

COMS31700 Design Verification: **Stimuli Generation**

Kerstin Eder

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

Last Time

- Coverage
 - Code
 - Structural
 - Functional
- Coverage analysis
 - Hole analysis

(Discuss plan for coming weeks!)

Outline

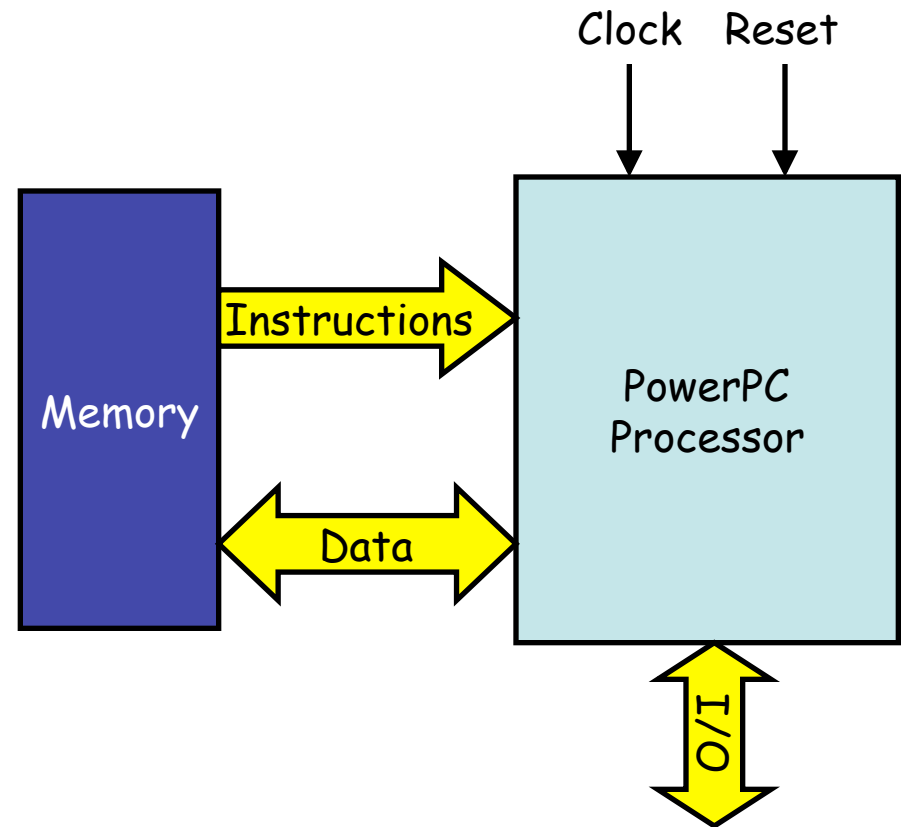
- Motivation: Advanced Stimulus Generation
- Running example – PowerPC processor
- Issues in stimuli generation
 - Level of stimuli, test length, etc.
- Randomness
- Constrained pseudo-random stimulus generation

Goals of Stimuli Generation

- Achieve all the items in the test scenarios matrix of the verification plan
 - Ensure that the scenarios in the matrix are happening
 - Ensure that “bad effects” are propagating to an existing checker
 - Hitting a bug without exposing it is worth nothing
- But also
 - Hitting and exposing all the problems we did not think about in the verification plan
 - Provide information about the design and help recreate and understand problems
 - Ensure that nothing gets broken over time

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
 - ...

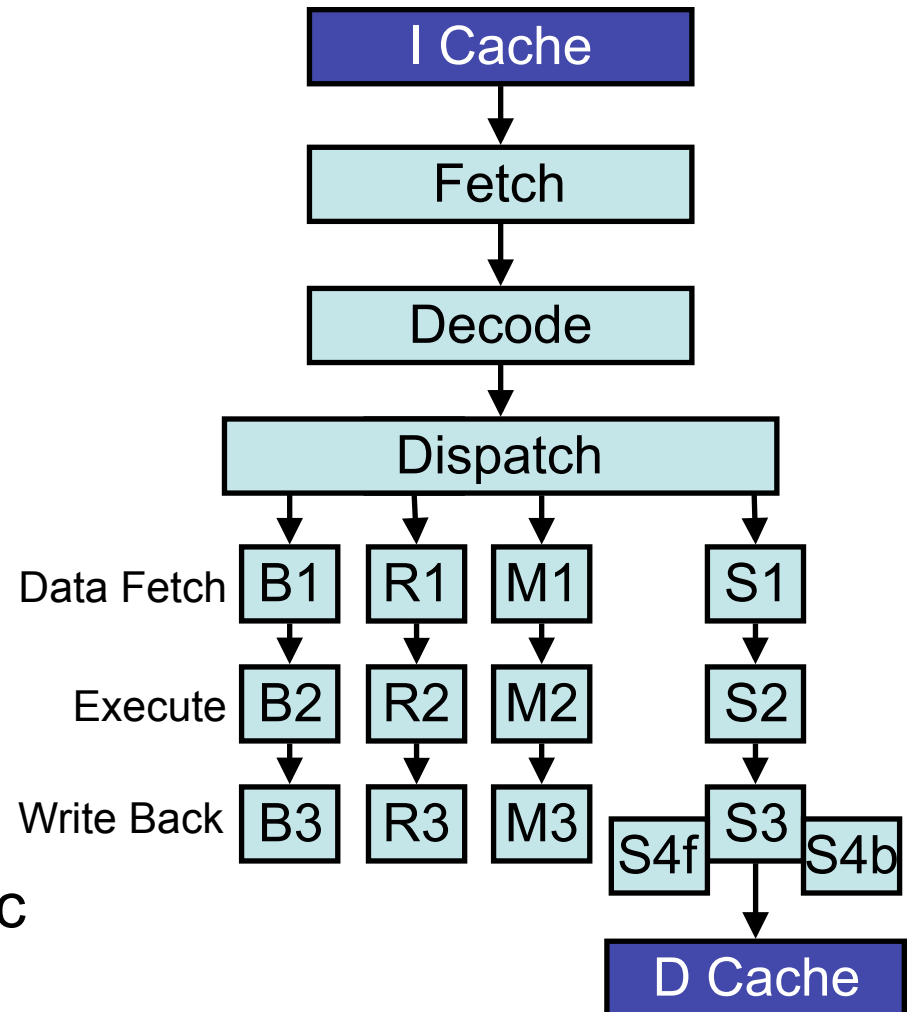


Architectural View

- 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

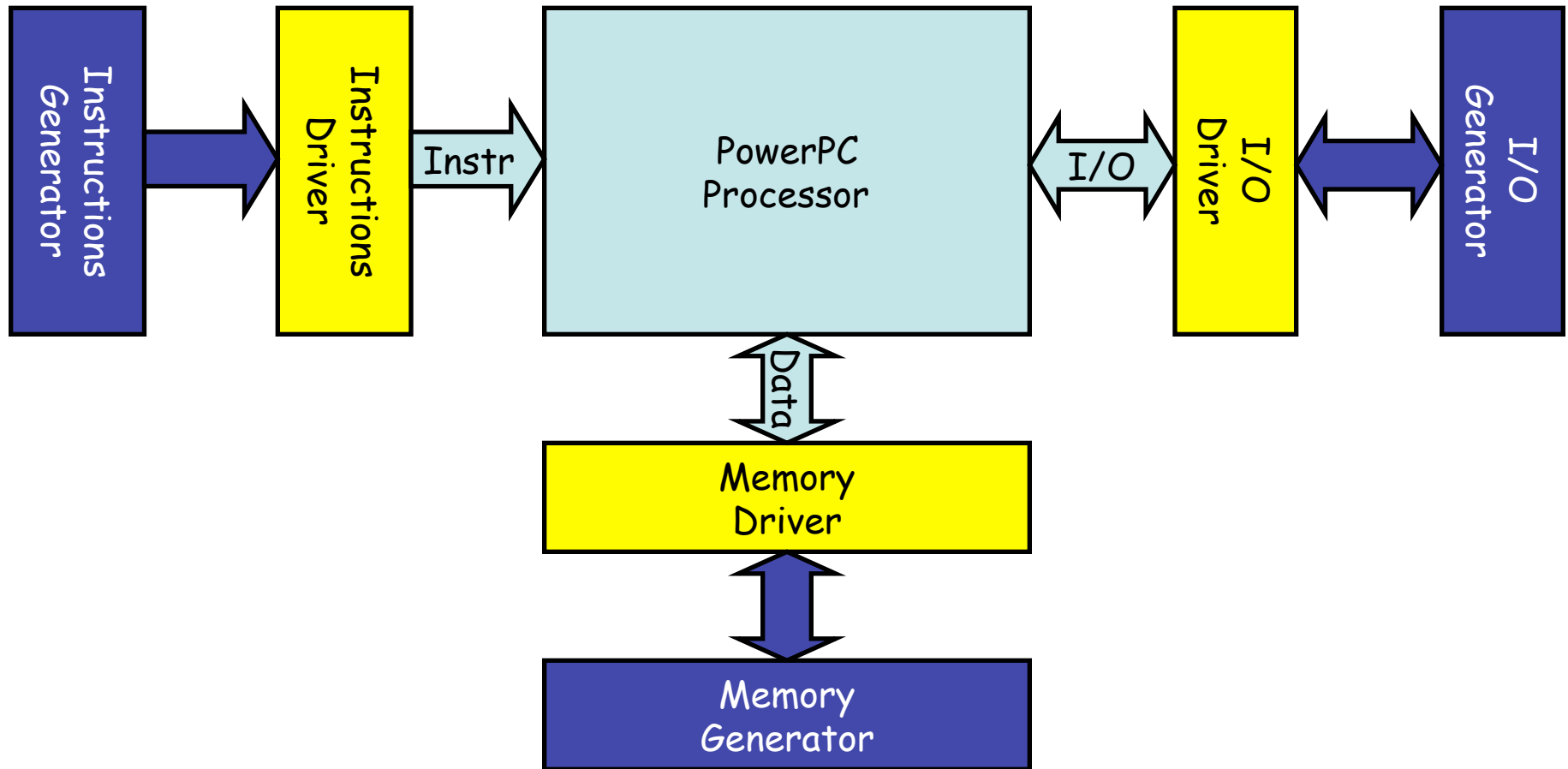
- Multi-threaded
- In-order execution
- Four instructions wide
 - Fetch
 - Decode
 - Dispatch
- Four execution units
 - B – Branch
 - S – Load Store
 - R – Simple Arithmetic
 - M – Complex Arithmetic



Extracts from the Verification Plan

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

Processor Verification Environment

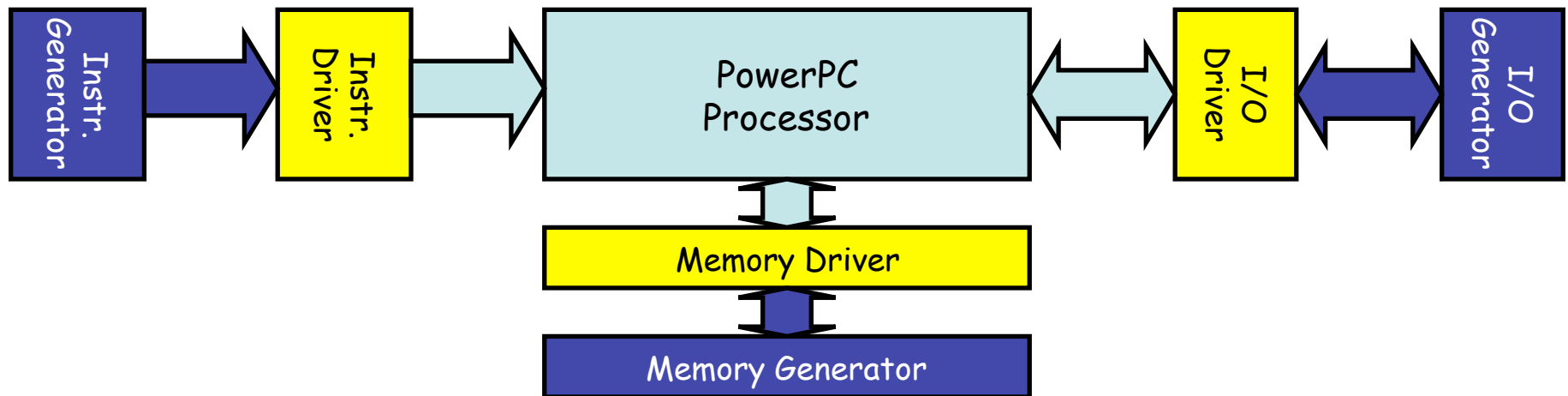


Issues in Stimuli Generation

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

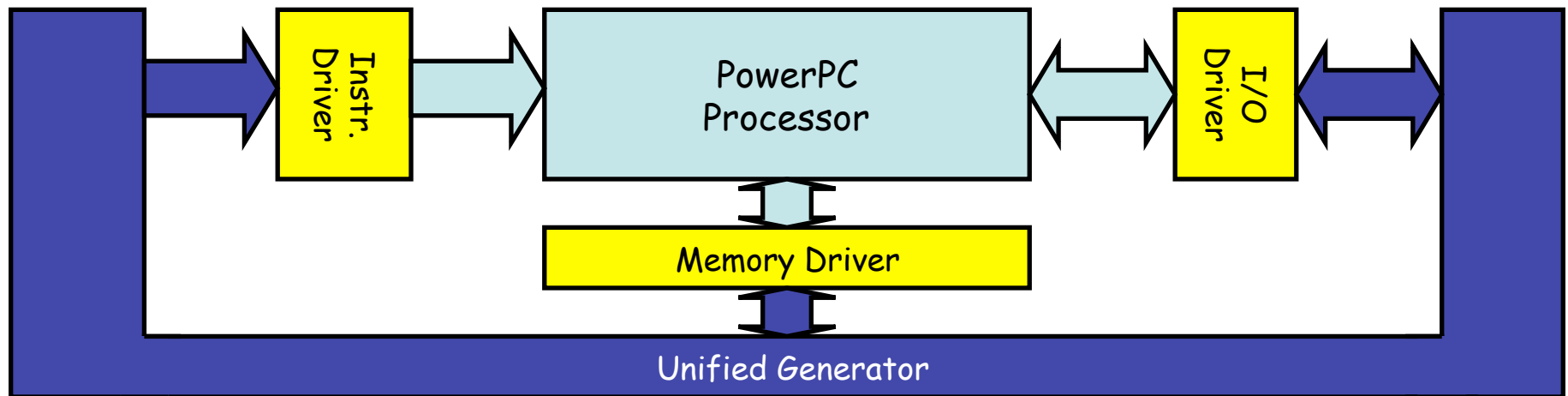
How Many Generators?

- Distributed generators
 - Each interface has its own generator
 - Each generator works on its own
 - Advantages
 - Simple
 - Easy to reuse
 - Disadvantages
 - Hard to reach corner cases in coordinated fashion



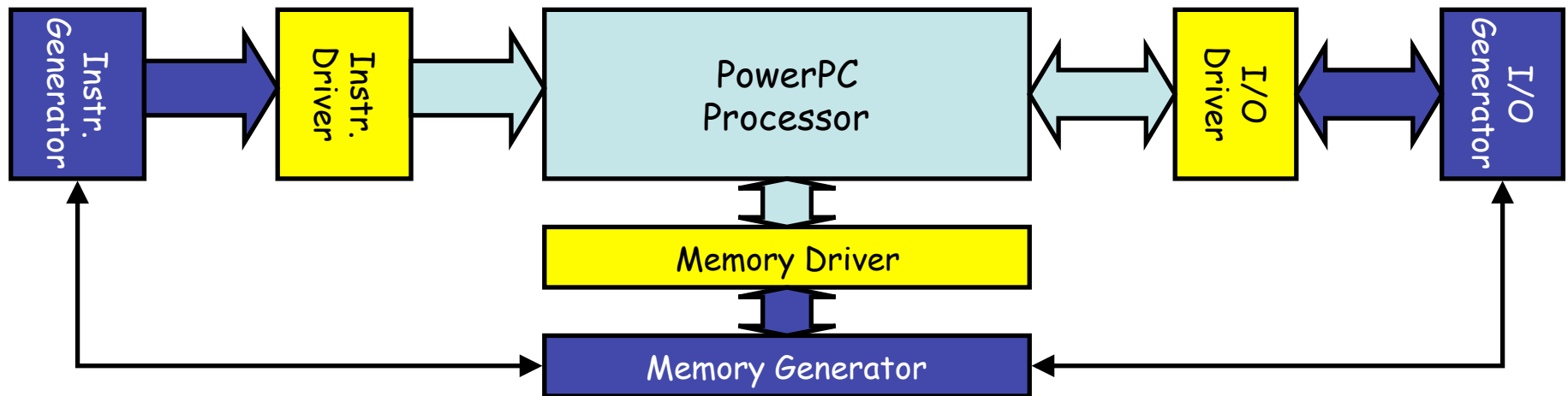
How Many Generators?

- Single generator
 - One generator controls all the interfaces
 - Advantages
 - All the interfaces can work together toward a common goal
 - Disadvantages
 - Complex
 - Hard to reuse

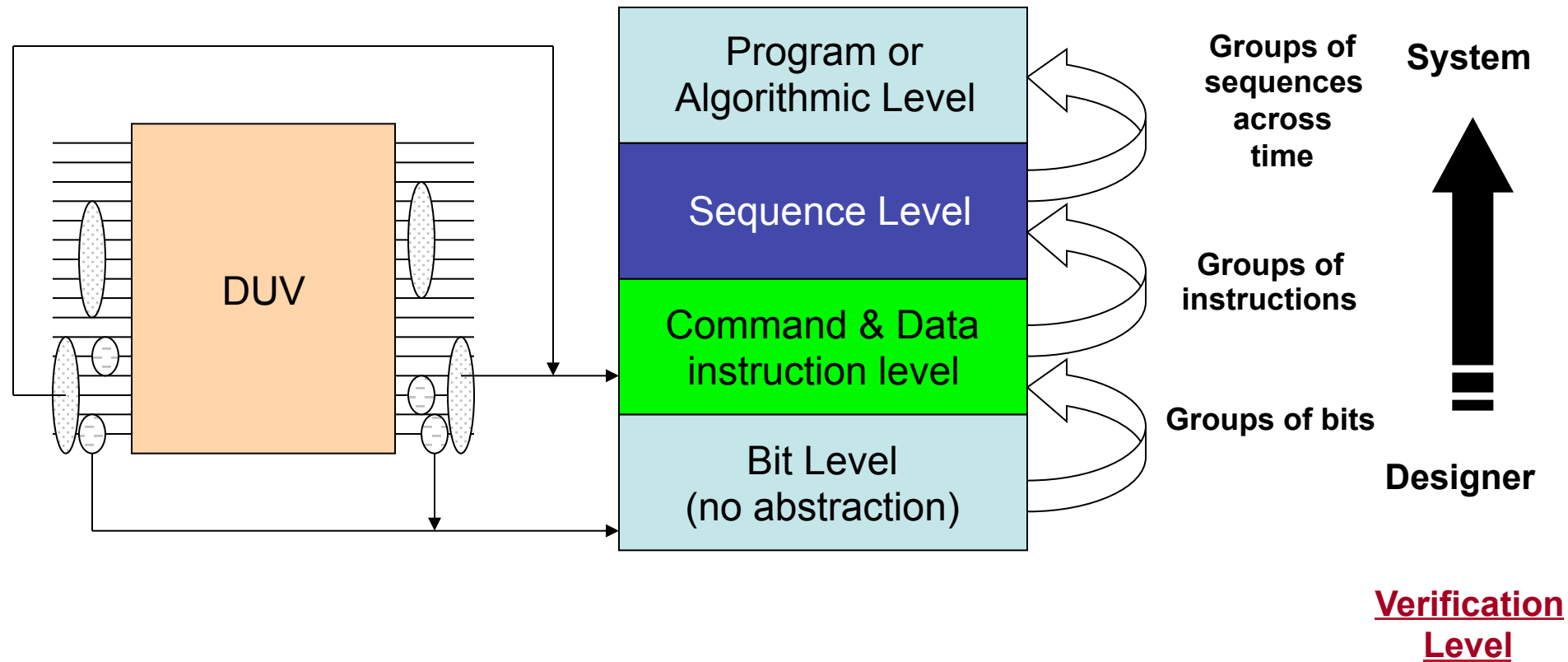


How Many Generators?

- Synchronized generators
 - Each interface has its own generator
 - The generators share information and synchronize
 - Advantages
 - Can reuse each generator separately
 - Can work together towards a common goal



Abstraction Level of Generation



What Does Abstraction Level Mean?

- Communication between:
the user and the generator
 - How the user specifies directives to the generator
- Internal representation and operation level in the generator
 - The level in which the generator generates the stimuli
- Communication between:
the generator and the driver
 - The generator sends information at high level of abstraction
 - The driver translates into bits using the appropriate protocol

Which Abstraction Level To Choose?

- Communication **between the user and the generator**
 - Use level similar to the level of the verification plan
 - In our case – the sequence level
- **Internal representation and operation level** in the generator
 - Conflicting requirements
 - Address user requests (at their level) → high level of abstraction
 - Need sufficient detail → low level of abstraction
 - In many cases we use two or more levels of generation
 - First we build a high-level skeleton of the stimuli based on the user request
 - Next we add lower-level details
- Communication **between the generator and the driver**
 - Use the lowest level in which the generator operates
 - Special case – error injection

Error Injection

- Error detection and recovery are very important mechanisms in hardware designs
 - They are also very hard to verify
- Error injection is usually **done at the lowest level of abstraction**
 - The value of a bit (or set of bits) is flipped when they are injected into the DUV
- To allow error injection, the generator needs to operate and **communicate** with the driver **at the bit level**
 - This creates extra burden and unnecessarily increases complexity for normal cases
- Possible solution – create **separate error injection interface** between the generator and driver
 - At the **low level** of the error injection
 - At the **normal level** with instructions on how to inject the error

Online Vs Offline Generation

When to generate stimuli?

- **Offline** generation (pre-run):
 - The entire stimuli are generated **before the simulation** begins
 - The generation and simulation can be two separated processes
- **Online** generation (on-the-fly):
 - Stimuli generation **during simulation**
 - The next element is generated when needed by the driver
 - The generator must be part of the verification environment

Offline Generation

■ Why

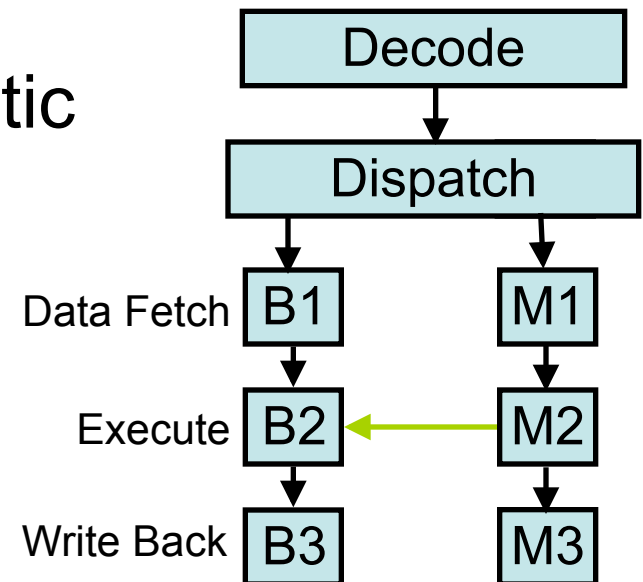
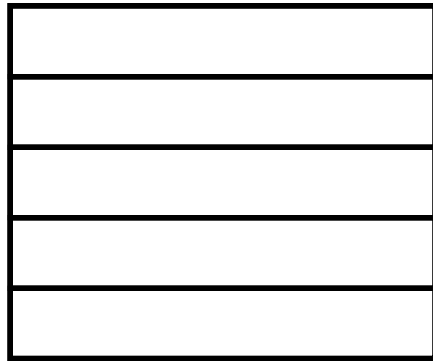
- Can separate the generation from simulation
 - Use external tools, emulation, ...
- Can use more complex algorithms for generation
 - For example, **generate out of order**, e.g. instruction sequences (processors) or action sequences (robotics)
- May be compulsory (Where?)

■ Why not

- Need to connect the generation output to the verification environment
- Cannot use information directly from the DUV and environment
- Hard to react to responders

Generating Instructions Out Of Order

- Goal – forward data from M2 to B2
 - Branch is dispatched after arithmetic instruction
 - Both reach stage 2 together
 - Branch waits for the arithmetic instruction to complete



How can we generate a test, i.e. a sequence of instructions, that achieves this goal (efficiently and effectively)?

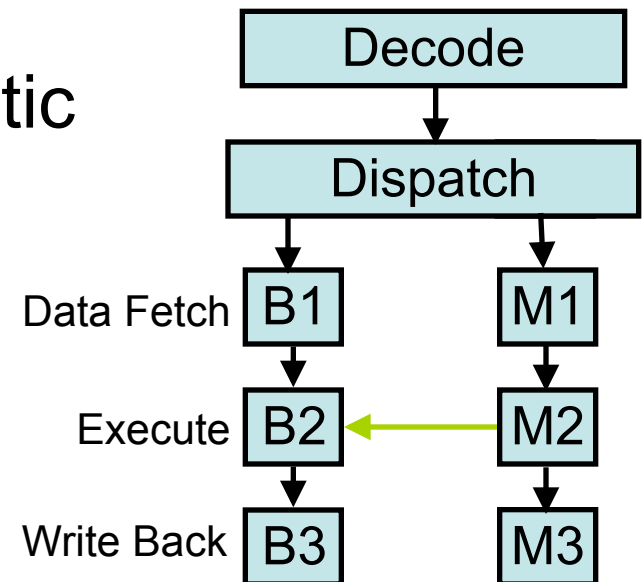
Generating Instructions Out Of Order

- Goal – forward data from M2 to B2
 - Branch is dispatched after arithmetic instruction
 - Both reach stage 2 together
 - Branch waits for the arithmetic instruction to complete

4	Lw G10, 60(G21)
4	Add G7, G9, G13
2	Mul G1, G2, G3
3	Div G4, G5, G6
1	Br 100(G1)

Generation Order: **Br** – **Mul** – Div – Lw – Add

Execution Order: Lw – Add – **Mul** – Div – **Br**



Online Generation

- Why
 - The generator can use information about the **state** of the environment and DUV for improving the quality of generation
 - Makes reaching corner cases easier
 - The only solution for responders
 - Generally small memory footprint
- Why not
 - Must generate items in order
 - Limited complexity
 - <any other reasons why not>

Mixing online and offline Generation

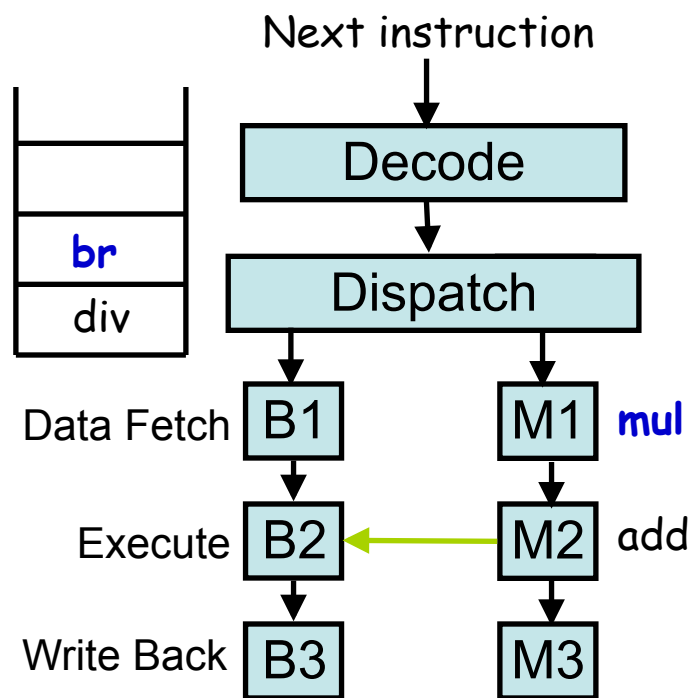
- Online and offline generation can be mixed within a verification environment
- Which designs would benefit from this combination?
 - (Example will be discussed in lecture.)

Dynamic vs. Static Generation

- In **static generation** the generator is not aware of the state of the DUV and the environment
 - Generation decisions are based entirely on the internal state of the generator
 - Less restrictive view: the generator is aware of what and when it is allowed to generate
 - In calc1 the generator knows not to generate a new command before a response for the previous command has been received
- In **dynamic generation** the generator is fully aware of the state of the DUV and the environment and generates based on this information
 - The generator can react to interesting states in the DUV

Dynamic Instruction Generation Example

- Goal – forward data from M2 to B2
 - The generator identifies the potential forwarding condition “on the fly”, i.e. when it spots the **mul** instruction
 - It generates instruction(s) that will block the **br**(anch) from dispatching with the **mul** instruction
 - It generates a **br** instruction with the same register as the destination of the **mul** instruction to create the dependency that triggers forwarding



Does This Example Work?

- This example may not work!
- Main reason:
 - There is a distance (in terms of time) from the entry point of instructions to the processor to the dispatch queue. This distance creates delays.
 - Many bad things can happen while the br instruction travels this distance
 - For example, exceptions that flush the pipes
 - By the time the br instruction reaches the relevant stage in the pipe to trigger forwarding, the interesting condition may already have gone

Dynamic Vs. Static Generation

- Dynamic generation is based on reaction while static generation is based on planning
- In general, reaction is harder than planning
 - Time is a factor
 - Unexpected events can get in the way
- Most generators use dynamic features lightly
 - Observe and react to shallow or stable states and resources
 - For example, architectural registers

Offline Dynamic Generation

- *Dynamic and static* generation should not be confused with *online and offline* generation
- An offline generator can use dynamic generation by using a **reference model** that provides information about the state of the DUV
 - The level and accuracy of the information depends on the abstraction level and accuracy of the reference model

Test Length

- Two extreme approaches for selecting the test length
- Use **short tests**
 - The shortest tests that can fulfill the requirement in the verification plan
 - For the instruction pairs requirement use tests with just two instructions 😊
- Use **long tests**
 - Combine many requirements in a single test
 - Wrap a test with initial and ending sequences

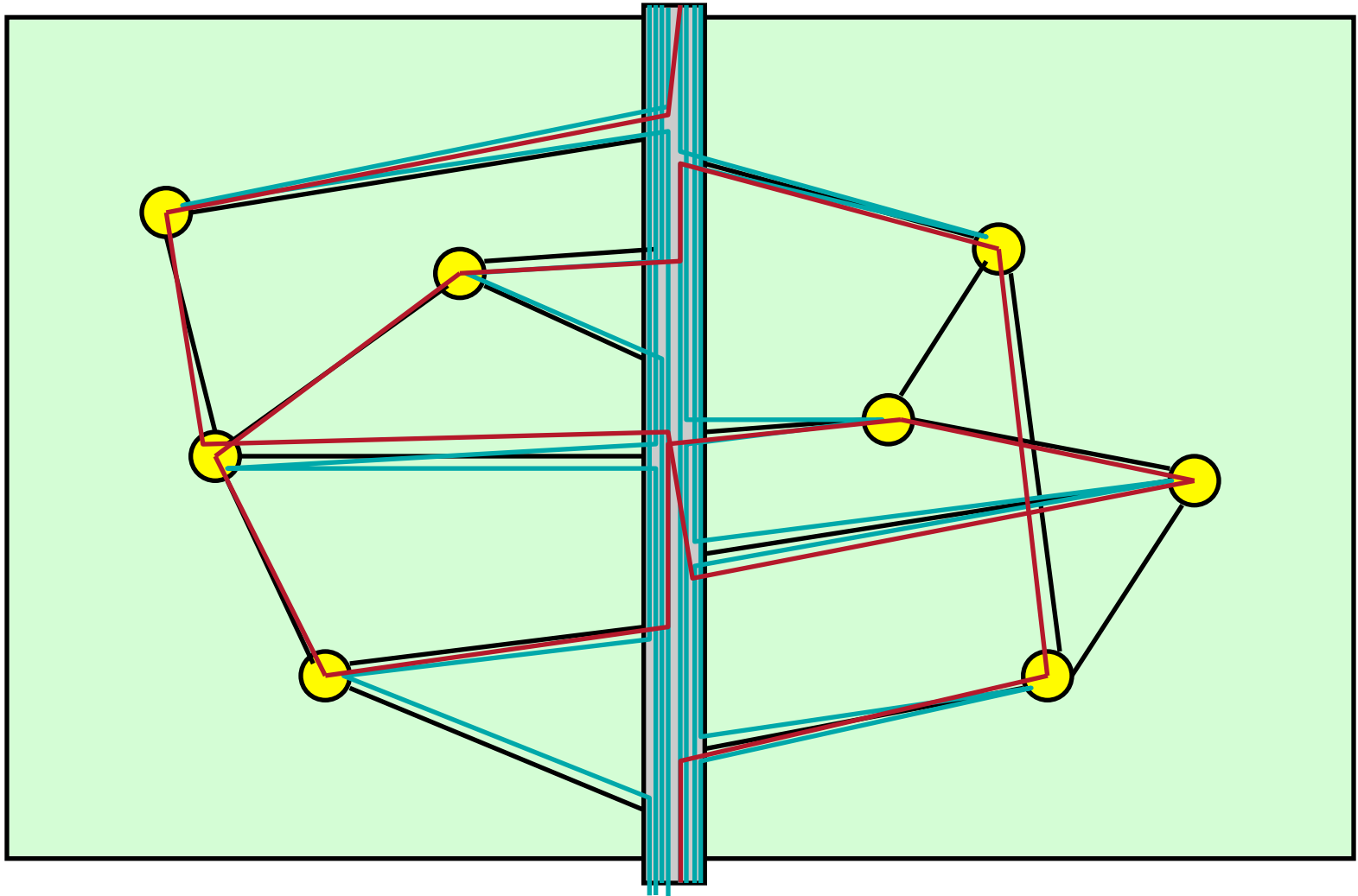
Why Short Tests?

- Easy to create
- Easy to debug
- Easy to maintain
- Short time to simulate each

Why Long Tests?

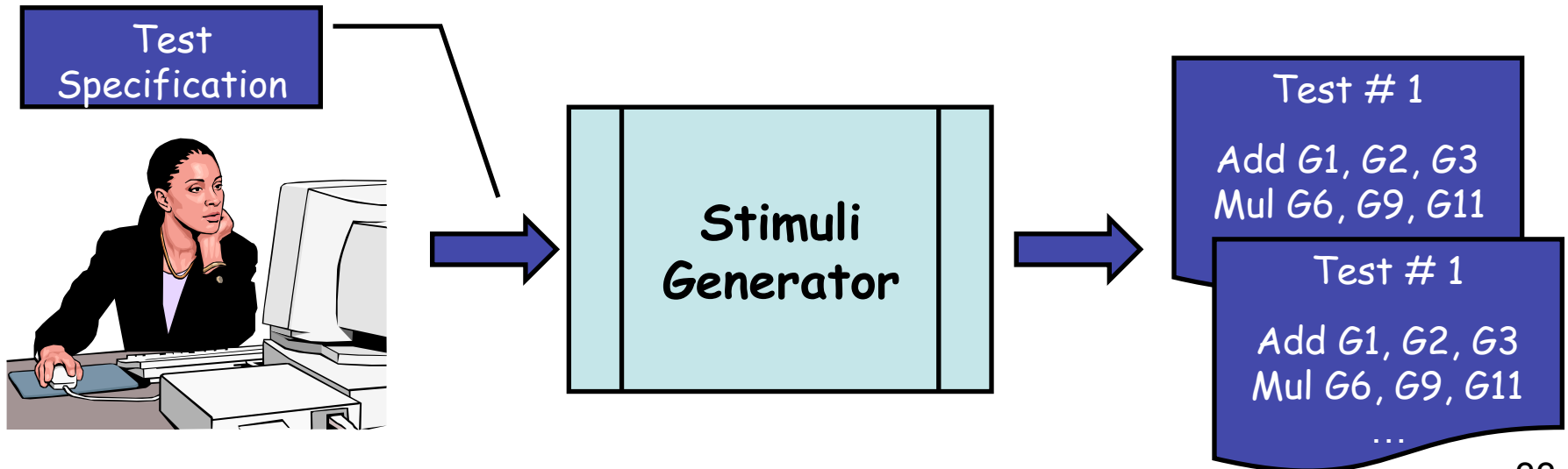
- Need fewer tests
- Less time to simulate
 - Overall less time as we do not need to repeat the initialization sequence for every test ;)
- Test is not at or near the initial state most of the time
- Use less traveled paths and greater variety of exploration
- Reach target in more ways
 - Often leads to reaching the target in unexpected ways

Short Vs. Long



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?

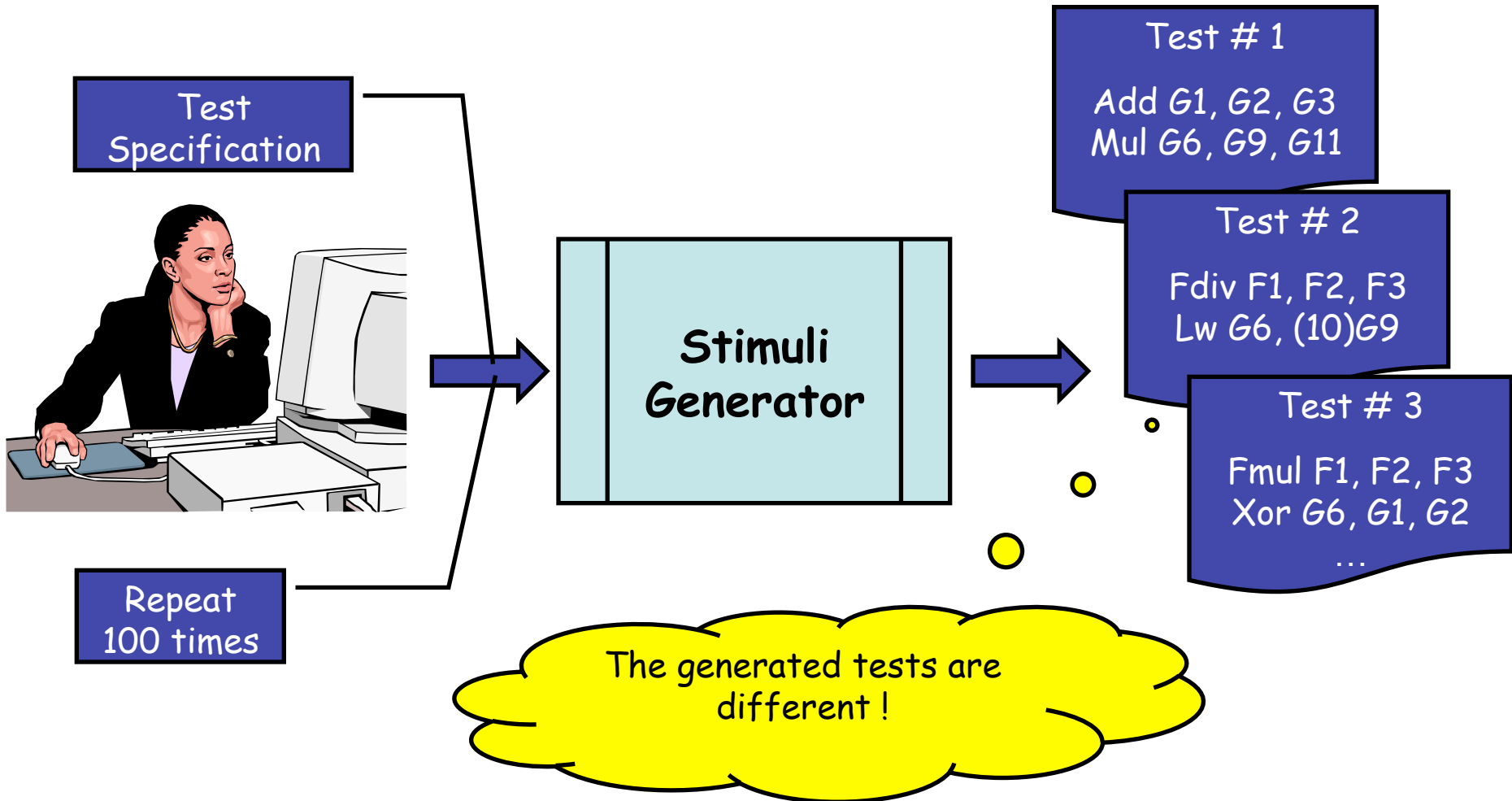
- Useful before random environment is ready
 - It is much easier to create a driver that reads deterministic tests and injects them into the DUV
- Previously developed test suite
 - For example, architectural compliance suite
- Known quality
- Avoid (potentially) extremely long generation times

(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
- What about hitting and exposing all the problems we did not think about in the verification plan?

Random Stimuli Generation



Pure 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.

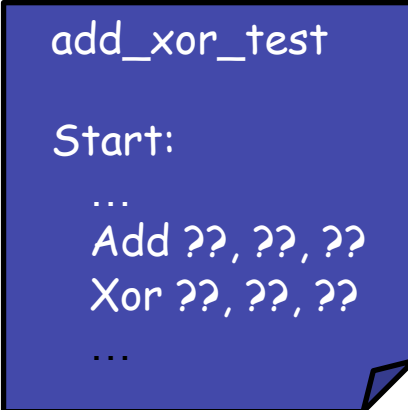
Constrained Random Generation

- The stimuli generator is **constrained** to generate
 - **Valid** tests
 - Tests that meet the user requests
- 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 is a test with 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

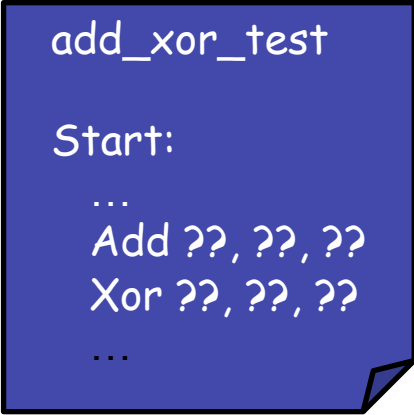
Start:
...
Add ??, ??, ??
Xor ??, ??, ??
...
```


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 ??, ??, ??
...
```

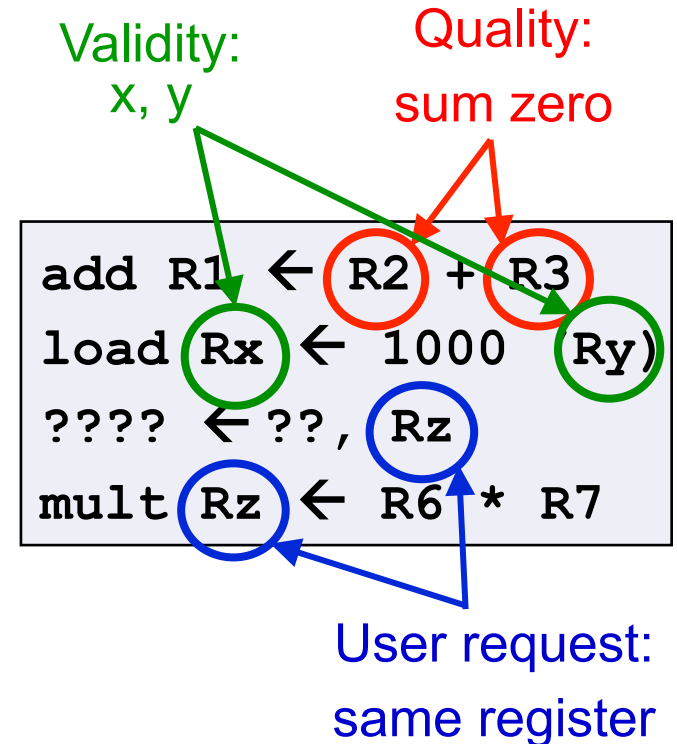
How To Make Random Decisions

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



“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 dependencies
- Start address of the program
 - Page 0
 - Start of page
 - Near end of page



These requirements can be expressed as *constraints*.

Smart Decisions

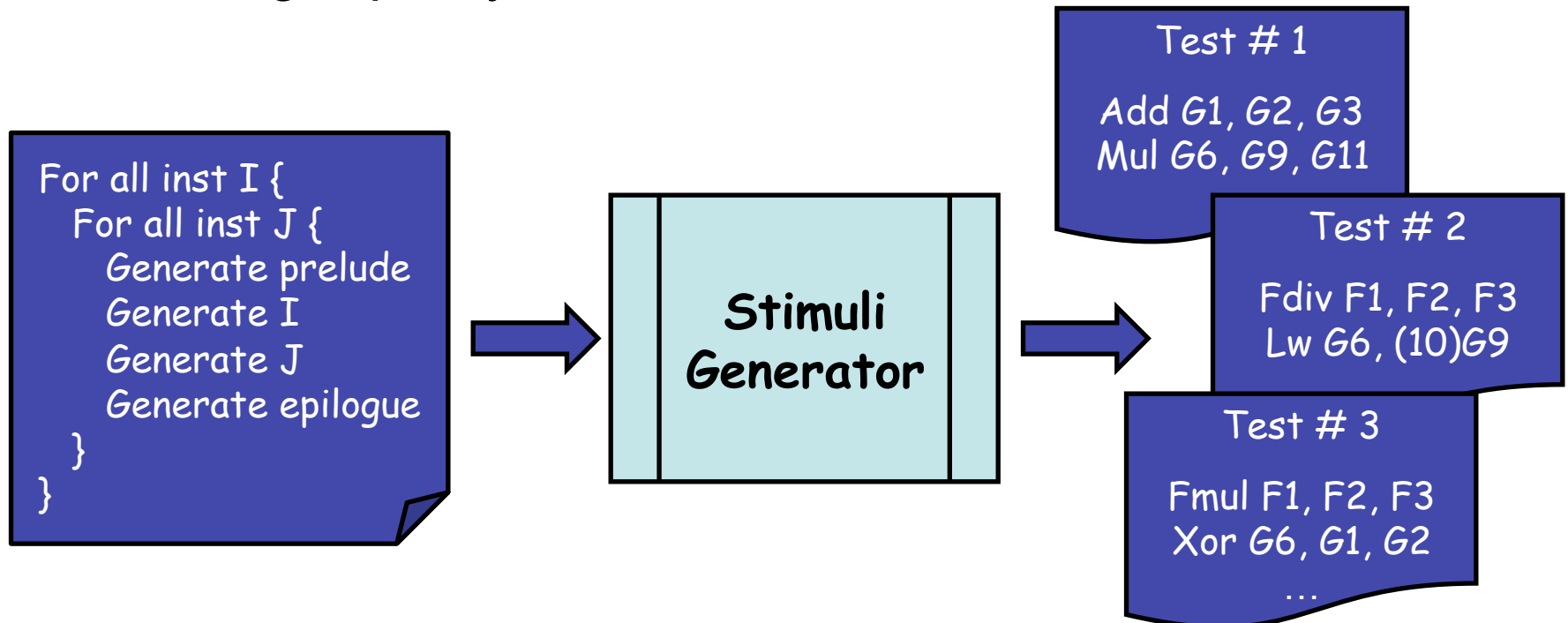
- These decisions usually represent **generic knowledge of what is interesting in verification**
 - Examples:
 - Add with result 0 is interesting in all addition operations
 - Interdependency between registers is interesting in all processors
 - G0 is an interesting operand in all PowerPC processors
- 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 to avoid missing 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
 - We will see examples later and can explore how this works when developing the testbench for A2. 😊

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



Abstraction level mismatch

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

Why?

Abstraction level mismatch

- The same approach cannot work for the forwarding path verification requirement – **Why?**
 - There is a difference between the language of the test and the language of the 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

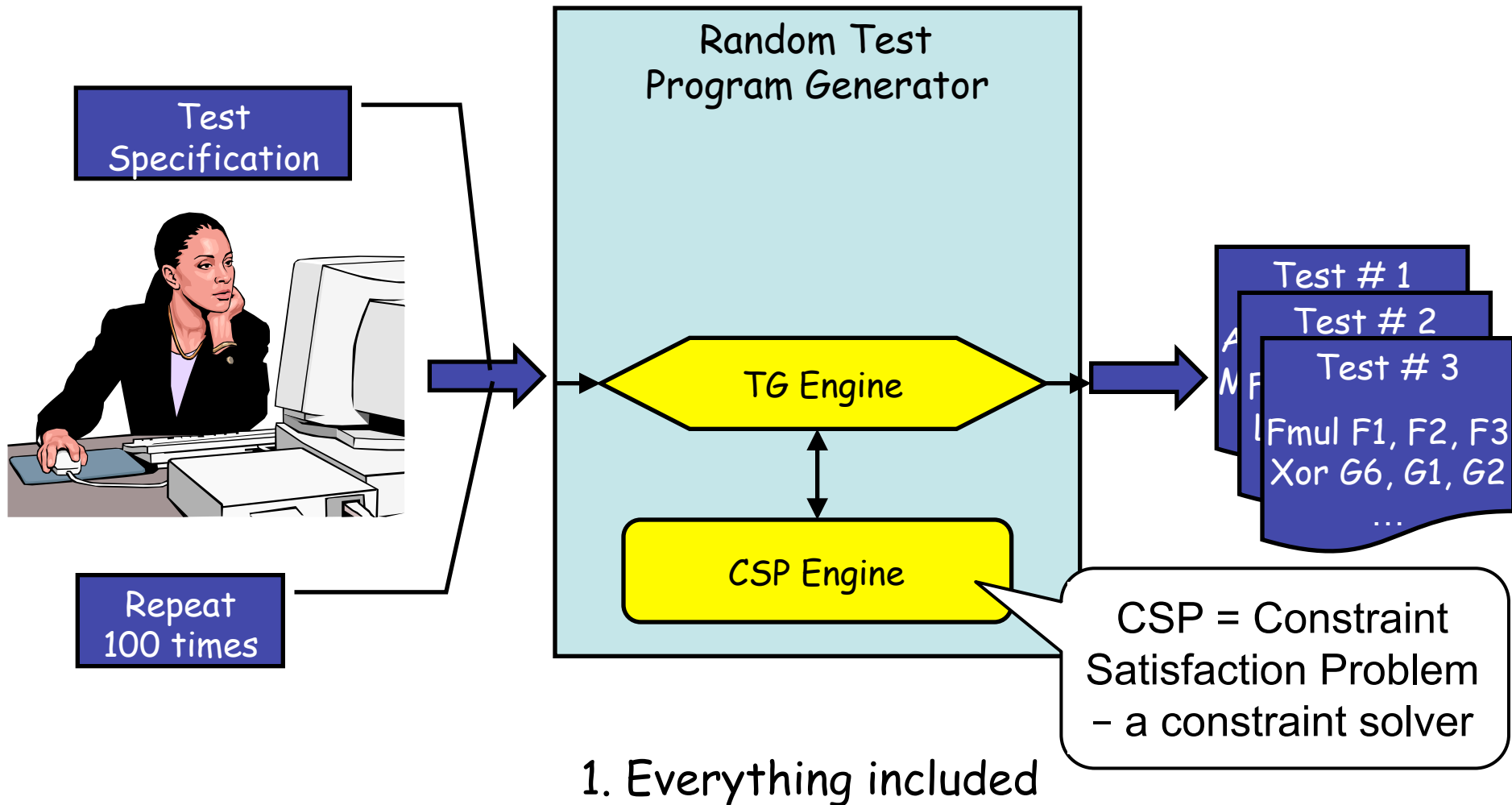
- The generator is aware of the microarchitecture of the processor and knows how to translate a microarchitectural request to a sequence of instructions
 - Such generators are often called **“Deep Knowledge”** test generators
- Advantages
 - Generated tests cover the requested event with high probability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

“Loose” Generation

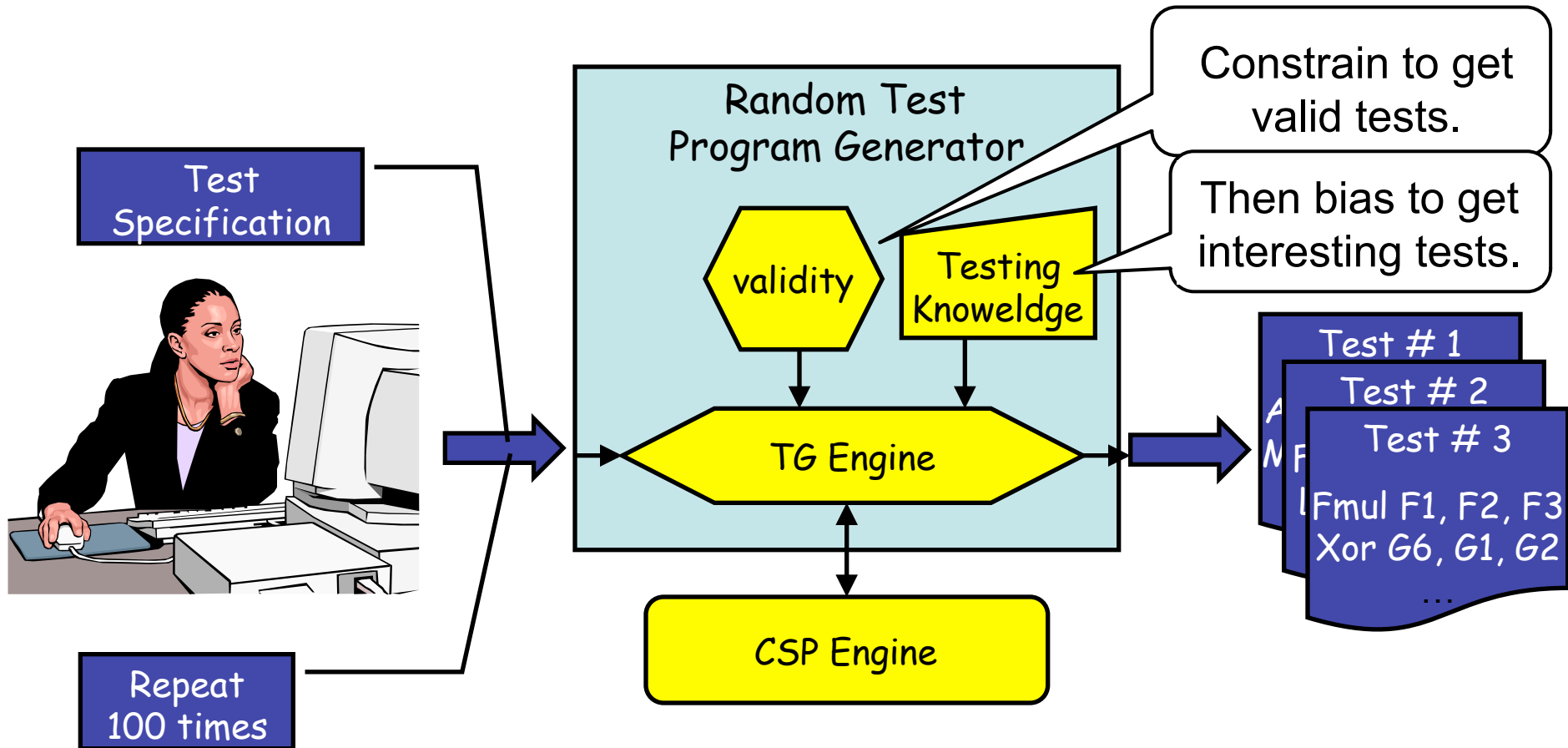
We exploit the power of *massive* generation:

- 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



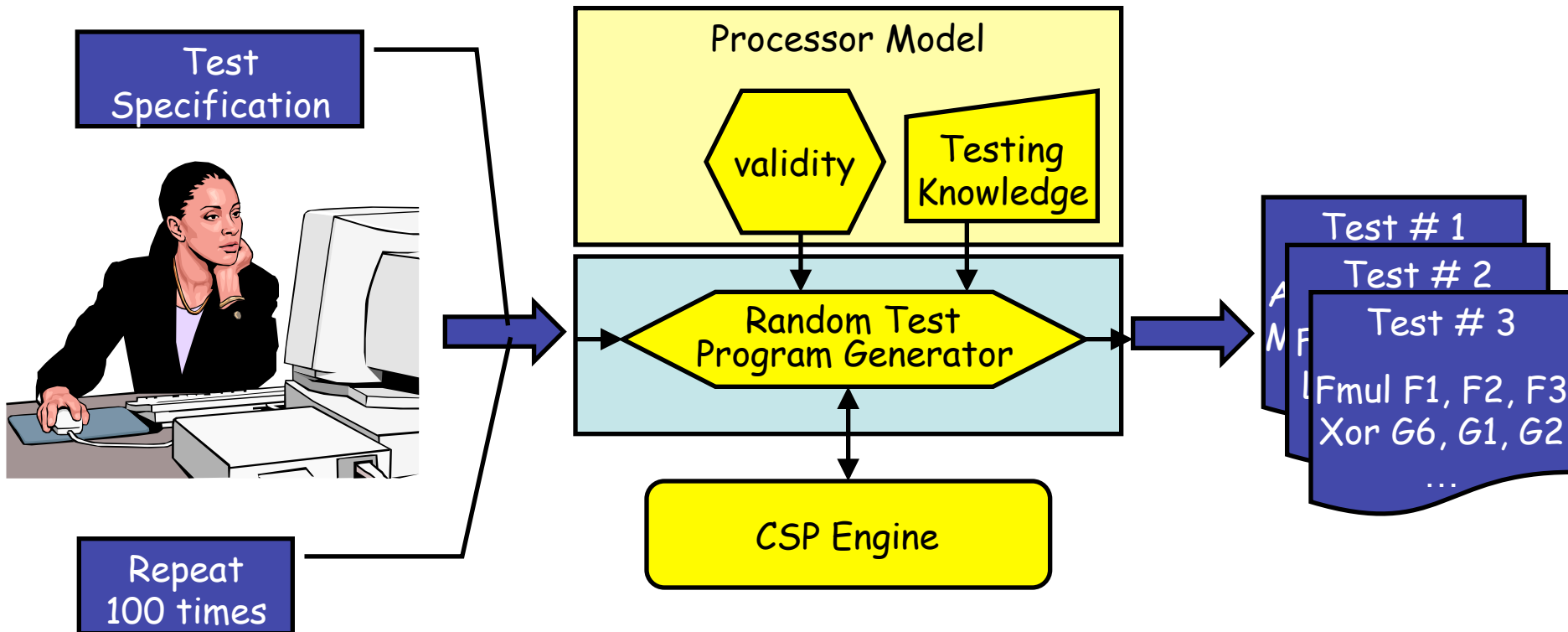
Putting It All Together: Building a Random Test Program Generator - II



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 and the CSP solution into a test program

Summary: Stimuli Generation

- Generated stimuli need to be
 - **Valid**
 - Behavior of DUV under the test is fully specified
 - NOTE: Valid is not necessarily legal
 - The verification environment can determine if the DUV behaved correctly
 - **Interesting**
 - Improve coverage
 - Reach corner cases
 - Find bugs
 - **Meet specific user requirements from the verification plan**
 - Resource reuse, interdependencies

Summary: 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 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.