`Pep9Milli`: Verification of CISC Processors using Software Verification Tools

Author(s) information removed for review

Author(s) information removed for review

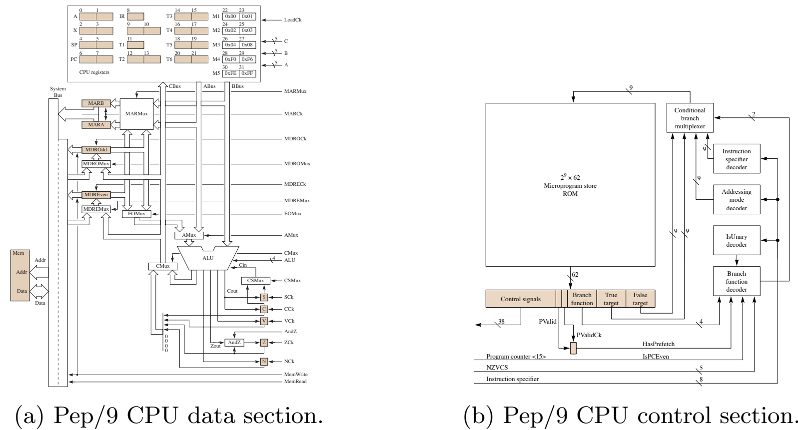
Abstract. This paper introduces a process for using open-source software verification tools to verify the correctness of CISC microprocessors. The three-part process introduces a new language (*millicode*) at a higher level than microcode. Millicode is compiled to a high-level language (i.e., `C`) as well as microcode. Software verification tools verify that the high-level language representation meets the processors's RTL. The process is demonstrated by verifying an existing CISC processor, Pep/9. A partial millicode implementation of Pep/9's RTL is created and verified using `Klee`.

Keywords: Microcode · Microcode Verification · Processor Verification · Microprocessors · Symbolic Execution · Pep9 · Klee.

1 Introduction

This paper introduces a process for using software verification tools for complex instruction set computer (CISC) processor verification, which allows testing early in the CPU design phase and supports CPUs lacking a hardware description. In the realm of open source, tools primarily verify Verilog and SystemC descriptions of processors [40, 49]. For CISC processors containing *microcode* (an additional level of abstraction between logic gates and the instruction set architecture) there is a dearth of open source verification tools. Companies like Intel, Centaur, IBM, and others have proprietary verification tools able to verify microcode and hardware layers [29, 30, 48]. Such proprietary tools (like Intel's MicroFormal [2,3])) are not widely available. Neither approach is tenable for microcoded, open source, CISC processors lacking hardware description language (HDL) implementations.

Introducing a new language at a level of abstraction above microcode allows software verification tools to be applied to the hardware domain. The new language, termed *millicode*, represents an off-chip layer of hardware control. Millicode is designed for ease-of-compiling to an on-chip hardware control language. It compiles to a high-level language (like `C`) with existing software verification tools. The high-level language description of the millicode program may then be fed to a software verification tool, transitively verifying the processor implementation.



(a) Pep/9 CPU data section. (b) Pep/9 CPU control section.

Fig. 1: Organization of Pep/9, the processor being verified[2].

Pep/9 is a pedagogical virtual machine described in *Computer Systems*, fifth edition [46], and was recently extended to include a complete microprocessor implementation [33]. This paper describes the `Pep9Milli` project[1] to demonstrate the implementation of this process. `Pep9Milli` uses a software verification tool (`Klee`) to verify the correctness of the Pep/9 processor via millicode. 75% of the Pep/9 processor has been verified to meet its specification using this process.

Pep/9 is a 16-bit CISC processor, which features expanding opcodes that are either unary (one byte) or non-unary (three bytes), and is designed as an instructional tool to teach computer systems and organization concepts to undergraduates [46]. *Computer Systems* describes the physical organization (figure 1) of the Pep/9 microprocessor as well as the semantics of the assembly language. The full Pep/9 microprocessor has passed many practical unit tests in the `Pep9Micro` CPU simulator, but the underlying processor microcode has not been previously verified—formally or functionally.

This paper and the `Pep9Milli` project offer the following contributions to the academic community and the Pep/9 ecosystem:

- Demonstrates how an additional hardware language facilitates processor verification.
- Describes *millicode*, a new hardware control language for Pep/9.
- Introduces a verification framework for millicode written in `C`.
- Verifies a millicode implementation of the Pep/9 processor using the software verification tool `Klee`.

Section 2 evaluates microprocessor designs from industry and introduces Pep/9. Section 3 explores related work on processor verification. Section 4 discusses levels of abstraction in verification. Section 5 describes how millicode

[1] Source code available at: link removed for review
[2] Figures adapted from [32], used with permission.

uses software verification tools to perform verification. Section 6 uses millicode to verify the correctness of the Pep/9 processor. Section 7 examines the validity of the verification framework. Section 8 discuss results of verification. Section 9 suggests directions for further research.

2 Background

This section evaluates microprocessor design approaches from industry. It describes Pep/9's microcode language, and motivates the introduction of a new hardware control language, millicode. Lastly, it explores how Pep/9's microcode implements its instruction set, facilitating a millicode implementation of the ISA.

2.1 Microprocessor Design

CISC CPU's (like Pep/9) are controlled through *microcode*, which is a hardware-level, machine-dependent control language that coordinates various functional units of a CPU to perform computations [14, 15, 39, 45]. Each CISC assembly language instruction is implemented via one or more *microinstructions* [11, 14]. Code reuse is facilitated between similar instruction implementations by decomposing a single assembly instruction into sequences of microinstructions [5].

A single microinstruction is composed of multiple *micro-operations*, each of which directs individual CPU resources or functional units for the duration of an instruction [1]. These micro-operations may reference the state of CPU, like status registers. For example, the x86 instruction `JZ` branches if the Z flag is set [42], therefore some microinstruction implementing `JZ` must evaluate a conditional branch based on the value of Z. Dependence on CPU state to determine the outcome of a microinstruction is termed *residual control* [15]. Reliance on previous state makes processors more difficult to verify since the verification model must extend beyond microcode to include processor state [45]. Due to residual control, verification requires a holistic view of the system; both hardware state and microcode semantics must be modeled.

Interposing an additional level between microcode and hardware (often termed *nanocode* or *millicode*) streamlines and structures code at both levels of abstraction [21, 30, 45]. In a multi-level control scheme, a single assembly language instruction is implemented by one or more microinstructions, each of which in turn is implemented by one or more *nanoinstructions*. Extra design overhead may be incurred specifying interactions between microcode, nanocode, and hardware, but practical experience with Nanodata's QM-1 architecture indicate additional abstractions ease formal verification attempts [45].

As the technology implementing logic gates has developed, microprocessor organization has evolved. Access time to microcode ROM used to be very high [23]; thus incentivizing schemes that maximized microcode ROM utilization. These schemes included overlapping microcode execution (*parallel microcode*) and highly compressed microinstruction formats [39]. Overall monetary cost of early computers encouraged *dynamic microprogramming*, which allowed end

```
is_fetch_e: A=6, B=7, MARMux=1; MARCk
// Initiate fetch, PC ← PC plus 1.
MemRead, A=7, B=23, AMux=1, ALU=1, CMux=1, C=7; SCk, LoadCk
MemRead, A=6, B=22, AMux=1, CSMux=1, ALU=2, CMux=1, C=6; LoadCk
MemRead, MDREMux=0, MDROMux=0; MDRECk, MDROCk
// IR ← MDREven, T1 ← MDROdd, PrefetchValid ← true
EOMux=0, AMux=0, ALU=0, CMux=1, C=8; LoadCk
EOMux=1, AMux=0, ALU=0, CMux=1, C=11, PValid=1; LoadCk, PValidCk;
goto end_is_fetch
```

Fig. 2: Pep/9 microcode fragment incrementing the `program counter` (PC) by 1. Neither the reference to PC nor the increment size are obvious.

users to modify a chip's microcode after fabrication [15, 43]. Improvements in ROM speed [25] and reduced expense [23] have proven user microprogramming untenable, and it never gained widespread adoption [5]. For better or worse, microcode has become highly proprietary, read-only, and only temporarily patchable at runtime to fix bugs and security vulnerabilities [11, 31].

2.2 Pep/9 Microcode

Only the CPU's data section was described in the original Pep/9 specification [46]; the control section was omitted, meaning the processor was only capable of executing code fragments without branches. Later works implemented the missing control section from the Pep/9 specification [32, 33]. The control section augments the microcode to allow for conditional and indirect branches. Using this augmented microcode, a complete implementation of the instruction set architecture (ISA), was written in microcode [33]. Design of a control section and microcode program unites various levels of abstraction within the Pep/9 virtual machine. However, the microcode is unverified, and provides no formal guarantee of correctness. Appendix A provides additional details on Pep/9's architecture and organization.

Pep/9's microcode directs individual circuits to route data through the processor to specific registers. Additionally, every microcode instruction explicitly specifies a branch (either conditional on processor state or unconditional) to successors instructions. Microcode is complex to reason about, as off-by-one arithmetic errors are not immediately visible. It contains only register numbers, not recognizable names or constant values, compounding the difficulty in finding errors. Additionally, microprogrammers must mentally track memory bus state; state which may change between cycles. A microcode program demonstrates the complex process of incrementing the program counter in figure 2. Microcode allows for the activation of nearly arbitrary combinations of circuits, which leads to high degrees of micro-operation parallelism. However, this power and flexibility comes at the cost of legibility. Other papers containing (short) microcode snippets appear to corroborate that microprogramming is non-trivial for other architectures [2, 19, 27, 28].

Pep/9 is accessible for user microprogramming, making it unusual amongst modern processors. User microprogramming allows exploration of low-level CPU

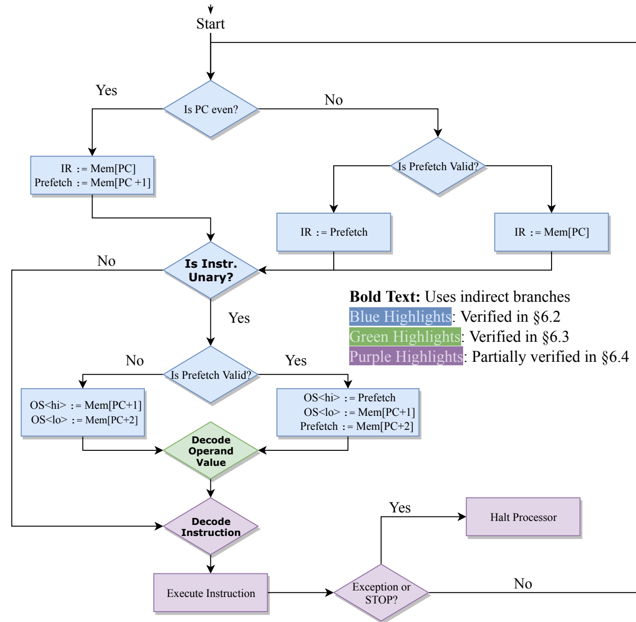


Fig. 3: Organization of Pep/9 instruction cycle.

details through examples ranging in complexity from adding two registers together to implementing a new instruction set. However, microcode is difficult validate as no off-the-shelf tools that integrate with it.

2.3 Pep/9 Instruction Cycle

Most computers implement individual instructions in their instruction set using a sequence of *fetch—decode—execute* stages [23]. These collective stages are termed the *instruction cycle*. Pep/9 implements the same three-stage instruction cycle to implement a single instruction in microcode. These stages repeat indefinitely until an exception is raised or a `STOP` instruction is encountered.

These three stages require 344 lines of microcode in total to implement. Figure 3 shows a high-level block diagram of Pep/9s instruction cycle[3]. No existing verification tools, either open source or proprietary, target the Pep/9 architecture. This paper describes the development of a new verification environment for the Pep/9 processor.

[3] To avoid a lengthy discussion about the implementation, those curious are referred to sources describing the processor [32, 46].

3 Related Work in Processor Verification

This section explores a history of processor verification, focusing on promising approaches from Rockwell [20, 34]. It also provides a high level overview of one form of verification, *symbolic execution*.

3.1 Microprocessor Verification

Since the dawn of processor development, researchers have been devising methods to verify processor correctness. Early verification attempts utilized formal methods like Hoare logic and mechanical theorem proving [9, 35, 43, 45]. Almost all of these methods were plagued by difficulties in creating generalized proof mechanisms for new processors. For example, any minor change to the Nanodata's QM-1 architecture would require the creation of a new proof/logic system [45]. Out of these early attempts, companies such as Centaur [14,18,26], IBM [17,19,30], Intel [2,3,7,16,22], and Rockwell [20,34] developed and adopted different verification methods.

Rockwell subjected two processors, AAMP5 and JEM1, to a mixture of functional and formal verification [20, 34]. These processors are relevant because they are on a similar scale to Pep/9. The Advanced Architecture Microprocessor (AAMP5) is a CISC stack machine intended for embedded systems [34]. Designed for ease of translation from high-level languages (such as Ada) to assembly language, it features a 2-stage pipeline with a 24-bit address space. The more advanced 32-bit JEM1 processor is the world's first Java microprocessor and is capable of executing any instruction specified by the Java Virtual Machine (JVM) [20]. The JEM1 is a microprogrammed computer whose ISA is a superset of the JVM.

Using the Prototype Verification System (PVS), Rockwell created formal proofs of correctness to verify portions of AAMP5's microcode [34]. AAMP5 was designed to be backwards compatible with earlier processors in the CAPS/AAMP family, which existed before Rockwell's early published attempts at verification. Miller *et al.* verified a select handful of instructions broadly representing the instruction set. When Rockwell later spun up development of the JEM1 processor, they decided to transition from formal verification to functional verification [20]. In retrospect, formal verification of the AAMP5 was incredibly costly for the marginal benefit derived from processor verification. Rockwell drew from their experience in PVS and modeled the nascent JEM1 processor in PVS. Instead of concocting formal microcode proofs, they performed symbolic simulation of the processor and its 1,689 lines of microcode within PVS. Experienced microprogrammers reviewed the output of symbolic simulation by hand and discovered multiple rarely encountered defects in the microprogram. Rockwell's semiformal verification was able to verify larger percentages of the JEM1 than had previously been accomplished using formal methods.

3.2 Symbolic Execution

Symbolic execution is a tool to explore programs using algebraic manipulations of program variables [4,9]. Tools find assertions and exceptional conditions symbolically and provide a concrete set of inputs that lead to the erroneous states [7]. Symbolic manipulation of program variables helps maximize the number of program states explored in a bounded amount of time [6]. From a user's perspective, tracing a failing condition to a concrete input allows for easier debugging.

Symbolic execution tools typically are designed to handle large state spaces efficiently [2]. But, they are a non-exhaustive form of verification. For certain classes of problems, especially those without complex branching, infinite loops, or small state spaces, symbolic execution can perform exhaustive verification. However, problems with large state spaces may not be explored exhaustively. Instead symbolic execution tools will explore as many program states as possible under time or resource constraints, prioritizing unexplored and "interesting" states. No formal guarantee of total correctness is provided in the non-exhaustive case. Tools such as `Klee`[4] are able to generate high degrees of test coverage (over 80%) for large application/library codebases exceeding 100,000 lines of code (e.g., `GNU COREUTILS` [6]). With the advancements in the field of symbolic execution over the last twenty years, tools are capable of exploring programs (and consequently, state spaces) at least an order of magnitude larger than what was attempted with the JEM1.

Based on successful verification attempts using symbolic execution, and continued advancements in the field, symbolic execution will be the tool of choice to perform verification of the Pep/9 processor. The verification goal of `Pep9Milli` is to prove that Pep/9's hardware implements *register transfer language* (RTL) specification of the Pep/9 instruction set. Before verification of the processor implementation, a new hardware control language is motivated and described.

4 Motivation For Millicode

This section explores current approaches to verifying processors via HDL implementations or analyzing microcode. Shortcomings in these approaches are discussed. Lastly, it introduces millicode as a new hardware abstraction for addressing those shortcomings.

4.1 Industry Verification at the Logic Gate Level

An approach to processor verification is to directly verify a processor's logic gate (or HDL) implementation. For microcoded processors, hybrid methods combine hardware-microcode verification have been used successfully in industry repeatedly [2,16]. Simulation across multiple levels of abstraction, including hardware-software co-verification has success in industry [19]. However, these tools will not work for processors lacking a HDL description, which occurs at early stages

[4] See https://klee.github.io/

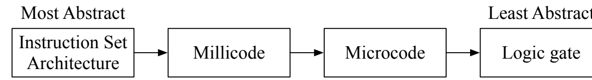


Fig. 4: Levels of abstraction in a processor. Millicode sits between microcode and the ISA.

in the design process, or for processors not meant for implementation in silicon (such as Pep/9). For these classes of processors, different methods must be developed.

4.2 Evaluating Pep/9 Microcode

Pep/9's microcode is capable of implementing a Turing-complete assembly language. However, it is difficult to develop and far more difficult to debug—difficulty aggravated by the presence of residual control. Case in point, `Pep9Micro`'s microcode[5] contains more comment lines than lines of executable microcode. Creating a verification model at this level is difficult due to high levels of micro-operation parallelism. Translation from unstructured microcode to structured high-level languages like `C` may be difficult [36].

Microcode is difficult to reason about, providing further motivation for additional layers of abstraction. Difficulty in understanding the function of microcode segments obscures the goals of verification. By reducing the burden to design control programs, enforcing more regular control flow, and reducing micro-operation parallelism, millicode provides a more suitable abstraction than microcode for large-scale hardware programming and verification. Even if the technical hurdles of modeling Pep/9 at the microcode level in a verification environment could be overcome, the goal of using layered abstractions to ease design and aid verification would not be fulfilled, indicating the need for a new level of abstraction.

4.3 Introducing Millicode

In system-on-chip (SoC) designs, simulation and verification are performed at multiple levels of abstraction [37]. For example, SoC simulations choose between multiple levels of abstraction, such as modeling bus architectures at a detailed level of individual pins and bus cycles (*pin accurate, bus cycle accurate* (PA-BCA)) or by more abstractly modeling entire bus transactions (*transaction level modeling* (TLM)) [8,37]. Higher levels of abstraction are available at early stages in system design, as they do not require finalized implementation and floorplan details. Early verification is advantageous since defects are cheaper to address when discovered earlier in the project cycle [41].

In a parallel to SoC designs, `Pep9Milli` introduces a new hardware control level, termed millicode. The millicode level of abstraction sits between microcode and the instruction set architecture (see figure 4). Just as the TLM level allows

[5] Microcode available at: link removed for review

```
MemRead(8,1)
RB9 := ADD(MDR, RB9)
MemRead(7, 1)
RB9 := ADD(MDR, RB9)
MDR := IDENT(RB9)
MemWrite(9)
```

Fig. 5: Millicode containing memory load and store operations.

for easier verification and simulation than the PA-BCA level, more abstract millicode should provide the same benefit over lower-level microcode. Millicode is easier to read, use, and verify than microcode. Multi-level control systems exist in practice [21,30,45] and have been subjects of successful verification attempts. Processor designs benefit from layering abstractions, and so should verification.

The overarching design goal of the millicode language is to abstract away complexities of the microcode implementation while retaining the same level of expressiveness and control as microcode. Each millicode instruction encodes one action, an optional symbolic identifier, and a branch to specifying the successive millicode instruction. An action is either a register-register computation, a load from memory, a store to memory, or a no-operation. Actions may induce side effects, such as modifying status flags. To perform a sequence of actions, multiple lines of millicode must be used—millicode actions may not be interleaved. Figure 5 contains a millicode sample. Millicode sacrifices one power of microcode, which is the ability to perform multiple non-conflicting computations at once. Additionally, program pre- and post-conditions are not encoded directly in millicode; they are added manually to the `C` or microcode translations of the millicode. There are currently no automated millicode translation tools; Pep/9's RTL was translated by hand into millicode, which is in turn verified using software verification tools.

Millicode is an off-chip control language; it has no corresponding on-chip control subsystem, which differs from industry [21,45]. Instead, millicode assembles to microcode. Millicode's lack of parallelism makes it more structured than microcode, and it will rely on an optimizing assembler to achieve the same levels of efficiency as microcode (see §5.2). From structured millicode it is easier to generate a structured high-level language representation, which may be fed to a software verification tool. Millicode brings additional structure to hardware control. As a hardware control language, millicode is processor-specific like a hardware description or microcode. Sitting at a high level of abstraction, millicode should be easier to create from an RTL specification than HDL or microcode. Like TLM in SoC designs allows for early verification, millicode provides the same benefit to hardware control.

5 Verifying and Translating Millicode

This section explores how millicode uses software tools to perform verification, thus verifying the processor implementation. It describes the design of the verification environment, which is constructed in the `C` language using `Klee`. It describes the translation from millicode to `C`. It highlights a major limitation of `Pep9Milli`, which is the lack of an automated millicode assembler.

5.1 Building a Verification Environment

The AAMP5 microprocessor [34] is more complex than Pep/9, but is still roughly comparable. Successes with AAMP5 indicates that it should be possible to verify Pep/9, whose design also ignored verification ease. Success with symbolic execution on the JEM1, which contains roughly four times as many microcode lines as Pep/9 [20], indicates that symbolic execution is an appropriate tool for a processor the size of Pep/9's. ISA and microarchitectural designs of AAMP5 and JEM1 parallel Pep/9, and thus the applied verification techniques (namely, symbolic execution) are prime candidates for Pep/9.

Many tools exist to perform symbolic simulation of source code programs, such as `Klee` [6], `Kite` [44], `SED` [24], and others. Each tool targets different source code languages. The entire Pep/9 program suite is implemented in `C++`, and includes a microcode simulator called `Pep9Micro`. Choosing a verification tool that operates on either `C/C++` allows easy translation of existing components from the `C++` model to form the basis of `Pep9Milli`'s verification environment. Successful verification using the `Pep9Milli` indirectly increases the level of correctness in the `C++` code base, since the implementations are related.

Within the verification environment, symbolic execution is used to explore CPU state space by initializing values symbolically for registers, memory locations, and status bits, effectively storing all possible values in each location. Individual millicode programs may constrain initial values by initializing some registers to concrete values. Using symbolic execution to execute millicode (that has been translated to `C`) symbolically explores all possible subsequent states reachable from initial conditions. Testing millicode from all possible initial states is the primary benefit derived from symbolic execution within `Pep9Milli`, and sets `Pep9Milli`'s testing apart from a limited set of unit tests. If symbolic execution terminates, it is able to exhaustively prove that a program meets its post-conditions from all starting states, proving the program correct. Showing that all starting states of the processor fulfill post-conditions derived from the RTL verifies the that the millicode correctly implements the instruction set.

The process of abstracting away the complexities of the hardware may lose important details. For example, Pep/9's memory model dictates that memory access may induce side effects on account of its memory-mapped IO system. Reading from a memory address (i.e. memory-mapped input devices) multiple times may yield distinct values. Memory-mapped IO requires repeated writes of the same value to the same memory address to be considered distinct operations. Writing to memory is not *idempotent*. `Pep9Milli`'s memory model assumes

memory access to be side-effect free, and that repeated writes are idempotent. Accessing additional or incorrect addresses in millicode programs may trigger deleterious effects by inadvertently activating memory-mapped IO components.

5.2 Translating Millicode

A major limitation of the `Pep9Milli` project is that no automated translation tool exists to convert millicode to `C` or millicode to microcode. Presently, these translations must be performed by hand. However, millicode is designed to be assembled via a future milli-assembler. The milli-assembler would emit a microcode representation for loading to Pep/9's processor in addition to a `C` representation for the purposes of verification. The `C` representation is fed to a symbolic execution engine named `Klee` (the verification process is explored in §6). Millicode is designed such that any valid millicode program has a corresponding valid microcode program. This prevents flawed analysis where the CPU's microcode must perform impossible operations (e.g., solve the halting problem) to perform the actions specified by the millicode.

In the `C` model, there is a corresponding function for each part of a millicode instruction (i.e., perform an action, update the program counter). Each milli-instruction is implemented as a `C` function, whose body contains the corresponding translation. All `C` functions are placed in a jump table. At each step, the jump table is indexed by the microprogram counter, and the associated function is invoked. This process repeats until the program terminates, the user aborts verification, or a maximum number of millicode steps has been exceeded.

Pre- and post-conditions are not directly written in millicode, as the language has no facilities to encode them (see §9). However, `Pep9Milli` provides function stubs to execute pre-conditions prior to execution and verify post-conditions after termination. These stubs must be manually filled in with pre- and post-conditions based on the source program's specification.

As millicode is an off-chip control subsystem, its verification does not immediately signal the correctness of the processor. Verified millicode must be assembled to machine-executable microcode. The resulting microcode is loaded into the Pep/9 processor, and only then is the processor known to implement the RTL of its instruction set correctly. Without loading the translation of the verified millicode, no claim about the processors correctness may be made.

5.3 Verifying Pep/9 using Millicode

Verification of the Pep/9 processor requires either verifying all three stages of the instruction cycle (see §2.3) in unison or verifying stages independently and connecting the verified segments via pre- and post-conditions. Pep/9's RTL was partially translated to millicode, maintaining the same structure as Pep/9's microcode. The microcode implementation of the instruction-operand fetch, and operand decode stages were translated to millicode in their entirety, while the microcode implementing the instruction fetch unit was only partially translated

(see layout in figure 3). Microcode instructions were split and collated by millicode operation type, which is necessary since microcode allows for parallelism while millicode does not. Data dependencies between microinstructions were respected, and common microinstruction subsequences were de-duplicated. Pre- and post-conditions for each stage were constructed from the RTL of Pep/9 [46]. The translated millicode serves as the basis for verification.

5.4 General applicability

Millicode and its verification environment as described above is designed specifically for the Pep/9 processor. The language is not immediately portable to a generic CISC with a different organization. However, the technique of creating an additional layer of abstraction to facilitate verification is portable. In SoC communications, techniques of layering abstraction have had great success recently [38], confirming older results in processor design [45].

Where this process differs from traditional abstractions is in its utilization of software verification tools to perform hardware verification. Instead of constructing a custom simulator [19] or creating a new SMT interface [3], existing software verification tools are reused. The key "trick" is creating a language that facilitates translation to a high-level language, such as `C`, and verifying the high-level representation. While this requires faith in the translation tools, verified compilation is a well-studied field [10, 12].

6 Verification Results

In this section, four programs are translated from millicode to `C` and corresponding verification results are shown. The first program computes the first 14 Fibonacci numbers using main memory to store intermediary results. This programs confirm that the verification environment is capable of verifying trivial programs, improving confidence that `Pep9Milli` can verify more complex programs. Each of the three remaining programs verify a stage of Pep/9's instruction cycle (see figure 3). Each stage—the instruction-operand fetch, operand decode, and the instruction execution units—depends on the correctness of all previous components, so they are presented in series.

Symbolic execution allows all registers and memory locations to be initialized symbolically, effectively filling each location with every possible value (see §5.1). By executing code fragments from all possible starting states, `Klee` is able to show that post-conditions hold for any possible execution of a millicode program.

6.1 Computing the First 14 Fibonacci Numbers using Dynamic Programming

The first program to be verified computed the first 14 Fibonacci numbers[6]. To confirm the correctness of the memory model, intermediary results were cached

[6] Fib(13)=233, which is the largest Fibonacci number fitting in a one-byte register.

```
      MDR   := IDENT(0)       // MDR = F(0) = 0
      MemWrite(0)             // Memory[0] = MDR
      MDR   := IDENT(1)       // MDR = F(1) = 1
      MemWrite(1)             // Memory[1] = MDR
      RB1   := IDENT(2)       // Loop variable "n".

loop: RB2   := SUB(RB1, 2)    // Fetch F(n-2).
      MemRead(RB2, 1)         // MDR = F(n-2)
      // Initial value of RB0 is F(n-1)
      RB0   := ADD(MDR, RB0)
      MDR   := IDENT(RB0)     // MDR = F(n)
      MemWrite(RB1)           // Memory[n] = MDR
      RB1   := ADD(RB1, 1)    // n = n + 1
      SUB(RB1, 13); NZ
      if LE goto next else loop

next: STOP()
```

Fig. 6: Computing the first 14 Fibonacci numbers via a loop using register-register, load, and store operations

in main memory. The algorithm constructs a 14-entry array starting at an arbitrary (i.e. `Klee` chosen) address in memory. For each Fibonacci number, the two previous Fibonacci numbers are loaded from memory, summed, and written back to the next location memory, which is a simple dynamic programming algorithm. After initializing $Fib(0) = 0$ and $Fib(1) = 1$, the remaining numbers may be computed using a loop, testing the ability of the verification model to perform conditional branching. Figure 6 depicts the algorithm.

The millicode program was translated by hand to `C`. Hand-constructed post conditions asserted that each memory addresses contained the correct values after execution. `Klee` required 63 seconds to verify that the program was correct. It executed a total of 1,457,582 statements across 29 unique paths through the program. Memory addresses in Pep/9 are words (two bytes), but the ALU only operates on single bytes. Therefore, chained byte additions are required to perform a single address computation. Upon examining `Klee`'s call tree, many of the unique paths occurred due to unsigned overflow on chained byte arithmetic. Successful verification of the Fibonacci algorithm indicates the verification model is sound using memory loads and stores, conditional branching, and multi-byte arithmetic.

6.2 Instruction-Operand Fetch

The instruction-operand fetch unit performs two jobs. First, it loads an instruction specifier from memory into the instruction register (i.e., register `IR`). If the particular instruction is unary, the fetch stage is finished. For non-unary instructions, which require a two-byte operand, an additional operand fetch is needed

to load the operand into the operand specified (`OS`) register. To determine if an instruction is unary, a special 256-entry decoder circuit ("IsUnary Decoder" from figure 1b) is used. The circuit takes in the value of the instruction specifier, and outputs a "1" if the instruction is unary or "0" otherwise. Output from the decoder is used to make a branch decision, introducing a level of indirect branching not present in the previous example. After fetching a single instruction and operand, the program terminates.

A further complication arises due to the design of the Pep/9 memory bus. Two bytes are always fetched during each memory read; however, instructions are either one or three bytes (i.e., not a multiple of 2). Memory access is slow, so the "extra" byte is placed in a special prefetch register. If a following memory access references the prefetch, the memory access is bypassed and the prefetch is used A status bit maintains the validity of the prefetch, which introduces a level of residual control persisting between ISA level instructions.

Pep/9's RTL was translated by hand to millicode, which in turn was hand-translated to `C`. Verification conditions were derived from Pep/9's RTL and textual description of the processor [46]. Verification required three seconds to complete, during which a total of 16 unique paths covering 74,382 statements were explored.

6.3 Operand Decoding

After non-unary instructions have fetched their operands, the operand must be further processed. This occurs when the operand specifies a memory address or stack offset. The operand decode unit interprets the operand for all non-unary instructions. The post-conditions of the operand decode unit are that register `T5` contains the address from which the operand value was fetched, and register `T6` contains the fully decoded operand value. After fetching a single instruction and operand, the operand is decoded, and the program terminates.

The combined instruction-operand fetch unit was augmented with millicode implementing operand decoding. Millicode for the operand decode unit was translated by hand to `C`, and the source program was manually annotated with verification conditions taken from the Pep/9 RTL. Verification of this section took nearly four hours—4,500 times longer than the previous instruction-operand fetch unit. Klee found 4,026 unique paths through the program by executing 13,378,289 statements. State space explosion most likely occurred to the addition of multiple level of indirect lookups and pointer arithmetic. The verification of the first seven addressing modes took nearly an hour, but addition of the final mode "Stack Deferred Indexed" (SFX) inflated time by an additional three hours. SFX addressing performs two separate memory lookups and performs three computations involving three registers and two memory values, generating a high number of possible paths through the segment.

Further augmenting this unit with the instruction execution unit would be uneconomical due to high verification time per run. Instead, verification of the instruction execution unit will solely rely on the post-conditions of operand decode unit, which describe the values held in registers `T5` and `T6`.

6.4 Instruction Execution

The instruction execution unit was verified as a separate component from the combine instruction-operand fetch/decode unit, due to the long run time of the previous millicode program. The instruction execution unit assumes as a pre-condition that the instruction and (if present) the operand have been loaded to the `IR` and `OS` register. If an operand is present, registers `T6` and `T5` contain the fully decoded operand value and address of the decoded operand value, respectively. These are the post-conditions of the instruction-operand fetch/decode unit. Using the composition rule from Hoare logic, the combined instruction-operand fetch/decode unit may be logically composed with instruction execution unit since they share post-/pre-conditions. Therefore, verifying the units separately is still a valid method of determining processor correctness. The instruction execution unit's post-conditions are drawn directly from the RTL of Pep/9. The bulk of the verification effort in the instruction execution unit involves checking if status bits are set correctly at the end of each instruction. This is in contrast to previous stages, where status bits are largely ignored and the majority of the work performed modifies register values. After executing a single instruction the program terminates.

Due to time constraints, millicode translations of instruction implementations were only created for 38 of 57 instructions. `Klee` verified that these selected instruction implementations were correct in 30 seconds by exploring unique 157 paths across 205,680 statements. Great difficulty was encountered with computing the correct values for status bits, as it required tracking CPU state over time, as well as performing bit-arithmetic. Issues translating bit arithmetic from the C++ implementation required constructing new logical expressions to describe values of status bits.

7 Validity of Results

Because all previous verifications of the Pep/9 architecture succeeded without errors, is it possible that the verification environment is flawed and always returns true? This section presents a new error in microcode that was discovered by the verification process. It explores injection of known and new faults into previously correct millicode segments. This section addresses the validity of exploring single instruction sequences.

7.1 Faults Detected

A new error was detected in existing microcode while verifying the `LDBr` instruction. The RTL specifies that the `LDBr` instruction must set the `N`(egative) bit to zero. The millicode implementation sets `N` correctly. When the `LDBr` millicode was compared to "equivalent" microcode, structural differences suggested that the microcode incorrectly modified `N`. The microcode was confirmed to be defective, and a patch was issued. The bug was not previously reported, making

```
KLEE: ERROR: badneg.c:118: ASSERTION FAIL: (cpu_get_pair(cpu, 2, 3) == 0) == cpu->PSNVCbits[Z]
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: badneg.c:106: ASSERTION FAIL: (WORD)(((WORD)(~cpu_get_pair(&starting_cpu, 0, 1)))
+1) == cpu_get_pair(cpu, 0, 1)
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 34972
KLEE: done: completed paths = 8
KLEE: done: generated tests = 7
```

Fig. 7: Post-condition errors (in red) indicating `Klee` verification failure.

it the first CPU fault definitively detected by software verification. Even though millicode lacks automated translation tools to microcode, comparative analysis between languages still detects latent errors.

7.2 Injecting Known Faults

In early versions of the microcoded CPU simulator for Pep/9 (named `Pep9Micro`), multiple microcode errors prevented the processor from implementing the instruction set correctly. Two such errors plagued the negate (`NEG`) family of instructions, which perform the two's complement operation on a register. These errors remained undiscovered for a fairly long time (around 6 months), since the relative rarity of the `NEG` instructions compounded with the rarity of the exceptional conditions. Showing that `Pep9Milli`'s verification can detect errors like these raises confidence in the verification process as a whole.

The two bugs affecting the `NEG` instructions occurred within the status bits. In both cases, one-byte operations were improperly chained to perform a word operation. In the first case, the carry bit from the low order byte was not preserved, rendering the high order byte computation incorrect in some cases. In the second case, the `Z`(ero) bit would be set if only the high order byte of the word was zero, when it should only be set if *both* bytes are zero. The millicode of the instruction execution stage was modified to contain these incorrect instruction implementations, but the post-conditions of the stage adhered to the RTL. `Klee` correctly generated errors, and verification failed (figure 7). In under an hour of development time, `Pep9Milli`'s verification was able to discover an error that existed in `Pep9Micro` for many months.

7.3 Injecting New Faults

In addition to known errors detected for `Pep9Micro`'s development, three new classes of errors were introduced. These errors were injected into the instruction fetch, operand fetch, and operand decode units. The program counter was decremented rather than incremented in the instruction fetch unit. This was achieved by swapping an "ADD" action for a "SUB" action. Branch targets were reversed in the operand fetch unit, causing execution to follow the wrong path. In the operand decode unit, a lookup table was modified so that instructions using "SFX" addressing would be decoded as using "I" addressing.

The first class of error, action substitutions, was injected into the operand fetch unit. After swapping an "ADD" action for a "SUB" action, `Klee` detected

that some paths through the instruction fetch unit set the program counter incorrectly. This class of error was common in design of the overall millicode program ("Am I supposed to use a shift or a rotate here?"), so `Pep9Milli`'s ability to catch this common class of errors is beneficial.

The second class of errors, branch target reversals, was injected into the instruction fetch unit. In the development of `Pep9Milli`, branch targets were accidentally reversed multiple times, causing an incorrect path to be taken through the program. The already-verified operand fetch unit was modified with incorrect branch targets. Unary instructions followed the non-unary path, and non-unary instructions followed the unary path. In both cases, `Klee` correctly found that this modification violated the post-conditions for either branch.

The third new error class, decoder entry errors, was injected into the operand decode unit. The operand decode unit uses a lookup table to select the correct addressing mode implementation for each instruction. All entries for "SFX" addressing were modified to decode as "I". `Klee` generated assertion errors in these handful of cases operands were decoded incorrectly.

Through injection of known and new faults the `Pep9Milli` verification framework has been shown to detect multiple classes of errors in a program. Additionally, any erroneous modification to any of the three stages of the instruction cycle (figure 3) would likely violate the post-conditions of the stage. Once translated to microcode, `Pep9Milli`'s verified millicode implementation decreases the likelihood of bugs in the Pep/9 ecosystem reaching end users.

7.4 Trusting Single Instructions Paths

All verification attempts described thus far only validate a single instruction at a time. Pep/9 has *residual control*, meaning state persists between instructions. Only validating single instructions at a time would seem to leave a large coverage gap in the correctness of `Pep9Milli`. Since verifying a single instruction can take up to 4 hours (§6.3), a more clever approach than applying iterated instruction executions is needed, as a naïve method may increase state space exponentially.

The solution to this problem stems from the design of the verification environment. `Klee` is allowed to pick all starting register, status flag, and memory values, within some constraints. For example, some registers are initialized to constants, and if the `prefetch` flag is set, then the prefetch register must contain the correct value. While verifying a single instruction, registers may take on any possible value so as to explore as many states as possible. In the concrete Pep/9 machine, registers are initialized to well-defined concrete values, such as 0. By allowing any register to have any value, it is as if the current instruction is at some point in an execution sequence that has evolved processor state to the current state from the initial state. Each instruction is entirely self-contained, excepting for the pre-fetch, which is specifically handled using a sequence of pre- and post-conditions. Evaluating a single instruction is akin to evaluating the next step of some indefinite instruction sequence. Thus, evaluating a single instruction is sufficient for validating Pep/9.

8 Discussion

Successfully verification of the instruction cycle indicates that 38 of 57 instructions in the Pep/9 instruction set are implemented correctly. The combined instruction fetch, operand fetch, and operand decode units were verified completely, while the instruction execution unit has partial results. Although the results are only partial, they provide a high degree of confidence that the new millicode correctly implements the Pep/9 instruction set.

At present, no tool exists to automatically translate millicode into a verifiable `C` program. `Pep9Milli`'s millicode was derived by hand from microcode, which is a process that may introduce errors. Results from verification and fault injection indicate that if there were errors in the millicode, they would have been caught. All millicode programs had to be translated by hand to `C`, and mistakes in the `C` program were hard to debug. These millicode programs were only translated to `C`, as it was deemed too tedious to translate them to microcode without the help of an automated milli-assembler. Nevertheless, these hand-translated programs were still found to meet their specifications. Using an automated milli-assembler would eliminate translation from microcode to millicode and from millicode to `C` as a source of introducing errors.

As stated earlier, `Pep9Milli`'s model assumes that memory access is side-effect free. Due to this limitation, a known defect in `Pep9Micro`'s microcode implementation was missed. For `store` instructions, additional (incorrect) memory reads are performed. This bug is well known, so the inability of `Pep9milli` to detect it highlights the deficiencies of the side-effect free memory model.

Initially, Pep/9's word size (which is two bytes) posed an issue while attempting to perform arithmetic. The ALU, modeled at the circuit level, operates on one-byte quantities, which are sourced from the register bank's 32 8-bit registers. Initial programs (§6.1) used these one-byte operations, so each millicode operation could be encoded as a single circuit operations. However, operations on words require chained one-byte arithmetic, and must therefore reference a series of one-byte registers to generate the requisite operand size. Pep/9 is a `little-endian` CPU, whereas the host machine running the verification is `big-endian`. When computing a 16-bit address using circuit operations, the circuit layer would reverse the byte order, causing the incorrect address to be generated. This was fixed by moving word-size operations from the circuit layer to the environment layer. These changes allowed successful verification of programs operating on word quantities. From a pedagogical perspective, these more abstract functions belong at the environment level, since the circuit level does not natively support 16-bit arithmetic.

Even relatively simple millicode programs experience a state space explosion. The looping implementation of the Fibonacci sequence (figure 6) took around a minute. Manual loop unrolling indicates that this program requires no more than 24 transitions to terminate, however the call tree indicates that 157 transitions were taken. The computation loop is visited repeatedly, probably due to arithmetic overflow on address computation, leading to multiple forks in execution.

Microcode loops forever—a CPU does not stop executing until explicitly shutdown—but only finite millicode instruction sequences are considered at present. Unbounded loops in millicode programs pose a difficulty to Pep9Milli. If a loop condition is not bounded (e.g. terminates after 13 iterations), verification may execute forever. The Pep/9 instruction set contains no instructions that loop inside the microcode or millicode. However, other instruction sets allow looping within an instruction. For example in x86 assembly language, an instruction can repeated an arbitrary number of times or until a condition is met. The REP MOVS instruction copies a string from one memory source to another memory destination. This complex behavior is encoded as a single assembly level instruction [42]. Pep9Micro's microcode (and thus millicode) is designed to be modified to implement an arbitrary instruction set, including instruction sets that include instructions such as REP MOVS. In its present form, Pep9Milli's verification model would not be able to verify these other instruction sets. Pep9Milli is limited to verifying instruction sets similar to Pep/9's, which is a subset of all instruction sets that Pep9Micro may implement. In general, Pep9Milli cannot verify the correctness of arbitrary, looping millicode. This issue may be alleviated by only exploring loops to some upper bound, but this comes at the cost of exhaustive verification.

Hardware control languages are significantly limited by their tight coupling to the hardware they describe. Much like microcode, a millicode program is not capable of driving a generic processor. For each processor to which this paper's process of abstraction and software verification is to be applied, a new "millicode" language and verification environment will have to be created. Ease of translation from RTL to millicode means that verification may be applied before the hardware is finalized. Through layered abstractions, software verification tools such as Klee are usable in a hardware domain.

9 Further Research

The immediate next step is to create a tool that is capable of assembling millicode into Pep/9 microcode as well as verifiable C code. The millicode language is already defined (see §5.2), so it is a matter of implementation. Generating optimal microcode will require compaction [1,13]; otherwise the microcode generated by the milli-assembler will be slower than microcode currently deployed in production. Additionally, assigning addresses to instructions at both the millicode and microcode level is an unresolved issue.

Following the creation of a milli-assembler, the next clear goal is to extend the millicode language to encode pre- and post-conditions. This would cause verification conditions to accompany the millicode, rather than being a manual addition to the generated C code. Encoding verification conditions in millicode would decrease typos and remove the manual translation step as a method for introducing errors. One major hurdle is devising a method to programmatically refer to CPU states across time, which is not modeled in the millicode language.

As discussed in §5.1, `Pep9Milli`'s address model assumes that memory access is side-effect free, and that memory writes are idempotent. These simplifying assumptions are incongruent with the reality of Pep/9's memory-mapped IO system, which requires side-effects to function. A form of read/write tracking needs to be implemented in the verification model. Next, the millicode language needs to be extended to specify which memory addresses should be accessed in a particular block. With these two components in place, `Pep9Milli` will be able to verify memory access in the presence of side effects.

10 Conclusion

Verification and design of microprocessors has been an active area of research in academia and industry for half a century. Successive advancements in microprocessor designs have created increasingly powerful microprocessors. Complex microarchitectures have evolved to squeeze every bit of performance out silicon. Despite advances, researchers and consumers still ask the same question "Does it work in all cases?". It turns out, the answer may be "no!" as discovered by Intel with the Pentium's FDIV bug [22]. However, verification techniques can validate interesting scenarios and improve overall microprocessor design and quality.

This paper introduces a process for using open-source software verification tools for CISC processor verification, which functions in situations that existing hardware verification tools are unequipped to handle. The approach consists of three steps:

1. Create a new, more abstract, off-chip hardware control language. This language compiles to an existing on-chip hardware control language, as well as a high-level language like `C`.
2. Create a verification environment in the high-level language, and leveraging existing software verification tools to verify the high-level language implementation correctly implements the RTL.
3. If verification succeeds, load the hardware control language program (i.e., microcode) on to the processor. This microcode verifiably implements the RTL of the instruction set.

The approach was demonstrated by verifying the implementation of the Pep/9 processor using the `Pep9Milli` project. `Pep9Milli` introduced a programming abstraction for Pep/9's hardware, *millicode*, that allows for simultaneous translation to microcode and `C`. Several millicode examples were verified to meet their specification using symbolic execution on the `C` translation. In total, 75% of the Pep/9 CPU is verified to be correct. For CISC processors lacking a hardware description, the methodology outlined in this paper allows for verification in circumstances that were previously difficult or impossible.

References

1. Ahmad, I., Dodhi, M.K., Saleh, K.A.: An evolutionary technique for local microcode compaction. Microprocessors and Microsystems **19**(8), 467 –

474 (1995). https://doi.org/https://doi.org/10.1016/0141-9331(96)82011-4, http://www.sciencedirect.com/science/article/pii/0141933196820114
2. Arons, T., Elster, E., Ozer, S., Shalev, J., Singerman, E.: Efficient symbolic simulation of low level software. In: 2008 Design, Automation and Test in Europe. pp. 825–830 (Mar 2008). https://doi.org/10.1109/DATE.2008.4484776
3. Arons, T., Elster, E., Fix, L., Mador-Haim, S., Mishaeli, M., Shalev, J., Singerman, E., Tiemeyer, A., Vardi, M.Y., Zuck, L.D.: Formal verification of backward compatibility of microcode. In: Etessami, K., Rajamani, S.K. (eds.) Computer Aided Verification. pp. 185–198. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
4. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3) (May 2018). https://doi.org/10.1145/3182657, https://doi-org.proxy.library.georgetown.edu/10.1145/3182657
5. Bauer, S.M.: Bell labs microcode for the ibm 360/67. In: Proceedings of the 8th Annual Workshop on Microprogramming. pp. 40–44. MICRO 8, ACM, New York, NY, USA (1975). https://doi.org/10.1145/800148.804859, http://doi.acm.org/10.1145/800148.804859
6. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. pp. 209–224. OSDI'08, USENIX Association, Berkeley, CA, USA (2008), http://dl.acm.org/citation.cfm?id=1855741.1855756
7. Cadar, C., Sen, K.: Symbolic execution for software testing: Three decades later. Commun. ACM **56**(2), 82–90 (Feb 2013). https://doi.org/10.1145/2408776.2408795, http://doi.acm.org/10.1145/2408776.2408795
8. Cai, L., Gajski, D.: Transaction level modeling: An overview. In: Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. pp. 19–24. CODES+ISSS '03, Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/944645.944651, https://doi-org.proxy.library.georgetown.edu/10.1145/944645.944651
9. Carter, W.C., Joyner, W.H., Brand, D.: Symbolic simulation for correct machine design. In: 16th Design Automation Conference. pp. 280–286 (Jun 1979). https://doi.org/10.1109/DAC.1979.1600119
10. Chlipala, A.: A verified compiler for an impure functional language. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 93–106. POPL '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1706299.1706312, https://doi.org/10.1145/1706299.1706312
11. Daming D. Chen, G.J.A.: Security analysis of x86 processor microcode (2014), https://www.dcddcc.com/docs/2014_paper_microcode.pdf
12. Dave, M.A.: Compiler verification: A bibliography. SIGSOFT Softw. Eng. Notes **28**(6), 2 (Nov 2003). https://doi.org/10.1145/966221.966235, https://doi.org/10.1145/966221.966235
13. Davidson, Landskov, Shriver, Mallett: Some experiments in local microcode compaction for horizontal machines. IEEE Transactions on Computers **C-30**(7), 460–477 (Jul 1981). https://doi.org/10.1109/TC.1981.1675826
14. Davis, J., Slobodova, A., Swords, S.: Microcode verification – another piece of the microprocessor verification puzzle. In: Klein, G., Gamboa, R. (eds.) Interactive Theorem Proving. pp. 1–16. Springer International Publishing, Cham (2014), https://www.kookamara.com/jared/2014-itp-ucode.pdf

15. Flynn, M.J., Rosin, R.F.: Microprogramming: An introduction and a viewpoint. IEEE Transactions on Computers **C-20**(7), 727–731 (Jul 1971). https://doi.org/10.1109/T-C.1971.223341
16. Franzén, A., Cimatti, A., Nadel, A., Sebastiani, R., Shalev, J.: Applying smt in symbolic execution of microcode. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design. pp. 121–128. FMCAD '10, FMCAD Inc, Austin, TX (2010), http://dl.acm.org/citation.cfm?id=1998496.1998520
17. Gerst, H.: Verification of the vlsi-/370 microprocessor. In: Proceedings. VLSI and Computer Peripherals. COMPEURO 89. pp. 5/128–5/133 (may 1989). https://doi.org/10.1109/CMPEUR.1989.93498
18. Goel, S., Slobodova, A., Sumners, R., Swords, S.: Verifying x86 instruction implementations (2019)
19. Goldman, S.P., Mohr, L.M., Smith, D.R.: Using microcode in the functional verification of an i/o chip. IBM Journal of Research and Development **49**(4), 581–588 (Jul 2005), https://search.proquest.com/docview/220687269?accountid=11091, copyright International Business Machines Corporation Jul-Sep 2005
20. Greve, D.A.: Symbolic simulation of the jem1 microprocessor. In: Gopalakrishnan, G., Windley, P. (eds.) Formal Methods in Computer-Aided Design. pp. 321–333. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
21. Gunter, T.G., Tredennick, H.L.: Two-level control store for microprogrammed data processor (Apr 13 1982), uS Patent 4,325,121
22. Harrison, J.: Formal verification at intel. In: 18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings. pp. 45–54 (June 2003). https://doi.org/10.1109/LICS.2003.1210044
23. Hennessy, J.L., Patterson, D.A.: Computer Architecture, Sixth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edn. (2017)
24. Hentschel, M., Bubel, R., Hähnle, R.: The symbolic execution debugger (sed): a platform for interactive symbolic execution, debugging, verification and more. International Journal on Software Tools for Technology Transfer **21**(5), 485–513 (Oct 2019). https://doi.org/10.1007/s10009-018-0490-9, https://doi.org/10.1007/s10009-018-0490-9
25. Hsu, S.K., Agarwal, A., Mathew, S.K., Krishnamurthy, R.K., Hansson, M., Alvandpour, A.: A 9ghz 320x80bit low leakage microcode read only memory in 65nm cmos. In: 2006 Proceedings of the 32nd European Solid-State Circuits Conference. pp. 299–302 (Sep 2006). https://doi.org/10.1109/ESSCIR.2006.307590
26. Hunt Jr, W., Kaufmann, M., Moore, J., Slobodova, Z.: Industrial hardware and software verification with acl2. Philos Trans A Math Phys Eng Sci. **375** (October 2017). https://doi.org/10.1098/rsta.2015.0399
27. Ivanov, L.: Formal verification of a microprocessor control. In: Proceedings of the 44th IEEE 2001 Midwest Symposium on Circuits and Systems. MWSCAS 2001 (Cat. No.01CH37257). vol. 2, pp. 646–650 vol.2 (Aug 2001). https://doi.org/10.1109/MWSCAS.2001.986272
28. Jackson, D.: Evolution of processor microcode. IEEE Transactions on Evolutionary Computation **9**(1), 44–54 (Feb 2005). https://doi.org/10.1109/TEVC.2004.837922
29. KiranKumar, V.M.A., Gupta, A., Ghughal, R.: Symbolic trajectory evaluation: The primary validation vehicle for next generation intel® processor graphics fpu. In: 2012 Formal Methods in Computer-Aided Design (FMCAD). pp. 149–156 (Oct 2012)

30. Koerner, S., Kuenzel, M., McCain, E.C.: Ibm eserver z900 system microcode verification by simulation: The virtual power-on process. IBM Journal of Research and Development **46**(4), 587–595 (Jul 2002), https://search.proquest.com/docview/220658828?accountid=11091, name - IBM Corp; Copyright - Copyright International Business Machines Corporation Jul/Sep 2002; Last updated - 2017-10-31; CODEN - IBMJAE
31. Koppe, P., Kollenda, B., Fyrbiak, M., Kison, C., Gawlik, R., Paar, C., Holz, T.: Reverse engineering x86 processor microcode. In: 26th {USENIX} Security Symposium ({USENIX} Security 17). pp. 1163–1180 (2017)
32. McRaven, M., Warford, J.S.: A microcode implementation of the pep/9 computer (September 2019), unpublished Manuscript
33. McRaven, M., Warford, S.: Pep9micro: Designing a microcoded cpu (2018)
34. Miller, S.P., Srivas, M.: Formal verification of the aamp5 microprocessor: a case study in the industrial use of formal methods. In: Proceedings of 1995 IEEE Workshop on Industrial-Strength Formal Specification Techniques. pp. 2–16 (April 1995). https://doi.org/10.1109/WIFT.1995.515475
35. Mueller, R.A., Duda, M.R.: Formal methods of microcode verification and synthesis. IEEE Software **3**(4), 38–48 (July 1986). https://doi.org/10.1109/MS.1986.233753
36. Oulsnam, G.: Unravelling Unstructured Programs. The Computer Journal **25**(3), 379–387 (08 1982). https://doi.org/10.1093/comjnl/25.3.379, https://doi.org/10.1093/comjnl/25.3.379
37. Pasricha, S., Dutt, N.: On-chip communication architectures: system on chip interconnect. Morgan Kaufmann (2010)
38. Pasricha, S., Dutt, N., Ben-Romdhane, M.: Fast exploration of bus-based communication architectures at the ccatb abstraction. ACM Trans. Embed. Comput. Syst. **7**(2) (Jan 2008). https://doi.org/10.1145/1331331.1331346, https://doi.org/10.1145/1331331.1331346
39. Redfield, S.R.: A study in microprogrammed processors: A medium sized microprogrammed processor. IEEE Trans. Comput. **20**(7), 743–750 (Jul 1971). https://doi.org/10.1109/T-C.1971.223343, http://dx.doi.org/10.1109/T-C.1971.223343
40. Snyder, W.: Verilator: the fast free verilog simulator. URL: http://www. veripool. org (2019)
41. Stecklein, J.M., Dabney, J., Dick, B., Haskins, B., Lovell, R., Moroney, G.: Error cost escalation through the project life cycle (2004)
42. Turley, J.L.: Advanced 80386 Programming Techniques. McGraw-Hill, Inc., New York, NY, USA (1988)
43. Ulrich, J.W.: The derivation of microcode by symbolic execution. SIGMICRO Newsl. **11**(3-4), 38–42 (Nov 1980), http://dl.acm.org/citation.cfm?id=1014190.802709
44. Val, C.G.d.: Conflict-driven symbolic execution: How to learn to get better. Ph.D. thesis, University of British Columbia (2014)
45. Wagner, A., Dasgupta, S.: Axiomatic proof rules for a machine-specific microprogramming language. SIGMICRO Newsl. **14**(4), 151–158 (Dec 1983). https://doi.org/10.1145/1096419.1096442, http://doi.acm.org/10.1145/1096419.1096442
46. Warford, J.S.: Computer Systems. Jones and Bartlett Publishers, Inc., USA, 5th edn. (2016)
47. Warford, S.: Documentation for exam handouts (2016)

48. Wilding, M.M., Greve, D.A., Richards, R.J., Hardin, D.S.: Formal verification of partition management for the aamp7g microprocessor. In: Design and Verification of Microprocessor Systems for High-Assurance Applications, pp. 175–191. Springer (2010)
49. Wolf, C., Glaser, J., Kepler, J.: Yosys-a free verilog synthesis suite. In: Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip) (2013)

A Overview of Pep/9

This appendix provides an overview of the Pep/9 virtual machine and describes the applications that implement it[7]. Pep/9 is a pedagogical virtual machine described in *Computer Systems*, fifth edition [46]. It is a 16-bit CISC processor, which features expanding opcodes that are either unary (one byte) or non-unary (three bytes), and is designed as an instructional tool to teach computer systems and organization concepts to undergraduates [46].

There are four applications in the Pep/9 suite of software.

- `Pep9`[8]—An assembly language and object code programming environment.
- `Pep9CPU`—A graphical simulator for the microcoded data section of the Pep/9 CPU.
- `Pep9Micro`—A unified microcode-assembly language programming environment, allowing users to debug multiple levels of abstraction simultaneously.
- `Pep9Term`—A command line utility designed to automate grading of programs from aforementioned applications.

Figure 8 visualizes the levels of abstraction spanned by these applications.

A.1 `Pep/9`: Assembly Language and ISA

At the ISA level, the CPU has five registers: the accumulator (`A`), the index register (`X`), the program counter (`PC`), the stack pointer (`SP`), and the instruction register (`IR`). The accumulator stores computation results, the index register facilitates array processing, the program counter contains the address of the next instruction to be executed, the stack pointer points to the top of the runtime stack, and the instruction register stores the instruction fetched during the *fetch* part of the von Neumann cycle. It has four status bits to indicate whether the result of an operation is: negative (`N`), zero (`Z`), signed overflow (`V`), or unsigned overflow (`C`).

Pep/9 has a 16-bit memory space that is byte-addressable. The entire memory space is accessible without segmentation or paging. It uses memory-mapped

[7] Programs and source code available at http://computersystemsbook.com/5th-edition/pep9/

[8] As a matter of notation, Pep/9 (regular font with a "/") describes the virtual machine, while `Pep9` (monospaced font with no "/") describes the application.

[9] Figure adapted from [46], used with permission.

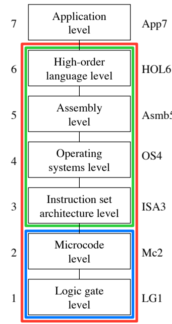


Fig. 8: Levels of abstraction spanned by the Pep/9 application suite[9]. Green corresponds to `Pep9`. Blue corresponds to `Pep9CPU`. Red corresponds to `Pep9Micro` and `Pep9Term`.

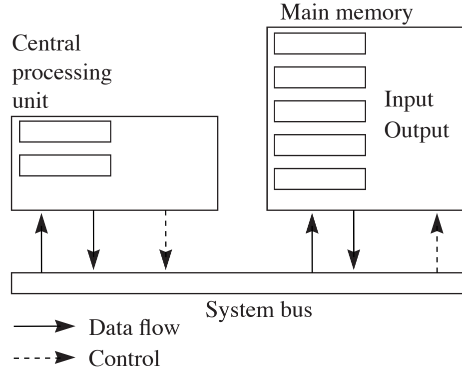


Fig. 9: Pep9 ISA System Model[10].

input and output to process ASCII character streams. Figure 9 shows the system model at the ISA level. Details like the number of temporary registers and processor circuitry are hidden from the user at the ISA level.

Figure 10 lists the instructions in the Pep/9 instruction set. Instructions specifiers are one byte and operands are two bytes, making unary instructions one byte wide and non-unary instructions three bytes wide. It has eight addressing modes that are used by non-unary instructions. These addressing modes aid translation from `C` to assembly, performing array addressing, pointer lookups, and addressing for globally-, heap-, and stack-allocated structs. Figure 11 details all addressing modes.

The `Pep9` application allows users to write and debug assembly language programs using a built in integrated development environment (IDE). It features

[10] Figure adapted from [46], used with permission.
[11] Figure adapted from [46], used with permission.

Fig. 10: Pep/9 instruction set[11]. Underlined instructions are unary, all others are non-unary.

Mnemonic	Instruction
STOP	Halt processor
RET	Return from CALL
RETTR	Return from trap
MOVSPA	Move SP to A
MOVFLGA	Move NZVC flags to A$\langle 12..15 \rangle$
MOVAFLG	Move A$\langle 12..15 \rangle$ to NZVC flags
NOTr	Bitwise invert r
NEGr	Negate r
ASLr	Arithmetic shift left r
ASRr	Arithmetic shift right r
ROLr	Rotate left r
RORr	Rotate right r
BR	Branch unconditionally
BRLE	Branch if $\leq$
BRLT	Branch if $<$
BREQ	Branch if $=$
BRNE	Branch if $\neq$
BRGE	Branch if $\geq$
BRGT	Branch if $>$
BRV	Branch if V
BRC	Branch if C
CALL	Call subroutine

Mnemonic	Instruction
NOP0	Unary no operation
NOP1	Unary no operation trap
NOP	Nonunary no operation trap
DECI	Decimal input trap
DECO	Decimal output trap
HEXO	Hexadecimal output trap
STRO	String output trap
ADDSP	Add to stack pointer (SP)
SUBSP	Subtract from stack pointer (SP)
ADDr	Add to r
SUBr	Subtract from r
ANDr	Bitwise AND to r
ORr	Bitwise OR to r
CPWr	Compare word to r
CPBr	Compare byte to r$\langle 8..15 \rangle$
LDWr	Load word r from memory
LDBr	Load byte r$\langle 8..15 \rangle$ from memory
STWr	Store word r to memory
STBr	Store byte r$\langle 8..15 \rangle$ to memory

a student-friendly machine language object code in a hexadecimal format, which gives students the ability to code directly in machine language, bypassing the assembler. There is an integrated debugger that allows for breakpoints, single- and multi-step execution, CPU tracing, and memory tracing. It also has the ability to recover from endless loops.

A.2 `Pep9CPU`: Microcode and Logic Gates

At the microcode level of abstraction, the CPU has two parts: the data section and the control section. The data section contains an 8-bit arithmetic logic unit (ALU), 8-bit data path, a 16-bit address bus, and a register bank containing 32 8-bit registers [47]. Two variations of the CPU exist, one with an 8-bit data bus and one with a 16-bit data bus. Figure 12 describes circuitry of the 16-bit data bus variant of the data section. Both variations are simulated within `Pep9CPU`, allowing users to understand the tradeoffs between different bus sizes.

The original Pep/9 specification lacks a CPU control section. Without a control section, which is responsible for branching between microcode instructions,

Fig. 11: Addressing modes for Pep. OprndSpec is the instruction operand. X is the index register. SP is the stack pointer register. Mem[...] indicates memory access.

Abbreviation	Name	Decoding
I	*I*mmediate	OprndSpec
D	*D*irect	Mem[OprndSpec]
N	i*N*direct	Mem[Mem[OprndSpec]]
S	*S*tack relative	Mem[SP + OprndSpec]
SF	*S*tack de*F*erred	Mem[Mem[SP + OprndSpec]]
X	inde*X*ed	Mem[OprndSpec + X]
SX	*S*tack inde*X*ed	Mem[SP + OprndSpec + X]
SFX	*S*tack de*F*erred inde*X*ed	Mem[Mem[SP + OprndSpec] + X]

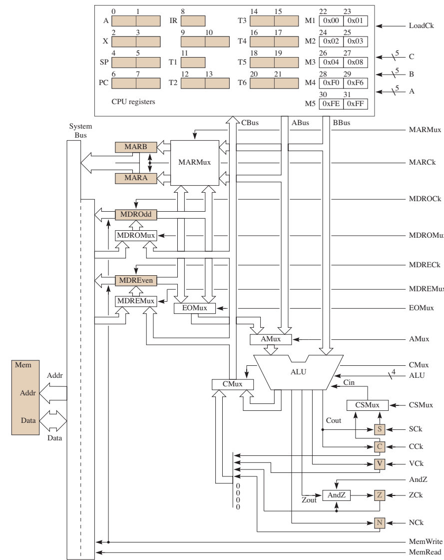


Fig. 12: Pep/9 CPU data section[12].

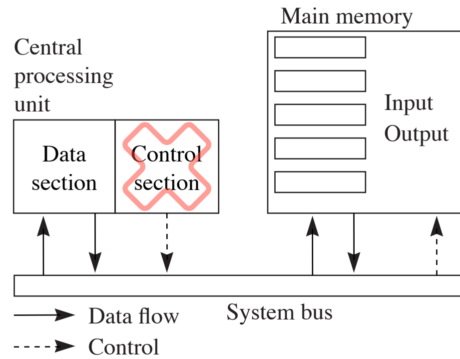


Fig. 13: Machine model at the microcode level[13]. The control section is crossed out, because `Pep9CPU` does not contain a control section.

no loops or conditional branches can be created in microcode. This limitation does not allow the simulation of the complete instruction cycle.

Of the register bank's 32 8-bit registers, 11 registers are used to implement the ISA-visible registers. A further 10 are hardwired to useful constant values. The remaining 11 registers are scratch registers for use in microprograms. The CPU contains an additional shadow carry `S` bit. It allows users to perform computations that do not modify the ISA-visible state, such as carry-out while incrementing the program counter. Figure 13 shows the system model of `Pep9CPU`.

The `Pep9CPU` application allows users to write Pep/9 microcode fragments and interact with a simulated Pep/9 CPU. Using `Pep9CPU`, users can visualize the execution of limited microcode fragments, and watch data flow through the processor. Its IDE allows users to find logical errors in microcode fragments using unit tests. By writing microcode programs, students learn how to control the data flow through in the CPU data section. Lack of a control section limits users to writing short, non-branching code fragments. `Pep9CPU` and `Pep9` simulate the system at two disjoint levels of abstraction.

A.3 `Pep9Micro`: Merging layers of abstraction

A later work implemented the control section of the CPU [33], completing Pep/9 at the microcode level. The control section uses the 16-bit data bus variant of the data section. Pep/9's microcode language was augmented with syntax for conditional and unconditional branching. Figure 14 presents the circuitry of the control section. A manuscript describing the design and use of the extended processor is currently being written [32].

[13] Figure adapted from [32], used with permission.

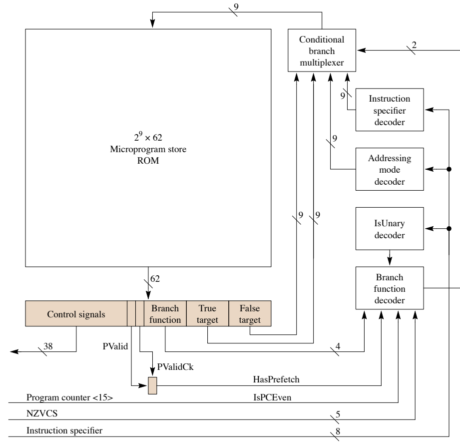


Fig. 14: Pep/9 CPU control section[14].

Each microcode instruction explicitly specifies its successor instructions. In the case of conditional branches, both the `true` and `false` targets are embedded in the instruction format. Programmers may choose from one of 16 *branch functions* for each micro-instruction.

A microcode program implementing the ISA instruction cycle was implemented using the RTL specification of Pep/9. This 344 line microcode program fetches instructions and decodes their operands. Then, it executes the instruction, before repeating the instruction cycle. This process continues until an exception is raised or `STOP` mnemonic is encountered. With a complete control section and microprogram implementation, there is no longer a divide between microcode and the instruction set architecture.

The `Pep9Micro` application provides a combined microcode-assembly environment, allowing users to write and debug programs at multiple levels of abstraction simultaneously. It is a synthesis of existing `Pep9` and `Pep9CPU` applications. It uses the same assembly language IDE, breakpoint, and debugging facilities in `Pep9`. `Pep9Micro` visualizes data flow through the data section of the CPU.

`Pep9Micro` augments the existing microcode IDE to allow for microcode breakpoints and debugging. While debugging, it is possible to switch between the ISA level and microcode level on-the-fly. Users are also allowed to modify or replace the microcode implementing the instruction set, allowing them to experiment with new instructions. These cumulative modifications allow users to see the relationship between all levels of abstraction in a processor at the same time.

[14] Figure adapted from [32], used with permission.