$

$r\texttt{.next <<= }addr, \quad v_0 = m\texttt{[}r\texttt{]}, \quad data_r \texttt{ <<= } v_n$

$\text{for } i = 1, \ldots, n: \quad v_i = \texttt{pyrtl.select(}en_i \texttt{ \& (}r \texttt{ == } wa_i\texttt{), }wd_i\texttt{, }v_{i-1}\texttt{,)}$

$T_{\text{Fwd}}[\![\text{Read}(addr, data_r) \circ \text{Async}]\!](m) ::= v_0 = m\texttt{[}addr\texttt{]}, \quad data_r \texttt{ <<= } v_n$

$\text{for } i = 1, \ldots, n: \quad v_i = \texttt{pyrtl.select(}en_i \texttt{ \& (}addr \texttt{ == } wa_i\texttt{), }wd_i\texttt{, }v_{i-1}\texttt{)}$

where $(en_1, wa_1, wd_1), \ldots, (en_n, wa_n, wd_n)$ are the write ports of m.

ReadWrite Port

$T[\![\text{ReadWrite}(en, addr, data_r, data_w)]\!](m) ::= T[\![\text{Read}(addr, data_r)]\!](m),\ T[\![\text{Write}(en, addr, data_w)]\!](m)$

Fig. 5. Formal translation rules to compile a fungible memory into a list of equivalent statements in PyRTL. $T[\![\cdot]\!]$ defines the translation function; $T_{\text{Fwd}}[\![\cdot]\!]$ is its forwarding variant.

This same casting procedure will be repeated for $\text{Mem}(32, 32, r_2, w)$ and $\text{Mem}(32, 32r_3, w)$ to form three memories composed in parallel each with two read–write ports: $\text{Mem}(32, 32, rw_1, rw') \oplus \text{Mem}(32, 32, rw_2, rw') \oplus \text{Mem}(32, 32, rw_3, rw')$.

3.4 Extending an HDL with Fungible Memories

To use fungible memories in real hardware designs, we extend the PyRTL HDL to support specifying fungible memories. We introduce a new class into PyRTL called `FungibleMemory`, which accepts a list of ports, specifies its width and height, and configures a number of features (e.g., write-to-read forwarding). The memory ports (inputs and outputs) are PyRTL `WireVector`-typed values.

The Memo compiler then *generates the corresponding logic* obtained from refining the fungible memory given a particular configuration of ports and parameters. This process follows the same "elaboration-through-execution" strategy that HDLs like PyRTL and Chisel [4] use.

Here, we formalize the rules that transform a given fungible memory into valid PyRTL code. Compilation proceeds by translating each fungible memory to a PyRTL MemBlock. Figure 5 presents the translation rules to produce a PyRTL memory from a given fungible memory. The `a <<= b` syntax denotes wire assignment in PyRTL; it "drives" `a` with `b`, connecting the two signals. The translation rules also handle specific memory semantics and optimizations that need to be compiled to compatible logic in PyRTL. For example, a PyRTL MemBlock does not support write–read forwarding, so this optimization maps to explicit logic that exists outside of the MemBlock (defined in the $T_{\text{Fwd}}[\![\cdot]\!]$ translation function in Figure 5).

Following the example from Section 2, we specify the same `3r1w` register file in the extension of PyRTL (switching on the write-to-read forwarding optimization):

```
regfile = FungibleMemory(32, 32, fwd=True, [ Read(rs1, rs1_data_o),
          Read(rs2, rs2_data_o), Read(rs3, rs3_data_o), Write(en, rd, w_data) ])
```

With fungible memories integrated into PyRTL, we can elaborate any PyRTL design with fungible memories into a simulation-capable PyRTL netlist. However, up to this point, the Memo compiler can only map fungible memories to technologies that have a one-to-one mapping. Next, we consider technology constraints in the Memo compiler.

4 The Memo Compiler

Here we show how we use the fungible memory formalism to build a memory compiler, describing how the algebra of fungible memories enables automated memory mapping following technology constraints. By default, fungible memories specified in PyRTL can elaborate into simulation-capable netlists. Then, given a particular memory technology and its constraints, we describe an algorithm to automatically map the fungible memory in a way that satisfies those constraints.

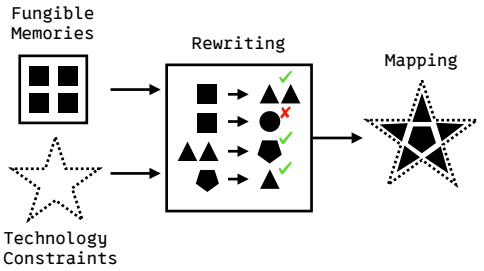


Fig. 6. An illustration of the Memo compiler flow.

Figure 6 illustrates the overall flow. The black squares are the fungible memories from the design. The rewrites manipulate and reconfigure their shapes at the "abstract" level such that they can be successfully mapped to the given technology (i.e., the star shape). Once the memory satisfies the technology's constraints, Memo passes it to the backend to emit code—for example, as module instances in Verilog or scripts for a particular hardware synthesis tool. Thus, fungible memories enable automated technology mapping via algebraic rewrites. Lastly, for the implementation of Memo, we show how to support different technology backends. As we describe in Section 6, our implementation of Memo supports five backends. We describe the necessary pieces for users to add backends for other memory technologies.

4.1 Memory Mapping with Technology Constraints

Here we present a technique for transforming fungible memories according to technology constraints. The idea is, given a fungible memory and set of specific technology constraints, find an equivalent representation for the fungible memory that satisfies the constraints.

The algorithm has two inputs: a fungible memory and a set of available memory technologies with their constraints. The technology library describes the specifications of each target memory, including data width, number of ports, different features like asynchronous read, write–read forwarding, etc., and a technology-defined cost. The output is a module with certain instantiations of the target memories and wrappers over them to satisfy the fungible memory interface.

Algorithm 1 describes how memory mapping works in two main phases. First, it applies a dynamic programming algorithm to compute the feasible port fitting plan for every memory with i read ports, j write ports, and k read–write ports given the technology library. The basis of this algorithm comes from the rules in Figure 4. The second phase is a greedy algorithm that deals with data width and other memory semantics. It selects memories from the technology library that have enough capacity to hold the data and satisfy the given features. It prefers memories with lower cost and those that already support optimizations such as write–read forwarding. Otherwise, it wraps the memories with additional logic; e.g., it adds forwarding logic when the target memory does not support it. Through dynamic programming, Algorithm 1 is guaranteed to find a solution and that solution will be optimal with respect to cost-per-bit (as specified in the technology's cost model).

Consider an example of mapping a fungible memory with 2 read ports and 2 write ports to a technology library with only one type of BRAM which has 2 read–write ports and costs 1, regardless of their capacity and features. First, the algorithm initializes a 3-dimensional array $f(i, j, k)$ to $+\infty$. Then, it sets $f(0, 0, 2) = 1$, since the target technology has 2 read–write ports. Then it starts from $f(2, 2, 0)$ and recursively searches for satisfying configurations. In order to implement $f(2, 2, 0)$, it needs to use two $f(3, 1, 0)$ by the write port splitting rule. For each $f(3, 1, 0)$, it again needs to use three $f(1, 1, 0)$ by the read port splitting rule where $f(1, 1, 0) = f(0, 0, 2)$. Finally, we have $f(2, 2, 0) = 6$, which means we need to use six 2-read–write-port BRAMs to implement this memory.

Algorithm 1 Port fitting dynamic programming algorithm.

Require: Number of read ports n_r, write ports n_w, read–write ports n_{rw},
Technology list *tech* with fields $(r, w, rw, cost\_per\_bit)$
Ensure: Minimum cost configuration of memories
$n_p \leftarrow n_r + n_w + 2 \times n_{rw}$
Initialize $f[i][j][k] \leftarrow (\infty, \text{None})$ for all $0 \le i, j, k \le n_p$ ▷ Initialization with technology options
for each memory type t in *tech* with (r, w, rw, c) **do**
  for $i = 0$ to r **do**
    for $j = 0$ to w **do**
      for $k = 0$ to rw **do**
        $f[i][j][k] \leftarrow \min(f[i][j][k], (c, t))$
for $k = n_p$ down to 0 **do** ▷ Dynamic programming transitions
  for $i = 0$ to n_p **do**
    for $j = 0$ to n_p **do**
      for $m = 1$ to $i - 1$ **do** ▷ Split read ports
        $f[i][j][k] \leftarrow \min(f[i][j][k], (f[m][j][k].cost + f[i - m][j][k].cost, \text{“split\_r”}))$
      if $k < n_p$ **then**
        if $i > 0$ **then** ▷ Cast read → read–write
          $f[i][j][k] \leftarrow \min(f[i][j][k], (f[i - 1][j][k + 1].cost, \text{“cast\_r”}))$
        if $j > 0$ **then** ▷ Cast write → read–write
          $f[i][j][k] \leftarrow \min(f[i][j][k], (f[i][j - 1][k + 1].cost, \text{“cast\_w”}))$
for $k = 1$ to n_{rw} **do** ▷ Cast read–write → read + write
  $f[n_r][n_w][n_{rw}] \leftarrow \min(f[n_r][n_w][n_{rw}], (f[n_r + k][n_w + k][n_{rw} - k].cost, \text{“cast\_rw”}))$
Reconstruct memory configuration from $f[n_r][n_w][n_{rw}]$
return Optimal cost and configuration

4.2 Extending Memo with Additional Backends

Besides PyRTL simulation, Memo targets specific technologies via backends. Our implementation (Section 6) supports five backends for three technology platforms. To support additional backends, there are two required pieces. The first is a technology “template”; either instantiating the module in HDL code, or generating a script such as for Vivado Tcl or OpenRAM. Figure 7a shows an example of a Verilog template for a single-port BRAM wrapper module. Filling in the template requires mapping the relevant parts from the fungible memory. For example, mapping address and data widths to the `ADDR_WIDTH` and `DATA_WIDTH` parameters; mapping the address port to `A`, the write data to `DIN`, and the read data to `DOUT`.

Second, for mapping scenarios where the memory cannot be directly mapped to the specified technology, constraints metadata is required. The constraints can be derived readily from vendor manuals. Figure 7b is an example of metadata for a dual-port BRAM for a Xilinx FPGA.

Our methodology is applicable to other frontends (HDLs) and backends (memory technologies). The Memo compiler can be extended to support a number of HDLs by interfacing with fungible

```
bram_1rw_wrapper
#(.DEPTH(%s), .ADDR_WIDTH(%s),
  .BITMASK_WIDTH(%s), .DATA_WIDTH(%s))
%s (.MEMCLK(%s), .RESET_N(%s),
    .CE(%s), .A(%s), .RDWEN(%s),
    .BW(%s), .DIN(%s), .DOUT(%s));
```

(a)

```
"xilinx": [ { "name": "xilinx_bram_2rw",
 "cost-per-bit": 3,
 "width": 36, "height": 1024,
 "read_ports": 0, "write_ports": 0,
 "read_write_ports": 2,
 "features": [ "Forward", "Sync" ] }, ...
```

(b)

Fig. 7. Examples of (a) a BRAM wrapper module and (b) metadata for technology mapping.

memories. The underlying rewrites over fungible memories and automated mapping algorithm are intermediate to the HDL and target technology.

5 The Memo Decompiler

Here we present the Memo decompiler with the goal of lifting memories in a gate-level netlist up to fungible memories. The input is a gate-level netlist with single-bit registers (D-flip flops or DFFs). As illustrated in Figure 8, we split the technique into four phases, each driven by rewrite rules in Memo: **(1)** group registers by common enable signals; **(2)** lift read logic from gates to multiplexers to behavioral reads; **(3)** lift write logic from gates to decoders to behavioral writes; **(4)** lift behavioral reads and writes into a fungible memory. In this section, we present the set of rewrite rules that enable memory decompilation, and then how we use equality saturation to decompile memories from netlists with optimizations that obscure the logic. Figure 9 presents a selection of rewrite rules for memory decompilation. Other rules include standard boolean identities for rewriting combinational logic. In the following, we illustrate the key rewrite rules for each phase.

5.1 Register Aggregation

Because Memo supports a multi-level hardware representation, we can use it to directly express designs in terms of gate-level logic as well as high-level abstractions. We start with the gate-level netlist.The first step is to group DFFs into wider registers. Sets of multi-bit registers of the same width will later serve as candidates for memory decompilation. To reduce the search space, we use the heuristic optimization of only considering DFFs with enable pins. This is not required; even if DFFs with enable pins are not specifically used, there is normally some logic which enables reading and writing to registers inside of a memory block for power saving.

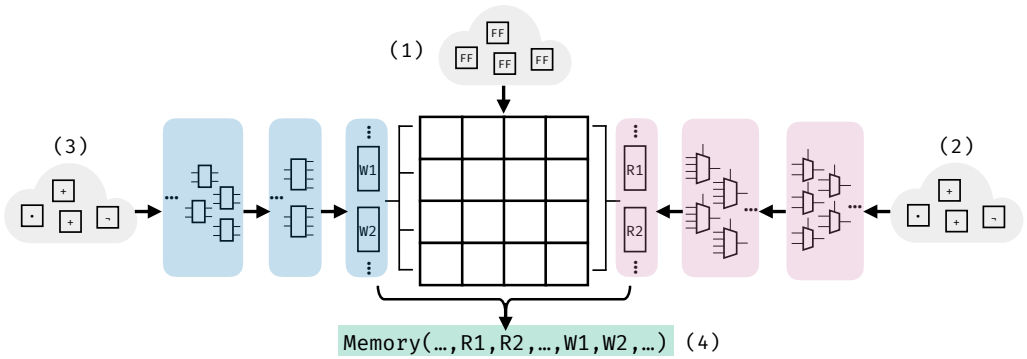


Fig. 8. A visual overview of memory decompilation, split into four phases: (1) Group registers; (2) Identify and reduce multiplexers to form read ports; (3) Identify and reduce decoders to form write ports; and (4) Extract a fungible memory.

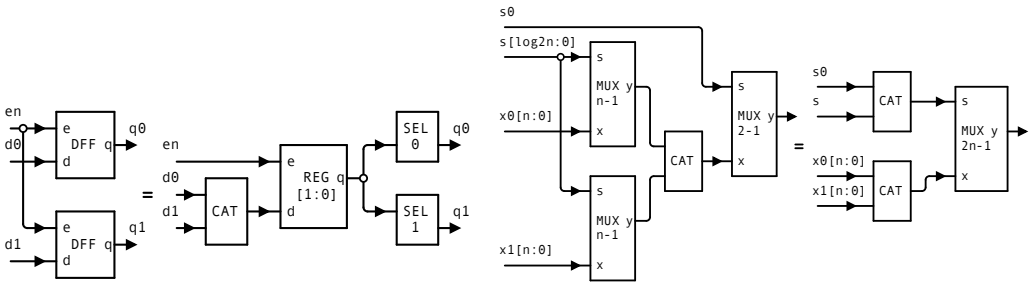
Register Aggregate

$$\text{Reg}(en, d_0 \mathbin{+\!\!+} d_1) = \text{Reg}(en, d_0) \mathbin{+\!\!+} \text{Reg}(en, d_1),$$
$$\text{where } \text{Reg}(en, d_0 \mathbin{+\!\!+} d_1)[0] = \text{Reg}(en, d_0), \text{Reg}(en, d_0 \mathbin{+\!\!+} d_1)[1] = \text{Reg}(en, d_1)$$

Mux Reduce

$$\text{Mux}(s, a \mathbin{+\!\!+} b) = (a \wedge \neg s) \vee (b \wedge s), \text{ where } s, a, b : \mathbb{B}$$
$$\text{Mux}(s_0 \mathbin{+\!\!+} s, x_0 \mathbin{+\!\!+} x_1) = \text{Mux}(s_0, \text{Mux}(s, x_0) \mathbin{+\!\!+} \text{Mux}(s, x_1)), \text{ where } x_0, x_1 : \mathbb{B}^{2^n},\ s : \mathbb{B}^n,\ s_0 : \mathbb{B}$$

Fig. 9. A selection of rewrite rules for memory decompilation. Reg is a constructor for a register (or DFF if the data signal is a single bit) which takes an enable signal and data signal. Mux is a constructor for a multiplexer which takes two arguments: a select signal and signal to select bits from (at least 2 bits in length). $x \mathbin{+\!\!+} y$ denotes concatenation of two bitvector values. $x[i]$ denotes selecting the ith bit from bitvector value x.



(a) (b)

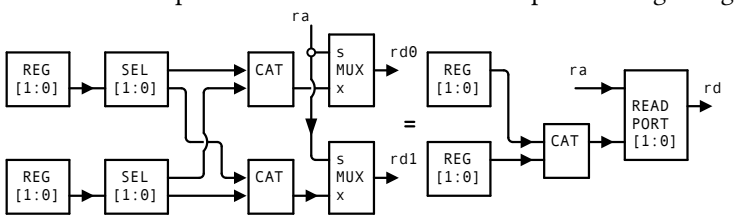
Fig. 10. (a) Illustration of the **Register Aggregate** rule. (b) Illustration of the **Mux Reduce** rule.

From Figure 9, the **Register Aggregate** rule searches for a group of DFFs that are connected to the same enable signal and then groups them into a single register with a wider data input and output. The rule inserts concatenate (CAT) and select (SEL) components to maintain equivalence of inputs and outputs of the original DFFs. Figure 10a illustrates a grouping of two DFFs but this rule generalizes to any number of DFFs with a common enable.[6]

5.2 Read Port Identification

The key idea for identifying read ports is to identify a multiplexer tree that selects a bit from the same position of each register in a register group. To identify multiplexer trees, the **Mux Reduce** rule reduces two n-1 multiplexers and a 2-1 multiplexer into a single $2n$-1 multiplexer, as illustrated in Figure 10b.

A read port consists of multiplexer trees connected to the outputs of a register group:



The illustration demonstrates this rewrite rule for the simplest case of a 1-bit address with two grouped registers. Similar to multiplexer reduction, read port identification is inductive over the widths of the address and grouped registers.

[6]This phase also groups by clock signal, but we omit the clock signals in the rewrite rules for ease of presentation.

5.3 Write Port Identification

Unlike read ports, address decoders form the basis for write ports and are conventionally implemented as and–inverter graphs. An n-bit address decoder is a family of functions $f_{target} : \mathbb{B}^n \to \mathbb{B}$, where $target \in \mathbb{B}^n$ is a constant bit vector representing the target address. The address decoder outputs 1 if and only if the input address matches the target address.

5.3.1 Single write port. In a single-write-port memory, the enable port of the i-th DFF is driven by:

$$f_i(write\_address) \wedge write\_enable.$$

Address decoders are recursively composed:

$$f_{[0]}([b]) = \neg b,\ f_{[1]}([b]) = b,$$
$$f_{target_1 +\!\!+ target_2}(bv_1 +\!\!+ bv_2) = f_{target_1}(bv_1) \wedge f_{target_2}(bv_2)$$

The base case is a single bit decoder. Similar to multiplexer reduction for read ports, we can simply apply reduction to the address decoder and recover the write address and enable signals.

5.3.2 Multiple write ports. The details become more subtle when it comes to memories with multiple write ports due to possible data races. In such cases, if more than one write port is enabled and writes to the same address, the final value in the memory is undefined [1]. With HDLs such as Verilog, users either write a memory that has priorities among write ports, or omit priorities if they are not concerned about non-deterministic behavior.

In open-source EDA tools such as Yosys[38], each write port without user-defined priority is assigned a priority determined by the order of the write ports in the code. Suppose there are k write ports ordered as $w_1, w_2, \ldots, w_k$ by their priorities. Then, the enable port of the DFF corresponding to the ith element in the memory is driven by:

$$\bigwedge_j \left(f_i\left(write\_address_j\right) \wedge write\_enable_j\right),$$

where j, $1 \leq j \leq k$, is the write port index. Then, the data port at position i for write port j is:

$$d_{\{i,j\}} = \mathsf{Mux}\left(f_i\left(write\_address_j\right), d_{\{i,j+1\}}, write\_data_i\right)$$

and $d_{\{i,k\}} = write\_data_k$. In the write port identification process, the decompiler analyzes both enable and data ports for each DFF and checks the common parts of them, which are the address, enable and data signals for each write port.

5.4 Fungible Memory Extraction

With all registers grouped, and read and write ports identified, the last step is to extract the fungible memories. This phase proceeds simply by extracting sets of read and write ports which share the same registers and interfaces.

5.5 Memory Decompilation via Equality Saturation

The general memory decompilation technique presents an ideal scenario. In practice, however, it is insufficient to decompile memories in arbitrary netlists generated from other languages and logic synthesis tools. The reason is that logic synthesis optimizations obscure the original structure of the high-level memory block. Additional transformations may be necessary to reveal the structure, although it is not always obvious which sequence of rewrites will do so.

Solving this problem requires searching over a large space of possible transformations. Our memory decompilation technique overcomes this by using *equality saturation* [32], a non-destructive term rewriting technique which uses the e-graph [22] data structure to group terms into equivalence

classes. Instead of applying a fixed sequence of transformations over a program, equality saturation applies all possible rewrites over the whole program in a convergent process, extracting the "best" term according to a given cost function. Previous work has seen equality saturation used in other EDA tasks such as synthesis and optimization [7, 8, 10, 11, 17, 24, 27, 28, 35, 36]. We give more details on how we specifically apply and adapt this technique for memory decompilation in Section 6.

6 Implementation

Figure 11 illustrates the overall Memo tool flow, with three use cases: (1) The user inputs a design expressed in PyRTL extended with fungible memories (as described in Section 3.4). (2) The user inputs a BLIF netlist [5] to the decompilation flow, to lift the registers in the netlist to fungible memories. (3) The user inputs an XCI XML file for a BRAM IP, which takes an expedited decompilation flow. After elaboration and mapping, Memo targets one of five backends for three technology platforms: Basejump STL (BSG) and OpenRAM for ASIC; Vivado Tcl scripts and synthesizable BRAMs for FPGA; and PyRTL for simulation.

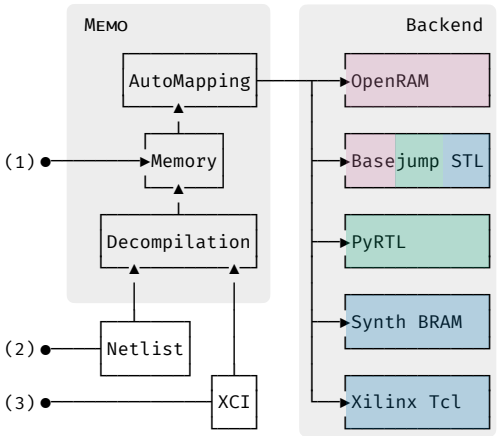


Fig. 11. The Memo tool flow: A user inputs either a design expressed in Memo, a netlist, or XCI file. After rewriting and mapping, Memo targets a particular backend. The backends are shaded by technology platform (pink = ASIC, blue = FPGA, green = simulation).

Our implementation spans two parts: (1) the compiler, and (2) the memory decompiler. We implement the Memo compiler in Python as an extension of the PyRTL compiler with fungible memories, along with the automated mapping algorithm described in Section 4.1. In a departure from conventional equality saturation (eqsat) frameworks, we implement the memory decompiler in SQL (`sqlite3` in Python) following the rules and technique described in Section 5.

6.1 Optimizations for Equality Saturation over Netlists

In the following three paragraphs, we describe three eqsat optimizations—structure-aware scoped rewriting, subsumption, and inverse transformations—which we used to scale to large designs. Finally, we present our reduced eqsat implementation, which allowed us to scale even further.

Structure-aware scoped rewriting. Netlists are large; applying rewrites over netlists in an e-graph can be costly. For example, Memo's general purpose boolean logic rewrites can apply anywhere boolean logic appears in the netlist, which increases rewriting time and grows the netlist unnecessarily. However, because the goal is memory decompilation, we only care about signals whose source or sink is a register. Thus, we scope our general-purpose rewrites to registers' sources or sinks. On the `nerv` benchmark (Table 2), scoped rewriting enabled a speed up in decompilation time from 17 seconds to less than a second.

Subsumption. We use subsumption to further control e-graph size [18, 40]. For example, during register grouping we choose a single representation of the grouped registers, rather than maintain every possible grouping as separate nodes in the e-graph. All other groupings are subsumed.

Inverse transformations. Nandi et al. [21] proposed *inverse transformations,* in which some rewrites insert speculative equalities into the e-graph, which are later validated. We utilize inverse transformations as an alternative technique to write port finding (which we compare against in Section 7). Write ports are generally decomposed into a demultiplexer tree and then aggressively optimized, and are thus difficult to find. For example, RISC-V, ARMv8-A, and MIPS register files have an unwriteable "zero" register, resulting in parts of the demultiplexer tree being optimized away. Rediscovering these paths comes down to profitably "undoing" optimizations to reveal higher-level structure. To do so, we apply an inverse transformation at each read port, which speculatively inserts an "inverse" write port followed by a write port. Using other rewrites, we then attempt to match portions of the write port with expressions in the netlist.

6.2 A Reduced Equality Saturation Implementation

Equality maintenance in eqsat engines is expensive. However, for decompilation purposes, we do not need completeness with regards to finding equalities. Missed equalities simply represent missed decompilation opportunities. Relaxing the need for completeness radically increases our performance by allowing us to skip expensive congruence maintenance steps. Meanwhile, our empirical evaluation in Section 7 clearly demonstrates that Memo can still adequately decompile.

Our reduced eqsat implementation relies on two observations: (1) our initial netlist already tracks a significant number of equalities, (2) we can preserve an adequate level of equality tracking by checking for equivalent expressions before adding a new expression.

Regarding (1), in our experimental setup, we assume netlists were synthesized from Yosys. Internally, Yosys works hard to maximally deduplicate nets, effectively finding equalities. Thus, the netlist synthesized from Yosys already tracks a significant number of equalities.

Regarding (2), equality saturation engines contain generic congruence maintenance data structures (e-graphs) and the routines to maintain them (rebuilding). These data structures can be expensive to maintain when the goal is to find all equalities implied by the rewrite set. For decompiling netlists with millions of gates, full congruence maintenance becomes too expensive and unnecessary. Instead, we maintain partial congruence. Consider a commutative rewrite $op(a, b) = op(b, a)$. Matching on $op(a, b) \rightarrow c$, the decompiler directly inserts $op(b, a) \rightarrow c$ instead of trying to add a new gate $op(b, a) \rightarrow d$ and merging c and d. As an example, on the `sparc_ffu` RISC-V benchmark (Table 2), turning off rebuilding led to a 10x speed up in decompilation time, while still uncovering the same set of memories.

Given these observations, we implement a reduced eqsat using the SQLite database software. Our initial e-graph is simply the output of Yosys's JSON backend, which tracks equalities via its concept of net IDs. Our rewrites are queries into the database (searching for matches via `SELECT`s, inserting new expressions via `INSERT`s). Importantly, rewrites preserve the existing equalities by checking whether a net ID exists before inserting it.

7 Evaluation

In this section, we evaluate Memo and our memory decompiler over a suite of representative hardware design benchmarks. We run all experiments on a workstation running Debian GNU/Linux 12 with an Intel i9-10850K CPU and 64 GB RAM. Through our evaluation, we seek to answer the following questions:

RQ 1. Can fungible memories target multiple technologies across simulation, ASIC, and FPGA platforms?

RQ 2. To what extent does our decompilation-based memory lifting technique identify memories that state-of-the-art memory inference cannot?

Table 1. Memo memory mapping validation tests across a range of port configurations (**1rw**, **1r1w**, and **2rw**) and memory sizes, for a single fungible memory targeting the Vivado Tcl and OpenRAM backends. The port notation '**1r1w**' indicates a memory with 1 distinct read and write port, whereas '**1rw**' indicates 1 port which can either read or write (but not at the same time). $\checkmark_{Tcl}$ and $\checkmark_{OpenRAM}$ stand for validated mapping for the Vivado Tcl and OpenRAM backends respectively.

Memory Size	1rw	1r1w	2rw
32 × 16	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$
32 × 32	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$
32 × 64	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$
32 ×128	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$

RQ 3. How much of a real-world open-source SoC design can be automatically retargeted from one memory technology to another using our composed techniques?

For the memory decompiler, we evaluate two techniques that we call the "baseline" and "heuristic" techniques. The "baseline" technique strictly uses the rewrite rules as described in Section 5 within our reduced-eqsat implementation. The "heuristic" technique further adds inverse transformations as described in Section 6.1 to identify write ports under scenarios where the netlist obfuscates the structure of the memory. This technique is a heuristic because it is possible that the decompiler speculatively introduces a write port where one did not exist in the original netlist. However, as we show in our experimental results, this heuristic correctly identifies write ports, and does so faster than the baseline technique for the largest benchmarks.

7.1 Micro-Benchmark Experiments

We answer RQ 1 and RQ 2 through microbenchmark experiments demonstrating Memo's automated memory technology mapping and memory decompilation.

7.1.1 Memory design in Memo. To answer RQ 1, we port a five-stage RISC-V processor, originally implemented in PyRTL, to use fungible memories. The processor has three memory blocks: a register file, an instruction memory, and a data memory with a write mask. The original design only targets simulation, with no other memory technologies.

With Memo, we shrink the memory implementation from 130 lines of code to *3 lines of code*, one line for each of the three memories. Because PyRTL memory blocks do not natively support write masks, the original design manually implemented the masking logic for the data memory. We can instead specify the entire data memory in one line:

```
dmem = FungibleMemory([Read(addr, rdata)], [Write(addr, wdata, mask)])
```

After porting the RISC-V core to Memo, the implementation still passes a comprehensive set of functional validation tests. Porting the design to Memo also comes with the added benefit of automatically targeting four additional backends. Two (Basejump STL and synthesizable BRAMs) generate Verilog module instantiations. The other two (Vivado Tcl and OpenRAM) generate configurations to be processed by their respective memory compilers. Table 1 shows that these backends generate valid memory configurations that successfully target the given technologies. We validate the configurations by running them through the backend's toolchain and verify that the memories mapped correctly to the technology (the tool will report a failure if the memory cannot be mapped).

7.1.2 Memory Decompilation. To answer RQ 2, we compare our memory decompilation technique against state-of-the-art memory inference in a proprietary synthesis toolchain and Yosys. Note that the comparison is not completely one-to-one because these techniques work through syntactic template-based matching in HDL code, whereas our technique works at the netlist level. Nonetheless,

Table 2. Memory decompilation results comparing against memory inference in Yosys and a state-of-the-art proprietary synthesis toolchain. The **Baseline** result uses the strict reversed rewrites as explained in Section 5. The **Heuristic** result applies heuristics presented in Section 6 to handle netlists with more obfuscated structure. **Gates** are measured in thousands (k). The **Memory** column indicates the expected memory in the design. Decompilation times are given in seconds (s).

Design	Gates	Memory	Baseline		Heuristic		Inference	
			Decompiled	Time	Decompiled	Time	Yosys	Proprietary
bsg_cache	92 k	1rw: (8 × 256) ×8	∅	0.9 s	✓	3 s	∅	∅
bsg_fifo	34 k	1r1w: (64 × 256)	✓	0.3 s	✓	0.7 s	∅	✓
bsg_fifo	527 k	1r1w: (512 × 512)	✓	5 s	✓	23 s	∅	✓
bsg_mem_3r1w	9 k	3r1w: (66 × 32)	✓	0.1 s	✓	0.1 s	✓	✓
nerv	8 k	2r1w: (32 × 32)	✓	0.8 s	✓	0.1 s	✓	✓
pico	10 k	2r1w: (32 × 32)	✓	0.7 s	✓	0.1 s	✓	✓
sparc_ffu	27 k	1r1w: (78 × 128)	✓	0.9 s	✓	0.4 s	✓	∅

we report if the tools' memory inference works over the original HDL code that generated the netlists we run through our memory decompiler. We consider a set of seven micro-benchmarks coming from netlists synthesized from open-source hardware design suites such as OPDB [34] and the Basejump STL [33] which contain a single memory block. Since not all modules in the Basejump STL have memories, we chose the cache, FIFO, and `3r1w` memory as representative micro-benchmarks. More variants of these modules (different sized FIFOs, 1r1w and 2r1w memory blocks) are exercised in the large-scale BlackParrot case study (Section 7.2).

We synthesize the netlists with Yosys, using standard synthesis scripts, including optimizations with ABC [6]. The selection of micro-benchmarks shows the range of memories Memo decompiles, as they are deployed in a variety of designs from caches to cores.

Table 2 presents the results, which show the memory decompiler finds the expected memory with the expected configuration for each micro-benchmark. Noteworthy are `bsg_cache`, `bsg_fifo`, and `sparc_ffu`: *we successfully identify memories where either Yosys or the proprietary synthesis tool, or both, did not.* Because the state of the art infers memories through syntactic template matching, memory inference is sensitive to the implementation. Memory decompilation, by contrast, works on the structure of the netlist.

There are instances where the heuristic approach is slower than the baseline (for example, the 512×512 `bsg_fifo`). For netlists where the cells are almost entirely memories, the baseline approach runs fast. However, as we show next in the larger case studies, netlists with more diverse and unrelated logic (e.g., an entire SPARC core) will slow down the baseline and the heuristics help to cut down the search space.

7.2 Large-Scale Case Studies

To answer RQ 3, we show how Memo memory decompilation enables technology retargeting on real-world open-source SoC designs with many memory blocks. We consider four larger case studies from OPDB with multiple expected memory blocks. Table 3 presents the memory decompilation and retargeting results. Again, the memory decompiler identifies the expected and exact memories for each case study. These results also demonstrate how the memory decompilation technique scales to large designs, with the L2 cache totaling 1.5 million gates. Using the original source code as a reference, the first three decompiled memory blocks for the L2 cache correspond to the port configurations and dimensions of the data, tag, and state memory blocks.

Table 4 provides fine-grain timing data for the case studies. This reveals a bottleneck for the baseline technique in the **Write Ports** phase, where memory decompilation spends a majority of the time. The data also shows the benefits of our equality saturation optimizations (Section 6.1). For

Table 3. Large-scale case studies from OPDB [34]. Since each benchmark contained multiple memories, the table reports them separately in the "Decompiled" column, for the baseline algorithm. Decompilation times are totals for each module including all decompiled memories. Times marked with † use the heuristic technique. ✓ *Tcl* or ✓ *OpenRAM* indicates successful retargeting to Vivado Tcl or OpenRAM, respectively.

Design	Gates	Decompiled	Time (s)	Retargeted
OPDB FFT	194 k	1r1w: (42 × 512) 1r1w: (34 × 128) 1r1w: (32 × 128) 1r1w: (36 × 64)	129 s 6 s†	✓ *Tcl* ✓ *OpenRAM*
OPDB l1.5 cache	242 k	1r1w: (158 × 512) 1r1w: (116 × 128) 1r1w: (8 × 128) 1r1w: (8 × 128)	18 s 13 s†	✓ *Tcl* ✓ *OpenRAM*
OPDB l2 cache	1.5 m	1r1w: (144 × 4096) 2r1w: (170 × 256) 1r1w: (64 × 16) 1r1w: (16 × 4)	71 s 120 s†	✓ *Tcl* ✓ *OpenRAM*
SPARC core	769 k	1r1w: (544 × 256) 1r1w: (576 × 128) 1r1w: (120 × 128) 1r1w: (78 × 128) 1r1w: (151 × 32)	254 s 56 s†	✓ *Tcl* ✓ *OpenRAM*

Table 4. Fine-grain timing results for memory decompilation over the large-scale case studies, comparing the baseline (**Base**) and heuristic (**Heur**) techniques. The phases are: **Load**ing the netlist into the database, **Rewrites** DFFs and base logic gates, **Read Ports** identification via multiplexer tree reduction, **Write Ports** identification, **Extract**ion of the final memory extraction based on the identified read and write ports.

	OPDB FFT		OPDB l1.5		OPDB l2		SPARC Core	
Phase	**Base**	**Heur**	**Base**	**Heur**	**Base**	**Heur**	**Base**	**Heur**
Load	0.3 s	2 s	0.5 s	2 s	4 s	13 s	2 s	7 s
Rewrites	0.1 s	2 s	0.1 s	3 s	0.1 s	4 s	.3 s	28 s
Read Ports	0.4 s	0.1 s	1 s	5 s	8 s	59 s	3 s	11 s
Write Ports	128 s	2 s	16 s	3 s	58 s	44 s	247 s	10 s
Extract	0.3 s	0.001 s	0.7 s	0.001 s	1 s	0.002 s	3 s	0.001 s

Table 5. BlackParrot memory decompilation results using the heuristic technique. The "Corresponding Modules" column maps the decompiled memories to a memory blocks in the source code based on the dimensions and portedness.

Decompiled Memories	**Corresponding Modules**
32×14, 64×184, 512×64 (×8)	Instruction Cache (data, tag, stat)
64×15, 64×184, 512×8 (×64)	Data Cache (data, tag, stat)
512×8, 64×50, 16×43	BHT, BTB, RAS
32×66 (2r1w, 3r1w)	Int & FP Register Files
8×80, 8×174, 4×114	Preissue/Issue/Cmd FIFOs
32×20 (×3), 32×67, 8×15 (×3), 4×10	Uncategorized
Core Size: 11.8 million gates	**Time**: 373 seconds

large netlists with diverse and unrelated logic, such as the OPDB FFT and SPARC Core, the heuristic technique successfully cuts down the search space and accelerates write port identification.

Next, we consider BlackParrot, an open-source RISC-V multicore SoC [23]. Table 5 presents the results from decompiling an entire BlackParrot core, totaling 11.8 million gates. Using the faster, heuristic technique, memory decompilation took 373 seconds, and identified 92 distinct memory blocks. We do not report results for the baseline technique because the memory decompilation did

not finish after two hours. It is noteworthy that memory decompilation *successfully identifies all expected memory blocks in the netlist of the BlackParrot core.*

The BlackParrot core itself is split between front end and back end (although in the netlist, there is no such distinction). Memories in the front end include a BHT (Branch History Table), a BTB (Branch Target Buffer), RAS (Return Address Stack), and 3 smaller queues as part of the PC generation module, as well as 3 memories that make up the instruction cache. Memories in the back end include an integer register file, a floating-point register file, and 3 memories as part of the data cache. The data memory block that is part of the data cache is decompiled as 64 memories of size 512×8, due to the original memory using a byte mask—recovering a common Verilog pattern used in the original source code which implements a byte mask using a `generate` for-loop, instantiating 64 1-byte-wide memories.

8 Limitations and Related Work

This section addresses the limitations of Memo and situates this work with previous research. We divide the discussion between memory compilation and hardware decompilation.

8.1 Memory Compilation

Limitations. Fungible memories model latencies of 0 or 1 cycles and thus only those timing-legal, technology-supported mappings are possible. Based on our experience using 9+ ASIC and FPGA memory compilers, they generally share similar latency limitations, with occasional support for an extra register on the input/output. In a conventional flow, it falls to the designer to correctly compose these basic building blocks to achieve more complex timing behavior. We speculate that Memo could support 2+ cycle latencies through retiming but without this, fungible memories cannot presently, e.g., automatically pipeline a multi-cycle read. Memory banking, as supported by memory compilers [14], splits wide memories into multiple narrower ones, which we represent with the *width* rule (fig. 4) and demonstrate when mapping to FPGA BRAMs. Supporting more fine-grained timing optimizations and architectural cache banking [16] (which effectively parallelizes port access under non-conflicting accesses) are worthy future extensions to fungible memories.

Related Work. Memories in commodity ASIC products are near-universally generated from commercial SRAM compilers. Memory options and configuration ranges vary significantly between technology nodes and even across foundries, and offer very limited design space exploration. Therefore, developers must manually specify memory designs for each compiler (and its supported options) for each project. Developers thus spend great effort optimizing configurations as SRAM often comprises the lion's share of the modern ASIC area. These come alongside a standard PDK during tape-out and so come with significant costs as well as usage restrictions. The Synopsys Generic Memory Compiler [14] is restricted to educational PDKs.

In the open-source space, OpenRAM [15] generates SRAM views in realistic technologies. OpenRAM offers more flexibility but is still restricted (for example, to two ports, necessitating implementation refactoring as in Section 2). Further, designers must still manually optimize configurations as with commercial offerings. In contrast, fungible memories live at the *language level* and are designed to be more flexible *within and across* technologies. In the end, if targeting ASIC, Memo still relies on a memory compiler as the backend target (e.g., OpenRAM), but Memo handles the work of implementing a satisfying memory mapping.

8.2 Hardware Decompilation

Limitations. Similar to memory inference techniques, memory decompilation leverages patterns commonly found in SoC designs. The rewrite-driven process will identify memories in netlists

generated from a process mirroring that of logic synthesis. However, aggressive logic optimizations may obscure memory logic. As we present in Section 6, we apply cutting-edge equational reasoning techniques, as well as some practical heuristics, to overcome this limitation. Nonetheless, the memory decompiler cannot identify memory features for which there are no rewrite rules. We leave it to future work to extend the rewrite rules to identify banking and other optimizations.

Related Work. Prior research in hardware decompilation focused on recognizing repeated logic in a netlist and decompiling it into HDL-level loops, producing more compact HDL code and speeding up simulation time [25]. The hardware loop rerolling technique uses suffix trees to find loop candidates and sketch-guided program synthesis to reroll candidates into valid loops that are semantically equivalent to the original netlist. Our work presents another design-focused application of hardware decompilation: technology retargeting for memories.

Our memory decompilation technique builds on previous research in reverse engineering sequential components in netlists. The most relevant prior work falls in two categories: (1) register aggregation, and (2) memory identification. However, in contrast to reverse engineering, we take steps beyond just *identification* of register and memory components, and also automatically generate high-level abstractions and semantically equivalent HDL code. This enables automated memory technology mapping and retargeting.

8.2.1 Register Aggregation. Register aggregation is the problem of finding multi-bit registers from one-bit D-flip flops in a gate-level netlist [3, 30]. DANA is a netlist reverse engineering framework and presents a general solution for the register aggregation problem [3]. Their technique uses a data-flow analysis to group D-flip flops into multi-bit registers. DANA generates a flip-flop dependency graph and groups flip-flops together based on shared clock and control signals. Grouping proceeds based on a predecessor and successor analysis in the dependency graph. Our register aggregation technique for memory decompilation can be viewed as a specialization of DANA's technique geared towards the structure of registers found in memories.

8.2.2 Memory Identification. Prior reverse engineering work describes a technique for identifying small RAMs and register files in netlists [30, 31]. Our memory decompilation technique mirrors this work's high-level algorithm for outlining the main boundary of the memory block's logic. Due to the regular structure of the synthesized logic for memory blocks, there is a clear pattern to identifying the multiplexer and decoder trees for a memory block's read and write port logic. Memo can identify memory blocks with richer semantics that previous work cannot—distinguishing more read/write port configurations and other memory block features used in common SoC components such as caches and branch predictors. However, we could not directly compare our memory decompilation technique to this past work because it uses a cell library we do not have access to.

9 Conclusion

This paper introduces the concept of a "fungible memory," which formalizes the transformations expert hardware engineers manually undertake to map their memory designs to specific technologies. By formalizing these transformations as behavior-preserving rewrites, fungible memories enable automated memory technology mapping *and retargeting* on real-world, open-source hardware designs. This work advances much-needed improvements for memory abstractions in hardware description languages and enables more automated design flows.

Acknowledgments

The authors thank the anonymous reviewers for their feedback and suggestions, and the online EGRAPHS community for feedback on an early version of this work. This material is based upon

work supported by the National Science Foundation under Award Nos. 2237379 and 2450426. AI programming tools were used in the process of proving several theorems and lemmas in the Lean formalization.

Data Availability Statement

The authors provide an artifact for reproducing the work presented in this paper [26]. The latest version of the Memo source code can be accessed at https://codeberg.org/FRACAS/memo.

References

[1] 2001. IEEE Standard Verilog Hardware Description Language. *IEEE Std 1364-2001* (2001), 1–792. https://doi.org/10.1109/IEEESTD.2001.93352

[2] Ameer M. S. Abdelhadi and Guy G. F. Lemieux. 2016. A Multi-ported Memory Compiler Utilizing True Dual-Port BRAMs. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 140–147. https://doi.org/10.1109/FCCM.2016.45

[3] Nils Albartus, Max Hoffmann, Sebastian Temme, Leonid Azriel, and Christof Paar. 2020. DANA - Universal Dataflow Analysis for Gate-Level Netlist Reverse Engineering. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020, 4 (Aug 2020), 309–336. https://doi.org/10.13154/tches.v2020.i4.309-336

[4] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[5] UC Berkeley. 1992. Berkeley logic interchange format (BLIF). *Oct Tools Distribution* 2 (1992), 197–247.

[6] Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–40. https://doi.org/10.1007/978-3-642-14295-6_5

[7] Chen Chen, Guangyu Hu, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. 2025. E-morphic: Scalable Equality Saturation for Structural Exploration in Logic Synthesis. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. 1–7. https://doi.org/10.1109/DAC63849.2025.11133110

[8] Chen Chen, Guangyu Hu, Dongsheng Zuo, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. 2024. E-Syn: E-Graph Rewriting with Technology-Aware Cost Functions for Logic Synthesis. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 124, 6 pages. https://doi.org/10.1145/3649329.3656246

[9] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–7. https://doi.org/10.23919/FPL.2017.8056860

[10] Samuel Coward, George A. Constantinides, and Theo Drane. 2022. Automatic Datapath Optimization using E-Graphs. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*. 43–50. https://doi.org/10.1109/ARITH54963.2022.00016

[11] Samuel Coward, Theo Drane, and George A. Constantinides. 2024. ROVER: RTL Optimization via Verified E-Graph Rewriting. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 12 (2024), 4687–4700. https://doi.org/10.1109/TCAD.2024.3410154

[12] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28 (Lecture Notes in Computer Science, Vol. 12699)*. Springer, 625–635.

[13] RISC-V Foundation. 2025. *The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture*. RISC-V Foundation.

[14] R. Goldman, K. Bartleson, T. Wood, V. Melikyan, and E. Babayan. 2014. Synopsys' Educational Generic Memory Compiler. In *10th European Workshop on Microelectronics Education (EWME)*. 89–92. https://doi.org/10.1109/EWME.2014.6877402

[15] Matthew R. Guthaus, James E. Stine, Samira Ataei, Brian Chen, Bin Wu, and Mehedi Sarwar. 2016. OpenRAM: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–6. https://doi.org/10.1145/2966986.2980098

[16] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture: A Quantitative Approach* (sixth ed.). Morgan Kaufmann.

[17] Matthew Hofmann, Berk Gokmen, and Zhiru Zhang. 2025. EqMap: FPGA LUT Remapping using E-Graphs. In *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9. https://doi.org/10.1109/ICCAD66269.2025.11240672

[18] Gerhard Köstler, Werner Kießling, Helmut Thöne, and Ulrich Güntzer. 1995. Fixpoint iteration with subsumption in deductive databases. *Journal of Intelligent Information Systems* 4, 2 (1995), 123–148. https://doi.org/10.1007/BF00961871

[19] Charles Eric Laforest, Zimo Li, Tristan O'rourke, Ming G. Liu, and J. Gregory Steffan. 2014. Composing Multi-Ported Memories on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 7, 3, Article 16 (Sept. 2014), 23 pages. https://doi.org/10.1145/2629629

[20] Charles Eric LaForest and J. Gregory Steffan. 2010. Efficient multi-ported memories for FPGAs. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '10)*. Association for Computing Machinery, New York, NY, USA, 41–50. https://doi.org/10.1145/1723112.1723122

[21] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

[22] Charles Gregory Nelson. 1980. *Techniques for program verification.* Stanford University.

[23] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael Bedford Taylor. 2020. BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro* 40, 4 (2020), 93–102. https://doi.org/10.1109/MM.2020.2996145

[24] Yan Pi, Hongji Zou, Tun Li, Wanxia Qu, and Hai Wan. 2023. ESFO: Equality Saturation for FIRRTL Optimization. In *Proceedings of the Great Lakes Symposium on VLSI 2023* (Knoxville, TN, USA) *(GLSVLSI '23)*. Association for Computing Machinery, New York, NY, USA, 581–586. https://doi.org/10.1145/3583781.3590239

[25] Zachary D. Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. 2023. Loop Rerolling for Hardware Decompilation. *Proc. ACM Program. Lang.* 7, PLDI, Article 123 (jun 2023), 23 pages. https://doi.org/10.1145/3591237

[26] Zachary D. Sisco, Sijie Kong, Daniel Ruelas-Petrisko, Jingtao Xia, Julian Springer, Varun Rao, Spencer Wang, Gus Henry Smith, Ben Hardekopf, and Jonathan Balkind. 2026. Artifact for Fungible Memories for Automated Technology Mapping and Retargeting. ACM. https://doi.org/10.5281/zenodo.19090603

[27] Gus Henry Smith, Colin Knizek, Daniel Petrisko, Zachary Tatlock, Jonathan Balkind, Gilbert Louis Bernstein, Haobin Ni, and Chandrakana Nandi. 2024. Scaling Program Synthesis Based Technology Mapping with Equality Saturation. *arXiv preprint arXiv:2411.11036* (2024).

[28] Gus Henry Smith, Zachary D Sisco, Thanawat Techaumnuaiwit, Jingtao Xia, Vishal Canumalla, Andrew Cheung, Zachary Tatlock, Chandrakana Nandi, and Jonathan Balkind. 2024. There and Back Again: A Netlist's Tale with Much Egraphin'. In *Workshop on Languages, Tools, and Techniques for Accelerator Design.* https://doi.org/10.48550/arXiv.2404.00786

[29] Wilson Snyder. 2024. Verilator. https://www.veripool.org/verilator/.

[30] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascon, W. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik. 2014. Reverse Engineering Digital Circuits Using Structural and Functional Analyses. *IEEE Transactions on Emerging Topics in Computing* 2, 01 (jan 2014), 63–80. https://doi.org/10.1109/TETC.2013.2294918

[31] Pramod Subramanyan, Nestan Tsiskaridze, Kanika Pasricha, Dillon Reisman, Adriana Susnea, and Sharad Malik. 2013. Reverse engineering digital circuits using functional analysis. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1277–1280. https://doi.org/10.7873/DATE.2013.264

[32] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. *SIGPLAN Not.* 44, 1 (jan 2009), 264–276. https://doi.org/10.1145/1594834.1480915

[33] Michael Bedford Taylor. 2018. BaseJump STL: SystemVerilog Needs a Standard Template Library for Hardware Design. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 73, 6 pages. https://doi.org/10.1145/3195970.3199848

[34] Georgios Tziantzioulis, Ting-Jung Chang, Jonathan Balkind, Jinzheng Tu, Fei Gao, and David Wentzlaff. 2022. OPDB: A Scalable and Modular Design Benchmark. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 6 (2022), 1878–1887. https://doi.org/10.1109/TCAD.2021.3096794

[35] Ecenur Ustun, Ismail San, Jiaqi Yin, Cunxi Yu, and Zhiru Zhang. 2022. IMpress: Large Integer Multiplication Expression Rewriting for FPGA HLS. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 1–10. https://doi.org/10.1109/FCCM53951.2022.9786123

[36] Andy Wanna, Samuel Coward, Theo Drane, George A. Constantinides, and Miloš D. Ercegovac. 2023. Multiplier Optimization via E-Graph Rewriting. In *2023 57th Asilomar Conference on Signals, Systems, and Computers*. 1528–1533. https://doi.org/10.1109/IEEECONF59524.2023.10476812

[37] Claire Wolf. 2024. Memory handling. https://yosyshq.readthedocs.io/projects/yosys/en/latest/using_yosys/synthesis/memory.html.

[38] Claire Wolf. 2024. Yosys Open SYnthesis Suite. https://www.yosyshq.com.

[39] Xilinx. 2024. UltraScale Architecture Memory Resources. https://docs.amd.com/v/u/en-US/ug573-ultrascale-memory-resources.

[40] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI, Article 125 (June 2023), 25 pages. https://doi.org/10.1145/3591239

Received 2025-11-14; accepted 2026-04-03