Design and Verification of an Image Processing CPU using UVM

Milos Becvar
EM Microelectronic-US, Inc.
Colorado Springs, CO
milos.becvar@emmicro-us.com

Greg Tumbush
Tumbush Enterprises
Colorado Springs, CO
greg@tumbush.com

***Abstract*— This paper presents the design and verification of a custom image processing CPU and associated hardware accelerator using UVM. The CPU and accelerator are targeting an ASIC. This work was performed at EM Microelectronics-US.**

Keywords—SystemVerilog, UVM, CPU, Image Processing

I. INTRODUCTION

This paper presents the design and verification of a custom image processing CPU and associated hardware accelerator using UVM. The CPU and accelerator are targeting an ASIC. This work was performed at EM Microelectronics-US.

The ASICs that EM Microelectronics-US develops are typically very lower power with standby current in the uA range and operating current in the 100's of uA range. Low power modes are used extensively. Their ASICs are optimized for area, hence, memory is to be kept to a minimum. The ASICs are typically dominated by analog, and the design teams are small, 1 or 2 designers and a similar number of verification engineers. Their design cycles are short, concept to tapeout in less than 12 months. Their ASICs must sell in a market that is typified by low cost and high volume.

For this project Mr. Becvar, a co-author, was the architect, designer, and firmware engineer. Dr. Tumbush was brought in as a consultant to verify the ASIC and to lead the small verification team. This ASIC was a major enhancement from a previous ASIC, the new blocks being the custom CPU and hardware accelerator. It was decided that the new blocks would be best verified in a block level verification environment. An existing verification environment written in VHDL would be leveraged to test the system.

The contributions this paper will make are:

1. Determining when a custom CPU design is appropriate.
2. How to choose when block level or system level verification is appropriate.
3. When using UVM at the block level is appropriate.
4. Special considerations when verifying a CPU.
5. A real world complete example of using UVM for verification with results.
6. A UVM verification environment is not just for huge ASICs. It can be appropriate for small ASICs and even block level verification.
7. Selling UVM to hesitant clients. Bring them kicking and screaming into the 21st century.

II. DESIGN OF A CUSTOM IMAGE PROCESSING CPU

A. CPU Requirements

The design and verification effort of a custom CPU core is significant and its benefits over using an off-the-shelf CPU core must be justified. A detailed analysis of the application requirements and currently available CPU cores is a prudent step before committing to the design of a custom CPU core. *Table 1* outlines the requirements driving the CPU core design in the author's application.

Table 1: CPU Requirements

Requirement	CPU Feature
Legacy software and HLL support not required	Minimal CPU core tailored to application
Efficient execution of target application	CPU word size and operations designed to minimize number of execution cycles
Small silicon area footprint	CPU complexity 8K gates, 1K of instructions required
Sufficient timing-closure design margin in targeted process	Pipelining of operations, avoiding complex logic before and after memories
Limited routing complexity in target process	Avoiding barrel shifter and bypassing networks

The requirement to support legacy software and/or a high-level language is a driving factor when selecting a CPU core for many applications. This requirement typically limits designer flexibility to a set of binary compatible cores with available software tools. In the author's application this requirement was not present, simplifying the task and allowing development of a very efficient CPU specific to the image processing algorithm but still allowing flexibility for algorithm exploration.

Unlike a general purpose CPU which has to support many different and sometimes unknown applications, the application for the CPU was known at design time. Consequently, a minimalist CPU which would efficiently support the algorithm's execution could be designed. Efficiency can be

measured as the number of execution cycles required to calculate a given algorithm. Minimization of execution cycles allows a reduction in clock frequency and operating voltage and consequently, lower power consumption.

B. Algorithm Requirements

A CPU that meets the hardware requirements but cannot execute the required algorithm is not useful. The CPU designed must not only meet the hardware requirements, speed, power consumption, etc. but also the following algorithm requirements.

1. 12-bit data words for core calculation with some limited use of 24-bit words
2. Limited code and data size – requiring only 10-bits for memory addressing
3. Support of multiplication and division
4. At least 16 word-size registers for core calculation to avoid frequent memory accesses (minimizing power consumption)
5. Efficient support for a hardware accelerator for pattern matching.

The required data word size and supported operations were one of the factors leading to a custom CPU core design. The algorithm requires data widths of greater than 8-bits and multiplication and division operations. This eliminates the possibility of using an off-the-shelf 8-bit CPU core. Porting the image processing application to a standard 8-bit core is possible, but very inefficient in terms of execution cycles and instruction count.

A standard 32-bit CPU core would be sufficient to efficiently support the algorithm. Unfortunately, it would have a significant silicon area footprint (>10,000 gates) and would require additional investment and payment of royalties over the lifetime of the project.

For the above reasons, the decision was made to build a custom 16-bit CPU core for an image processing application. In order to extend the usability of the core and match available memory modules, the CPU word was extended from a minimally required 12-bit to a more practical 16-bit. The implementation of a pipelined CPU with multi-word arithmetic support was based on the Mr. Becvar's previous work in [4] and [5].

C. CPU Overview

After taking the hardware and algorithms requirements into account it was determined that the CPU must have the characteristics listed in Table 2.

Table 2: Custom CPU Core Overview

CPU Characteristic	Implementation
Instruction Set Architecture	16-bit Load/Store Harvard Architecture RISC
Data word supported by ISA	16 bit, 32bit result of multiplication in register pairs
Number of registers	16 x 16bit
Instruction encoding	Fixed , 16bit instruction word
Clock frequency (0.18 um)	50 MHz implemented (scalable up to 100 MHz)
Number of operands in instruction	2 (1 destination/source, 1 source)
Addressable code space	1 K instructions used (up to 64 K possible)
Addressable data space	512 words used (up to 64 K possible)
Peripherals connection	Memory mapped
Memory addressing modes	Immediate, register-indirect, direct addressing of peripherals. R14 behaves as stack-pointer, R15 supports post-increment memory addressing
Special features in ISA	Multi-word operation support (flags), fused addition and shift operation, fractional division, support of saturation arithmetic, synchronization with accelerator by WAIT instruction
Features not supported	Interrupts, byte operations

D. CPU Architecture

The CPU architecture can be seen in Figure 1. A single instruction is fetched every cycle and subsequently executed in 3 to 4 pipeline stages. The number of stages was conservatively selected to avoid timing closure problems during implementation. Most operations are completed within a single cycle with the exception of multiplication which is calculated in two stages. The pipeline supports independent execution of division instruction while contining to execute other integer instructions.

Instruction pipelining introduces the well-known issues of data, control and structural hazards [6]. These problems have well-documented HW solutions but they generally increase CPU complexity and some of them might introduce routability and time closure issues (notably bypassing the network for Read After Write (RAW hazards)). An alternative to HW solutions to the hazard problem is to push the responsibility for execution correctness to the programmer and/or compiler. In this approach, the latencies of the pipeline are fully exposed to the software. This approach dates back to early RISC architectures [7] (branch and load delay slots) and it is exploited by compiler scheduled VLIW processors [8] and [9].

As an example, consider the assembly code in Code Snippet 1. General purpose register R1 is the destination for the ADD instruction and a source for the SUB instruction. The result of the ADD instruction is still in the execute pipeline when the operands for the SUB instruction are fetched. Therefore the SUB instruction will use the "old" value of R1, not the value calculated by the ADD. A typical CPU controller would stall the SUB instruction to eliminate this RAW hazard. For the author's custom CPU, the controller is not checking for this type of hazard and allows the SUB instruction to read the "old" value of register R1.

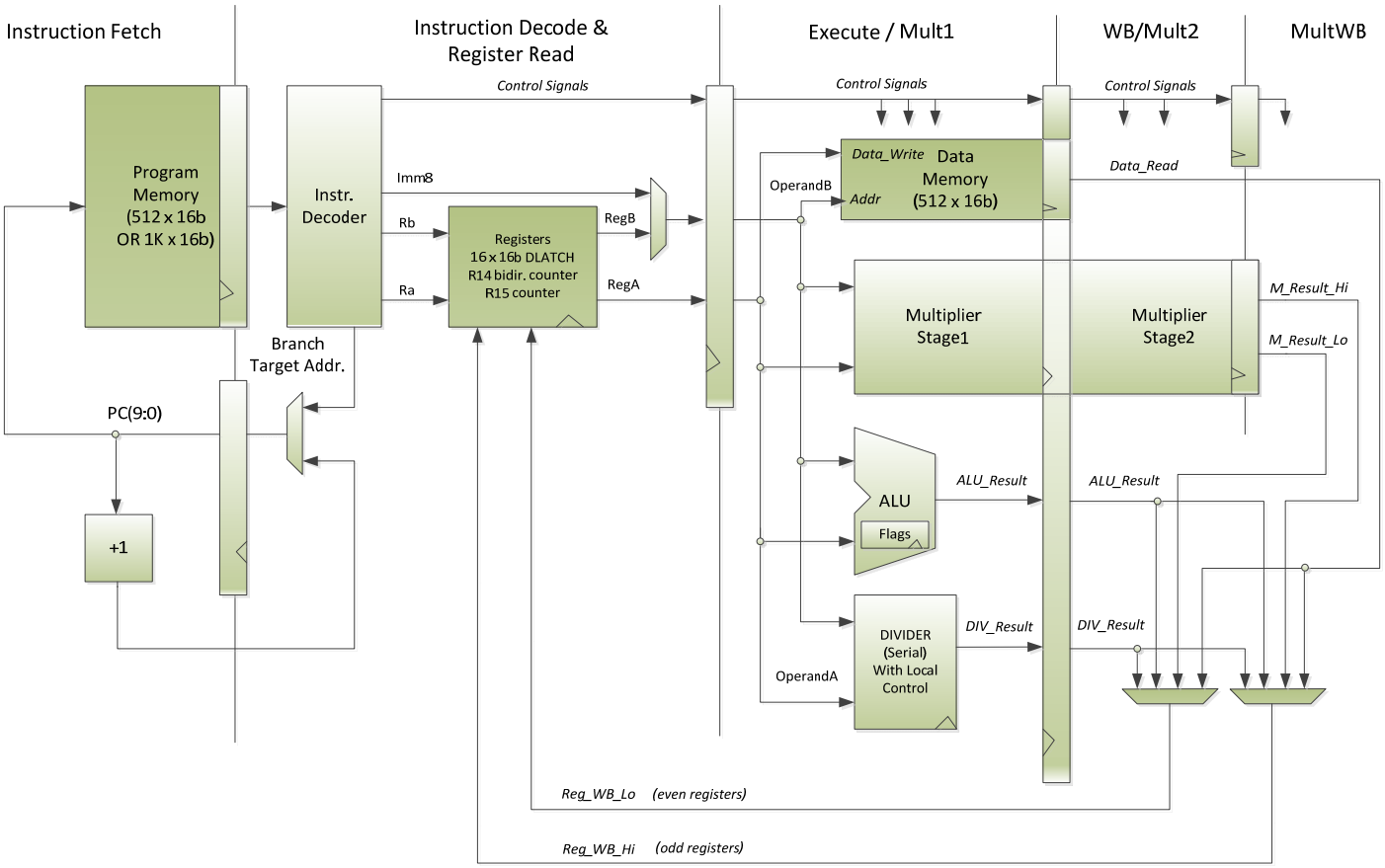

Figure 1: CPU Architecture

It is up to the programmer to ensure correctness and utilize each execution slot.

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

Code Snippet 1: Data Hazard

Analysis of the algorithm indicates that there is enough parallelism to assume that the programmer would be able to efficiently utilize available instruction slots. Examples of constraints when developing software for this CPU are described in Table 3.

Table 3: Software hazards

Type of Hazard	Programmer constraint
Data Hazard	Result of instruction is available with one cycle latency (multiplication has two cycles latency). Flags are available with 0-cycle latency, allowing chaining of multi-word operations. Division latency depends on result precision.
Control Hazard	Instruction after branch is always executed (1 cycle delay slot)
Structural Hazard	Instruction after multiplication cannot write into register

Most common instructions in the Instruction Set Architecture (ISA) have a short latency (1-2 cycles) which could be overlapped by independent instructions. However, external pattern-matching acceleration and division take much longer and requires special ISA support.

Pattern matching is a very data intensive calculation which is supported by an independent HW accelerator which has direct connection to memory. Execution is performed in multiple sweeps which compares two spatially shifted bitstreams (internal forms of images). The HW accelerator is controlled through memory mapped control registers. Since each sweep takes around 30 cycles, there is a special need to inform the CPU when the sweep is completed. This synchronization is performed by WAIT instruction which selects one out of 16 possible synchronization signals. Fetching of instructions is stalled until a given synchronization signal is asserted by the accelerator. The CPU clock is gated to save power during waiting on external synchronization. This synchronization mechanism can be utilized to connect other external accelerators with variable latency.

Similar to the accelerator interface, the division operation is broken into two parts (see Code Snippet 2). The DIV instruction reads source operands from a register file and starts a multi-cycle division operation. Result of division is

transferred by separate MOVDRES instruction into register file.

The author's custom CPU implements a fractional division (result a/b<1) as required by the application. In order to produce an n-bit fractional result, n+1 cycles are required. 17 cycles is required to produce a full 16-bit fractional result. Depending on the required precision of the result, the programmer can insert independent instructions between DIV and MOVDRES. These instructions are executed in parallel with division. If no more independent instructions are available, the WAIT instruction can be used to synchronize for the end of division. A special control register defines the required number of cycles for the divider synchronization signal. This reduces need to include NOPs while waiting for the division result.

```
DIV R1 R0  ;;; calculate R1 /  R0
Independent instructions
…
WAIT divider
MOVDRES R2 ;;; store division result to R2
```

Code Snippet 2: Division

Statistics of the implementation indicate that only 6% of executed instructions are NOPs demonstrating high efficiency of the custom CPU when executing a given application. Not implementing hazard resolution logic simplifies the CPU controller design and moves the burden up to the programmer. However, it also introduces unique challenges for creating a transaction accurate reference CPU model for verification.

III. Verification of a Custom Image Processing CPU

A CPU presents a unique verification challenge, particularly one with a pipeline, jumps, branches, multi-cycle instructions, and hazards. The chosen verification methodology needs to:

1. have a quick ramp up time
2. randomly generate instructions
3. steer the randomly generated instructions into interesting corner cases
4. use functional coverage to stop the testbench once functional coverage is obtained

UVM was chosen as the verification methodology because it is a proven methodology, fully supports randomization, and fully supports a coverage driven testbench. It has been Dr. Tumbush's experience that selling a UVM based testbench to clients is difficult. They see UVM as only for huge ASICs and have been burned by employees and consultants blowing the schedule by creating a much more complex testbench than is necessary. To allay these fears the client must be educated in how a functional coverage driven testbench results in quicker verification closure and less bugs which means an earlier tapeout and less chance of a re-spin. The verification environment used that meets the above requirements is depicted in Figure 2 and will be explained in the following subsections.

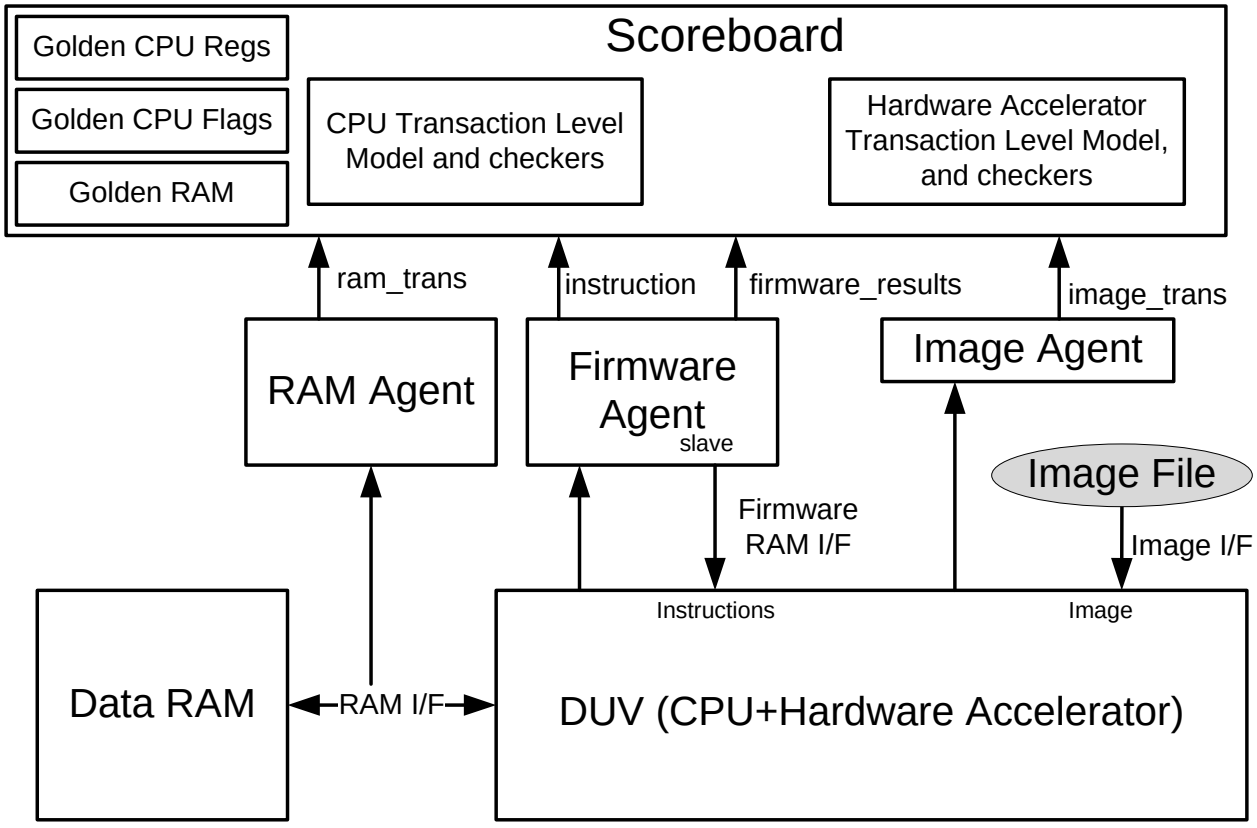

Figure 2: Verification Environment

A. *Agents*

The verification environment created uses 3 agents to observe and possibly drive the 3 main interfaces, the Data RAM, Firmware, and Image. A generic UVM agent is depicted in Figure 3. A decision that a verification engineer needs to make is whether to create separate transaction and results classes for each agent or create a single class for each agent that encompasses both the transaction and results. For this verification environment it was decided to create separate classes for the transaction and the results. The authors felt that this makes the Sequencer, Driver, and Monitor simpler because they have less functionality and operate more autonomously.

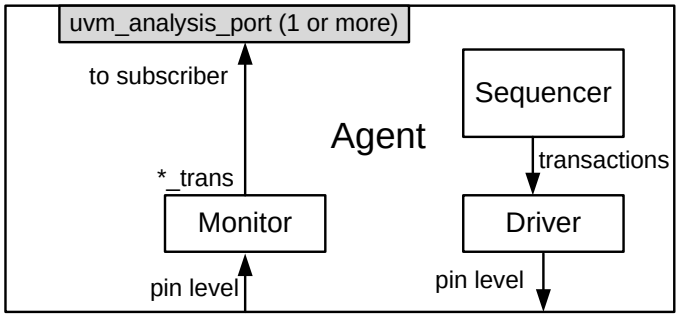

Figure 3: UVM Agent

The Sequencer simply creates transaction objects. The Driver is responsible for deconstructing the transaction objects into the necessary pin level interface to drive the DUV. The Monitor observes the pin level interface to 1) collect the result of a transaction and 2) collect any necessary information about the transaction to predict the result. The collected information is packaged into an object and passed to 1 or more *uvm_analysis_ports* that the Scoreboard can subscribe to.

When researching UVM Dr. Tumbush was perplexed by the recommendation to not pass transaction objects from the sequencer directly up to the scoreboard. Why reconstruct portions, or all, of a transaction at the monitor interface? It is known what the transaction is why not just pass it directly to the subscriber? By collecting coverage at the interface of the DUV, coverage is truly being collected on what the DUV sees, not what one thinks the DUV sees. In one situation Dr. Tumbush thought that the sequencer/driver was correct but it turned out that it was not driving the DUV with the transactions intended. It is very easy to think the test is driving the DUV as expected since a non-intended transaction will be collected and the golden and RTL will match even though the transaction was not as expected. An object oriented verification environment does not obviate the need to continue to look at simulation waveforms.

1) *RAM Agent*

The RAM Agent monitors the RAM I/F bus for memory transactions and creates *ram_trans* objects from the bus traffic as seen in Code Snippet 3. Since any errant read transactions will be caught when the destination register (PC, General Purpose registers, etc) is compared, the RAM I/F is only watching for write transactions. The RAM Agent is a passive agent meaning it only collects data, never driving the Device Under Verification (DUV) as an active agent would do so it does not have a Sequencer or Driver.

```
class ram_trans extends uvm_sequence_item;
  `uvm_object_utils(ram_trans)
  function new(string name="");
    super.new(name);
  endfunction
  logic [RAM_ADDR_WIDTH-1:0] write_addr;
  logic [RAM_DATA_SIZE-1:0]  data_write;
endclass
```

Code Snippet 3: class ram_trans

2) *Firmware Agent*

The Firmware Agent is an active agent because it supplies randomized instructions to the DUV. The Sequencer in the Firmware Agent creates instruction objects as seen in Code Snippet 4. An object of class *instruction* is randomized so any variable denoted as rand such as the opcode, source operand, destination operand, etc, is randomized. The instruction class has a static variable of type *string* called *instruction_str* which holds the current instruction. Since *instruction_str* is static it can be viewed in a simulation waveform which is indispensible for debug. Function *get_instruction()* acts as a disassembler, converting machine code into an instruction string.

The Driver in the Firmware Agent then decomposes instruction objects into a firmware RAM interface, acting as an assembler. The Monitor in the Firmware Agent collects the state of the CPU, stores this information in a *firmware_results* object (see Code Snippet 5) and passes the object to a subscriber. The Monitor also collects the machine code actually fetched by the DUV, stores this information in an instruction object, and passes the object to a subscriber.

Note that class *firmware_results* also has fields for any predicted ram write transactions. These fields will be filled out in the scoreboard and compared to *ram_trans* objects from the RAM Agent.

```
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 [15:0] read_data;
  static string instruction_str;
  rand bit [7:0] Imm8;
  rand bit [9:0] Addr;
  bit [3:0] sig;
  function string get_instruction();
     …
  endfunction
endclass
```

Code Snippet 4: class instruction

```
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
```

Code Snippet 5: class firmware_results

3) Image Agent

The Image Agent collects results from the hardware accelerator that processes the image, records the image, and packages this data into an *image_trans* object. The image is supplied to the DUV via the Image interface from an image file. The Image Agent is a passive agent because its only function is to create *image_trans* objects and pass them to the Scoreboard.

B. Scoreboard

The Scoreboard in Figure 2 calculates golden results from *firmware_results* or *image_trans* objects. The calculated golden results are compared to the actual results from the *ram_trans*, *firmware_results*, or *image_trans* objects. The Hardware Accelerator transaction level model accepts *image_trans* objects and processes the image to create golden results. The CPU transaction level model accepts *firmware_results* objects, and from that, calculates the golden "State of the CPU".

Due to the proprietary nature of the hardware accelerator this paper will not go into detail on this portion of the scoreboard. In the next sections the paper will go over a detailed explanation of the CPU transaction level model and checkers. This should have wider interest.

The CPU transaction level model and checker blocks of the Scoreboard will perform the following functions:

1. Collect instruction objects from the Firmware Agent's Monitor block
2. From the instruction objects predict the expected results
3. Collect *firmware_results* objects from the Firmware Agent's Monitor block
4. Compare expected results to actual results

A block diagram of this portion of the verification environment is in Figure 4.

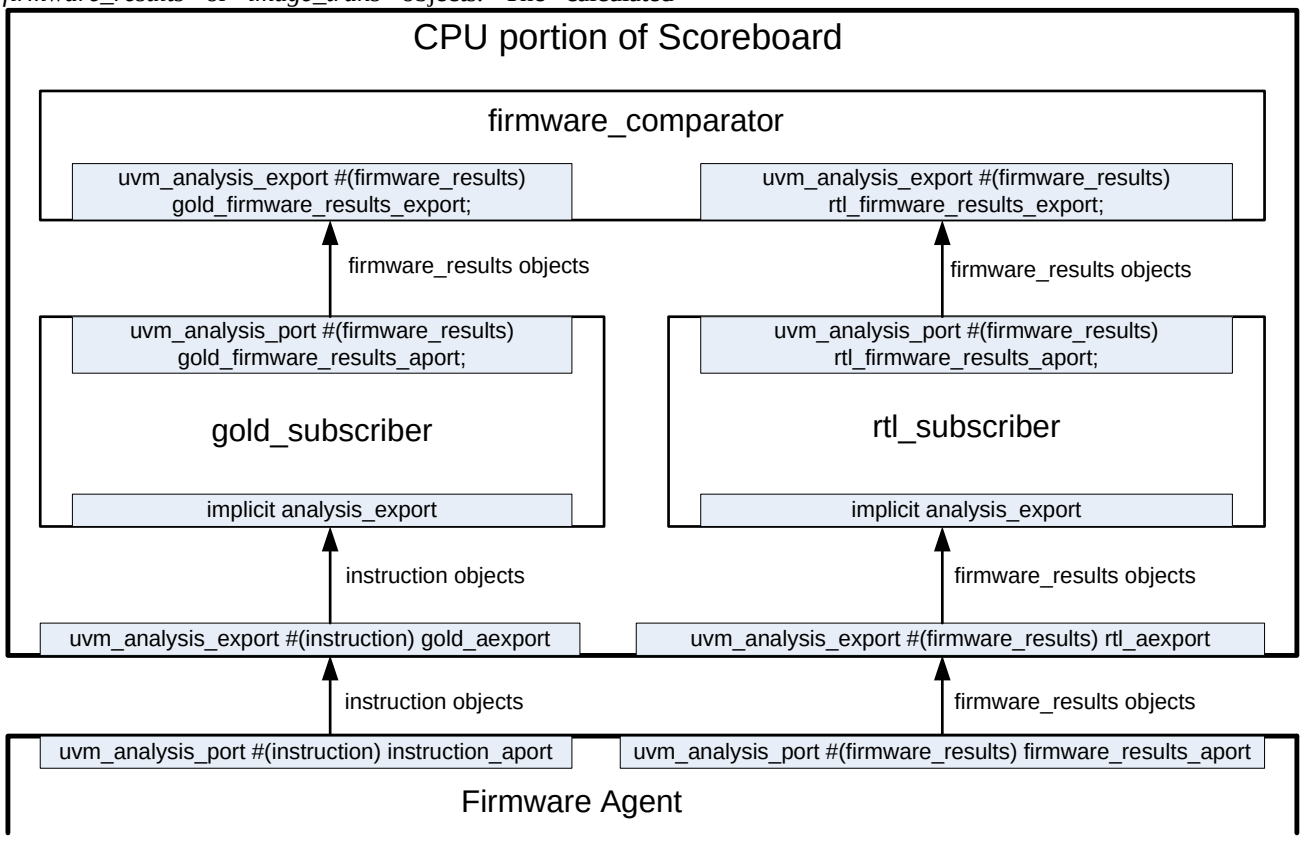

Figure 4: CPU Portion of Scoreboard

1) gold_subscriber

The *gold_subscriber* accepts instruction objects through its implicit analysis export. From the instruction objects the expected state of the CPU is calculated, packaged into a *firmware_results* object, and sent to the *gold_firmware_results_aport* with a *write()* function. A subset of the *gold_subscriber* class is in Code Snippet 6. This code shows an ADD instruction being calculated at the transaction level. The un-shown *val()* function determines the current 16-bit value of a general purpose register from its enumerated name. The golden model or predictor is the white elephant in the room very few people talk about. To create a golden model or predictor requires a designer's level of knowledge about the DUV and can take a very long time to create. It is critical to think in terms of transactions. In this case, it was important to match the state of the CPU at every clock cycle due to the pipeline and jumps. Since the firmware engineer is using the structural and control hazards to his advantage these must be modeled in the golden model as well. This is not always the case. Only model as low a level as absolutely necessary.

```
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);
      gold_firmware_results_aport = new("gold_firmware_results_aport", this);
   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
```

Code Snippet 6: class gold_subscriber

2) rtl_subscriber

The *rtl_subscriber* accepts *firmware_results* objects through its implicit analysis export. It then sends the objects to the *rtl_firmware_results_aport* with a *write()* function. At this point the *rtl_subscriber* performs no functionality, it is included for symmetry with the *gold_subscriber* and for future expansion. The *rtl_subscriber* class is in Code Snippet 7.

```
class rtl_subscriber extends uvm_subscriber #(firmware_results);
   `uvm_component_utils(rtl_subscriber) // Register class subscriber
   uvm_analysis_port #(firmware_results) rtl_firmware_results_aport;
   firmware_results firmware_results_h;
   function new(string name, uvm_component parent);
     super.new(name, parent);
   endfunction
   function void build_phase (uvm_phase phase);
     rtl_firmware_results_aport = new("rtl_firmware_results_aport", this);
   endfunction
   function void write(firmware_results t);
     firmware_results_h = t;
     rtl_firmware_results_aport.write(firmware_results_h);
   endfunction
endclass
```

Code Snippet 7: class rtl_subscriber

3) firmware_comparator

The *firmware_comparator* class has two *uvm_analysis_export* ports, both accepting *firmware_results* objects. One port is the golden firmware results and the other is the rtl firmware results. A *uvm_tlm_analysis_fifo* is used to hold the golden and rtl results for comparison. Each *uvm_tlm_analysis_fifo* is connected to the *uvm_analysis_export* port through the *connect()* function. Using the *get()* function, *firmware_results* objects are removed from the fifos and pushed onto the appropriate queue. Due to the pipelined nature of the CPU a gold and rtl queue of depth 3 were maintained. This queue was examined for pipeline dependencies and the gold results possibly

modified. A subset of the *firmware_comparator* class is in Code Snippet 8.

```
class firmware_comparator extends uvm_component;
  `uvm_component_utils(firmware_comparator)
  uvm_analysis_export #(firmware_results) gold_firmware_results_export;
  uvm_analysis_export #(firmware_results) rtl_firmware_results_export;
  uvm_tlm_analysis_fifo #(firmware_results) gold_fifo, rtl_fifo;
  firmware_results gold_firmware_results, rtl_firmware_results;
  firmware_results gold_queue[$], rtl_queue[$];
  function new(string name, uvm_component parent);
    super.new(name, parent);
    gold_firmware_results = new();
    rtl_firmware_results = new();
  endfunction
  function void build_phase(uvm_phase phase);
    gold_firmware_results_export = new(
    .name("gold_firmware_results_export"), .parent(this));
    rtl_firmware_results_export = new( .name("rtl_firmware_results_export"),
    .parent(this));
    gold_fifo = new("gold_fifo", this);
    rtl_fifo = new("rtl_fifo", this);
  endfunction: build_phase
  function void connect_phase(uvm_phase phase);
    gold_firmware_results_export.connect(gold_fifo.analysis_export);
    rtl_firmware_results_export.connect(rtl_fifo.analysis_export);
  endfunction: connect_phase
  task run_phase (uvm_phase phase);
    forever begin
      gold_fifo.get(gold_firmware_results);
      gold_queue.push_front(gold_firmware_results);
      rtl_fifo.get(rtl_firmware_results);
      rtl_queue.push_front(rtl_firmware_results);
    end
  endtask
endclass
```

Code Snippet 8: class firmware_comparator

4) scoreboard connections

The Scoreboard also connects all the ports of type *uvm_analysis_port* to ports of type *uvm_analysis_exports* in the *connect()* phase. In the interest of space only those connections pertaining to the verification of the CPU are included. The *connect()* phase of the Scoreboard class is seen in Code Snippet 9.

```
function void connect_phase(uvm_phase phase);
  super.connect_phase(phase);
  gold_aexport.connect(gold_subscriber.analysis_export);
  rtl_aexport.connect(rtl_subscriber.analysis_export);
  gold_subscriber.gold_firmware_results_aport.connect(firmware_comparator.gold_firmwa
  re_results_export);
  rtl_subscriber.rtl_firmware_results_aport.connect(firmware_comparator.rtl_firmware_
  results_export);
endfunction: connect_phase
```

Code Snippet 9: Scoreboard class connect() phase

C. Environment

The Environment class, which extends from *uvm_env*, creates all the blocks in Figure 2 in the *build_phase* and then connects them in the *connect_phase*. For the CPU portion of the verification environment the code in Code Snippet 10 connects the *instruction_aport* of the Firmware Agent to the *gold_aexport* of the scoreboard and the *firmware_results_aport* of the Firmware Agent to the *rtl_aexport* of the scoreboard.

```
firmware_agent_h.instruction_aport.connect(scoreboard_h.gold_aexport);
firmware_agent_h.firmware_results_aport.connect(scoreboard_h.rtl_aexport);
```

Code Snippet 10: environment class connections

IV. Functional Coverage

The CPU's instructions can be grouped according to their type and number of operands. For example, an ADD instruction has two 16-bit operands, Ra, and Rb. Other instructions such as NOT have 1 operand, Ra. Cross coverage was used extensively to fully verify that each opcode was executed with all possible operands. As an example, consider the group of instructions having as operands, Ra, and an 8-bit immediate such as a load immediate (LDR) or store immediate (STR). Coverpoints for the opcode, operand Ra, and operand immediate were created. These 3 coverpoints were then crossed. The weight on each individual coverpoint was set to 0 to not include it in the coverage results, just the cross is included. To reduce the number of cross coverage bins on instructions using an immediate, 3 bin were created, 0, max, and 1 to max-1. See Code Snippet 11 for the relevant code.

```
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;
```

Code Snippet 11: coverpoints for LDR instruction

As one would expect, obtaining functional coverage on an instruction with only one operand is simpler than an instruction having more operands. As the simulation progressed, coverage was monitored and the weight reduced on an instruction if the coverage was 100%. In this way, if 100% cross coverage was obtained on an instruction having only one operand, the likelihood of that instruction being randomly generated again was reduced. This will tend to steer the random generation of instructions into uncovered areas.

V. Results

Total verification time for the image processing CPU and hardware accelerator was 12 weeks. The verification environment described herein was developed completely from scratch. Mr. Becvar had created his own directed testbench so the bugs that were found with the UVM block level testbench tended to be corner case bugs. These finds delighted the customer. The UVM block level testbench typically required between 350,000 and 450,000 random instructions to achieve functional coverage. Approximately 14,000 functional coverage bins were created. Code coverage was 100% after 100% functional coverage was obtained indicating that the functional coverage was sufficient to declare verification complete. No additional bugs in the CPU or hardware accelerator were found by the system level testbench. Silicon has been evaluated and is considered to be a first pass success.

In addition to verifying the CPU and hardware accelerator, the UVM block level testbench was able to verify the assembler tool. This is because the testbench generates instructions, not machine code. It must "assemble" the instructions before the Firmware Agent's Driver block passes the 16-bit code to the Firmware RAM I/F. The testbench was instructed to write the approximately 350,000 instructions to a file as well as the machine code to other file. The assembler was then executed on the file containing the 350,000 assembly level instructions. The result was then compared with the file containing the 350,000 machine code level instructions. This additional check revealed a number of bugs.

[1] Mentor Graphics Inc., "UVM Cookbook", http://verificationacademy.com/uvm-ovm"

[2] G. Tumbush, Class slides for Advanced Verification Methodology, http://www.uccs.edu/~gtumbush/4280/4280.html

[3] G. Tumbush, C. Spear, "SystemVerilog for Verification: A Guide to Learning the Testbench Language Features", 3rd Edition, 2012

[4] M. Becvar, A. Pluhacek, and J. Danecek, "DOP- A CPU Core for Teaching Basics of Computer Architecture", Proceedings of the 2003 Workshop on Computer Architecture Education.

[5] M. Becvar, "Teaching Basics of Instruction Pipelining with HDLDLX", Proceedings of the 2004 Workshop on Computer Architecture Education.

[6] J.Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, 5th edition, September 30, 2011, Morgan Kaufman Publishers.

[7] J.Hennessy, J.L., Jouppi, N., Baskett, F., Gill, J, "MIPS: A VLSI Processor Architecture", Proceedings CMU Conference on VLSI Systems and Computations, Computer Science Press, October 1981.

[8] J. A. Fisher, "Very Long Instruction Word architectures and the ELI-512", Proceedings of the 10th annual international symposium on Computer architecture, June 13-17, 1983, Stockholm, Sweden, pp.140-150

[9] M. Becvar, S. Kahanek, "VLIW-DLX Simulator for Educational Purposes", Proceedings of the 2007Workshop on Computer Architecture Education.