Chapter 1
Introduction

Validation is an important task in the processor design cycle. The validation phase is considered to be a major bottleneck because of the progressive increase in processor design complexity and heavy time-to-market constraints. There have been continuous efforts by researchers to make this phase *faster* and more *complete*. The processor design cycle goes through several phases before reaching the bit-accurate level with the granularity and design complexity increasing with every phase of the design. The design space exploration starts with an initial design of the instruction set formally known as the ISA (Instruction Set Architecture). A host of software tools like compilers, assemblers, linkers, loaders and simulators are required in this phase for architecture simulation and evaluation. The ISA design is followed by the implementation of a functional simulator which is goldenized using various random and directed testing methodologies. Typically, a set of benchmarks are run on the functional simulator to evaluate the architecture in this phase. This gives way to the pipeline design phase where an efficient pipeline architecture for the ISA is designed. Other than the complex pipeline and various functional units designed in this phase, methodologies like register renaming, score boarding, out-of-order execution, reservation stations, branch prediction, speculative execution and various advanced pipelining techniques are incorporated into the processor design in this phase. These additional features enhance the performance of the processor by improving the throughput in terms of instructions per cycle. The executable of the design in the pipeline design phase is built in the form of a cycle accurate pipeline simulator which is used for architectural simulation and benchmarking at this stage.

One of the biggest challenges in the architecture design phase is to verify the correctness of the pipelined design with respect to the ISA. The process of architecture exploration is iterative, where the iteration continues till the performance requirements of the design are met. Every iteration of the exploration phase results in new features and components incorporated into the architecture which improve performance of the design. After every iteration which results in a change in the

detailed architecture, it is necessary on the part of the designer to check whether the semantics of the pipelined execution is consistent with the semantics of the functional (ISA) specifications. Along with the additional functionality which is introduced into the pipeline, timing behavior of the various instructions and their interaction with other units/environment is also modeled in the cycle-accurate pipeline simulator. This adds to the complexity of the pipeline phase making the verification of the pipeline an even more difficult task.

There has been significant research in the domain of processor validation, using both simulation based verification and formal methods. Formal methods like Model checking [85] , SAT based methods [138] and their variants have been used in the domain of processor validation. Several formal approaches for checking the pipelined implementation with respect to the input ISA specification have been reported in the literature. These include methods based on term rewriting [23], model checking [28], theorem proving [53], finite state machine abstraction [126], and graph modeling [122]. While formal methods have been effective in relatively smaller designs, some attempts have been made to verify microprocessor pipeline implementations. For example, Levitt and Olukotun [109] used formal verification for pipeline implementation to verify a design against itself. There have been efforts to verify formally almost all the features of a pipeline, including register renaming [146], speculative execution [22], and Tomasulo's algorithm [118]. However, these methods require detailed specification of the architecture as input to the verifier.

Simulation Based Verification is still the most widely used validation methodology for processor designs. It is the state of the art verification methodology adopted for large scale industrial designs since available formal methods do not scale well enough for large state space based designs. Simulation Based Verification techniques have been proposed to validate special pipeline features like pipeline interlocks[58], rename-buffers[155], etc. There has been considerable research work in the domain of test generation of pipeline processors[15],[29],[121], [123],[124]. Aarti et al. [71] propose a coverage driven hybrid method to cut down on the number of tests. Wagner et al. [159] have presented a Markov-model driven random-test generator with activity monitors that provides assistance in locating hard-to-find corner-case design bugs and performance problems.

This thesis explores Simulation Based Verification techniques in the domain of pipeline processor validation. This chapter has been organized as follows. We present related work in Section 1.1 followed by motivation and objectives of this work in Section 1.2. We then present a summary of the contributions of this thesis in Section 1.3 followed by thesis organization in Section 1.4.

1.1. Related Work

In this section, we present related research work in the domain of simulation based verification. We classify the related work into the following categories *(i)* Comparison with ISA, *(ii)* Assertion Based Verification and *(iii)* Test Generation. We present the popular methodologies adopted along with the recent advances in the respective domains. Firstly, we present the simulation trace based validation techniques followed by assertion based verification techniques and finally end with test generation methodologies.

1.1.1. Comparison with ISA

Several formal and semi-formal approaches developed for verification of pipeline behavior have been reported in the literature. In the field of pipeline verification, one of the popular methods used is to show the equivalence of the ISA and the corresponding micro-architecture design. An ISA is a non-pipelined abstract machine which specifies the effects of individual instructions, while a micro-architectural design represents the pipeline architecture of the actual machine implementation. The equivalence between these levels is established by defining a commutative relationship with an abstraction function mapping a micro-architectural state to an ISA state. The methodologies used to prove this equivalence relationship comprise of formal methods like theorem proving, usage of logic and abstraction functions and Term Rewriting Systems. There are two broad categories into which the methodologies can be divided, namely:

Bottom Up - This is the more traditional approach in which the pipeline specification is extracted from an already existing pipelined implementation and is compared with the ISA for equivalence.

Top Down - The behavioral description of the pipeline specification is provided by the user and this is used for the purpose of verification.

Most of the reported methodologies except for the ADL driven methods are bottom up approaches. In the top down approach, the processor pipeline specification should satisfy a set of properties to ensure correct behavior. These properties ensure that the pipelined execution adheres to the correct semantics of sequential execution. We discuss some of the significant contributions in this domain in the following sections.

1.1.1.1. Use of Logic and Abstraction Functions

Burch and Dill [40] describe a technique for verifying the control logic of pipelined microprocessors. The verification process begins with the user providing behavioral descriptions of an implementation and a specification. In case of processor

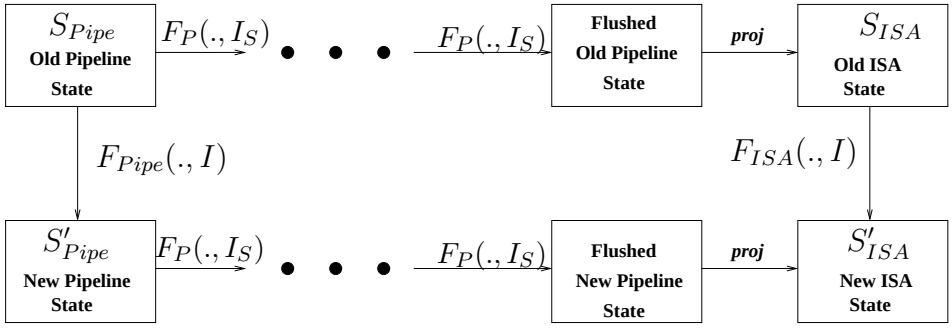

Figure 1.1: Correctness Criteria defined by Burch and Dill[40] where F_P stands for F_{Pipe} and I_S stands for I_{Stall}

verification, the specification describes how the programmer-visible parts of the processor state are updated when one instruction is executed every cycle. The implementation description is at the highest level of abstraction which reveals the pipelining details. The steps of the above mentioned methodology are as follows.

Each of the descriptions of the specification and implementation is automatically compiled into a *transition function*, which takes a state as its first argument, the current inputs as its second argument, and returns the next state. The transition function is encoded as a vector of symbolic expressions with one entry for each state variable. F_{ISA} and F_{Pipe} are specified to denote the transition function of the specification and the implementation, respectively. It is required that the implementation and the specification have corresponding input wires. They use a methodology called flushing to establish the equivalence between the specification and implementation, which is described next.

Almost all processors have an input setting that causes instructions already in the pipeline to continue execution while no new instructions are initiated, which is typically referred to as *stalling* the processor. If I_{Stall} is an input combination that causes the processor to stall, then the function $F_{Pipe}(., I_{Stall})$ represents the effect of stalling for one cycle. All instructions currently in the pipeline can be completed by stalling for a sufficient number of cycles. This operation is called *flushing* the pipeline, and it is an important part of the verification methodology.

The fundamental principle of correctness which the verifier tries to establish is that if the implementation and specification start in any matching pair of states, then the result of executing any instruction will lead to a matching pair of states. The execution of instructions in the ISA is sequential, where as there are multiple instructions executing simultaneously in the pipeline. This makes the problem of checking the equivalence between the ISA and the pipeline a difficult problem to solve. Finding a synchronization point where the states from both the descriptions can be compared is difficult to establish. The verifier circumvents this problem by simulating the effect of completing every instruction in the pipeline before doing

the comparison, which is achieved by *flushing* the pipeline.

This has been represented figuratively in Figure 1.1. The implementation can be in a arbitrary state S_{Pipe}, which has been labeled as "Old Pipeline State" in the Figure 1.1. To complete the partially executed instructions currently in the pipeline, in the state S_{Pipe}, the pipeline is flushed, producing "Flushed Old Pipeline State". The next step is to do a projection (represented by the edge labeled *proj*) to get a programmer's view of the implementation. This produces S_{ISA}, the "Old ISA state". The state S_{ISA} matches the pipeline state S_{Pipe}, as S_{Pipe} has been tranformed to S_{ISA} through a sequence of *stall* transitions. In the presence of an input I to the pipeline specification, the pipeline makes a transition marked by $F_{Pipe}(., I)$ from the present state to a new state called S'_{Pipe}, the "New Pipeline State". Similarly, the ISA specification will change state to S'_{ISA} through a transition labeled as $F_{Spec}(., I)$. The pipeline implementation is said to be equivalent to the ISA specification *iff* S'_{Pipe} matches S'_{ISA}. There are two paths from the S_{Pipe} to S'_{ISA}, one that goes through $F_{Spec}(., I)$ - the *specification side* and other takes the $F_{Pipe}(., I)$ - the *implementation side*. For both these paths, there is a corresponding function which is a composition of the various functions along the various edges on the path. The implementation is said to satisfy the specification iff the function corresponding to the implementation side of the diagram is equal to the function corresponding to the specification side of the diagram in Figure 1.1.

In this approach Burch and Dill[40] use quantifier-free logic of uninterpreted functions and predicates with equality and propositional connectives. Uninterpreted functions are used to represent combinational ALUs, for example, without detailing their functionality. Description and comparison of control in the specification and the implementation is done using propositional connectives and equality. Firstly, the operational descriptions of the specification and implementation are compiled, then a logical formula is constructed that is valid if and only if the implementation is correct with respect to the specification. The second phase is a decision procedure that checks whether the formula is valid. They show experimental results for a subset of the DLX processor.

Skakkebaek et al.[148] extend Burch and Dill's work [40] and present a two-part approach that deals with the out-of-order scheduling logic and the in-order buffering mechanisms separately. First, the implementation is modified to derive an in-order abstraction. These modifications bypass the out-of-order logic and result in instructions executing in order. By exploiting domain-specific knowledge, they are able to establish a functional equivalence relation between the out-of-order implementation and the abstraction. The second step of this technique shows that the in-order abstraction is functionally equivalent to the ISA. This is accomplished

via a technique introduced called *incremental flushing*[40], based on the Burch-Dill automatic flushing approach.

Velev and Bryant[158] extend Burch and Dill's flushing technique for formal verification of microprocessors to be applicable to designs where the functional units and memories have multicycle and possibly arbitrary latency. They also show ways to incorporate exceptions and branch prediction and use it to study the modeling of the above features in different versions of dual issue superscalar processors. Egon et al. [34] use a refinement methodology using sequential evolving algebras[73] to prove the equivalence of the sequential model with the pipeline model. In this section we discussed the application of logic and abstraction functions to establish the equivalence between the pipeline design and the ISA. We now discuss other methodologies like Theorem Proving, usage of Term Rewriting Systems and other methodologies adopted to establish this equivalence relationship.

1.1.1.2. Theorem Proving

Theorem proving, currently the most well-developed subfield of automated reasoning, is the proving of mathematical theorems by a computer program. Commercial application of theorem proving is mostly in the domain of integrated circuit design and verification. Since the famous *Pentium FDIV* bug in October 1994, extra care is taken to design the complicated floating point units of modern microprocessors. In the latest processors from AMD*, Intel†, and others, automated theorem proving has been used to verify that division and other operations are correct. Theorem proving techniques have been used extensively for verification of pipelines[51], [143], [149]. The methodology comprises of three steps: *(i)* Given the *specification* state machine, *(ii)* Given the *implementation* state machine, and *(iii)* Prove that the implementation machine *satisfies* the specification machine. Sawada and Hunt [143] verified the correctness of a micro-architectural specification of a pipeline processor with respect to the ISA-level specification by defining a table-based execution abstraction called MAETT (Micro-Architectural Execution Trace Table). They use MAETT to specify the various pipeline properties and incrementally prove that these properties are invariantly held by all micro-architectural states reachable from a flushed pipeline state and finally use this fact to prove the correctness criteria which says that the pipeline implements the ISA. They verify a simple out-of-order completion pipelined microprocessor design using the ACL2 theorem prover [1]. We briefly discuss the methodology here.

A MAETT M is an *unbounded* list of all instructions I_1, I_2, $\cdots$, I_n which

*Advanced Micro Devices, http://www.amd.com
†http://www.intel.com

are currently being executed or have already retired (finished execution) since the machine started.

$$M = (I_1, I_2, \cdots, I_n)$$

The ISA executes I_1, I_2, $\cdots$, I_n in that order. Each instruction I_i is represented by a record like:

$$I_i = (Flg_i, PC_i, Inst_i, RA_i, RB_i, RC_i, Stg_i, Regs_i, Mem_i, Misc_i)$$

where PC_i, $Inst_i$, RA_i, RB_i, RC_i, Stg_i, $Misc_i$ are respectively the instruction address; instruction word; operand register identifiers of RA, RB and RC; current stage of the instruction; and miscellaneous stage-dependent information. Flg_i is a flag indicating whether I_i is speculatively fetched. $Regs_i$ and Mem_i are the correct register file and memory states after completing all previous instructions I_1,$\cdots$,I_{i-1} and before executing I_i with the ISA. Let $\texttt{ISA-state}(PC, Reg, Mem)$ denote the ISA state with PC, Reg, Mem as its program counter, register file and memory, respectively. An "ideal" ISA state S_i corresponding to I_i is defined by:

$$S_i = \texttt{ISA-state}(PC_i, Reg_i, Mem_i)$$

The next ISA state S_{i+1} is related to S_i by:

$$S_{i+1} = \texttt{ISA-state-step}(S_i)$$

In this way a MAETT records an ISA execution sequence. At the same time, a MAETT also represents the micro-architectural state, by recording the current status of each instruction. They define a MAETT update function `MAETT-step()` to simulate the micro-architectural state transition `micro-state-step()`. If s is a micro-architectural state and M is its MAETT representation, `MAETT-step(`M, s`)` gives a MAETT representation of the next `micro-state-step(s)`. `MAETT-step` updates appropriate fields of the instruction records already in a MAETT.

Along with the MAETT representation, the user defines certain pipeline invariant properties of the pipeline implementation using the richness and regularity of MAETT representation. One such invariant property which they model is the WAW hazard free predicate "`no-WAW-hazards?`". The WAW hazard between the instructions I_k and I_l is defined as follows:

$$\begin{aligned}\texttt{WAW-violation?}(I_k, I_l) = (&\texttt{reg-writeback-inst?}(\texttt{entry-inst}(I_k))\\ &\wedge \texttt{reg-writeback-inst?}(\texttt{entry-inst}(I_l))\\ &\wedge \texttt{same-destination-reg?}(I_k, I_l)\\ &\wedge \texttt{out-of-order-retire?}(I_k, I_l))\end{aligned}$$

This definition assumes that I_k is an earlier instruction than I_l, with respect to the execution order in ISA. The definition says that a WAW hazard between I_k and I_l occurs when both of the instructions have to write-back to the same destination register and I_l retires earlier than I_k. Each of the predicates in the definition of `WAW-violation?` is further defined as a relation of the fields of I_k and I_l. Finally, they define a global WAW hazard free property `no-WAW-hazards?` as a recursive predicate which checks `WAW-violation?` for all pairs of instructions in MAETT M:

$$\texttt{no-WAW-hazard?}(M) = \forall I_k, I_l \in M \text{ s.t. } I_k <_M I_l, \neg\, \texttt{WAW-violation}(I_k,I_l),$$

where $I_k <_M I_l$ means that I_k appears in M earlier than I_l. Similarly, the other global invariants like `no-structural-hazard?`, `in-order-issue?` are defined which are essential for exhibiting correct functionality.

Srivas and Bickford [149], have used Clio[30],[32] a functional language based verification system, to prove the correctness of a large-scale, realistic microprocessor design architecture. Clio verifies properties of programs written in Caliban[30] a polymorphic, strongly typed, lazy functional language. The properties to be proved are expressed in the Clio assertion language and proved interactively with the Clio theorem prover. The microprocessor, Mini Cayuga, used is a three-stage instruction pipelined RISC processor with a single interrupt. Mini Cayuga is a scaled down version of a RISC processor, Cayuga [31]. In all of these cases, either the processor was extremely simple or a large amount of labor in the form of manual intervention was required.

Kumar and Tahar [103] present a generic technique for the formal verification of pipeline conflicts in RISC cores. They model the different kinds of pipeline conflicts formally and present automated proof techniques for each kind of conflict which is implemented within the HOL verification system [69]. When conflicts are detected during the proof process, the conditions under which these occur are generated, thus aiding a designer in the debugging of these conflicts. They demonstrate the efficacy of their approach on a DLX RISC processor. The details of their implementation of the interpreter model specifications and the proof methodology is reported in [152]. Theorem proving may not require manual intervention at every step, but requires the detailed information regarding the specification and implementation model.

1.1.1.3. Term Rewriting Systems

A term rewriting system(TRS) is defined as a tuple (S, R, S_0), where S is a set of terms, R is a set of rewriting rules, and S_0 is a set of initial terms ($S_0 \subseteq S$). The state of a system is represented as a TRS term, while the state transitions are represented as TRS rules. The general structure of rewriting rules is:

s_1 if $p(s_1)$
$\rightarrow s_2$

where s_1 and s_2 are terms, and p is a predicate. We can use a rule to rewrite a term if the rule's left-hand-side pattern matches the term or one of its subterms and the corresponding predicate is true. The new term is generated in accordance with the rule's right-hand side. If several rules apply, then any one of them can be applied. If no rule applies, then the term cannot be rewritten any further.

Term rewriting systems (TRSs)[27],[93] are used to conveniently specify parallel and asynchronous systems and prove the correctness of an implementation with respect to a given specification. Arvind and Shen [23] use TRS to describe the operational semantics of a single-cycle, non-pipelined, in-order execution processor and the speculative processor capable of register renaming and out-of-order execution. They take a minimalist RISC instruction set, where they specify the semantics of the instruction as a rewrite rule, specifying how the state is modified after each instruction executes. They define the operational semantics of instructions using a single-cycle, non-pipelined, in-order execution processor model called P_B, and the speculative and out-of-order processor model P_S.

The operational semantics of the single-cycle non-pipelined processor model is defined using TRS as follows. The processor comprises of a program counter (pc), a register file (rf), and an instruction memory (im). The TRS term corresponding to the processor is defined as *Proc(pc,rf,im)*. The processor with the data memory(dm) form the complete system which is succinctly represented by the following TRS term:

Sys(Proc(pc,rf,im),dm)

The semantics of each instruction is specified as a rewrite rule specifying how the state is modified after each instruction executes. Two of the instructions supported by the AX instruction set are *Jz* and *Store*. The branch instruction $Jz(r_1, r_2)$ sets the program counter to the target instruction address specified by register r_2 if register r_1 contains value zero; otherwise the program counter is simply incremented by one. The store instruction $Store(r_1, r_2)$ writes the content of register r_2 into the memory cell specified by register r_1. The semantics of a *Jz*(jump on zero) and *Store* instruction is specified as follows:

Jz-Jump rule
Proc(ia, rf, im) if im[ia] = Jz(r_1, r_2) and rf [r_1] = 0
$\rightarrow$ Proc(rf [r_2], rf, im)
Jz-NoJump rule
Proc(ia, rf, im) if im[ia] = Jz(r_1, r_2) and rf [r_1] $\neq$ 0

```
Store rule
Sys(Proc(ia, rf, im), dm) if im[ia] = Store(r1, r2)
→ Sys(Proc(ia+1, rf, im), dm[a := rf [r2]]) where a = rf [r1]
```

In this description, *ia* represents the Instruction address(PC), r_1 and r_2 are register indexes, and *dm[a]* refers to the content of memory location a, and *dm[a := v]* represents the memory with location a updated with value v.

The speculative out-of-order pipeline model has two extra components which involves (i) A branch translation buffer - BTB and (ii) A reorder buffer - ROB. The speculative instruction's address is determined by consulting the branch target buffer (BTB) and ROB is a buffer which holds instructions that have been decoded but have not completed execution. An instruction template buffer (*itb*) in *rob*(the ROB in the specification) contains the instruction address, opcode, operands, and some extra information needed to complete the instruction. For instructions that need to update a register, the *Wr(r)* field records destination register r. For branch instructions, the *Sp(pia)* field holds the predicted instruction address *pia*, which will be used to determine the prediction's correctness. As an example of instruction fetch rules, consider the *Fetch-Op* rule, which fetches an *Op* instruction and after register renaming simply puts it at the end of the rob as follows:

```
Fetch-Op rule
Proc(ia, rf, rob, btb, im) if im[ia] = r := Op(r1, r2)
→ Proc(ia+1, rf,rob ⊕ Itb(ia, t := Op(tv1, tv2), Wr(r)), btb,
   im)
```

where t represents an unused tag, and tv_1 and tv_2 represent the tag or value corresponding to the operand registers r_1 and r_2, respectively. $\oplus$ represents the associative operator, which states that the *Op* instruction is added to the reorder buffer(*rob*).

They show that the speculative processor produces the same set of behaviors as a simple non-pipelined (sequential) implementation. They prove that the speculative processor is a correct implementation of the instruction set by showing that P_B and P_S can simulate each other with respect to some observable property, which in this case is the programmer visible state including the program counter, the register file, and the memory from the system. If model A can simulate model B, then for any program, model A should be able to print whatever model B prints during execution.

In [79], the authors verify a simple pipelined microprocessor in Maude, by implementing an equational theoretical model of systems. Maude is an equationally-based language, with an efficient term rewriting implementation, and effective meta-level tools. Velev[157] combines rewriting rules and Positive Equality [38]

automatically and use it to formally verify out-of-order processors that have a reorder Buffer, and can issue/retire multiple instructions per clock cycle. The verification is based on the Burch and Dill correctness criterion [40]. Rewriting rules are used to prove the correct execution of instructions that are initially in the Reorder Buffer, and to remove them from the correctness formula. Positive Equality is then employed to prove the correct execution of newly fetched instructions. The formal verification is done by correspondence checking comparison of an implementation pipelined processor against a nonpipelined specification (the Instruction Set Architecture), based on the Burch and Dill[40] commutative property. The methodologies based on TRS also require detailed pipeline specification similar to the Theorem Proving counterpart.

In the last section we discussed the application of TRS for proving ISA equivalence, we present the other FSM and ADL driven methodologies in the following sections.

1.1.1.4. ADL Driven Pipeline Validation

Most of the approaches used for pipeline verification are bottom up methods[80],[86]. A bottom up approach involves abstracting the pipeline behavior from an existing implementation and comparing it with the ISA. Hauke et al. [80] compare extracted ISA level description with the given ISA level specification. Architecture Description Languages (ADLs) are widely being used today for description of processor architectures. Recent work on language-driven Design Space Exploration (DSE) ([77], [66], [76]) uses Architectural Description Languages (ADL) to capture the processor and memory architecture, generate automatically a software toolkit (including compiler, simulator, assembler) for that architecture, and provides feedback to the designer on the quality of the architecture. Mishra et al. [128] present a graph-based modeling of architectures that captures both the structure and the behavior of the processor, memory and co-processor pipelines. Based on the model, they proposed several properties that need to be satisfied to ensure that the architecture is well-formed. The graph model of the architecture is generated automatically from this ADL description and has information regarding architecture's structure, behavior, and the mapping between them. They apply these properties on the graph model of the MIPS R10K, TIC 6x, ARM, DLX, and PowerPC architectures to demonstrate the usefulness of their approach.

Mishra et al. [126] propose a approach which leverages the system architect's knowledge about the behavior of the pipelined processor, through Architecture Description Language (ADL) constructs, thereby allowing a powerful top-down approach to pipeline verification.

1.1.1.5. FSM Based Methods

Tomiyama et al. [153] presented FSM based modeling of pipelined processors with in-order execution. Their model can handle only simple processors with straight pipelines. Based on the modeling, they presented a set of properties which are used to verify the correctness of in-order execution in the pipeline. If a given pipelined processor satisfies all the properties, its pipeline behavior is guaranteed to be correct. Mishra et al. [126] extend this model to handle more realistic processor features such as fragmented pipelines and multicycle functional units leveraging the system architect's knowledge of the pipeline behavior through Architecture Description Language (ADL) constructs. Iwashita et al.[89] and Ur and Yadin [154] presented work where they model the pipelined processor using FSM. The difference is that they used the FSM model to automatically generate test programs for simulation based validation of the processors.

1.1.1.6. Other Methodologies

Mark Aagaard [13] presented a design framework for pipelines with structural hazards. He created a framework with four parameters that are used to characterize and verify pipelines. They are as follows:

Protocol schemes describe how transactions are transferred between stages in the pipeline.

Arbitration schemes specify how to prevent and or handle structural hazards in the pipeline.

Control schemes determine how transactions are routed through the pipeline and how stages know what operation to perform.

Ordering schemes describe a method for matching up transactions as they leave the pipeline with transactions that entered the pipeline.

In order to simplify the pipelining proof their framework which has four parameters that are used to characterize pipelines. Each parameter has an associated specification. They have proved that pipelines that conform to these specifications obey the pipelining correctness criteria.

Mark Agaard et al.[12] use combinational equivalence verification with the modular and composable verification strategy of completion functions and apply this technique to verify VHDL implementations of a 4-stage, in-order, 32-bit OpenRISC[136] processor implementing 47 instructions. In this work, the task of the verification engineer was to write one completion function for each in-flight instruction in the pipeline. The completion function for a given stage describes the

effect on the architectural state of completing the partially executed instruction in the stage. Executing a completion function for a stage has essentially the same effect as flushing the instruction in the stage. Executing all completion functions flushes the entire pipeline. The end result of verification with completion functions is the same as verification by flushing.

Levitt and Olukotun [108] present an automatic formal verification technique to verify the pipeline control logic called *unpipelining* specifically developed to tackle the complexity due to pipelining. It eliminates the pipeline stages from an implementation while preserving the implementation's behavior, collapsing it into a single stage through a sequence of transformations. The added complexity due to pipelining is completely alleviated and the transformed and deconstructed pipeline can be compared directly to the ISA specification. They present an inductive proof methodology that verifies that pipeline behaviour is preserved as the pipeline depth is reduced via deconstruction. They claim that this inductive approach is less sensitive to pipeline depth and complexity than previous approaches. They present experimental results from the formal verification of a DLX five-stage pipeline using this technique. Levitt and Olukotun [109] extend *unpipelining*, making it more general, more automatic and easier to use. The key benefits of unpipelining are: it verifies a design against itself, thus removing the need for an external specification; it can be used within the framework of hierarchical verification; and it is scalable to longer, more complex pipelines. However, *unpipelining* has its limitations. Although more automatic and scalable than other methods, *unpipelining* is less general in its applicability. It is not suitable for verifying pipelines that implement instruction set architectures (ISAs) which use pipeline features such as delayed loads and branches. Also, *unpipelining* relies on the use of a standard design style in order to deconstruct a pipeline.

Manolios and Srinivasan [116] describe an approach to bit-level pipelined machine verification that uses the tool UCLID[39] to reason about term-level models and the theorem prover ACL2[92] to relate the term-level models to bit-level models and to establish the correctness of the various abstractions used in the term-level models. They use a refinement based proof methodology to show that an executable complex pipelined machine model, mostly defined at the bit-level, refines its instruction set architecture. The authors [115] introduce compositional proof rules that guarantee that this sequence of refinement proofs implies that the final pipelined machine has the same behaviors as the instruction set architecture.

Mishra et al.[131] propose a top-down processor validation approach using symbolic simulation. Symbolic simulation has proved to be an efficient technique, bridging the gap between traditional simulation and full-fledged formal verification. They define a set of properties and verify the correctness of the processor

by verifying if the properties are met. They applied their methodology to verify several properties on a Memory Management Unit (MMU) of a microprocessor that is compliant with the PowerPC instruction-set architecture.

It is evident from the presented related work that comparison with the ISA is a popular method to ensure pipeline correctness. One of the major drawbacks is that the approaches require a detailed pipeline specification which is not always available. We now discuss the related work in the domain of assertion based verification.

1.1.2. Assertion Based Verification

Assertions specify the correctness criterion of a design in terms of pre-conditions and post-conditions. Assertion Based Verification(hence forth refereed to as **ABV**) involves writing the correctness criteria in terms of assertions and using them to verify the correctness of a design. Assertions are employed for validation in both static and dynamic methods. Assertions are represented in some form of specification logic (which is often *temporal*) also known as formal properties which have to be verified statically in Model Checking and other Formal Property Verification (FPV) methods. Purely formal methods of verification face capacity issues and purely simulation based verification face coverage related problems. Assertion Based Verification tries to take the best from both of these worlds, where the assertions are specified in some form of temporal logic and uses them to validate a simulation trace.

1.1.2.1. Benefits of ABV

Designing at RTL level is the most popular level methodology adopted in industry. The assertions are concurrently written with the RTL design and they are closely coupled with the RTL code. There are significant benefits of incorporating assertions in the digital design and verification process. The primary benefit is that assertions help to detect functional bugs earlier in the process and enable efficient fault location as the assertions are tightly coupled. This in turn leads to fewer bugs propagating in the production process resulting in shorter verification time and faster debugging.

A secondary benefit is that the very act of formulating and writing assertions can give the designer a better understanding of the design. This in turn aids in uncovering bugs in the specification and avoiding introduction of bugs into the design in the first place. Once written, bound to the RTL code and proven, the assertions play the role of formal comments. Unlike conventional comments written informally, assertions are executable and continue to monitor the design for functional correctness through all levels of abstraction of the design.

Assertions assist the designer in formally defining a coverage metric to the verification process. Assertions increase the observability of the state of the design during verification. The coverage of the assertions provide a measure of the percentage of completion of the verification process.

Adopting an ABV methodology have led to significant reduction in simulation debugging time (as much as 50% in Industry) due to improved observability [14]. Moreover, a framework which supports ABV is also adaptive to advanced verification techniques which improves the over all verification quality [65]. ABV facilitates the reuse as the assertions created for block-level verification are also valid and can be used at the chip-level [42]. We now discuss some of the languages used for the specification of assertions.

1.1.3. Languages for Assertion Based Verification

The main challenge here is to express key features of the design in terms of formal properties. Formal properties are typically expressed by two different classes of temporal logics, namely branching time logic and linear temporal logics. Branching time logics allow the specification of properties over the computation tree created by the state traversal of the state machine. The basic notion is to unfold the state machine into an infinite tree where each path of the tree is a run or a computation of the state machine. Linear time logics allow the specification of properties over linear traces or runs of a finite state machine - intuitively, the property holds on the machine if it holds on all runs of the machine.

Designers and verification engineers typically express and interpret the RTL in terms of simulation semantics of the HDL. They are accustomed to verifying the correctness of the RTL by checking certain behaviors over simulation traces. Therefore, linear temporal logics have been the natural choice for design validation, and the backbone of most exciting property specification languages, including PSL (Property Specification Language) [3] and SVA (SystemVerilog Assertions) [5].

In this thesis, we propose a specification logic for succinct representation and simulation based verification of data oriented specifications. We also provide a comparison of the logic with LTL and SVA in terms of expressive power. The following sections introduce LTL and SVA with examples. Most of the examples and definitions in this section have been taken from the book [55].

1.1.3.1. LTL - Linear Temporal Logic

The formal syntax and semantics of LTL is defined over a Kripke structure. Formally, we define a Kripke structure as a tuple, $K = \langle AP, S, \tau, s_0, F \rangle$, where:

- S is a finite set of states,
- AP is a set of atomic propositions labeling of each state $s \in S$,

- $\tau \subseteq S$ x S is the transition relation, which must be total(for all states $s_i \in S$, there exists a state $s_j \in S$ such that $(s_i, s_j) \in \tau$),
- $s_0 \subseteq S$ is the set of start states,
- $F : S \rightarrow 2^{AP}$ is a labeling of states with atomic propositions true in that state.

A *path* (alternatively a run or a trace), π, in the Kripke structure is an infinite sequence of states, $s_0, s_1, \cdots$, such that for all i, $s_i \in S$, and $(s_i, s_{i+1}) \in \tau$. s_0 is called the starting state of π. We use π^j to denote the suffix of π starting from s_j.

The formal syntax of LTL can be recursively defined as follows:

- Each atomic proposition in AP is an LTL formula.
- if f and g are LTL formulas, then so are $\neg f$, $f \wedge g$, $X\ f$, $f\ U\ g$, $G\ f$, and $F\ g$.

The Boolean operators have their usual meaning. There are four temporal operators in LTL, these are X (next time), U (until), G(globally), and F (eventually).

Given a run(path) π, we will also use the notation $s_k \models f$ to denote $\pi^k \models f$. In other words, a property (say f) is said to be true at an intermediate state of the run iff the fragment of the run starting from that state satisfies the property.

Let f and g be LTL formulas; p is an atomic proposition and $\pi = s_0, s_1, \cdots$ is a path in a Kripke structure K. Then the formal semantics of LTL may be defined as follows:

- $\pi \models p$ *iff* $p \in F(s_0)$
- $\pi \models \neg f$ *iff* $\pi \not\models f$
- $\pi \models f \wedge g$ *iff* $\pi \models f$ and $\pi \models g$
- $\pi \models X(f)$ iff $\pi^1 \models f$
- $\pi \models f\ U\ g$ iff $\exists\ j$, such that $\pi^j \models g$ and $\forall\ i\ i < j$, we have $\pi^i \models f$

$F\ g$ is a short form for TRUE$U\ g$, and $G\ f$ is a short term for $\neg F\ \neg f$. We say that the property f holds on a state machine J, iff f holds on all paths of the state machine starting from its start state. To illustrate further, we provide an example that illustrates the basic operators.

Example: Consider the specification of an arbiter that arbitrates between two master devices. Let the request and grant lines of the two master devices be r_1, g_1 and r_2, g_2 respectively. The arbiter obeys the following specification.

- P_1:If master 1 requests (r_1 is asserted), the arbiter must give the grant g_1 in the next cycle.
- P_2: If anytime the arbiter finds that r_1 is de-asserted, it (by default) parks the grant on master 2 by asserting g_2 in the next cycle.
- P_3: At any time, only one master should be granted.

These requirements are expressed in LTL as follows:

$$P_1 : G(r_1 \Rightarrow X(g_1))$$
$$P_2 : G((\neg r_1) \Rightarrow X(g_2))$$
$$P_3 : G((\neg g_1) \vee (\neg g_2))$$

1.1.3.2. Applications and Extensions

LTL and its variants are used for specification of assertions. These assertions are either used for formal verification or they are used for synthesize monitors which assert those properties over a simulation trace either *offline* which is *post simulation* and *online* which is *at par* with simulation. Apart from verifying formally and by simulation, assertions have also been synthesized to built hardware monitors and to synthesize behavioral models which behave according to the assertion specification [62]. A finite state extension of LTL known as FLTL, which defined the semantics of LTL for finite simulation trace as defined by Hoffman et al.[88]. They extend the semantics of LTL for finite simulation trace which involves redefining the unbounded operators in bounded form. A deterministic finite state automata is synthesized from the FLTL specification and the FLTL assertions are checked on-the-fly with the simulation. LTL has been used in tools such as Temporal Rover [59] and Java Pathexplorer [82].

1.1.3.3. System Verilog Assertions

This section outlines the structure of SVA properties. The building blocks of SVA properties are called *Sequence Expressions(SE)*, that are used to describe the temporal behavior of the system. The most basic sequence expressions are the signals and Boolean expressions over the signals. Temporal sequence expressions can be constructed using the *time range* operators. The syntax of a sequence expression is defined as follows. *SE* denotes a sequence expression over a set of atomic propositions AP.

- SE → SE TIME_RANGE SE | SEQUENCE_ABBRV | EXP BOOLEAN_ABBRV | SE SEQUENCE_OP SE | first_match(SE) | EXP throughout SE | EXP |(SE)
- TIME_RANGE → ##k |##[k_1 : k_2]

- SEQUENCE_ABBRV → [*k] | [*k_1:k_2]
- BOOLEAN_ABBRV → [*k] | [*k_1:k_2] | [*> k_1:k_2] | [*= k_1:k_2]
- SEQUENCE_OP → and | or | intersect |within
- EXP → (EXP || EXP) | (EXP && EXP) | !EXP | p, where p ∈ AP| ϵ

The structure of a SVA property is as follows:

PROPERTY → **property** PROP_EXP **endproperty**

where PROP_EXP can be of two types, namely:
[CLOCK_EVENT] [**disable iff** EXP][**not**] SE
→ [**disable iff** EXP][**not**] SE or,
[CLOCK_EVENT] [**disable iff** EXP][**not**] SE
⇒ [**disable iff** EXP][**not**] SE

In the above forms:

- **CLOCK_EVENT** represents the name of the clock against which the property is evaluated.
- **disable iff** EXP allows the user to specify asynchronous reset. if the EXP becomes true then the evaluation stops and the property is accepted as true.
- Having a not operator before a sequence expression s implies that whenever s matches, not s fails, and vice-versa.
- | ⇒ and → are the implication operators. A property is said to match iff for every match of the antecedent part there is a corresponding match of the consequent part. The start match of the consequent part may start at the same time stamp at which the antecedent matches or one time step later, depending on the implication operator being | ⇒ or → respectively. The property matches vacuously if the antecedent fails.

A property defines a behavior of a design. A property can be used for verification as an assumption, a checker, or a coverage specification. In order to use the behavior for verification, an assert, assume or cover statement must be used.

The assert statement is used to enforce a property as a checker. When the property for the assert statement is evaluated to be true, the pass statements of the action block are executed. Otherwise, the fail statements of the action block are executed. For example, “property abc" has been declared as a checker.

property abc(a,b,c);

disable iff (a==2) **not @clk** (b ##1 c);

endproperty

env_prop: **assert property** (abc(rst,in1,in2)) pass_stat **else** fail_stat;

In this section, we introduced SVA and indicated how it is used for specification of desired behavior of systems. In this thesis we are interested in presenting a logic where one of the objectives is to allow the user to model data-oriented behavior. SVA also allows reasoning about data dependencies which we describe next.

1.1.3.4. Data-Orientation in SVA

Data dependencies also form a part of design intent which need to be specified and checked over a simulation trace or verified formally. SVA incorporates local variables in order to manipulate data in a sequence and allows correlating the data values when the end of the sequence is reached. The local variables are dynamically created when needed within an instance of a sequence and removed when the end of the sequence is reached. The dynamic creation of a variable and its assignment is achieved by using the local variable declaration in a sequence or property declaration and making an assignment in the sequence. In brief, an SVA local variable provides the benefit of sampling and manipulating data in a property or sequence without requiring the property writer to define auxiliary state machines to model the intended behavior.

As an example given in SVA Language Reference Manual [5], if in the given SVA sequence

```
a ##1 b[->1] ##1 c[*2]
```

it is desired to assign `x = e` at the match of `b[->1]`, the sequence can be rewritten as

```
a ##1 (b[->1], x = e) ##1 c[*2]
```

The local variable can be reassigned later in the sequence, as in

```
a ##1 (b[->1], x = e) ##1 (c[*2], x = x + 1)
```

For every attempt, a new copy of the variable is created for the sequence. The variable value can be tested like any other *SystemVerilog* variable.

We present a detailed example from SVA LRM [5] which will demonstrate the usability of local variables in SVA for handling data-orientation. Assume a pipeline that has a fixed latency of `5` clock cycles. The data enters the pipe on `pipe_in` when `valid_in` is *true*, and the value computed by the pipeline appears

5 clock cycles later on the signal pipe_out1. The data as transformed by the pipe is predicted by a function that increments the data. The following property verifies this behavior:

```
property e;
int x;
(valid_in,(x = pipe_in)) |-> ##5 (pipe_out1 == (x+1));
endproperty
```

Property e is evaluated as :

1. When valid_in is *true*, x is assigned the value of pipe_in. If five cycles later, pipe_out1 is equal to x+1, then **property** e is *true*. Otherwise, **property** e is *false*.

2. When valid_in is *false*, **property** e evaluates to *true*.

One of the methodologies adopted for writing assertions is to use a set of templates which are defined in an assertion library. In this approach, the user chooses the template which he/she is needed to use for verification and fill in the parameter values appropriately. One of the popular libraries is OVL [11], the open Verification Library, a set of assertion templates is defined using VHDL, Verilog, SVA[5] and PSL[3]. Usage of SVA has been reported in functional verification [135] of SystemC [8] transaction level models(TLMs). Research efforts have been reported where a significant subset of SVA is used to synthesize checkers in BlueSpec System Verilog[140] and to hardware [112].

Apart from SVA, PSL is also used for specification and verification for designs. Boule and Zilic present a technique for automata-based checker generation of PSL *properties* for dynamic verification[36] [37]. They define a set of rewrite rules that account for all the various PSL operators. A methodology to generate hardware assertion checkers for enhancing ABV capability for hardware emulation has been proposed by Boule and Zilic[35]. Dahan et al. [52] developed a translation mechanism from PSL to SystemC [8] by using the FoCs (Formal Checkers) [14] environment. FoCs take as input PSL/Sugar specification and translates them into assertion checking modules which are integrated with the simulation environment. Habibi and Tahar [75] present a efficient approach to verify PSL assertions added on top of the SystemC [8] library.

After the property specification is done, the next task is to choose one of the two broad methodologies for property verification. These are *Dynamic Property Verification* (alternatively called Dynamic assertion based verification) and *Formal Property Verification*. In the following sections we will outline these techniques.

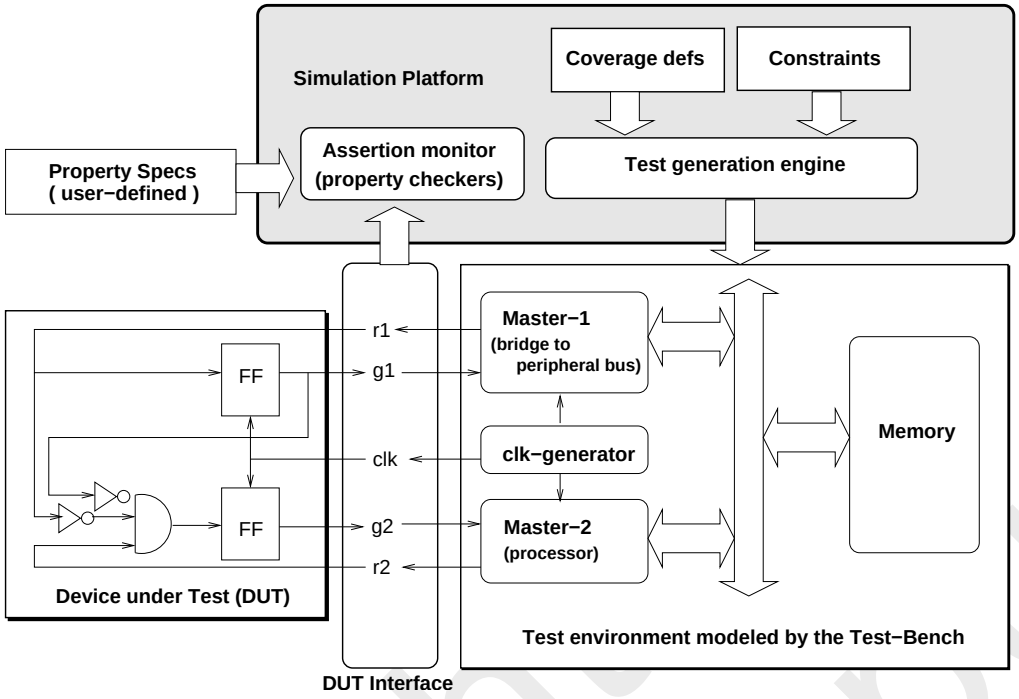

Figure 1.2: Framework for Dynamic Property Verification

1.1.3.5. Dynamic Property Verification

A typical dynamic property verification setup is shown in Figure 1.2. In the assertion based verification approach, a testbench is written to model the behavior of the environment for a design block. The testbench drives meaningful inputs into the implementation. The properties act as monitors and report any violation that occurs. The complexity of modeling a testbench depends on the design complexity.

There are two key advantages of dynamic assertion based verification. Firstly, it is built over the traditional existing simulation framework and requires less effort from the validation engineer. Secondly, unlike the formal verification counterpart, it does not have any major capacity concerns(in terms of space), since the verification is done during simulation.

The most important task of a verification engineer in dynamic property verification is to identify the set of interesting behaviors where they believe bugs may remain undiscovered. Sometimes the verification engineer may desire to ensure that a complex sequence of events is covered during simulation. Current property specification languages offer constructs to model these. These identified scenarios are called *coverage points*. Each of these *coverage points* are then modeled using a standard property specification language. After the property specification is over, the properties are stitched to the design under test. In this thesis, we present a specification logic called TAB logic and present algorithms to perform dynamic verification of TAB logic specifications over a simulation trace.

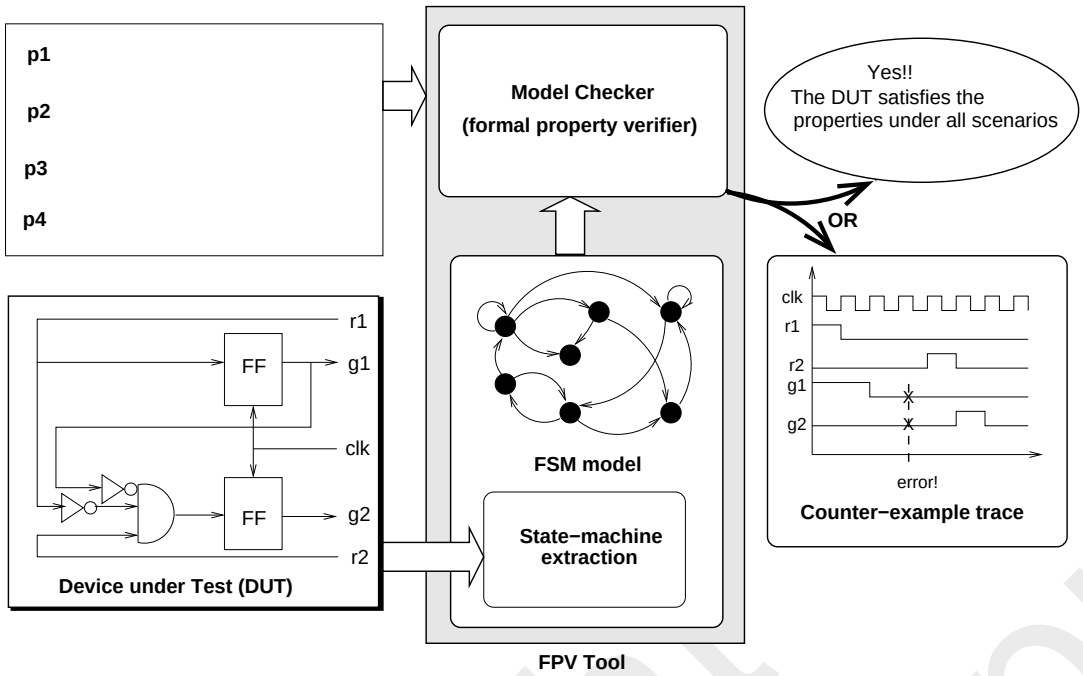

Figure 1.3: Framework for Formal Property Verification

1.1.3.6. Formal Property Verification

Formal Property Verification (FPV)[44] popularly referred to as *model checking* is an automatic verification technique for finite state concurrent systems. In this verification approach, properties specified in temporal logic are checked by an exhaustive search of the entire state space of the concurrent system.

A typical formal property verification set up is shown in Figure 1.3. The core of this approach is the *model checking* tool. Formally, a model checking algorithm [44] has two main inputs - a formal property and a finite state machine representing the implementation. The role of the algorithm is to explore all possible paths of the state machine for a path which refutes one or more properties. If such a path exists, then this path trace is reported as a counter-example. In the absence of such a path, the model checker asserts that the property holds on the implementation.

In model checking, the system to be verified is formally represented by a finite Kripke Structure (refer to Section 1.1.3). In essence a Kripke structure can be visualised as a directed graph with the vertices labeled by a set of atomic propositions. The vertices represent states and the edges represent the transitions. There could be a set of initial states.

Given a Kripke structure $K = \langle AP, S, \tau, s_0, F \rangle$ and a specification φ in LTL, the model checking problem is the problem of finding whether

$$K \models \varphi$$

A Kripke structure $K \models \varphi$ iff φ is true along all paths starting from the initial

state. An explicit state model checker is a program which performs model checking directly on the Kripke structure.

LTL Model Checking [110] algorithm first builds a Buchi automaton, also known as *tableau*, corresponding to an LTL formula φ, which subsumes every satisfying run of the formula. Given a model M and an LTL formula φ, the steps of the model checking algorithm are as given below:

- Build the tableau for $\neg\varphi$ (call it $T_{\neg\varphi}$).
- Compute the product of M and $T_{\neg\varphi}$.
- Check the product machine for emptiness.

If the product is non-empty, i.e., it contains a run, then the model M has a run which satisfies $\neg\varphi$ (and therefore, refutes φ). Hence the model checker reports the existence of a bug, and reports the run as he counter-example. If the product is empty, then it reports that M satisfies the property, φ.

Capacity limitation is one of the major issues with model checking technology. The main capacity bottleneck observed is in representing the state machine that is extracted from the design. The number of states in the machine typically grows exponentially with the number of concurrent components. Even for designs with moderate size, this leads to state space explosion.

The complexity of verification [105] for linear temporal logic has been very well studied. For a given transition system of size n and a temporal logic formula of size m, model checking algorithms runs in time $n2^{O(m)}$. Since LTL model checking is PSPACE-complete, the latter bound probably cannot be improved.

In this section, we have presented related work in the domain of assertion based verification. We discussed specification languages used for specification of design intentions and the methodologies employed for verifying these intentions statically and dynamically. We now discuss methodologies adopted for generation of test cases for pipeline processors.

1.1.4. Test Generation

In this section, we discuss the various methods of test generation for pipelined behaviors. The approaches can be broadly classified into various categories depending on the following characteristics: *(i)* Type of specification i.e., Graph based, language or ADL based or HDL based, *(ii)* Level of abstraction of the specification i.e.,functional, micro-architectural level, *(iii)* Methodology used - random test pattern generation, directed as is done with model checking and property decomposition based methods or satisfiability based methods. We first present the works which use graph based input model of the pipeline specification followed

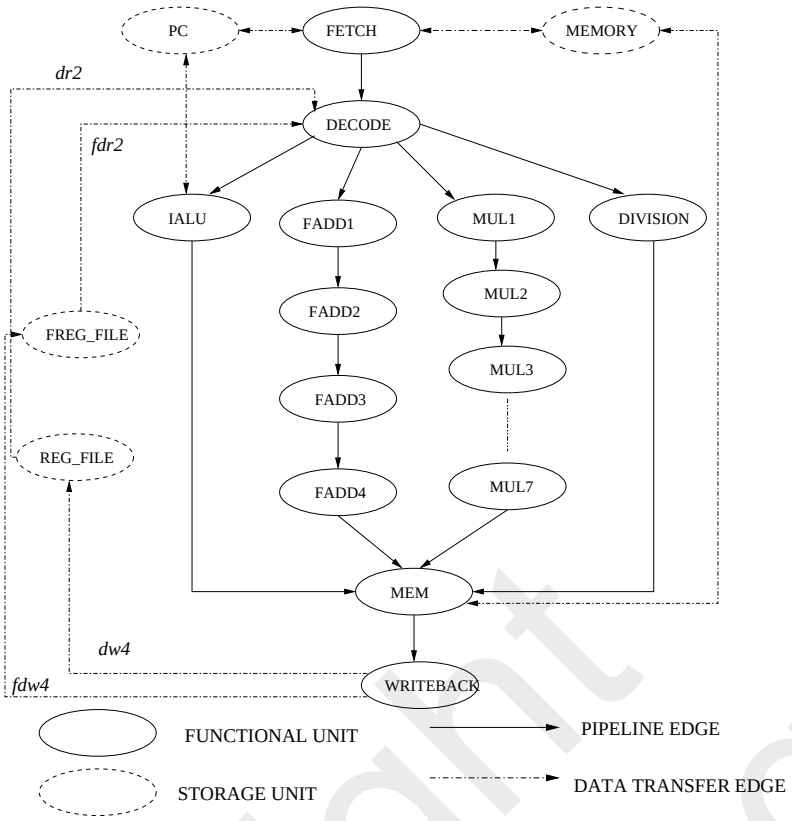

Figure 1.4: Graph Model of the Pipeline as proposed by Mishra et al. [121]

by ADL (Architecture Description Language) and HDL (Architecture Description Language) based methods of test generation ending with approaches using model checking.

1.1.4.1. Graph Model Based Test Generation

Mishra and Dutt [121] present a graph coverage based functional test program generation approach for pipelined processors. In their approach, they generate the processor model from the ADL specification using functional abstraction. This is followed by extraction of a graph model of the pipeline from the functional specification. The graph $G = (V, E)$ models the structure of the pipeline architecture with the components as nodes and the connectivity between the nodes as the edges. Each node V in the pipeline graph G corresponds to a functional unit (module) or storage component in the processor. There are two kinds of edges *pipeline edges* and *data-transfer edges*. A figure representing the graph structure of an example pipeline configuration of the DLX pipeline is presented in 1.4. The solid edges in the figure are pipeline edges and the dotted edges are data-transfer edges. A pipeline edge transfers an instruction from a parent unit to a child unit. A data edge is used to communicate data between two components in the pipeline. The behavior of each node is described using SMV [2] language. They define functional coverage of the pipeline in terms of the coverage of nodes and edges of the

generated pipeline graph. In order to cover all scenarios, they define all possible interactions between the instruction and pipeline stages through graph coverage. The nodes in the graph could be in 4 states namely *active*, *stalled*, *exception* and *flushed*, and the edges could be in 3 states *active*, *stalled* and *flushed*. They generate test cases for covering the edge and node states using a test program generation algorithm that traverses the pipeline graph to generate test programs based on the coverage metric. The algorithm breaks one processor level property into multiple module level properties and applies them to the SMV model for every module. This led to significant reduction in memory and time requirements in comparison to the case where SMV of the complete pipeline is described. They have applied this methodology on the DLX processor to demonstrate the usefulness of their approach.

Mishra and Dutt [123] propose a general graph-theoretic model that can capture the structure and behavior (instruction-set) of a wide variety of pipelined processors and a functional fault model that is used to define the functional coverage for pipelined architectures which is similar to that presented in [121]. The structure of the pipeline architecture is captured as a graph model as described above. Additionally, the behavior of the instructions present in the ISA, are modeled as a behavior graph. The behavior models the operation and internal execution semantics of the instruction. In the behavior graph, there are two types of nodes: opcode and argument nodes. There are two kinds of edges operation and execution. The operation edges link the fields of the operation and also specify the syntactical ordering between them and the execution edges specify the execution ordering between the fields. They categorize the various computations in a pipelined processor into the following categories (i) register read/write, (ii) operation execution, (iii) execution path and (iv) pipeline execution. They define fault models for each of these functions. They present test generation procedures that accept the graph model of the architecture as input and generate test programs to detect all the faults in the functional fault model. We use this graph based description of the pipeline as the input to the test generation framework as it can be used to specify a wide range of pipeline behavior.

1.1.4.2. Specification Language Driven Test Generation

Configurable processor cores are becoming increasing popular with SoC designs because of their capability to quickly adapt to the changing designer requirements. Xtensa [68], a fully configurable and extensible processor core, allows users to add new instructions to the processor core optimized for their application. The instruction set and the pipeline model of the processor is no more fixed. The new register files for the data, new opcodes and instruction implementation is

specified using Tensilica Instruction extension(TIE) language [6]. This requires newer methods of verification of such processors. After the user adds new features like register files, functional units and instructions, the pipeline implementation needs to be verified. These additions leads to new data and control hazards which are not known *apriori*. A verification methodology has been presented in [29] which tests critical micro-architectural features of a processor without any *apriori* knowledge of the instructions, or specific implementation details. They generate test sequences which covers all control-hazard and data-hazard cases. Control hazard cases comprise of all cases where the control flow of an instruction is affected as in the presence of branch and exceptions. Data hazard cases comprise of read-write dependencies between in-flow instructions present in the pipeline. The tests are automatically derived from the ISA description and scheduling information of the instructions.

1.1.4.3. SAT based methods

With the increasing capability of modern SAT solvers [134, 117, 67, 160], handling large state space problems has become possible. The complexity of modern day processor pipelines is increasing considerably making it difficult to handle the large state spaces using existing tools. SAT based approach is increasingly being used for test generation of complex pipelines as it can handle large designs. Mishra et al. [130] address the challenges in language-based validation in the domain of test generation and equivalence checking using satisfiability checking. In order to generate test cases for complex situations like multiple exception scenarios, they present a SAT-based bounded model checking (BMC) approach. The basic idea behind this approach is to restrict the search space to states which are reachable within k transitions from the initial state. They unwind the model k times and build a SAT instance out of it. The value of the bound is not known *apriori*. They have also proposed a method to determine the bound for each property, which is the longest temporal distance from the *Fetch* unit to the nodes under consideration. They have shown that SAT based bounded model checking (BMC) performs well for finding shallow counterexamples for test generation.

In [96], Koo and Mishra present a similar approach for efficient test generation using SAT-based bounded model checking (BMC). The difference being that they translate the functional fault S_i in the fault model to a temporal logic property P_i. The first step is similar to above which involves determining a bound k_i for the property S_i. SAT-based BMC takes processor model M, negated property $\neg P_i$ and bound k_i as input and generates a counter-example, which serves as a test-program for the fault. They applied their methodology on a simplified MIPS architecture and used Cadence SMV [2] as a model checker and zChaff [134] as

a SAT solver. They observed that the *bound* for each property reduces the test generation time by 90% compared to using BMC with maximum bound, which is the depth of the pipeline. In this thesis, one of the methods presented for generating test cases uses a SAT based approach to generate test cases from the user given scenario.

1.1.4.4. Property Checking based methods

In [120], Mishra et al. propose a model checking based approach to automatically generate functional test programs for pipelined processors. Firstly, the processor model is extracted from the ADL specification. The user specifies certain properties which the processor architecture should satisfy. The processor model and the properties are written in SMV language. They classify test cases into various categories including pipeline flow, feedback paths, branch prediction, execution mode etc. They write properties for each category of the test cases. The properties are then applied to the processor model using the SMV Model checker [2]. In fact, they negate the property and apply it to the processor model thereby getting a counter-example which can be used as a test case which violates the property. They apply their approach on a single-issue DLX procesor to demonstrate the usability of the approach.

Koo et al. [97] present a directed test generation technique at micro-architectural level for functional validation of microprocessors. A processor model is described in a temporal specification language at micro-architecture level. They target directed test generation towards micro-architectural faults. Like the previous approach, they also use a SMV model of the processor. In their case, the micro-architectural faults are specified as properties. These properties are then broken into sub-properties (decomposed properties) which represent the interaction between instructions and the respective units. Each of these properties are then applied to appropriate modules and the result composed to form the final test case. They take care of temporal ordering of the various instructions in the individual test programs while composing them to form the complete test case. A similar methodology which uses decompositional model checking with some additional results have been presented by Koo and Mishra [95].

1.1.4.5. FSM based methods

One of the methods of abstraction used by designers is to abstracting out the state transition behavior of the pipeline in terms of a Finite State Machine(FSM). This FSM is then used an input representing the pipeline model for test case generation. Shen and Abraham [147] have proposed an abstraction technique for micro-architectural validation. Firstly, the FSM is extracted from the Verilog/VHDL followed by generation of all transition paths in the FSM of a given

finite length. They use path coverage as a measure of coverage of a given test-suite. The uncovered paths are then used to guide the generation of directed test cases. Although they demonstrate through results that they achieve 100 % path coverage, the number of paths grow non-linearly often exponentially with the pipeline depth, making this method unsuitable for handling complex pipeline behavior. However, since the designer view starts from a functional specification and there after evolves into the micro-architectural level after going through several phases, this approach cannot be used for test generation in the initial phases of the design.

Ho et al. [86] use a similar methodology to extract FSM model from the structural HDL description, building the complete state graph with all the valid transitions. They cover all the transition edges in the state graph to generate transition tours and generate sequence of instructions which enables the transitions in the transition tour. Their system works from a *verilog description* of the original machine and is used to validate an embedded dual-issue processor in the node controller of the Stanford FLASH Multiprocessor. In order to expose bugs, they exercise all control edges in the state transition graph and enumerate those covered edges on the implementation thereby listing the bugs present.

1.1.4.6. Other methods

Apart from the methodologies listed above, a gamut of other test generation techniques have been applied to pipeline validation. We discuss the various methodologies here. In [125], the authors present a test generation and functional coverage estimation framework for pipelined processors using Specman Elite [10]. The input specification is in the form of an ADL, from which an "e"[10] specification is extracted. Specman Elite is used to generate random and constrained-random test programs. They compute functional coverage of a pipelined processor for a given set of test programs as the ratio between the number of faults detected by the test programs and the total number of detectable faults in the fault model. The fault model adopted is same as the one defined in [123]. They have applied this methodology on a VLIW DLX architecture to demonstrate the usefulness of their approach. They analyze the cause for the low fault coverage in pipeline execution for the random and constraint-driven test generation approaches. These two approaches covered all the stall scenarios and majority of the single exception faults. However, they could not activate any multiple exception scenarios in their case.

Constraint Solving is one of the methodologies adopted by IBM where the test scenario is generated by solving a constraint satisfaction problem. Piparazzi [16], the test case generator from IBM, generates test cases at the architectural and micro-architectural levels. The main inputs to the framework are: a model of the

micro-architecture and the user's specification of a required event. A CSP problem is constructed from these inputs and this problem is solved to generate a test sequence. IBM built on the model-based test generation methodology supported by CSP to built Genesys[111], Genesys-Pro[15]. Aharon et al. [17] present a model based test generator which comprises of an architectural model, a testing knowledge data-base, a behavioral simulator, architecture independent generator and a graphical user interface for user inputs. Ur and Yadin [154] use Genesys [111] in their coverage driven test generation framework. They use FSM abstraction and the tours on the FSM are translated to architectural constraints which is then provided as input to Genesys [111].

Fine and Ziv[63] address one of the main challenges of simulation based verification (or dynamic verification), by providing a new approach for Coverage Directed Test Generation (CDG) based on Bayesian networks and computer learning techniques. The approach provides an efficient way for closing a feedback loop from the coverage domain back to a generator that produces new stimuli to the tested design.

In this section, we have presented research work related to the focus of this thesis. We present the motivation behind this work and the objectives of this thesis in the next section.

1.2. Motivation and Objectives

Simulation Based Verification(SBV) involves building an executable of the system and providing the executable with test stimulus and observing that the output conforms to the specification. This validation is done by comparing the output of the simulation against an existing golden output or by checking the output traces with assertions. Checking assertions over traces is widely known as Assertion Based Verification (ABV), which could be done offline or post-simulation as well as online or at par with simulation. Additionally, test stimulus form an integrated part of the simulation based verification process. Test stimulus provided to the simulation framework can be absolutely random or directed towards a certain user specific scenario, or towards meeting certain objectives like coverage. We explore the applicability of the above mentioned SBV methodologies in the domain of pipelined processors.

The three objectives of our work involve: *(i)* Exploring techniques for verification of a pipeline simulator against the goldenized functional simulator, *(ii)* Investigating the usage of Assertion Based Verification in Pipeline Validation, and *(iii)* Developing methods for Directed Test generation for pipelined processors. The first two objectives are verification methods defined on a simulation trace. The first approach uses the simulation traces of both the functional and

pipeline simulators while the second objective requires the execution trace of just the pipeline simulator and a set of assertions. The third objective is to allow the user to generate test cases for user given scenarios. We now describe these objectives in detail.

1.2.1. Trace Equivalence:

Simulation based Verification methods for processor validation involve techniques proposed to cut down on the number of tests [71], for validating pipeline interlocks [58], and also for simulation-based verification using buffer-oriented approaches [155], [156]. Utamaphethai et al. [155], [156] systematically generated test patterns and used buffer-oriented micro-architecture validation (BMV) models for rename buffers, reservation station, and branch-prediction. All the above approaches required detailed pipeline specifications. Austin [24] introduced a dynamic implementation verification architecture (DIVA) technique in which the core processor is augmented with a lightweight checker before the commit stage which corrects erroneous results produced by the core processor. In his model, the formal verification task is hence lightened to the functional verification of the simple checker only. Mneimneh et al. [132] demonstrated the use of DIVA architecture to formally verify an ISA by reducing the burden of verifying a core processor. This method requires additional hardware and the overhead of verifying it.

The designer builds a functional simulator for the ISA, goldenizes it using benchmarks[25] and random instruction test sequences [10]. A considerable amount of time is thus spend in validation of the functional simulator. This goldenized functional simulator is passed onto the pipeline design phase. The processor pipeline design, as we have already mentioned in Section 1, is a feedback based iterative process, where the iterative process continues till the performance specification is satisfied. Verifying the correctness of the pipeline implementation at every step is a very important task as it is used for performance evaluation at every iteration. Since, the designer has a fully tested functional simulator at hand, it could serve as a golden reference model for the pipeline simulator. Most of the reported approaches employed for pipeline validation require detailed specification of the pipeline(*white box*). It is often the case that the validation engineer has an executable model of the pipeline which needs to be validated. The absence of the detailed pipeline design makes the application of known validation techniques difficult. However, the executable model of the pipeline generates execution traces. It would be interesting to do a *comparative analysis* of the simulation traces of the pipeline and functional simulator and deduce a notion of correctness and containment. We investigate the existence of an *equivalence relationship* between the simulation traces generated by the functional and pipelined implementation of

the ISA, which does not require a detailed architecture specification(*black box*). Hence, our *first* objective is to leverage the existence of a fully tested functional simulator and use it for verifying the correctness of a pipelined implementation of the ISA. Next, we examine techniques for pipeline validation from the perspective of assertion based verification.

1.2.2. Assertion Based Verification of Pipeline:

Assertion Based verification is rapidly gaining acceptance in the design community. Assertions are specified using some form of temporal logic or their extensions and are checked on simulation traces online or offline. LTL(Linear temporal Logic) and CTL (Computational Tree Logic) are the de-facto standards used in specification of design intentions[113],[114]. In [104], non-deterministic automata are built from LTL to check violations of formulas over finite traces and the complexity of these problems are analyzed. In tools such as the *Temporal Rover* [59] and *Java Pathexplorer* [82], LTL based specifications have been used. There are two major limitations of conventional temporal logic, namely: (i) incorporating data-orientation especially in the presence of timing constraints and (ii) modeling and association of timing constraints in design intentions. Consider the following specifications of a bus integrated pipelined processor system given below:

Example 1:

(a) The address floated on the system bus should not increase by more than 4 from its value at the immediately previous time the bus was ready.

(b) The system allows the address to increase by more than 4, but keeps a count of how many times such a violation has occurred. That is, this count should increase by one after such a violation has occurred and stay so until another violation.

(c) The system does not allow 5 such violations to occur within 100 time units.

We examine such pipeline specifications and other related timing properties of instructions and interrupt processing from the point of view of assertion based verification. There is no convenient way to express these assertions in conventional temporal logic owing to their inability to model data dependencies across various temporal contexts. PSL(Property Specification Language)[3] and SVA(System Verilog Assertions)[5] are two industry standard property specification languages. Both languages primarily use free variables for data orientation and manipulate data in assertions using auxiliary state machines written by the user. These machines are often complicated and become a source of error. Moreover, the assertions for data orientation turn out to be more procedural in nature than declarative. Hence, it is important to provide an elegant, essentially declarative solution

to specification of data-oriented assertions, which is also a succinct representation in a manner similar to control-oriented assertions. Additionally, it is necessary to provide a loosely coupled framework that enables easy automated verification across various levels of specification without the user having to write additional assertions. The *second* objective of this thesis is to develop a specification logic which could succinctly represent data-orientation and timing constraints in design intentions, and investigate its applicability in the domain of pipelined processors.

1.2.3. Test Generation

Typically, a combination of random and directed test programs are applied for performing simulation based validation of microprocessors. Test cases are an inherent component of a comprehensive Simulation Based Verification(SBV) framework. The effectiveness of SBV depends heavily on the quality of test cases used as test stimulus. The input space of complex pipelines is quite large and testing them for each input is a difficult task to be accommodated within the already constrained time-to-market. Random test pattern generation could prove to be a costly validation technique in terms of time as coverage is not guaranteed. Hence, designers prefer methods for directed or targeted test generation. Using their detailed design knowledge, they test the processor for critical conditions where certain kind of instructions interact with each other i.e. cases where there can be hazards, exceptions etc. and there would be situations, where they would want to test the register renaming protocol etc, or an entirely user specific scenario, extensively. Piparazzi [16] is a test generator, developed at IBM that generates (architectural) test programs for micro architectural events using constraint satisfaction. Test generation for processors has been reported using FSMs [89],[94], coverage of pipeline behavior [121], functional fault model of the pipeline[123]. Iwashita et al. [90] generate behavioral level test environments in VHDL for specific processor mechanisms.

All of the known test generation approaches for pipelined processors generate test cases for standard scenarios like hazards and exceptions. Many a times the designer would want to test the design for a particular type of scenario of his or her choice, like a structural hazard at the *MEM* stage or a *RAW* hazard or an implicit hazard or any specific corner case scenario. One of our objectives is to provide the designer flexibility to specify the scenarios for which he/she requires the test cases to be generated. Corner case scenarios in the pipeline are often complex and require multiple events to be triggered. A typical example is a scenario where multiple exceptions or multiple hazards or a combination of both occur in the pipeline. The scenarios that are considered critical to the pipeline execution are when instructions interact with each other like when hazards, exceptions, or

interrupts occur and so on. Consider this scenario: a designer desires to validate the logic of the pipeline when a multi-cycle hazard, which is a specific case of a structural hazard, occurs. The user describes this scenario as an instance where an instruction *'I'* is waiting to occupy a unit *'U'*, but the instruction *'J'* in *'U'* occupies *'U'* in the next cycle too. Now *'I'* would get stalled in its current unit. Generating test cases for such scenarios is a difficult task. Also, it is important to provide the user with a framework which could generate test cases at all levels of design granularity. This would enable reuse of test scenarios and seamless testing across all design levels. Our *third* and final objective is to provide the user with a flexible framework which enables the user to specify such types of scenarios and exhaustively generate test cases for the specified scenario.

1.3. Contribution of this Thesis

The contribution of this thesis in the domain of simulation based verification of pipelined processors is three fold. The first important problem that this thesis solves is the problem of functional equivalence between pipeline and functional simulator using the register access traces. The thesis then presents a specification logic(TAB logic) for assertion based verification of a pipelined processor. The final contribution is in the domain of directed test pattern generation of pipelined processors. The thesis presents a flexible framework for scenario driven test pattern generation for functional validation of pipelined processors. The first approach is a *black box* verification methodology and the last two approaches are *white box* methodologies for verifying pipeline designs. We summarize the contributions of the thesis in this section. All experimental results reported have been taken on a 1.7 GHz CPU with 1 GB of RAM.

1.3.1. A systematic framework for Validation and Debugging of Pipelined Simulators

We propose a simple yet versatile model with which we can validate the correctness of a pipelined simulation with respect to a functional simulation based on a data flow analysis approach. Unlike methods based on formal verification, our approach *does not* require as input detailed specification of the pipelined architecture. It is based on the availability of simulators that produce sequences of read/write operations of instructions. Unlike pure simulation-based approaches that match final data values, we present a new equivalence approach that not only detects errors as soon as possible, but also attempts to debug the error source. We allow the pipelined simulator to use more registers than the functional simulator, perform out-of-order execution, speculative instruction execution, and register renaming. We indicate how this approach captures architectures that use methods

like Tomasulo's algorithm[84]. In our framework, the verifier receives streams of register accesses by instructions from both the functional and pipelined simulators. Based on the schedule of instructions executed by the two simulators and the registers read and written, we propose a definition of semantic equivalence called D^* equivalence of two such instruction schedules. We propose an algorithm which under certain reasonable assumptions verifies the aforementioned semantic equivalence in time polynomial in the number of instructions executed and the number of registers. We present some extensions to the algorithm to handle some more generic cases. Another important aspect of our work is debugging errors during simulation. We demonstrate that the proposed approach can be used to locate faults in the source code of the pipeline simulator.

Given an executable code in the ISA, we execute the code on both the simulators. In an in-order un-pipelined execution of the code, as in a functional simulator, an instruction will perform all its register accesses (reads/writes) before the next instruction starts execution. However, in an out-of-order pipelined simulator, multiple instructions may be accessing the registers without waiting for another instruction to finish. A sequence of read/write operations to registers by instructions will be called the schedule produced by the particular simulator. Thus, in the schedule produced by a functional simulator, we will have a sequence of all the requisite access operations to the registers by a particular instruction strictly before the sequence of access operations of the subsequent instruction. On the other hand, the pipeline simulator can produce a schedule in which accesses to registers by multiple instructions is interspersed. It is to be noted here that such a sequence of operations induces a data flow graph among the instructions. In our framework, the verifier is a module, separate from the simulators, which receives the schedules produced by both the simulators. The specific problem we formulate and solve in this work is whether the data flow among the instructions induced by the schedule produced by the pipeline simulator is semantically equivalent to the schedule produced by the functional simulator. For sake of experimentation, we have used the functional and pipeline simulator for *pisa* architecture available with simplescalar tool suite[26]. We present the verification results for various application from the Mibench benchmark[25] test suite. We executed all of the tests mentioned and collected the register dumps. We observe that the proposed method has a linear time complexity in practice. The checker validated the pipeline simulator with respect to the functional simulator for each of the tests. We also introduce faults into the source of the pipeline simulator and demonstrate that each of these faults are detected very early during simulation thereby saving a significant number of simulation cycles on the part of the designer, up to $> 99\%$ for certain benchmark applications. For example, in case of the Blowfish application, the bug detected

was at cycle - 7, and the total number of simulation cycles reported is 7878162. We also show that this leads to improved methods of fault location as we can map the failure type of the algorithm to the possible error locations.

1.3.2. Simulation Based Verification using Temporally Attributed Boolean Logic

We propose a specification logic called Temporally Attributed Boolean(TAB) Logic which allows us to: *(i)* represent assertions succinctly, *(ii)* incorporate data-orientation and *(iii)* associate timing to design intentions. TAB Logic enables us to write specifications functionally linking system variables from different temporal contexts. Temporal logics have constructs(temporal operators) to qualify past, present and future time instants. These temporal operators are associated with properties, propositions or events. Similarly, in TAB Logic, there are three kinds of operators which register events in the past, present and future. The basic difference being that the TAB logic operators evaluate to a time instant in past, present or future, whereas temporal operators as in LTL, CTL in general signify the existential truth of the proposition or event qualified by the temporal operator. For example, if the LTL formulae Fq holds in some state it implies that q is true in some state in the future. In TAB logic, $N[q]$ evaluates to the immediately next time instant when q is true in the future. The three basic TAB logic operators are *(i)* Previous, *(ii)* Current; and *(iii)* Next. One such time instant that a temporal operator in TAB logic evaluates to is called a temporal context. The signal values attributed by already defined temporal contexts are called temporally attributed signals, and are used in the definition of newer temporal contexts, thus allowing nesting in the definition of temporal contexts. We illustrate this by expressing the specification in Example 1 using TAB logic. We mentioned in Section 1.2 that one of the objectives of TAB Logic is to incorporate data-orientation in specifications; the conditions monitored by the temporal operators of TAB Logic are data dependencies that are modeled as functions. These functions are defined over temporally attributed signal values as operands with arithmetic and logic operators. Finally, sets of assertions are defined which assert high level data dependencies utilizing temporally attributed signal values. In the nomenclature used in TAB logic, the temporal contexts are known as Temporal Expressions and the data dependencies and conditions are modeled in the form of TAB or Temporally Attributed Boolean Expressions. We combine example 1(a), 1(b) and 1(c) into a single TAB Logic specification.

Example 1(a,b,c). system variables : *bus_address, bus_ready,counter*

time $t\_pb = P[bus\_ready == TRUE]$;
time $t\_v = P[bus\_address > bus\_address(t\_pb) + 4]$;
time $t\_n100 = C[\,] + 100$;
assertion a_1: $(bus\_address \geq bus\_address(t\_pb))$
$\wedge\, (bus\_address \leq bus\_address(t\_pb) + 4)$;
assertion a_2: $counter == counter(t\_v) + 1$;
assertion a_3: $counter(t\_n100) \leq counter + 5$;

Explanation: The temporal variable, t_pb, corresponds to the temporal context in the immediate past when the bus was ready, signified by the TAB expression, $bus\_ready == TRUE$. The assertion states that if the current address on the bus, signified by the variable $bus\_address$ is greater than the value of $bus\_address$ at time t_pb, then the increase in value of $bus\_address$ is at most *4*. The number of violations signified by the value of the variable *counter* increments every time the address on the bus increases by more than *4* from the value of the address when the bus was ready the last time. The time instant is signified by the temporal variable t_v. The temporal variable t_n100 signifies the temporal context *100* time units from the current instant. The total number of violations within this interval is signified by the value of the variable, *counter* at t_n100. This value should be within *5* units of the value of *counter*, *100* time units in the past.

One of the important results that we arrive at is that TAB logic is at least as powerful as LTL in terms of expression capability. We also show how data-oriented specification can be specified in TAB which is not easy to do in LTL. We formally define TAB Logic, formulate the problem of verification on a simulation trace and present efficient algorithms to check TAB assertions, both offline and online. We present results of application of TAB Logic for Instruction Semantics, Bus Transaction Verification, and Interrupt mode behavior of a bus integrated pipelined processor core implementation. We also demonstrate the effectiveness of TAB Logic in fault detection, typically of data-oriented faults. As an off-shoot, we investigated the applicability of TAB Logic for simulation based verification of analog circuits like Operational Amplifiers and DC-DC Converters and we obtained some interesting results. We illustrate through examples, the kind of assertions that can be specified using TAB logic in the domain of reasoning about output waveforms which is one of the primary necessities in case of simulation based validation of analog circuits.

1.3.3. A Framework for Scenario Driven Test Case Generation for Functional Verification of Pipelined Processors

We propose a flexible directed test generation framework to generate test cases for microprocessor pipelines targeted towards user defined scenarios. There has been significant amount of research in this domain and there are results reported in random test pattern generation [15], fault model based test generation[123], specification driven test generation [29],[124] and coverage driven test generation[121]. It is important to provide a systematic test generation framework which will provide the designer the flexibility to control all the components of the test generation process. There are three important independent components of a test generation framework for pipeline processors: *(a)* Scenario Description, *(b)* Pipeline Specification, and *(c)* Test Generation Algorithm. Control over these components enable efficient test case generation for pipelined processors. It is necessary for the designer to generate test cases across all levels of abstraction of design. Most of the reported methodologies lack the flexibility to work across various levels of abstraction and generate test cases either only at functional level or at the architectural level. Along with this, the designer would also be interested in extensively testing the functionality of a new pipeline methodology he/she had designed using minimal architectural details of the pipeline. In order to facilitate this the test generation process should be aware of the granularity of the pipeline specification. Another important aspect of the pipeline test generation process is the test generation algorithm employed. It is often a requirement of the test engineer to evaluate various algorithms for generating test cases directed towards a given class of targeted scenarios. This can be facilitated if the pipeline test generation framework is flexible enough to allow test generation algorithms to be plugged in. We present a framework which will address these issues and provide the designer with the adequate amount of flexibility to perform efficient testing of the pipeline designs.

We propose a Scenario Description Language (SDL) where the user can describe the scenarios using simple predicates pertaining to the pipeline. This would enable generation of test programs for complex user defined scenarios. An abstract model of the pipeline similar to the model proposed by Mishra and Dutt[121] is used as input for test generation in our case. Mishra and Dutt [121] use the behavioral knowledge of the pipelined architecture specified in an Architecture Description Language (ADL). However, in our approach we also have the scheme of specifying the hazards/exceptions as inputs to the test generation framework which enables the test generation framework to generate multiple hazard based test cases. We present two test generation algorithms which take the scenario and

pipeline specification as input and generate hazard free and exhaustive hazard based test cases respectively. In our first approach, we present a fast algorithm for hazard free test case generation using the pipeline specification and the user given scenario as input. In order to generate the hazard based test cases, we present a method to model the pipeline as a SAT instance. In this case, the hazard and exception conditions form a part of the pipeline specification which is not the case in the first approach. The desired scenario of the pipeline is automatically converted into a state of the pipeline, which is translated into an equivalent set of boolean propositions which is used to generate minimal test cases using an ALL-SAT approach. We generate test cases for three different configurations of the DLX pipelines for various well known hazard scenarios.

1.4. Thesis Organization

This thesis presents three enhancements in the field of processor pipeline verification. Firstly, the thesis defines a data flow correspondence relationship between pipeline and functional register access traces and presents equivalence checking algorithms for the same. Secondly, the thesis presents a specification logic called Temporally attributed boolean logic for assertion based verification of timing properties of instructions and interrupt processing in the processor pipeline. Lastly, this thesis presents a framework for directed test pattern generation of pipelined processors. The rest of the thesis is organized as presented next:

Chapter 2 proposes a concept of semantic equivalence between two architectural simulation register access traces and enumerates algorithms to check the equivalence relationship.

Chapter 3 presents a specification logic called TAB Logic which allows us to write specifications functionally linking system variables from different temporal contexts and algorithms for checking the same.

Chapter 4 presents a flexible framework for scenario driven test generation for functional validation of pipelined processors.

Chapter 5 presents conclusions and scope for future work.