Selective State Retention Design using Symbolic Simulation

Ashish Darbari, Bashir M. Al Hashimi
School of Electronics and Computer Science
University of Southampton
Southampton, England
{*ad06v,bmah*}@*ecs.soton.ac.uk*

David Flynn, John Biggs
ARM, Cambridge, England
{*david.flynn,john.biggs*}@*arm.com*

***Abstract*—Addressing both standby and active power is a major challenge in developing System-on-Chip designs for battery-powered products. Powering off sections of logic or memories loses internal register and RAM states so designers have to weigh up the benefits and costs of implementing state retention on some or all of the power gated subsystems where state recovery has significant real-time or energy cost, compared to resetting the subsystem and re-acquiring state from scratch. Library IP and EDA tools can support state retention in hardware synthesized from standard RTL, but due to the silicon area costs there is strong interest in only retaining certain selective state for example the "architectural state" of a CPU to implement sleep modes. Currently there is no known rigourous technique for checking the integrity of selective state retention, and this is due to the complexity of checking that the correctness of the design is not compromised in any way. The complexity is exacerbated due to the interaction between the retained and the non-retained state, and exhaustive simulation rapidly becomes infeasible. This paper presents a case study based on symbolic simulation for assisting the designers to design and implement selective retention correctly. The main finding of our study is that the programmer visible state or the architectural state of the CPU needs to be implemented using retention registers whilst other micro-architectural enhancements such as pipeline registers, TLBs and caches can be implemented using normal registers without retention. This has a profound impact on power and area savings for chip design. By selectively retaining the state of the programmer's "architectural" model and not the increasing proportion of extra state, one can incorporate energy-efficient sleep modes. To the best of our knowledge this is the first study in the area of rigourous design and implementation of selective state retention.**

I. Introduction

Power consumption and battery life in particular are major challenges in the design of ever more complex System-on-Chip designs at the heart of many products. Minimizing the standby power consumption as well as the active power is crucial. Power gating (or MTCMOS) [1, 2] is an effective technique for switching off sections of a design which is gaining support in mainstream EDA tools.

State-Retention can be supported in power gated designs either explicitly in software or transparently in hardware using a balloon latch [1, 3]. Total hardware state retention, and power gating, can be implemented with current EDA tools, together with the addition of unified power format [4] (UPF) annotation of power intent today. UPF specifies the supply network, switches, isolation, retention and other aspects relevant to power management of an electronic system. Much like clock gating that can be inferred transparently from a RTL design description that has been coded to meet certain style guidelines, retention of all registers may be inferred transparently without re-coding of the RTL. RTL descriptions in fact assume state retention globally; every sequential process relies on the fact that every logical state value is persistent between activation events.

Hardware state retention of every register in a design however has a cost. A retention register has a larger footprint than a normal (volatile) register. The impact on a design with many registers may not just be an increase in area but also a drop in performance due to the longer wires.

Selective, or partial, state retention sounds attractive since the area cost of retention can be scaled down in proportion to the state being retained. As much as the idea of selective state retention is compelling, to ensure that it can be done safely and correctly is a major challenge in low power design. Traditionally, a designer validates the behavior of a design from reset, using simulation or formal verification techniques. With selective retention the problem grows with the product of the number of different logical states that the selective retained state could arbitrarily hold. Conventional simulation (using 0s and 1s) rapidly becomes infeasible [5] even when there is no retention. In case of retention the state-space grows massively because of the interaction between the retained and non-retained state.

Recently some work has been done in the area of functional verification of low-power designs. Crone and Chidolue [6] have investigated RTL level verification of power management techniques for low power designs. Although the general theme of their work and ours appear very similar - the use of verification or symbolic simulation for low power, there is a fundamental difference. In our work we use symbolic simulation and verification to design and implement "selective" state retention for low power circuits, whereas in [6] they verify a low-power design with a given power management scheme usually given by a UPF format.

To the best of our knowledge this is the first study in the area of rigourous design and implementation of selective state retention. This paper presents a case study based on symbolic simulation for assisting the designers to design and implement selective retention correctly. We believe that an approach such as ours that is grounded in symbolic simulation and model checking, is the right step towards sound design and implementation of selective retention for modern day low power SOC design. The main finding of our study is that the programmer visible state or the architecural state of the CPU needs to be implemented using retention registers whilst other micro-architectural enhancements such as pipeline registers, translation look-aside buffers (TLBs) and caches can be implemented using normal registers without retention. This has a profound impact on power and area savings for chip design.

II. Selective State Retention

If implemented using a special register known as a *retention register*, it is possible to restore the entire subsystem to match the RTL abstraction before and after entering low power mode provided every state-bit is preserved. Retention registers have some form of "sleep" or "retain" control that enables the register to retain a state. An example retention register is shown below in Figure 1. When NRET is high, the register works like a normal register (without retention), and we say it is in *sample* mode. When NRET is held low, the register goes in a *hold* mode and retains the state it was in just before NRET was held low. In order to reset the register one can assert NRST low.

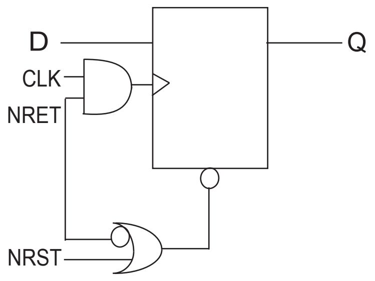

Fig. 1. An Emulated Retention Register

Note that the register shown in Figure 1 is an emulation of a retention register shown at the gate-level. In practice [2], retention registers have standard manufacturing scan support multiplexing and capture state into a weak, low-leakage, retention latch structure.

Since every retention register contributes to additional leakage power, so partial state retention instead of full retention should result in lower standby power, and a reduction in high-fan-out buffers of retention controls. However, selective state retention designs are more complex to design and verify than full-state retention designs. Take the case of a microprocessor core, where there is typically a certain state (register banks, processor status flags and mode information for example) that is visible to the programmer and must be preserved from the software perspective by any hardware state retention scheme. As a starting point the task that the programmer or operating system requires, appears to be a minimum level of state to retain. However, in a system that supports virtual memory, the program counter and address pointers assume that the memory management is programmed and persistent. Therefore the entire kernel-level configuration and programming state must also be retained. Caches are another example of state that may be desirable to retain in some modes of power-down. Although the cache is theoretically transparent to the user-state program, the cost of discarding and re-loading the cache is costly in terms of both energy and time.

A. Challenges with selective retention

As much as the idea of selective state retention is compelling to design and implement, to ensure that it can be done safely and correctly is a major challenge in low power design.

One of the goals of our project has been to discover the minimal architectural state of the CPU that needs to be retained in case of selective state retention without compromising the correctness. The aim (shown in Figure 2) is to ensure that a design with selective retention makes the transition from present state via the sleep state to a resumed state such that when it makes a transition to a next state from the resumed state, the next state is identical to the state that is reached from present state without retention.

For our analysis we are mostly concerned with the architectural state of the CPU also sometimes known as the programmer visible state. This usually comprises state of the PC, register banks, and instruction and data memory. When we use the term state, we mean the architectural state.

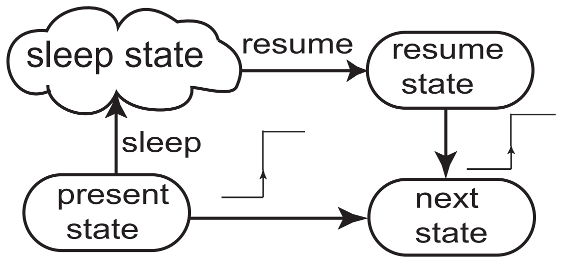

Fig. 2. Selective State Retention. The goal is to ensure that a design with selective retention makes the transition from present state via the sleep state to a resumed state such that if a transition is made from the resumed state to the next state (on a clock edge) the resulting next state is identical to the next state reached from the present state without retention.

III. Symbolic Simulation Aided Approach

We made use of the formal model checking technique called Symbolic Trajectory Evaluation (STE). STE has been used successfully in large-scale datapath verification [7] of microprocessors and it gains its strength by combining the ideas of *ternary modelling* (using 0, 1 and X) with *symbolic simulation* (using symbolic variables) over *time*. The presence of Xs provides abstraction necessary to handle the complexity of verifying symbolic properties during model checking. This happens by assuming a monotonicity property of the simulation algorithm — any binary value resulting when simulating patterns containing X's would also result when each X is replaced by either a 0 or a 1.

Symbolic trajectory evaluation employs a ternary circuit state model, in which the usual binary values 0 and 1 are augmented with a third value X that stands for an unknown. We introduce a partial order relation $\sqsubseteq$, with X $\sqsubseteq$ 0 and X $\sqsubseteq$ 1. The relation orders values by information content: X stands for a value about which we know nothing, and is therefore below the specific values 0 and 1. This ordering is lifted to an ordering over three-valued states and sequences.

Specifications in STE, take the form of what are known as *symbolic trajectory formulas*. Formally, we define the syntax of formulas [7, 8] as follows:

Definition 1. *Syntax of STE formulas*

$$\begin{array}{lll} f & \triangleq & n \textsf{ is } 0 & \text{- node } n \text{ has value } 0 \\ & | & n \textsf{ is } 1 & \text{- node } n \text{ has value } 1 \\ & | & f_1 \textsf{ and } f_2 & \text{- conjunction of formulas} \\ & | & f \textsf{ when } G & \text{- } f \text{ is asserted only when } G \text{ is true} \\ & | & \textsf{N}f & \text{- } f \text{ holds in the next time step} \end{array}$$

where f_1 and f_2 range over formulas, $n \in string$ ranges over the nodes of the circuit, and G is a propositional formula over Boolean variables (i.e. a Boolean 'function') called a *guard*. The advantage of using a Boolean expression is in specifying conveniently many different operating conditions in a compact form. The various guards that occur in a trajectory formula can have variables in common, so this mechanism gives STE the expressive power needed to represent interdependencies among node values. We also use a convenient form of expressing the temporal formula, by using *from* and *to* functions [9].

$$f \textsf{ from } i \textit{ to } j \triangleq \textsf{N}^i f \textsf{ and } \textsf{N}^{i+1} f \textsf{ and } \ldots \textsf{ and } \textsf{N}^{j-1} f$$

where the convention is that $\textsf{N}^0 f = f$.

The key feature of STE logic is that for any trajectory formula f and assignment ϕ, there exists a unique weakest sequence that satisfies f. This sequence is called the *defining sequence* for f and is written as $[f]^\phi$.

Definition 2. *Defining Sequence*

$$\begin{array}{lcl} [m \textsf{ is } 0]^\phi\ t\ n & \triangleq & \textit{if } (m{=}n) \wedge (t{=}0) \textit{ then } 0 \textit{ else } X \\ [m \textsf{ is } 1]^\phi\ t\ n & \triangleq & \textit{if } (m{=}n) \wedge (t{=}0) \textit{ then } 1 \textit{ else } X \\ [f_1 \textsf{ and } f_2]^\phi\ t\ n & \triangleq & ([f_1]^\phi\ t\ n) \sqcup ([f_2]^\phi\ t\ n) \\ [f \textsf{ when } G]^\phi\ t\ n & \triangleq & \textit{if } (\phi \models G) \textit{ then } ([f]^\phi\ t\ n) \textit{ else } X \\ [\textsf{N}f]^\phi\ t\ n & \triangleq & \textit{if } (t{\neq}0) \textit{ then } ([f]^\phi\ (t{-}1)\ n) \textit{ else } X \end{array}$$

Defining trajectory of a formula is its defining sequence with the added constraints on state transitions imposed by the circuit model $\mathcal{M}$.

Definition 3. *Defining Trajectory*

$$\begin{array}{lcl} [\![f]\!]^\phi\ \mathcal{M}\ 0\ n & \triangleq & [f]^\phi\ 0\ n \\ [\![f]\!]^\phi\ \mathcal{M}\ t\ n & \triangleq & [f]^\phi\ t\ n \sqcup \mathcal{M}\,([\![f]\!]^\phi\ \mathcal{M}\ (t{-}1))\ n \end{array}$$

Verification takes place by testing the validity of an *assertion* or property against a given *model*. A property is of the form $(A \Rightarrow C)$, where both A and C are trajectory formulas. Intuitively, the antecedent A provides the stimuli to the circuit model $\mathcal{M}$, and the consequent C expresses what the designer expects to see.

At the heart of STE model checking is an efficient implementation algorithm, that relies on the calculation of finite weakest sequences (defining sequence) and trajectories (defining trajectory) of the formulas, and comparing them (via the lattice ordering $\sqsubseteq$), point-wise for all nodes in C, up to the depth of the next-time operators in C [8].

$$\mathcal{M} \models A \Rightarrow C \ \triangleq\ \forall t\, n.\ [C]^\phi\ t\ n \sqsubseteq [\![A]\!]^\phi\ \mathcal{M}\ t\ n$$

In practice, every successful STE run i.e., a run that returns the value True, is a theorem that holds for all the Boolean variables mentioned in the property. However, when the outcome of an STE model checking run is a counter-example, it is an indication of a bug in the hardware, and in the context of STE, it means that if we can come up with a satisfying assignment of Boolean values True (logic 1) and False (logic 0) to the Boolean variables in the counter-example, one can explicitly reveal the trace (consiting of 0s and 1s) that would be responsible for the bug. Usually there is more than one way to satisfy the counter-example, and this means that in one symbolic model checking run, we can succintly capture all the possible traces.

A. Proposed Approach

To evaluate the effect of selective state retention we architected a 32-bit RISC core adapted from [10]. Retention register shown in Figure 1 was used for all the state holding elements that comprise the programmer visible state namely the PC, the Instruction Memory, Register Banks and Data Memory. The core is then synthesized using Altera's Quartus II to a Berkeley Logic Interchange Format (BLIF). The BLIF model is then compiled to a finite-state machine (FSM) using *exlif2exe* that is provided with the STE model checker Forte [11]. We then carried out property checking using STE to check if the selectively retained state ensures that the CPU goes from a given arbitray present state to the next state through the sleep and resume modes. Retention has priority over reset. This means that if NRET is in sample mode or held high, reset will have the usual effect of resetting the retained state. To prevent the contents of the retained state from being reset, NRET needs to be held low. The desired sequence of operations to put the CPU in sleep mode is as follows:

1. Stop the clock.

2. Assert NRET low, i.e., put it in hold mode.
3. Reset NRST is then asserted active low.

The resume mode is chronologically reverse of the sleep mode. To resume, we assert NRST as high, then NRET as high and start the clock. We usually give a unit delay in between switching these on and off.

In our approach we use the logic of STE to write properties to specify:

1) Any arbitrary present state of the CPU.
2) Specify when (what times) and how (by controlling clock, NRET, and NRST), the CPU goes to sleep and resumes afterwards.
3) State of the CPU during sleep and after resume.
4) Expected next state

The challenge is in precisely mapping these events onto STE properties and to ensure that the resulting properties are verified efficiently. Fundamental to our property checking approach is the constrcution of two sets of properties for each functional unit in question such as fetch, decode, control, execute and write-back. The skeleton of the first set of properties (**Property I**) is shown below:

Property I

$$\mathcal{M} \models \texttt{clock and } A \texttt{ and } (\mathsf{NRET} \texttt{ is } \mathtt{T} \text{ from } i \text{ to } j) \Rightarrow C$$

where i and j are natural numbers postulating the start and stop times for NRET.

This property is designed to verify that the CPU works correctly (as if without retention) in the case when NRET is held high throughout a given run. This is done to check the fact that when NRET is in sample mode it makes the retention registers behave as normal registers and therefore enables the CPU to function as if there is no retention at all. The trajectory formula `clock` specifies an uninterrupted rising edge clock while the antecedent A specifies the present (but arbitrary) state, and C the desired next state. Because we use Boolean symbolic variables to denote the states and these variables can take any combination of scalar 0s and 1s, a single symbolic state is an abstraction of many diffferent states obtained by instantiating the variables to 0 and 1. In the next step we introduce the sleep and resume operations making sure we preserve the sequence of operations outlined above. Thus, a sleep operation (`sleep`) specifies that the clock is stopped first, then NRET is held low, and then NRST is asserted. Similarly, a resume operation (`resume`) asserts that the reset NRST is de-asserted first, then NRET is asserted high, followed by the resumption of clock. The skeleton of the second set of properties (**Property II**) we verify, is presented below:

Property II

$$\mathcal{M} \models (\texttt{clock and sleep and resume and } A) \Rightarrow C$$

The formula `clock` in this case represents the rising edge clock specifying the start of the simulation. Note that we do

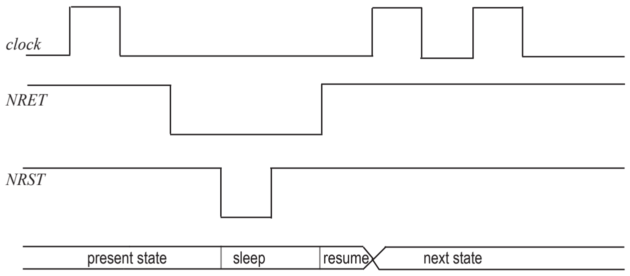

Fig. 3. Waveforms showing how the present state evolves to the next state through the sleep and resume states. The sleep (`sleep`) and resume operations (`resume`) are specified by controlling the clock, NRET, and NRST. In our approach using STE, the antecedent specifies the present state, and `sleep` and `resume` operations. The consequent specifies the next state, and the sleep and resume state.

stop this clock and restart again, but this is defined by `sleep` and `resume` operations as explained above. Present state is given by A, whilst the consequent C describes the state of the CPU during sleep, resume state, and the expected next state. Note the difference between `sleep` and `resume` operations and sleep and resume states. We define the operations by specifying the state of the clock, NRET, and NRST, whilst the respective states would typically be specified for all the nodes in a given functional unit. For example the sleep and resume states could talk about the state of the PC during sleep and after resumption, in case of Fetch unit. Figure 3 shows the relationship between the sleep and resume operations and their impact on how the present state evolves via the sleep and resume states to a next state.

B. Selective Retention in Control Unit

Using the above approach we constructed a suite of STE properties describing all the functional units of the 32-bit RISC core shown in Figure 4. The motivation to keep the core unpipelined was to discover a minimal architectural state that needs to be retained during sleep mode. In total for **Property I**, we developed 26 properties (2 for fetch, 6 for decode, 11 for control, 6 for execute and 1 for write back), to check the functionality of the core in the presence of NRET being held high throughout the simulation.

In line with **Property II**, these properties were then modified to incorporate the sleep and resume operations, and were then re-checked again to see if they still hold.

What we discovered in this process was that when the CPU would resume post a sleep operation, most of the programmer visible state was retained properly, however the control unit would malfunction. The reason is that during sleep, an asynchronous reset (NRST) signal resets the input values of the control unit, and we need to properly initialise them after the resume operation, or else, the state of the control would be some incorrect value that would subsequently cause an incorrect operation of the CPU. Programmer visible state was retained properly because it was designed using retention registers. To fix this problem, we inserted a 6-bit pipeline register - Instruction Fetch Register (IFR) (designed using normal non-retention registers) between $\texttt{Instruction}[31{:}26]$ and the control unit. On a reset, the values of this register would be reset. Subsequent to the resume operation, (see Figure 3), correct values would be updated from the retained Instruction Memory on the next rising edge of the clock. We do not pipeline $\texttt{Instruction}[25{:}0]$ since these values are not affected during the reset operation. But in theory one can pipeline these as well without compromising correctness. In a unpipelined, simple CPU, an IFR is not necessary. In a more realistic pipelined CPU design the instruction access would run in parallel with internal decode/execute (and the data would only become available late in the memory access cycle) so a local "IFR" is introduced to capture and hold the instruction for a full cycle. This state is not "architectural" like the general purpose registers - but arises from micro-architectural features.

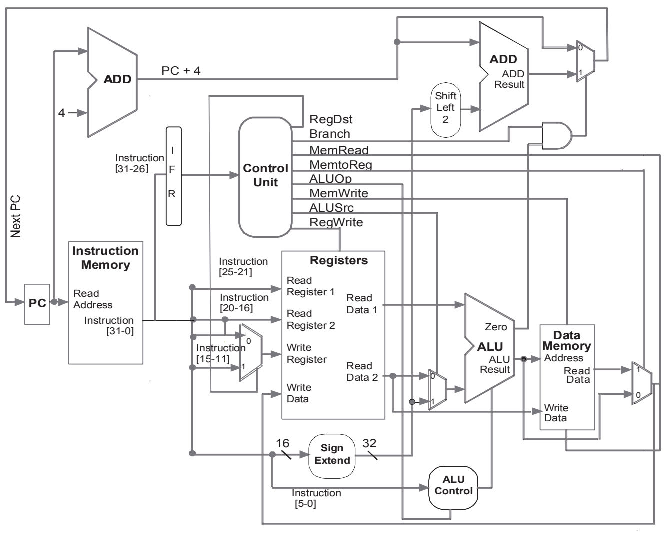

Fig. 4. 32 bit RISC Core showing programmer's visible state comprising the PC, the Instruction Memory, Register Banks and Data Memory. Retention registers are used for all these whilst the register IFR is designed using ordinary register.

Now we shall present the property that checks that the state of the control unit is adequately retained in presence of the 6-bit pipeline register IFR. This property is an instance of **Property II** shown in Section III-A. Due to lack of space we omit the instance for **Property I**. We shall begin by constructing assertions that describe the initial state of the instruction Memory, the memory read and the write signals, and the read and the write addresses. We will then write to the memory with a new data and would subsequently read it out as the Instruction stream. We will then show that this output Instruction stream from bits 31 downto 26 is preserved across the sleep and resume operations, and accordingly updates the next state of the IFR register on the rising edge of the clock post resume. For illustration, our Instruction Memory is 256 deep and 32 bits wide. It is initialised with a symbolic state by defining a trajectory formula IM that assigns symbolic values $\texttt{mem}_0,\ldots,\texttt{mem}_{255}$ to memory locations $\texttt{IMem}_0,\ldots,\texttt{IMem}_{255}$ respectively, between time 0 and 1. We define the trajectory formula WData that states that the Write port of the memory takes on a symbolic 32-bit vector WD vector from time 0 to 1.

$$\texttt{let WData} = \text{"}\texttt{WriteData}[31:0]\text{"}\ \texttt{is WD from 0 to 1}$$

Similarly we assume the existence of symbolic BDD variables (RA) and (WA) for ReadAddress and WriteAddress ports respectively. We define the assertion WAdd that the WriteAddress port takes on the symbolic WA value.

$$\texttt{let WAdd} = \text{"}\texttt{WriteAdd}[7{:}0]\text{"}\ \texttt{is WA from 0 to 1}$$

For the ReadAddess port a similar assertion RAdd is defined. We define that Memory Write (MemWrite) is asserted between 0 and 1 and de-asserted afterwards. MemRead is shown below.

$$\begin{aligned}\texttt{let MemRead} = {} & \text{"}\texttt{MemRead}\text{"}\ \texttt{is F from 0 to 2 and}\\ & \text{"}\texttt{MemRead}\text{"}\ \texttt{is T from 2 to 6 and}\\ & \text{"}\texttt{MemRead}\text{"}\ \texttt{is F from 6 to 9 and}\\ & \text{"}\texttt{MemRead}\text{"}\ \texttt{is T from 9 to 10}\end{aligned}$$

The Boolean expression that expresses the condition under which the memory is read after write is given by the RAW below. This function states the fact that if we write to a location i and read it back from the same location we should get the new data else the old content. $\texttt{Zero}, \texttt{One}, \ldots, \texttt{TwoFiftyFive}$ are 32-bit scalar (over 0s and 1s) representations of addresses.

$$\begin{aligned}&\texttt{let RAW} =\\ &(\texttt{RA} = \texttt{Zero}) \to ((\texttt{we}\ \wedge\ (\texttt{WA} = \texttt{Zero})) \to \texttt{WD} \mid \texttt{mem}_0)\\ &\mid (\texttt{RA} = \texttt{One})\ \to ((\texttt{we}\ \wedge\ (\texttt{WA} = \texttt{One}))\ \to \texttt{WD} \mid \texttt{mem}_1)\\ &\qquad\vdots\\ &\mid (\texttt{RA} = \texttt{TwoFiftyFive}) \to\\ &\quad((\texttt{we}\ \wedge\ (\texttt{WA} = \texttt{TwoFiftyFive}))\quad \to \texttt{WD} \mid \texttt{mem}_{255})\end{aligned}$$

We now show the property that captures our intention of reading the Instruction Memory correctly across the pipeline register IFR in presence of sleep and resume modes of the CPU. The formulas that describe the symbolic Read Address and Write Address are denoted by RAdd and WAdd respectively.

$$\begin{aligned}\texttt{let A} = {} & \texttt{WAdd and RAdd and MemWrite and MemRead and}\\ & \texttt{WData and IM and (}\text{"}\texttt{NRST}\text{"}\ \texttt{is T from 0 to 6) in}\end{aligned}$$

$$\begin{aligned}\texttt{let clock} = {} & (\text{"}\texttt{clock}\text{"}\ \texttt{is F from 0 to 1) and}\\ & (\text{"}\texttt{clock}\text{"}\ \texttt{is T from 1 to 2) and}\\ & (\text{"}\texttt{clock}\text{"}\ \texttt{is F from 2 to 3) and}\\ & (\text{"}\texttt{clock}\text{"}\ \texttt{is T from 3 to 4) in}\end{aligned}$$

$$\begin{aligned}\texttt{let Sleep} = {} & (\text{"}\texttt{clock}\text{"}\ \texttt{is F from 4 to 9) and}\\ & (\text{"}\texttt{NRET}\text{"}\ \texttt{is T from 0 to 5) and}\end{aligned}$$

```
                    (“NRET” is F from 5 to 8) and
                    (“NRST” is F from 6 to 7) in
let Resume = (“NRST” is T from 7 to 10) and
                    (“NRET” is T from 8 to 10) and
                    (“clock” is T from 9 to 10) and
                    (“clock” is F from 10 to 11) in
let C = (“IFR_Instr[31:26]” is RAW from 3 to 6) and
          (“IFR_Instr[31:26]” is [F,F,F,F,F,F] from 6 to 9)
            and (“IFR_Instr[31:26]” is RAW from 9 to 10)
 in (clock and Sleep and Resume and A) ⇒ C
```

This property shown above also checks that the Instruction Memory itself is designed correctly enuring the read and write work correctly. Thus we are able to preserve the state of the Control using a pipelined register IFR. Because IFR is only a 6-bit register it would not matter much if this was also implemented using retention registers for our study, but if this is to be done for all the different pipelined stages for example, in a multi-stage pipelined CPU it would have an extremely detrimental effect on the increase in area and drop in performance.

Using model checking, the most complex properties to check are the ones that involve parts of the data path with state holding elements [12]. However, using a combination of property decomposition [9] and symbolic indexing [13] we are able to cut down on verification time and the size of BDDs. For example, the use of symbolic indexing reduces the linear time and space complexity of symbolically checking SRAMS, to logarithmic. It took us 10.83 seconds to check the above property on an Intel Centrino 1.7 Ghz machine with 2 GB RAM running Linux in a virtual machine. This was the maximum time taken to check any property amongst the ones we checked for the CPU.

The key benefit of our approach is that it scales very well with the size of the CPU, primarily due to the fact that we extensively employ property decomposition techniques using STE inference rules [9]. For example, verifying a pipelined CPU would involve the decomposition of the properties that describe the functionality of the whole data path into several smaller properties across each pipelined stage, which in turn can be checked using model checker. Detailed expositions demonstrating how STE inference rules are used in practice exist in [7, 9].

IV. Conclusion

We have presented a case study based on symbolic simulation and model checking, to design and implement selective retention safely. The main finding of our study is that the programmer visible state or the architectural state of the CPU needs to be implemented using retention registers whilst other micro-architectural enhancements such as pipeline registers, TLBs and caches can be implemented using normal registers without retention. This has a profound impact on power and area savings for chip design. For a 3-stage, 5-stage and 7-stage CPU the programmers visible “architectural state” is basically the same but the micro-architectural state roughly doubles every generation as more complex write buffering, branch prediction and address translation/virtual memory structures grow to improve the CPU performance. Only implementing hardware state retention for the programmers model is highly desirable given that retention registers may be 25-40% larger area per flop.

In our experience implementing selective state retention without the support of rigourous formal analysis, is somewhat ad-hoc. One needs to do a careful analysis to identify a micro-architectural state that is definitely not required after sleep states. For example, a CPU entering a wait-for-interrupt sleep state, would typically have to empty the write buffer of any pending write transactions. Therefore, by design, though it is possible to not implement retention on this write buffer subsystem, and simply reinitialize state on wake-up - and check this with standard event simulation, in practice such an approach is not scalable to generic non architectural CPU state. Using symbolic simulation, we can apply our methods to any generic pipelined CPUs.

To the best of our knowledge this is the first study in the area of rigourous design and implementation of selective state retention.

References

[1] S. Shigematsu, S. Mutoh, Y. Matsuya, Y. Tanabe, and J. Yamada, “A 1-V High-Speed MTCMOS Circuit Scheme for Power-Down Application circuits,” *IEEE Journal of Solid State Circuits*, vol. 32, pp. 861–869, 1997.

[2] Michael Keating, David Flynn, Rob Aitken, and Alan Gibbons, *Low Power Methodology Manual*, 1st ed. Springer, 2007.

[3] V. Zyuban and S. V. Kosonocky, “Low Power Integrated Scan-Retention Mechanism,” in *Proceedings of ISLPED 2002*. ACM, 2002, pp. 98–102.

[4] “Accellera UPF Standard Version 1.0,” February 2007.

[5] Paul Hoxey and Clayton McDonald, and David Guinther, “An Introduction to Symbolic Simulation,” published at http://www.eetimes.com/, Sep 2005.

[6] Allan Crone and Gabriel Chidolue, “Functional Verification of Low Power Designs at RTL,” in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*. Springer, 2007, pp. 288–299.

[7] 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 TCAD*, vol. 24, no. 9, pp. 1381–1405, 2005.

[8] C.-J. H. Seger and R. E. Bryant, “Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories,” *Journal of FMSD*, vol. 6, no. 2, pp. 147–189, 1995.

[9] S. Hazelhurst and C.-J. H. Seger, “A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDDs,” *IEEE Tran. on CAD of Integrated Circuits*, vol. 14, no. 4, pp. 413–422, 1995.

[10] James O. Hamblen and Michael Furman, *Rapid Prototyping of Digital Systems: A Tutorial Approach*, 2nd ed. Springer, 2001.

[11] “The Forte Formal Verification System,” available from http://www.intel.com/.

[12] B. Siarkowski, “Memory Overwhelms Current Verification Techniques,” published at http://www.eetimes.com/, April 2003.

[13] M. Pandey, R. Raimi, R. E. Bryant, and M. S. Abadir, “Formal Verification of Content Addressable Memories Using Symbolic Trajectory Evaluation,” in *Proceedings of DAC 1997*. ACM Press, New York, NY, USA, pp. 167–172.