

COMS30026 Design Verification

Stimuli Generation

(Part I)

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

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



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 any anomalies are propagating to an existing checker
 - Hitting a bug without exposing it is worth nothing



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 any anomalies 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 ←



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 any anomalies 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
 - Providing information about the design and helping recreate and understand problems identified



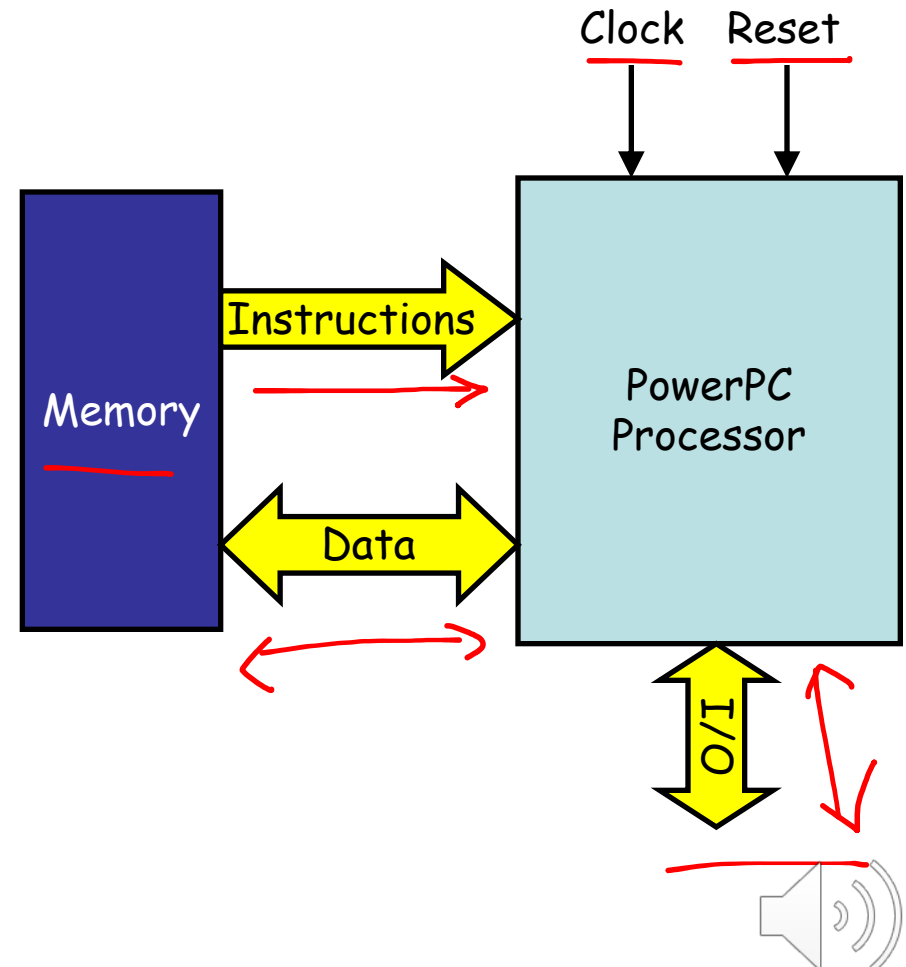
Goals of Stimuli Generation

- Achieve all the items in the test scenarios matrix of the verification plan
 - Ensure that the scenarios in the matrix are happening
 - Ensure that any anomalies 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
 - Providing information about the design and helping recreate and understand problems identified
 - 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
 - ...



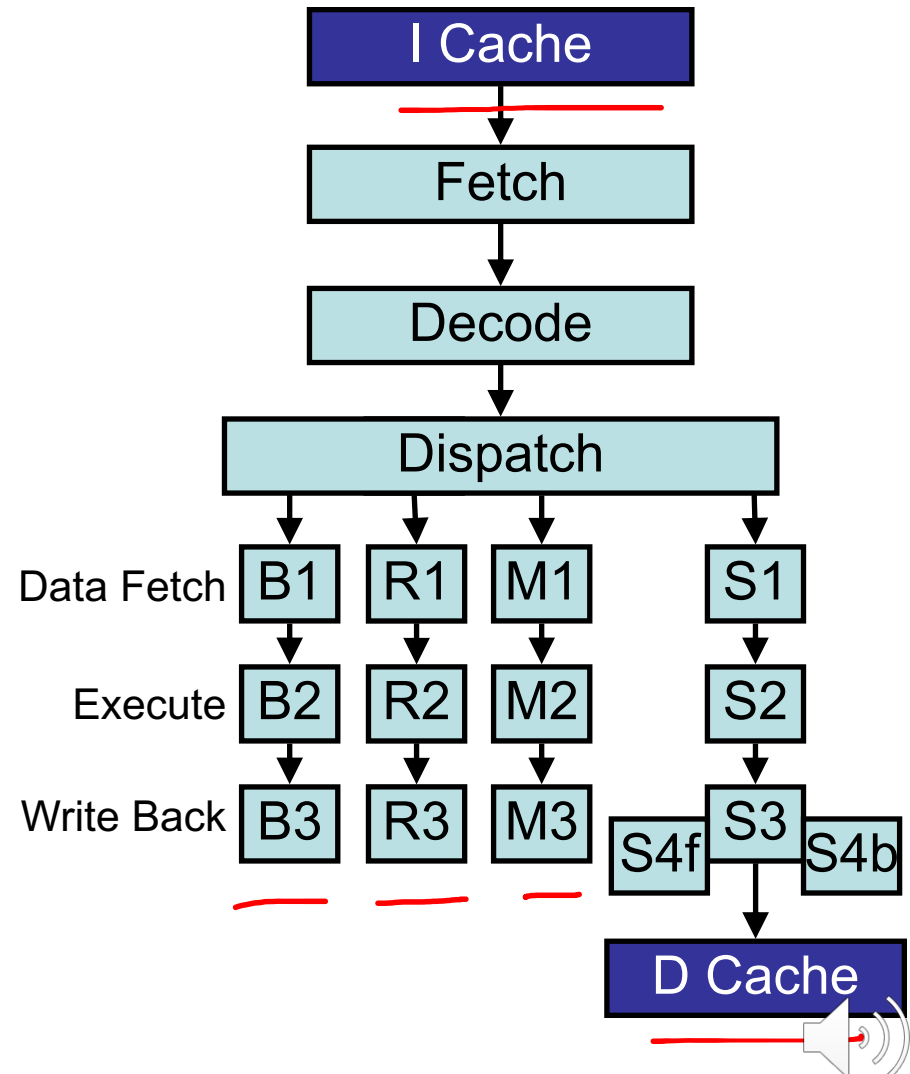
Architectural View

- 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
 - R: Simple Arithmetic
 - M: Complex Arithmetic
 - S: Load Store

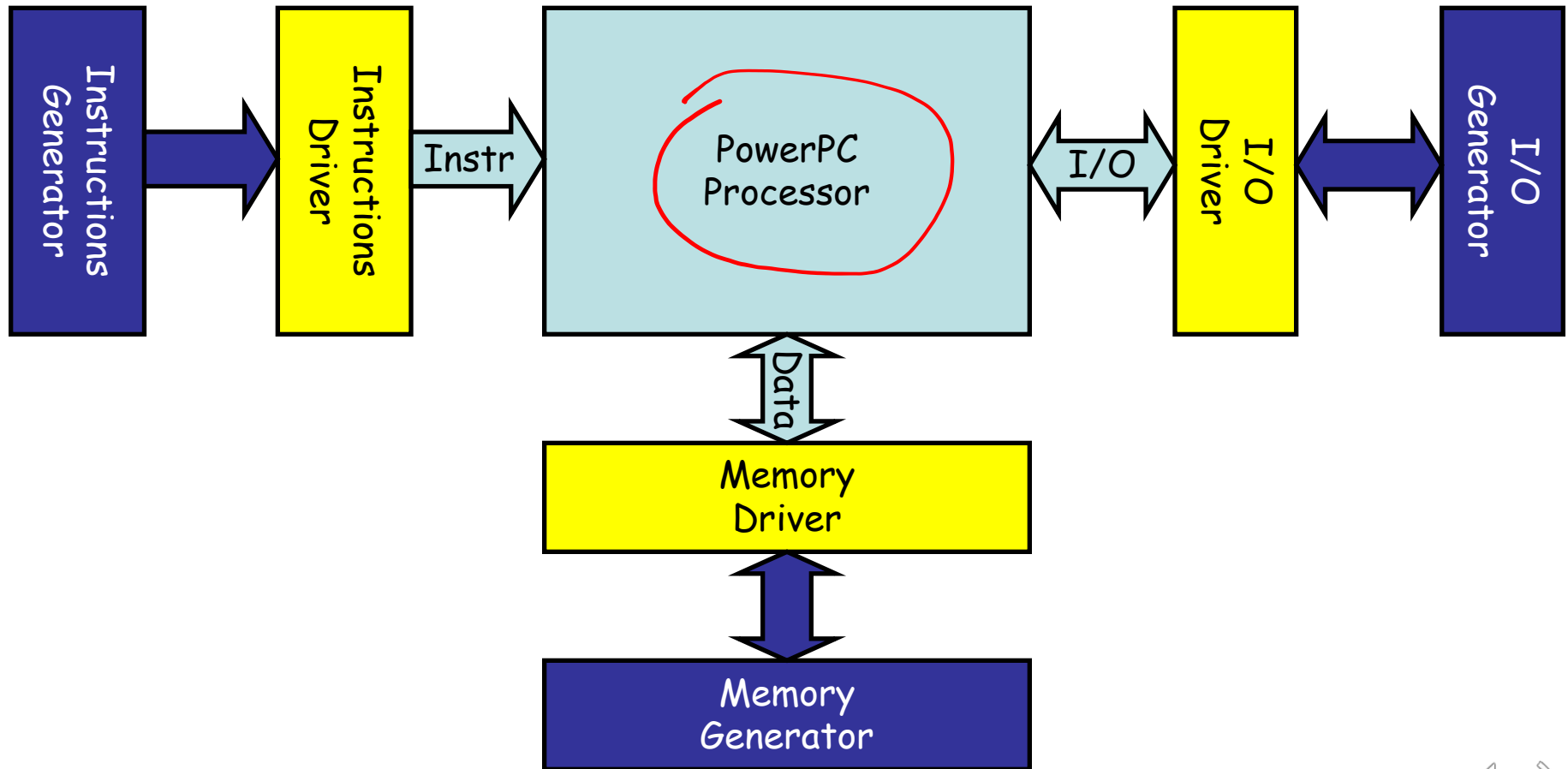


Extracts from the Verification Plan

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



Processor Verification Environment



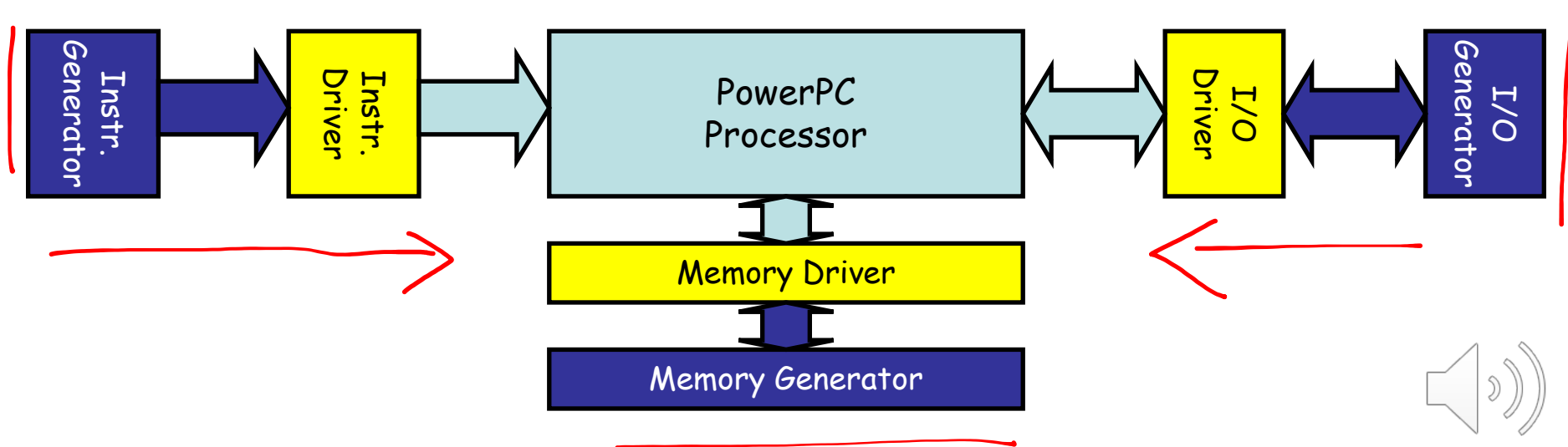
Issues in Stimuli Generation

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



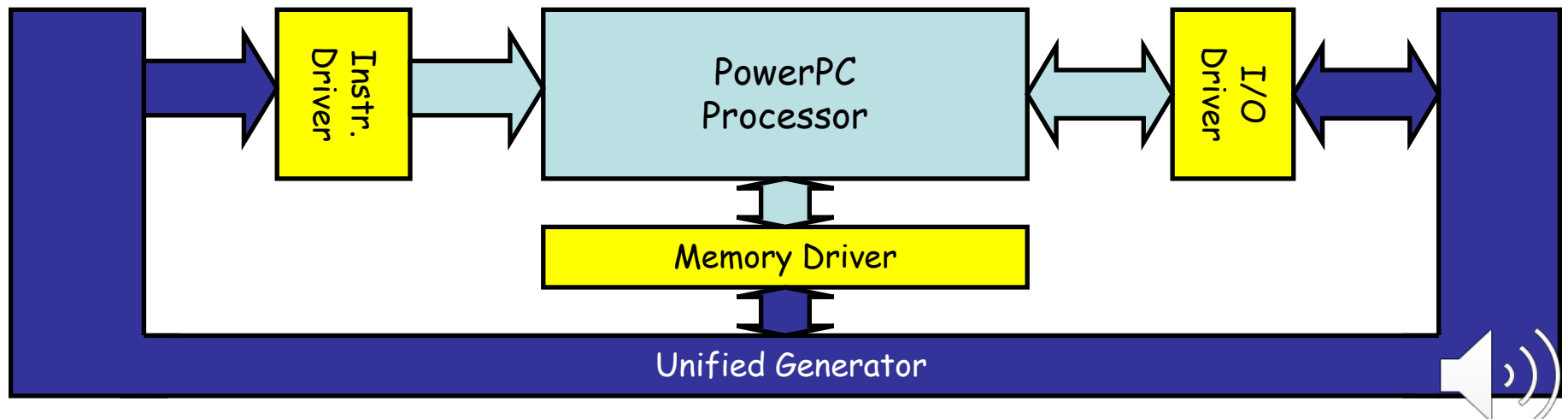
How Many Generators?

- 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



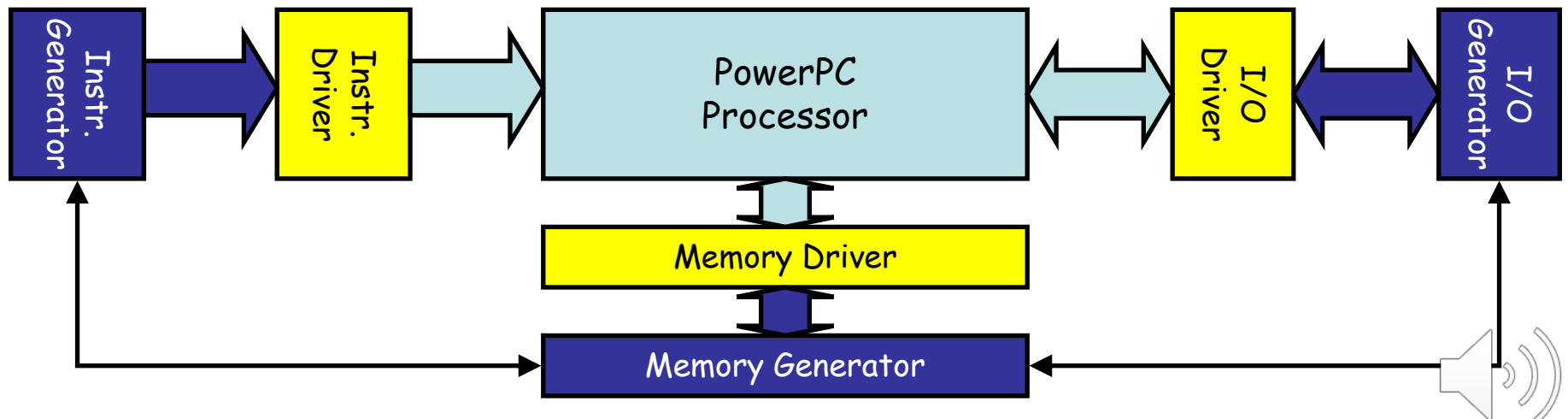
How Many Generators?

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

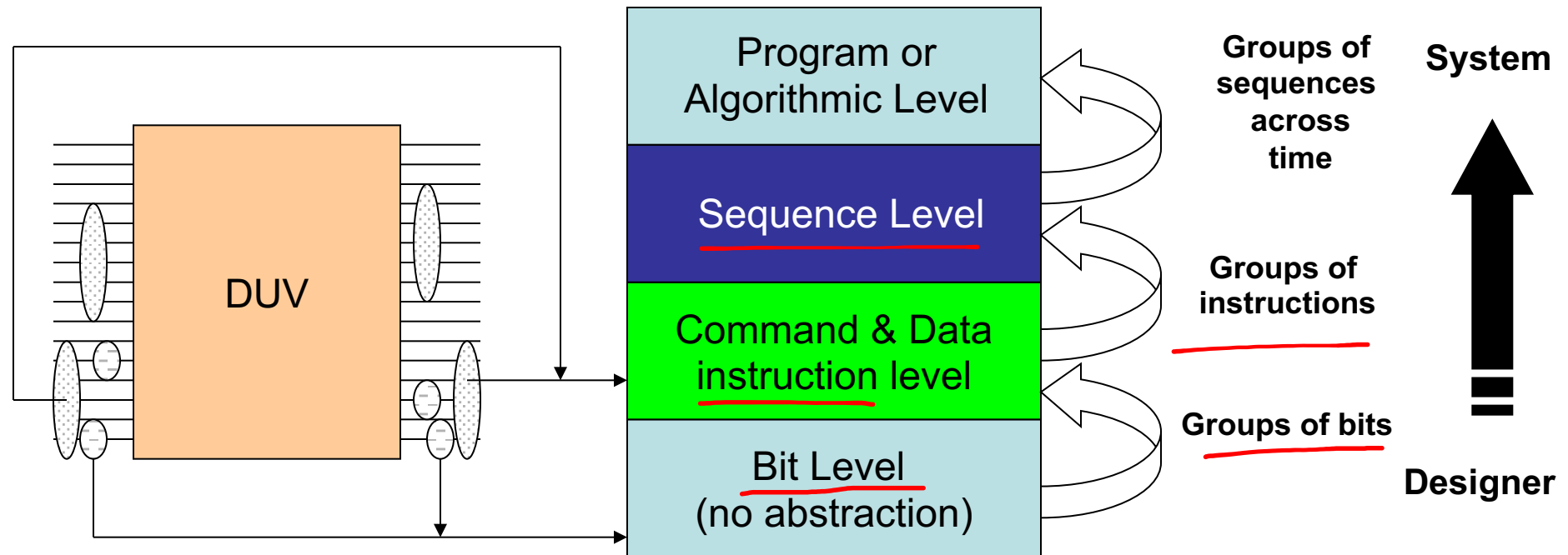


How Many Generators?

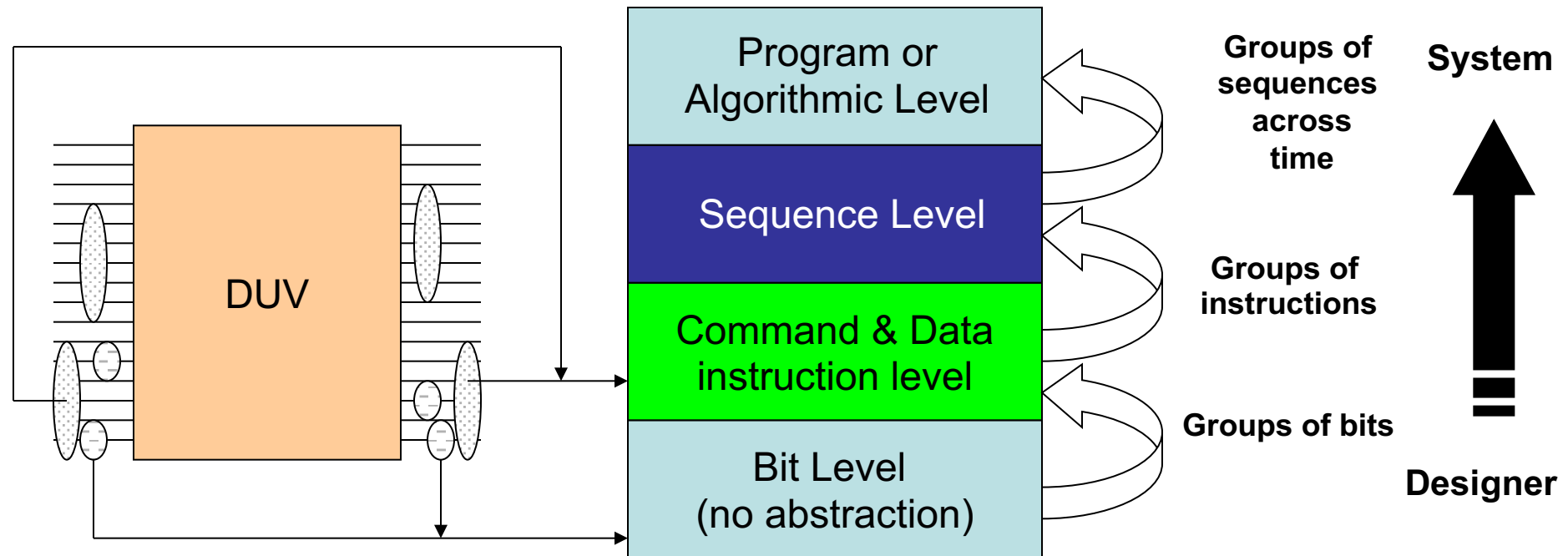
- Synchronized generators
 - Each interface has its own generator
 - The generators share information and synchronize
 - Advantages
 - Can reuse each generator separately
 - Can work together towards a common goal



Abstraction Level of Generation



Abstraction Level of Generation

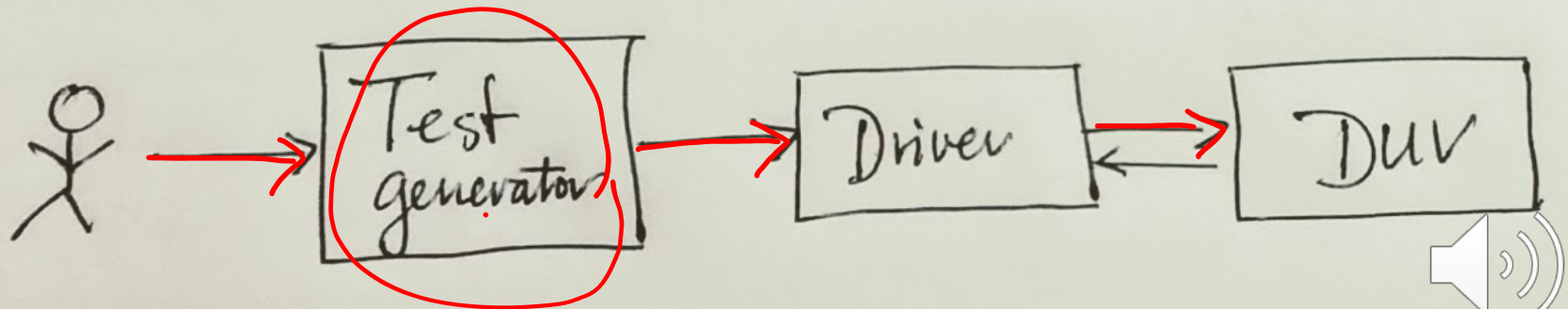


<u>0+14</u>	<u>7+7</u>	<u>10+4</u>
0000	0111	1010
+ 1110	+ 0111	+ 0100
<u>1110</u>	<u>1110</u>	<u>1110</u>

While bit-level representation allows us to see how the data exercises the carry chain, at a higher level of abstraction (looking at the integer values) we may not spot this.

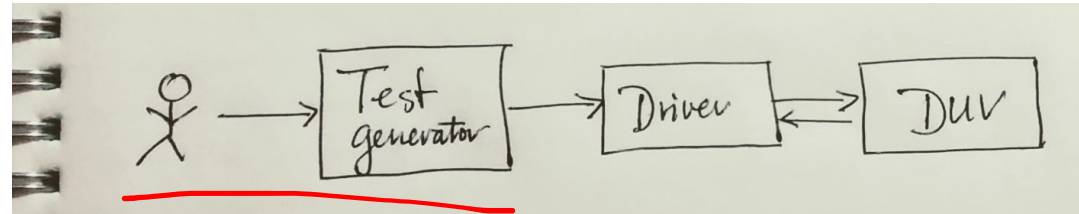
What Does Abstraction Level Mean?

- 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 (verification engineer) and the generator
 - Use a level similar to the level used in the verification plan
 - In our case (running example) – the sequence level

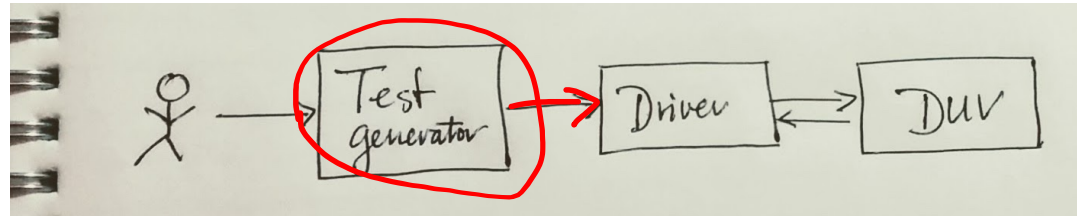


Which Abstraction Level To Choose?

- Communication between the user (verification engineer) and the generator

- Use a level similar to the level used in the verification plan
- In our case (running example) – the sequence level

- Internal representation i.e. the operation level in the test generator



- Conflicting requirements

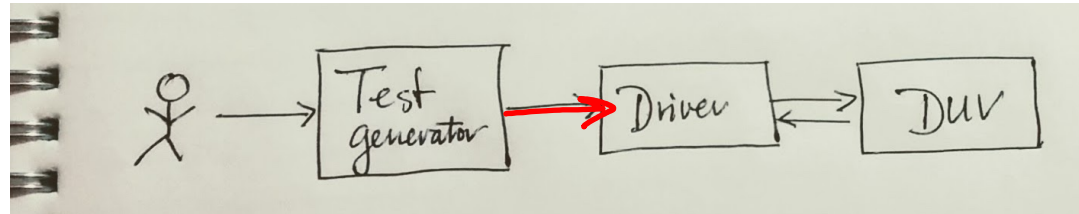
- Address user requirements (at their level) → high level of abstraction
- Need sufficient detail → low level of abstraction

- In many cases we use two or more levels for stimuli generation
 - First we build a high-level skeleton of the stimuli based on the user requirements
 - Next we add lower-level details



Which Abstraction Level To Choose?

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



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 test 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 a **separate error injection interface** between the test generator and driver
 - – At the **low level** of the error injection, i.e. directly injecting the error
 - – 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 test generation from simulation
 - Use external tools, emulation, ...
 - Can use more complex algorithms for test generation
 - For example, **generate “out of order”**, e.g. instruction sequences (processors) or action sequences (robotics)
 - Offline test generation may be compulsory **Where?**
- Why not



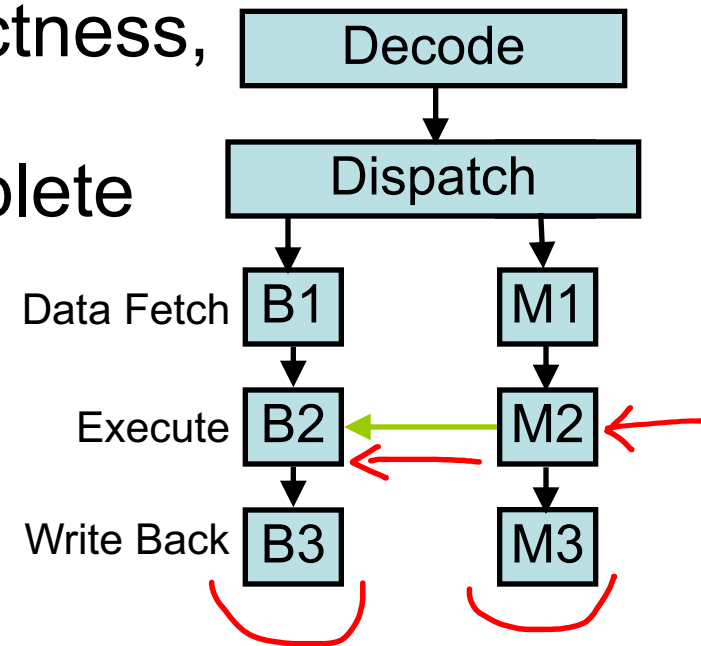
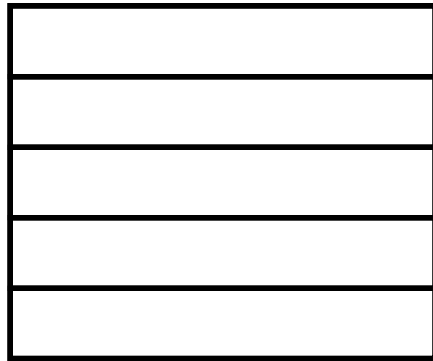
Offline Generation

- Why
 - Can separate the test generation from simulation
 - Use external tools, emulation, ...
 - Can use more complex algorithms for test generation
 - For example, **generate “out of order”**, e.g. instruction sequences (processors) or action sequences (robotics)
 - Offline test generation may be compulsory **Where?**
- Why not
 - Need to connect the test generation output to the verification environment
 - Cannot use information directly from the DUV during simulation, nor from the environment
 - Hard to react to unexpected but valid responses from the DUV



Generating Instructions Out Of Order

- Verification goal: forward data from M2 to B2
- – Branch is dispatched after arithmetic instruction
- – Both reach stage 2 together
- To preserve functional correctness, the branch must wait for the arithmetic instruction to complete

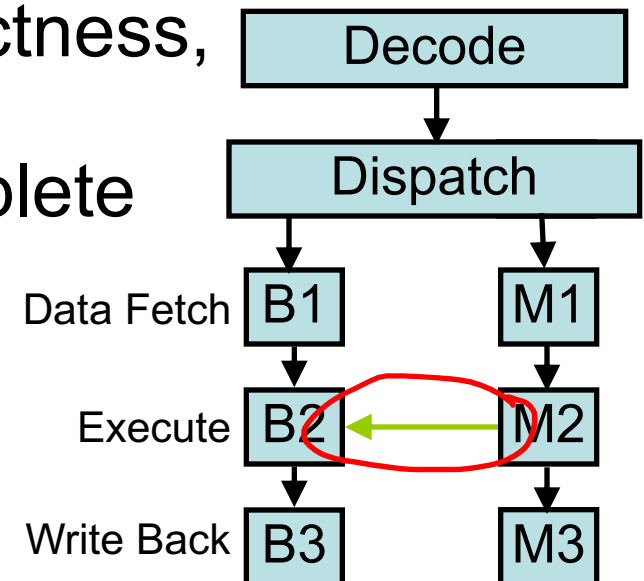
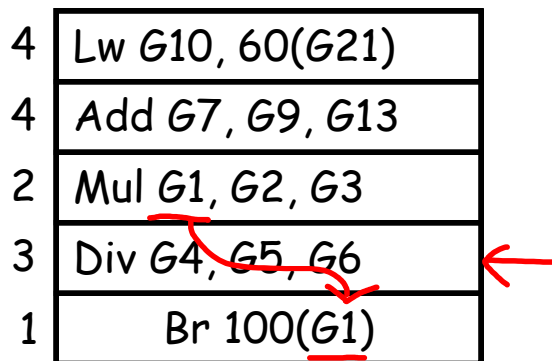


How can we generate a test, i.e. a sequence of instructions, that achieves this goal (efficiently and effectively)?



Generating Instructions Out Of Order

- Verification goal: forward data from M2 to B2
 - Branch is dispatched after arithmetic instruction
 - Both reach stage 2 together
 - To preserve functional correctness, the branch must wait for the arithmetic instruction to complete



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 to react to unexpected but valid behaviour of the DUV
 - Generally small memory footprint
- Why not



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 to react to unexpected but valid behaviour of the DUV
 - Generally small memory footprint
- Why not
 - Must generate items in order
 - Limited complexity
 - <any other reasons why not>



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 to react to unexpected but valid behaviour of the DUV
- Generally small memory footprint

■ Why not

- Must generate items in order
- Limited complexity
- – Performance: online test generation slows down simulation



Mixing online and offline Generation

- Online and offline generation can be mixed within a verification environment
- Which designs would benefit from this combination?



Mixing online and offline Generation

- Online and offline generation can be mixed within a verification environment
- Which designs would benefit from this combination?

Processor verification: instruction sequences are generated from high-level programs through compilation, i.e. offline using an external tool – the compiler, but the interrupts are generated online, when the processor is in an interesting state. 😊



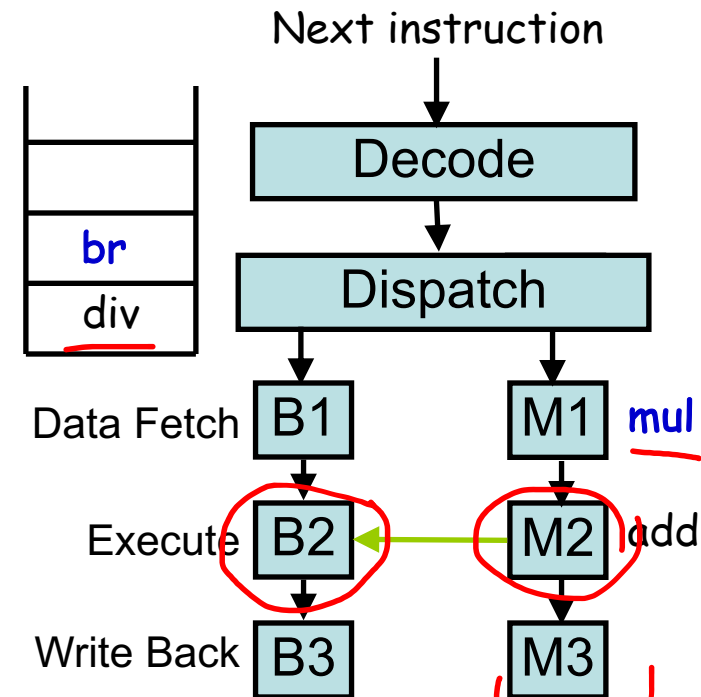
Dynamic vs. Static Generation

- In **static generation** the test generator is not aware of the state of the DUV and the environment
 - Generation decisions are based entirely on the internal state of the test generator
- Alternatively, we can take a less restrictive view on **static generation**: the test 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 test generator is fully aware of the state of the DUV and the environment and generates based on this information
 - The test generator can react to interesting states in the DUV




Dynamic Instruction Generation Example

- Verification 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 that uses 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 into 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 test generation** is based on **reaction** while **static test 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 of the DUV
 - For example, architectural registers or the state of a fifo, e.g. it being almost full.



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 end sequences

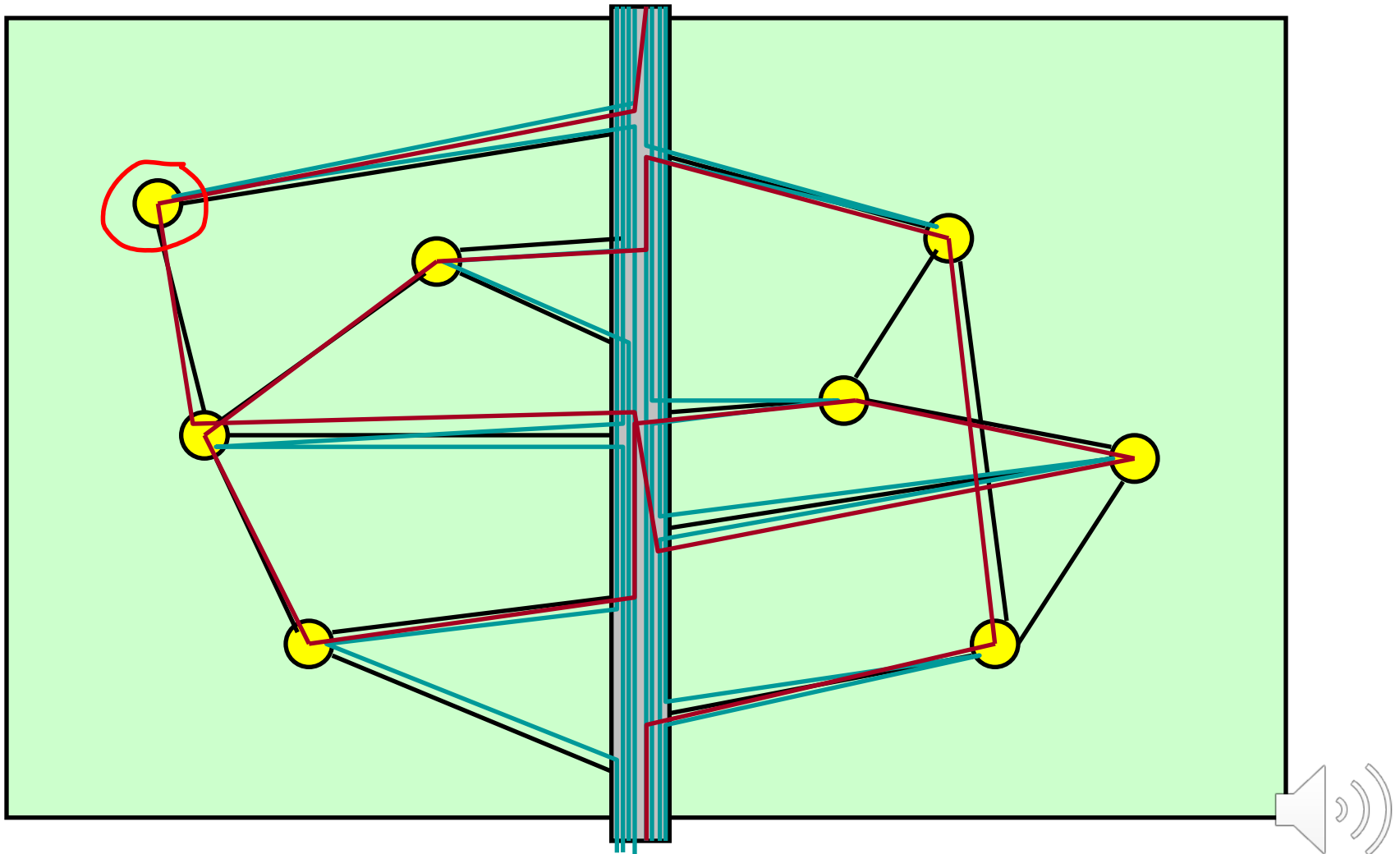


Why Short Tests?

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



Short tests vs. long tests



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 ;)



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, which is the case when using short tests
- Go along less traveled paths, which results in a greater variety in terms of exercising the logic
- Reach verification targets in different ways
 - Often leads to reaching the targets in unexpected ways



Summary of 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

