Formal verification of RTL Systolic Arrays for GEMM

Pedro Palacios Almendros Adrian Schemel

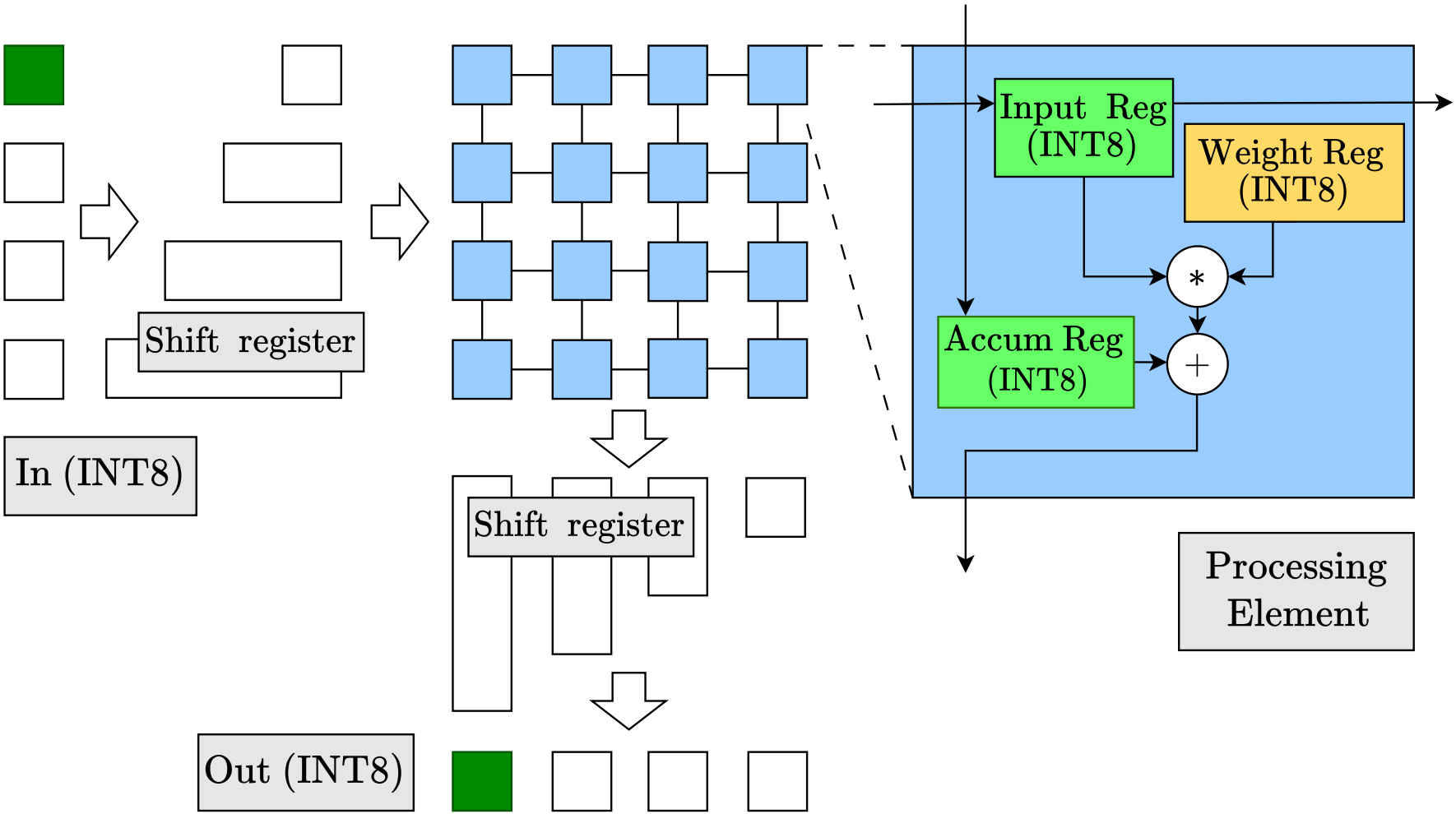
École Polytechnique Fédérale de Lausanne

2024-2025

Project goals

- **Verify** that a fast ($O(n)$ time), parallel ($O(n^2)$ processing elements), systolic array **RTL hardware** implementation matches the standard triple-loop $O(n^3)$ mathematical definition.
- Explore the **potential** and behavior of industry RTL hardware **formal verification tools**.
- Explore the **trade-off** between verification capabilities and expressiveness of hardware **interfaces**.
- Explore the **impact** of carefully crafted **assertions** on verification **speed**.
- Perform **Design Space Exploration** to analyze the verification performance impact of different trade-offs.

Systolic Arrays



Temporal Logic

- **Temporal logic** extends boolean logic with primitives that express how properties evolve **over time**.
- Specific timings are abstracted away into **time-steps** or clock cycles.
- The fundamental primitive is $next\, A$, which means that A will be true on the **next time-step**.
- Henceforth operator $\Box A$: Formula A is true now and will be true forever. Let $next^0 A = A$ and $next^{n+1} A = next\, next^n A$, then the temporal logic operator $\Box A = \bigwedge_{i=0}^{\infty} next^i\, A$.
- Used to **specify** and **verify hardware** systems, as done in our background paper[1].

[1] Bochmann. "Hardware Specification with Temporal Logic: An Example". In: *IEEE Transactions on Computers* C-31.3 (1982), pp. 223–231.

System Verilog Assertions

- Part of the System Verilog HDL to express temporal properties.
- The $next$ operator over clock cycles can be expressed through delay `##N` and the function `$past`.
- `s1 |-> s2`: whenever sequence s_1 matches, s_2 must match too.
- `assert property` have an implied $\Box$ operator.

```
assert property (
  ##1 A
);
```

$$\equiv \quad \Box\,(\,next\,A)$$

```
assert property (
  A ##1 B ##1 C
);
```

$$\equiv \quad \Box\,(\,A \wedge next\,(B \wedge next\,C))$$

```
assert property (
  A |-> B ##1 C
);
```

$$\equiv \quad \Box\,\,(A \rightarrow (B \wedge next\,C))$$

Liveness properties

- **Liveness** properties assert that a property will **hold in some cycle**, but specify **no bound**.
- Temporal logic operator $\nabla A = \bigvee_{i=0}^{\infty} next^i\, A$.
- In SVA, it's modelled by `s_eventually`.
- Example modelling a liveness property $Request \implies \nabla Response$:

```
assert property (
  request_signal |-> s_eventually(response_signal)
);
```

Bounded Model Checking & k-Induction

- Two different verification approaches:
  - **Bounded Model Checking**: Check sequences up to fixed length j.
  - **k-Induction**: Check sequences of any length.
- Bounded Model Checking (BMC) as seen in class:

Construct formula T_j, choosing j to be a relevant bound (in our case $2 \cdot N + n_{input\_rows}$, where N is the systolic array size), and prove UNSAT.

$$T_j \equiv Init[\overline{s} := \overline{s}^0] \wedge \left(\bigwedge_{i=0}^{j-1} R_i \right) \wedge E[\overline{s} := \overline{s}^j]$$

The formal verification tool constructs $Init$ from the assumptions and reset hardware register values, R from the RTL hardware description and E from the System Verilog Assertions.

k-Induction

- Prove that a property $P(n)$ holds $\forall n$ (**for any sequence length**).
- Middle-ground between strong induction and normal induction.
- Check $P(0) \wedge ... \wedge P(k-1)$ and $P(n-k) \wedge ... \wedge P(n-1) \implies P(n)$
- $P(n)$ contains the SVA assertions at cycle n, and the formal verification tool also adds the hardware-defined relationship between the signals in the conjunction $P(i) \wedge P(i+1)$.
- Both the **base** and **inductive** case are converted to **SAT** problems.
- **Carefully choose k**: For a systolic array size N, need $k \geq 2 \cdot (N+1)$ to relate the output of the systolic array with the relevant inputs.

Challenges

- Due to **not having a license of JasperGold**, we used an evaluation license of SymbiYosys, which lacks some SVA features (such as sequences with local variable capturing).
- The formal verification tools required **very precise configuration** (such as clock and reset domains), which took a **long time to debug** through trial and error.
- It was impossible to determine a priori the correct hardware interface for the systolic array, **requiring several iterations**.
- We needed to **learn System Verilog Assertions**, and how to translate the correctness properties to SVA, reconciling the mathematical Temporal Logic description with the actual hardware implementation.

Golden reference model

```
function logic golden_model_matrix_vector_multiply_check (
    input logic[INT_WIDTH-1:0] input_vector[SA_SIZE],
    input logic[INT_WIDTH-1:0] actual_sa_output[SA_SIZE]
);
    logic[INT_WIDTH-1:0] expected[SA_SIZE];
    logic does_match = 1'b1;
    // First compute expected result
    for (int i = 0; i < SA_SIZE; i++) begin
        expected[i] = '0;
        // Compute dot product
        for (int j = 0; j < SA_SIZE; j++) begin
            expected[i] += input_vector[j] * weights[j][i];
        end
    end
    // Now compare expected with actual
    for (int i = 0; i < SA_SIZE; i++) begin
        if (expected[i] != actual_sa_output[i]) begin
            does_match = 1'b0;
        end
    end
    return does_match;
endfunction
```

Interfaces

- The **module interface** with the outside world is **essential for verification**.
- **Impacts** the **definition of correctness**, **speed** of verification and even the **provability** using k-Induction.
- We have implemented and verified **three** different **interfaces**:
  1. Preloaded weights, one input per cycle
  2. Preloaded weights, stallable (maybe multiple cycles per input)
  3. Fully programmable: weight-load and input-load commands

Interface 1: Preloaded weights, one input per cycle

- The preloaded weights are modeled as `(* anyconst *)`, so Yosys checks every possible weight assignment.
- As an input is loaded every cycle, we can precisely compute the delay from input to output as $2 \cdot \texttt{SA\_SIZE}$:

```
assert property (!output_valid |-> ##(2*SA_SIZE) output_valid);
```

- Therefore, the correctness check is simple:

```
assert property (
    output_valid |-> golden_model_mv_mult_check(
        $past(inputs, 2*SA_SIZE), out
    ) == 1'b1
);
```

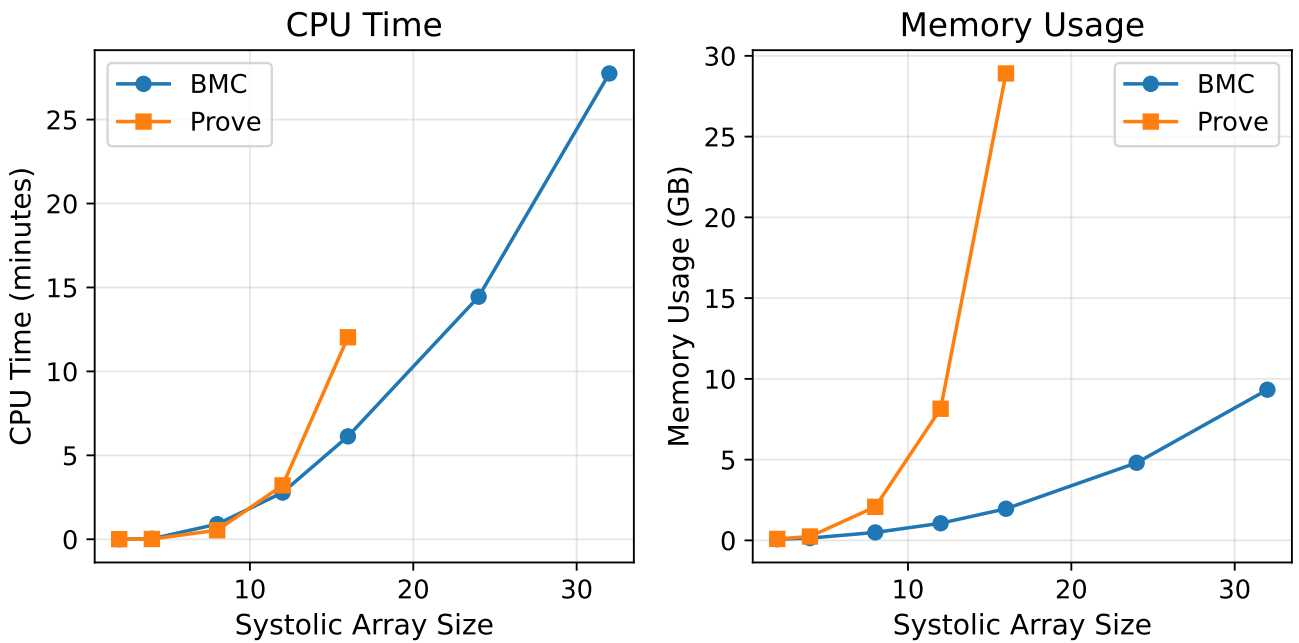
- We can also use `s_eventually` for liveness (slower):

```
assert property (s_eventually(output_valid));
```

Interface 1: Performance metrics

- Verified BMC up to 32×32 (28min, 10GB), prove up to 16×16 (12 min, 29 GB). **Memory usage explosion for prove!**
- Liveness (`s_eventually`) only verified up to 2×2. **Too slow!**
- Ran Yosys auto-benchmark to choose best SAT solver.

BMC vs Prove Performance Comparison for Interface 1



Interface 2: Preloaded weights, stallable

- The weights are still modeled as `(* anyconst *)`.
- We no longer have a fixed delay, need to add a counter for verification, and store input snapshots (`MAX_NUM_INPUTS_ROWS`):

```
assert property (
    should_advance_computation &&
    should_advance_counter == 2*SA_SIZE-1 |=> output_valid
);
```

- Correctness check much more complex (need counter):

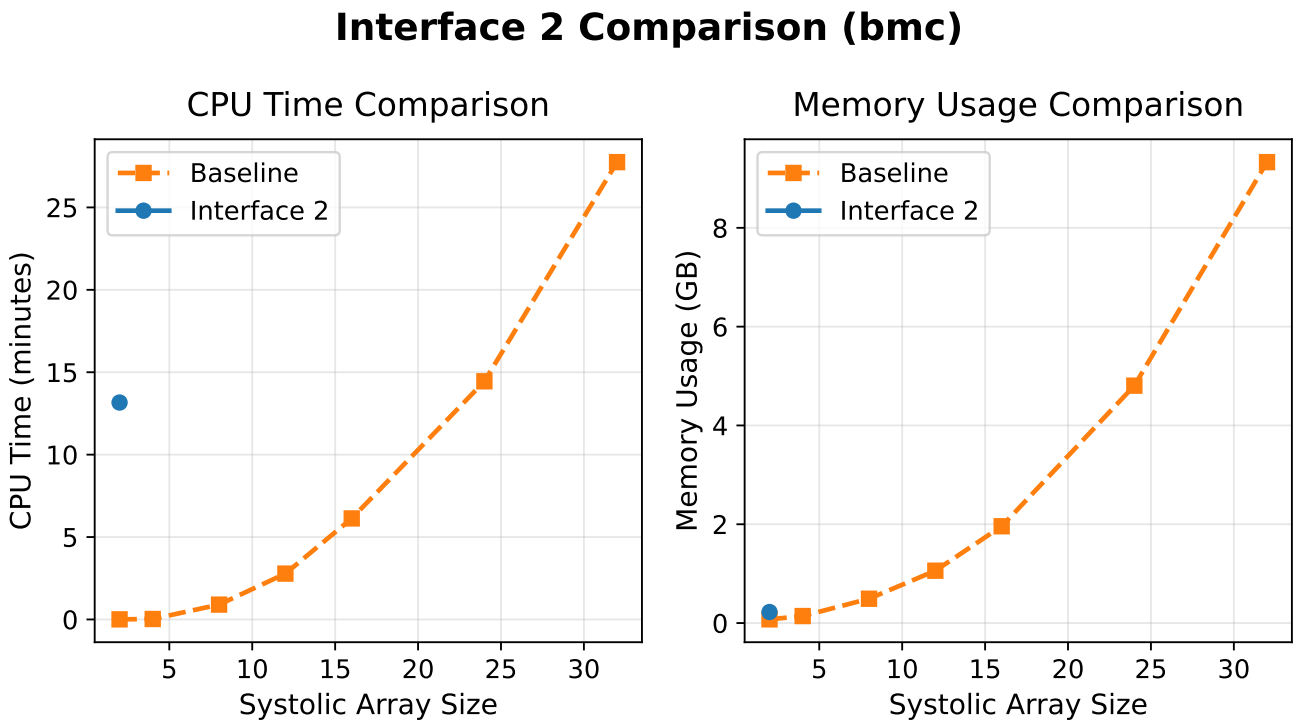
```
assert property (output_valid &&
   (should_advance_counter < 2*SA_SIZE + MAX_INPUTS)
       |-> (should_advance_counter >= 2*SA_SIZE)
           && golden_model_matrix_vector_multiply_check(
                should_advance_counter - 2*SA_SIZE,
                out ) == 1'b1);
```

- Liveness stays the same (very slow):

```
assert property (s_eventually(output_valid));
```

Interface 2: Performance summary

- Cannot use k-Induction (arbitrary delay). **Only BMC**.
- Could only verify BMC up to 2×2 (13min, 0.22 GB). 4×4 aborted after > 2 hours. **Very slow:** 2×2 takes the same as 24×24 for interface 1.
- Liveness managed to verify up to 12×12 (2 hours, 15.57GB)



Interface 3: Fully programmable

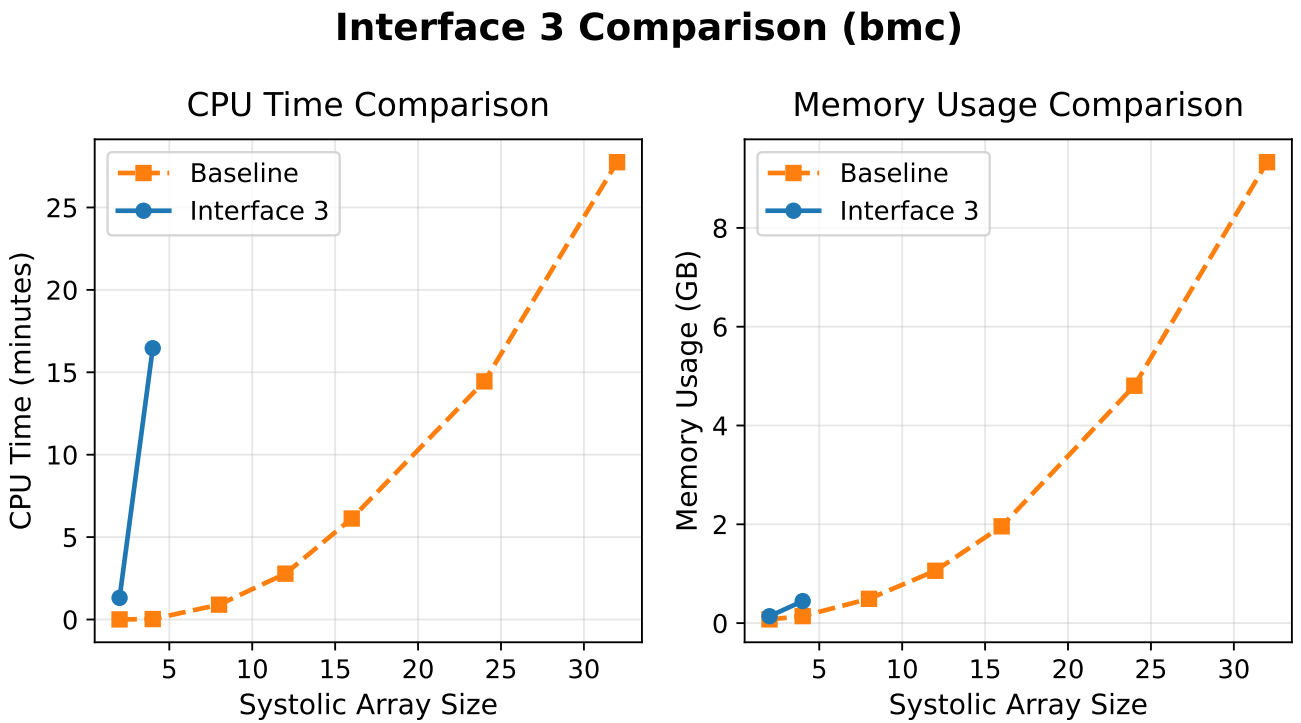
- The module supports Weight-Load and Input-Load commands.
- As we saw in Interface 2 that arbitrary delays lead to terrible results, **drive the module** with a testbench **state machine**, that only initially loads the weights and loads inputs each cycle.
- Correctness check can use state-machine values:

```
logic match;
always_comb begin
    match = 1'b1;
    for (int i = 0; i < SA_SIZE; i++) begin
        match = match && (activation_outputs[i]
            == reference_outputs[output_row_idx][i]);
    end
end

assert property (
    (output_valid && output_row_idx < INPUT_SIZE)
        |-> match
);
```

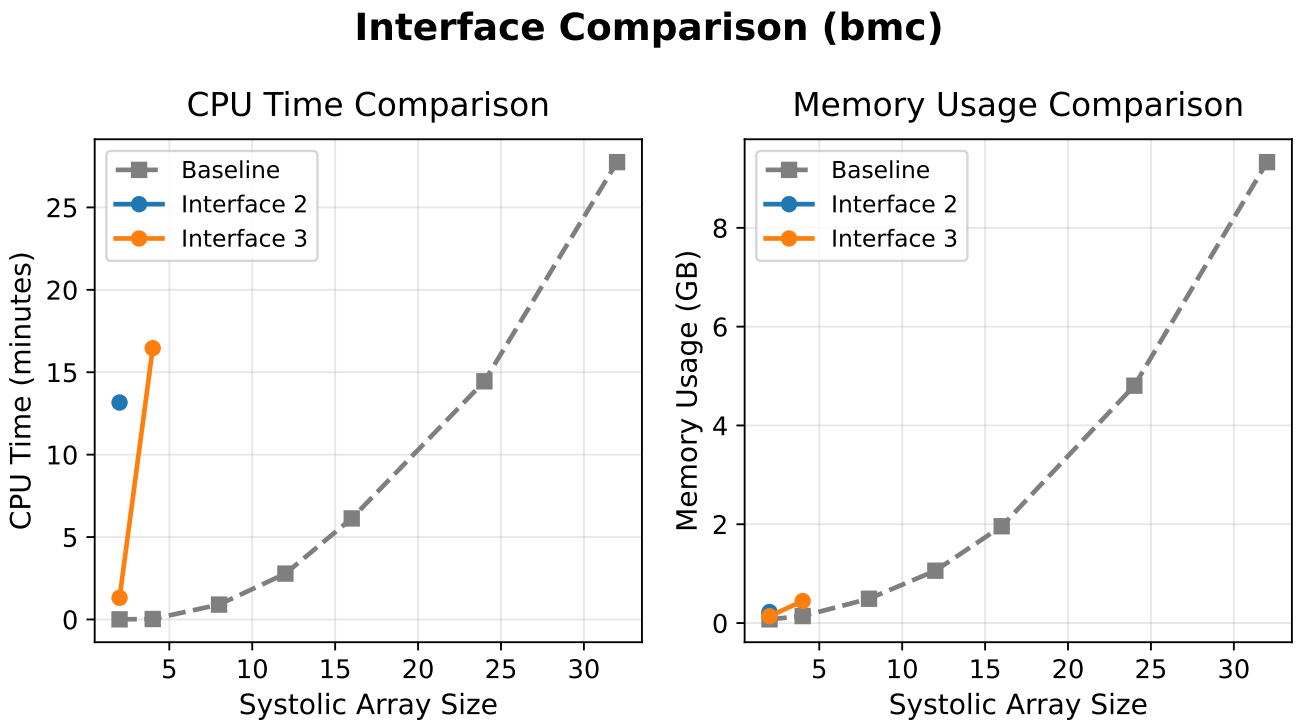
Interface 3: Performance summary

- Even though it should be possible in theory (fixed testbench state machine), Yosys is unable to use k-induction. **Only BMC**.
- Could only verify BMC up to 4×4 (16min, 0.45 GB). **Very slow.**
- Liveness managed to verify up to 4×4 (49min, 5 GB).



Interfaces summary

- Interface choice **matters a lot**.
- Yosys strongly prefers one-input-per-cycle schemes for dataflow.
- `s_eventually` liveness is very slow, try to instead find fixed bounds.
- Even a simple input scheme (one-input-per-cycle testbench state machine) is slow if the underlying interface is complex.



Custom assertions to accelerate verification

- We saw that verification is **slow**.
- Idea: add **additional assertions** or *invariants* that may help Yosys and the SAT solver **prove** the formulas **faster**.
- Describe intuitive and **mathematical properties** of the systolic array using System Verilog Assertions.
- Focus on interface 1 (the fastest) and k-induction proofs.

Assertion 1: Sub-vector products

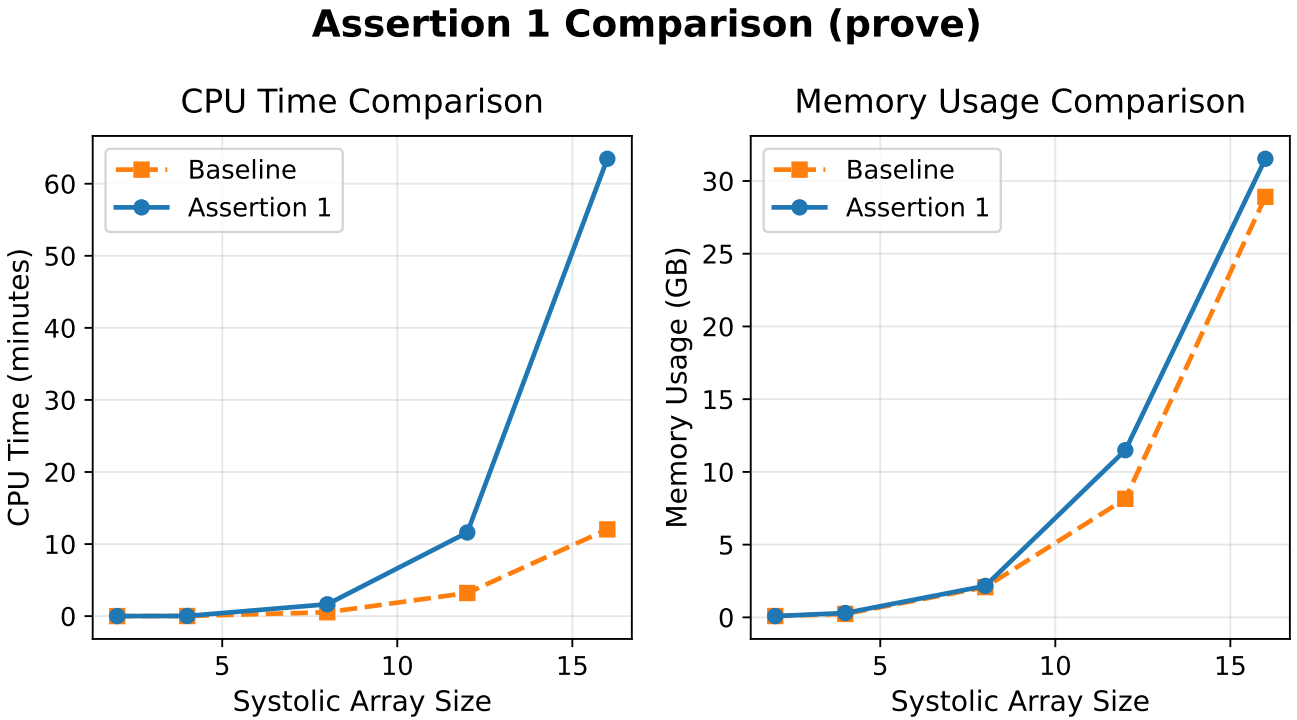
- Intuition: at each time step, the output of each PE is a product of 2 sub-vectors of the input and weight matrices.
- With $\langle u, v \rangle$ being the vector dot product, and using colon notation:

$$\forall i, j \in \{1, \ldots, n\}. \square \left(\langle I(1\!:\!i), W(1\!:\!i, j) \rangle = \mathbf{next}^{i+j-1} \, Out(P_{i,j}) \right)$$

```
generate
  for (genvar i = 0; i < SA_SIZE - 1; i++) begin
    for (genvar j = 0; j < SA_SIZE; j++) begin
      assert property (
        ##(i+j+2) u_GEMM.u_SA.pe_outs[i][j] ==
        compute_sub_element(i,j,$past(inputs,i+j+1),weights)
      );
    end
  end
endgenerate
```

Assertion 1: Performance summary

- Unfortunately, this assertion makes the proof **slower** and consume **more memory**.



Assertion 2: Input propagation

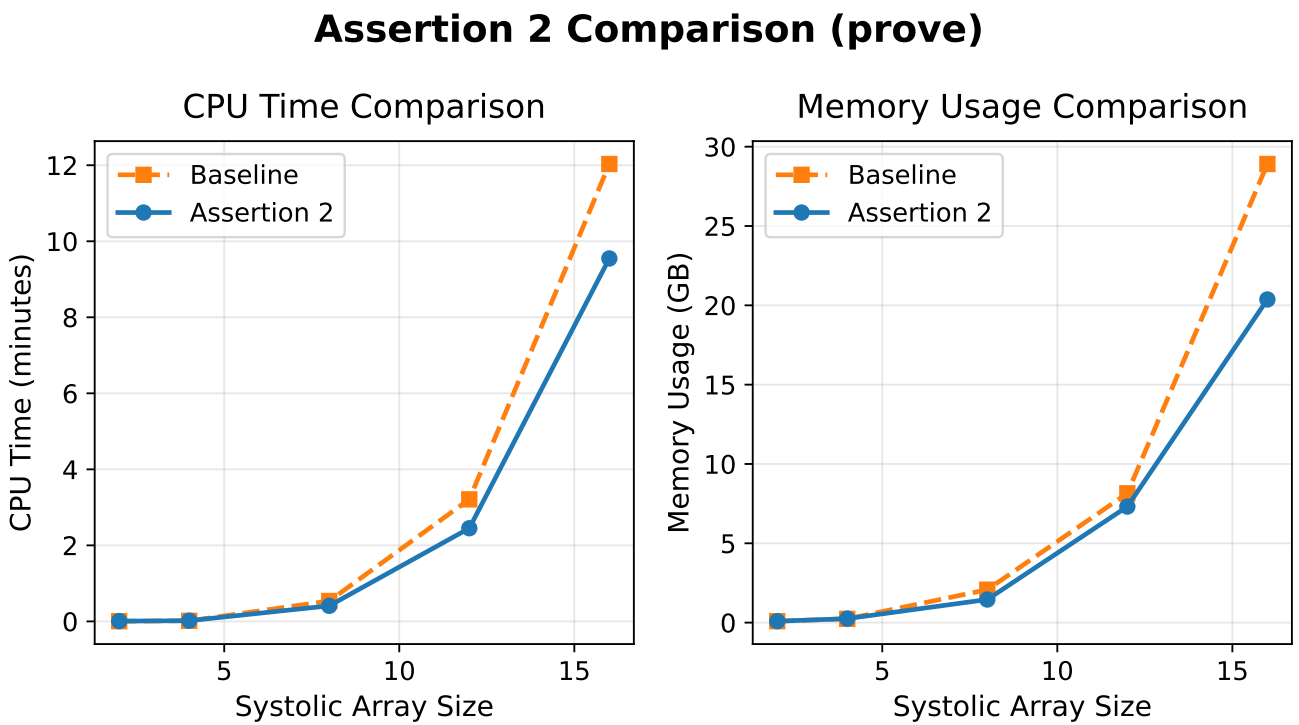
- Intuition: at each time step, the input of each PE is traceable back to a past input vector.
- Mathematically:

$$\forall i, j \in \{1, \ldots, n\},\ \square \left(I_i = \mathbf{next}^{i+j-1}\ In_*(P_{i,j}) \right)$$

```
generate
    for (genvar i = 0; i < SA_SIZE; i++) begin
        for (genvar j = 0; j < SA_SIZE; j++) begin
            assert property (
                ##(i+j+2) u_GEMM.u_SA.pe_ins[i][j]
                   == $past(inputs[i],i+j+1)
            );
        end
    end
endgenerate
```

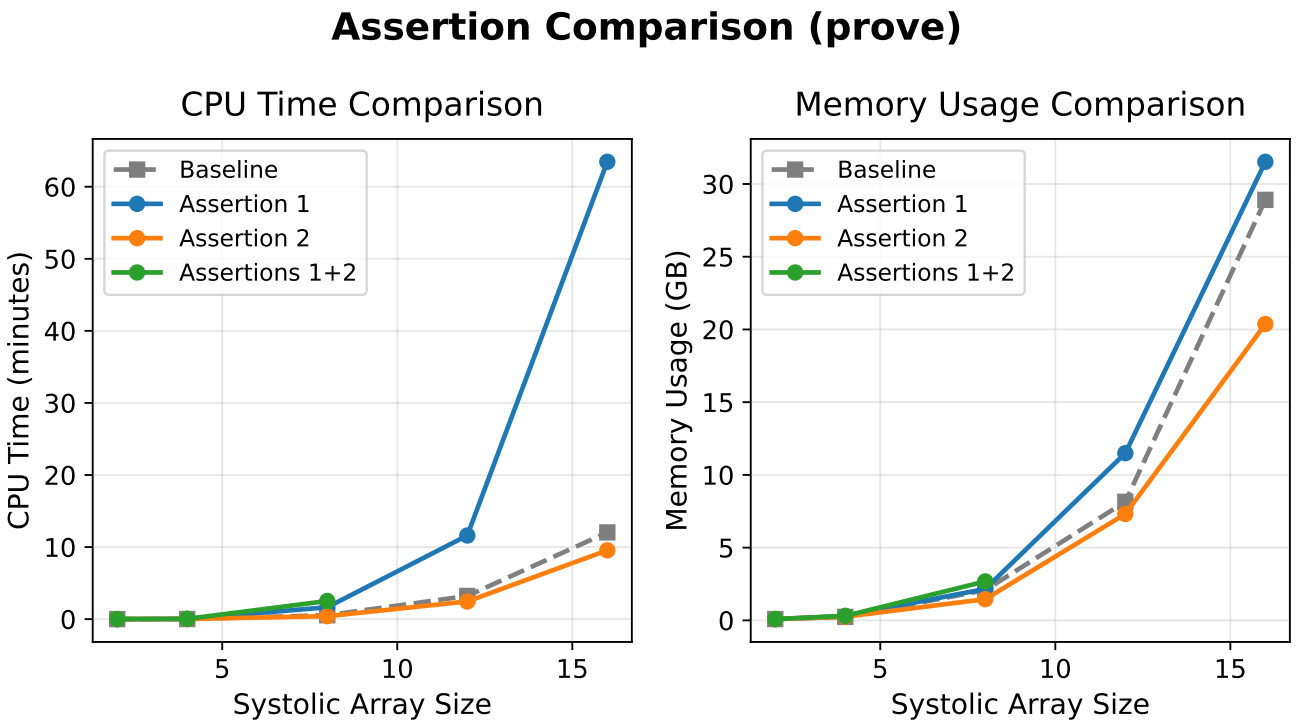
Assertion 2: Performance summary

- This assertion does make the proof **faster** and consume **less memory**.



Assertions summary

- Carefully crafted assertions may lead to **faster verification**, but **it's not a guarantee**.
- We also played with other assertion ideas, with even worse results. It's hard to predict what will help the SAT solver and what will hinder it by adding more work.



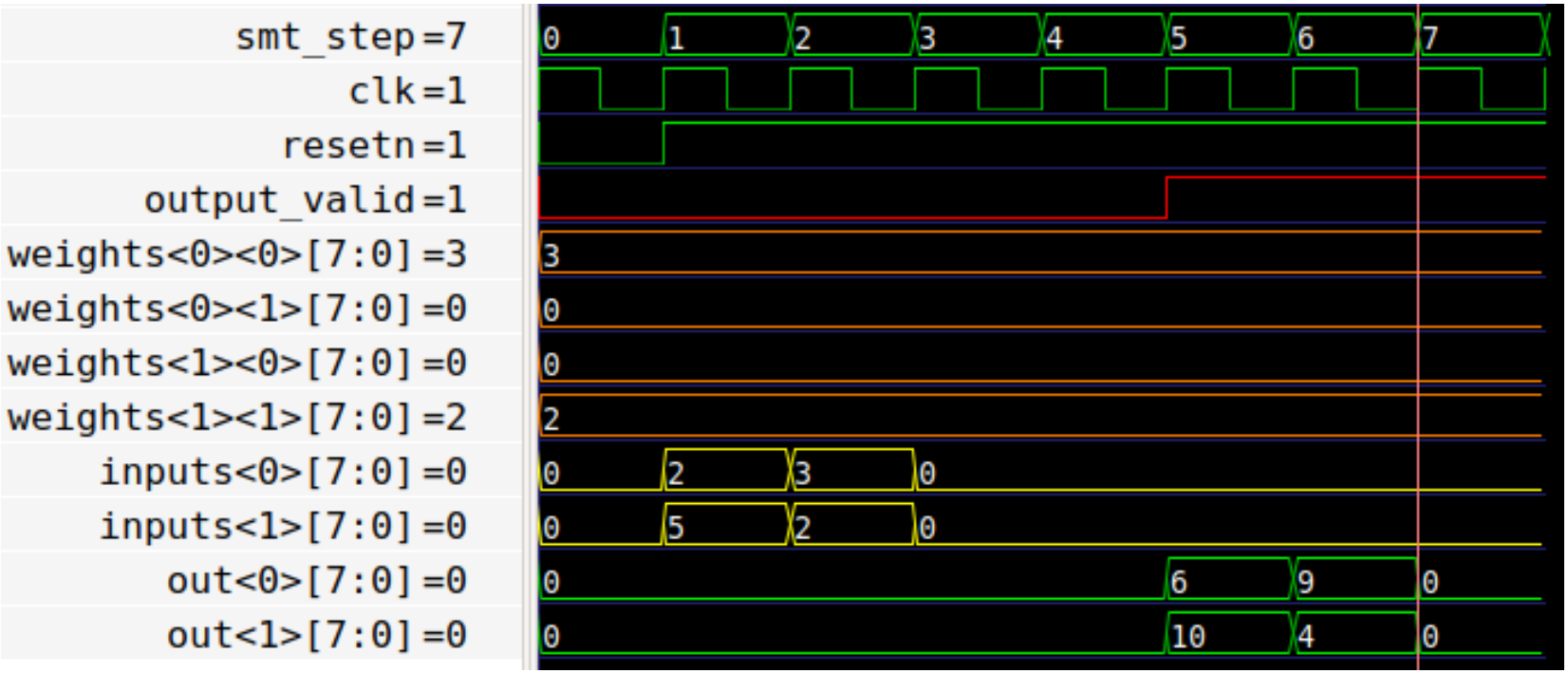
Bonus: Matrix inversion and LU factoring through `cover`

- A fundamental part of SVA and hardware formal verification are `cover` statements.
- The formal verification tool generates the **shortest trace** that **satisfies** the **SVA**.
- Important in the initial stages of verification for **sanity checks**: **constrain the inputs** in some interesting ways and examine the trace. Important to prevent vacuous assertion passes.
- Can also use `cover` statements in a more interesting way: **constrain the output** in some interesting way, and let the tool figure out the necessary inputs / intermediate signals.
- Can therefore use **Yosys** as a **feature-rich** (can use the full System Verilog) **constraint solver**.

Example simple cover trace

- cover trace for a 2×2 matrix multiplication (Find output O such that $I \cdot W = O$):

$$\begin{pmatrix} 2 & 5 \\ 3 & 2 \end{pmatrix} \cdot \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 6 & 10 \\ 9 & 4 \end{pmatrix}$$



Bonus: Matrix inversion and LU factoring through `cover`

- Matrix Inversion (Find I such that $I \cdot W = O \mod 2^8$):

$$\begin{pmatrix} \texttt{0x4C} & \texttt{0x60} & \texttt{0x6B} \\ \texttt{0xB5} & \texttt{0x86} & \texttt{0xBB} \end{pmatrix} \cdot \begin{pmatrix} 3 & 255 & 1 \\ 4 & 2 & 7 \\ 23 & 42 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \mod 2^8$$

- LU factoring (Find L, U such that $L \cdot U = O \mod 2^8$):

$$\begin{pmatrix} \texttt{0x81} & \texttt{0x00} & \texttt{0x00} \\ \texttt{0x04} & \texttt{0xDD} & \texttt{0x00} \\ \texttt{0x83} & \texttt{0x46} & \texttt{0xD2} \end{pmatrix} \cdot \begin{pmatrix} \texttt{0x81} & \texttt{0x81} & \texttt{0x81} \\ \texttt{0x00} & \texttt{0x8B} & \texttt{0xB7} \\ \texttt{0x00} & \texttt{0x00} & \texttt{0xC3} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 4 & 3 & 255 \\ 3 & 5 & 3 \end{pmatrix}$$

Conclusion

- Systolic Arrays and other dataflow-intensive circuits can be verified with modern hardware formal verification tools up to 32×32.
- The chosen module **interface** has a **huge impact** on verification **speed** and **capabilities**.
- `s_eventually` liveness properties are much **slower** than **fixed delay** bounds.
- **Carefully-crafted assertions** can **speed-up** verification, but they can also make it **slower**. It's hard to predict the effect of an assertion without trying it.
- Hardware formal verification tools can be used together with `cover` statements as **feature-rich constraint solvers**.

Any questions?