DVCon 2013
Design & Verification Conference & Exhibition
February 25-28, 2013
DoubleTree, San Jose
accellera
SYSTEMS INITIATIVE

New and active ways to bind to your design

by
Kaiming Ho

Fraunhofer IIS

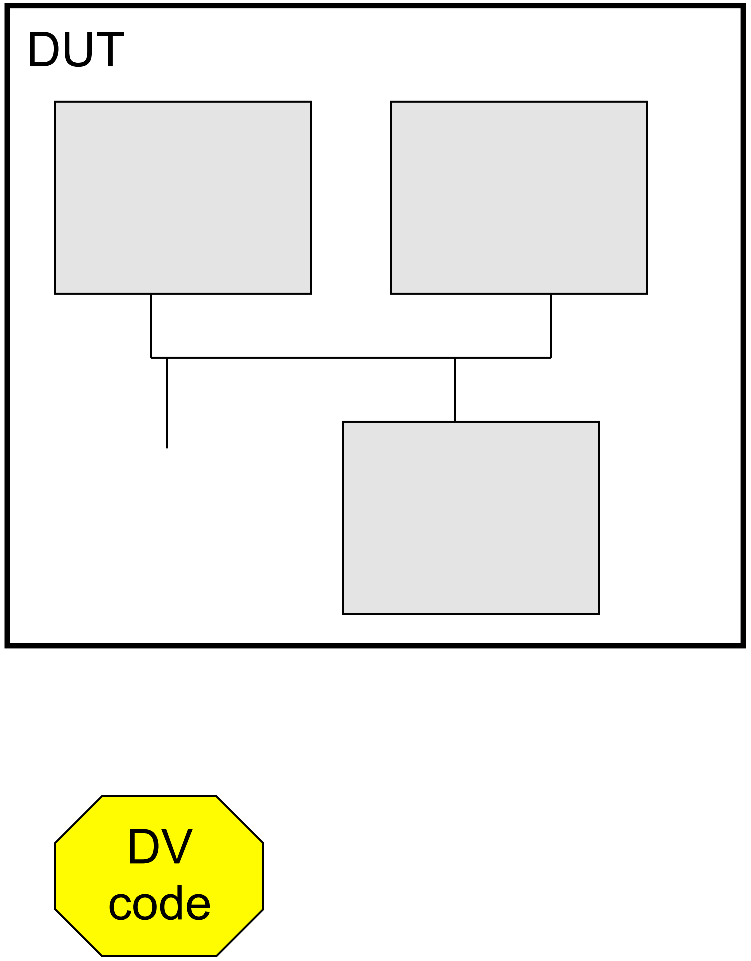
Fraunhofer
IIS

Overview:

- Introduction and Motivation
- Key techniques and language constructs from SystemVerilog
- Example use case

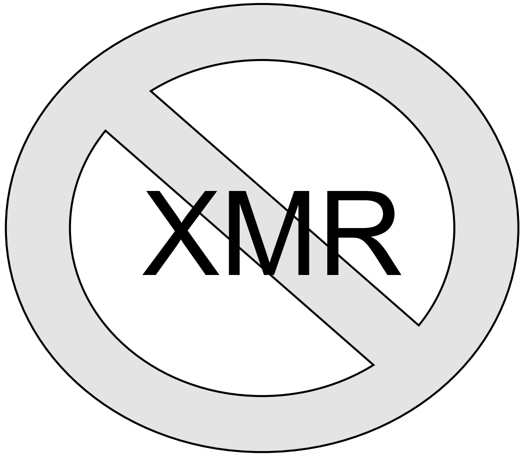
Introduction

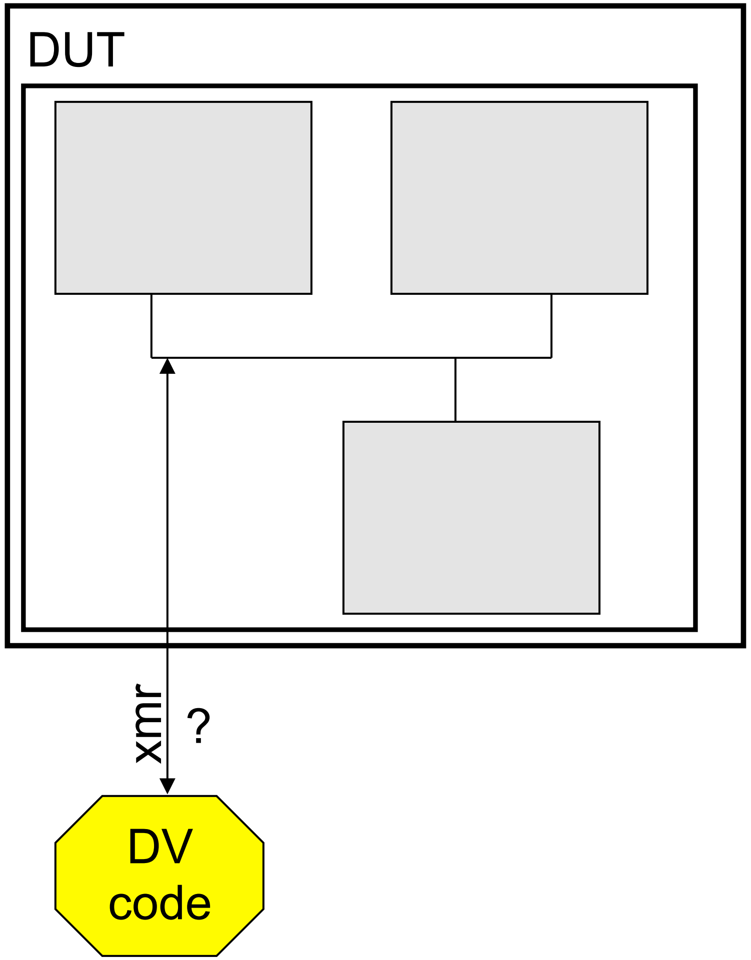
- What does a 'bind' do?
  - Insert "your" code into "others" code.
  - as if you actually modified other designers code
- Why?
  - Modifying others code not allowed / undesirable
  - Keep DV code separate from design code
    - Hope to reuse.



Introduction

- Could have kept code separate at top level and used hier. references
- Susceptible to hierarchy changes.

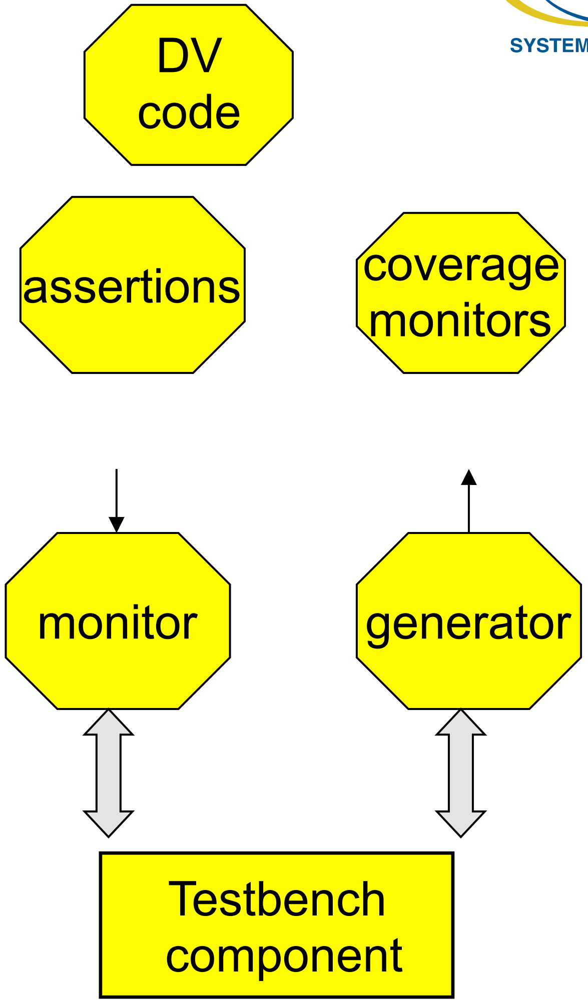




What to bind?

- What kind of DV code?
  - assertions
  - coverage

  passive
- Active code?
  - Transactors
  - Generators
  - Monitors
- Once bound, how to communicate with and control?



SystemVerilog constructs that help

Need to understand the following constructs and design patterns:

- Abstract base classes.

```
virtual class <...>;
  pure virtual task ...
  pure virtual function ...
```

- Packages
  - Encapsulate abstract base classes
  - Implement lookup table
- up-module references

  LRM §23.8.1 (upwards name referencing)

Abstract base classes

All transactors have an API.

  – Translate your wishes into signal wiggles.

- Define tasks/functions which make up API.
- Implement as pure virtual class in package.

```
virtual class xactor_api;
  pure virtual task drive_transaction(...);
  pure virtual function T wait_for_transaction();
```

- Implement transactor.
  - Map signal wiggles to API & vice versa.
  - Extend base class. Construct an object.
  - Register object.

Extend/Implement/Register

- API in previous slide is abstract base class
  - Need to extend and implement
- Construct object of extended class.
- Register into dropbox

```
class my_xactor_api extends
                         xactor_api;
  task drive_transaction(...);
       // blah blah
  endtask
  function T wait_for_transaction
   ();
       // blah blah
  endfunction
endclass
```

fill-in details

```
my_xactor_api _my_api = new;
```

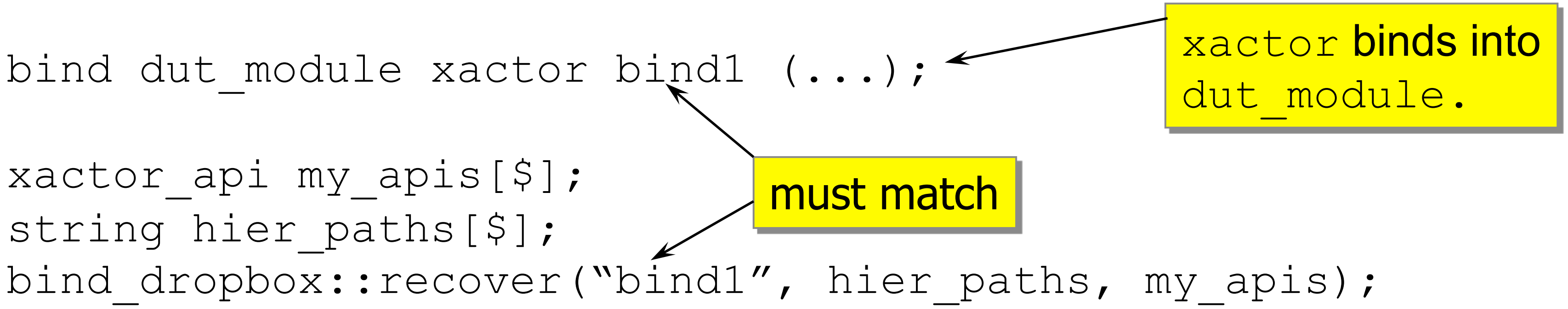
```
initial bind_dropbox::register(
        $psprintf(“%m”),
        _my_api);
```

unique string

Bind & Recover

BIND
- TB binds xactor into DUT

RECOVER
- A lookup table implemented in a package.
  - Globally visible. Singleton. (because it is in a package)
  - Associates “strings” with “API objects”
- Transactors REGISTER their APIs into dropbox @ time 0.
- TB components RECOVER API objects afterwards.



```
bind dut_module xactor bind1 (...);

xactor_api my_apis[$];
string hier_paths[$];
bind_dropbox::recover(“bind1”, hier_paths, my_apis);
```

`xactor` binds into `dut_module.`

must match

Bind dropbox details

REGISTER

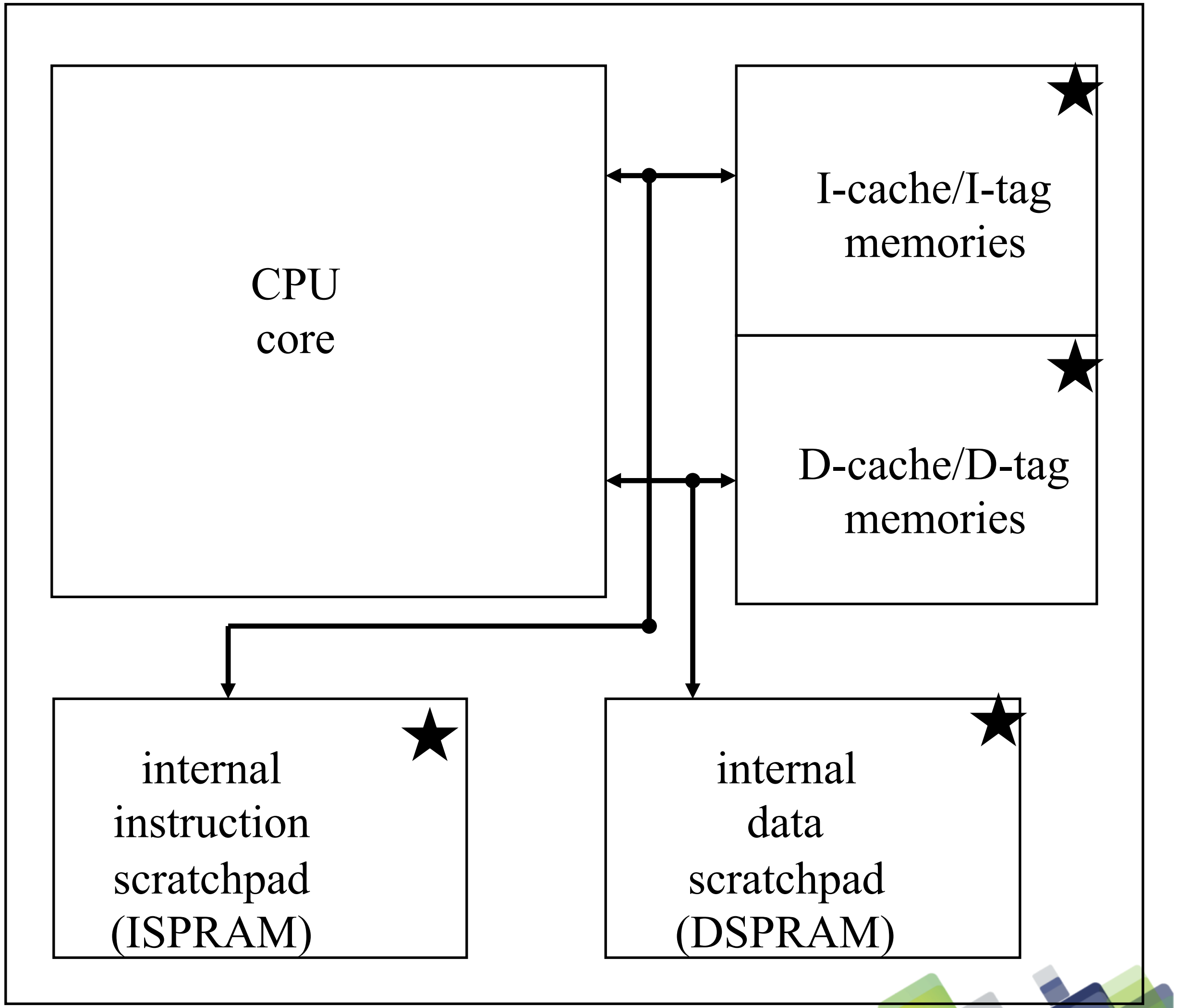
- bound xactor registers API in dropbox.
- %m is unique string -> used as key.
  - last part is bind instance name (e.g. bind1)
- xactor could be multiply instantiated (if target module is)
  - multiple entries in dropbox.
  - last part for all are identical.

RECOVER

- For each bind, TB component calls RECOVER
  - Use bind instance name as key.
- dropbox returns
  - array of API objects matching key.
  - array of %m strings matching key.
- TB uses %m strings to separate multiply instantiated case.

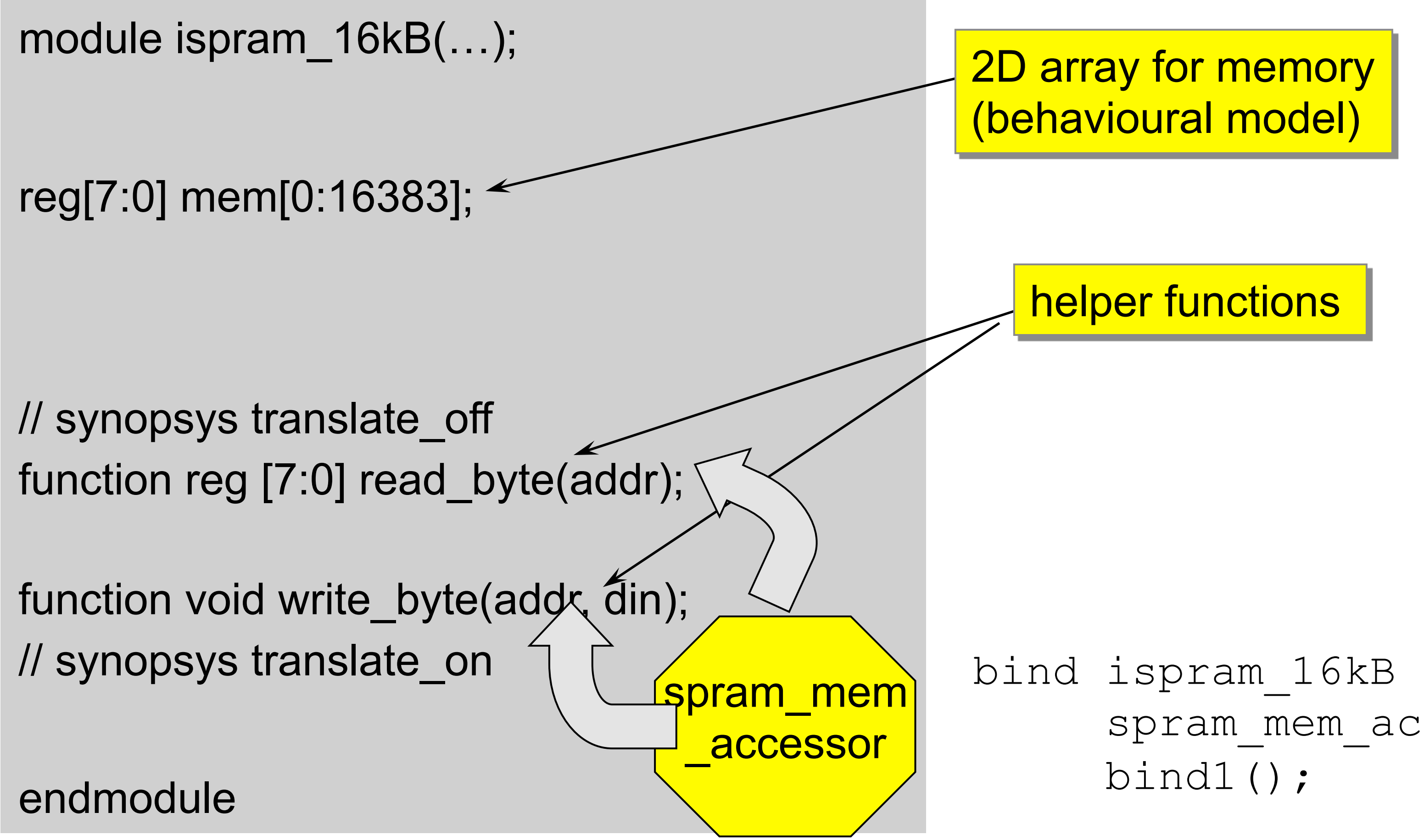
Real Life example

- Initializing internal RAMs in SoC.
  - Contents of SPRAM contain code/data for CPU.
  - Must be initialized before reset.
- Need backdoor (zero time) methods



Real Life example

- Memory wrappers organized as follows:



```
module ispram_16kB(…);


reg[7:0] mem[0:16383];



// synopsys translate_off
function reg [7:0] read_byte(addr);

function void write_byte(addr, din);
// synopsys translate_on


endmodule
```

2D array for memory (behavioural model)

helper functions

spram_mem_accessor

```
bind ispram_16kB
     spram_mem_accessor
     bind1();
```

Up-module reference

- spram_mem_accessor uses up-module reference.

```
module spram_accessor();

function reg [7:0] _read_byte(addr);
  return read_byte(addr);
endfunction

function void _write_byte(addr, din);
  write_byte(addr,din);
endfunction

class my_mem_accessor extends
                    mem_accessor_base;
 ...

my_mem_accessor _api = new
initial bind_dropbox::register(...);
endmodule
```

up-module ref.
into host module.

abstract base class

CREATE/
REGISTER

Putting it all together

- "xactor" to bind: `spram_mem_accessor`. API:
  - `function reg [7:0] read_byte(input [31:0] addr);`
  - `function void write_byte(input [31:0] addr, [7:0] din);`
- bind target: SoC internal memories
  - `bind ispram_16kB spram_mem_accessor bind1();`
  - `bind dspram_16kB spram_mem_accessor bind2();`
- WHY? Runtime backdoor access. Init uProc program.
  - compile 'C' program. gcc -> ELF file.
  - convert ELF to SREC. objcopy –O srec *.elf
  - parse SREC to obtain addr/data pairs to write.
  - map addr to appropriate memory. Use API to write bytes.
    - API was recovered with bind_dropbox::recover.

Conclusion

- Presented binding transactors which can be actively controlled at runtime.
- Key concepts:
  - Abstract base classes
  - Packages
  - bind dropbox
  - upwards name referencing.
- Example use-case
  - Loading ‘C’ programs into full-chip SoC simulations.
  - integrated hardware/software verification

Thank you!

QUESTIONS?