2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS) | 978-1-6654-0162-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICECCS54210.2022.00013

Parameterized Design and Formal Verification of Multi-ported Memory

Mufan Xiang*, Yongjian Li†‡[1] , Sijun Tan†, Yongxin Zhao*[1], Yiwei Chi†
* Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
† State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
‡ School of Intelligent Science and Technology, Hangzhou Institute for Advanced Study, UCAS

***Abstract*—Multi-ported memories are essential modules to provide parallel access for high-performance parallel computation systems such as VLIW and vector processors, etc. However, the design of multi-ported memories are rather complex and error-prone, which usually causes the high implementation cost. Therefore, the designs and verification of multi-ported memories become challenging. In this paper, we firstly present a modular and parameterized approach based on Chisel to design and implement multi-ported memory concisely. Furthermore, to verify the correctness of the design, we formalize properties of multi-write-read operations of the memories by generalized symbolic trajectory assertion (GSTE) graphs and verified them by two kinds of approaches: SystemVerilog Assertions-based, and GSTE-based approaches. Our verification through SVA and STE/GSTE successfully finds an error caused by misusing one parameter in our high-level design.**

***Index Terms*—Hardware Formal Verification, Parameterized Design, Multi-ported Memory, Chisel**

I. Introduction

Multi-ported memories are essential modules for high-performance computation systems that require frequent communication, sharing, queuing, and synchronization among distributed functional units and compute nodes [1]. Modern processors such as VLIW, CGRAs, DSPs, CMPs need multi-ported memories to support concurrent access [2]. However, the high cost of multi-ported memory implementation prevents FPGA vendors from providing more complex designs than dual-ported block RAMs. Therefore, the method of designing efficient multi-ported memory from dual-ported block memory is important as it frees FPGA vendors from having to include hard BRAMs with more than two ports in their fabrics.

In essence, the design idea of multi-ported memories is to trade space for function, which implements simultaneous reading and writing of multiple data in a unified address space through multiple memory banks. This makes the data storage control logic of multi-ported storage more complex and the signal wiring very complicated. It is very difficult to design a multi-ported module from bottom to top by means of a structured module building method. Like previous work by Ameer M. S. Abdelhadi et al [2]), they summarized and proposed three multi-ported memory designs and implemented them using Verilog. Rather than using Verilog, we propose to use Chisel to design multi-ported memories, which makes the design concise, parameterized and modular.

[1]Corresponding authors: Yongjian Li, Yongxin Zhao

Chisel is a new hardware construction language that supports advanced hardware design with the aid of highly parameterized generators and layered domain-specific hardware languages. Embedded in the Scala programming language, which natively supports object-orientation, functional programming, parameterized types and type inference, Chisel helps to raise the level of hardware design abstraction [3].

At the same time, the correctness verification of multi-ported memories is important but tricky. The complexity of the design determines the complexity of verification. In fact, the complexity of multi-ported memory verification is proportional to the product of the number of addresses times the number of write ports times the number of read ports. Here, we used two kinds of verification approaches on the multi-ported memory modules: SystemVerilog Assertion-based, and GSTE-based.

At present, assertion-based verification (ABV) is gradually recognized as a key methodology to tackle functional verification challenges in the processes of software and hardware development [4]. SystemVerilog Assertions (SVA) is a set of tools that allow engineers to integrate ABV into their hardware designs. And SVA has a rich syntax such as sequences, properties, and assertions to specify logical and sequential relationships [5]. However, verification always starts with Verilog or other low-level HDLs and then uses SymbiYosys [6] to convert the SVAs into the SMT2 [7], which will be checked by an SMT solver, like z3 or Boolector. And there is currently no formal verification method directly from Chisel. In this paper, we modeled the parametric Chisel module and designed the assertions, and then check them.

Symbolic Trajectory Evaluation (STE) is a typical implementation of symbolic simulation, that has been widely used to verify hardware models [8]. By now, STE has been extensively applied in verifying the properties of circuits containing large data paths, such as memories and FIFOs. However, STE is restricted to verify properties within finite time steps and only involving forward reasoning [9]. Generalized STE (GSTE) is an extension of STE that can verify all ω-regular properties with the aid of fix-point iterations and backward symbolic simulation. GSTE has introduced the notion of *assertion graph* to represent properties with much expressive ability. The backend of STE/GSTE is implemented by BDD engine.

In this paper, we present a method for parametrically designing hardware in Chisel and formal verification from Chisel. We use Chisel to design circuits promptly, and extend

978-1-6654-0162-3/22/$31.00 ©2022 IEEE
DOI 10.1109/ICECCS54210.2022.00013

Authorized licensed use limited to: East China Normal University. Downloaded on September 20,2022 at 07:22:33 UTC from IEEE Xplore. Restrictions apply.

Chisel with Chisel-based assertion (CA) to specify circuits at the same time. We get an insight into standard multi-ported memory designs starting from ordinary dual-ported block memory and implemented them in Chisel. Besides, we verified the correctness of the implementation through formal methods. To the best of our knowledge, our work is the first one to design multi-ported memories by Chisel and then use STE/GSTE to verify them. The contributions of this work are the followings:

- **Designing by Chisel.** We parameterized three different versions of multi-ported memories and implemented them using Chisel: live-value-table (LVT), XOR, and invalidation-based live-value-table (I-LVT) designs (the detailed implementation of LVT-based multi-ported memory is in this paper, and see the others in GitHub repository [10]). With the help of high-level programming language features, our multi-ported memory designs have higher readability, modularity, and extendability.
- **Formal Verification.** We formalized properties of multi-write-read operations of the memories by GSTE assertion graphs and verify them by two kinds of approaches: SystemVerilog Assertions-based, and STE-based approaches. We use SVA to describe the properties and use SymbiYosys to check SVA. At the same time, we adopt a formal method and leverage the power of the symbolic method to solve the verification scale problems caused by the number of ports and addresses. By verification through STE/GSTE, we can get more insights into the correctness of parameterized multi-ported memory designs through writing invariants tagged with nodes in assertion graphs.

The rest of the paper is organized as follows. Proposing our methodology for designing and verifying in Section II. Designing multi-ported memories in Chisel is proposed in Section III. The verification of memory is given in Section IV. And the experiments are shown in Section V. Related work of multi-ported memory design and verification is in Section VI. Section VII concludes the paper and introduces the future work.

II. Methodology

Our methodology is shown in Figure 1.

The cornerstone of our methodology is the integration of agile design with formal verification. The former is achieved by adopting Chisel to design circuits, while the latter is achieved by extending Chisel with assertions and compiling Chisel designs and assertions into low-level models and properties such as SVA assertions and AIG netlists and using different formal tools to verify them.

Chisel offers features for agile design. Unlike using Hardware Description Languages (HDLs), which is the time-consuming process of designing and implementing circuits. Chisel is one of the Hardware Construction Languages (HCLs), which tries to bring high-level software principles into a hardware-level abstraction, to provide advanced configuration and generation capabilities. Designing generators instead of describing circuits proves to be an efficient approach that fully exhibits its agility as soon as successive design iterations occur. Chisel integrates powerful software engineering concepts into hardware development, unlocking higher abstraction levels while still mastering generated hardware. It can be easier to build highly reusable design libraries with HCLs, like Chisel. [11].

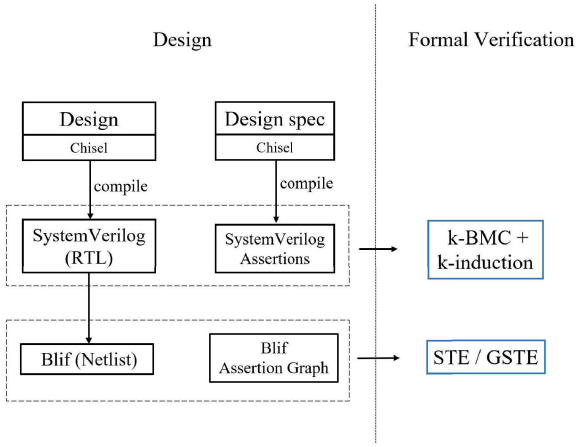

Fig. 1: Integration of Design and Verification

However, Chisel does not support the verification, so we extend Chisel with features of assertions. As shown in Figure 1, Formal Check is based on SystemVerilog Assertions. While Chisel compiler does not support generating assertions, which describes the temporal properties of the design. We implemented Chisel's extension, added syntax support for assertions, and converted assertions to SystemVerilog Assertions when Chisel compiles. Now, Chisel is not only a hardware generator but also an abstract level, assertion-based simulation, and formal verification tool. Although we are now using it to generate SVA to formally verify the design in HDLs.

In our verification part, we first should understand the hardware design requirements, and then formalize the properties of the module, and use Chisel to design parameterized specifications. Since the Spec is implemented using Chisel, it is easy to instantiate with the configuration corresponding to the concrete design. Furthermore, we can design the parameterized specification decoupling from the specific internal design. And we can use the specification to verify modules with the same requirements of various different designs.

Combining the instantiation specification (SVA) and the module to be verified, we then introduce two kinds of approaches. The first one is to synthesize SystemVerilog into a netlist structure and then verify using STE/GSTE technology, and the other is using SymbiYosys to check SVA. In SymbiYosys, we can use the combination of k-BMC and k-induction to prove the correctness of the hardware transition system.

Next, we apply this method to the design and formal verification of multi-ported memory.

Authorized licensed use limited to: East China Normal University. Downloaded on September 20,2022 at 07:22:33 UTC from IEEE Xplore. Restrictions apply.

III. Parameterized Design

This section proposes the detailed implementation of LVT-based multi-ported memory in Chisel. See the GitHub repository [10] for the other two designs.

The multi-ported memory design given in this paper is based on multiple dual-ported memories. It is mainly instantiated with the Mem() encapsulated in the Chisel language, which provides a parameterized implementation of the dual-ported memory. The Chisel module of Memory is shown as follows:

```
1 // a dual-ported memory with write-enable
      signal.
2 // c: number of the memory unit.
3 // w: length of a memory unit.
4 class Memory(c: Int, w: Int) extends Module {
5   // calculate the length of address.
6   val nw = (math.log(c)/math.log(2)).toInt;
7   val io = IO(new Bundle{
8     val rdAddr = Input(UInt(nw.W))
9     val rdData = Output(UInt(w.W))
10    val wrEna = Input(Bool())
11    val wrData = Input(UInt(w.W))
12    val wrAddr = Input(UInt(nw.W))
13  })
14  val mem = Mem(c, UInt(w.W))
15  when(io.wrEna) {
16    mem.write(io.wrAddr, io.wrData)
17  }
18  io.rdData := mem.read(io.rdAddr)
19 }
```

As shown in the code, first we need to calculate the bit length required by the address according to the c parameter (line 6). The main logic is to write data under the control of the enable signal and read the data in the Mem() by wrAddr at the same time (line 16-18).

In this work, we use Chisel to implement three kinds of multi-ported memory with M write ports and N read ports. And the basic I/O of the implemented module is shown in Figure 2.

A. LVT-based Approach

The multi-ported memory implementation based on Live Value Table (LVT) is actually an extension of the single write and multi-read memory implementation. In multi-read memory, the whole memory bank can be replicated while keeping common write address and data. Any data will be written to all bank replicas, so reading from any bank is equivalent. In the LVT-based design, every write port is associated with a group of bank replicas, and every read port should connect all the groups of bank replicas. The example design of 2 read and 2 write ports memory is shown in figure 3.

The figure 3 leaves out the write address input, enable signal input of each write port, and the read address input of each read port. It can be seen from the design that it is troublesome for read ports to determine which memory block to be read, since two sets of memory for writing data do not maintain data consistency. Therefore, a mapping table, namely Live Value Table(LVT), is introduced to store the ID of the bank replica that holds the latest data. As depicted in Figure 4, while reading

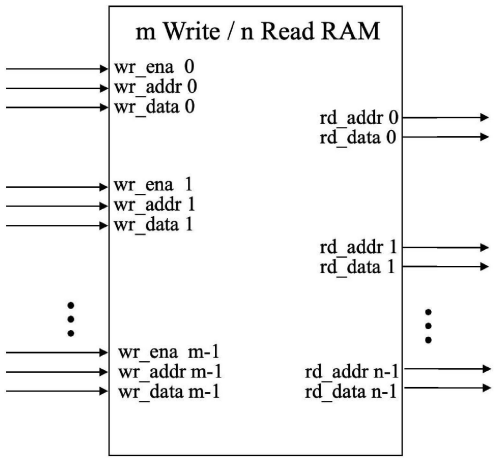

Fig. 2: M Write Ports / N Read Ports Memory

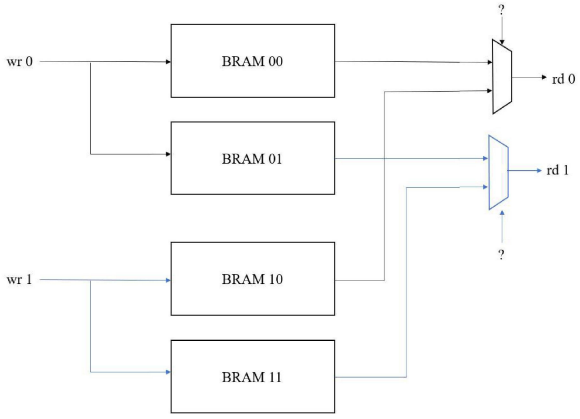

Fig. 3: Preliminary Design of LVT Multi-ported Memory

the data on a certain address, the ID of the bank that stores the latest version of data at that address can be retrieved from LVT. The design is shown in Figure 4.

And according to this example, it can be concluded that M times N block memories are required for the M write N read LVT memory.

We implement the design of LVT-based memory in Chisel. And the module of Live Value Table is shown as follow:

```
1 // M: M write ports.
2 // N: N read ports.
3 // size: the max address + 1.
4 // w: length of a memory unit.
5 class LiveValueTable(M: Int, N: Int, size:
      Int, w: Int) extends Module {
6   val io = IO(new Bundle{
7     val wrAddr = Input(Vec(M, UInt(w.W)))
8     val wrEna = Input(Vec(M, Bool()))
9
10    val rdAddr = Input(Vec(N, UInt(w.W)))
```


Authorized licensed use limited to: East China Normal University. Downloaded on September 20,2022 at 07:22:33 UTC from IEEE Xplore. Restrictions apply.

```
11    // calculate the length to store the ID
          of bank.
12    val rdIdx = Output(Vec(N, UInt(math.ceil(
          math.log(M) / math.log(2)).toInt.W)))
13  })
14
15  // initialize the LVT Registers.
16  val lvtInitArray = new Array[Int](size)
17  for(i <- 0 until size) {
18    lvtInitArray(i) = 0
19  }
20  val lvtReg = RegInit(VecInit(lvtInitArray.
        map(_.U(math.ceil(math.log(M) / math.log
        (2)).toInt.W))))
21
22  // store the ID.
23  for(i <- 0 until M) {
24    when(io.wrEna(i) === true.B) {
25      lvtReg(io.wrAddr(i)) := i.U
26    }
27  }
28
29  // output the ID.
30  for(i <- 0 until N) {
31    io.rdIdx(i) := lvtReg(io.rdAddr(i))
32  }
33}
```

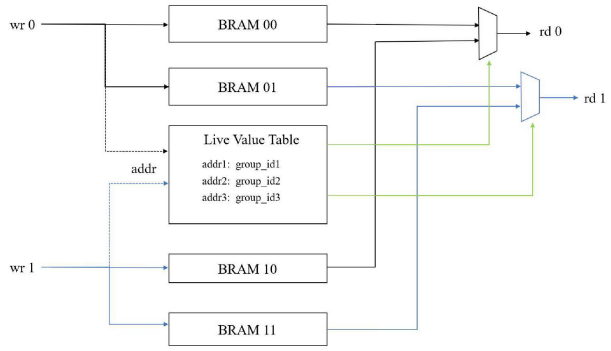

Fig. 4: Multi-ported Memory Design based on LVT

As shown in the above code, LiveValueTable maintains a mapping from the address to the ID of the most recently updated Mem Bank (at this address) through the register array lvtReg (defined in line 11). When data is written to the memory, through wrAddr and wrEna signal(line 23-27), write ID into lvtReg; when reading data, read the ID from the LiveValueTable according to the rdAddr signal and output it through the rdIdx signal (line 30-32)

An LVT is implemented as a set of registers and can be accessed by multiple ports at the same time. There is no restriction of a single read and write port like Block RAM. And the overall implementation of LVT-based multi-ported memory is shown in the following code:

```
1 // M: M write ports.
2 // N: N read ports.
3 // size: the max address + 1.
4 // w: length of a memory unit.
5 class LVTMultiPortRams(M: Int, N: Int, size:
      Int, w: Int) extends Module{
6   // calculate the length of address.
7   val addrW: Int = math.ceil(Math.log(size) /
        math.log(2)).toInt
8   val io = IO(new Bundle{
9     val wrAddr = Input(Vec(M, UInt(addrW.W)))
10    val wrData = Input(Vec(M, UInt(w.W)))
11    val wrEna = Input(Vec(M, Bool()))
12
13    val rdAddr = Input(Vec(N, UInt(addrW.W)))
14    val rdData = Output(Vec(N, UInt(w.W)))
15  })
16  // initialize the Block Memroys and LVT.
17  val mems = VecInit(Seq.fill(M * n)(Module(
        new Memory(size, w)).io))
18  val lvt = Module(new LiveValueTable(M, n,
        size, addrW))
19  // input of mems
20  for(i <- 0 until M) {
21    for(j <- 0 until N) {
22      mems(i * n + j).wrEna := io.wrEna(i)
23      mems(i * n + j).wrAddr := io.wrAddr(i)
24      mems(i * n + j).wrData := io.wrData(i)
25      mems(i * n + j).rdAddr := io.rdAddr(j)
26    }
27  }
28  // input of lvt
29  for(i <- 0 until M) {
30    lvt.io.wrEna(i) := io.wrEna(i)
31    lvt.io.wrAddr(i) := io.wrAddr(i)
32  }
33  for(i <- 0 until N) {
34    lvt.io.rdAddr(i) := io.rdAddr(i)
35  }
36  // output rdData rely on sel.
37  for(i <- 0 until N) {
38    val sel = lvt.io.rdIdx(i)
39    io.rdData(i) := mems(sel * n.U + i.U).
          rdData
40  }
41}
```

As shown in the code above, the implementation conforms to the LVT-based memory structure designed in figure 4. First, calculate the length of the address field (line 7), and define the input and output (lines 8-14). Instantiate M times N Memory() modules (defined previously) with the right parameters (line 17), instantiate the LiveValueTable() module (line 18), and then wire the input and output (line 20-40). Note that when writing data, the address and the enable signal need to be input to the LiveValueTable module at the same time; and when reading, the LiveValueTable helps to steer the read data out of the right bank by holding the ID of the last write bank for each address (line 37-40).

Compared with the Verilog version of the multi-ported memory module, our parameterized module in Chisel has a clearer structure and higher extendability. Each module is a class in Scala and can be easily instantiated or reused with

Authorized licensed use limited to: East China Normal University. Downloaded on September 20,2022 at 07:22:33 UTC from IEEE Xplore. Restrictions apply.

parameters. In Chisel, the hardware details are hidden, so that the development work is focused on the description of the design. In our work, Chisel helps to reduce the amount of code in Verilog [12] by nearly 70%.

IV. Formal Verification

This section gives formal properties of multi-write-read operations of the memories by GSTE assertion graphs and verifies them by two kinds of approaches.

A. Our Verification Method

First, we give the multi-ported memory specification described in natural language.

> *Write data to a certain address of the memory, after a certain period of time without writing to the address, and then read the address, the data will remain consistent.*

Here we use an GSTE assertion graph to formalize the properties of the memories. Formally,

definition 1 *A tagged assertion graph is a five-tuple* $G \equiv (V, init, E, ant, cons, \mu)$*, where* V *is a set of vertices containing a vertex* $init$ *which is called the initial vertex;* E *is a set of edges. Each edge is a pair of vertices. Finally,* ant *and* $cons$ *are two functions from edge to formula.* $\mathsf{ant}(e)$ *is the antecedent of* e*,* $\mathsf{cons}(e)$ *the consequent of* e*. A tag function* μ *assigns a formula to a vertex* n*. For an edge* $e = (n_1, n_2)$*, we define* $\mathsf{source}(e) \equiv n_1$*, and* $\mathsf{sink}(e) \equiv n_2$*.*

A vertex represents a symbolic state which can be reached from the initial vertex, and an edge a symbolic transition. The antecedent of the edge formulates the stimulus enforced to the input nodes of the circuit, and the consequent of the edge specifies the values expected on circuit nodes as a response. A tagged invariant $\mu(n)$ stands for a constraint specification for states which are represented by n.

We use the method of tagged invariants proposed in [13] to verify these graphs.

definition 2 *Let* $G = (V, init, E, ant, cons)$ *be an assertion graph,* M *be a transition function of a circuit.* G *is called to be generally inductive w.r.t. a circuit* M *if the following condition holds:* $\mathsf{induct}(G, M, \mu)$ *if and only if for all* $e \in E$*, for any* s*,*

$$M \models ant(e) \wedge \mu(\mathsf{source}(e)) \longrightarrow cons(e) \wedge \mathsf{O}(\mu(\mathsf{sink}(e))) \quad (1)$$

Here $\mathsf{O}(f)$ is the simple LTL-formula operator NEXT (f at next time), $M \models f$ means transition function implies that f holds.

In order to verify (1), we use two different approaches:

1) SVA-based assertions:
   assume $(@posedge\ clk) ant(e) \&\& \mu(\mathsf{source}(e))$
   assert $(@posedge\ clk) cons(e) \&\&\#\#1 \mu(\mathsf{sink}(e))$.
   We use SymbiYosys [6] to check SVA, however, SymbiYosys has limited support for symbolic simulation. Symbolic constants representing ports, addresses, and data are still not supported, so we need to do a case split on these constants, and verify each case; Besides, SVA checking needs to be carried out at the Verilog code which is automatically generated from Chisel source.
2) STE-based assertions:
   $\mathsf{CSTE}\ < \mathsf{ant}', \mathsf{cons}', \mathsf{cstr} >$, where ant' and $cons'$ are assertions to encode symbolic values assigned to the nodes of the circuit at the initial and next time, while $cstr$ is a Boolean formula to encode constraint which should be satisfied by the symbolic values. We use CSTE command in Forte [14] to check STE assertions, and symbolic simulation is fully supported. The verification based on STE is done on the blif netlist generated from Verilog code.

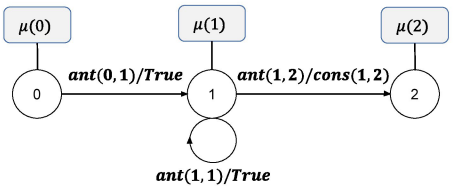

Fig. 5: Multiport Memory Assertion Graph

B. GSTE assertion graphs of the multi-poted memories

Roughly speaking, a topological structure of an assertion graph of a multi-ported memory design is shown in Fig 5, where $V = \{0, 1, 2\}$. Namely, at the first moment after initialization, 1)edge $(0, 1)$ represents a writing phase: in the port $\overline{k}$, data $\overline{dataIn}$ at the addresses $\overline{addr}$; 2) edge $(1, 1)$ represents a holding phase: and then in any cycle, there is no write to the address (or no write operation, or to other addresses Write operation); 3) edge $(1, 2)$ represents a reading phase: the content of address $\overline{addr}$ on port $\overline{l}$, we need to confirm that the read data is $\overline{dataIn}$. By different verification schemes, we define according to different antecedents and consequents of edges of the graph; By different designs, we define different tagged invariants of all the nodes which formalize the intuitions of these designs.

In this work, we do both verifications of a write-read operation at a single port with a single Address and that of a write-read operation at full ports with full address. Here we only show the latter. Here the word "full" means that we write at all the write ports with different addresses and read the written data from all read ports.

- edge $(0, 1)$: a writing action.
  $ant(0, 1) \equiv$

$$\bigwedge_{k=0}^{M-1} wrAddr(k) = \mathsf{addr_k} \wedge wrEna(k) = \mathsf{True} \wedge wrData(k) = \mathsf{dataIn_k} \wedge \bigwedge_{i=0}^{M-1} \left(\bigwedge_{j=0}^{M-1} j \neq i \rightarrow \mathsf{addr_j} \neq \mathsf{addr_i} \right) \quad (2)$$

  $ant(0, 1)$ is the stimula to write some data dataIn from different ports with different addresses addr in parallel;

Authorized licensed use limited to: East China Normal University. Downloaded on September 20,2022 at 07:22:33 UTC from IEEE Xplore. Restrictions apply.

$cons(0,1)$, and $\mu(0)$ are just $true$ which represents no constraint, and for all kinds of multi-ported memories, their $ant(0,1)$ ($cons(0,1)$) are the same. However, $\mu(1)$ is the constraint which represents the state of the memory or live-table after the action; As shown in later parts, different designs have different tagged functions which represent different design intuitions.

- edge $(1,1)$: a holding action.
  $ant(1,1) \equiv$

$$\bigwedge_{i=0}^{M-1}(wrEna(i) = \mathsf{False} \vee (\bigwedge_{k=0}^{M-1} wrAddr(i) \neq \mathsf{addr_k})) \tag{3}$$

  $ant(1,1)$ is the stimula that no new data is written into the above address addr, then data is held by the memory, thus $\mu(1)$ still holds;
- edge $(1,2)$: a reading action.
  $ant(1,2)$ is the stimula that content of data can be read by providing the according address addr is provided at some prot l.
  $ant(1,2) \equiv$

$$\bigwedge_{l=0}^{N-1} rdAddr(l).rdAddr = \mathsf{addr}'_\mathsf{l} \tag{4}$$

  cons:

$$\bigwedge_{l=0}^{N-1}\bigwedge_{k=0}^{M-1}(\mathsf{addr}'_\mathsf{l} = \mathsf{addr_k} \rightarrow rdData(l) = dataIn_k) \tag{5}$$

For all kinds of designs, $\mu(0)$ and $\mu(2)$ are simply true, different designs have different tagged constraints $\mu(1)$ for node 1:

For LVT-based one: $\mu_{lvt}(1) =$

$$\bigwedge_{k=0}^{M-1}\bigwedge_{j=0}^{N-1}(lvtReg(Addr_k) = k \rightarrow mem(k)(j)(Addr_k) = DataIn_k)$$

For the next two implementations' design and Chisel code, see the GitHub repository. Here we give their tagged constraints $\mu(1)$.

For XOR-based one: $\mu_{xor}(1) =$

$$\bigwedge_{k=0}^{M-1}\bigwedge_{j=0}^{M-2}(\oplus_{i=0}^{M-1} memW(i)(j)(Addr_k) = DataIn_k) \wedge \bigwedge_{j=0}^{N-1}(\oplus_{i=0}^{M-1} memR(i)(j)(Addr_k) = DataIn_k)$$

For ILVT-based one: $\mu_{i-lvt}(1) =$

$$\bigwedge_{k=0}^{M-1}\bigwedge_{j=0}^{M-2}((\bigwedge_{i=0}^{M-2}(memsW(k)(j)(Addr_k)[i] \oplus \begin{Bmatrix} i<k & \overline{memsW(i)(j)(Addr_k)[k-1]} \\ else & mems(i)(j)(Addr_k)[k] \end{Bmatrix} = 0) \rightarrow (memsW(k)(j)(Addr_k)) >> (M-1) = DataIn_k)) \wedge \bigwedge_{j=0}^{N-1}((\bigwedge_{i=0}^{M-2}(memsR(k)(j)(Addr_k)[i] \oplus \begin{Bmatrix} i<k & \overline{memsR(i)(j)(Addr_k)[k-1]} \\ else & mems(i)(j)(Addr)[k] \end{Bmatrix} = 0) \rightarrow (memsR(k)(j)(Addr_k)) >> (M-1) = DataIn_k)$$

C. Demonstration of verification scripts

a) SVA verification scripts: According to the formula given in the previous section, we use the SVA syntax to describe the properties that need to be maintained based on the LVT multi-ported memory. Now SymbiYosys still has limitations for symbolic checking. Namely, for symbolic constants like $\mathsf{addr_k}$, SymbiYosys can't directly define them. We have to do a case split on them and verify each case. Here is an SVA implementation of a verification of a multi-ported read and write to a 2x2 multi-ported memory with an assignment of concrete values to symbolic constants $addr_0$ and $addr_1$. For symbolic constants $\mathsf{data_k}$, we don't do case analysis.

```
integer addr_0 = 4'h0...f;
integer data_0 = 8'h1f;
integer addr_1 = 4'h0...f;
integer data_1 = 8'h7;
assume addr_0 != addr_1;

sequence ant0;
  (reset == 0) && (io_wrAddr_0 == addr_0) &&
      (io_wrData_0 == data_0) && (io_wrEna_0
      == 1'b1) && (io_wrAddr_1 == addr_1) && (
      io_wrData_1 == data_1) && (io_wrEna_1 ==
       1'b1);
endsequence
sequence u0;
##1 (lvt.lvtReg[addr_0] == 0) && (Memory.mem[
    addr_0] == data_0) && (Memory_1.mem[addr_0
    ] == data_0) && (lvt.lvtReg[addr_1] == 1)
    && (Memory_2.mem[addr_1] == data_1) && (
    Memory_3.mem[addr_1] == data_1);
endsequence
assume property(@(posedge clock) ant0);
assert property(@(posedge clock) u0);

sequence ant1;
  ((reset == 0) && (lvt.lvtReg[addr_0] == 0)
      && (Memory.mem[addr_0] == data_0) && (
      Memory_1.mem[addr_0] == data_0) && (lvt.
      lvtReg[addr_1] == 1) && (Memory_2.mem[
      addr_1] == data_1) && (Memory_3.mem[
      addr_1] == data_1) && (io_wrEna_0 == 1'
      b0 || (io_wrAddr_0 != addr_0 &&
      io_wrAddr_0 != addr_1)) && (io_wrEna_1
      == 1'b0 || (io_wrAddr_1 != addr_0 &&
      io_wrAddr_1 != addr_1))) ##1 ((reset ==
```


Authorized licensed use limited to: East China Normal University. Downloaded on September 20,2022 at 07:22:33 UTC from IEEE Xplore. Restrictions apply.

```
        0) && (io_wrEna_0 == 1'b0 || (
        io_wrAddr_0 != addr_0 && io_wrAddr_0 !=
        addr_1)) && (io_wrEna_1 == 1'b0 || (
        io_wrAddr_1 != addr_0 && io_wrAddr_1 !=
        addr_1)))[*1:$];
18 endsequence
19 sequence u1;
20   ##1 (lvt.lvtReg[addr_0] == 0) && (Memory.
       mem[addr_0] == data_0) && (Memory_1.mem[
       addr_0] == data_0) && (lvt.lvtReg[addr_1
       ] == 1) && (Memory_2.mem[addr_1] ==
       data_1) && (Memory_3.mem[addr_1] ==
       data_1);
21 endsequence
22 assume property(@(posedge clock) ant1);
23 assert property(@(posedge clock) u1);
24
25 sequence ant2;
26   (reset == 0) && (lvt.lvtReg[addr_0] == 0)
       && (Memory.mem[addr_0] == data_0) && (
       Memory_1.mem[addr_0] == data_0) && (lvt.
       lvtReg[addr_1] == 1) && (Memory_2.mem[
       addr_1] == data_1) && (Memory_3.mem[
       addr_1] == data_1) && (io_rdAddr_0 ==
       addr_0) && (io_rdAddr_1 == addr_1);
27 endsequence
28 sequence cons2;
29 (io_rdData_0 == data_0) && (io_rdData_1 ==
     data_1)
30 endsequence
31 assume property(@(posedge clock) ant2);
32 assert property(@(posedge clock) cons2);
```

```
//M: write port number
//N: read port number
//size: the max address + 1
//w: length of a memory unit/a write data

let steVerify M N size w =
 let lvtw = ⌈log2(M)⌉ in //length of an LVT entry
 let aw=⌈log2(size)⌉ in   // length of an address
 let wraddri = ["io_wrAddr_i<0>",...,"io_wrAddr_i<aw-1>"]
 in
 let wrdatai = ["io_wrData_i<0>",...,"io_wrData_i<w-1>"]
 in
 let memoryi[a] = ["Memory_i.mem<a><0>",...,"Memory_i.mem<a>
 <DLEN-1>"]
 in
 let lvt[a]=["lvt.lvtReg_a<0>",...,"lvt.lvtReg_a<lvtw-1>"]
 in
 let aini=[bvariable "a_i<0>",..., bvariable "a_i<aw-1>"]
 in
 let dini=[bvariable "d_i<0>",..., bvariable "d_i<w-1>"]
 in
 let allAddrsCases = extract M [0...(size-1)] in
 let wrAddrAssert = ⋀_{i=0}^{M-1}(wraddri = aini) in
 let wrDataAssert = ⋀_{i=0}^{M-1}(wrdatai = dini) in
 let memConsAssert i a = memoryi[a] = dini in
 let lvtConsAssert i a = lvt[a] = i in
 let ConsAssert = ⋀_{ads∈allAddrsCases}{⋀_{i=0}^{M-1}(aini = ads[i]) ⇒
 ⋀_{i=0}^{M-1}((memConsAssert i ads[i]) ∧ (lvtConsAssert i ads[i]))}
 in
 let ant_t = wrAddrAssert & wrDataAssert & wrEnaAll & nrst & clk
 in
 let cons_t = Next(ConsAssert) in
 let ant = trajForm2FiveTuples ant_t in
 let cons = trajForm2FiveTuples cons_t in
 let cnstr = ⋁_{ads∈allAddrsCases}(⋀_{i=0}^{M-1}(aini = a)) in
 CSTE "" ckt [] ant cons [] [cnstr] => T | F
;
```

The SVA code corresponds to the previous formal description of the properties: the verification of the correctness of the LVT-based multi-ported memory is divided into three steps. The first step is to write data for each port, that is, assuming $ant(0,1)$ (ie the formula 2) is satisfied, and after a clock asserting $\mu_{lvt}(1)$ (ie the formula IV-B) (line 7-15). And the second step is a holding action, that is, assuming $ant(1,1)$ (ie the formula 3) is holding for any clocks, asserting $\mu_{lvt}(1)$ (ie the formula IV-B) (line 17-24). The thrid step is a reading action, that is assuming $ant(1,2)$ (ie the formula 4), asserting the cons is satisfied (ie the formula 5) (line 26-33).

For all the proof scripts for all verification cases, we use a python program to automatically generate them by case analysis. For the generator in python and the SVA code of the other two designs, see the GitHub repository [10] of MultPortedRAM mentioned above.

b) STE-based verification scripts: STE is an enhanced simulation method with symbolic constants. In the simulation, the values of a circuit node could be symbolic constants such as $\mathsf{addr_k}$ represented by BDD variables. Direct support for symbolic values make STE powerful, thus we need not do a case split on symbolic constants. However, an assertion will cause BDD explosion and trigger Garbage Collection in FORTE if there are too many BDD variables in it. As a glimpse of verification implemented in FORTE, we also give out the verification pseudocode of write action to 2×2 memory below.

As mentioned before, write address $wraddr_i$ and write data $wrdata_i$ of port i are given symbolic constants ain_i and din_i correspondingly. $memory_i[a]$ and $lvt[a]$ stand respectively for a memory unit and an LVT entry. When writing at all ports simultaneously, it's required that different write ports must have a different address. In $allAddrsCases$, we calculate out all the possible evaluation cases each of them assigning each port with a unique address. Here for convenience, we use function $extract$ to generate all the possible assignment cases for all write ports, namely extracting M different addresses from the address set $\{0,\ldots,size-1\}$ and assigning them to each write port in turn. For each case ads aforementioned, there is a corresponding verification that checks if the writing action actually wrote at the target port and address ($memConsAssert\ i\ a$) and updates the LVT with expected value at the appropriate location($lvtConsAssert\ i\ a$). In ant_t, we postulate the preconditions on writing addresses, writing values, writing enable and reset signals, while in $cons\_t$ we define the consequence $Next(ConsAssert)$ which says the conjunction of the previous two assertions $memConsAssert$ and $lvtConsAssert$ holds at next step. In $cnstr$, we specify that during each verification, each assignment a to a different writing address ain_i is different.

D. Bug Finding

When verifying the correctness of the LVT-based multi-ported memory, the design passed the verification of the properties after several modifications. One design defect is caused by the wrong length of the rdAddr/wrAddr port in the LiveValueTable module. For this defect, it is difficult to expose

Authorized licensed use limited to: East China Normal University. Downloaded on September 20,2022 at 07:22:33 UTC from IEEE Xplore. Restrictions apply.

through traditional testing methods, and it needs to make enough corner cases to find out. However, through the formal method based on the SVA describing the property formula, the vulnerability is quickly exposed and repaired in this step.

The $\mu(1)$ in the second step (holding action) can't be held because of this design defect. And the SymbiYosys output gives a counter-example of vcd file, shown in figure 6. As shown in the figure, In the third clock cycle, a writing action occurs at address 31 and address 29, and then the holding property is no longer held. Then we verified it some times with SymbiYosys and found that there is always at least one writing operation that the writing address is far from the initial writing address by an integer power of two. And then the assertion is broken. Then we guessed that the length of some ports was set incorrectly, and unsurprisingly, we got the defect.

V. Experiments

We implement the corresponding SVA code according to the verified formula given above, use SymbiYosys tool to convert it with the SystemVerilog code generated from the Chisel module, into an SMT solution file, use yosys-smtbmc, and use the z3 solver to perform model checking. The cost is shown in Table I.

- Software environment: CentOS Linux release 8.2
- Hardware environment: Intel Xeon Gold 6254, 754.3 GiB.

TABLE I: Usage of Time and Memory in SymbiYosys Verification

Type	Data Length(Bit)	M Write/ N Read ports(M * N)	Time (sec.)	Memory (MiB)
LVT	16	2 * 2	8.7294	63.48
		3 * 3	13.6782	119.43
	32	2 * 2	29.3718	123.59
		3 * 3	38.9039	241.43
XOR	16	2 * 2	50.1115	76.23
		3 * 3	68.8651	161.29
	32	2 * 2	75.1162	144.65
		3 * 3	118.3078	354.26
I-LVT	16	2 * 2	45.7550	82.37
		3 * 3	60.31	165.79
	32	2 * 2	72.8372	148.43
		3 * 3	110.4382	349.81

As shown in the table above, as the complexity of the design and the length of the memory unit increase, the time and memory of verification increase rapidly, even though the designs are actually generated from the same Chisel module with different parameters. Therefore, the next step is to consider how to perform high-level verification for the implemented Chisel module.

As for STE verification work, we need the aid of a STE engine, FORTE, provided by INTEL. Our trajectory evaluation theory and verification are defined in FL, a.k.a. function language. We run STE model chcking in a workstation with an Intel Xeon processor, 8 GB memory and 64-bit Fedora release 20. Experiment results are summerized in Table II.

It is precise because there are several optimizing mechanisms such as automatic garbage collection and re-ordering

TABLE II: Time and Memory Usage of STE Verification

Type	Data Length(Bit)	M Write/ N Read ports(M * N)	Time (sec.)	Memory (MiB)
LVT	16	2 * 2	0.92	28.24
XOR	16	2 * 2	1082.24	142.95
I-LVT	16	2 * 2	90.84	120.44

of BDD variables that it is quite time-consuming for STE to finish the verification of XOR and I-LVT which contains a large number of BDD variables for representing the symbolic constants like read-in data and memory units. But beyond that, STE model checking can achieve much more efficiency than SymbiYosys while accomplishing the same tasks in view of the comparison experiment on LVT. In the future, we will consider how to integrate symmetry reduction into our STE model checking and adopt more high-level symbolic indexing [15] to reduce the complexity of one assertion.

VI. Related work

To increase throughput in accessing memory, a straightforward approach is to provide multi-ported memory [16][17]. With the increase of ports, delays can increase accordingly, and area consumption can be even unacceptable. A number of studies focus on the schemes of multi-ported memory. Conventionally, the multi-ported accesses can be achieved through replication, multi-banking, multi-pumping [18], live value table (LVT) [19], and coding-based approaches [1]. In this paper. we focus on the verification of the LVT-based, coding-based, and hybrid implementations.

With the growth of hardware design and development using high-level abstract languages for hardware design and implementation, hardware verification work built on Chisel hardware construction language has been widely proposed in academia. Dobis and Petersen's group proposed ChiselVerify [20], which is the beginning of a verification library built on Chisel for digital hardware. And the work in [21] purposed a feasible verification flow in chisel-based deep learning accelerator design. However, most of these works are not formal, our methodology fills this gap.

Symbolic Trajectory Evaluation has also a comparatively broad application in memory verification. The work in [22] conducts STE model checking on memory arrays at switch-level, meanwhile proposing the approach of exploiting symmetry to overcome the bottleneck of STE on large complex memory. And at the same period, M.Pandey et al. [23] integrated a new Boolean encoding technique into STE, successfully verify a variety of CAMs, such as TLBs, cache tags, and branch target buffers, which avoids the OBDD space explosion problem. The work in [24] exploits the symmetry between netlist and trajectory assertions to verify the typical actions of the CAMs such as associative-date reading operation. Li [13] extends the Generalized Symbolic Trajectory Evaluation into term-level and develops a parametrized verification framework, which is suitably applied in verifying classic hardware models such as FIFO and shift memory.

Authorized licensed use limited to: East China Normal University. Downloaded on September 20,2022 at 07:22:33 UTC from IEEE Xplore. Restrictions apply.

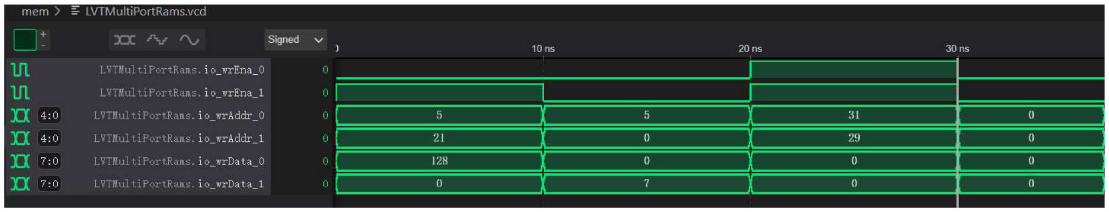

Fig. 6: the VCD File of Counter Example given by SymbiYosys

VII. Conclusion and Future Work

In this work, we use Chisel to design three kinds of multi-ported memories and use formal approaches to verify them. We focus on the design's modularity, brief, and correctness. Choosing Chisel can guarantee the two former ones while choosing SVA and STE can guarantee the last feature. We only need an undergraduate student to design the memories and use a master student to verify them. Besides, we can get more insights into the correctness of parameterized multi-ported memory designs through SVA/STE assertions.

At present, there is some limitation. We design the memories by Chisel but verify them by SVA at Verilog level and by STE at netlist level. Therefore, when we write SVA and STE assertions, we need to know the mapping from a high level to a low level; and when we find bugs at a low level, we need to restore the counterexample from low level to high level. Besides, we need to trust the compiler from Chisel to Verilog and that from Verilog to Exlif.

In the future, we look forward to integrating the designs and verification techniques into multi-core CPUs, and therefore, we can't verify the related design directly on the chisel language. We hope to use the assertion-based at the level of the Chisel language to directly verify multi-ported memory designs, which will automatically compile into SVA or formulas which can be directly checked by some formal tools.

Acknowledgments

Mufan Xiang and Yongxin Zhao is supported by Shanghai Science and Technology Commission Program under Grant 20511106002, Shanghai Trusted Industry Internet Software Collaborative Innovation Center; Yongjian Li et al. is supported by Chinese Academy of Sciences Strategic Leading Science and Technology Project (Class A): RISC-V Agile Design, and Cross-Layer Optimization.

References

[1] C. E. LaForest, M. G. Liu, E. R. Rapati, and J. G. Steffan, "Multi-ported memories for fpgas via xor," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, 2012, pp. 209–218.

[2] A. M. Abdelhadi and G. G. Lemieux, "Modular multi-ported sram-based memories," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014, pp. 35–44.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 1212–1221.

[4] Y. Tao, "An introduction to assertion-based verification," in *2009 IEEE 8th International Conference on ASIC*. IEEE, 2009, pp. 1318–1323.

[5] S. Vijayaraghavan and M. Ramanathan, *A practical guide for SystemVerilog assertions*. Springer Science & Business Media, 2006.

[6] Wolf. Symbiyosys (sby) – front-end for yosys-based formal verification flows. [Online]. Available: https://github.com/YosysHQ/SymbiYosys

[7] C. Barrett, A. Stump, C. Tinelli *et al.*, "The smt-lib standard: Version 2.0," in *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, vol. 13, 2010, p. 14.

[8] C.-J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, 1995.

[9] J. Yang and A. Goel, "Gste through a case study," in *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, 2002, pp. 534–541.

[10] https://github.com/VerificaticationStudio/MultPortedRAM.

[11] J. Bruant, P.-H. Horrein, O. Muller, T. Groleat, and F. Pétrot, "Towards agile hardware designs with chisel: a network use-case," *IEEE Design & Test*, 2021.

[12] https://github.com/AmeerAbdelhadi/Multiported-RAM.

[13] Y. Li and B.-y. Wang, "Parameterized hardware verification through a term-level generalized symbolic trajectory evaluation," in *International Conference on Formal Engineering Methods*. Springer, 2019, pp. 403–419.

[14] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme, "An industrially effective environment for formal hardware verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1381–1405, September 2005.

[15] T. Melham and R. Jones, "Abstraction by symbolic indexing transformations," 11 2002, pp. 1–18.

[16] F. Anjam, M. Nadeem, and S. Wong, "A vliw softcore processor with dynamically adjustable issue-slots," in *2010 International Conference on Field-Programmable Technology*. IEEE, 2010, pp. 393–398.

[17] H. J. Mattausch, Y. Tatsumi, K. Kishi, T. Gyoten, and K. Yamada, "Area-efficient multiport memories for the tb/s bandwidth era," in *Proceedings of the 25th European Solid-State Circuits Conference*. IEEE, 1999, pp. 126–129.

[18] H. E. Yantir, S. Bayar, and A. Yurdakul, "Efficient implementations of multi-pumped multi-port register files in fpgas," in *2013 Euromicro Conference on Digital System Design*. IEEE, 2013, pp. 185–192.

[19] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for fpgas," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, 2010, pp. 41–50.

[20] A. Dobis, T. Petersen, K. J. H. Rasmussen, E. Tolotto, H. J. Damsgaard, S. T. Andersen, R. Lin, and M. Schoeberl, "Open-source verification with chisel and scala," *arXiv preprint arXiv:2102.13460*, 2021.

[21] Z. Li, Y. Chen, and D. Zhao, "A method of verification in chisel based deep learning accelerator design," in *2020 IEEE International Conference on Information Technology,Big Data and Artificial Intelligence (ICIBA)*, vol. 1, 2020, pp. 789–792.

[22] M. Pandey and R. Bryant, "Formal verification of memory arrays using symbolic trajectory evaluation," 09 1997, pp. 42 – 49.

[23] M. Pandey, R. Raimi, R. E. Bryant, and M. S. Abadir, "Formal verification of content addressable memories using symbolic trajectory evaluation," 1997.

[24] Y. Li, W. Hung, X. Song, and N. Zeng, "Exploring structural symmetry automatically in symbolic trajectory evaluation," *Formal Methods in System Design*, pp. 1–27, 2011, 10.1007/s10703-011-0119-z. [Online]. Available: http://dx.doi.org/10.1007/s10703-011-0119-z

Authorized licensed use limited to: East China Normal University. Downloaded on September 20,2022 at 07:22:33 UTC from IEEE Xplore. Restrictions apply.