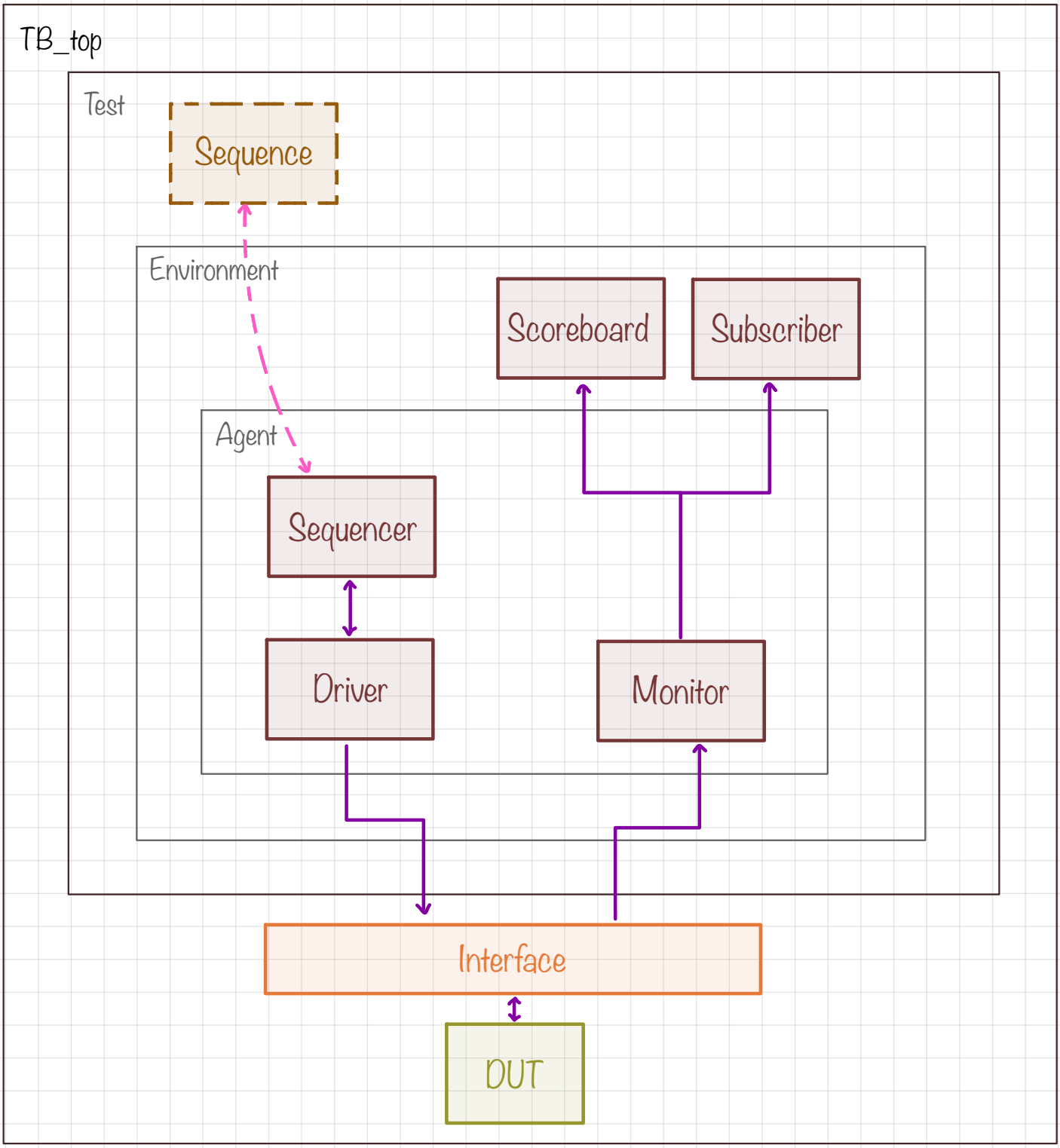
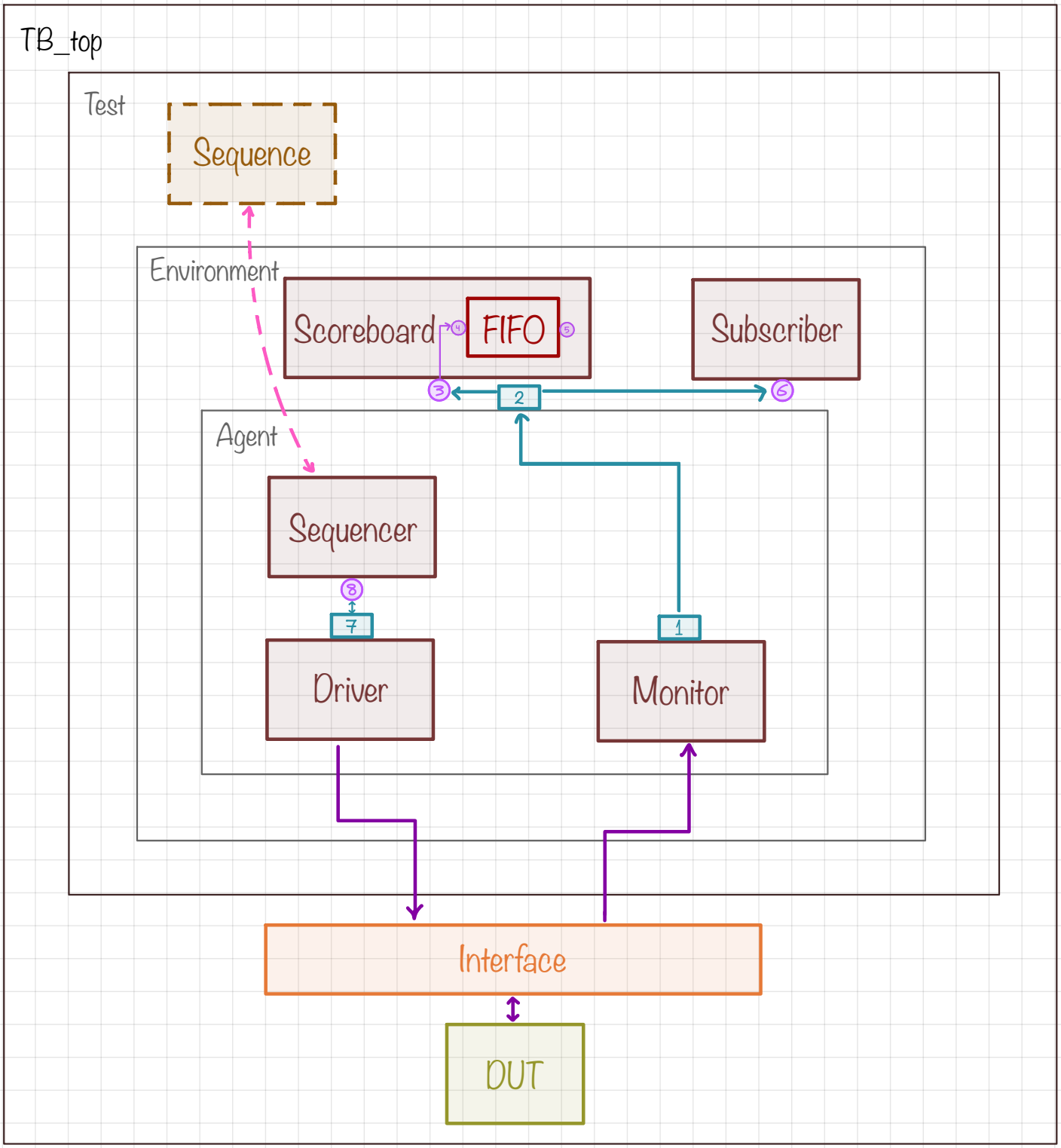
UVM Basic TestBench Environment For 16x32 Memory

Contents:

UVM Basic Architecture



UVM Basic Architecture with Connections



1- Monitor Analysis Port (for broadcasting)
2- Agent Analysis Port (for passing monitor port outwards)
3- Scoreboard Analysis Export (for passing monitor port inwards to FIFO)
4- FIFO built-in analysis_export connector (for writing data in FIFO)
5- FIFO built-in get_peek_export connector (for getting data from FIFO)
6- Subscriber built-in analysis_export connector (for getting data)
7- Driver built-in seq_item_port connector (for getting data from sequence through sequencer and sending back a response)
8- Sequencer built-in seq_item_export connector (for getting requests and responses from driver)
9- Dynamic Seq-to-Seqr connection

DUT 16x32 Single-Port Sync-Reset Memory

```
1   module memory_16x32 (
2       input logic clk,
3       input logic re,
4       input logic en,
5       input logic rst,
6       input logic [3:0] addr,
7       input logic [31:0] data_in,
8       output logic [31:0] data_out,
9       output logic valid_out
10  );
11
12  // Declare a 2D array of 32-bit words with 16 rows
13  logic [31:0] mem [15:0];
14
15  // Write data to memory when enabled and address is valid
16  always @(posedge clk) begin
17    if (!rst) begin
18          mem <= '{default:'h0}; // Reset memory to zero
19          data_out <= 'h0; // Reset output data to zero
20          valid_out <= 'h0; // Reset output validity to zero
21      end else if (en) begin // Check enable and address range
22          mem[addr] <= data_in; // Assign memory location from input data
23           valid_out <= 'h0;
24      end else if (re) begin
25        data_out <= mem[addr]; // Assign output data from memory location
26          valid_out <= 'h1; // Set output validity to one
27      end
28  end
29
30  endmodule
```

General Notes

1- Each entity will be in its own file, each file will be a Package except for interface , tb_top and DUT file.

2- in each file you'll need to add `` `include "uvm_macros.svh" `` and `import uvm_pkg::*;` inside the Package ,outside class

3- in each file you'll need to import the Packages of the other files you'll use .. like "driver_pkg" inside "agent_pkg" and so on for till you reach tb_top where you'll import "test_pkg".

4- run.do file : to automate working with Questasim

5- Source UVM files are highlighted in this Colour , lines of Code are highlighted with this Colour.

6- each step has a provided link that goes to a Commit in the github repo.

UVM Notes and Steps

1 UVM Introduction :

* It stands for **universal verification methodology** and it's a standardized way for verifing digital designs.
* It's based on system verilog and provides a set of classes than can be **extended** to create testbench components such as drivers, monitors, agents, etc.
* It also **standandized** transaction-level communication between tb components.

2 UVM Class Tree :

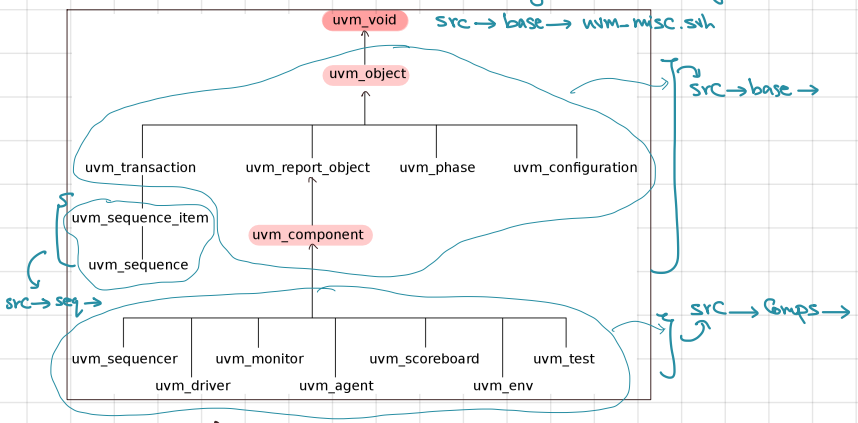
Both have a Parent/child relation

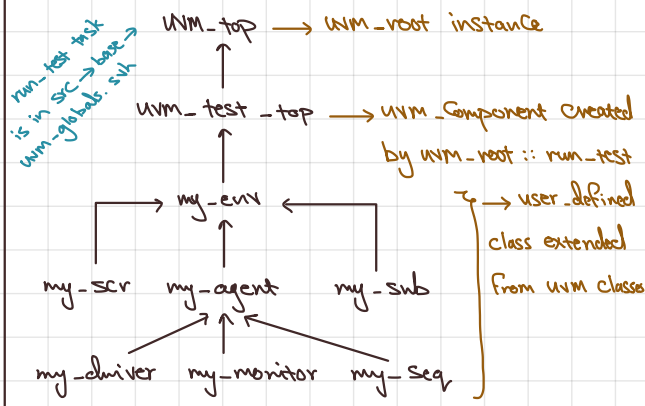
A- Inheritance Tree : "Partial" ⟷ But differ in meaning ⟷ **B- Structural / Hierarchical Tree :**

- Called : "is-a" relation
- root : uvm_void
- Defines : DNA of an object
- Purpose : Code sharing and allows polymorphism
- Established via "extends" keyword

- Called : "has-a" relation
- root : uvm_root
- Defines : where an object lives.
- Purpose : organizes testbench.
- Established in build_phase (explained in 4) and new functions.

you can find it in github under : src → base → uvm_misc.svh





* uvm_void is an **empty class**, acts as the **base class for all UVM classes**
* uvm_object extends "inherits" uvm_void, acts as a **virtual base class** for all components (like driver, monitor) and transactions. It defines a set of methods like Copy, Paste, etc.
* uvm_component is the base class for all the uvm components like env, agent, etc.
  → it supports hierarchical structure, each comp. can have child comp.
  → components can have phases that organize simulation, comp performs diff. tasks during each phase.
  → it also supports factory interface, the factory keeps track of created comp.
* The classes uvm_agent, uvm_env, etc will be used to extend from to make the user defined classes for ex. my_agent or my_env.

Note : uvm_object is said to be a child of uvm_root ; because it's extended from it.

* uvm_root is **automatically created** and available throughout simulation, its instance is **uvm_top** and users shouldn't create other instances, it's a **singleton** class, meaning only 1 instance exist in simulation.
  → it's extended from **uvm_component** and it serves as the container for all UVM components, its instance **uvm_top** is also the "implicit top-level component.
* my_agent is a child of my_env, because it's created inside it.
* uvm_test_top is of type uvm_component and it's created inside uvm_top "run_test", this task is global and it's called in tb_top and given the type of the test class as its argument. run_test("my_test"); where my_test is a user-defined class extended from uvm_test.

3 Virtual vs non-virtual classes and functions :

A- Virtual class means : no instance can be created from this class, but it can be inherited. like uvm_component, it's a virtual class.

B- function vs virtual function :

* a function is defined as virtual in a parent class to allow child class to override it with its own functionality.
* if a function "best" is virtual in a parent class, then in all child classes of this parent, the function "best" is automatically virt. like phase functions (build_phase (will be explained in 5), connect_phase, etc) inside uvm_component.
* This matters if the handle-type "like a pointer" is of parent type but it actually points to a child. ∴ when calling the function using the handle, simulator goes to the function definition in the parent and only if it's virtual does it go look at the object-type that handle is pointing at (the "child") and executes the child function instead of the parent.

Ex:

```
class parent;
  virtual function void iam();
    $display("i am in parent");
  endfunction
endclass
```

```
class child extends parent;
  function void iam();
    $display("i am in child");
  endfunction
endclass
```

```
Parent P1; // handle type is parent
child C1 = new();
P1 = C1; // parent handle points to child object
P1.iam(); // output is "i am in child" because of virtual.
```

c- **Pure Virtual functions**: used in virtual class to force all extended child classes to implement this function.

4 User-defined classes, step 1: links to git-repository commits

https://github.com/yasmine-eldesoukie/uvm_based_verification_environment_for_memory/commit/1acc50f2c4b7df123e69ed7fba5da430d25276bc
https://github.com/yasmine-eldesoukie/uvm_based_verification_environment_for_memory/commit/ba61a1a3db76339b45bl6e5127f90560df63e4d5

* **The needed classes for the environment are:**

1. Components: these are created once at runtime and stay throughout sim. Hence, they're called quasi-static.
   - ↳ test, env., agent, driver, monitor, sequencer, scoreboard and subscriber.
   - ↳ Extended from uvm_test, uvm_env, etc ⇒ these source codes can be found in github uvm 1.2 @ src/comps/..
   - ↳ Belong to a structural tree, for ex: driver is a child of agent which is a child to env. and so on.

2. objects: created and destroyed as needed during runtime. Hence, they're dynamic and don't belong to the structual tree.
   - ↳ sequence & Packet or transaction class
   - ↳ sequence is extended from uvm_sequence and Packet is extended from uvm_sequence_item, found in src/seq/..

* **When extending from a class, note if:**

1. it's parameterized ⇒ `class uvm_subscriber #(type T=int) extends uvm_component;`
   - Parameterization is either of type like the example ↑ or of value `#(p1=5)`.
   - `#(type T=int)`: parameterizes the data type "int" and gives it a name "T", the logic behind this is simple; it's to make this class generic and can work for any data type indep. of the one it was written in.

   ⇒ why "int"? because it can be overridden with any data type when extending from this class.

   ⇒ How will it be used? when we define our scoreboard class say "my_scoreboard" we'll override this parameter with the data type of our choice which is another user-defined class "my_seq_item" for ex ↳ extended from uvm_sequence_item

   ∴ if the parent class is param, you'll have to override these params as needed

   `my_sb extends uvm_subscriber #(my_seq_item);` → this class file must be included in the scoreboard file.

   Note: Don't forget that a class is a data type for ex: `class x; endclass` | `x c1; // c1 is a class instance of type "x".`

2. it has pure virtual functions ⇒ these must be implemented or code won't run.
   for ex: "write" function in uvm_subscriber.

* **Default Constructor "new":**

1. for objects, it just creates an instance of the class with the provided "name" in the parameters
   `function new(string name="sequence");`
2. for components, it creates an instance with "name" and the hierarchial parent handle, as explained before .. env is the hierar. parent of agent and so on. Therefore, 2 steps are needed for this definition:

   ⇒ 1 declare that the instance of this component has a parent and assign it with null for now. ↳ its actual value will be in step 2.

   `function new(string name="agent", uvm_component parent=null);`

   ⇒ 2 in the component's hierar. parent class, for this example it's "my_env" class, create an instance of the agent and assign a value of its parent which is my_env or use the keyword "this" it'll point to the class you're in.

   ↳ The exact syntax will be explained in 6

   inside build_phase function of env ⇒ `agent = my_agent::type_id::create("agent",this);`

* **Calling the parent's default constructor** using super.new inside the child's constructor. (↳ inheritance parent)
  - super points to the parent class, this child extended from for ex. inside your "my_env" that's extended from uvm_env, "super.new" calls constructor of uvm_env because it won't be automatically called.

5 Phasing :

* Phasing Concept is important in systemVerilog testbenches because it introduces classes that can be created in the middle of the simulation ⇒ this needs organization.
* Phasing does that through a set of defined methods for "uvm_Component" and they can be grouped into 3 Catagories :
  1. Build time phases : functions executed before start of simulation. → at 0 time.
     - most used ⇒ 1.1 : "build_phase" function : instances of all uvm_Component and extended class are created along with their sub_Comp for ex: my_env → uvm_env → uvm_Component and all that's inside my_env like my_agent and so on.
     - 2.2 : "Connect_phase" function : after all instances have been created, their Connections "TLM" are established. → explained in [11] → Transaction Level Modeling
  2. Run time phases : The actual simulation (running time) happens in these phases.
     - 2.2 : "run_phase" task : is the most common but not the only one, runs in parallel with other run-time phases. and it Contains the behaviour of the Component
  3. Clean_up phases : after end of simulation, like report_phase or final_phase ↳ for last minute operations before existing the sim.

∴ All uvm_Component and extended class of it are aware of the phase Concept. So, all Components go through a pre_defined set of phases and can't proceed to the next phase until all Components finish the current phase ⇒ UVM phases act as a synchronising mechanism, so no Comp. is called before it's created for ex.

6 Factory :

* in systemVerilog environment, we use the "new" function to build an instance of a class. it's hard_coded, for example, if you build a driver inside an agent using "new" method, you'd have to specify the type of this driver inside the agent. This's fine, but what if you have 2 drivers, 1 for normal functionality and another for a special one... then you'd have to change the Code where this driver is used each time you use a different one.
  What the factory in UVM does is that it manges the process of creating instances of classes, adding new "overriding" features to the process... in the prev. ex, all you have to do is tell the factory to override the driver class by the special driver class in the "test" without going back to the agent class... imagine doing this on a large scall, where you can change the elements of your tb environment from the test Code only without the headache of changing every Code.

* This's done by a set of classes defined in UVM, all you have to do is 3 steps : → a macro is a substitution of text, for example, a Code, so instead of repeating lines of Code, it's replaced by a macro, it could be user_defined or uvm_defined
  1. Register all objects and Components in the factory using these uvm_defined_macros : `uvm_Component_utils(class_handle); // for Components , and `uvm_object_utils(class_handle); // for objects (class name, my_test, my_env, etc.)
     ⇒ This steps means uvm_factory (which has a single instance in the whole sim.) adds this item to its table, to be known by factory.
     ⇒ what does it do ?
     a. Create an object inside your class to represent it, it's a parameterized instance of uvm_Component_registery class, Param. with your class type ⇒ uvm_Component_registery #(class_handle, "name"). (or _object_ for objects)
        This registery is your class's door to uvm_factory, it's like the middle man uvm_factory orders to call "new" for your item.
     b. Define a new data type using typedef for that registery named type_id, so it's a standard name for all Comp. registery without having to know the specific class name. typedef uvm_Component_registery #(class_handle, "name") type_id ;
  2. keep the definition of the "new" Constructor as it is ⇒ objects take string "name" and Components take "name" and a handle to "parent" why? because the factory will call the "new" function for you, with the right type of parent or overridden parent. and if you change its arguments, the factory won't be able to adapt.
  3. Create the item with the defined "create" method, ex: driver my_driver inside the agent
     driv = my_driver::type_id::create("my_driv", this); , instead of driv = my_driver.new("my_driv", this);
     the 2nd Code will be called either way, it's just that calling "new" by yourself means you're asking for a certain type of driver, "my_driver". but through the create method, it checks with the factory 1st if there're override request for "my_driver". if there is, for ex: an order to override any "my_driver" with "special_driver", "create" method will handle this and build your driver with the wanted type. ↑ issued in the test Code

Note that "create" is a registry function that calls the factory's create function called "create_component_by_type" (or _object_)
⟹ what about "this" keyword ? it's a special pointer to the current instance of the class, it's used for path accuracy. in the "create" line of code, it points to the wanted parent for your item, that was assigned "null" in "new".
⟹ what if instead of using "this" you use a fixed parent handle for ex "my_agent" ?
this works fine until you decide to override "my_agent" with another agent, "special_agent" for ex.
now "create" orders to builds an instance of my driver inside the agent, but which agent ?
using "this", it will point to the correct agent ⇝ advanced_agent
using fixed_parent method will point to my_agent always ⟹ driver will attached to wrong parent.

Note : when creating any component, "this" keyword will point to its parent, except my_test. it's parent will remain as "null" and this tells UVM that this instance will be the top. Explained in 7

https://github.com/yasmine-eldesoukie/uvm_based_verification_environment_for_memory/commit/e40c55790401488daaaa5fe602910ae28bfdc6a1
https://github.com/yasmine-eldesoukie/uvm_based_verification_environment_for_memory/commit/4f572e058bba8ec21956a49d1d31fa6fb2e4ba08

7 Building the environment Components, Step 2 : links to Commits

Remember :
an instance of uvm_root called uvm_top is automatically instantiated and is the root for the hierarchical tree.

* Building is done Top_down ⟹ Test ⟶ Env. ⟶ Agent ⟶ ...
* let's divide this process to 3 steps :

1- Building from Test down to driver and monitor :
a) Register the classes in the factory. `uvm_component_utils(component_name); (or _object_ for objects.)
b) Implement the build_phase function. function void build_phase(uvm_phase phase);
c) Call build_phase of the parent. super.build_phase(phase);
d) Create instances of sub-components and objects, for ex: env and sequence inside test, using "create" method.

2- Building and running "my_test" :
• Call the global task "run_test" in the tb_top module inside the initial block ⟹ run_test("my_test"); (this's the name used when registering in factory or the class name.)
• This task can be found in src ⟶ base ⟶ "uvm_globals.svh" and it calls "uvm_top.run_test" that :
a) Calls factory to build an instance of the "my_test" class but name it "uvm_test_top" and assign it to parent = null so that "uvm_top" becomes its parent ⟹ this's defined by UVM.
b) Starts the phasing mechanism, build ⟶ connect ⟶ run ⟶ clean-up. This's like pushing the last piece of domino
c) Calls $finish; to end simulation. That's why any code after "run_test( );" doesn't execute.

3- Overriding the building blocks - if needed. This step is done in the "build_phase" of "my_test" before the "create" lines. and it can be by name or by type, it's to tell the factory that whenever something asks for "my_driver" for ex. give them an "advanced_driver" ⟹ ex: set_type_override_by_type(my_driver::get_type(), advanced_driver::get_type());

8 UVM Resources and Databases :

Note :
A) Associative Array : int x[]; is a normal array which has an index starting from 0 / int x[string]; is still an array but its index is a key of type "string". ∴ instead of x[0]=32; it's x[bus_width]=30;
B) Static Class : a class that contains static methods and variables, it's not meant for instantiation or inheritance. (↳ no new();) It acts as a toolbox, you don't need to "own" the toolbox you can just grab the tool and use it.
⇝ it's contents are accessed by the class name and the scope operator "::"

* The testbench environment has many components that will eventually need data from one another, we can declare these data as global variables but it's not safe; anyone can change it ! and it's a nightmare to debug.
* How does UVM resolve this issue ? it created an organized mechanism for storing, searching and retrieving these data through a set of classes :

1. uvm_resource #(data_type) : uvm stores each item created in the sim as a "resource" using "uvm_resource" class. and each resource has a name, a type, a value and context "scope" for visibility or where and who can access it.
2. uvm_resource_base : this's the parent or base class of "uvm_resource", it's not tied to a certain data type and will be used for the queue that will store these resources.
3. uvm_queue #(data_type) : this's a queue of "data_type", this "data_type" will be param. by "uvm_resource_base" so that it's not limited to "int" items for ex, or even uvm_resource #(int) ... it can store any uvm resource because we used the generic uvm_resource_base.

4. uvm_pool : this's basically an associative array with added features. ∴ it can carry an associative array of these uvm_queues of uvm_resources.

5. uvm_resource_pool : this class has 2 groups of "uvm_pool" instances :

a) Name uvm_pool : the key of the associative array is a string name, and the entries of this array are queues of diff. resources sharing the same name.

b) Type uvm_pool : same as name uvm_pool but the key is a data type and the stored resources are of the same type.

Each uvm_resource will be stored inside both of these uvm pools, this will make things more organized and faster to get when called for.

6. uvm_resource_db : (↳ database) the uvm_resource_pool is like an actual library storing resources and uvm_resource_db is like the librarian, it's an API an interface to uvm_resource_pool. it's a static class. This's what we use in the tb, the prev. classes are the behind-the-scenes workers.

⟶ when used uvm_resource_db #(int)::set("global", "bus_width", 32); it creates a uvm_resource #(int) to be stored in uvm_resource_pool : ↳ by name under "bus_width" in name uvm_pool ↳ by type under "int" in type uvm_pool → stored in both. "global" is the scope, that needs to be matched why you use ::get(" ", " ",);

⟶ It's used for items that don't belong to a physical path but rather created, run and destroyed in sim. like sequences.

7. uvm_config_db : this's a restricted database, when used it has the feature to include the physical path as the scope, and the getter has to be in that scope to get! unlike uvm_resource_db scope, it has no idea that the tb is a tree of components and it has no idea where that resource is, "global" to it is just a string to match at the getter.

also a static class

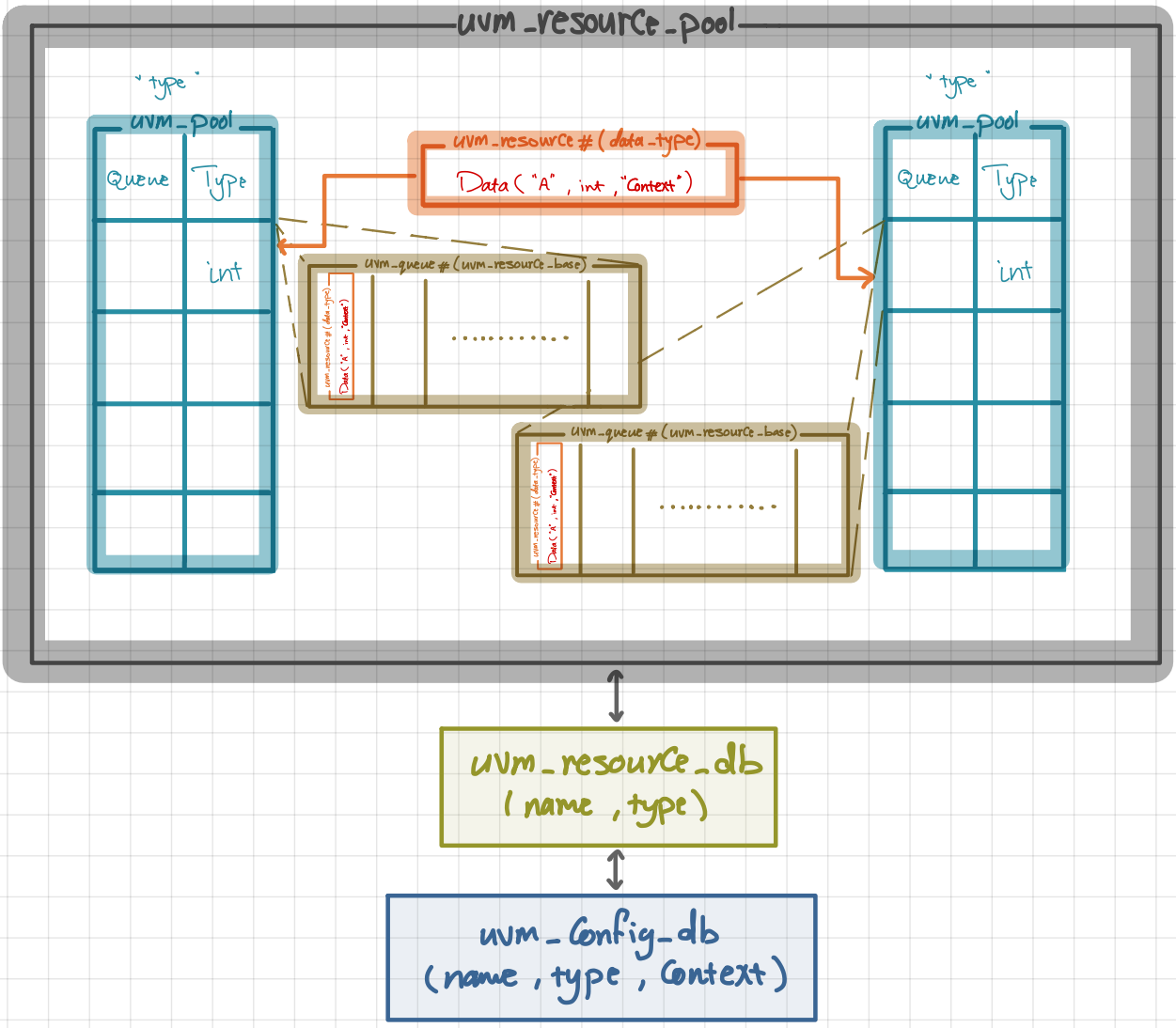
⟶ ∴ uvm_config_db is used in tb for sharing components and restricting the other components access to it. it's mostly used for passing the virtual interface handle from tb_top to driver and monitor. Explained in 9

⟶ Syntax : uvm_config_db #(data_type)::set(Context, Instance_path, Field_name, Value); to store

* Context : start of the path. "null" for modules and "this" for classes.
* Instance_path : string representing "name_of_component_inside_path".
* Feild_name : key "Name" to be stored under in uvm_resource_pool.
* Value : actual Data to store.

uvm_config_db #(data_type)::get(Context, Instance_path, Field_name, Value); to get

by default it's get by name from the uvm_resource_pool to search by name in its "name uvm pool" and match the exact type in that queue, rather than search by type because then, uvm jumps straight to the "type pool" and gets the most recent resource.



https://github.com/yasmine-eldesoukie/uvm_based_verification_environment_for_memory/commit/15e65be9d9396487776161e3e4cbf7f082f5f08b

9 Passing Virtual Interface Handle, Step 3 : link to Commit

Note: "include "interface.sv" must be outside tb_top module to be global

* Interface instance is created at tb_top and needs to be passed down to driver and monitor. This will be done using uvm_config_db::set/get.
* It Could be made global to the whole env. but this's useless since not all Components need the interface and it will just take too long to find when needed since the scope is wide ⇒ Performance ↓. What's better is to pass it down layer by layer test → env → agent → driver/monitor
* Steps :

1. in tb_top module, create an instance of the interface ⇒ intf my_intf ;
2. in tb_top, pass it down to test as virtual : uvm_config_db#(virtual intf)::set("null","uvm_test_top","my_vif", my_intf);
   "null" because this's a module, "uvm_test_top" is my_test instance where you want to deliver this handle.
3. in test class, also create a virt. handle to store the passed handle from tb_top to be able to pass it down again to env.
   ⇒ virtual intf test_vif ; // a diff name just to since the difference btwn field_name that can't change and "value" that's like the specific container for the passed handle in each class.

get function return type is not void, it has to be written like that to avoid errors.

→ in build_phase

```
if (!(uvm_config_db#(virtual intf)::get( this , "" , "my_vif", test_vif ))) // this and " " mean to get here because in tb_top we specified this exact location ⇒ "uvm_test_top"
  `uvm_fatal( "my_test" , "Failed to get my_intf") ; // fatal because not getting the interface handle is severe.
uvm_config_db#(virtual intf)::set ( this , "env" , "my_vif" , test_vif ) ; // set it to "my_env" which is the inst. of my_env in my_test.
```

(empty)

4. Repeat in my_env, my_agent and my_driver/my_monitor classes.
   Just remember to set twice in agent, 1 for driver and another for monitor
   also note to only get in driver and monitor, no more set!

Note: if interface is parameterized, for ex:
interface intf #(parameter width = 8)( );
then it has to be declared as (virtual intf #())
in the uvm_config_db line

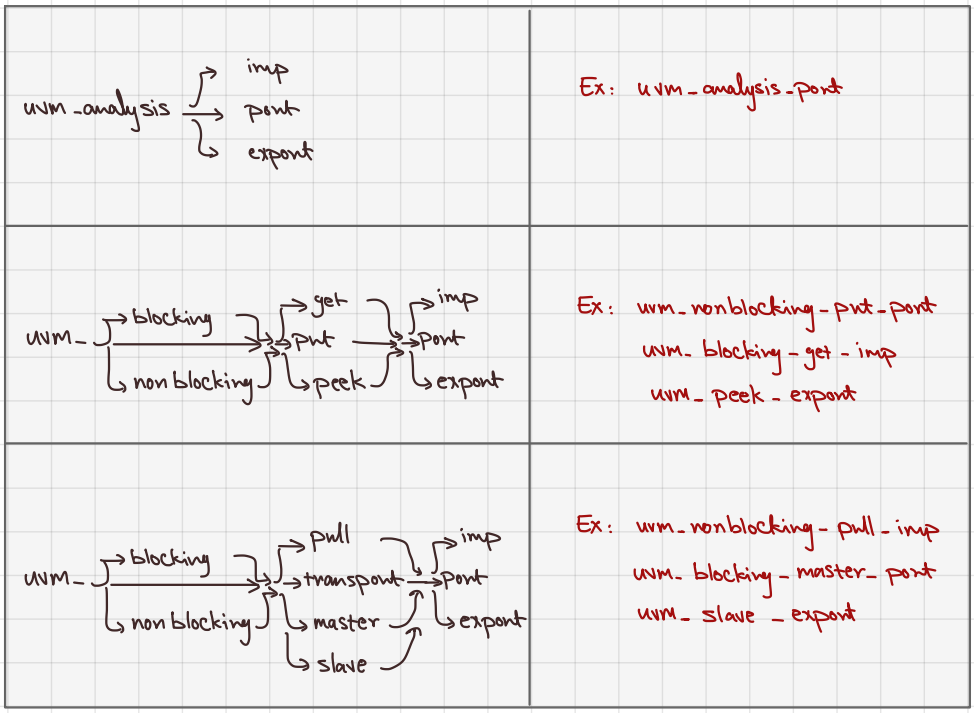
10 Transaction Level Modeling (TLM) :

* TLM has multiple advantages over using mailboxes in terms of flexability, direction and added features that make TLM smarter.
* The mechanism is like a faucet and a drain, each is defined in its prespective Component without knowing or caring about the other Comp. unlike mailboxes that have to be defined in both Components (both share handles to the same mailbox), if one is to be swapped, the other Comp. code has to change.
* UVM Provides many versions of TLMs and they can be categorized based on :

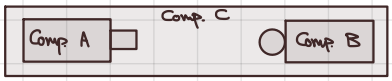
1. Role : who starts the call and where the logic lives.
   A) Port : initiates transaction by calling a method on the other end.
   B) Imp : "implementation", terminates transaction, this's where that method is implemented
   C) Export : Middleman, passes a connection through a Component boundry. Explained in 11
2. Interface : action and direction, what is being done and in which direction does data flow.
   direction ⇒ A) unidirectional : (Put, get, peek) family → Methods
   Actions ⇒ → Put : initiator "puts" data to target. → get : initiator asks to "get" data from target.
   → Peek : initiator "looks" or "peeks" on data at target but it's not removed from target.
   B) bidirectional : (Master, Pull) family, used when a request-responce handshake is needed
   → Pull : This one is actually built-in for driver and sequencer, driver pulls data using "get_next_item" method and then pushes a completion status "item_done".
   → other examples like master/slave and transport.
   C) analysis : (Broadcast) family 1-to-many
   → write : this's the only method, initator "writes" in each target, and this's the method used between monitor and both scoreboard and subscriber or even better ... if there are many of them not just one. and this is another advantage of TLM over mailboxes, monitor having an analysis_port just writes and it doesn't care if it's 1 Comp. or a 100.
3. Execution :
   A) Blocking : implemented methods are task, so they take time, it forces initiator to wait.
   related methods are like "put" and "get".
   B) Non-blocking : implemented methods are functions, so they execute in no time.
   related methods always start with "try_" or "can_" ⇒ "try_put" : returns 1 if target is ready to accept transaction (just checking needs no wait)
   C) Combined : for ports that can do both you can use the plain "uvm_put_port" instead of either "uvm_blocking_put_port" or "uvm_nonblocking_put_port"

All methods can be found in uvm
src → tlm1 → uvm_ports.svh
→ uvm_tlm_ifs.svh → for all available methods

Note: analysis_ports are by default non-blocking, no reciever target should stall the process. That's why "write" is a function not a task.



For the available methods of each connection, check uvm src → tlm1 → "uvm_tlm_ifs.svh"



* General Method :

1- Create a `uvm_*_port` at initiator, for ex: `uvm_blocking_put_port #(packet) CompA_p;` at Comp. A. and call its "new()" constructor in Comp. A "build_phase" : `CompA_p = new("CompA_p", this);`

2- Call the defined method for the chosen port at initiator in its "run_phase". for this ex, it's "put" : `CompA_p.put(pkt);` (remember in 5, "run_phase" task contains behaviour of Component.)

3- Create a `uvm_*_imp` at target, for ex: `uvm_blocking_put_imp #(packet) CompB_imp;` at Comp. B. and call its "new()" constructor.

4- Implement the defined method "put" at target. This method defines what the target does with the recieved data.

5- Connect them in their hierarchical parent "Connect_phase", for this ex: it's Comp. C.

`CompA.CompA_p.Connect(CompB.CompB_p);`

11 TLM Connections in TB, Step 4 :

https://github.com/yasmine-eldesoukie/uvm_based_verification_environment_for_memory/commit/d7ceba589e675ebf5a6dcc4b7eef61c5f7794e31

link to Commit

Note: building is top-down and Connecting is bottom-up

* Before we start, we have to add Connect_phase function and run_phase task to all Comp. just like build_phase in 7

1- Monitor to Scoreboard/subscriber :

* one-to-many → analysis ports will be used. Monitor is the initiator ⇒ "uvm_analysis_port" → naming is for historic reasons.
* subscribers extended from uvm_subscriber have built-in "uvm_analysis_imp" connection called "analysis_export" That's why it was mandatory to implement the "write" function in "my_subscriber" class.
* for the scoreboard, it's one of two scenarios : and implement the write function.
  - → one monitor ⇒ we will create a "uvm_analysis_imp" in scoreboard and connect it to the monitor's port in "env".
  - → multiple monitors working with diff. speeds, so to avoid data loss ... each monitor will store its data in a FIFO inside scoreboard. uvm provides FIFOs src → tlm1 → "uvm_tlm_fifos.svh". This's the scenario we'll use to learn how to use FIFOs.
* Step-by-step following the general method in 10 :

1- Monitor (The initiator) :

a) Create a `uvm_analysis_port #(my_seq_item) mon_ap;` and call "new()" in build phase `mon_ap = new("mon_ap", this);`

b) Call "write" function in its run_phase `mon_ap.write(seq_item);` notice how monitor just "writes" not caring which Comp recieves! ∴ if sub. was to be swapped by another, monitor code won't change, that wouldn't be the case if we're using mailboxes.

2- Subscriber (Target) :

a) its "imp" is already built-in and named "analysis_export".

b) implement "write" function and what you want sub. to do with the received "seq_item".

3- Scoreboard (Target) :

a) Create a `uvm_tlm_analysis_fifo #(my_seq_item) scr_fifo;` / `scr_fifo = new("scr_fifo", this);`

This FIFO also has a "uvm_analysis_imp" called "analysis_export"

b) create a `uvm_analysis_export #(my_seq_item) scr_ax;` / `scr_ax = new("scr_ax", this);`

remember in [10] when "export" was described as a middleman to pass connections through component boundries.

This export connection just connects the "mon" with the "scr_fifo", why? to understand the reasoning behind this we have to go a couple of steps forward.

Don't touch a code that isn't broken!

when connecting the monitor with the FIFO of scoreboard inside env connect_phase, if we use:

`agnt.mon.mon_ap.connect(scr.scr_fifo.analysis_export);` This means env. has to be aware that

1) scoreboard has a FIFO that has an analysis_export. if any of that changes, env. breaks or you have to change its code.
2) agent has a monitor that has mon_ap ⇒ same thing and we will fix it in [ ] with the same apreach.

∴ to avoid that it'll be `agnt.agnt_ap.connect(scr.scr_ax);` ⇒ env. doesn't know about its children contents.

c) connect scr_ax with scr_fifo's analysis_export in the connect_phase: `scr_ax.connect(scr_fifo.analysis_export);`

d) call "get" or "peek" from FIFO in "run_phase", it's established that the FIFO has an anlysis_imp called analysis_export and it's the front-door of the FIFO for writing data. Well, there's another "imp" acting as the "back-door" for collecting data from FIFO, it's a "uvm_get_peek_imp" named `get_peek_export` providing both "peek" methods for retieving without removing from FIFO and "get" methods for retriving and removing from FIFO. including both blocking/non blocking versions like get, try_get, can_get, etc.

∴ in run_phase call `scr_fifo.get_peek_export.get(seq_item);` for ex.

4- Agent (Parent of Monitor):

a) create a door for passing mon_ap connection to env. Just like scr_ax, the only difference is that the connection in scoreboard is going top-down from a parent "scr" to a child "FIFO" ⇒ export is used. But in the agent's case it's bottom-up from child "monitor" to parent "agent" ⇒ port is used.

`uvm_analysis_port #(my_seq_item) agnt_ap;` / `agnt_ap = new("agnt_ap", this);` // in build_phase

b) connect it to mon_ap in connect_phase: `mon.mon_ap.connect(agnt_ap);`

Connect Rules:

Port to Imp/Export ⇒ Port.connect(Imp/Export) Port to Port ⇒ child_port.connect(parent_port) ex: "agnt_ap"

Export to Imp ⇒ Parent_export.connect(child_imp) ex: "scr_ax"

5- Environment (Parent of init. & targets):

a) connect agnt_ap to both sub. and scr in connect_phase:

`agnt.agnt_ap.connect(scr.scr_ax);` / `agnt.agnt_ap.connect(sub.analysis_export);`

2- Driver-Sequencer Connection:

* in SystemVerilog Verification environment, we had a stimulus generator, a driver and a shared mailbox. in UVM two things changed: the use of TLM connections over a mailbox and dividing the stim_gen into a sequencer and sequence ... while stim_gen works fine in simple designs, it can get very messy and hard to implement in more complex designs, that's why it's divided into a "script" ⇝ sequence and a "director" ⇝ seqr.
* in the sequence, you write the scenarios for ex: reset_seq, write_seq, read_seq, etc and in seqr, you can manage how they run, their priorities and so on.
* before jumbing to the TLM connection of driver_seqr, let's talk about two important tasks in the sequence:
  1- start_item(seq_item): This's a request to the seqr, it says "I have a "seq_item", tell me when the driver is ready to recieve it". This request blocks seq code until seqr is not busy and driver is ready.
  2- finish_item(seq_item): once the sequence is granted the permission, it randomizes or assigns data in general to its packet's data "seq_item" upon the req. of driver of course. once it's done with generating data, it tells seqr that it's done and that "seq_item" can be sent to driver. and then this line also blocks seq code until driver announces that it finished with this data.
* How to Establish this connection:

1- Driver: ↝ src → tlm1 → "uvm_sqr_connections.svh"

a) it has a special built-in port of type uvm_seq_item_pull_port named "seq_item_port" and is ready to use. it has both blocking and non-blocking methods to accommodate for your driver's protocol. src → tlm1 → "uvm_sqr_ifs.svh"

b) call the desired methods, the standard protocol is a simple blocking "get_next_item" and nonblocking "item_done" as a responce.

* get_next_item: tells seqr that driver is ready for new data → this's the signal seq is waiting for to generate data.
* item_done: tells seqr that it finished with this data → this's the signal seq is waiting for at finish_item.

⇒ There've other methods for diff. driver functionality.

Note: This's a bidirectional conn. because driver "gets" data from seq. through seqr. and then sends some kind of responce dep. on the protocol.

2- Sequencer :

* it also has a built-in port of type uvm_seq_item_pull_imp named "seq_item_export". and it has all the methods implemented you don't need to do anything here.

3- Agent (Parent):

* in its Connect_phase, you call `driv.seq_item_port.Connect( seqr.seq_item_export);`

https://github.com/yasmine-eldesoukie/uvm_based_verification_environment_for_memory/commit/bce6e6bdc1cc88db718427392ffb7019faea308b

https://github.com/yasmine-eldesoukie/uvm_based_verification_environment_for_memory/commit/317085782aa2725dba33440cb160ccfb8ae141a

12 Sequence to Sequencer Connection, Step 5 : links to Commits ↑

* Unlike the TLM hand Connections, this Conn. is dynamic because the seq is an object created in runtime.
* The Connection happens in "my_test" class "run_phase" as so: note that "seq" is an instance of my_sequence and "seqr" is of my_sequencer ↑

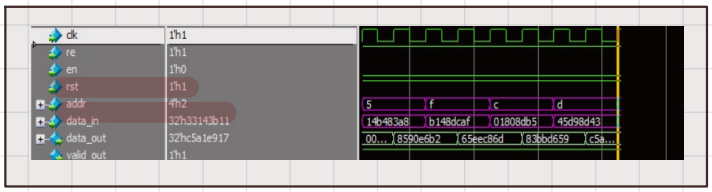
`seq.start(env.agnt.seqr);`

* That line of code on its own finishes right away, to make the test wait until the sequence is actually finished, we use the Concept of objections, this's like a global Counter, to ↑ the Counter use "raise_objection", to ↓ it, use "drop_objection" so that code looks like this : ⟹

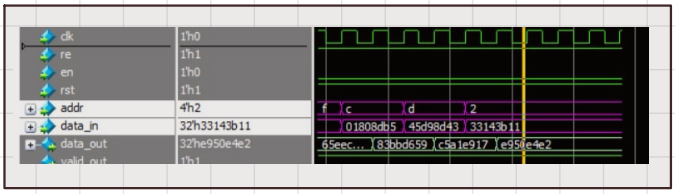
```
task run_phase(uvm_phase phase);
    phase.get_objection().set_drain_time(this, 50ns);
    phase.raise_objection(this);
    seq.start(env.agnt.seqr);
    phase.drop_objection(this);
endtask
```

* Drain_time: to insure "test" doesn't finish right after the last Packet is sent, and that it has time to be processed, set drain_time.

Before drain time



After drain time



up to this point, the tb environment is generic, next it will be specialized for the DUT in Page 4

13 Sequence Item Class, step 6 :

* This is the class carrieng the "Packet", it's extended from "uvm_sequence_item". it Carries the dut signals and it's where we add their Constraints.
* It's also registered in the factory (as an object) and it has a Constructor.

* 16x32 Memory Signals: clk, Control signals (rst, en, re) → write_enable, read_enable, data busses([31:0] data_in & data_out) and a valid_out flag.
* Signals that are to be randomized are declared as rand and for Cyclic_Randomization ⟹ randc, to hit all values before repeating

https://github.com/yasmine-eldesoukie/uvm_based_verification_environment_for_memory/commit/daf7932bba7fa3625cc2dc36ceebdea25e44509e

14 Sequence Class(es), Step 7 : link to step 6,7 Commit ↑

* for simplicity, you Can write all your scenarios in one class, but the better approach is to write each unique sequence in its own class, this allows reusability and also gives you the freedom to run them in parallel from your "test" and also add priority.
* Assume having multiple sequences, this's how you run them in "test" sequentially Vs in parallel with priority:

```
phase.raise_objection(this);
    write_seq.start(env.agnt.seqr);
    reset_seq.start(env.agnt.seqr);
phase.drop_objection(this);
```

```
phase.raise_objection(this);
    fork
        write_seq.start(env.agnt.seqr, null, 100);
        reset_seq.start(env.agnt.seqr, null, 200);
    join
phase.drop_objection(this);
```

null refers to parent seq.

seqr has a built-in arbitration algorithm and by default it's (First Come, First served) and if two Come at the same time, it looks at the higher priority.

The 2nd code runs as so:

⟹ both sequences are alive at the same time but the sequencer sees that reset has a higher priority and calls it first once driver asks for next item, then it goes to write_seq.

* UVM_sequence class that extends your seq classes, has a set of virtual empty methods that the seqr knows of and executes them in order when you use "start( )" method in test. (seqr: extended from uvm_sequencer)
→ These methods are: pre-start( ) function / Pre-body( ), body( ) & post-body( ) tasks and post-start function.
→ usually only "body" is needed and you need to keep its name that way or it will never run!
* Where can you "create" an instance of the Packet class my_seq_item? in the "body" task, inside the loop to create a new instance with every randomization to avoid overwritting in the same memory location that other Components might still be using.

Overview of the sequence Class for 16x32 Memory module:

- Inside the "body" task, create a reset packet where reset is ON, if you want to add delay to keep this pattern for 50ns for ex, add this delay #50ns after finish_item(reset_packet); and a new packet won't be sent to driver until this delay finishes.
- Then make 2 loops, each of n iterations, for ex: 16 as the memory addresses. (between start_item( ); & finish_item( );)
- for the 1st loop, adjust the input signal for a "write" scenario. And for the 2nd, adjust them for a "read" scenario. The addr signal could be incremental from 0 to 15 or randomized, i will use the latter.
- Randomization: to insure all addresses values are hit before repeating we use randc for the addr signal in my_seq_item class, and it works just fine if we create one instance of my_seq_item and overwrite it with each iteration. But, if we create a new one with each iteration, then each one will have the full range and randc will be = just rand ... The solution is 1. declare a randc logic [3:0] local_addr; signal in "my_sequence" class. 2. randomize it in the loop using this.randomize( ); 3. Assign local_addr to write_pkt.addr for ex write_pkt.addr = local_addr;
- Remember using "assert" in SV environment to detect if randomization failed, in UVM, we use "uvm_fatal" macro, it's a reporting macro.

```
if (! this.randomize)
  `uvm_fatal ("Randomization Failed", "local_addr rand. failed");
```
("ID" / "message")

- You can add different patterns for input signal dominance and so on.

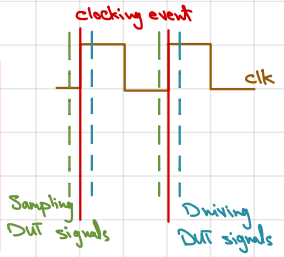
15 Interface, step 8:

* Later on when we get to the driver and monitor we will have to take care of driving and sampling timing and race conditions, the robust way to avoid race conditions is to use a clocking block for each driver and monitor.
* In a clocking block, there's an 1) output skew: defines when to drive the output after the clocking event (like posedge clk).
2) input skew: defines when to sample the inputs before the clocking event, to insure data is stable.

Clocking Blocks of Driver and Monitor in the Interface:

```
clocking driv_cb @(posedge clk);
    default output #1ns ,
    output rst, en, we, data_in, addr ;
endclocking
```

```
clocking mon_cb @(posedge clk);
    default input #1ns ,
    input rst, en, we, data_in, addr, data_out, valid_out ;
endclocking
```



https://github.com/yasmine-eldesoukie/uvm_based_verification_environment_for_memory/commit/a80532cc31505e470a7bbb36f474a5602cccld9b

16 Driver and Monitor Classes, step 9: (link to commit of step 8 and 9 ↑)

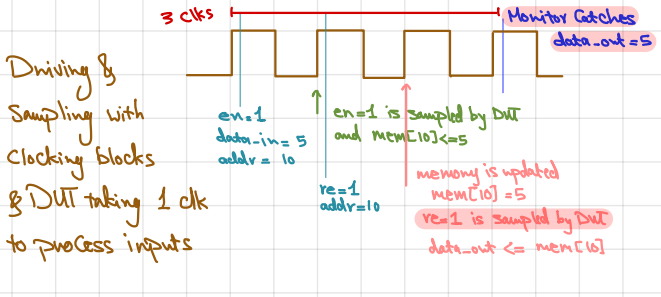
* Driver issues the DUT signals at the "driv_cb" event like so @(driv_vif.driv_cb) using non_blocking assignments "<=" in "run_phase"
* Monitor samples at its clocking block event as well @(mon_vif.mon_cb) but with blocking assignments "=" in "run_phase".

https://github.com/yasmine-eldesoukie/uvm_based_verification_environment_for_memory/commit/e4eb0fe84767943dd605c3b3901lcd50a5bf60af

17 Scoreboard Class, step 10: (link to Commit ↑)

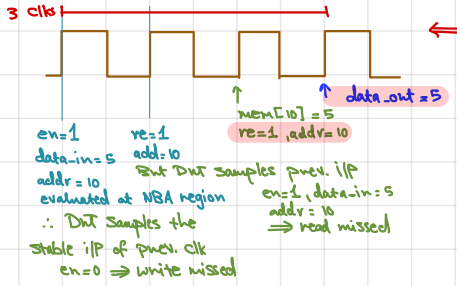
* The scoreboard has to mimic the DUT behaviour with a ref model, since this's a memory module ... we will create a 2D array to store the data_in with the write command and retrieve that data to compare in case of a read command. [31:0] shadow_mem [int]; (↑ associative array)
* The DUT takes 1 clk cycle to process its commands:
1- Reset is sync., so at clk 1 for ex, DUT samples "rst=0" but memory, data_out & valid_out = 0 at clk 2.

2- Read Command is sampled at edge X but the expected data_out is only available at edge X+1 whether you use a clocking block or not



Driving at clk edge creates a race condition and depends on simulator scheduling

using clocking blocks avoids that and is safer



∴ When Scoreboard gets a packet with :

1- (rst = 0) :

- raises a flag "check_reset" for ex, for next Call. `check_reset = 1'b1 ;`
- Deletes "shadow_mem" to mimic DUT `shadow_mem.delete();`
- In the next Call there must be a peice of code controlled by the flag and checks output signals

```
if (check_reset) begin
    if ( seq_item.data_out !== 'bx || seq_item.valid_out !== 'bx)
        `uvm_error ("Data_Mismatch", $sformatf (" data_out = %0h ", seq_item.data_out));
        check_reset = 1'b0 ;
    end
```

(checks 0, 1 & "x" not just 0 & 1 ; ID ; like $display)

`uvm_error is a reporting macro, it also counts # errors.

2- (re = 1) :

- Raises a "check_read_data" flag for next call.
- Stores this Packet addr in a local variable "read_addr" `read_addr = seq_item.addr;`
- In the next Call, Compares the data_out of new Packet with that stored in "shadow_mem [read_addr]" and turn down the flag.

3- (en = 1) :

- Just store data_in of that Packet in "shadow_mem [seq_item.addr]" seq_item is the Packet

Complete Scoreboard Code:

```
// Process Prev. Command
    if (check_reset)
        ...
    else if (check_read_data)
        ...
// Process Current Command
    if (! seq_item.rst)
        ...
    else if (seq_item.en)
        ...
    else if (seq_item.re)
        ...
```

https://github.com/yasmine-eldesoukie/uvm_based_verification_environment_for_memory/commit/d0742e0613b31914ec9541152578a955a258cec

18 Subscriber Class, Step II : link to Commit ↑

* This's where we write the Functional Coverage code to check if the input signals hit the desired cases or test, or to find holes in them to create dedicated tests to cover these scenarios.
* Before the Constructor Code, add the Covergroups with their Coverpoints, cross_Coverage, bins and so on.
* Then inside the Constructor, use `cg = new();` where cg is the name of the Covergroup.
* Inside "write" function : parameterize it with "my_seq_item" class

  sample the data of the recieved signals from monitor packet "seq_item". But make it under the Condition of rst off.

```
virtual function void write ( my_seq_item t);
    seq_item = t;   // seq_item is an instance of my_seq_item
    if (t.rst)
        cg.sample();
endfunction
```

(created in build_phase)

* Covergroup cg ; (keyword)

```
name_of_Coverpoint : Coverpoint seq_item.input_signal_1 ;
name_of_Coverpoint : Coverpoint seq_item.input_signal_2 {
    illegal_bins name_of_bins = { value1, value2 };
    bins name_of_bins [ ] = { [value1 : value2] };
    ignore_bins name = { value }; }
```

```
cross_Cover_1: cross name_of_Cp1, name_of_Cp2 ;
cross_Cover_2: cross name_of_Cp2, name_of_Cp3 {
    bins name = cross_Cover_2 with (Condition); }
endgroup
```

* Each Covergroup has Coverpoints, each Coverpoint can have bins (for 1 or multiple values), illegal_bins (must not happen) and ignore_bins (to exclude from Coverage). Simulator creates "bins" for each Covergroup, each bin can have 1 or more values dep. on the total number of possible values, naming them like `bins all_zeros = {'h0};` is for readability.
* The difference between `cp_a Coverpoint seq_item.a;` & `cp_a Coverpoint seq_item.a { bins all_zeros = {'h0}; }` is that the 1st only hits 100% if all possible value of a are hit, The latter hits 100% if all its "bins" are hit, in this case it's "h0".
* Cross Coverage is between two Coverpoints, it checks all their combinations if `cross_Cover : cross cp_a, cp_b;` and checks certain combinations of them that follow a certain condition `cross_Cover : cross cp_a, cp_a { bins cross_1 = cross_Cover with (Condition); }`

Multiple Examples can be found in the provided link above.

Examples of Coverage for Memory DUT :

1. Check data_out hit special cases : all 1's, all 0's, toggle, etc.
2. Check different combinations of en & re : 2'b10, 2'b01, 2'b11.
3. Check addr hit all its possible values, all values transitioned to every other value, or that it went from all 1's to all 0's and vise versa.
4. Check that each addr has been written in and read from with cross coverage between 2 & 3.

Note: in this step, you'll probably need to make adjustment to constraints in "seq_item" class and "sequence" class and maybe add directed tests.