UNIVERSITY OF CAMBRIDGE

Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification

Yash Shah

November 2024

Background

Simulation based verification

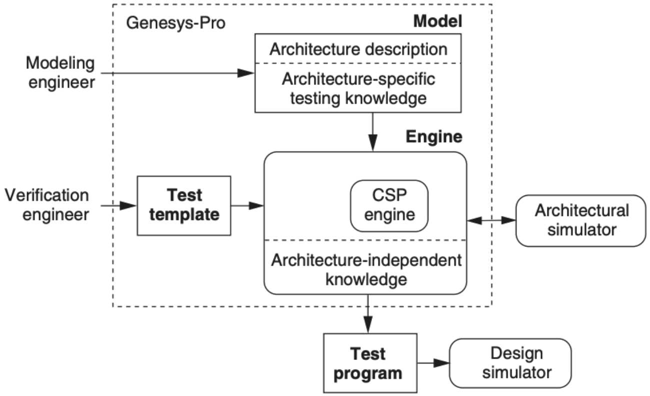
1. Generator
    - Complete randomness will lead to poor coverage of DUT (Design Under Test)
    - Totally directed tests require laborious, specialist manual effort
    - Requires adding constraints, forming a CSP (Constraint Satisfaction Problem)—this is constrained-random testing
    - Test programs (output assembly at architectural-level) are solutions of the CSP

Simulation based verification

2. Simulator
    - Runs the generated test program
3. Monitor
    - Check for disparities between output of the simulator against expected behaviour/output
    - Track coverage of the design, until a satisfactory level reached

Overview of Genesys-Pro

Architecture Overview [1]



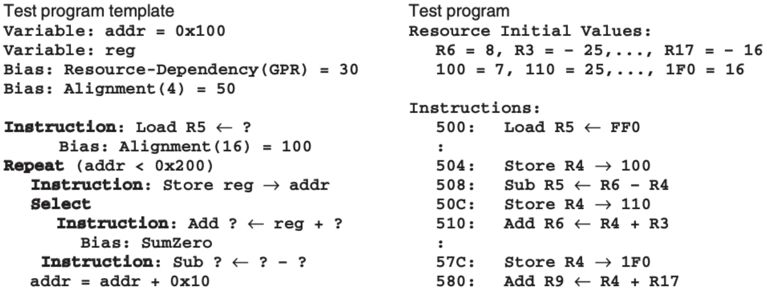
Test template language

Composed of four types of statements:

1. Basic instruction
2. Sequencing-control
3. Standard programming constructs
4. Constraint

Can view as a powerful meta-programming language

Test program template example [1]



Test program template

```
Variable: addr = 0x100
Variable: reg
Bias: Resource-Dependency(GPR) = 30
Bias: Alignment(4) = 50

Instruction: Load R5 ← ?
     Bias: Alignment(16) = 100
Repeat (addr < 0x200)
  Instruction: Store reg → addr
  Select
    Instruction: Add ? ← reg + ?
      Bias: SumZero
   Instruction: Sub ? ← ? - ?
  addr = addr + 0x10
```

Test program

```
Resource Initial Values:
  R6 = 8, R3 = - 25,..., R17 = - 16
  100 = 7, 110 = 25,..., 1F0 = 16

Instructions:
  500:  Load R5 ← FF0
  :
  504:  Store R4 → 100
  508:  Sub R5 ← R6 - R4
  50C:  Store R4 → 110
  510:  Add R6 ← R4 + R3
  :
  57C:  Store R4 → 1F0
  580:  Add R9 ← R4 + R17
```

Basic instruction

Can explicitly specify instructions in the generated test

Ability to control properties of instruction e.g:

- Registers
- Immediate
- Importantly allows random selection with **?**

Sequencing-control

- Sequence
- Select
- Repeat
- Permute
- Concurrent

Standard programming constructs

Variables can provide partial specification

If value not concretely specified, but specified in multiple places, the variable can take a random value, with additional constraint that the value must be the same in all of these places

Pre- and post-assertions to determine whether a statement should be part of the test

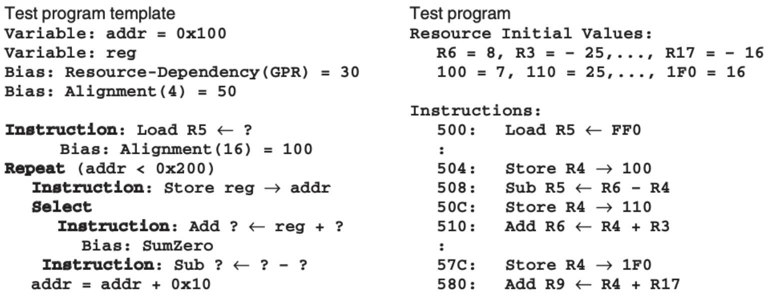
Post-assertions especially powerful when cannot pre-assess a statement's condition as instructions may have complex semantics

Constraint statements

Bias generator's randomness towards interesting areas e.g.

- Aligment (cache-line size)
- Cache (generate memory access patterns to cause hits, misses, line replacements)
- Translation (trigger various address translation mechanisms)
- Resource dependency (source-target dependencies)

Understanding the test program template [1]



Test program template

```
Variable: addr = 0x100
Variable: reg
Bias: Resource-Dependency(GPR) = 30
Bias: Alignment(4) = 50

Instruction: Load R5 ← ?
     Bias: Alignment(16) = 100
Repeat (addr < 0x200)
   Instruction: Store reg → addr
   Select
      Instruction: Add ? ← reg + ?
        Bias: SumZero
    Instruction: Sub ? ← ? - ?
   addr = addr + 0x10
```

Test program

```
Resource Initial Values:
   R6 = 8, R3 = - 25,..., R17 = - 16
   100 = 7, 110 = 25,..., 1F0 = 16

Instructions:
   500:   Load R5 ← FF0
   :
   504:   Store R4 → 100
   508:   Sub R5 ← R6 - R4
   50C:   Store R4 → 110
   510:   Add R6 ← R4 + R3
   :
   57C:   Store R4 → 1F0
   580:   Add R9 ← R4 + R17
```

Modeling framework

Increased modularity over predecessor

Architecture-specific details $\rightarrow$ model

Architecture-generic details $\rightarrow$ engine

Modeling framework provides:

High-level building blocks

Constraint-based representation of modeled components

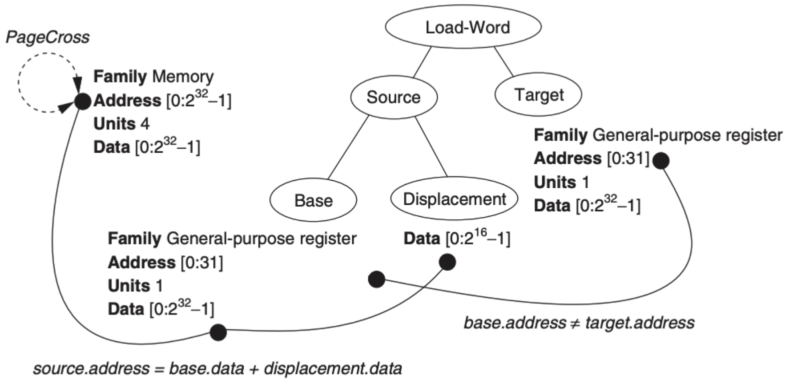
Model

Declarative description of the processor:

- Instructions
- Design resources (registers, caches)
- High-level mechanisms (address translation)

Design-specific testing knowledge to increase coverage

Model of a Load-Word instruction [1]



Test generation engine

Unfortunately intractable to solve entire problem (model + test template) as a single CSP

Hybrid stream and instruction generation approach

Stream generation

Primarily driven by sequencing-control statements

Full CSP only for instruction “leaves”

Only previously generated instructions considered when generating the next statement

Special instructions inserted to detect errors before they manifest

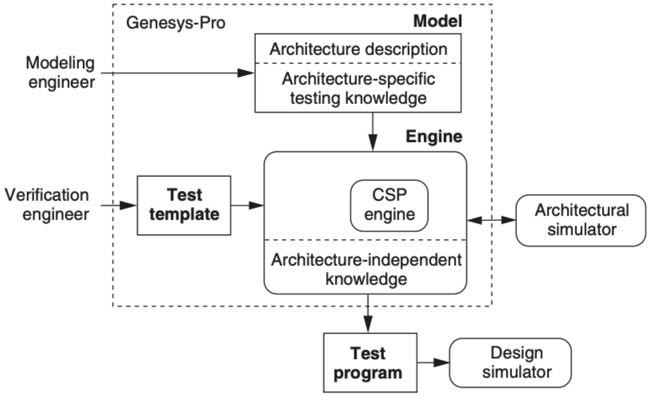
Prolog-like back-tracking

Instruction generation

3-step process for instruction generation:

1. Formulate CSP
   - Variable for each instruction attribute (e.g. `base.address`)
   - Architectural + testing-knowledge constraints
   - Test template constraints super-imposed
2. Find solution (MAC-based algorithm) and produce instruction instance
3. Invoke architectural reference model for simulating the generated instruction

Architecture Overview [1]



Critique and future development

Strengths

- Modular, architecture-independent design
- Expressive test template language
- Hybrid stream and CSP-based approach allows for long instruction sequences
- MAC-based CSP-solver allows satisfaction of simultaneous constraints by postponing early heuristic decisions
- Support for concurrency allows for testing of multi-core designs

Weaknesses

- Manual constraint modeling
- Limited micro-architectural support
- Coverage feedback not accounted for during subsequent program generation
- CSP-solving is NP-hard [2]

Manual constraint modeling

Issues

Time-consuming and specialist-dependent

Error-prone, especially when C++ used for more complex constraints

Makes adapting to new architectures challenging

Manual constraint modeling

Possible mitigation

Architecture Description Languages (ADLs) already in use for writing unambiguous, executable specifications e.g. ARM ASL [3]

ADLs can also be used to automatically derive modeling constraints as shown by MicroTESK [4]

Only need to define translators to process ADLs into constraints

Limited micro-architectural support

Issues

Mechanisms including multiple execution units, out-of-order execution, pipelining, and caching are each complex themselves and have even more perplexing interactions.

This motivates micro-architectural events in the verification plan, but Genesys-Pro only supports architectural-level simulation

Formal methods like FSMs of processor operation lead to state-space explosion

Critical for verifying performance optimizations like register forwarding, which require precising timing in tests

Limited micro-architectural support

Possible mitigation

Piparazzi [5] offers declarative parameterized building blocks

- Parameters for a cache include size, associativity, and replacement policy
- Building blocks contain instruction-flow and timing variables whose values, selected during generation, determine behavior of object e.g. variables for opcode, issue time, and pipeline for an instruction building block

Limited micro-architectural support

Possible mitigation

Verification engineer's view still relatively uncomplicated

- Request two instructions are executed in the same pipeline e.g. `I[1].PIPELINE = I[3].PIPELINE`

Unfortunately, adds significant generation time limiting test length

Additional work required for tighter integration—currently just another stage after architectural-level to increase coverage

Additional modeling complexity—perhaps scope to mitigate through assisted constraint extraction from Hardware Description Language (HDL)

Coverage feedback not accounted for during subsequent program generation

Issues

Coverage information is not being used in Genesys-Pro to guide future generated programs

Possible mitigation

“Coverage-directed test generation through automatic constraint extraction” [6]

We are interested in the signals produced within the simulated microprocessor for coverage, but have to come up with the inputs to produce those desired signals

Automatic Test Pattern Generation (ATPG) intractable for complex designs

Coverage feedback not accounted for during subsequent program generation

Possible mitigation

Instead of statistical approach is taken by analyzing the simulation output (history trace of signals + inputs)

Look at patterns in input driving change in desired signal to inform our decision of the ideal input set to improve coverage

Coverage feedback not accounted for during subsequent program generation [6]

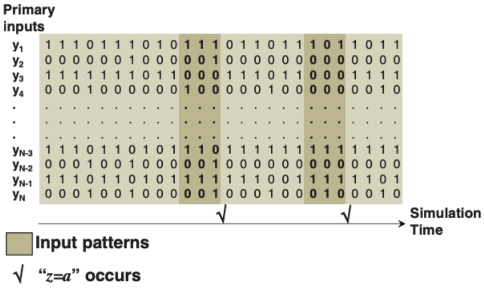

Primary inputs
y1 1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1
y2 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
y3 1 1 1 1 1 1 1 0 1 1 0 0 0 1 1 1 0 1 1 0 0 0 1 1 1 1
y4 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0
yN-3 1 1 1 0 1 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1
yN-2 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
yN-1 1 1 1 0 1 1 0 1 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 1
yN 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0
√ √
Simulation Time
Input patterns
√ "z=a" occurs

Fig. 3. **Target dataset to extract patterns** $P_1,\ldots,P_k$

Coverage feedback not accounted for during subsequent program generation [6]

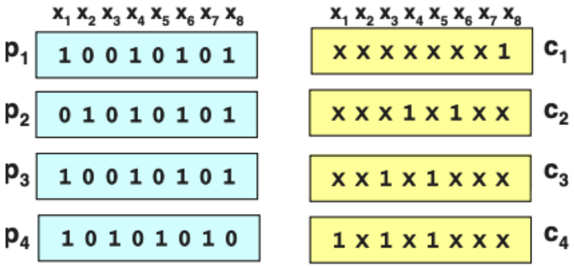


Fig. 4. **Extracting constraints from patterns**

CSP-solving is NP-hard

Issues

Major barrier to increasing modeling complexity like with micro-architectural models

Possible mitigation

“Automated Coverage Directed Test Generation Using a Cell-Based Genetic Algorithm” [7]

Split the input domains into sub-ranges, each having weight representing probability of generating values from that range

Each range is a cell

Genetic algorithm optimizes widths, heights, and distributions of these cells, using coverage metric(s) as a fitness function

References i

[1] A. Adir et al. "Genesys-Pro: innovations in test program generation for functional processor verification". In: *IEEE Design & Test of Computers* 21.2 (2004), pp. 84–93. DOI: 10.1109/MDT.2004.1277900.

[2] Azza Gaysin. *Proof complexity of CSP*. 2023. arXiv: 2201.00913 [math.LO]. URL: https://arxiv.org/abs/2201.00913.

[3] Alastair Reid. "Trustworthy specifications of ARM® v8-A and v8-M system level architecture". In: *2016 Formal Methods in Computer-Aided Design (FMCAD)*. 2016, pp. 161–168. DOI: 10.1109/FMCAD.2016.7886675.

References ii

[4] Alexander Kamkin and Andrei Tatarnikov. **“MicroTESK: A Tool for Constrained Random Test Program Generation for Microprocessors”.** In: Jan. 2018, pp. 387–393. ISBN: 978-3-319-74312-7. DOI: `10.1007/978-3-319-74313-4_28`.

[5] A. Adir et al. **“Piparazzi: a test program generator for micro-architecture flow verification”.** In: *Eighth IEEE International High-Level Design Validation and Test Workshop*. 2003, pp. 23–28. DOI: `10.1109/HLDVT.2003.1252470`.

References iii

[6] Onur Guzey and Li-C. Wang. "Coverage-directed test generation through automatic constraint extraction". In: *2007 IEEE International High Level Design Validation and Test Workshop*. 2007, pp. 151–158. DOI: 10.1109/HLDVT.2007.4392805.

[7] Amer Samarah et al. "Automated Coverage Directed Test Generation Using a Cell-Based Genetic Algorithm". In: *2006 IEEE International High Level Design Validation and Test Workshop*. 2006, pp. 19–26. DOI: 10.1109/HLDVT.2006.319996.