

COMS30026 Design Verification

# Stimuli Generation

(Part II)

Kerstin Eder

(Acknowledgement: Avi Ziv from the IBM Research Labs in Haifa has kindly permitted the re-use of some of his slides.)

# Outline

## **Motivation: Advanced Stimuli Generation**

- Running example: PowerPC processor (repeated from Part I)

## **Part I: Issues in stimuli generation**

- How many generators?
- Level of abstraction
- Online vs. offline generation
- Dynamic vs. static generation
- Test length

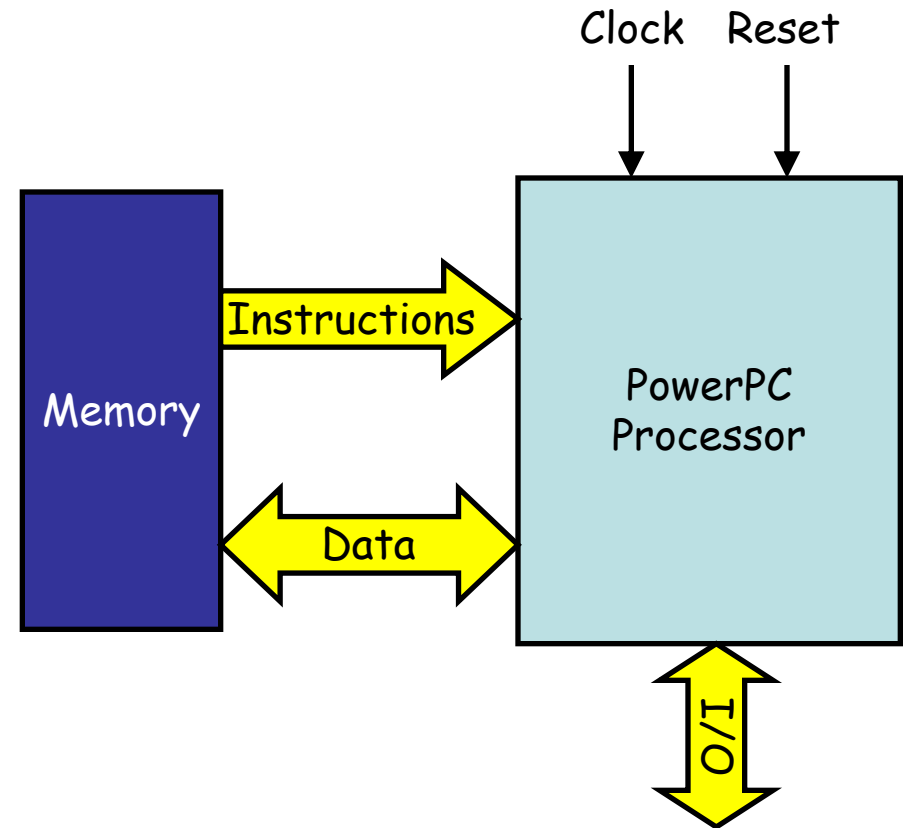
## **Part II: Test Automation**

- Randomness
- Constrained pseudo-random stimulus generation



# Running Example – PowerPC Processor

- Black box view
  - Interface to memory (via caches)
    - For instruction fetching
    - For data fetching and storing
  - Interface to I/O devices
    - For data fetching and storing
    - Interrupts
  - Miscellaneous interface
    - Clocks
    - Reset
    - ...



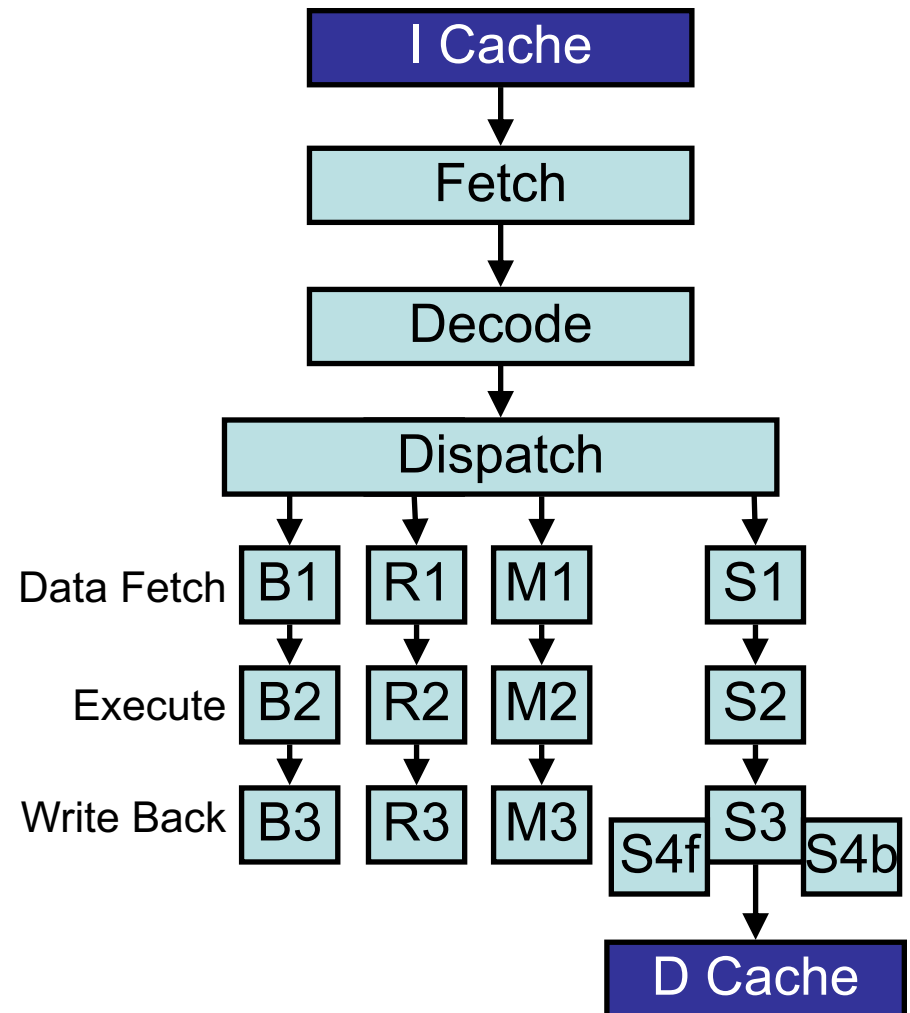
# Architectural View

- RISC (Reduced Instruction Set Computer) processor
  - “Small” number of instructions (~400)
  - One simple operation per instruction
  - Fixed length instructions (32 bits = 1 word)
  - Specific load and store instructions to access memory
    - All other instructions use registers for operands
- Large register files
  - 32 general purpose registers (GPR)
  - 32 floating-point registers (FPR)
    - Used only for floating-point operations
  - Several special purpose registers
    - Condition register, link register, status register, etc.
- Complex memory model
  - Multiple level address translation
  - Coherency rules
  - (not in the scope of the lecture)



# Microarchitectural View

- Multi-threaded
- In-order execution
- Four instructions wide
  - Fetch
  - Decode
  - Dispatch
- Four execution units
  - B: Branch
  - R: Simple Arithmetic
  - M: Complex Arithmetic
  - S: Load Store



# Extracts from the Verification Plan

---

1. Check that **all pairs of instructions** are executed correctly together
  - Basic architectural requirement
  - Appears in most verification plans of processors
  - Fulfilling it is not as easy at it seems
2. Check that **all forwarding mechanisms** between pipeline stages are working properly
  - Basic microarchitectural requirement
  - Source for many bugs in previous designs



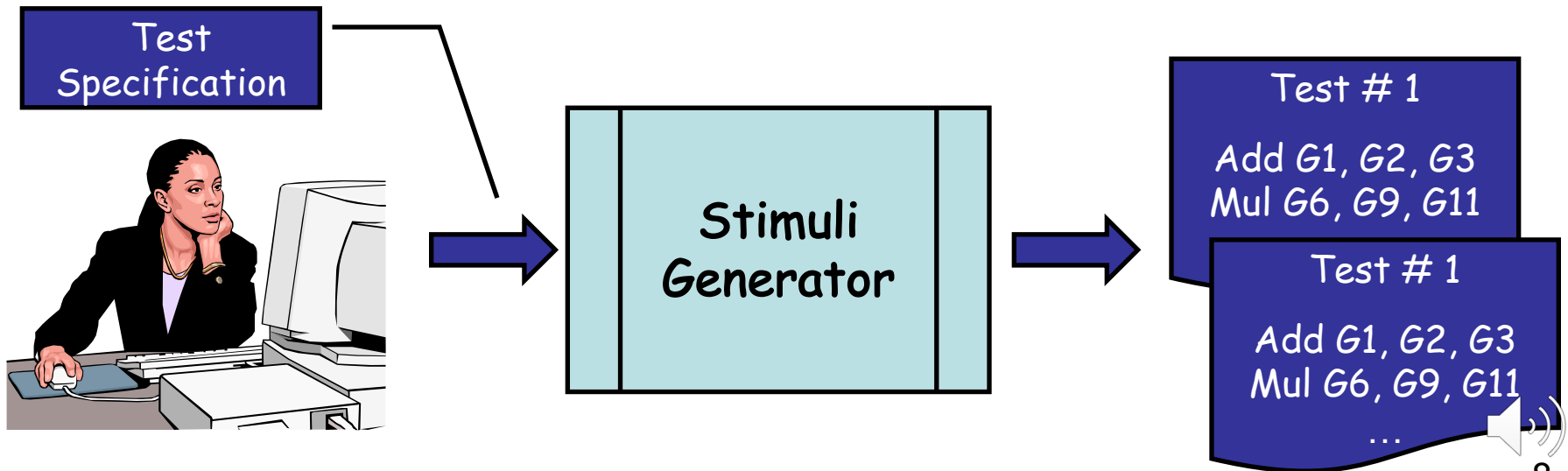
# Test Automation

From **random** test generation  
to **constrained pseudo-random** test  
generation



# Randomness - Motivation

- The first time we press the button a test is created
- What happens when we press the button a second time?
  - **The same test appears**
    - our stimuli generator is deterministic





# Why Deterministic?

---

- Useful before random environment is ready
  - It is much easier to create a driver that reads deterministic tests and injects them into the DUV
- Previously developed test suite
  - For example, architectural compliance suite
- Known quality
- Avoid (potentially) extremely long generation times

(Do not confuse deterministic with manual!)



# Why Not Deterministic?

---

- A given test can be used only once
  - It is useless unless something has changed in the
    - DUV
    - Environment
- The test specification has limited reuse capabilities
- Modern verification methodology employs many workstations that simulate many test cases
  - We cannot afford to provide different test specifications for each test case to be simulated

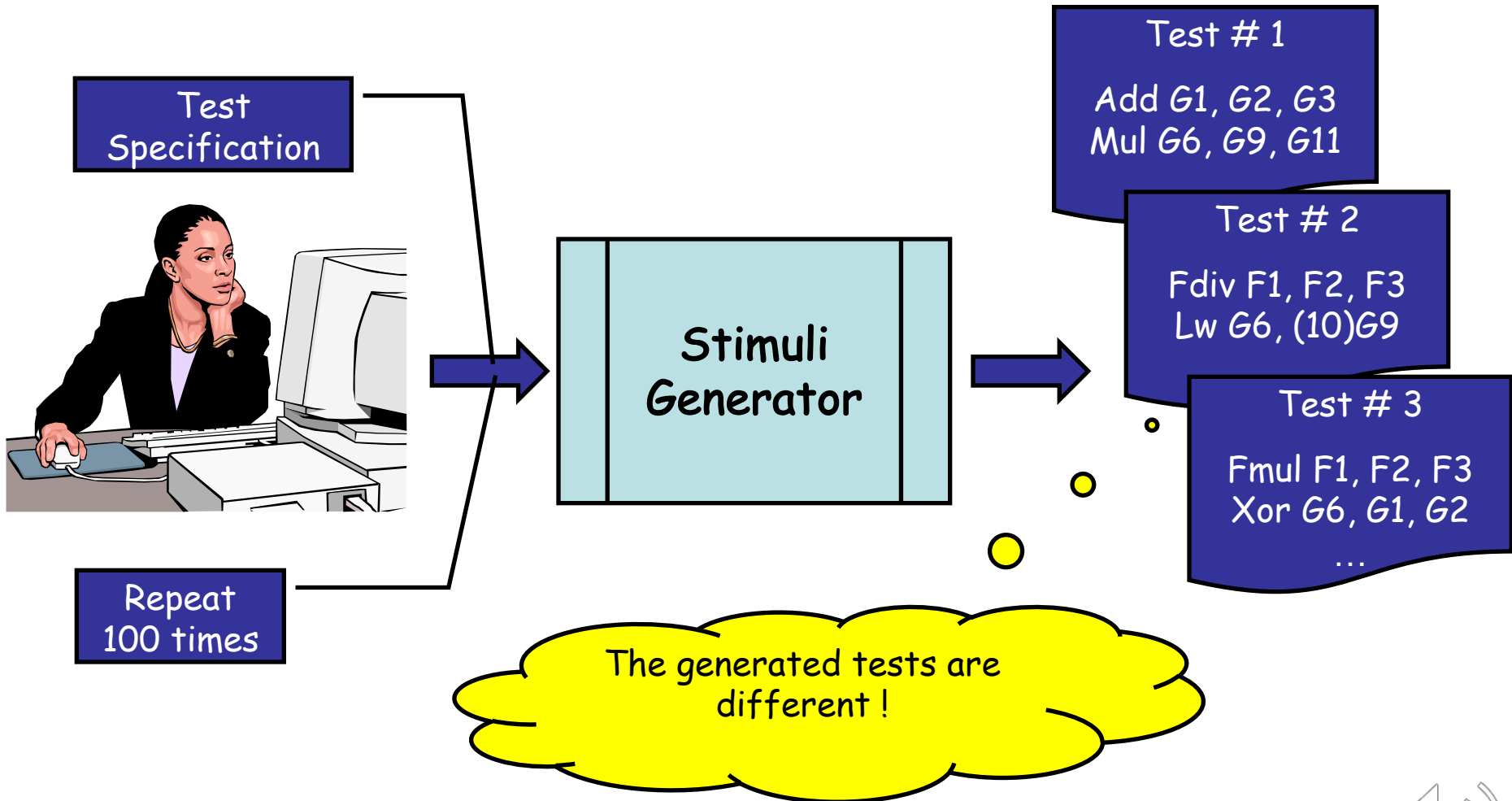


# Why Not Deterministic?

- A given test can be used only once
  - It is useless unless something has changed in the
    - DUV
    - Environment
- The test specification has limited reuse capabilities
- Modern verification methodology employs many workstations that simulate many test cases
  - We cannot afford to provide different test specifications for each test case to be simulated
- What about hitting and exposing all the problems we did not think about in the verification plan?



# Random Stimuli Generation



# Purely Random Generation

- The opposite end of the spectrum to deterministic generation
- The generator generates random sequences of '0's and '1's that are packed into instructions
- Theoretically, this might seem like the ideal solution
  - Avoid blind spots in the verification plan
- BUT practically,  
**not very useful for verification**
  - Most generated test cases are invalid
  - Most valid test cases are not interesting



# Side Note – *Pseudo Random*

- When using random number generators, “random” decisions are controlled by a **seed**
  - Given the value of the seed, random decisions are deterministic
- **Pseudo random** is essential in verification because of the need to reproduce specific tests
  - For example, to reproduce bugs
- Essential requirement for **Pseudo Random Test Generator**:
  - Need (at least) **repeatability!**
    - Achieved by using the same seed to seed the generator.



# Constrained Random Generation

- The stimuli generator is **constrained** to generate
  - **Valid** tests
  - Tests that meet the user requirements
- There are many (infinite number of) tests that fulfill these constraints
- The generator can choose any such test



# Example: Instruction Pair Generation

- The test specification requires a test that contains an **add** instruction followed by an **xor** instruction
  - Comes from the first extract of our verification plan
- The test should look like
- Everything else can be randomized

```
add_xor_test

Start:
...
Add ??, ??, ??
Xor ??, ??, ??
...
```





# Random Decisions for add\_xor\_test

- Registers of add instruction
- Data of add instruction
- Registers of xor instruction
- Data of xor instruction

... *but also*

- Prelude sequence
- Epilogue sequence
- Start address of the program
- Processor operation mode
- Behavior of caches, I/O, ...
- ...

```
add_xor_test

Start:
...
Add ??, ??, ??
Xor ??, ??, ??
...
```



# How To Make Random Decisions

- **Purely random** decisions
  - Most tests will be invalid
- **Constrained random** decisions
  - Limit random decisions to those that lead to **valid tests**
  - Choose uniformly among valid tests
  - Result
    - Generated tests are valid
    - Most random decisions are not interesting
      - ➔ Small gain in test quality
- **“Smart” constrained random** decisions
  - **Bias** decisions toward interesting cases
  - Can lead to significant **improvement in test quality**



# “Smart” Decisions for add\_xor\_test

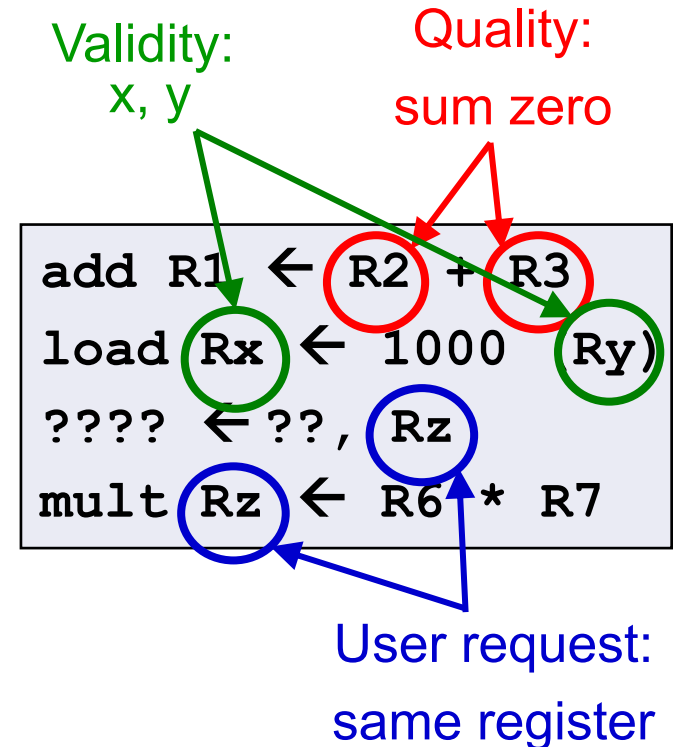
- Data of add instruction
- Registers of add instruction
- Registers of xor instruction
- Start address of the program

```
add_xor_test  
  
Start:  
...  
Add ??, ??, ??  
Xor ??, ??, ??  
...
```



# “Smart” Decisions for add\_xor\_test

- Data of add instruction
  - Result = 0
  - Overflow
  - Long sequences of ‘1’s (long carry chains)
- Registers of add instruction
  - special registers, e.g. G0 for PowerPC
- Registers of xor instruction
  - Same registers as used for add instruction to create data dependencies
- Start address of the program
  - Page 0
  - Start of page
  - Near end of page



These requirements can be expressed as *constraints*.



# Smart Decisions

- These decisions usually represent **generic knowledge of what is interesting in verification**

## Examples:

- Add with result 0 is interesting in all addition operations
  - Interdependency between registers is interesting in all processors
  - G0 is an interesting operand in all PowerPC processors
- This collection of knowledge is often called **“Testing Knowledge”**
  - The testing knowledge is usually **incorporated in the generation environment**
    - The generation tool you buy
    - The generator / driver you develop



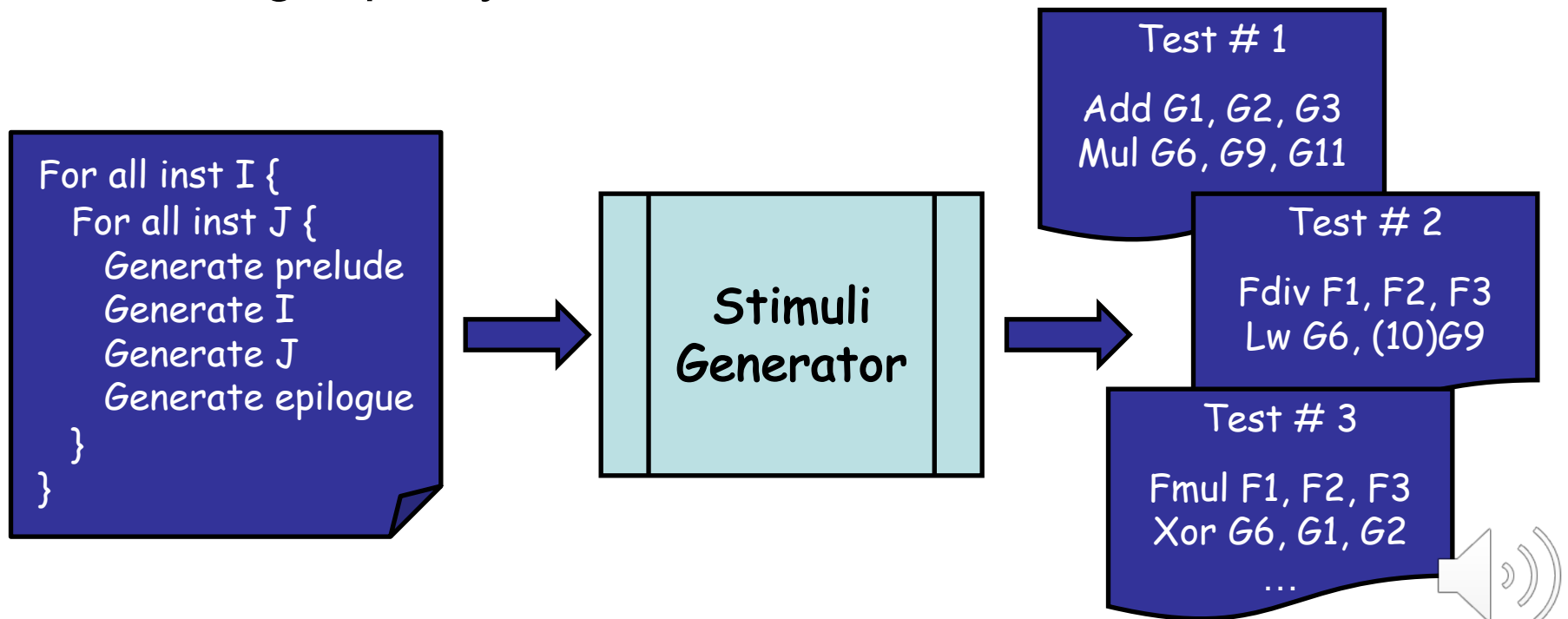
# Using Testing Knowledge

- Ideally, the testing knowledge can be applied automatically during stimuli generation
- The generator **biases** random decisions towards interesting scenarios using the testing knowledge
  - Other cases are not shut-down completely so that test generation can reach cases we never thought about.
- Stimuli generators that use testing knowledge are often called “**biased random stimuli generators**”
- Users can **change the bias** to reach verification goals



# All Instruction Pairs Generation

- With a **biased random stimuli generator** we can generate tests that cover all the specific items of the **all instruction pairs** extract from the verification plan
- Every activation of the test specification will produce a new high-quality test suite



- The same approach cannot work for the forwarding path verification requirement

2. Check that **all forwarding mechanisms** between pipeline stages are working properly
  - Basic microarchitectural requirement
  - Source for many bugs in previous designs

# Why?





# Abstraction level mismatch

- The same approach does not work for the forwarding path verification requirement
  - There is a difference between the language of the test and the language of the verification requirement
    - The test language is instructions, registers, memory
      - This is what we can influence
    - The requirement language (i.e. scenario in the verification plan) is based on **microarchitectural** events, e.g. control signals (flags) for the forwarding logic
      - This is our target wrt the verification plan and possibly functional coverage
- Three possible solutions
  - Manual translation
  - Automatic translation
  - “Loose” generation



# Manual Translation

---

- The user provides a description of an instruction sequence that creates the event
  - For example, mul followed by div followed by br, where br uses the same register as the target of mul
- The generator randomly fills in missing details
  - For example, registers and data of div
- Suffers from all the **disadvantages of manual test creation**
  - Labor intensive
  - Error prone
  - Hard to maintain



# Automatic Generation

- The generator is aware of the microarchitecture of the processor and knows how to translate a microarchitectural verification requirement into a sequence of instructions
  - Such generators are often called **“Deep Knowledge”** test generators
- Advantages
  - Generated tests cover the requested event with high probability
- Disadvantages
  - High development cost
  - Potentially long generation time
  - Sensitive to changes in the design
    - ➔ attract high maintenance costs



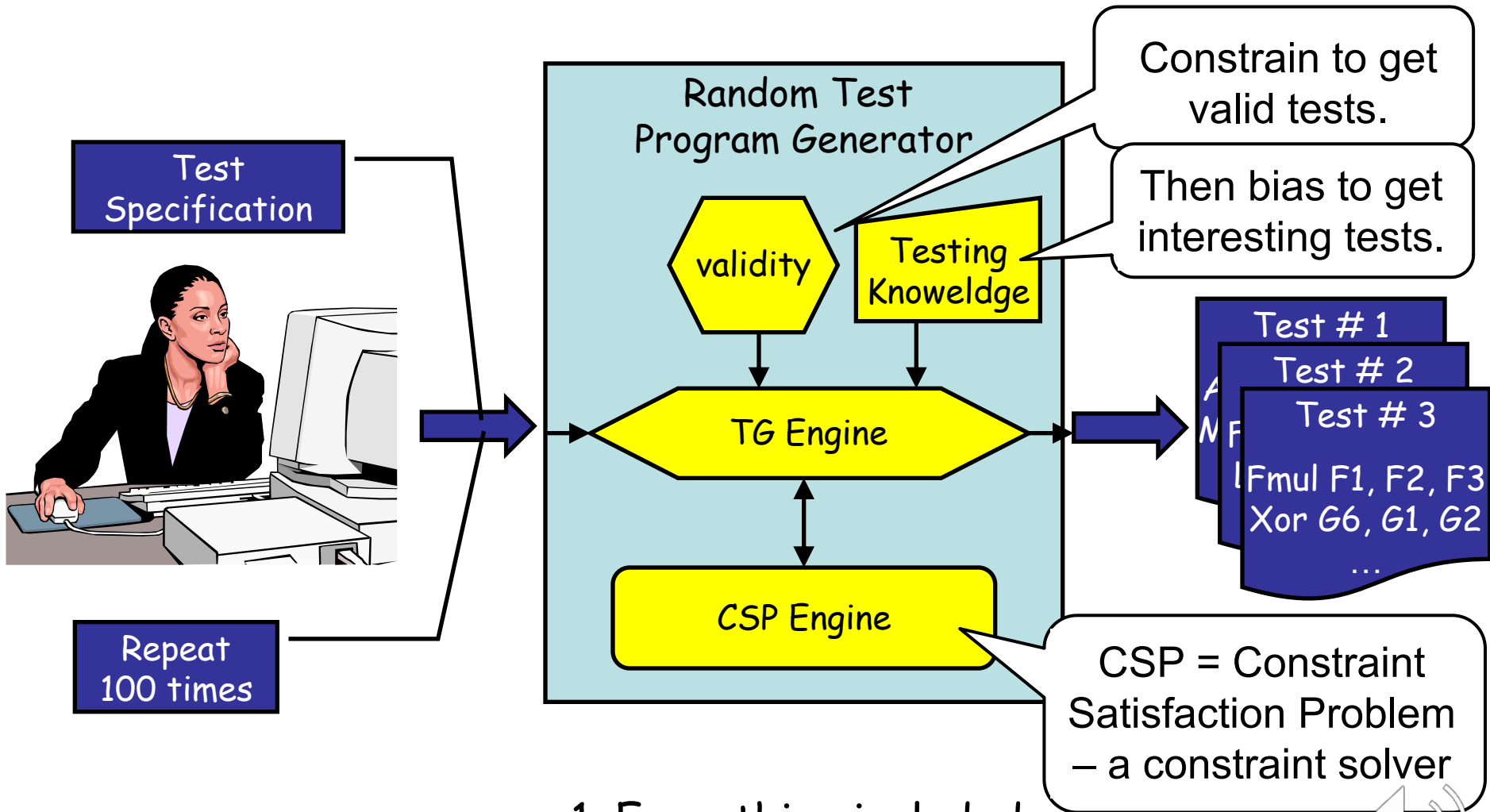
# “Loose” Generation

We exploit the power of *massive* generation:

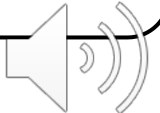
- Use the “normal” test vocabulary to **bias** the generated tests **toward tests that *improve the probability* of hitting the requested event**
  - Increase probability of complex arithmetic and branch instructions
  - Increase probability of read after write dependencies
    - Reduce the number of registers available
- How do we know whether this was successful, i.e. whether the desired events have been created?
  - **Coverage** is used to determine success
- In practice, this is an iterative process



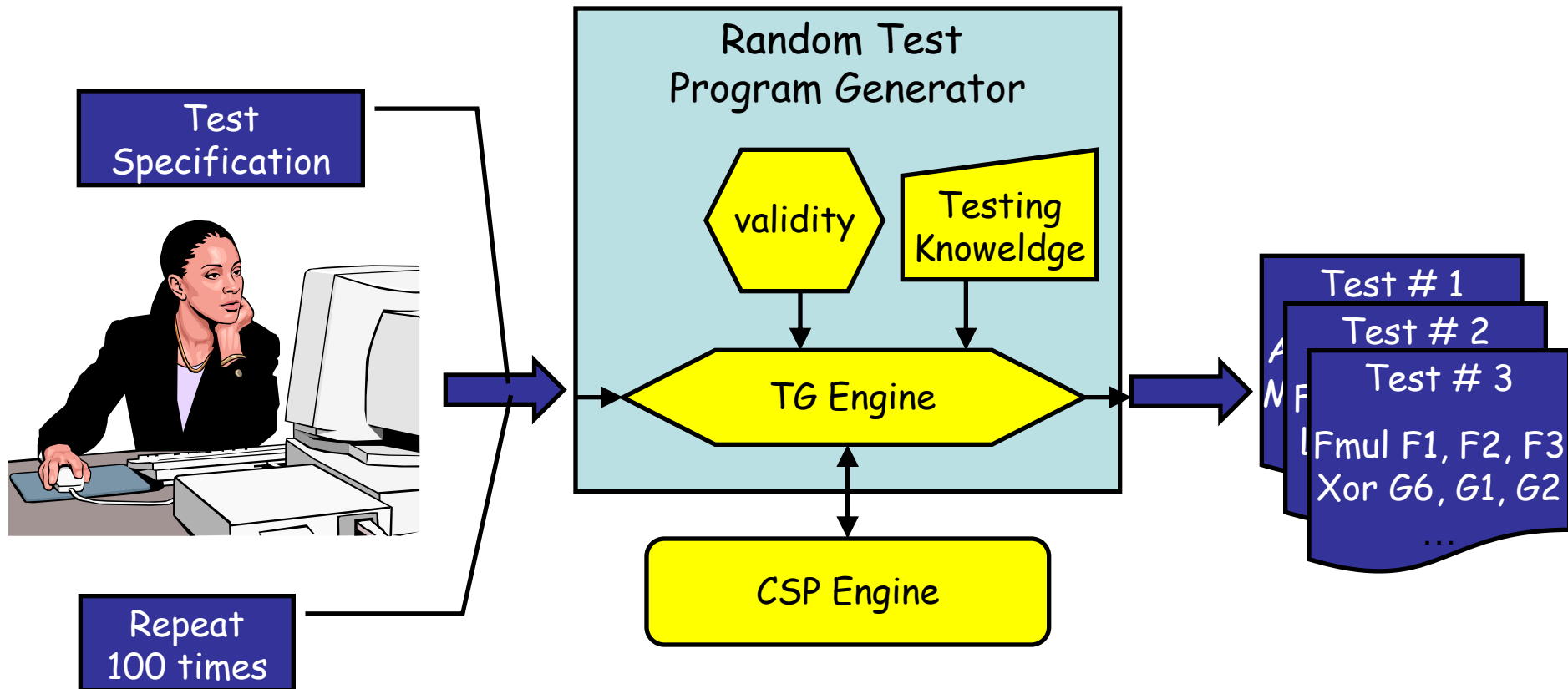
# Putting It All Together: Building a Random Test Program Generator - I



1. Everything included



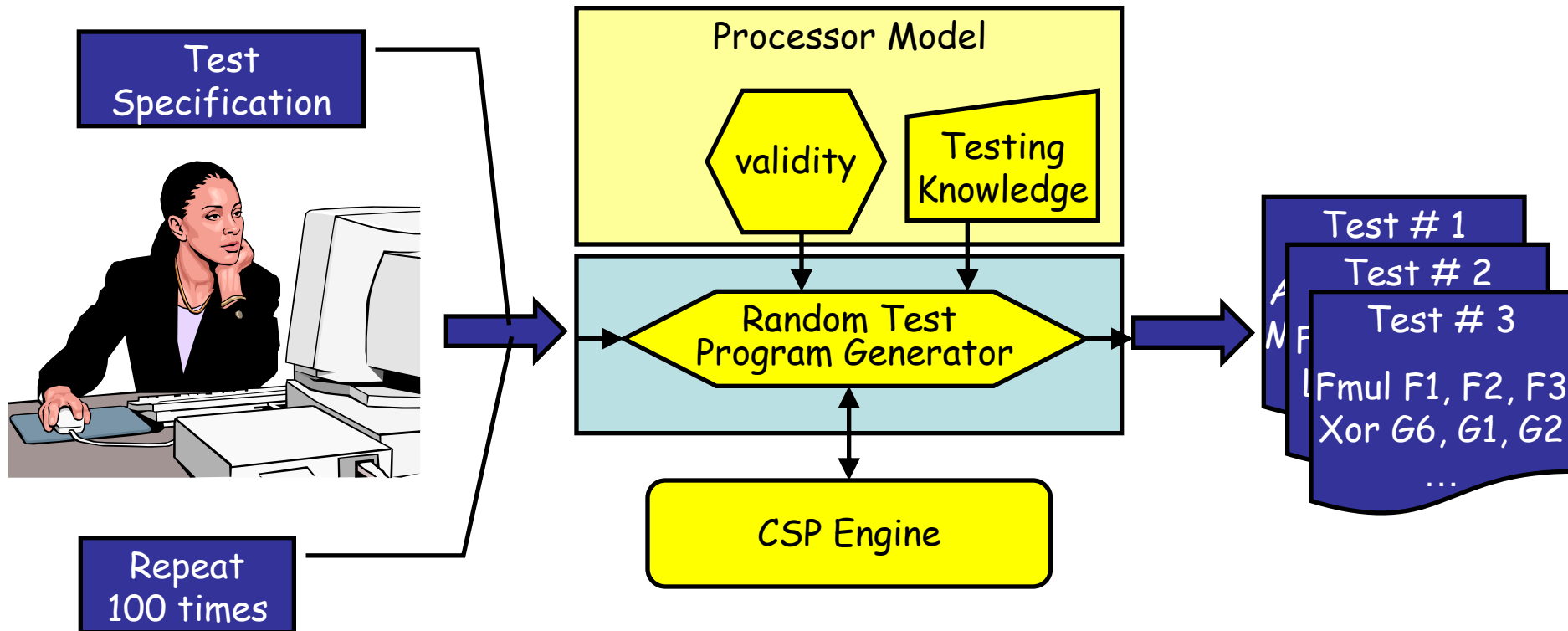
# Putting It All Together: Building a Random Test Program Generator - II



2. External CSP Engine



# Putting It All Together: Building a Random Test Program Generator - III



## 3. Model-based test generator



# Model-based Test Generator

## Three main layers:

- **General purpose CSP engine (solver)**
  - May be specific for stimuli generation, but can be shared among various tools
- **Processor model**
  - Description of a specific processor
    - Instruction set, registers, memory model, etc.
  - Testing knowledge specific to the processor
- **Processor test generation engine**
  - Knows about the concept, vocabulary of processors
  - Generic testing knowledge of processors
  - Can translate the user request, processor model, and testing knowledge into a CSP
  - Is capable of turning the solution to the CSP into a test program





# Summary: Stimuli Generation

- Generated stimuli need to be
  - **Valid**
    - NOTE: Valid is not necessarily legal
  - **Interesting**
    - Improve coverage
    - Reach corner cases
    - Find bugs
  - **Meet specific user requirements from the verification plan**
- Part I: Issues in stimuli generation
- Part II: From **random** test generation to **constrained pseudo-random** test generation

Homework challenge on test generation on next slide 😊



# REVISION: Main Principles of Test Generation

Online Generation (during sim)      Offline Generation (prior to sim)

Mainly Deterministic  
(i.e. written for a specific scenario)

Mainly Biased Pseudo  
Random (i.e. created using bias control)

This slide allows you to test your knowledge on test generation. Fill in the quadrants to see whether you would have got these combinations right. ;)



# REVISION: Main Principles of Test Generation

Offline Generation  
(prior to sim)

Online Generation  
(during sim)

Mainly Deterministic  
(i.e. written for a specific scenario)

Single scenario test.  
Usually **written by hand** to verify a specific scenario.

Most often **early** in the verification process.

Single scenario test cases with **some random generation** of peripheral inputs.

Random generations used only for inputs not critical to the test case intent.

Mainly Biased Pseudo Random (i.e. created using bias control)

Test case **generators** using random parameters to bias the stimulus.

**Architecturally correct tests** are created and then exercised via simulation.

Stimulus generated **each cycle** using parameter biasing to determine that cycle's input.

The environment must have the knowledge of legal and illegal scenarios.

