Universal Memory Automaton and Automated Verilog HDL Code Generation for a Cache Coherency Snooping Protocol

Matthias W. Fertig

***Abstract*—This paper introduces the concept of Universal Memory Automata (UMA) and automated compilation of Verilog Hardware Description Language (HDL) code at Register Transfer Level (RTL) from UMA graphs for digital designs. The idea is based on the observation that Push Down Automata (PDA) are able to process the Dyk-Language - commonly known as the balanced bracket problem - with a finite set of states while Finite State Machines (FSM) require an infinite set of states. Since infinite sets of states are not applicable to real designs, PDAs appear promising for types of problems similar to the Dyk-Language. PDAs suffer from the problem that complex memory operations need to be emulated by a specific stack management. The presented UMA therefore extends the PDA by other types of memory, e.g. Queue, RAM or CAM. Memories that are eligible for UMAs are supposed to have at least one read and one write port and a one-cycle read/write latency. With their modified state-transfer- and output-function, UMAs are able to operate user-defined numbers, configurations and types of memories. Proof of concept is given by an implementation of a cache coherency protocol, i.e. a practical problem in microprocessor design.**

***Index Terms*—Finite state machines, sequential circuits, sequential synthesis, high-level and register-transfer level synthesis, methodologies for EDA, automata extensions, processors and memory architectures, push down automata, HDL compilation, digital design automation.**

I. Introduction

A. Finite State Machines

In Digital Engineering, Finite State Machines (FSMs) [1]–[3, 7, 8, 12] are a standard design element to process regular languages. They are typically utilized to implement control logic. FSMs are given by a set

$$FSM = (S, S_0, F, \Sigma, \Gamma, \delta, \omega) \tag{1}$$

where S is a finite set of states, S_0 is the initial state, $F \subseteq S$ is a finite set of final states, Σ is the input alphabet, Γ is the output alphabet, δ is the state transfer function and ω is the output function. State transfers are defined by the *state transfer function*

$$\delta : \begin{cases} S \times \Sigma & \rightarrow & S \\ s' & = & \delta(s, \sigma) \end{cases} \tag{2}$$

Matthias W. Fertig, matthias.fertig@htwg-konstanz.de, University of Applied Sciences HTWG Konstanz, Alfred Wachtel Straße 8, 78462 Konstanz.

where a destination state $s' \in S$ is reached from a source state $s \in S$ by processing an input $\sigma \in \Sigma$. Outputs are defined by the *output function*

$$\omega : \begin{cases} S \times \Sigma & \rightarrow & \Gamma \\ \gamma & = & \omega(s, \sigma) \end{cases} \tag{3}$$

where $\gamma \in \Gamma$ is an output symbol derived from a state $s \in S$ and input $\sigma \in \Sigma$. This architecture is of type *Mealy* since the output function depends on inputs and states. *Moore* and *Simple Moore* architectures are deduced from simplified output functions. FSMs are called *finite* because they are built from a finite set of states, implemented by a state memory. *State memory* is of size $log_2(N_S)$ and required in all known FSM architectures, where N_S is the number of states in the finite set of states.

B. Push Down Automata

Push Down Automata (PDA) [4]–[6, 9]–[11, 13], also called Stack Automata, are given by a set

$$PDA = (S, S_0, F, \Sigma, \Gamma, \delta, \omega, \mathbf{A}, \mathcal{X}) \tag{4}$$

where common elements equal those of the FSM. $\mathcal{X}$ is the *stack alphabet* (I-B1) to operate the *stack memory* (II-A2) $\mathbf{A}$.

$$\mathcal{X} = \{PUSH(), POP(), TOP(), NOP()\} \tag{5}$$

The state transfer function of a PDA δ is

$$\delta : \begin{cases} S \times \Sigma \times \mathcal{X} & \rightarrow & S \times \mathcal{X} \\ (s', x) & = & \delta(s, \sigma, x) \end{cases} \tag{6}$$

where $x \in \mathcal{X}$ are stack operations and $w \in \Sigma^*$ is a word on the input alphabet. The output function of the PDA is

$$\omega : \begin{cases} S \times \Sigma \times \mathcal{X} & \rightarrow & \Gamma \times \mathcal{X} \\ (\gamma, x) & = & \omega(s, \sigma, x) \end{cases} \tag{7}$$

1) Stack alphabet: A set of operations on the stack $\mathbf{A}$. PUSH($\mathbf{A}$,w) puts an element $w \in \Sigma^*$ on the stack, POP($\mathbf{A}$) returns the first element from the stack and deletes the first element, TOP($\mathbf{A}$) returns the first element and keeps the first element, NOP($\mathbf{A}$) is a no-operation on the stack, sometimes called the empty operation. Stack operations are utilized in state-transfer- and output-functions to conditionally read and write the stack.

II. The Universal Memory Automaton

Universal Memory Automata (UMA) are an innovative concept for operating multiple (parallel) memories in a finite state graph. While FSMs utilize state memory and PDAs utilize state and stack memory only, UMAs extend the state memory by multiple (k) memories of selectable and configurable type. Eligible types of memories in this paper are last-in first-out (Stack) $\mathbf{A}$ like in PDAs, first-in first-out memory (Queue) $\mathbf{Q}$, random-access (RAM) $\mathbf{R}$ and content-addressable (CAM) memory $\mathbf{C}$. UMAs allow an arbitrary number and a user-defined configuration of those memories. PDAs are of course able to emulate all these types of memory by a specific stack management, but the additional effort makes PDAs more of a theoretic model of computation than an applicable tool for real designs. If several and potentially different types of memories are desired, the effort to model those with the single stack of a PDA becomes even higher. To resolve this issue, this paper introduces the idea of operating multiple memories of variable type and configuration, which is particularly relevant for real applications. This is performed by the UMA.

Universal Memory Automata (UMA) are given by a set

$$UMA = (S, S_0, F, \Sigma, \Gamma, \delta, \omega, \mathbf{X}^k, \mathcal{X}^k) \qquad (8)$$

where common elements equal those of the PDA and $\mathbf{X}^k$ is an k-dimensional set of memories of selectable type, i.e. $\mathbf{X} \in \{\mathbf{A}, \mathbf{Q}, \mathbf{R}, \mathbf{C}\}$, operated by a k-dimensional memory alphabet $\mathcal{X}^k$. UMAs thereby control k memory instances $\mathbf{X}^k$, $k \in \mathcal{N}_0^+$, each of different type if desired.

Memory operations PUSH(), POP(), TOP() and NOP() are contained in each of the k-dimensional memory alphabet, $\mathcal{X}^k$. Dimensions of memory and memory alphabet are connected to each other so that the i-th memory alphabet $\mathcal{X}_i$ operates on the i-th memory instance $\mathbf{X}_i$, where $1 \leq i \leq k$. Operations are shared from PDA-theory and implemented by virtual functions in UMA-theory to perform according to the principle of operation of the adjacent memory. For memories of type $\mathbf{R}$ and $\mathbf{C}$ an address is required while for memories of type $\mathbf{Q}$ and $\mathbf{A}$ accesses is self-organized by the memory using an internal address pointer. The address width of the RAM or CAM is $\log_2(N)$, where N is the number of addressable entries

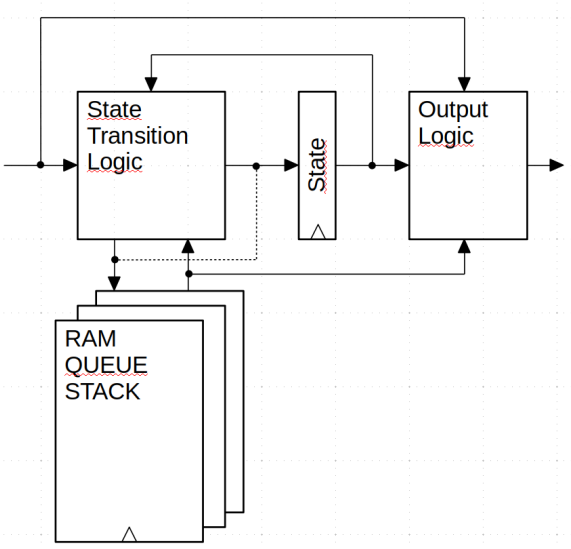


Figure 1. Universal Memory Automata (UMA) architecture blockdiagram.

of the memory. Such type of UMA is of type *Mealy* architecture. *Moore* and *Simple-Moore* architectures are gained from a simplification of the output function similar to PDA and FSM. The UMA architecture blockdiagram is shown in Figure 1.

A. Types of memory

1) State memory: The UMA state memory is equal to the state memory of FSM and PDA.

2) Last-In First Out (Stack): A stack $\mathbf{A}$ is a memory with N_A entries, each of width n_A. If $n_A = 8$, the operation PUSH($\mathbf{A}$,8'h00) stores eight bits, all of them zero, on top of the stack. A subsequent operation PUSH($\mathbf{A}$,8'hFF) stores eight bits, all of them one, on top of the stack. 8'hFF is returned by POP($\mathbf{A}$) and 8'h00 by another POP($\mathbf{A}$). Using two TOP($\mathbf{A}$) operations would return 8'hFF twice. POP on an empty and PUSH on a full stack returns a zero entry plus an error indication.

3) First-In First Out (Queue): A queue $\mathbf{Q}$ is a memory with N_Q entries, each of width n_Q. For $n_Q = 8$ the operation PUSH($\mathbf{Q}$,8'h00) stores eight bits, all of them zero, at the end of the queue. A subsequent operation PUSH($\mathbf{Q}$,8'hFF) stores eight bits, all of them one, at the end of the same queue. 8'h00 is returned by POP($\mathbf{Q}$) and 8'hFF by another POP($\mathbf{Q}$). The queue is then empty. Using two TOP($\mathbf{Q}$) operations would return 8'h00 twice. POP on an empty and PUSH on a full queue returns a zero entry plus an error indication.

4) Random-Access Memory (RAM): A RAM $\mathbf{R}$ is a memory with N_R entries, each of width n_R and an address of width $log_2(N_R)$. For $N_R = 8$, the operation PUSH($\mathbf{R}$, 3'b000, 8'hFF) stores eight bits, all of them one, at address 0. 8'hFF is returned by POP($\mathbf{R}$, 3'b000) and 8'h00 by another POP($\mathbf{R}$, 3'b000). Using two TOP($\mathbf{R}$, 3'b000) operations would return 8'hFF twice.

5) Content-Addressable Memory (CAM): A CAM $\mathbf{C}$ is a memory with N_C entries, each of width n_C and an address of width $log_2(N_C)$. CAMs are high-speed search engines to return the address of content-specific memory entries in one cycle. For $n_C = 8$ the operation PUSH($\mathbf{C}$,3'b000,8'h00) stores eight bits, all of them zero, at address 0. A subsequent operation PUSH($\mathbf{C}$,3'b111,8'hFF) stores eight bits, all of them one, at address 7. TOP($\mathbf{C}$,8'h00) returns the first address with content $8'h00$, i.e. 0. POP($\mathbf{C}$,8'hFF) returns the first address with content 8'FF, i.e. 7. $\mathbf{C}$ is assumed to be initialized zero. While POP deletes the entry, TOP will keep the entry. It is left up to the designer of the CAM how to organize deleted entries, to indicate if entries are not found and to return the entry if required. UMA-theory is able to manage all types of indications within a single-cycle boundary.

B. n-dimensional memory operations

If memory operations do not return status information on write accesses for evaluation in state transfer logic, read memory operations $\mathcal{X}_-$ are assigned to the input side of the state transfer function (eq. 6) and write operations $\mathcal{X}_+$ are assigned to the output side. A set of *read operations* is defined

$$\mathcal{X}_- = \{TOP(), POP()\} \tag{9}$$

for the input side of the state transfer function and a set of *write operations*

$$\mathcal{X}_+ = \{PUSH(), NOP()\} \tag{10}$$

for the output side of the state transfer function, where $\mathcal{X} = \mathcal{X}_- \cup \mathcal{X}_+$. In state transfers, only one operation per memory instance is allowed for cycle alignment reasons. This concept is called l-dimensional reading and m-dimensional writing in this paper. For l- plus m-dimensional memory operations, at least $max(l, m)$ memories are required. The state transfer function δ then becomes

$$\delta : \begin{cases} S \times \Sigma \times \mathcal{X}_-^l & \rightarrow & S \times \mathcal{X}_+^m \\ (s', x_{1+}, ..., x_{m+}) & = & \delta(s, \sigma, x_{1-}, ..., x_{l-}) \end{cases} \tag{11}$$

and the output function

$$\omega : \begin{cases} S \times \Sigma \times \mathcal{X}_-^l & \rightarrow & \Gamma \\ \gamma & = & \omega(s, \sigma, x_{1-}, ..., x_{l-}) \end{cases} \tag{12}$$

where $x_{i-} \in \mathcal{X}_{i-}$ and $x_{i+} \in \mathcal{X}_{i+}$ are read and write operations on memory X_i respectively.

C. Example of three-dimensional memory operations in state transfers

In case of a cache coherency snooping protocol with inputs rd, tag and idx, states `ID` and `RD` and RAM memories $\mathbf{TAG}$ and $\mathbf{MESI}$, a state transition from idle state (`ID`) to read state (`RD`) is given by

$$\begin{aligned} &(RD,\ PUSH(\mathbf{TAG}, idx), \\ &PUSH(\mathbf{MESI}, 4'b0100)) \\ &= ID \wedge rd \wedge (tag\ != TOP(\mathbf{TAG}, idx)) \end{aligned} \tag{13}$$

where tag is the address tag and idx is the address index, i.e. the cache line index and the read/write addresses for $\mathbf{TAG}$ and $\mathbf{MESI}$. In this transition $l = 2$ and $m = 1$, i.e. a two-fold write operation and a one-fold read operation. A sequence of three memory accesses plus state transfer is coded into a single state transition. Eq. 13 reads as follows:

(Right-hand side:) **IF** *the UMA is in idle state* `ID` *and a read operation occurs, i.e.* rd *is true, and the address tag does not equal the tag in the memory* $\mathbf{TAG}$,

(Left-hand side:) **THEN** *transfer to read state* `RD`, *store the tag to RAM* $\mathbf{TAG}$ *at address* idx, *store the exclusive bit to RAM* $\mathbf{MESI}$ *at address* idx.

III. Cache Coherency Protocol

Caches are fast and comparably small memories nearby processor cores to provide low-latency data access and to avoid expensive accesses to main memory. As caches store copies of data, data consistency problems arise in case of multiple processors operating on the same (shared) data.

A cache coherency protocol performs book-keeping of memory entries loaded and modified by one or more processor cores in a multiprocessor system. The protocol aims to secure consistent data exchange between processor cores and the memory of the system, called coherency. *Dedicated caches* are caches dedicated to and therefore accessed by only a single processor core while *shared caches* are accessed by more than one processor core. In this paper, a cache coherency protocol for dedicated caches is implemented, where cache and cache coherency hardware are assigned to a single processor core. Such assignments of core, cache and cache coherency protocol are called *adjacent* in this paper. The coherency protocol engine is implemented by a UMA which observes (snoops) the address bus for memory access activities. A well known algorithm is the MESI-protocol, where a set of control bits indicates whether a cache line is modified (M), exclusive (E), shared (S) or invalid (I).

In this implementation, the UMA has five states. The **idle state** `ID` indicates no load/store operations on the address bus. The **read state** `RD` indicates that a read operation is performed by the adjacent processor. The **write state** `WR` indicates that the adjacent processor performs a write operation. The **remote read state** `rRD` indicates that a remote and not the adjacent processor performs a read operation. The **remote write state** `rWR` indicates that a write operation is performed

by a remote and not the adjacent processor. Slightly deviating from the original protocol, cache coherency status bits M, E, S and I are defined as follows for the UMA implementation.

A cache line is set to status **modified (M)**, if a write operation is performed by the adjacent processor on data associated with the cache line. A cache line is set to status **exclusive (E)**, if a read operation is performed by the adjacent processor on data associated with the cache line. A cache line is set to status **shared (S)**, if a read operation is performed by a remote processor on data associated with the cache line. A cache line is set to status **invalid (I)**, if a write operation is performed by a remote processor on data associated with the cache line.

MESI status indications are mutually exclusive and based on the following **two assumptions** for dedicated caches.

First, it is not relevant for data coherency if a remote processor core keeps a copy of data in its cache while an adjacent processor reads data from memory. Processor cores reading data will mark the respective cache line exclusive (E) while all others, keeping the cache line as well, will mark it shared (S) at this moment. Consistency problems occur in case of remote processors writing on this address and are resolved by setting all duplicate cache entries in the system to status invalid (I).

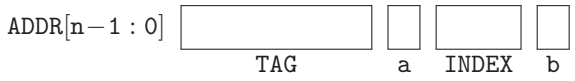
Second, it is not relevant for data coherency if an adjacent processor core keeps a copy of data in its cache while a remote processor writes data to memory. Processor writing data will mark the respective cache line modified (M) while all others, keeping the cache line as well, will mark it invalid (I) at this moment.

The two observations make a book-keeping of remote core memory activities obsolete in dedicated caches and thereby reduce the overhead for the protocol engine.

For cache entries marked invalid by the coherency protocol, an invalid is indicated by the output function, as shown by `I=1` in transitions (3.2), (C.2), (14.2) and (17.2). This supports counting of cache misses as part of hardware performance analysis. Transitions (1), (F) and (16) are split into (*.1) and (*.2) to avoid unnecessary memory action and thereby save power. If power consumption is not critical, `SET_TAG` is performed regardless of `TAG_MATCH`, i.e. `TAG_MATCH` in (*.1) can be removed and (*.2) dropped completely.

A. Address organization and cache coherency

In computer systems, memory addresses are organized by page-tag and byte-index, where byte-index is a defined set of low order address bits to index bytes in random access memory, and page-tag is the remaining high-order bits. As every cache line might store a power of two number of bytes, the number of bytes in a cache line is given by $N = 2^b$ where $0 \leq b < n$. Hence, $\mathtt{b} = log_2 N$ low order index bits of the memory address might be unused in the cache address. In case of associative cache organization, a certain number of low order bits of the tag `a` are used to be associated with sets of associative cache memory.



In case of an m-way associative cache, $\mathtt{a} = log_2(m)$ low order bits of the tag become part of the cache address to associate cache blocks. The tag is reduced by `a` bits.

$$tag = \mathtt{ADDR[\,n-1:a+INDEX+b\,]} \quad (14)$$

$$idx = \mathtt{ADDR[\,a+INDEX+b-1:b\,]} \quad (15)$$

One-way associative caches are obtained for $a = 0$. For simplicity let $a = 0$ in this paper so that the cache is one-way associative.

B. UMA graph for a cache coherency protocol

A refined version of the protocol in [14] has been implemented in this work. The cache coherency protocol is implemented by a set of states $S = \{ID, RD, WR, rRD, rWR\}$ and an initial state $S_0 = ID$, named "idle". RD and rRD are states to account for "read" and "remote read" situations while WR and rWR are states to indicate "write" and "remote write" situations. The inputs to decide on situations are $rd, wr, cp, res\_n, addr(31:0)$ where rd indicates read accesses, wr indicates write accesses on address $addr$. cp indicates whether the access is performed by the adjacent core ($cp = 1$) that the coherency protocol is accounting for or performed by a remote core ($cp = 0$). For this implementation 32-bit addresses, six index bits $idx = addr(7:2)$ and 24 tag bits $tag = addr(31:8)$ are used. The protocol operates on two memories of type RAM, called **MESI** and **TAG**. **MESI** and **TAG** are addressed by idx to store the cache coherency status and address tag respectively. Figure 2 shows the UMA graph. State transfer and output expressions are shown in Table IV. To simplify expressions, the implementation allows the use of constants (Table III).

IV. Automated HDL generation of UMA graphs

The tools chain is built on an XML-like format to define the UMA graph for automated Verilog HDL complilation. A later version is supposed to support a visual graph representation with automatic import and export function of the shown configuration files. Simple configuration files consist at least of the series of tags, `name`, `inputs`, `outputs`, `states`, `stateTransfers` (Figure 3). The `memory` tag is required for PDAs and UMAs. The `expr` tag is required to define constant expressions.

Table I
UMA STATE TRANSFER EXPRESSION FOR A TRANSITION FROM IDLE STATE TO READ STATE.

Comment	Expression (to be entered in a single line)
Source state	`ID`,
Condition	`rd && cp && addr[31 : 8] != TOP(TAG, addr(7 : 2))`,
Destination state	`RD`,
Output assignment	`I = 1'b0`,
Memory activity	`PUSH(TAG, addr[7 : 2], addr[31 : 8]) PUSH(MESI, addr[7 : 2], EXCLUSIVE)`

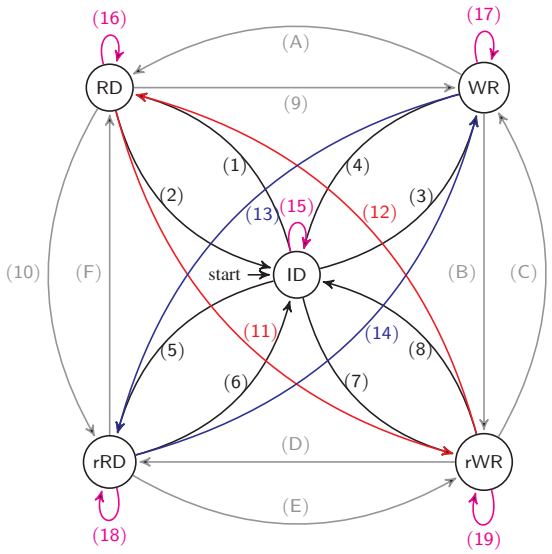


Figure 2. UMA graph for cache coherency protocol. State transition conditions are given in Table III and Table IV.

A. Tags definitions

1) name: $<$ `name` $>$ `MESI` $<$ /`name` $>$ defines an UMA named MESI, which becomes the HDL module name.

2) inputs: $<$ `inputs` $>$ `clk`, `res_n`, `rd`, `wr`, `cp`, `addr`[31 : 0] $<$ /`inputs` $>$ defines a series of inputs to be used in expressions, state transfer and output functions.

3) outputs: $<$ `outputs` $>$ `I` $<$ /`outputs` $>$ defines an output `I` to indicate an invalid situation, where a cache entry is marked invalid in case of a remote write access on a memory location accounted by the protocol.

4) states: $<$ `states` $>$ `ID`(100), `RD`(000), `WR`(001), `rRD`(010), `rWR`(011) $<$ /`states` $>$ defines a set of states with manual encoding.

5) stateTransfers: $<$ `stateTransfers` $>$ *state transitions* $<$ /`stateTransfers` $>$ defines the state transfer function by a series of state transitions, line by line.

6) state transitions: Each state transition is defined by a single line with expression of the form

$$source\ state,\ condition,\ destination\ state,\ output\ assignment,\ memory\ activity$$

```
<name> MESI </name>
<type> UMA, Mealy </type>
<inputs> clk,res_n,rd,wr,cp,addr[31:0] </inputs>
<outputs> I </outputs>
<triggeredge> posedge clk,negedge res_n </triggeredge>
<stateCoding> man </stateCoding>
<states> ID(100), RD(000), WR(001), rRD(010), rWR(011) </states>
<memory>
TAG,ram,24,64; // 64 entries, 24 bit each
MESI,ram,4,64; // 64 entries, 4 bit each
</memory>
```

Figure 3. UMA configuration file (header section) for a cache coherency protocol.

Table II
UMA MEMORY CONFIGURATIONS FOR THE CACHE COHERENCY PROTOCOL.

memory name	memory type	memory width	memory depth
`TAG`	`ram`	24	64
`MESI`	`ram`	4	64

where src is a source state, $condition$ is a boolean expression containing l-fold memory read operations, i.e. $\mathcal{X}_{-}^{l}$. dst is the destination state and $output\ assignment$ is an assignment to one or more outputs. $Memory\ activity$ indicates m-fold memory write operations, i.e. $\mathcal{X}_{+}^{m}$. An example state transition corresponding to the expression in eq. 13 is shown in Table I. There, a memory named TAG of type `ram` is defined. TAG has a width of 12 bits and sixteen entries, i.e. the address width is four bits.

7) memory: $<$ `memory` $>$ $memory\ definitions$ $<$ /`memory` $>$ defines a series of memories.

8) memory definitions: A memory is defined by a line with an expression of the form

$$memory\ name,\ memory\ type,\ width,\ depth$$

where $memory\ name$ is the name of the memory, here TAG and $MESI$. $Memory\ type$ is `cam`, `ram`, `queue` or `stack`. $Depth$ is the number of bits per memory entry and $number$ is the number of entries. The address width is determined automatically from the log_2 of the number of entries. An example memory configuration is shown in Table II.

9) expr tag: $<$ `expr` $>$ $constant\ expression\ definitions$ $<$ /`expr` $>$ defines a constant expression to be used in state transfer and output expressions.

10) constant expression definitions: Constant expression can be used to easily avoid large expressions

```
RAM #(4,64)        MESI_RAM(.clk(MESI_RAM_clk), .res_n(MESI_RAM_res),
                   .wr(MESI_RAM_wr), .wr_addr(MESI_RAM_wr_addr[5:0]), .wr_data(MESI_RAM_wr_data[3:0]),
                   .rd(MESI_RAM_rd), .rd_addr(MESI_RAM_rd_addr[5:0]), .rd_data(MESI_RAM_rd_data[3:0]) );
wire READ;         assign READ = res_n & cp & rd;
wire TAG_MATCH;    assign TAG_MATCH = POP_TAG_RAM(addr[7:2])==addr[31:8];
always @( * )      begin
                   case ( STATE )
                   ID : begin
                        READ & ~ TAG_MATCH ) begin
                        NEXT_STATE <= RD;
                        TAG_RAM_status <= PUSH_TAG_RAM(addr[7:2],addr[31:8]);
                        MESI_RAM_status <= PUSH_MESI_RAM(addr[7:2],EXCLUSIVE);
                   end ...
```

Figure 4. Auto-generated Verilog HDL for memory instantiation and the state transition shown in Table I

Table III
EXPRESSION CONSTANTS FOR THE CACHE CONHERENCY PROTOCOL SHOWN IN TABLE IV ON PAGE 41.

Constant		Expression
READ	=	res_n & cpu & rd
WRITE	=	res_n & cpu & wr
R_READ	=	res_n & !cpu & rd
R_WRITE	=	res_n & !cpu & wr
NOP	=	!res_n || (!rd & !wr)
TAG_MATCH	=	TOP(R,addr(7:2))==addr(31:8))
IS_MODIFIED	=	TOP(MESI,addr(7:2))==4'b1000)
IS_EXCLUSIVE	=	TOP(MESI,addr(7:2))==4'b0100)
IS_SHARED	=	TOP(MESI,addr(7:2))==4'b0010)
IS_INVALID	=	TOP(MESI,addr(7:2))==4'b0001)
SET_TAG	=	PUSH(R,addr(7:2),addr(31:8))
SET_MODIFIED	=	PUSH(MESI,addr(7:2),4'b1000)
SET_EXCLUSIVE	=	PUSH(MESI,addr(7:2),4'b0100)
SET_SHARED	=	PUSH(MESI,addr(7:2),4'b0010)
SET_INVALID	=	PUSH(MESI,addr(7:2),4'b0001)

in state transfers, in particular when complex memory operations are involved. A constant expression is defined by a line of the form $constant = expression$. The constant expressions used in the cache coherency protocol are shown in Table III.

11) Others: Other tags are for example `type` to choose between `uma`, `pda`, or `fsm` and architectures `mealy`, `moore` and `smoore`. `TriggerEdge` is used to configure a positive (`pos`) or negative (`neg`) clock edge, `stateCoding` to select manual, one-hot or gray encoding. These types of tags are defaulted automatically if unused or otherwise contained in the header section of a UMA definition file as shown in Figure 3.

B. HDL generation

Boolean logic for state transfer, output functions and memory instances are compiled at Register Transfer Level (RTL) with Verilog HDL. The entire protocol including testbench is derived from a 140 lines configuration file.

The Cache Coherency Protocol for the finite state graph shown in Figure 2 instantiates two memories of type `ram`, named `TAG` and `MESI`, each with 64 entries, addressed by 6 address bits, i.e. $index = addr(7:2)$. Every cache line is 32 bits wide and thereby stores four bytes so that the b-field of the address is 2 bits wide, i.e. $index(1:0)$. Memory `TAG` stores the 24 bits wide address tag $addr(31:8)$ and memory `MESI` stores one-hot encoded coherency settings for cache lines addressed by $index(7:2)$, i.e. Modified, Exclusive, Shared and Invalid.

C. Module interface

The module interface is derived from the header section shown in Figure 3. Name, inputs and output definitions are used straight forward in the module interface.

D. Memory instantiation

Memories are instantiated as defined in the configuration file. The code fragments in Figure 4 shows the definition and instantiation of the RAM $MESI$ and the state transition discussed in Table I.

E. State transfer with memory read and output function with write access

State transfers are derived from the $<$ `stateTransfers` $>$ tag as shown Table I. Verilog code is compiled as shown in Figure 4, where the state transfer function is built on a case statement encapsulated in an always block. There, the current state is evaluated and the state transitions shown in the UMA graph (Figure 2) provide the expressions for the state transition logic (Table IV).

1) Example: In Table I and arc 1.1 in Table IV: **IF** the protocol engine is in state idle (`ID`), a read operation occurs and the corresponding read access to the `TAG` RAM causes a tag match, i.e. (`READ` & `TAG_MATCH`), **THEN** the UMA transfers to read state (`RD`) and a write access is performed to change the coherency status of the cache entry to Exclusive when the read state is reached. This complex operation is defined by a single state transfer. The corresponding waveform is shown in Figure 5.

F. Output function

Outputs are set according to the state and state transfer conditions in the case statement in case of a *Mealy* architecture and in a separate case statement and according to the state only in case of a *Moore* or *Simple Moore* architecture.

Table IV
STATE TRANSITION DEFINITIONS FOR THE GRAPH REPRESENTATION OF THE CACHE COHERENCY PROTOCOL IN FIGURE 2. EXPRESSION CONSTANTS ARE DEFINED IN TABLE III ON PAGE 40.

Arc	SRC State	Condition	TGT State	Output	Memory operation
(start,2,4,6,8,15)	*	NOP	ID		
(1.1)	ID	READ & TAG_MATCH,	RD,	,	SET_EXCLUSIVE
(1.2)	ID	READ & !TAG_MATCH,	RD,	,	SET_TAG SET_EXCLUSIVE
(3.1)	ID	WRITE & TAG_MATCH & !IS_INVALID,	WR,	,	SET_MODIFIED
(3.2)	ID	WRITE & TAG_MATCH & IS_INVALID,	WR,	I=1,	
(5)	ID	R_READ & TAG_MATCH,	rRD,	,	SET_SHARED
(7)	ID	R_WRITE & TAG_MATCH,	rWR,	,	SET_INVALID
(9)	RD	WRITE & TAG_MATCH,	WR,	,	SET_MODIFIED
(A)	WR	READ & TAG_MATCH,	RD,	,	SET_EXCLUSIVE
(B)	WR	READ & TAG_MATCH,	rWR,	,	SET_INVALID
(C.1)	rWR	WRITE & TAG_MATCH & !IS_INVALID,	WR,	,	SET_MODIFIED
(C.2)	rWR	WRITE & TAG_MATCH & IS_INVALID,	WR,	I=1,	
(D)	rWR	R_READ & TAG_MATCH & IS_MODIFIED,	rRD,	,	SET_SHARED
(E)	rRD	R_WRITE & TAG_MATCH,	rWR,	,	SET_INVALID
(F.1)	rRD	READ & TAG_MATCH,	RD,	,	SET_EXCLUSIVE
(F.2)	rRD	READ & !TAG_MATCH,	RD,	,	SET_TAG SET_EXCLUSIVE
(10)	RD	R_READ & TAG_MATCH,	rRD,	,	SET_SHARED
(11)	RD	R_WRITE & TAG_MATCH,	rWR,	,	SET_INVALID
(12.1)	rWR	READ & TAG_MATCH,	RD,	,	SET_EXCLUSIVE
(12.2)	rWR	READ & !TAG_MATCH,	RD,	,	SET_TAG SET_EXCLUSIVE
(13)	WR	R_READ & TAG_MATCH,	rRD,	,	SET_SHARED
(14.1)	rRD	WRITE & TAG_MATCH & !IS_INVALID,	WR,	,	SET_MODIFIED
(14.2)	rRD	WRITE & TAG_MATCH & IS_INVALID,	WR,	I=1,	
(16.1)	RD	READ & TAG_MATCH,	RD,	,	SET_EXCLUSIVE
(16.2)	RD	READ & !TAG_MATCH,	RD,	,	SET_TAG SET_EXCLUSIVE
(17.1)	WR	WRITE & TAG_MATCH & !IS_INVALID,	WR,	,	SET_MODIFIED
(17.2)	WR	WRITE & TAG_MATCH & IS_INVALID,	WR,	I=1,	
(18)	rRD	R_READ & TAG_MATCH,	rRD,	,	SET_SHARED
(19)	rWR	R_WRITE & TAG_MATCH,	rWR,	,	SET_INVALID

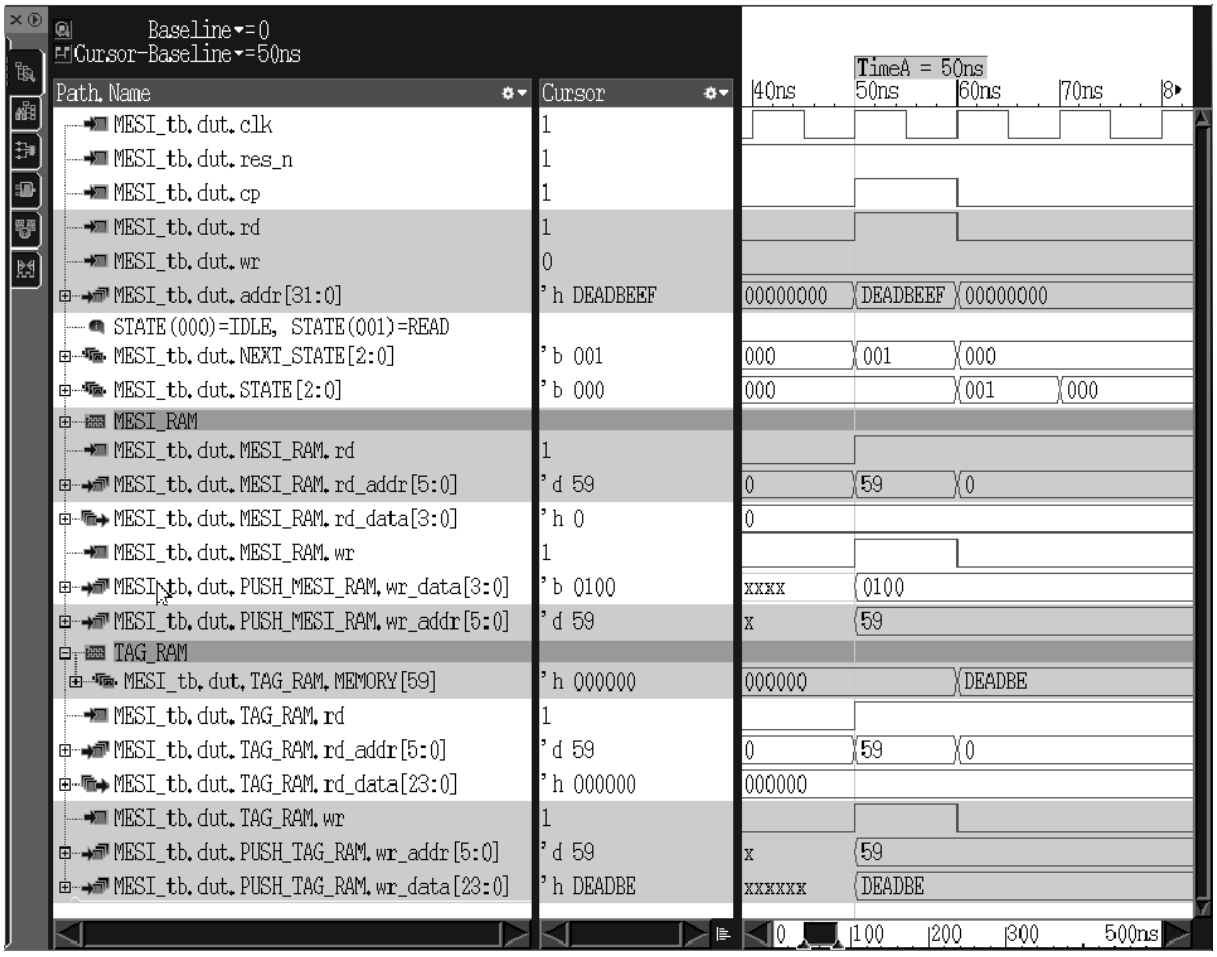


Figure 5. Read access of own CPU ($cp = rd = 1$) causing a TAG to be stored in TAG_RAM and coherency status 'exclusive', i.e. 0100, into MESI_RAM.

V. Conclusions

Universal Memory Automata (UMA) provide a formalism for using complex memory configurations and operations in finite state graphs. Compared to Push Down Automata (PDA), where only a stack and state memory are utilized or Finite State Machine (FSM), where only a state memory is utilized, Universal Memory Automata (UMA) extend these concepts by additional memory organizations, e.g. Queue, RAM and CAM. UMAs allow an arbitrary set and configuration of memories to be operated at the same time, where complex and multiple read/write memory operations are included in the state transfer and output functions. This feature does not enable UMAs to process another class of languages in the Chomsky Hierarchy but it makes them more flexible and intuitively applicable than PDAs and FSMs when implementing complex applications. With only 140 lines of parametrized configuration, a complex and adaptable Cache Coherency Protocol including test-bench is automatically implemented in the Hardware Description Language (HDL) Verilog at Register Transfer Level (RTL). The proposed theory is in general applicable to all types of memories within the state transfer cycle alignments, i.e. single-cycle read/write access. This shows the potential of Universal Memory Automata (UMAs) and provides the possibility for further extensions.

References

[1] T.L. Booth. 1962. *Sequential Machines and Automata Theory* (1st ed.). Number 67-25924. John Wiley and Sons, Inc., New York. Library of Congress Card Catalog.

[2] J. Carroll and D. Long. 1989. *Theory of Finite Automata with an introduction to Formal Languages*. Prentice Hall.

[3] A. Gill. 1962. *Introduction to the Theory of Finite-state Machines*. McGraw-Hill.

[4] J.E. Hopcroft and J.D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, MA. ISBN 0-201-02988-X.

[5] L. Boasson, J.-M. Autebert, J. Berstel. 1997. *Context-Free Languages and Push-Down-Automata*. Vol. 1. Springer-Verlag. 111-174.

[6] J.D. Ullman, J.E. Hopcroft. 1967. "Nonerasing Stack Automata". *Journal of Computer System Sciences 1 (1967)*, 166–186. https://doi.org/10.1016/s0022-0000(67)80013-8.

[7] E.J. McCluskey. 1965. *Introduction to the Theory of Switching Circuits* (1st ed.). McCraw-Hill, New York. Library of Congress Card Catalog.

[8] M. Minski. 1967. *Computation: Finite and infinite Machines* (1st ed.). Prentice-Hall, New Jersey.

[9] L.J Stockmeyer, R.E. Ladner, R.J. Lipton . 1984. "Stack Automata and Compiling". *SIAM J. Comput.* 13, (1984), 135–155. ISSN 0097-5397. https://doi.org/10.1137/0213010.

[10] M.A. Harrison, S. Ginsburg, S.A. Greibach. 1967. "One-way Stack Automata". *J. ACM 14*, 1 (1967), 389–418. https://doi.org/10.1145/321386.321403

[11] M.A. Harrison, S. Ginsburg, S.A. Greibach. 1967. "Stack Automata and Compiling". *J. ACM 14, 1 (1967)*, 172–201. https://doi.org/10.1145/321371.321385.

[12] S. Seshu. 1963. "Introduction to the theory of finite-state machines". *Proc. IEEE 51*, 9, Sep. 1963, 1275–1275. ISSN 0018-9219. https://doi.org/10.1109/PROC.1963.2548.

[13] M. Snipser. 1997. *Introduction to the Theory of Computation*. PWS Publishing. ISBN 0-534-94728-X. Section 2.2: Pushdown Automata, pp. 101-114.

[14] P. Zyska. 2018. *Implementation of a cache coherency protocol with extended Push Down Automata*. Bachelor Thesis, HTWG Konstanz, University of Applied Sciences.

Matthias W. Fertig received his academic degree Dipl. Inf. (MSCE) from the University of Mannheim in 2001. From 2003 to 2011 he worked at the IBM Research and Development GmbH in Böblingen, Germany, in the field of microprocessor design for IBM P- and Z-Servers. He holds patents in the field of computer architecture and silicon photonics and is a recipient of several IBM innovation and plateau awards. In 2011 he received a Dr. rer. nat. (PhD) for his research on electromagnetic simulation at the department of Optoelectronics of Heidelberg University. From 2011 to 2015 he worked as a project manager for Dialog Semiconductor and as a CPM for Volvo CE. Since 2015 he has been a professor of computer engineering and holds the professorship for digital systems at the Konstanz University of Applied Sciences.