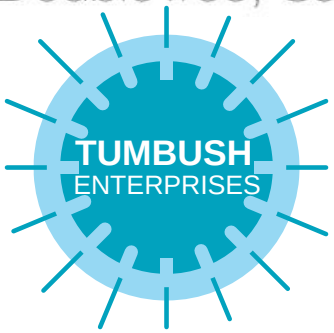
February 25-28, 2013
DoubleTree, San Jose

accellera SYSTEMS INITIATIVE



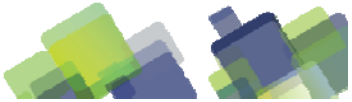
Design and Verification of an Image Processing CPU Using UVM

by
Greg Tumbush
Tumbush Enterprises

Co-author
Milos Becvar
EM Microelectronic-US

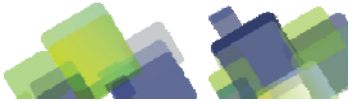
Agenda:

- Introduction
- Design of CPU
- Verification of CPU
- Functional Coverage
- Results
- Summary



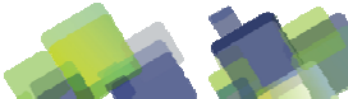
Agenda:

- **Introduction**
- Design of CPU
- Verification of CPU
- Functional Coverage
- Results
- Summary



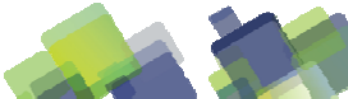
Introduction:

- Work performed at EM Microelectronic-US
- Co-author, Milos Becvar, did design and software
- EM's ASICs
  - Mixed signal
  - Very low power
  - Optimized for area
  - Small design teams
  - Short development times
  - Low cost, high volume
- Not satisfied with previous verification results



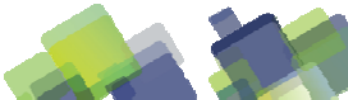
Agenda:

- Introduction
- **Design of CPU**
- Verification of CPU
- Functional Coverage
- Results
- Summary

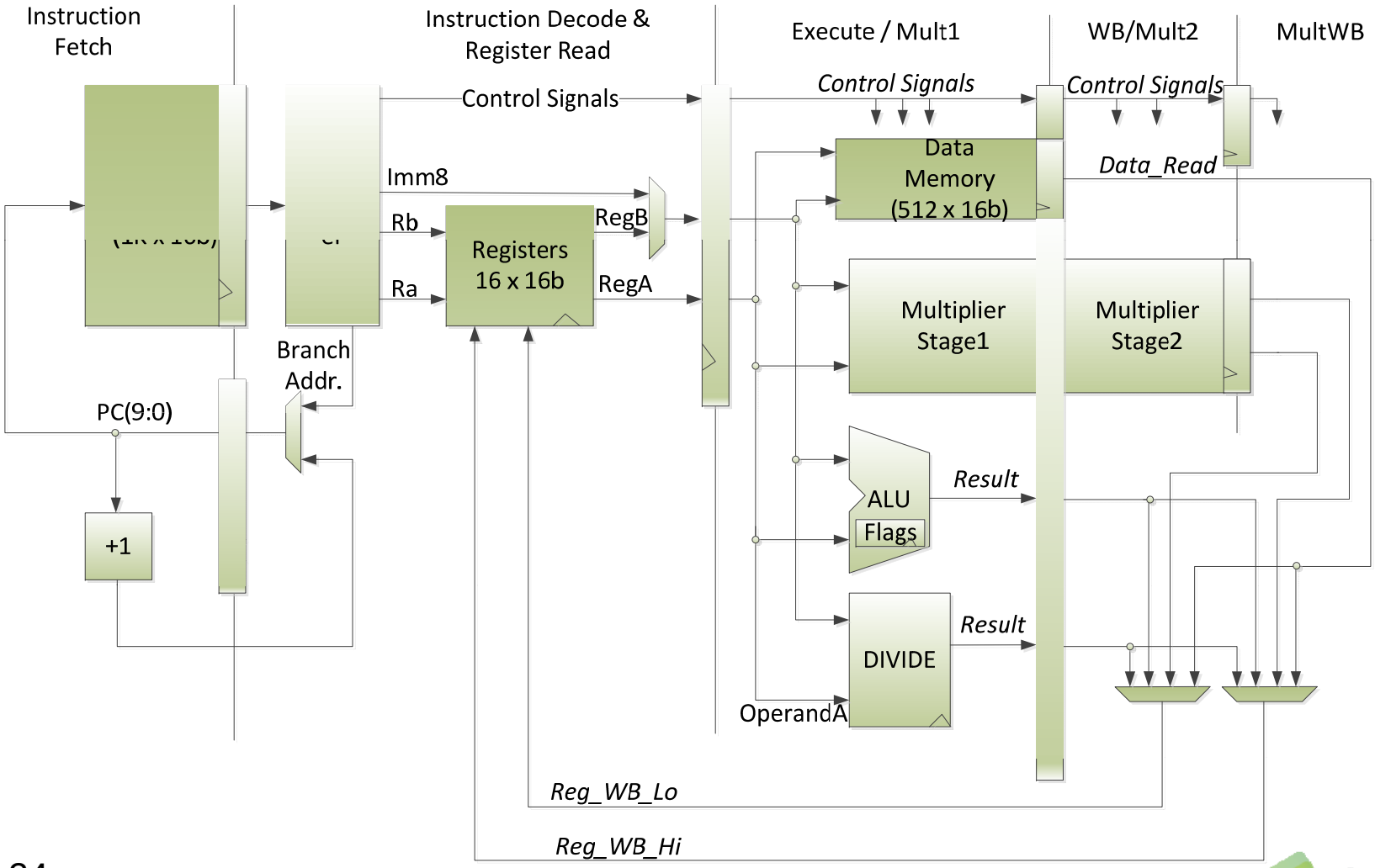


Design of CPU: overview

- Why a CPU?
- Not a trivial effort, need to justify
- CPU requirements for our application
  - No support for C/C++ or legacy SW
  - Efficient execution of application
  - Small silicon area
- Algorithm requirements
  - 12-bit data words
  - Multiplication and division
  - Support of hardware accelerator



Design of CPU: architecture



Design of CPU: hazards

- Data hazards

```
ADD R1, R0 ;; R1=R1+R0
SUB R2, R1 ;; R2=R2-R1
```

Old value of R1 used

- Control hazards

```
<branch instruction>
<instruction after branch>
```

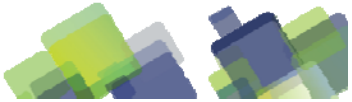
Always taken

- Structural hazards

```
<multiply instruction>
<cannot write into register>
```


Agenda:

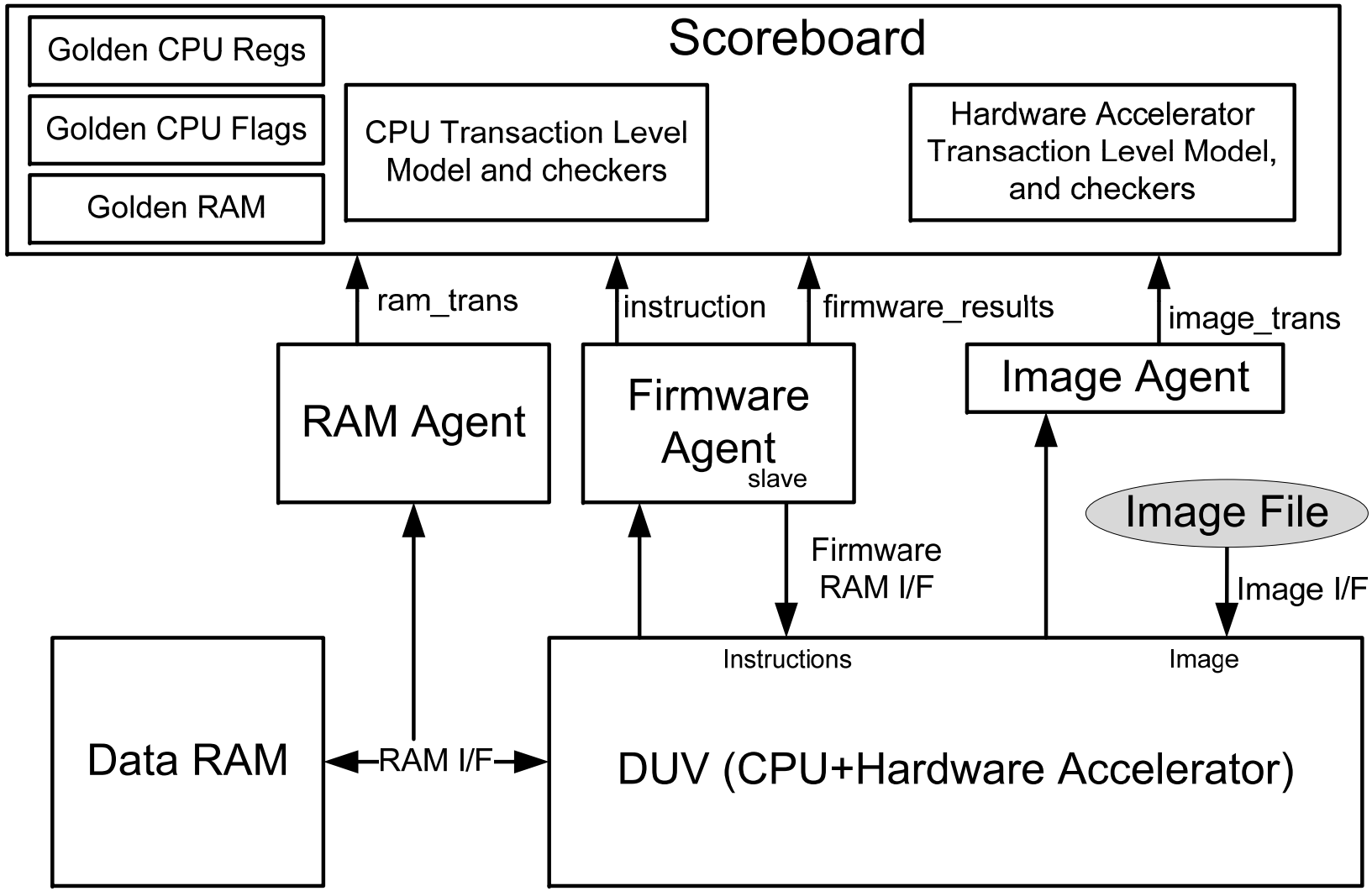
- Introduction
- Design of CPU
- **Verification of CPU**
- Functional Coverage
- Results
- Summary



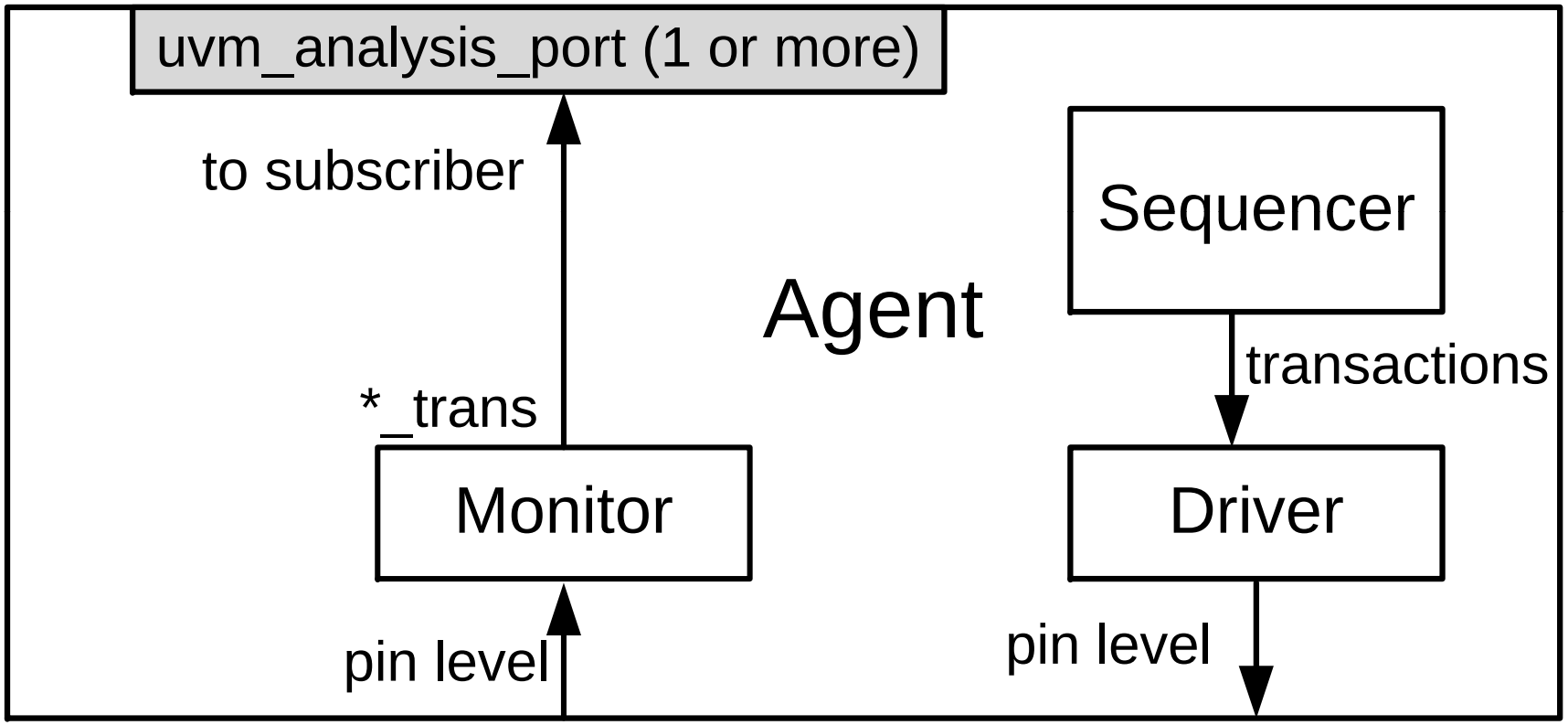
Verification of CPU: overview

- Unique verification challenges
- Chosen methodology must:
  - have a quick development time
  - randomly generate instructions
  - steer the randomly generated instructions into interesting corner cases
  - use functional coverage to stop the testbench once functional coverage is obtained
- UVM is a tough sell to the uninitiated

Verif. of CPU: architecture

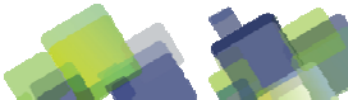


Verif. of CPU: UVM Agent



Verif. of CPU: Firmware Agent

- An active agent
- Sequencer creates *instruction* objects
- Driver “assembles” *instruction* objects
- Monitor:
  - Collects state of CPU
  - Stores information in a *firmware_results* object
  - Passes *firmware_results* object to subscriber
  - Collects machine code fetched by DUV
  - Stores machine code in *instruction* object
  - Passes *instruction* object to subscriber



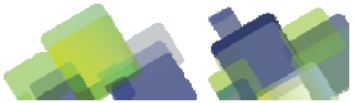
Verif. of CPU: instruction

```
class instruction extends uvm_sequence_item;
  `uvm_object_utils(instruction)
  function new(string name="");
    super.new(name);
  endfunction
  rand opcode_e opcode;
  rand src_dest_e Rb, Ra;
  rand bit [9:0] Addr;
  ……
  static string instruction_str;

  function string get_instruction();
  endfunction
endclass
```

See in waveform!

Disassembler



Verif. of CPU: firmware_results

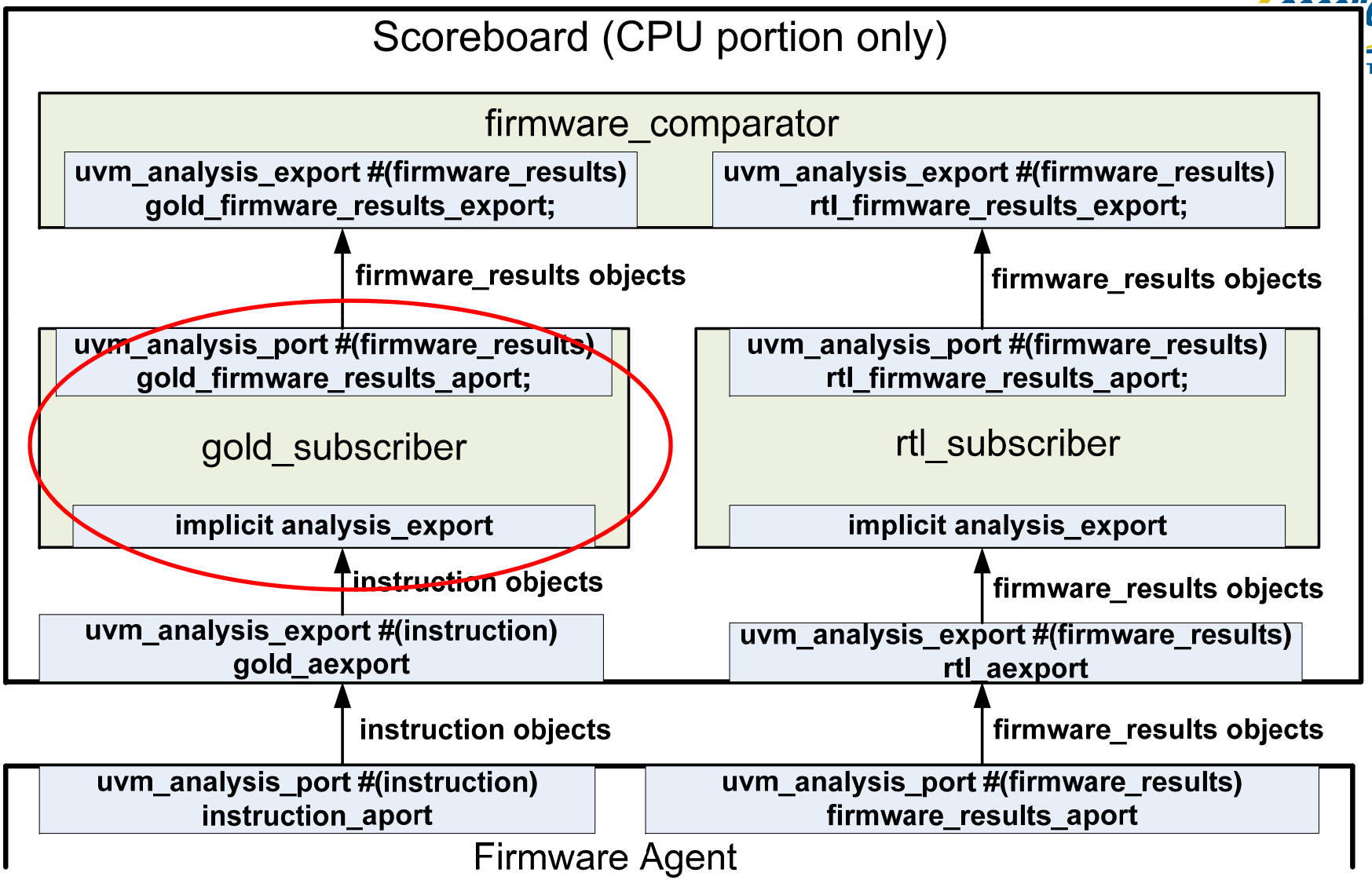
```
class firmware_results extends  uvm_sequence_item;
  `uvm_object_utils(firmware_results)
  logic signed [15:0] gp_reg[16];
  logic SF, OF, ZF, CF;
  logic [FIRMWARE_RAM_ADDR_WIDTH-1:0] PC, RAR;
  logic [15:0] instr_code;

  bit ram_write;
  bit [RAM_ADDR_WIDTH-1:0] write_addr;
  bit [RAM_DATA_SIZE-1:0] data_write;
  ...
endclass
```

State of CPU

Write transaction

Verif. of CPU: scoreboard



Verif. of CPU: gold_subscriber

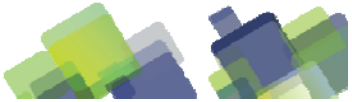

```
class gold_subscriber extends uvm_subscriber #(instruction);
  `uvm_component_utils(gold_subscriber)
  firmware_results firmware_results_h;
  uvm_analysis_port #(firmware_results) gold_firmware_results_aport;
  function void build_phase (uvm_phase phase);
    // create gold_firmware_results_aport object
  endfunction
  function void write(instruction t);
    case (t.opcode)
      ADD: firmware_results_h.gp_reg[t.Rb] = val(t.Ra) + (t.Rb);
      ....
    endcase
    gold_firmware_results_aport.write(firmware_results_h);
  endfunction
endclass
```

Golden model

Send to gold_firmware_results_aport

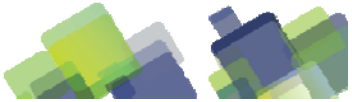
Agenda:

- Introduction
- Design of CPU
- Verification of CPU
- **Functional Coverage**
- Results
- Summary



Functional Coverage

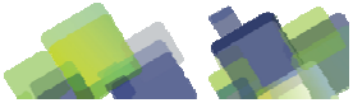
- Instructions can be grouped
  - Two 16-bit operands
  - One 16-bit operand
  - One 16-bit operand and 8-bit immediate
  - etc.
- Create coverpoints for each operand, immediate,etc
- Cross coverpoints for each instruction group
- Monitor cross coverage, reduce probability if coverage=100%



Functional Coverage: LDR

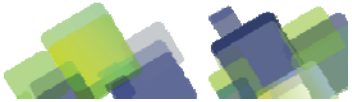

```
all_opcodes_Ra_Imm8: coverpoint instr.opcode {
   bins opcodes[] = {…, LDR, STR, …}; option.weight = 0; }
all_Ra: coverpoint instr.Ra{ option.weight = 0;}
Imm8_range: coverpoint instr.Imm8 {
   bins maximum = {(2**8)-1};
   bins mid     = {[1:(2**8)-1]};
   bins minimum = {0};
   option.weight = 0; }
all_opcodes_Ra_Imm8_x_Ra_x_Imm8_range: cross
                    all_opcodes_Ra_Imm8, all_Ra, Imm8_range;
```

Monitored during simulation



Agenda:

- Introduction
- Design of CPU
- Verification of CPU
- Functional Coverage
- **Results**
- Summary



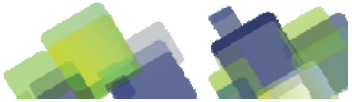
Results

- Design spec to verification closure: 12 weeks
- ~14,000 coverage bins
- 350,000-450,000 random instructions required
- 100% statement coverage with no extra tests.
- No additional bugs found during system level verification
- Verified assembler as well
- Silicon evaluation revealed no new bugs

First Pass Success!

Agenda:

- Introduction
- Design of CPU
- Verification of CPU
- Functional Coverage
- Results
- **Summary**



Summary

- A custom cpu is warranted in some situations
- Don’t forget the software task
- Verifying a CPU is hard!
- Don’t try to test everything at the block level
- UVM appropriate for small ASICs/FPGAs
- UVM appropriate for block level verification
- Thanks to EM Microelectronic-US