

Veriest

Formal Verification of a Custom uController – A Case Study

Elchanan Rappaport,
Formal Verification Tech Lead
elchananr@veriests.com



V

solutions

Veriest Solutions introduction

- ASIC Engineering company, founded in 2007
- Headquartered in Israel, 4 additional sites in Europe
- ~100 engineers
- Customers:
 - Tier1 international Semi companies
 - Start-up companies
 - EDA companies
 - System companies

Some of our professional services



Front-end design



Functional Verification



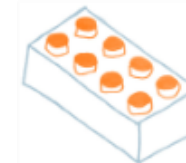
Formal Verification



Firmware/
Embedded
SW




IP/VIP
Design



FPGA
Design



- > Fabless semiconductor company, established in 2006
- > Headquartered in Israel
 - ~350 employees, with four subsidiaries worldwide (US, China, Japan and Germany)
- > Founder of the  HDBT™
- > Addressing two major markets:
 - Leader in the proAV market, with more than 90% adoption, thousands of HDBaseT-enabled products and millions of chips delivered
 - Qualified vendor to the automotive market for in-vehicle connectivity



Winner of a 67th Annual Technology & Engineering **Emmy** Award (2015) for the “*Development and Standardization of HDBaseT Connectivity Technology for Commercial and Residential HDMI/DVI Installation*”



Elchanan Rappaport – Formal Verification Tech Lead

- 30+ years Design / Verification experience
- 15 years Formal Verification experience
- Formal Services for Who's Who of Semiconductor Industry

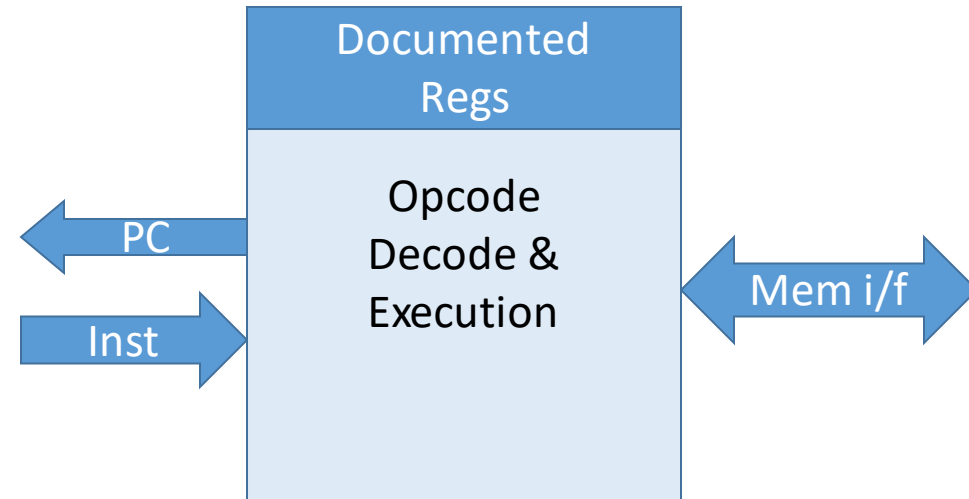


Clarke's 3rd Law: "Any sufficiently advanced technology is indistinguishable from magic."



Idealized RISC

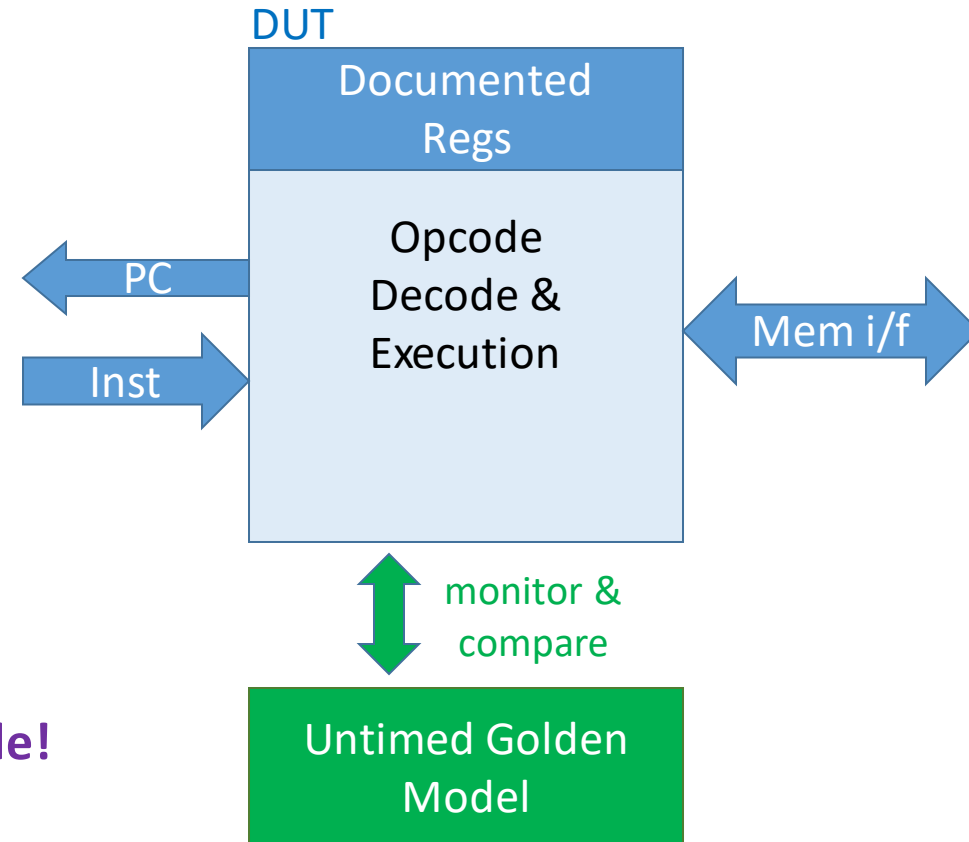
- All instructions execute in one cycle
- No internal regs other than documented
 - PC
 - R0 –Rn
 - Status
 - Interrupt Reg?
 - Timer?



Idealized RISC – Formal Verification

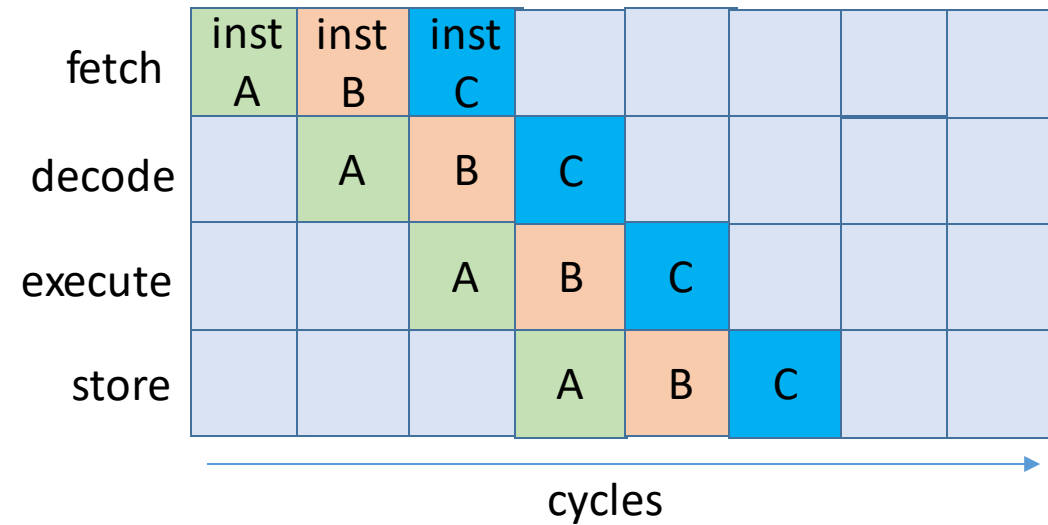
- 1) Remove resets from documented regs, start at arbitrary reg state
 - constrain status reg to legal?
- 2) Feed identical inputs to DUT and Golden Model
- 3) Clock once
- 4) Compare

Formal proof depth of ALL properties is just 1 cycle!
This is complete verification of DUT!



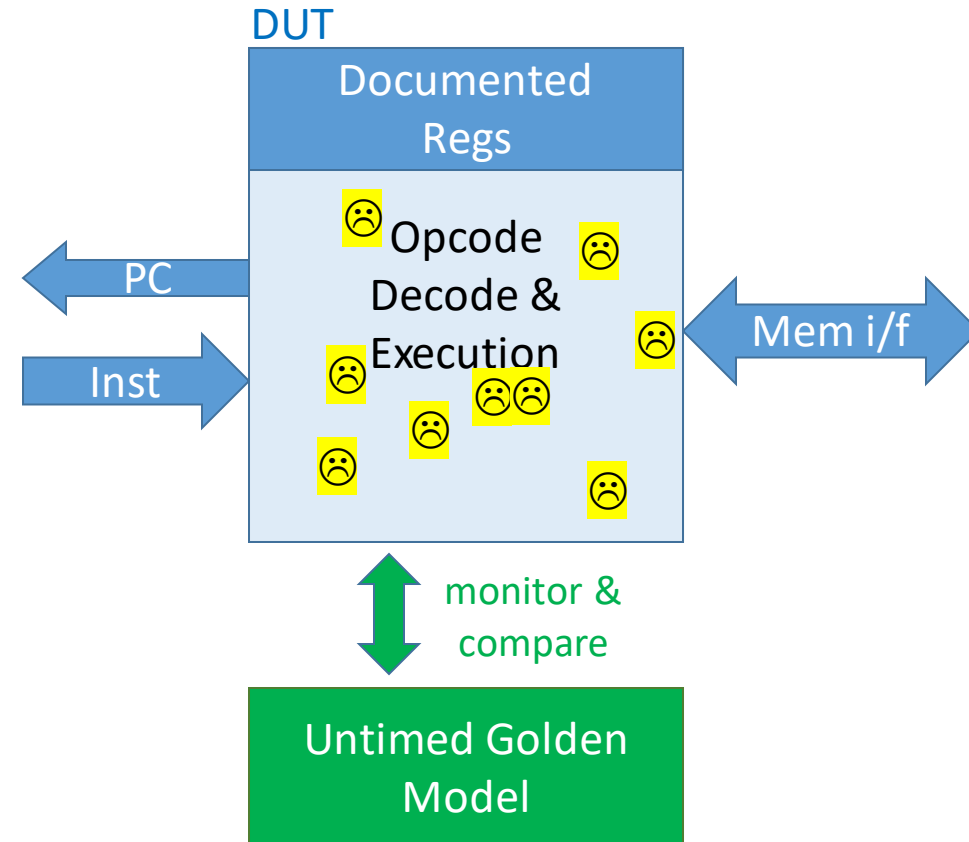
Multi-Phase Execution

- But in the real world, each instructions take multiple phases to execute. (e.g. fetch, decode, execute, store)
 - Bad for CPU performance
 - For us – not so bad.
 - We'd just make sure to always start on the first phase, and we'd have a maximum Formal proof depth of 4, which still isn't bad.
- Design solution: pipeline execution phases.
 - This creates additional undocumented internal registers to keep track of pipeline control and interim results.
 - (e.g. which inst is currently in which phase)



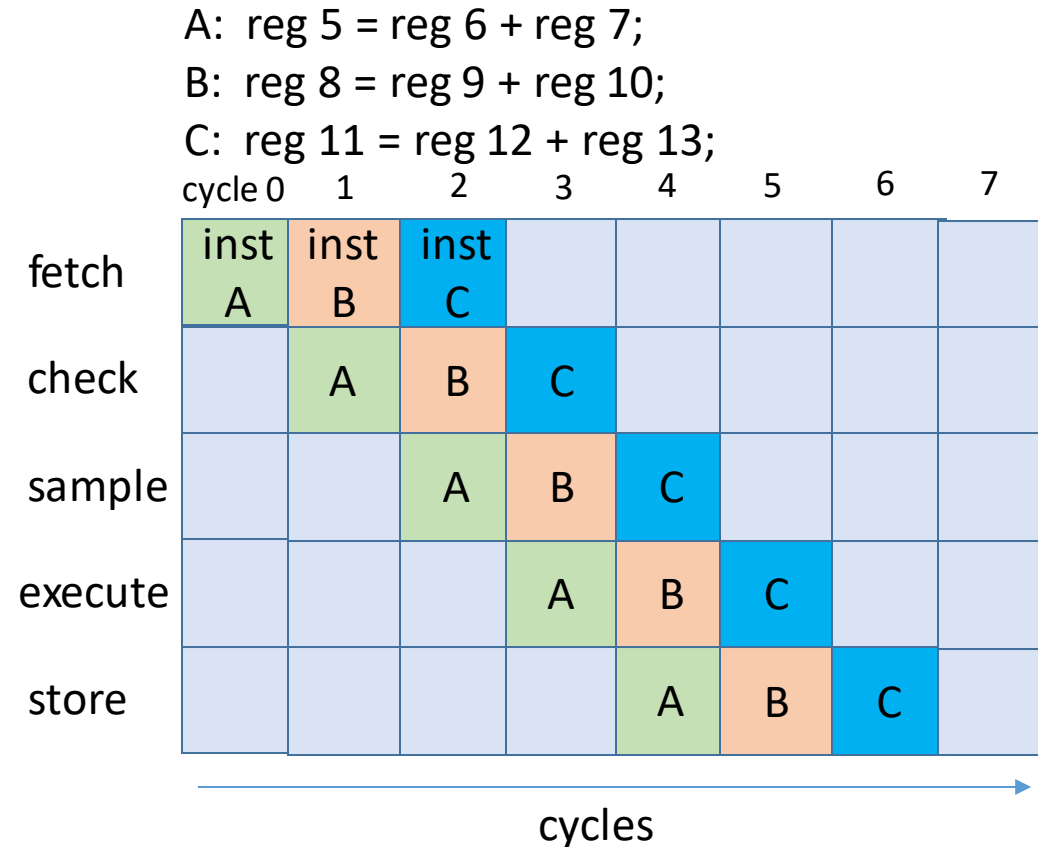
Pipelined Execution – internal regs

- Need additional internal registers to keep track of pipe control and interim results.
 - (e.g. “which inst is currently in which phase”)
- Can't start from arbitrary reg state
 - Would get inundated with false failures
 - need to ensure that start values for internal regs are legal and consistent.
- Options:
 - A) Work closely with designer to define / constrain legal internal reg states
 - **B) Always start from reset – longer traces** ☹️



Pipelined Execution - Phases

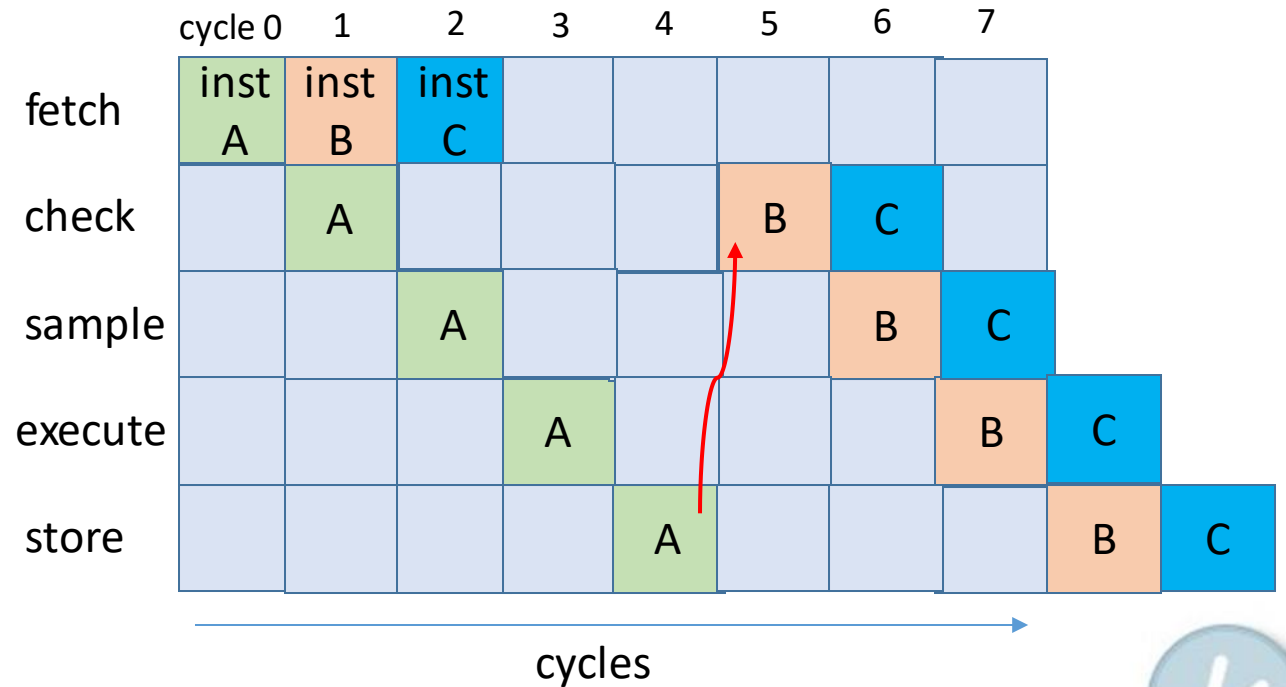
- Need to find a phase to key off of – i.e. for every executed instruction, to compare with untimed golden model on this phase.
- Typically, this will be the phase that stores back the results.
- For our uController, it's the “check” phase. When we've reached this phases, it means that all the predecessors are available.



Pipelined Execution – Stalled Pipe

- Inst B can't enter check phase until the new value of reg 5 has been calculated.
- This only happens after inst A is done with its store phase.
- Area ripe for control bugs.

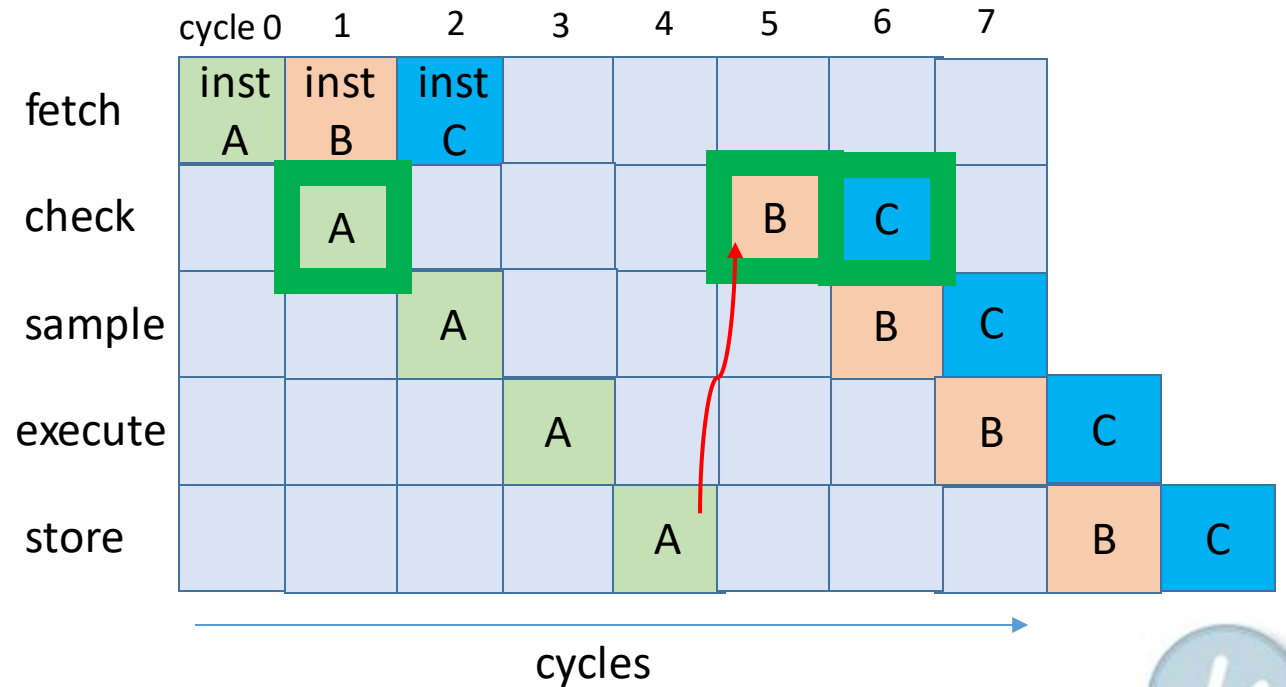
A: **reg 5** = reg 6 + reg 7;
B: reg 8 = **reg 5** + reg 10;
C: reg 11 = reg 12 + reg 13;



When to compare with Golden Model?

- We compare with untimed golden model on the check phase.

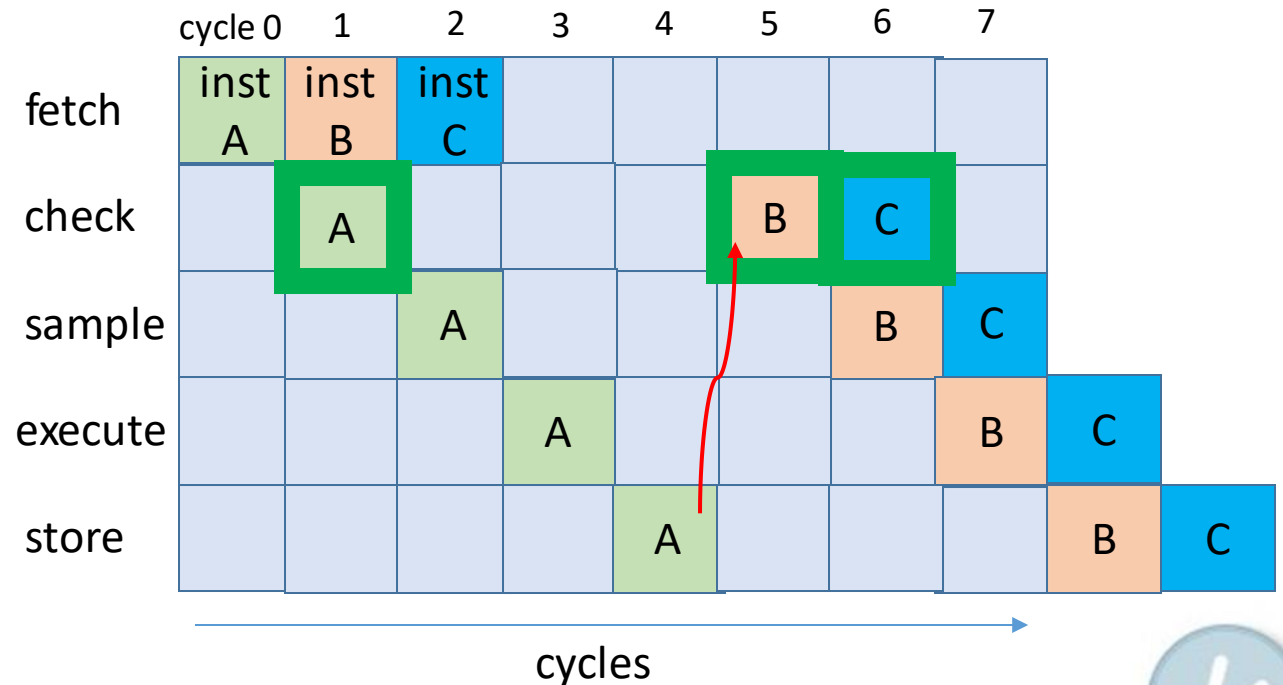
A: **reg 5** = reg 6 + reg 7;
B: reg 8 = **reg 5** + reg 10;
C: reg 11 = reg 12 + reg 13;



What to compare with Golden Model?

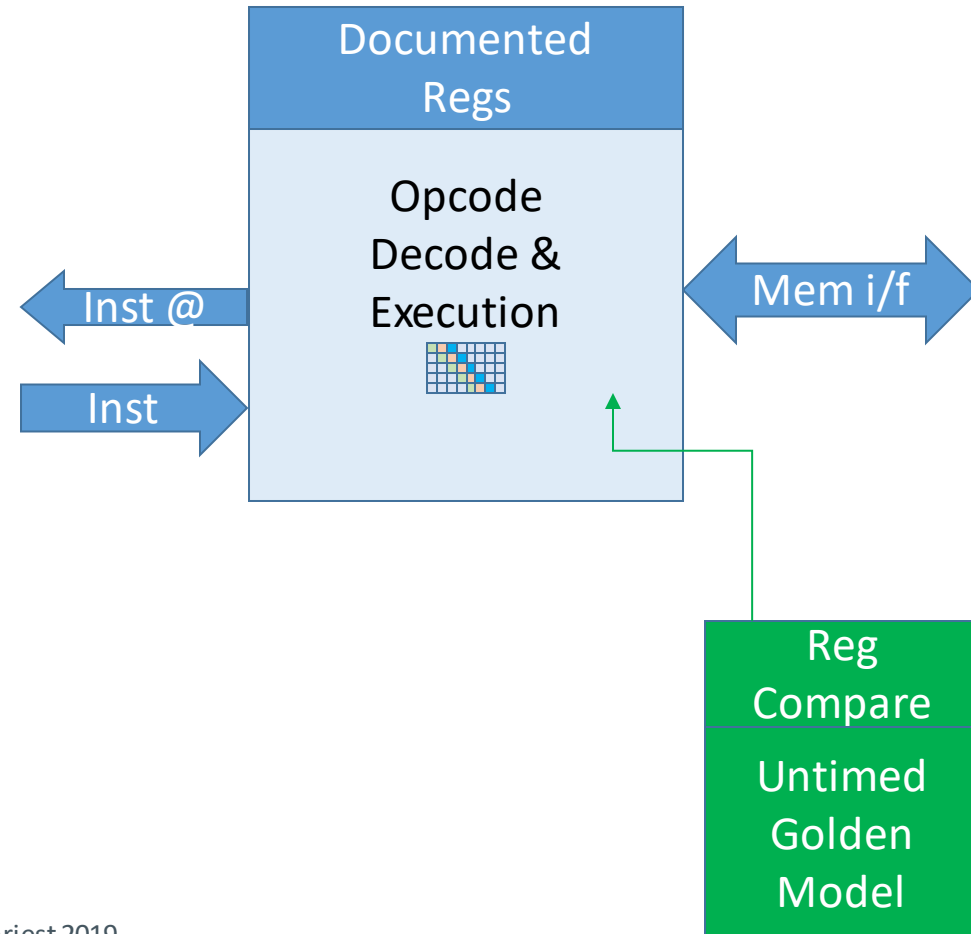
- Comparing ALL documented regs won't work.
 - e.g. at cycle 6, our untimed golden model has already updated reg 8 (the output of inst B), but the DUT is still 3 cycles away from doing that.
- Only compare those regs which are predecessors to **this** instruction.
 - e.g. on cycle 6, for inst C, we only compare regs 12 and 13.
- We don't need to worry about comparing the other regs now. Any errors will be picked up by the next instruction which uses the bad reg as a predecessor.

A: **reg 5** = reg 6 + reg 7;
B: reg 8 = **reg 5** + reg 10;
C: reg 11 = reg 12 + reg 13;



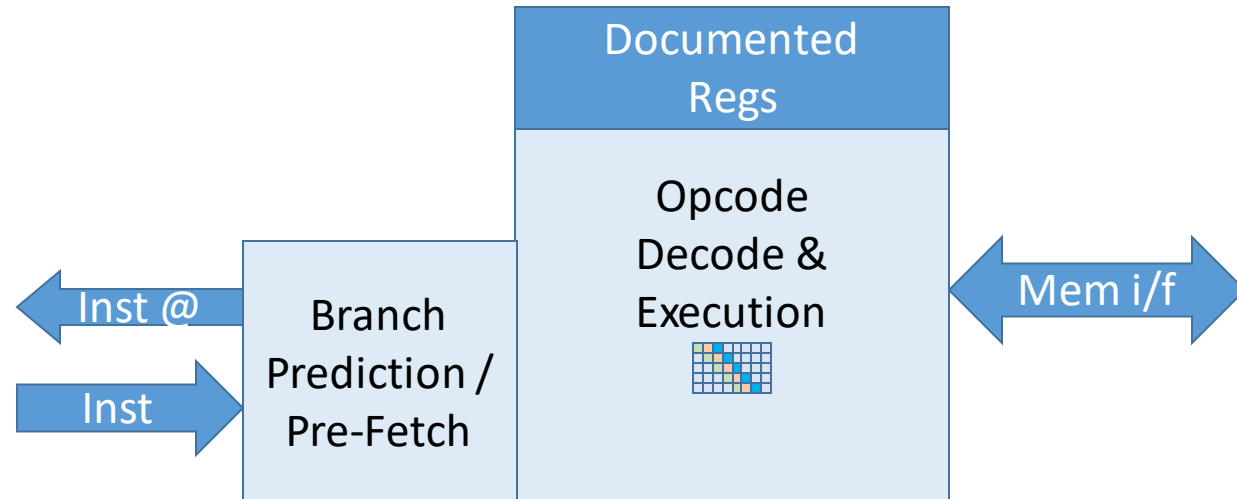
Predecessor Compare

- On “check” phase, compare inst predecessors with untimed golden model.



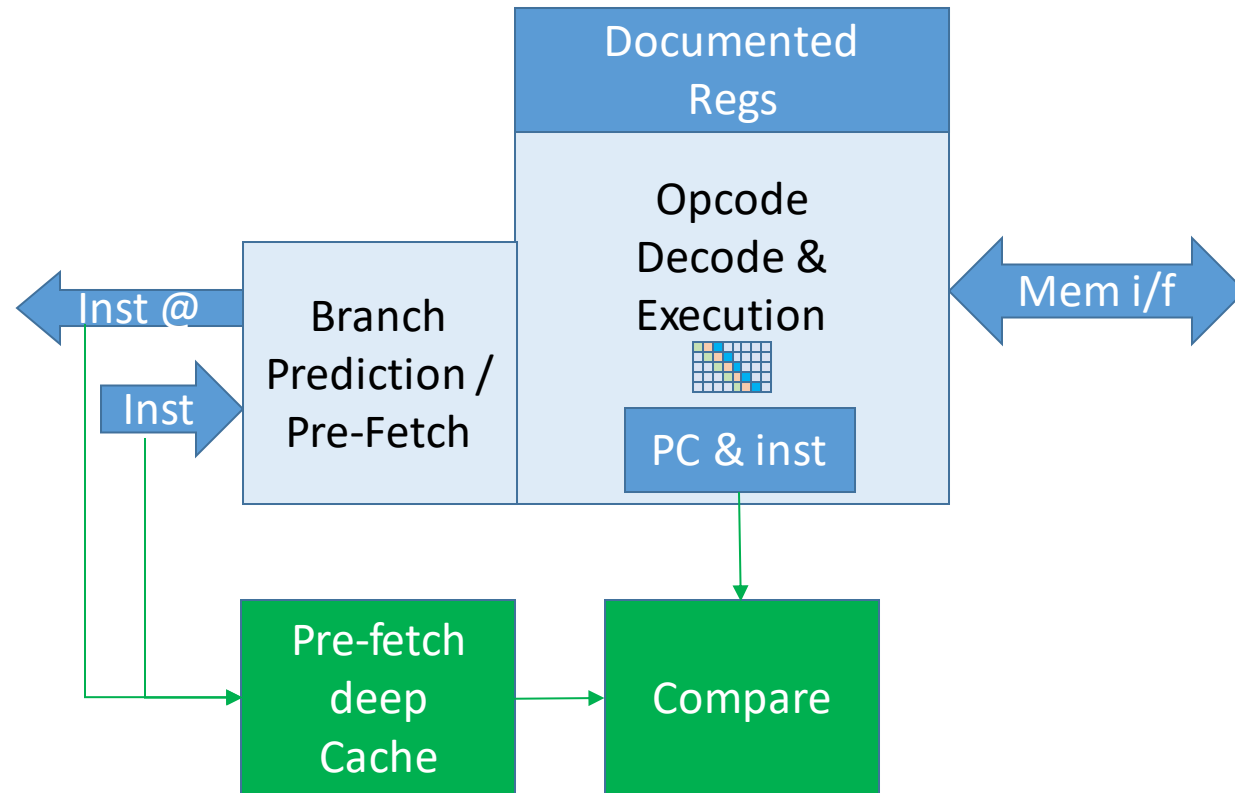
Pre-Fetch

- In order to reduce waiting time from instruction memory, pre-fetch is implemented in the design.
- We also need to test that the pre-fetch block operates correctly.



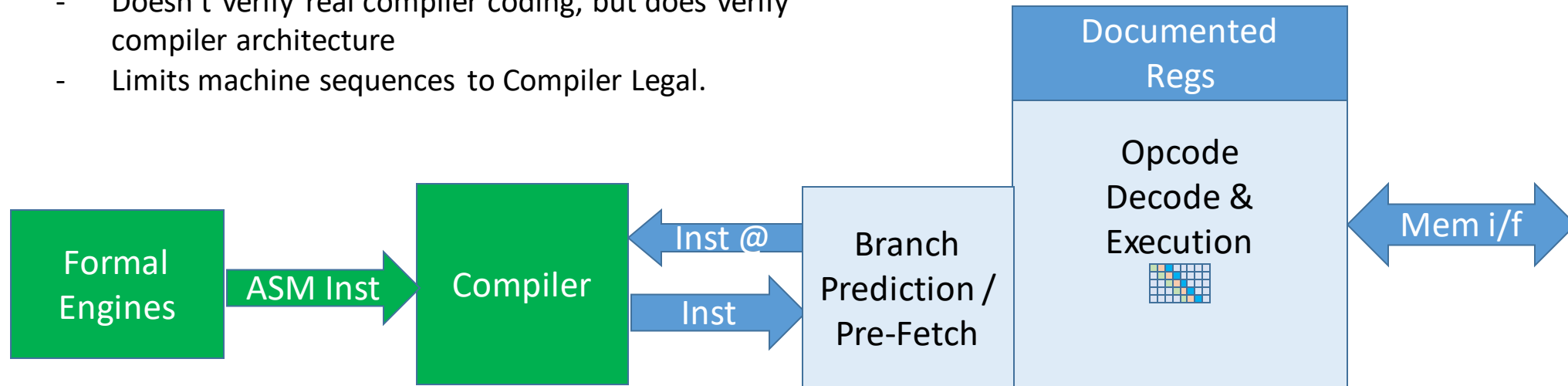
Pre-Fetch - Formal

- Save instruction address, and instruction in pre-fetch-deep [throw out oldest] cache.
- Compare to PC and instruction reg at check phase for this instruction.
- Can't use regular scoreboard because some instructions are fetched but never used. (incorrect branch prediction)



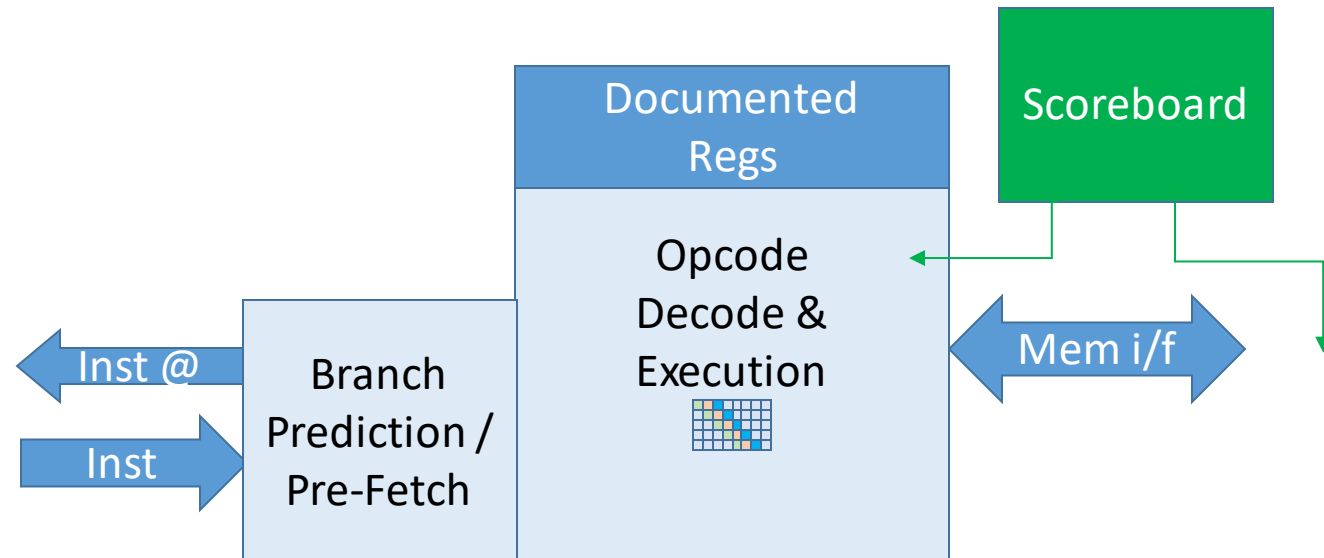
Compiler in Verilog

- Implement “Compiler” in Verilog as part of test environment
 - Doesn't verify real compiler coding, but does verify compiler architecture
 - Limits machine sequences to Compiler Legal.

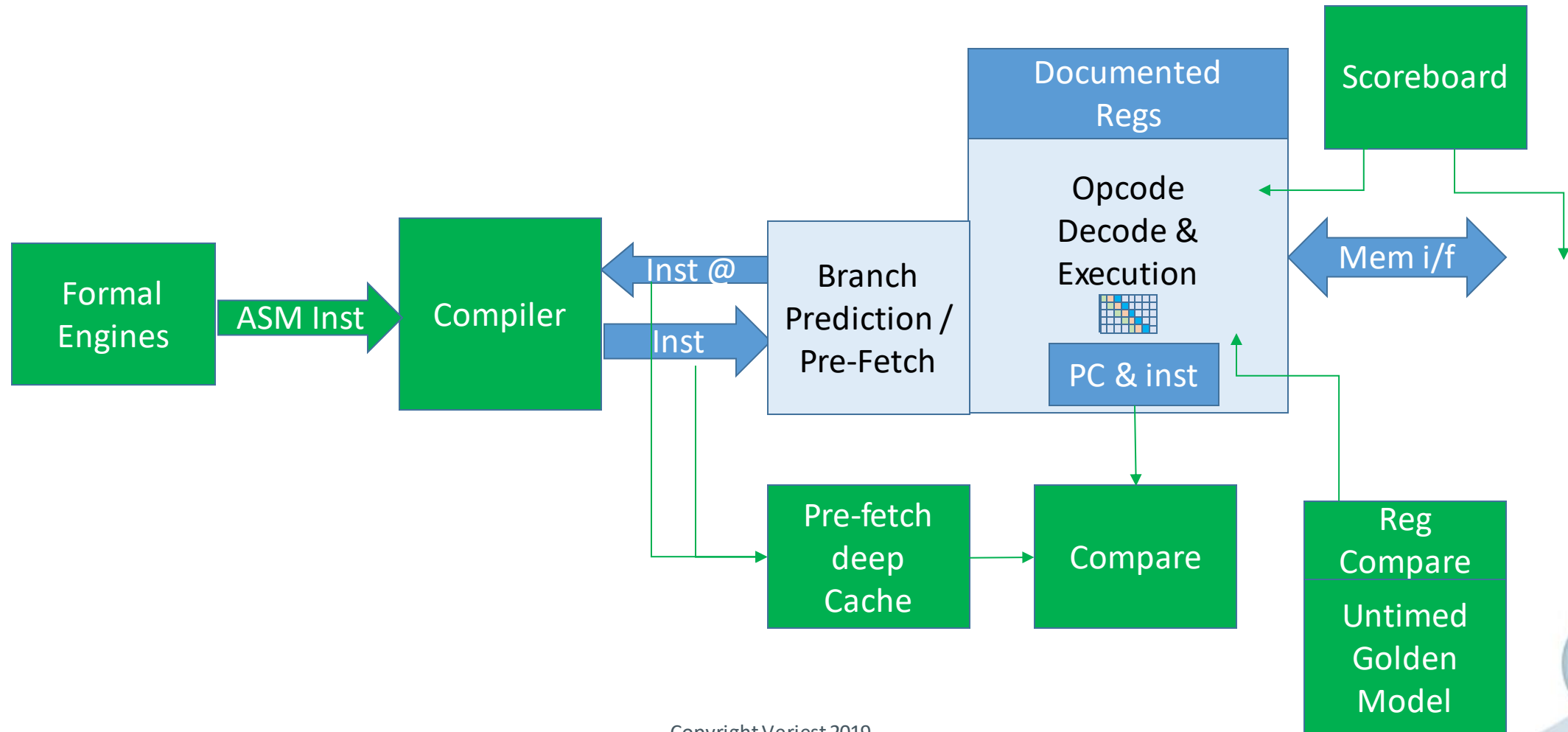


Memory access

- Scoreboard memory access instructions



Full Environment



Pre-Fetch Compare

```
// Store in the table
always @(posedge clk or negedge reset_n)
  if (!reset_n)
    begin
      wptr <= 0;
      for (int loop=0; loop<IFETCH_TABLE_SIZE; loop++)
        ifetch_table[loop] <= 0;
    end
  else if (opcode_ack_s1)
    // This is the cycle when opcode_din is valid.
    begin
      ifetch_table[wptr] <= {1'b1, opcode_add_s1, vl_uCtrl.opcode_din};
      wptr <= wptr + 1;
    end

// now check that we don't execute anything we didn't fetch
// We're no longer on the opcode interface.
// Now we're inside the pipe, at the chk_vld phase
logic found_opcode_in_table;
always @(*)
  begin
    found_opcode_in_table = 0;
    for (int loop=0; (loop < IFETCH_TABLE_SIZE) && (found_opcode_in_table==0); loop++)
      // 81 bits - this checks both the address and the opcode line
      if (ifetch_table[loop] == {1'b1, vl_uCtrl.vl_uCtrl_base.chk_data})
        found_opcode_in_table = 1;
  end

// Has the opcode (with address) that we're seeing been fetched?
AST_opcode_fetched: assert property (@(posedge clk) vl_uCtrl.vl_uCtrl_base.chk_vld |-> found_opcode_in_table);
```



Identify Predecessors

```
always @(*)
begin
  // defaults
  sreg_pred = 0;

  case (chk.opc)
  3: // ADDSUB
    case (chk.indirect)
      0, 2, 4, 6: sreg_pred = 1;
      default: ;
    endcase
  4: // MULT
    case (chk.indirect)
      0, 2: sreg_pred = 1;
      default: ;
    endcase
  5: // Logic Operation
    case (chk.indirect)
      0, 1, 2, 3: sreg_pred = 1;
      default: ;
    endcase
  7: // Shift and Rotate
    case (chk.indirect)
      8, 9, 10, 11, 13, 15: sreg_pred = 1;
      default: ;
    endcase
  default: ;
  endcase // case (chk.op)
end // always @ (*)
```



Compare Predecessors

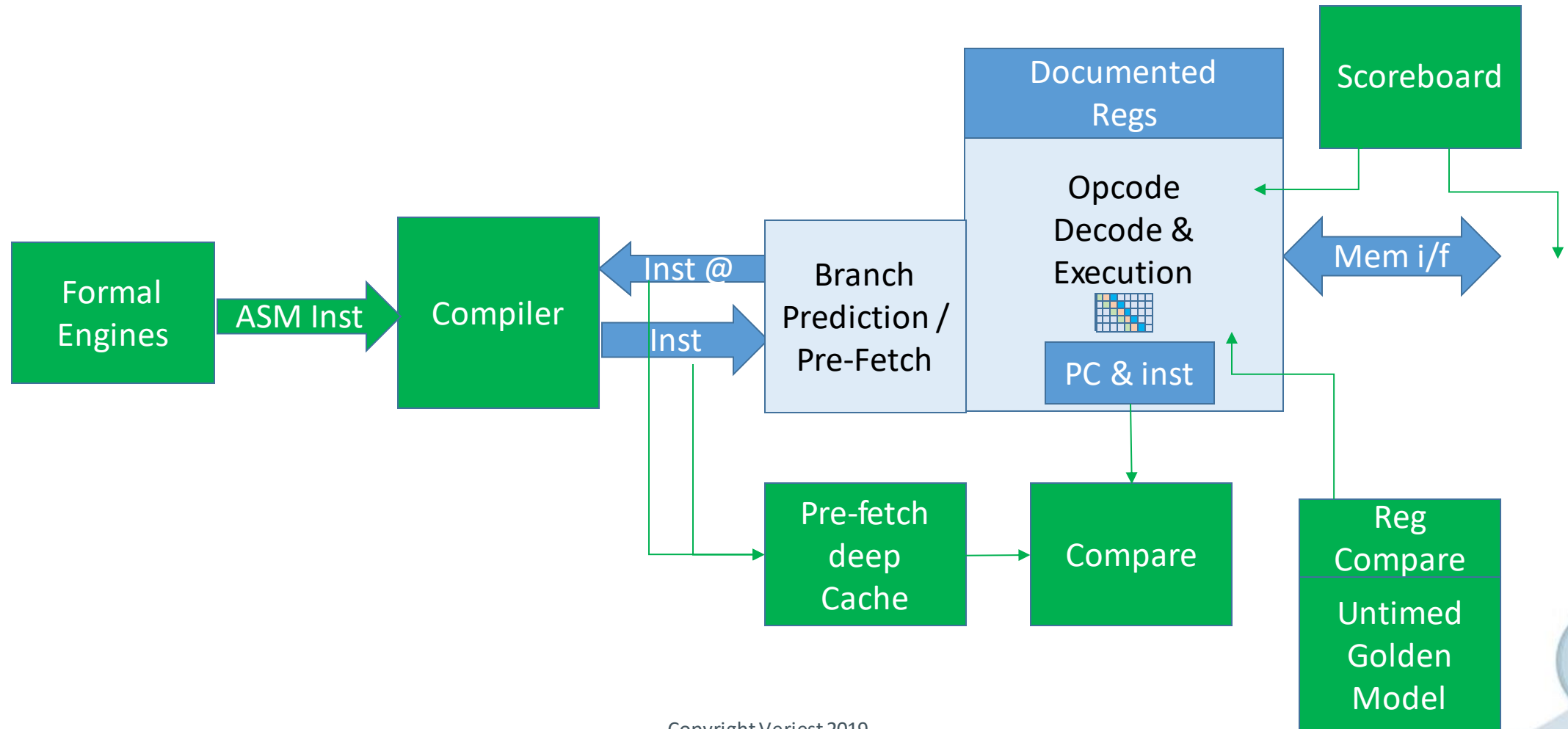
```
// check Program Counter - for every opcode
AST_pc: assert property (@(posedge clk)
    vl_uCtrl.vl_uCtrl_base.chk_vld |-> vl_uCtrl.vl_uCtrl_base.chk_data[79:64] == g_pc);

// Check DREG
AST_dreg_pred: assert property (@(posedge clk)
    vl_uCtrl.vl_uCtrl_base.chk_vld && dreg_pred |-> chk_dreg == g_reg_array[chk.dreg_index]);

// Check SREG
AST_sreg_pred: assert property (@(posedge clk)
    vl_uCtrl.vl_uCtrl_base.chk_vld && sreg_pred |-> chk_sreg == g_reg_array[chk.sreg_index]);
```



Full Environment



Results

- Tested 85 out of 94 Assembler Instructions (ongoing)

Bug Type	#
Control Timing	1
Signed Math	1
Interface	2
Decode error	4



Conclusions

- Verification would be easier if designers just created simple designs 😊
- Verification can be optimized if designers are willing to work closely with the Formal team.
- Intelligent reduction of sequence depth is a goal.
- **Need to identify when and what to compare to the golden model.**
- Opcode testing lends itself to a very structured test environment.
- It's not just testing opcodes!
- **uController designs DO lend themselves to Formal.**



Veriest

Thank You!

www.VeriestS.com



V

solutions