Verification-Driven Design for Asynchronous VLSI

Xiang Wu
Yale University
New Haven, CT, USA
xiang.wu@yale.edu

Rajit Manohar
Yale University
New Haven, CT, USA
rajit.manohar@yale.edu

Abstract—Although verification has widespread use in hardware design, there is still a gap in verifying the functional description of hardware against its detailed design. In this paper, we present our effort to close that gap. Drawing an analogy from test-driven development in software development, we propose the idea of verification-driven design for hardware design—the hardware design flow should be engineered in a verification-friendly way. We showcase this idea by leveraging a methodology in asynchronous circuit design that is similar to software optimization—complicated, parallel microarchitectural optimizations are derived from a simple, sequential functional description through iterative rewriting. We propose a technique to formally verify these transformations of hardware designs by extending the state-of-the-art translation validation algorithm for software verification. This approach is demonstrated by a superscalar processor example. We assert that our parallel superscalar processor design is equivalent to a sequential non-superscalar processor specification using the translation validation technique presented in this paper.

Index Terms—formal verification, asynchronous circuits, translation validation

I. INTRODUCTION

A chip is designed by starting from an initial functional description (sometimes called the golden model) that captures its specification. The golden model is detailed in a software programming language (typically C/C++), and the resulting executable is used as the reference for detailed hardware design. The manual chip design process creates a much more detailed hardware model in a hardware description language, and simulators are used to compare the behavior of the hardware model against the golden model. Logic synthesis tools translate the hardware model into gates implemented with transistors and their interconnections. Finally, these gates and connections are translated into physical geometry that specifies the individual layers that implement the devices and wires that implement the final physical realization of the chip [1].

Unlike software bugs that are routinely corrected by distributing electronic patches, hardware bugs require a change in the design of the chip and thus cannot be corrected so easily (with the notable exception of field-programmable logic). Due to the expensive nature of hardware bugs, significant effort is devoted to making the design of hardware bug-free. Extensive testing, at best, can only expose bugs but can never guarantee the absence of them. As opposed to this, formal methods can prove the absence of bugs in a hardware system.

However, even with formal methods widely applied in the state-of-the-art synthesis flow, bugs are still common in modern commercial hardware. This is because most formal methods applied in mainstream VLSI design flows are either at a low-level (such as equivalence checking of combinational logic) or only check certain properties instead of the functional correctness of the entire design. An executable golden model is not integrated into the formal verification flow but rather serves as a reference for the functionality. This is understandable because the traditional synchronous VLSI design flow is fundamentally unfriendly to such verification due to the disparity between the golden model and the detailed hardware model.

In this paper, we propose a design philosophy of verification-driven design—the hardware design flow should have formal verification in mind. We argue that the design flow itself should be designed in a formal-verification-friendly way. We are inspired by the idea of stepwise refinement [2], [3] in software, where the final program is derived through iteratively rewriting a higher level program starting from a program that serves as a specification for the high-level design. Formal verification of stepwise refinement can be hard in software development due to challenges in scaling up verification techniques to the problem of verifying millions of lines of code. However, it is a simpler problem for hardware design as the programs that specifies and implements the hardware are usually of a smaller scale and more constrained than a complex piece of software. The idea of stepwise refinement in the context of asynchronous design is sometimes referred to as the formal synthesis approach, where a design is manually translated into its final concurrent implementation [4]. Verification-driven design builds on this idea by incorporating automated formal verification support with some user-assistance during the design process.

The most prevalent method for designing digital chips is the synchronous approach, where a global signal (the clock) is used to sequence all operations on the chip, and the chip is viewed as a large finite state machine with the clock tick advancing the global state. This is reflected in detailed hardware models specified in languages such as Verilog. However, there is a disparity between the golden model described in a software programming language (which does not include an explicit clock) and the implementation in Verilog, which contributes to the difficulty of formally connecting the two. It also makes a stepwise refinement approach and its formal
verification hard due to the fact that we have to introduce the global clock somewhere in the refinement and verify it, which is likely a trade-off between the accuracy of the formal model used in verification and the performance of the chip.

In this paper, we look at the asynchronous (a.k.a. self-timed) approach in digital chip design which abandons a globally synchronized clock. In this approach, the chip is modeled as a concurrent system with components interacting through explicit signaling, both in the high-level specification and in the detailed hardware model. We use translation validation to validate the correctness of each particular microarchitectural optimization. Using these methods, we can formally verify the correctness of a superscalar microprocessor design whose final pipeline resembles those of state-of-the-art superscalar microprocessors [5].

We use the CSP- language for the golden model. CSP- is an imperative version of a subset of Hoare’s Communicating Sequential Processes (CSP) [6] and written using a syntax borrowed from Dijkstra’s guarded commands language [7]. It specifies a chip as a concurrent system with channels for communication. In this paper, we show that it is possible to make microarchitectural optimizations such as superscalar execution at the CSP- level and formally validate the correctness of such optimizations.

We view our design flow as consisting of a set of steps of high-level microarchitectural optimizations, which enable optimizations such as pipelining, superscalar execution, and caches. Such optimizations are based on a set of transformations that are applied manually based on design requirements. We start from a CSP- golden model and end up with an optimized CSP- program.

We showcase our approach by formally verifying a superscalar microprocessor. In our approach, we do not start from a superscalar design as our golden model because such a design is already too complicated to assert its correctness. Instead, we use a simpler golden model—a sequential, general processor that implements the same instruction set architecture (ISA). We view superscalar execution as an optimization of the golden model, and we formally verify it as such. This is different from verifying the implementation of a superscalar specification. To the best of our knowledge, such verification of a superscalar design as an implementation has not been done before. We remark that our approach is a general one and not specific to the superscalar microprocessor.

In this paper, we make the following contributions:

1) An extension to the state-of-the-art translation validation technique to support hardware programs that are possibly reactive and parallel. In particular, we support CSP-*, an explicitly parallel, message-passing language.

2) A case study of deriving a superscalar processor design from a functional specification of a general processor through stepwise refinement, in a verification-driven manner.

3) Automated formal verification of this verification-driven superscalar processor design against a functional specification through translation validation.

### II. ASYNCHRONOUS VLSI DESIGN FLOW

In this section, we briefly introduce the asynchronous VLSI design flow, which is the target flow for our verification framework.

#### A. Asynchronous Design

Fig. 1 shows the design flow for an asynchronous VLSI (AVLSI) implementation of a chip. In this paper, we use the term AVLSI flow to refer to Martin Synthesis [4], which is the synthesis flow that our verification techniques target.

We divide the AVLSI flow into three phases: microarchitectural optimizations, logic synthesis and physical implementation, and tape-out. In this paper, we focus on the microarchitectural optimizations. An asynchronous VLSI designer can use standard techniques in AVLSI, such as syntax-directed translation [4], [8], to obtain a lower-level implementation of the hardware.

Most formal verification works in hardware are focused on the synthesis part, including logical- and physical- synthesis, which will be discussed in Section V. The translation validation technique we used for stepwise microarchitectural optimizations is discussed in Section III.

#### B. CSP- Language

As we have mentioned in Section I, hardware designers start from a golden model that describes the functionality of the chip. To describe such models and high-level microarchitectural optimizations, we use CSP- as our hardware description language (HDL) at the highest level.

The CSP- language is a subset of Hoare’s CSP. Like CSP, CSP- features explicit process-level concurrency, message-passing via point-to-point channels and no shared variables. Hence, in CSP-, processes can only communicate through explicit channel actions. CSP- further restricts that no communications can be used in selection guards. All guards are therefore expressions. A brief summary of the syntax of a CSP- process $p$ is shown below, where $v$ is a variable, $C$ is a channel name, and $e$ corresponds to an expression.

\[
p ::= v := e \mid C!e \mid C?v \mid p;p \mid ![p] \mid [e \rightarrow p]..[e \rightarrow p]
\]

We use the syntax $C!e$ and $C?v$ for send and receive respectively. Our communication is zero-slack, which means
that the channel cannot buffer any values (this is sometimes called rendezvous synchronization). The communications in CSP- are one-sender-one-receiver and blocking—every send operation must be matched with a receive operation, finish the communication and proceed. An example of usage can be found in Fig. 2. The expression “a[p]” stands for the infinite repetition of p. We use “::=” for assignment, “::” for sequencing, “f(...)” for function call, and “arr[i]” for array indexing. The expression \[ e_1 \rightarrow p_1 \] \[ \ldots \] \[ e_n \rightarrow p_n \] is a guarded selection: when \( e_i \) is evaluated to be true, \( p_i \) is selected. In CSP-, we assume that the guards are syntactically mutually exclusive; hence, selection statements are deterministic.

A CSP- program can contain multiple processes in parallel, divided by \( || \). For example, processes \( p_1 \) and \( p_2 \) run in parallel in \( p1 || p2 \). In CSP-, \( || \) is not used inside a process; thus, a single process is always sequential. Note that all CSP- processes do not share variables; thus, inter-process communication can only be performed by message-passing via channels. All CSP- variables and channels are declared prior to the programs and are omitted for simplicity.

C. Microarchitectural Optimizations

By factoring out ISA details into black-box functions, we specify the golden model of a processor using the following CSP- program.

\[
\text{pc} := \text{init}; \\
\ast [ \text{instr} := \text{imem}[\text{pc}]; \delta := \text{EXEC}(\text{instr},\text{pc},\sigma); \text{pc} := \text{PCUPDATE}(\text{instr},\text{pc},\sigma); \sigma := \text{UPDATE}(\sigma, \delta) ]
\]

In this description, \( \sigma \) captures the entire state of the processor, and \( \delta \) is used to capture the state update that must be performed. \( \text{imem} \) is an array corresponding to the instruction memory, \( \text{pc} \) is the value of the program counter, and \( \text{instr} \) is the instruction to be executed.

The skeleton of a microprocessor can be specified in a few lines as shown above: it simply loops forever, reading an instruction from the program counter, executing the instruction and updating the program counter and the state of the machine (such as memory and registers) accordingly. The functions \text{EXEC}, \text{PCUPDATE} and \text{UPDATE} are ISA-specific. These functions can remain black boxes when we apply certain microarchitectural optimizations.

The program in Fig. 2 describes a two-way superscalar microprocessor, with the assumption of \text{PCUPDATE} that the length of an instruction is the constant \( \text{WL} \) Bytes. Recall that in the guarded selection expression \[ e_1 \rightarrow p_1 \] \[ \ldots \] \[ e_n \rightarrow p_n \], an entry \( p_i \) is selected and proceeded with when \( e_i \) holds.

As we can see from the example, a two-way superscalar processor reads two instructions at a time and checks if it is possible to run the two instructions in parallel (specifically (i) the first instruction is not jumping or branching and (ii) the two instructions have no data dependency). If it is possible to run them in parallel, the processor issues both concurrently. Otherwise, the processor falls back to the sequential version.

In general, a designer would perform a large number of such re-writes to obtain the final concurrent CSP- description of the micro-architecture of the hardware design.

D. Array Decomposition

We use the term dynamic array access to refer to an array operation where the array index cannot be determined statically during synthesis (or, to use a software analogy, at compile time). These accesses are not automatically synthesized in AVLSI flow, because dedicated, pre-verified circuit implementations (memory macros) are much more efficient, especially when the array size is large. In our processor case, both the register file and the memory have dynamic array accesses. To solve this problem, we apply a technique called array decomposition (an example is shown in Section IV-D), which turns all array indices into constants. After this step, all array accesses will be factored out into separate memory processes.

E. Verifying the Rest of the Design Flow

In the AVLSI flow, CSP- programs can be translated into handshaking expansions (HSE), and those are further translated into production rules (PRS), which are a representation of logic gates [4]. While this is not the focus of this paper, we briefly summarize existing approaches for verifying those parts of the AVLSI flow.

The correctness of an HSE program against a PRS program can be model-checked by previous techniques [9]. The syntax-directed decomposition approach mentioned above uses a fixed set of building blocks; hence this final HSE/PRS verification is independent of the CHF program and has to only be completed once.

From PRS, automated layout tools can generate the physical layout of circuits via gate-level technology mapping. Commercial tools for layout-versus-schematic (LVS) and manufacturing design rule checking (DRC) are used to verify that the geometry meets the manufacturer’s guidelines, and matches the gate-level PRS description, completing the verification flow.

III. Translation Validation for CSP-

In this section, we describe the validation of CSP-level microarchitectural optimizations. At CSP-level, designers
adopt a range of techniques to optimize the chip design. They can be surprisingly similar to software optimizations, such as pipelining, loop unrolling and loop distribution. Based on this observation, we design a translation validation tool to formally verify the correspondence between two CSP programs. We adopt an approach that is similar to state-of-the-art translation validation tools [10], [11], [12]. In particular, our tool extends the algorithm introduced in PEC [10]. The goal is to automatically show a bisimulation relation between two control flow graphs (CFGs) with a facilitating SMT solver. We have two unique challenges that software validators do not face:

1) Our target programs are typically reactive and non-terminating. Due to their reactive nature, we need input from the users to facilitate the verification process, which corresponds to providing hints for matching loop invariants between two programs.

2) Our target language is explicitly parallel with message-passing on communication channels. Since we are dealing with parallel programs, we parse each process in the program into a CFG and merge all of them via synchronization points defined by channels into a graph that represents the whole program.

The original PEC algorithm (developed for software) cannot handle non-terminating, reactive programs or concurrency. Our extensions permit the use of translation validation in the context of parallel CSP programs.

A. Equivalence

We need an equivalence definition that accommodates the fact that most programs that describe hardware are reactive. The trivial “return value equivalence” used in terminating software systems is not enough. Instead, we rely on trace equivalence, which defines the equivalence of observable behaviors in reactive programs [13]. Moreover, we define an \textit{\nu-state trace equivalence}, where \textit{\nu} is the set of variables whose state we observe. An \textit{\nu-state trace t} is defined as a (possibly infinite) list where a new program state \( \sigma_v \) is appended to the tail if it is not equal to the current head. For example, the \( \{x\} \)-state trace of the program \( x := 1; x := y + 1; x := 2 \) with an initial program state \( \{x \mapsto 0, y \mapsto 0\} \) is \( \{x \mapsto 0, \{x \mapsto 1, \{x \mapsto 2\}\}\} \)—the state \( y \) is ignored as it is not part of the observing variable set \( \{x\} \), and the effect of assignment statement \( x := y + 1 \) is not recorded as it does not change the value of \( x \).

In Section IV, we will show that memory-trace equivalence—i.e., \textit{\nu}-trace equivalence where the variable set corresponds to the memory—can be used to compose a meta correctness theorem about the processor in our superscalar processor example.

B. Bisimulation

Our validator tries to establish a “matching” relation to relate the original program and the transformed program. This relation is called \textit{bisimulation}. Before we give its rigorous definition later in Definition 2, we need to introduce several auxiliary definitions.

A control-flow graph (CFG) [14] is a standard compiler technique [15] to abstract a sequential program’s logic from its specific syntax. Each CSP process can be translated trivially into a control-flow graph (CFG), as they are sequential: recall that concurrency does not exist inside a CSP process (however, CSP programs do have inter-process concurrency, whose verification is tackled by the technique introduced in Section III-D2). An example of a CSP process and its CFG is shown in Fig. 3. We use \( \pi \) to refer to CFGs. In this section, we use “program” and “CFG” interchangeably.

A location is a specific point that a program is at during its execution. When a program begins, it is at its initial location, which we refer to as \( i \). Further, we define anchor locations to be locations that are used to “match” two programs in verification: since CSP programs can be non-terminating, we need to pinpoint certain “matching” locations respectively in two programs to prove their equivalence. Labels are the CSP-syntax used to refer to locations. For example, in Fig. 3, the colored “a” and “b” are labels. In CSP-1, labels are used to mark anchor locations.

A successor relation for anchor locations is defined to be: \( l \xrightarrow{\text{path}} l' \) holds iff \( l \) and \( l' \) are both anchor locations and there is a path in the CFG from \( l \) to \( l' \) such that none of the locations on the path except end points are anchor locations. For example, in Fig. 3, the path from anchor location \( a \) to \( a \) (i.e., a loop iteration) is “\( \text{pc} := \text{TGT(instr)}; \) \text{instr} := \text{imem[pc]}; \) \text{is_jump(instr)=true}”. The expression “\( \text{is_jump(instr)=true} \)” is an assumption, which is a predicate of the program state.

A correlation relation \( \mathcal{R} \) of two programs \( \pi_1 \) and \( \pi_2 \) is defined to be a set of triples \( \{(l_1, l_2, \psi)\} \), where \( l_1 \) and \( l_2 \) are anchor locations in \( \pi_1 \) and \( \pi_2 \), and \( \psi \) is a formula over \( \sigma_1 \) and \( \sigma_2 \), the states of \( \pi_1 \) and \( \pi_2 \) at \( l_1 \) and \( l_2 \) respectively.

Note that \( \sigma \) need not be the state of all variables present in the program; otherwise, the optimizations that can be validated

![Fig. 3. A CSP process of a simple CPU which only has jump and ALU instructions and its CFG following standard practice [15].](image-url)
are very limited. We use $\tau$ for the set of variables that are used in $\sigma$. A simulation relation is dependent on $\tau$. Recall that $\iota$ stands for the initial location, and thus $l_1$ and $l_2$ are the initial locations of $\pi_1$ and $\pi_2$ respectively.

**Definition 1.** ($\tau$–Simulation). A correlation relation $\mathcal{R}$ is a $\tau$–simulation relation for $\pi_1, \pi_2$ if it satisfies:

1. $(l_1, l_2, \sigma_1 = \sigma_2) \in \mathcal{R}$.
2. For any $(l_1, l_2, \psi) \in \mathcal{R}$, if $l_1 \xrightarrow{\text{path}_1} l_1'$ then there exists $l_2', \psi', \text{path}_2$ such that $(l_1', l_2', \psi') \in \mathcal{R}$ and $l_2 \xrightarrow{\text{path}_2} l_2'$ and $\forall (\sigma_1, \sigma_2, \psi'(\sigma_1, \sigma_2)) \bullet \psi'(\sigma_1, \sigma_2, \text{step}(\text{step}(\sigma, p))$ is the new $\tau$–state obtained by executing $p$ starting from state $\sigma$.
3. For any $(l_1, l_2, \psi) \in \mathcal{R}$, $\psi \implies \sigma_1 = \sigma_2$.

**Definition 2.** ($\tau$–Bisimulation). A correlation relation $\mathcal{R}$ is a $\tau$–bisimulation relation for $\pi_1, \pi_2$ if $\mathcal{R}$ is a $\tau$–simulation relation for $\pi_1, \pi_2$ and $\mathcal{R}^{-1}$ is a $\tau$–simulation relation for $\pi_2, \pi_1$. We define $(l_2, l_1, \psi) \in \mathcal{R}^{-1}$ iff $(l_1, l_2, \psi) \in \mathcal{R}$.

We establish the following theorem about $v$–state trace equivalence (defined in Section III-A), which states that a $\tau$–bisimulation (which can be validated by our tool) implies state trace equivalence of any subset of $\tau$. We define a write to a variable as a statement that may change the value of the variable (for example, in CSP, a write is either an assignment $v := e$ or a receive $C?v$). A write to a set of variables $\tau$ is a write to any variable in $\tau$. We say a location $l$ is in a relation $\mathcal{R}$ if there exists a triple $(l_1, l_2, \psi) \in \mathcal{R}$ and $l$ is either $l_1$ or $l_2$.

**Theorem 1.** ($\tau$–Bisimulation to $v$–State Trace Equivalence). If $v$ is a $v$–trace in $\pi_1$ then $\mathcal{R}$ is a $v$–trace in $\pi_2$. It is obvious that the $\tau$–bisimulation is also a $v$–bisimulation. For any $v$–trace in $\pi_1$, there must exist a sequence of $\pi_2$ locations in $\mathcal{R}$ that produces the same trace (since $\mathcal{R}$ contains the locations of all writes to $v$). We use the $\tau$–bisimulation hypothesis to find a simulating sequence of locations in $\mathcal{R}$, which is in $\pi_2$’s $v$–trace set.

**C. Checker Algorithm**

To check the equivalence of two CFGs, we adopt a simplified version of PEC’s checker algorithm [10]. Our checker algorithm is shown in Fig. 4.

1. **Core PEC Checker Algorithm:** The general idea of the PEC checker algorithm [10] is to augment a given correlation relation set $\mathcal{R}$ (by adjusting $\psi$ in each triple $(l_1, l_2, \psi) \in \mathcal{R}$) to construct a bisimulation relation. The checker first computes the set $\mathcal{P}$ that contains all the possible matching path pairs (Line 2), based on the initial correlation relation set $\mathcal{R}$. Once $\mathcal{R}$ and $\mathcal{P}$ are computed, it tries to establish a bisimulation relation (Line 4). The checker augments the condition of each matching locations so that the condition at the end location can be implied by that at the beginning (Line 25), i.e., clause 2 in Definition 1 is established for the specific $(l_1, l_2, \psi)$ triple. It repeats this procedure until either of the following happens: (i) a closure is reached, in which case it exits the loop and returns SAT; or (ii) it is trying to to augment the $(l_1, l_2)$, i.e., the matching condition at the beginning is not strong enough to imply a bisimulation relation, in which case it returns UNSAT.

WP in Line 25 is the standard weakest-precondition [16] predicate transformer, and $\mathcal{P}$ in Line 17 is the strongest-postcondition [16]. Predicates computed by these functions can be solved using SMT solvers [17], [18].

When we call $\text{Solve}$, we ask an SMT solver to decide if a formula is unsatisfiable. In Line 26, the checker asks the solver to check the negation of formula $F$: an UNSAT result means that $F$ is valid, i.e., always true.

D. **Our Contributions to the Checker Algorithm**

As we have mentioned, the state-of-the-art equivalence checking algorithms (designed for software programs) such as PEC cannot handle reactive and parallel CSP programs.

1. **User-specified matchings:** We ask the user to supply a “specification” which contains a set of matching labels and variables as hints. The user-provided specification describes the variables from each program that must match at a selected set of anchor locations. Note that the case of terminating
programs is simple—*return values* must match at program exit points. Hence, in the case of PEC, this step can be automated [10]. The reactive nature of CSP- programs requires the user to provide this information. Moreover, as we have mentioned in Section III-B, the matching relation often does not cover all variables in the state—which allows for more types of optimizations.

From the user-supplied specification, we derive the initial relation $R$. The user needs to label anchor locations (defined in Section III-B) in the program by adding explicit labels in our system like in Fig. 3, and specify the matching location pairs and variables (an example can be found later in Section IV-B in our case study).

2) Extensions for parallelism: PEC is not designed for parallel programs, as they cannot be converted into one single CFG easily: recall that each sequential process corresponds to a CFG and that a parallel program has multiple processes (and there is no “standard practice” to turn a parallel program into CFG(s), which we discuss in Section V). To handle concurrency-introducing transformations, we introduce a new technique called CFG merger that permits us to extend the use of the checker to parallel programs with synchronization.

We extend our algorithm to support parallel programs by implementing a merger of CFGs. Most works on concurrent CFGs try to establish a new formalism as the software languages they target are more complicated, but since we are targeting HDL programs and the relative simplicity of CSP-, we can just *merge* multiple CFGs into one.

a) CFG Merger: We can merge CFGs of CSP- processes into one for a *subset* of CSP- programs with an extension to the validation framework we described so far. The subset condition corresponds to processes that have a *global synchronization point*. The framework is extended by replacing the notion of a path between two labels with a directed acyclic graph (DAG) between two labels that represent concurrent paths.

For parallel CSP- programs, we do not examine arbitrary interleavings of paths—that would be too costly. Instead, we *structurally merge* multiple CFGs into one with a concise representation of parallel paths. Since (i) only channels are used to communicate between processes, and (ii) all variables are local, assignments in different processes are non-interfering, simplifying the state computation of merged paths. We mark that this non-interfering property is not a restriction or requirement of the merger but a result of CSP- language disallowing shared variables (thus, all processes in a CSP-program are data-independent).

The algorithm for merging two CFGs is shown in Fig. 6. The function CollectSameChannels tries to collect the paths in $\pi_2$ from $l_2$ the same collection of channels with the same occurrences. The function also pruning all the infeasible paths similarly as in Fig. 4. Fig. 5 shows an example of MergePaths. Note here since we are always merging an existing CFG $\pi_1$ with that of a CSP- process $\pi_2$, we can assume that $\text{path}_2$ is always a trivial DAG, i.e., a linear path. Also, whenever $\text{MergePaths}$ computes a cyclic graph, we detect a deadlock in the CSP-program.

![Fig. 5. Merge CFG paths. Each path is a DAG. $D_1$ and $D_2$ are first merged into $D_1||D_2$. Then $D_1||D_2$ and $D_3$ are merged into $D_1||D_2||D_3$. Note that when two communication vertices are merged into one assignment vertex, the new vertex inherits both communication vertices’ predecessors and successors.](image)

![Fig. 6. The merger algorithm. The expression $\{l_1 \neq l_2\}$ is the new name of a location by merging $l_1$ and $l_2$. The function $\text{MergePath}$ is shown empirically in Fig. 5.](image)

We propose an extension of weakest-precondition ($WP$) and strongest-postcondition ($SP$) computation to handle paths that are DAGs. In Fig. 4, the computation of $WP$ or $SP$ of a path is well-defined when the path is a sequence of instructions but not so when a path is a DAG. To solve this problem, we pick one topological order of the DAG $l$ (which is a sequence of instructions) and define the $WP$ or $SP$ of a DAG to be that of $l$. This practice is theoretically sound. We know that all topological orders of the DAG have the same $WP$ and $SP$ because they are the legal interleavings of CSP-processes which are data-independent, i.e., assignments in different processes are non-interfering. More formally, the data dependencies of instructions are captured by the edges in the DAG—any two instructions in the topological order that can be swapped (i.e., without violating the partial order) are data-independent (thus having the same $WP$ and $SP$), and a finite number of these swaps can change a topological order to any other, which means any two topological orders must have the same $WP$ and $SP$.

b) Global Synchronization Point: Any labels of a parallel CSP-program must be a *global synchronization point*. Particularly, the label $l_1 \neq l_2$ in Fig. 6 must be reachable in all possible parallel executions of the program. This usually means one label is on a pending communication which serves as a barrier in the parallel execution. Not all CSP- programs meet such requirements. However, in our experience, we
find that programs used to describe asynchronous circuits typically meet this constraint. This is because we usually derive a parallel version of a CSP- program from a sequential one. Thus, the parallel processes usually follow a fork-join paradigm which does not eliminate global synchronization points.

IV. CASE STUDY: VERIFICATION OF A MICROPROCESSOR

In this section, we demonstrate the verification-driven design using the example of deriving a superscalar design from a sequential golden model. We also formally verify the refinements in this section using the validator discussed in Section III. The memory-trace equivalence introduced in Section III-A (using the memory as the variable in the equivalence) allows us to reschedule reads of the memory in the superscalar optimizations. We implement the validator in C++ with around 4.4K lines of code (LOC). We use Z3 [19] as the external SMT solver.

A. Golden Model

We choose the user-level RISC-V 32-bit base integer (RV32I) ISA [20] as our target ISA and implement a sequential CSP- program for it. We implement all 40 instructions in RV32I with only M-mode. Note that our design and verification flow is mostly independent of the choice of ISA.

Our golden model is shown in Fig. 7. Note that all functions have an implementation and are eventually expanded and inlined before synthesis. But for validation, most can be treated as abstract unless they are necessary for certain steps, such as the step for register/memory access optimization where we eliminate unnecessary reads to register and memory.

Our design for superscalar execution is to conditionally issue the second instruction and unconditionally write back. One could also imagine a design that unconditionally issues and conditionally writes back, i.e., a speculative design.

We remark that our golden model is at a higher level than any n-way superscalar execution optimization. Table I summarizes four stages in introducing superscalar execution. We omit the detailed CSP- of each refinement step due to page limitation and use pseudocode to show the idea instead. Their LOCs are reported in Table I, which does not include ISA-specific function definitions around 200 LOC in CSP-.

B. Microarchitectural Optimizations

Instead of checking the final result against the golden model, we approach the verification in a stepwise fashion. In each refinement step, we only introduce one type of optimization so that it is easier for the user to supply the auxiliary information as we explained in Section III-C. An example of user-supplied information is shown in the following.

```c
"p1,p2":{"input": [ "pc_init", "dmem", "reg" ],
"matchings": [ "l,l": [ "dmem", "reg", "pc" ]]
```

Here p1 and p2 are the name of a CSP- program. The two l are a label in p1 and p2, respectively. The names such as dmem, pc are the matching variables. In all of our transformations, the number of matching locations needed in the specification does not exceed 3. We briefly introduce two sets of optimizations next and show the runtime of different optimizations in Table I. Note that our approach is general for any n-way superscalar execution optimization.

C. Introducing Superscalar

We use the validator introduced in Section III to verify that the superscalar optimization is equivalent to the golden model. We have shown a skeleton of superscalar design result in Section II-C previously. Table I summarizes four stages in introducing superscalar execution. We omit the detailed CSP- of each refinement step due to page limitation and use pseudocode to show the idea instead. Their LOCs are reported in Table I, which does not include ISA-specific function definitions around 200 LOC in CSP-.

1) Loop unrolling: We unroll the CPU loop once. Its general idea is shown below.

```c
pc := pc_init;
*[
  LOOP_BODY1;
  [issue_second -> LOOP_BODY2
   [] else -> skip ]
]
```

Here the two loop bodies are identical except for the variable issue_second is computed in LOOP_BODY1.

Note that this is not a generic two-way loop unrolling where the loop body is repeated twice. In our case, the second
loop body is conditional, which corresponds to the conditional issuing of the superscalar design.

2) Instructions reordering: In this refinement step, we move the FETCH and DECODE in the second loop body next to the first ones and move two execution units next to each other in the selection entry where we can issue_second. This refinement step allows us to isolate the execution units.

We start from the unrolling result of the previous step, and we expand each LOOP_BODY into four stages for demonstration purposes.

```
pc := pc_init;
*[
  FETCH1; DECODE1; EXEC1; WRITE_BACK1;
  [issue_second ->
    FETCH2; DECODE2; EXEC2; WRITE_BACK2;
    {} else -> skip ]
]
```

This program is refined into the following.

```
pc := pc_init;
*[
  FETCH1; FETCH2; DECODE1; DECODE2;
  [issue_second ->
    EXEC1; EXEC2; WRITE_BACK1;
    WRITE_BACK2
    {} else -> EXEC1; WRITE_BACK1]
]
```

The correctness of this refinement step depends on the condition issue_second, since we swapped EXEC2 and WRITE_BACK1 in the superscalar path. It is the designer’s responsibility to design this optimization, i.e., to choose issue_second to be a value under which EXEC2 and WRITE_BACK1 are non-interfering.

3) Register/memory access optimization: In this step, unnecessary register and memory reads are eliminated. For example, a RISC-V immediate add instruction addi does not read rs2. In the golden model, the processor reads the rs2 unconditionally. In this refinement step, the read is dropped. We do not choose to implement this optimization in the golden model because the golden model should contain minimum code that is sufficient for describing functionality.

This refinement step, unlike others, is ISA-dependent: we need to supply the implementation of the RISC-V execution unit and its ISA encoding so that the validator can verify that the elimination of redundant operations does not affect functional correctness.

4) Projection: Projection [21] is a well-known design technique in asynchronous VLSI. Using this technique, we project the two execution units to two separate processes so that they run in parallel.

```
pc := pc_init;
*[
  FETCH1; FETCH2; DECODE1; DECODE2;
  [issue_second ->
    EXEC1; EXEC2; WRITE_BACK
    {} else -> EXEC1; WRITE_BACK1]
]
```

The program above is refined into the following, where EXEC1 and EXEC2 runs in parallel. The correctness of this refinement step depends on the non-interference of two execution units.

```
p := pc_init;
*[
  FETCH1; FETCH2; DECODE1; DECODE2;
  [issue_second ->
    EXEC1; EXEC2; WRITE_BACK
    {} else -> EXEC1; WRITE_BACK1]
]
```

Note that CSP program does not allow shared variables, so even one identical name may occur in multiple processes, they are all local. The validation of this refinement step requires the merger extension we introduced in Section III-D2. A similar but simplified example was demonstrated in Fig. 5.

In general, projection can be done on two sets of disjoint variables and introduce concurrency to a CSP program. To the best of our knowledge, the correctness of this technique has been proved on paper [21] but has not been formally verified. We use our extended translation validation approach to formally verify projection transformations. While we do not detail this here, projection can also be used to reason about other micro-architectural transformations (e.g., FETCH-DECODE-EXEC-WRITE_BACK pipelining [5], [21]).

5) Summary: Our tool allows both interpreted and uninterpreted functions and provides the flexibility to switch between them. While the register/memory access optimization step is ISA-dependent, other parts where we can just omit the details of execution and use uninterpreted functions are not.

We remark that design choices are made through stepwise refinements. For example, an alternative speculative design that unconditionally issues both instructions would need to introduce a commit_second variable to guard WRITE_BACK2. This speculative idea is applicable to optimizations like branch prediction.

D. Array Decomposition

We decompose instructions that involve dynamic array accesses (Section II-D) into separate modules, which include the register file and the memory, to make our final design synthesizable in AVLSI flow. Using the register file as an example, we first delegate all reads and writes to a separate process shown below and then replace the original occurrences with the corresponding sequence of communications (for example, reg[n] := d is replaced with REG_NUM!:n; REG_WR!true; REG_W_DATA!:d; REG_W_ACK!).

```
* [REG_NUM?:num; REG_WR?is_write; REG_W_DATA?:wdata; 
  [is_write -> reg[num] := wdata; REG_W_ACK! 
  [{} else -> REG_R_DATA!:reg[num] ]]
```

Then we use a macro script to expand the array to eliminate dynamic access. For example, a register file of size 2 will be expanded to the following.
TABLE I

<table>
<thead>
<tr>
<th>Optimization Name</th>
<th>Time(s)</th>
<th>Step #</th>
<th>Final LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial program</td>
<td>N/A</td>
<td>N/A</td>
<td>23</td>
</tr>
<tr>
<td>Loop unrolling</td>
<td>0.5</td>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>Reordering</td>
<td>6.5</td>
<td>7</td>
<td>62</td>
</tr>
<tr>
<td>Reg./Mem. Optimization</td>
<td>3.6</td>
<td>1</td>
<td>75</td>
</tr>
<tr>
<td>Projection</td>
<td>3.2</td>
<td>1</td>
<td>115</td>
</tr>
<tr>
<td>Array Decomposition</td>
<td>25.4</td>
<td>1</td>
<td>141</td>
</tr>
</tbody>
</table>

* [ REG_NUM?num; REG_WR?is_write;
REG_W_DATA?wdata;
[ num=0 -> [ is_write -> reg[0] := wdata;
   REG_W_ACK! ];
   [ else -> REG_R_DATA!reg[0] ] ]
[ num=1 -> [ is_write -> reg[1] := wdata;
   REG_W_ACK! ];

Our final CSP- program contains the following modules after the microarchitectural optimizations:

`MAIN||EXEC1||EXEC2||REG_FILE||MEMORY`

E. Meta Theorem

We establish following meta theorem.

**Theorem 2.** The golden model is memory-trace equivalent to the final implementation.

**Proof.** A successful validation run establishes memory-trace equivalence relation if we always include `dmem` as a matching variable. All the transformations used in our microarchitectural optimizations satisfy this condition.

F. Discussion

Our approach uses memory decomposition and an explicit CSP- memory description. Memory is usually designed as a separate module apart from the microprocessor. Nonetheless, we decompose the memory (Section IV-D) since our meta theorem is stated on memory contents. One can imagine an alternative approach where we separate the memory and verify the microprocessor based on an equivalence defined by channel communications to the memory module. Note that if memory access patterns are re-ordered by the superscalar transformation, then this would require significantly more complex formalisms to state a meta theorem about the golden model and detailed design (for example, one needs to specify a memory consistency model to begin with).

We also note that while our work is focused on asynchronous design, the techniques we have introduced can also be used for clocked circuits that use ready/valid signals and use latency-insensitive interfaces throughout.

V. RELATED WORKS

a) Model Checking: Model checking is a popular technique used in formal verification of hardware. Often this approach is applied to specific circuits instead of parameterized ones, for example, verification of processors using model checking [22, 23]. An abstract model of the circuit is hand-created, and temporal logic specifications are written to verify that the abstract model meets certain requirements [24]. These works are analogous to the HSE/PRS model checking [25] in an AVLSI flow. We adopt translation validation [26] on a much higher level.

CSP-derived languages have been cooperated into automatic checkers, most notably CSP_M used in FDR [27]. FDR checks refinement based on concrete models. It can be extended to be general if the CSP program is data independent [28] (which a CPU is not). Our checker adopts the validation technique that depends on an SMT solver, which allows abstract data/function and enables parameterized validation inherently.

b) Hardware Verification Through Theorem Proving:

In this section, we identify a series of works on hardware verification through theorem proving. We make two general remarks on the differences between these works and ours: (i) our work targets microarchitectural optimizations, which is at a higher-level than these works; (ii) we use translation validation rather than theorem proving, allowing us to verify user-specified transformations, not just pre-defined ones.

Recent works on modular verification of hardware are described in the Bluespec language [29, 30] and in a subset of Verilog [31]. These approaches focus on the verification of synchronous circuits, while our work is focused on a different, asynchronous VLSI methodology.

FM9001 [32] and VAMP [33] are two formally verified processors through theorem proving, both of which leverage synchronous design methodology. While our work also has a processor example, we focus on a general method for compilations. We are also at a higher level than these works: they start from a “detailed design” which is what we target.

Recently, theorem-proving was also used in asynchronous circuits verification. Longfield et al. verified that CHP descriptions of chips satisfied certain basic safety properties [25]. This is different from our work and other works discussed in this section, all of which focus on functional correctness. Chau et al. have done a series of works on the formal verification of asynchronous circuits through theorem proving using ACL2 [34, 35]. While these works also focus on functional correctness, they use the DE system to describe asynchronous circuits and target a lower-level phase in the design flow.

c) High-Level Synthesis: High-level synthesis (HLS) adopts a seemingly similar methodology to the microarchitectural optimizations procedure in our flow. A series of works is done on formal verification of HLS [36, 37, 38, 39]. However, HLS and microarchitectural optimizations in our flow are fundamentally different. First, the high-level languages used in HLS are *software* languages such as C and C++, whereas our approach starts from CSP-, which is a *hardware* description language that is a better model for the physical implementation. Second, the HLS tools generate the hardware implementations automatically, whereas our ap-
proach verifies the designer-specified detailed hardware model. Therefore, our approach enables the designer to make manual microarchitectural level optimizations and verify them, which HLS tools do not.

d) Compiler Verification: One could also draw a similarity between our work and any software compilation/optimization verification works using translation validation [26], from which we took inspiration. But since we are dealing with a different set of languages that are hardware-oriented, we have major differences with the software compilation verification work. Most notably, we are more interested in reactive and parallel behaviors. We also focus on message-passing protocol, which is different from the shared-memory-based concurrency model that is commonly explored in software compilation verification.

We are also inspired by the method of stepwise refinement [2], [3], which we combine with formal verification method for what we call verification-driven design. A work on using refinement for hardware design has been done on communication protocol [40]. They target a very specific example in synchronous circuits. We target a more complicated example and demonstrate the generality of this method in asynchronous design.

e) Concurrent CFGs: Most works on concurrent CFGs focus on developing a new formalism for a set of CFGs instead of merging them into one, such as threaded CFG [41] and Parallel Program Graphs [42]. Moreover, works on data-flow analysis [43], [44] also adopt the idea of using a DAG to represent information from a CFG. However, these works are on languages where processes communicate through shared variables. Thus, they use more complicated techniques to solve the problem. We are able to approach the concurrent CFG problem by simply merging CFGs because our model is simpler: CSP has no shared variables, and processes communicate through channels. Thus, for example, we can simply merge two vertices of matching communications instead of introducing extra mechanisms such as synchronization edges [42].

VI. CONCLUSION

In this paper, we presented our work on the verification framework for asynchronous VLSI and demonstrated it by formally verifying a processor. Due to the difference between high-level microarchitectural optimizations, which involves a large set of stepwise transformations, we adopt translation validation for its formal verification. Specifically, we propose verification-driven design for hardware design through stepwise refinement and its verification. We designed and implemented a validator and validated the superscalar execution optimization of a processor.

We plan to use our framework to verify more complicated hardware designs, such as a state-of-the-art processor with interrupts and exceptions support. We also plan to use our validator to verify more microarchitectural optimizations, such as virtual memory and cache.

ACKNOWLEDGMENT

This work was supported in part by DARPA IDEA grant FA8650-18-2-7850, and in part by DARPA POSH grant HR001117S0054-FP-042.

REFERENCES