An Abstraction of Multi-Port Memories with Arbitrary Addressable Units*

Lukáš Charvát, Aleš Smrčka, and Tomáš Vojnar

Brno University of Technology, FIT, IT4Innovations Centre of Excellence
Božetěchova 2, 612 66 Brno, Czech Republic
{icharvat,smrcka,vojnar}@fit.vutbr.cz

Abstract. The paper describes a technique for automatic generation of abstract models of memories that can be used for efficient formal verification of hardware designs. Our approach is able to handle addressing of different sizes of data, such as quad words, double words, words, or bytes, at the same time. The technique is also applicable for memories with multiple read and write ports, memories with read and write operations with zero- or single-clock delay, and it allows the memory to start with a random initial state allowing one to formally verify the given design for all initial contents of the memory. Our abstraction allows large register-files and memories to be represented in a way that dramatically reduces the state space to be explored during formal verification of microprocessor designs as witnessed by our experiments.

1 Introduction

As the complexity of hardware is growing over the last decades, automation of its development is crucial. This also includes automation of the process of verification of the designed systems. Verification of current microprocessor designs is typically performed by simulation, functional verification, and/or formal verification (often using various forms of model checking or theorem proving). The complexity of the verification process is usually significantly influenced by the presence and size of the memories used in the design because of an exponential increase in the size of the state space of the given system with each additional memory bit. Therefore the so-called *efficient memory modeling* (EMM) techniques that try to avoid explicit modeling of the memories are being developed.

In this work, we present an approach to automatic generation of abstract memory models whose basic idea comes from the fact that formal verification often suffices with exploring a limited number of accesses to the available memory, and it is thus possible to reduce the number of values that are to be recorded to those that are actually stored in the memory (abstracting away the random contents stored at unused memory locations). Around this basic idea, we then build an approach that allows one to represent memories with various advanced features, such as different kinds of endianness (big or little),

* The work was supported by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education (project MSM 0021630528), the Czech Ministry of Industry and Trade (project FR-TI1/038), the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, and the internal BUT projects (FIT-S-11-1 and FIT-S-12-1).

read and write delays, multiple read and write ports, and different sizes of addressable units (e.g., bytes, words, double words). As far as we know, the ability to handle all of the above mentioned features differentiates our approach from the currently used ones. Moreover, our technique is applicable in environments requiring a very high level of automation (e.g., processor development frameworks), and it is suitable for formal verification approaches that aim at verifying a given design for an arbitrary initial contents of the memory. Moreover, our abstract memory models can be used within formal verification in a quite efficient way as proved by our experiments.

2 Related Work

Numerous works have focused on memory abstraction, notably within the area of formal verification. Some of the proposed abstractions are tightly coupled with the verification procedure used: for instance, many of them rely on that SAT-based bounded model checking (BMC) [1] or BDD-based model checking [2] are used.

More general approaches, i.e., approaches not tailored for a specific verification procedure, often exploit theories for reasoning about safety properties of systems with arrays, such as [3, 4] and especially the work on an extensional theory of arrays [5]. Intuitively, this theory formalizes the idea that two arrays are equivalent if they have the same value at each index. An example of such an approach has been presented in [7]. In the work, an automatic algorithm for constructing abstractions of memories is presented. The algorithm computes the smallest sound and complete abstraction of the given memory.

In [6], the authors introduce a theory of arrays with quantifiers which is an extension of [5]. Moreover, they define the so-called *array property fragment* for which the authors supplement a decision procedure for satisfiability. A modification of the decision procedure for purposes of correspondence checking is proposed in [8] and implemented in [9].

Another method of large memory modeling is described in [10]. The memory state is represented by an ordered set containing triples composed of (i) an expression denoting the set of conditions for which the triple is defined, (ii) an address expression denoting a memory location, and (iii) a data expression denoting the contents of this location. For this set, a special implementation of write and read operations is provided. The abstracted memory interacts with the rest of the circuit using standard *enable*, *address*, and *data* signals. The size of the set is proportional to the number of memory accesses. Further, in [11], the same author extends the approach such that it can be used for correspondence checking by applying the so-called shadowing technique for read operations (we will get back to this issue in Section 3.4).

A recently published work [12] formally specifies and verifies a model of a large memory that supports efficient simulation. The model is tailored for Intel x86 implementations only in order to offer a good trade-off between the speed of simulation and the needed computational resources.

A common disadvantage of [7, 8, 10, 11] is the fact that they omit a support for addressing of different sizes of data which is considered, e.g., in [12]. On the other hand, in [12], the authors assume starting from the nullified state of the memory, not from a random state.

Fig. 1. Memory interface

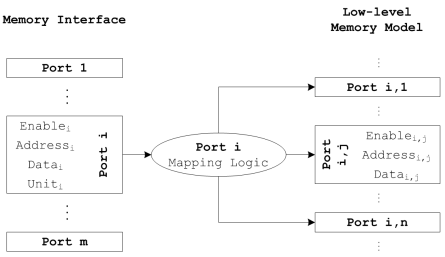

Fig. 2. Memory mapping

Some of the other proposed works describe a smarter encoding of formulas, including a description of memories, into CNF [13, 14]. In this work, the problems linked to the CNF transformation are not discussed, however, the ideas in [13, 14] can be potentially applied here. An example of a tool based on the method coupled with CNF is the *Bit Analysis Tool* [13] (BAT) which automatically builds abstraction of memories to be used in BMC of a certain depth. As its input, BAT uses a custom LISP-based language. A model of the verified system using abstracted memories is created in the following steps: (1) The design to be verified is simplified through pre-defined rewrite rules applied on the level of terms of the BAT language. (2) An *equality test relation* that relates memories that are directly compared for equality is built over the set of memory variables. (3) The transitive closure of the test relation is computed. The closure is an equivalence relation. (4) An *address set* is computed for each of the equivalence classes. The address set contains only the addresses that are relevant for the given class. (5) For all addresses in an address set, a shorter bit vector for addressing the abstract memories is created. The size of the vector is proportional to the number of memory accesses. (6) The behavior of memories is changed to be compatible with the new addressing style. (7) Original memories and addresses are replaced with their abstract counterparts. A description of a system together with the checked properties is then efficiently transformed to a CNF formula. Similarly to previous approaches, there is no support for addressing different sizes of data.

To sum up, our approach can generate abstractions of memories that support addressing of arbitrary addressable units, such as bytes and words (unlike [7, 8, 10, 11]), with multiple read and write ports (in contrast with [7, 8]), and it allows the memory to start from a random initial state (not available in [12]). The algorithm is not bound to any specific verification technique (unlike [13, 14]).

3 Large Memory Abstraction

We are now going to describe our technique of automated memory abstraction. As we have already said, its basic idea is to record only those values in the memory that are actually used (abstracting away the random contents stored at unused memory locations).

3.1 Memories To Be Abstracted

In our approach, we view a memory as an item of the verified design with the interface depicted in Fig. 1. The interface consists of (possibly multiple) read and write ports.

Each port is equipped with `Enable`, `Address`, `Data`, and `Unit` signals. When the `Enable` signal is down, the value of the `Data` signal of a read port is undefined. When dealing with a write port, no value is stored into the memory through this port. On the other hand, when the `Enable` signal is up, the memory returns/stores data from/into the cell associated with the value of the `Address` signal. In the special case when multiple ports are enabled for writing into the same memory cell, the result depends on the implementation of the memory. We support two variants: (i) either a prioritized port is selected or (ii) an undefined (random) value is stored to the multiply addressed memory cell.

The size of the addressed unit can be modified by the `Unit` signal. When the size of the accessed unit is smaller than the size of the greatest addressable unit, the most significant bits of the `Data` signal are filled up with zeros. It is also assumed that the size of any addressable unit is divisible by the size of the least addressable unit, and thus for the `Data` signal it is sufficient to transfer the size of the addressed unit expressed as a multiple of the least addressable unit only (instead of the actual number of bits of the unit). Finally, if the memory allows addressing of a single kind of units only, then the `Unit` signal can be omitted.

3.2 Abstraction of the Considered Memories

Our abstraction preserves the memory interface, and hence concrete memories can be easily substituted with their abstract counterparts. We will first describe the basic principle of our abstraction on memories with a single addressable unit only. An extension of the approach for multiple addressable units will be discussed later. Moreover, we assume reading with no delay and writing with a delay of one cycle. An extension to other timings will be described in Section 3.4.

The abstract memory effectively remembers only the memory cells which have been accessed. Internally, the memory is implemented as a table consisting of some number d of couples of variables storing corresponding pairs of addresses and values (a, v). When using bounded model checking (BMC) as the verification technique, the needed number d of address-value pairs can be easily determined from the depth k of BMC as the following holds $d = k * (m + n)$ where m and n denote the number of read and write ports, respectively. For unbounded verification, the number d can be iteratively incremented until it is sufficient. The incrementation is finite since the number of memory cells is finite. The memory also remembers which of the pairs are in use by tracking the number $r \in \{0, ..., d\}$ of couples that were accessed (and hence the number of the rows of the table used so far).

When the memory is accessed for reading, the remembered address-value pairs $(a_1, v_1), ..., (a_r, v_r)$ that are in use are searched first. If a location a_{rd} that is read has been accessed earlier, then the value v_i associated with the appropriate address $a_i = a_{rd}$ is simply returned. On the other hand, if a location that has never been accessed is read, a corresponding pair is not found in the table, and a new couple (a_{rd}, v_{rd}) is allocated. Its address part a_{rd} will store the particular address that is accessed while the value v_{rd} is initialized as unconstrained. However, the variable representing the value v_{rd} associated with the accessed location a_{rd} is kept constant in the future (unless there occurs a write operation to the a_{rd} address). This ensures that subsequent reads from a_{rd} return

the same value. In case of writing, the address a_{wr} and value v_{wr} are both known. When writing to a location that has not been accessed yet, a new address-value pair (a_{wr}, v_{wr}) is allocated to store the given address-value pair. Otherwise, a value v_i associated with the given address $a_{wr} = a_i$ is replaced by v_{wr}.

3.3 Dealing with Differently Sized Data

In order to support dealing with different sizes of addressable data (including reading/writing data smaller than the contents of a single memory cell of the modeled memory), we split our abstract memory into a low-level memory model and a set of functions mapping accesses to ports of the modeled memory to ports of the low-level memory. The idea of this approach is shown in Fig. 2 and further discussed below.

The low-level memory consists of cells whose size equals the size of the least addressable unit of the modeled memory, and therefore, for low-level memory, the `Unit` signal can be omitted. In the low-level memory, values of units that are larger than the least addressable unit are stored on succeeding addresses. In order to allow for reading/writing the allowed addressable units (including the greatest one) in one cycle, the number of read and write ports of the low-level memory is appropriately increased. The resulting number of ports of the low-level memory is equal to $m * n$ where m is the number of interface ports and n is the number of distinct addressable units. The latter can be expressed as the quotient of bit-widths of the greatest (w_{gau}) and the least (w_{lau}) addressable unit. In other words, for each port of the memory interface there are n corresponding ports of the low-level memory model. Therefore, we use double indices for the low-level memory ports in our further description.

In particular, let $enable_i$, $data_i$, $address_i$, and $unit_i$ be values of signals of the port i of the memory interface, and let $enable_{i,j}$, $data_{i,j}$, and $address_{i,j}$ have the analogical meaning for the low-level memory port i, j. Then, the value of the $enable_{i,j} \in \mathbb{B}$ signal can be computed as $enable_i \wedge unit_i \geq j$ where $enable_i \in \mathbb{B}$ and $1 \leq unit_i \leq n$. This means that the required number of low-level memory ports are activated only. Next, the value of $address_{i,j}$ can be expressed as $address_i + j - 1$ for the little endian version of the memory and $address_i + unit_i - j$ for the big endian version, respectively. These expressions follow from the fact that larger units of the original memory are stored as multiple smallest addressable units stored at succeeding addresses in the low-level memory.

Further, for transfers of data, separate mappings for read ports and write ports must defined. In the case of a write port, the data flow into the low-level memory, and the value of the $data_{i,j}$ signal can be computed as $slice(data_i, unit_i * w_{lau} - 1, (unit_i - 1) * w_{lau})$ where $slice$ is a function extracting the part of the first argument (on the bit level) that lies within the range given by the second and third arguments (with the bit indices being zero-based). Finally, for a read port, for which data flow from the low-level memory, the value of the $data_i$ signal can be expressed as $concat(ite(enable_{i,n} \vee \neg enable_{i,1}, data_{i,n}, 0), ..., ite(enable_{i,2} \vee \neg enable_{i,1}, data_{i,2}, 0), data_{i,1})$ where $concat$ is a bit concatenation and ite ("if-then-else") is the selection operator. Thus, the data value is composed from several ports of the low-level memory, and the most significant bits are zero-filled when the read unit is smaller than the greatest one. Note that according to the semantics of the `Enable` and `Data` signals (described in Section 3.1), in

the case when $enable_{i,1}$ is false (i.e., no unit is read), the value of the $data_i$ signal is undefined.

3.4 Further Extensions of the Abstract Memory Model

To broaden the range of memories that we can abstract, we further added a support for more memory timing options, in particular for one-cycle-delay reading and zero-delay writing. The former can be achieved by simply connecting a unit buffer to the data signal of the memory interface. For the latter case, a special attention must be paid to the situation when both read and write operations over the same address are zero-delayed. In such a situation, it is required to append an additional logic that ensures that written data are propagated with zero delay to a given read port.

Moreover, for a practical deployment in correspondence checking, our model has also been extended by applying the shadowing technique described in [11]. In particular, during correspondence checking, both models are executed in a sequence. The shadowing technique deals with potential inconsistencies that can arise when both models read from the same uninitialized memory cell—indeed, in this case, a random value is to be returned, but the same one in both models. To ensure this, when shadowing is used, the return value of the read operation is obtained from the memory in the design executed first whenever the value is not available in the second design.

4 Implementation and Experiments

The memory abstraction that we generate in the above described way can be encoded in any language for which the user can provide templates specifying (i) how to express declarations of state and nonstate variables, (ii) how to encode propositional logic expressions over state and nonstate variables, (iii) and how to define initial and next states of state variables. We currently created these templates for the Cadence SMV language [15].

In order to prove usefulness of the described abstraction technique, we used our abstract memory generator within the approach proposed in [16] for checking correspondence between the ISA and RTL level descriptions of microprocessors, which we applied to several embedded microprocessors. Briefly, in the approach of [16], the ISA specification and VHDL model of a processor are automatically translated into behavioral models described in the language of a model checker (the Cadence SMV language in our case). These models are then equipped with an environment model, including architectural registers and memories, which can be abstracted using the technique proposed in this article. All these models are composed together, and BMC is used to check whether if both of the processor models start with the same state of their environment (including the same instruction to be executed), their environments equal after the execution too. The described approach was integrated into the Codasip IDE [17] processor development framework.

Our approach was tested on three processors: *TinyCPU* is a small 8-bit test processor with 4 general-purpose registers and 3 instructions that we developed mainly for testing new verification approaches. *SPP8* is an 8-bit ipcore with 16 general-purpose

Table 1. Verification results

Processor	Reg. File Size	Memory Size	Explicit Memory	Abs. Reg. File	Abs. Memory	All Abs.
TinyCPU	4 x 8bit	-	0.151 s	0.41 s	-	-
SPP8	16 x 8bit	256 x 8bit	5.06 s	1.11 s	3.66 s	0.452 s
SPP16	16 x 16bit	2048 x 8bit	266 s	92.2 s	1.23 s	0.822 s
Codea2_single	32 x 16bit	32768 x 16bit	o.o.m.	o.o.m.	4.30 s	4.44 s
Codea2_mult	32 x 16bit	65536 x 8bit	o.o.m.	o.o.m.	4.75 s	4.89 s

registers and a RISC instruction set consisting of 9 instructions. *SPP16* is a 16-bit variant of the previous processor with a more complex memory model allowing one, e.g., to load/store both bytes and words from/to the memory. *Codea2* is a 16-bit processor with 4 pipeline stages partially based on the MSP430 microcontroller developed by Texas Instruments [18]. The processor is dedicated for signal processing applications. It is equipped with 16 general-purpose registers, 15 special registers, a flag register, and an instruction set including 41 instructions, where each may use up to 4 available addressing modes. Our experiments were evaluated for two modifications of the processor—using memory with and without multiple addressable units.

Our experiments were run on a PC with Intel Core i7-3770K @3.50GHz and 32 GB RAM using Cadence SMV (the build from 05-25-11) and GlueMinisat (version 2.2.5) [19] as an external SAT solver. The results can be seen in Table 1. The first three columns give the processor being verified, the size of its register file, and the size of the memory. The next columns give the results obtained from the verification—in particular, the average time needed for verification of a single instruction with the abstraction applied or not-applied in different combinations on the register file and the memory. In the first case, both the register file and the memory were modeled explicitly which, for larger designs such as Codea2, led to out-of-memory errors ("o.o.m."). Next, the abstraction was only used for register files. Even though better results were obtained this way for the SPP8 and SPP16 processor designs, the verification still ran out of system resources for Codea2 because of the explicitly modeled memory. In the last two cases when either only memories or both memories and register files of the verified processors were abstracted, verification was able to finish even for larger designs. We explain the 10 % deterioration between verification times for the Codea2 processor with and without presence of multiple addressable units by the complexity of the additional logic.

Finally, we note that for very small memories and memories with many possible accesses (caused by, e.g., a higher verification depth during BMC), the overhead brought by the abstraction can result in worse verification times as can be seen in the case of the register file of the TinyCPU and Codea2 processors. Moreover, for SPP8, where only a few instructions directly access the memory, and thus only a few instructions influence the average verification times, the overhead caused by the abstraction introduces worse than expected average verification time when abstracting the memory only. In practice,

we deal with this problem by defining a heuristics that computes whether or not it is better to use an explicit or abstract description of a given memory.

5 Conclusion

We have presented an approach of memory abstraction that exploits the fact that formal verification often suffices with exploring a limited number of accesses to the available memory, and it is thus possible to reduce the number of values that are to be recorded to those that are actually stored in the memory. Our approach allows one to abstract memories with various advanced features, such as different kinds of endianness, read and write delays, multiple read and write ports, and different sizes of addressable units. The techniques is fully automated and suitable for usage within processor development frameworks where it can bring a significant improvement in verification times.

References

1. A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Advances in Computers*, Vol. 58, 2003.
2. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, Vol. 98, No. 2, 1992.
3. J. McCarthy. Towards a mathematical science of computation. *In IFIP Congress*, 1962.
4. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, Vol. 1, No. 2, 1979.
5. A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *Proc. of Logic in Computer Science*, IEEE Computer Society, 2001.
6. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays?. In *Proc. of VMCAI*, LNCS 3855, 2006.
7. S. M. German. A Theory of Abstraction for Arrays. In *Proc. of FMCAD*, Austin, TX, 2011.
8. A. Koelbl, J. Burch, and C. Pixley. Memory Modeling in ESL-RTL Equivalence Checking. In *Proc. of DAC*, IEEE Computer Society, 2007.
9. A. Koelbl, R. Jacoby, H. Jain, and C. Pixley. Solver Technology for System-level to RTL Equivalence Checking. In *Proc. of DATE*, IEEE Computer Society, 2009.
10. M. N. Velev, R. E. Bryant, and A. Jain. Efficient Modeling of Memory Arrays in Symbolic Simulation. In *Proc. of CAV*, LNCS 1254, 1997.
11. R. E. Bryant and M. N. Velev. Verification of Pipelined Microprocessors by Comparing Memory Execution Sequences in Symbolic Execution. In *Proc. of ASIAN*, LNCS 1345, 1997.
12. W. A. Hunt, Jr. and M. Kaufmann. A Formal Model of a Large Memory that Supports Efficient Execution. In *Proc. of FMCAD*, IEEE Computer Society, 2012.
13. P. Manolios, S. K. Srinivasan, and D. Vroon. Automatic Memory Reductions for RTL Model Verification. In *Proc. of ICCAD*, ACM/IEEE Computer Society, 2006.
14. M. K. Ganai, A. Gupta, and P. Ashar. Verification of Embedded Memory Systems using Efficient Memory Modeling. In *Proc. of DATE*, ACM/IEEE Computer Society, 2005.
15. K. L. McMillan, Cadence SMV, `www.kenmcmil.com/smv.html`.
16. A. Smrčka, T. Vojnar, and L. Charvát. Automatic Formal Correspondence Checking of ISA and RTL Microprocessor Description. In *Proc. of MTV*, 2012.
17. Codasip Studio for Rapid Processor Development, `www.codasip.com`.
18. Codea2 Core IP in Codasip Studio, `www.codasip.com/products/codea2/`.
19. H. Naneshima, K. Iwanuma and K. Inoue. GlueMinisat, appeared in *SAT Competition 2011*, `sites.google.com/a/nabelab.org/glueminisat/`, 2011.