

ESE5320: System-on-a-Chip Architecture

Day 21: November 11, 2024
Verification 2



Penn ESE5320 Fall 2024 -- DeHon

1

Today

- Assertions (Part 1)
- Proving correctness (Part 2)
 - FSM Equivalence
- Timing and Testing (Part 3)

Penn ESE5320 Fall 2024 -- DeHon

2

2

Message

- If you don't test it, it doesn't work.
- Testing can only prove the presence of bugs, not the absence.
 - Full verification strategy is more than testing.
- Valuable to decompose testing
 - Functionality
 - Functionality at performance

Penn ESE5320 Fall 2024 -- DeHon

3

3

Assertions

Penn ESE5320 Fall 2024 -- DeHon

4

4

Assertion

- Properties expect/demand to hold
- Predicate (Boolean expression) that must be true
- Add to code
 - Can use variables in code to write expression
- Example: `assert (num < 100);`
- Invariant
 - Expect/demand this property to always hold
 - **Never vary** → never not be true

Penn ESE5320 Fall 2024 -- DeHon

5

5

Equivalence with Reference as Assertion

- Match of test and golden reference is a heavy-weight example of an assertion
- `r=fimpl(in);`
- `assert (r==fgolden(in));`

Penn ESE5320 Fall 2024 -- DeHon

6

6

Assertion as Invariant

- May express a property that must hold without expressing how to compute it.
 - Different than just a simpler way to compute

```
int res[2];
res=divide(n,d);
assert(res[QUOTIENT]*d+res[REMAINDER]==n);
```

Penn ESE5320 Fall 2024 -- DeHon

7

7

Lightweight

- Typically, lighter weight (less computation) than full equivalence check
- Typically, less complete than full check
- Allows continuum expression

Penn ESE5320 Fall 2024 -- DeHon

8

8

Preclass 1

What property needs to hold on l ?

Note: divide: s/l

```
s=packetsum(p);
l=packetlen(p);
res=divide(s,l);
```

Penn ESE5320 Fall 2024 -- DeHon

9

9

Check a Requirement

```
s=packetsum(p);
l=packetlen(p);
assert(l!=0);
res=divide(s,l);
```

Penn ESE5320 Fall 2024 -- DeHon

10

10

Preclass 2

What must be true of `my_array[loc]` after call?

```
int findloc(int target, int *a, int limit);
.
.
int loc;

loc=findloc(my_target,my_array,MY_ARRAY_LEN);
// property on my_array[loc] should hold here?
.
.
.
```

Penn ESE5320 Fall 2024 -- DeHon

11

11

Merge using Streams

Day 13

- Merging two sorted list is a streaming operation
- `int aptr; int bptr;`
- `astream.read(ain); bstream.read(bin)`
- For (`i=0;i<MCNT;i++`)
 - If (`(aptr<ACNT) && (bptr<BCNT)`)
 - If (`ain>bin`)
 - { `ostream.write(ain); aptr++; astream.read(ain);` }
 - Else
 - { `ostream.write(bin) bptr++; bstream.read(bin);` }
 - Else // copy over remaining from astream/bstream

Penn ESE5320 Fall 2024 -- DeHon

12

Merge Requirement

- Require: *astream*, *bstream* sorted
- `int aptr; int bptr;`
- `astream.read(ain); bstream.read(bin)`
- For (`i=0;i<MCNT;i++`)
 - If (`((aptr<ACNT) && (bptr<BCNT))`)
 - If (`ain>bin`)
 - { `ostream.write(ain); aptr++; astream.read(ain);`}
 - Else
 - { `ostream.write(bin) bptr++; bstream.read(bin);`}
 - Else // copy over remaining from *astream*/*bstream*

Penn ESE5320 Fall 2024 -- DeHon

13

13

Merge Requirement

- Require: *astream*, *bstream* sorted
- `Int ptr; int bptr;`
- `astream.read(ain); bstream.read(bin)`
- For (`i=0;i<MCNT;i++`)
 - If (`((aptr<ACNT) && (bptr<BCNT))`)
 - If (`ain>bin`)
 - { `ostream.write(ain); aptr++;`
 - `int prev_ain=ain; astream.read(ain);`
 - `assert(prev_ain>=ain);`
 - }

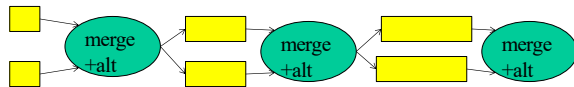
Penn ESE5320 Fall 2024 -- DeHon

14

14

Merge with Order Assertion

- When composed
 - Every downstream merger checks work of predecessor



Penn ESE5320 Fall 2024 -- DeHon

15

15

Merge Requirement

- Require: *astream*, *bstream* sorted
- Requirement that input be sorted is good
 - And not hard to check
- Not comprehensive
 - Weaker than saying output is a sorted version of input
- What errors would it allow?

Penn ESE5320 Fall 2024 -- DeHon

16

16

What do with Assertions?

- Include logic during testing (verification)
- Omit once tested
 - Compiler/library/macros (`#define`) omit code
 - Keep in source code
- Maybe even synthesize to gate logic for FPGA testing
- When assertion fail
 - Count
 - Break program for debugging (dump core)

Penn ESE5320 Fall 2024 -- DeHon

17

17

Assertion Roles

- Specification (maybe partial)
 - May address state that doesn't exist in gold reference
- Documentation
 - This is what I expect to be true
 - Needs to remain true as modify in the future
- Defensive programming
 - Catch violation of input requirements
- Catch unexpected events, inputs
- Early failure detection
- Validate that something isn't happening

Penn ESE5320 Fall 2024 -- DeHon

18

18

Assertion Discipline

- Worthwhile discipline
 - Consider and document input/usage requirements
 - Consider and document properties that must always hold
- Good to write those down
 - As precisely as possible
- Good to check assumptions hold

Penn ESE5320 Fall 2024 -- DeHon

19

19

Equivalence Proof

FSM
Part 2

Penn ESE5320 Fall 2024 -- DeHon

20

20

Prove Equivalence

- Testing is a subset of Verification
- Testing can only prove the presence of bugs, not the absence.
- Depends on picking an adequate set of tests
- Can we guarantee that all behaviors are the correct? Same as reference? Seen all possible behaviors?

Penn ESE5320 Fall 2024 -- DeHon

21

21

Idea

- Reason about all behaviors
 - Response to all possible inputs
- Try to find if there is *any* way to reach disagreement with specification
- Or can prove that they always agree

- Still demands specification
 - ...but we can also relax that with assertions

Penn ESE5320 Fall 2024 -- DeHon

22

22

Testing with Reference Specification

Day 19

Validate the design by testing it:

- Create a set of test inputs
- Apply test inputs
 - To implementation under test
 - To reference specification
- Collect response outputs
- Check if outputs match

Penn ESE5320 Fall 2024 -- DeHon

23

23

Formal Equivalence with Reference Specification

Validate the design by proving equivalence between:

- implementation under consideration
- reference specification

Penn ESE5320 Fall 2024 -- DeHon

24

24

Testing FSM Equivalence

- Exhaustive:
 - Generate all strings of length $|state|$
 - (for larger FSM = the one with the most states)
 - Feed to both FSMs with these strings
 - Observe any differences?
- How many such strings?
 - $(N \text{ binary input bits to FSM, } S \text{ states})$
 - $2^{N \cdot S}$

Penn ESE5320 Fall 2024 -- DeHon

25

25

FSM Equivalence

- Illustrate with concrete model of FSM equivalence
 - Is some implementation FSM
 - Equivalent to reference FSM

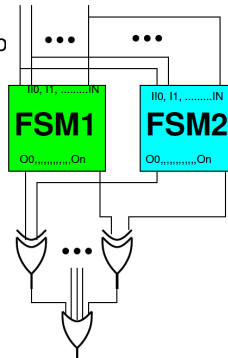
Penn ESE5320 Fall 2024 -- DeHon

26

26

Compare

- Start with golden model setup
 - Run both and compare output
- Create composite FSM
 - Start with both FSMs
 - Connect common inputs together (Feed both FSMs)
 - XOR together outputs of two FSMs
 - Xor's will be 1 if they disagree, 0 otherwise



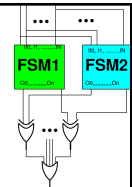
Penn ESE5320 Fall 2024 -- DeHon

7

27

Compare

- Create composite FSM
 - Start with both FSMs
 - Connect common inputs together (Feed both FSMs)
 - XOR together outputs of two FSMs
 - Xor's will be 1 if they disagree, 0 otherwise
- Ask if the new machine ever generate a 1 on an xor output (signal disagreement)
 - Any 1 is a proof of non-equivalence
 - Never produce a 1 \rightarrow equivalent



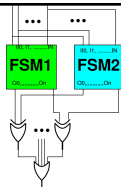
Penn ESE5320 Fall 2024 -- DeHon

28

28

Creating Composite FSM

- Assume know start state for each FSM
- Each state in composite is labeled by the pair $\{S1_i, S2_j\}$
 - How many such states?
- Start in $\{S1_0, S2_0\}$
- For each input a , create a new edge:
 - $T(a, \{S1_0, S2_0\}) \rightarrow \{S1_i, S2_j\}$
 - If $T_1(a, S1_0) \rightarrow S1_i$, and $T_2(a, S2_0) \rightarrow S2_j$
- Repeat for each composite state reached



Penn ESE5320 Fall 2024 -- DeHon

29

29

Composite FSM

- How much work?
- Hint:
 - Maximum number of composite states (state pairs)
 - Maximum number of edges from each state pair?
 - Work per edge?

Penn ESE5320 Fall 2024 -- DeHon

30

30

Composite FSM

- Work
 - At most $|2^N| * |\text{State1}| * |\text{State2}|$ edges == work
- Can group together original edges
 - *i.e.* in each state compute intersections of outgoing edges
 - Really at most $|E_1| * |E_2|$

Penn ESE5320 Fall 2024 -- DeHon

31

31

Non-Equivalence

- State $\{S1_i, S2_j\}$ demonstrates non-equivalence iff
 - $\{S1_i, S2_j\}$ reachable
 - On some input, State $S1_i$ and $S2_j$ produce different outputs
- If $S1_i$ and $S2_j$ have the same outputs for all composite states, it is impossible to distinguish the machines
 - They are equivalent
- A **reachable** state with differing outputs
 - Implies the machines are not identical

Penn ESE5320 Fall 2024 -- DeHon

32

32

Answering Reachability

- Start at composite start state $\{S1_0, S2_0\}$
- Search for path to a differing state
- Use any search
 - Breadth-First Search, Depth-First Search
- End when find differing state
 - Not equivalent
- OR when have explored entire reachable graph without finding
 - Are equivalent

Penn ESE5320 Fall 2024 -- DeHon

33

33

Reachability Search

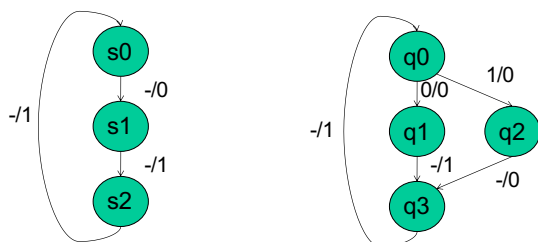
- Worst: explore all edges at most once
 - $O(|E|) = O(|E_1| * |E_2|)$
- Can combine composition construction and search
 - *i.e.* only follow edges which fill-in as search
 - (way described)

Penn ESE5320 Fall 2024 -- DeHon

34

34

Preclass 3



- Means don't-care. Can read as (0 or 1) here.

Penn ESE5320 Fall 2024 -- DeHon

35

35

Creating Composite FSM

- Assume know start state for each FSM
- Each state in composite is labeled by the pair $\{S1_i, S2_j\}$
- Start in $\{S1_0, S2_0\}$
- For each symbol a , create a new edge:
 - $T(a, \{S1_0, S2_0\}) \rightarrow \{S1_i, S2_j\}$
 - If $T_1(a, S1_0) \rightarrow S1_i$ and $T_2(a, S2_0) \rightarrow S2_j$
 - Check that both state machines produce same outputs on input symbol a
- Repeat for each composite state reached

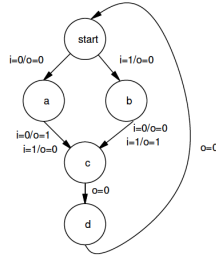
Penn ESE5320 Fall 2024 -- DeHon

36

36

Preclass 4

i	State	NextState	o
0	S0	S1	0
1	S0	S2	0
0	S1	S3	1
1	S1	S4	0
0	S2	S4	0
1	S2	S4	1
-	S3	S5	0
-	S4	S5	0
-	S5	S0	0



Penn ESE5320 Fall 2024 -- DeHon

37

37

FSM → Model Checking

- FSM case simple – only deal with states
- More general, need to deal with
 - operators (add, multiply, divide)
 - Wide word registers in datapath
 - Cause state exponential in register bits
- Tricks
 - Treat operators symbolically
 - Separate operator verification from control verif.
 - Abstract out operator width
- Similar flavor of case-based search
 - Conditionals need to be evaluated symbolically

Penn ESE5320 Fall 2024 -- DeHon

38

38

Assertion Failure Reachability

- Can use with assertions
- Is assertion failure reachable?
 - Can identify a path (a sequence of inputs) that leads to an assertion failure?

Penn ESE5320 Fall 2024 -- DeHon

39

39

Formal Equivalence Checking

- Rich set of work on formal models for equivalence
 - Challenges and innovations to making search tractable
 - Used with processor validation
- Common versions
 - Model Checking (2007 Turing Award)
 - Bounded Model Checking

Penn ESE5320 Fall 2024 -- DeHon

40

40

Timing

Part 3

Penn ESE5320 Fall 2024 -- DeHon

41

41

Issues

- Cycle-by-cycle specification can be overspecified
- Golden Reference Specification not run at target speed

Penn ESE5320 Fall 2024 -- DeHon

42

42

Tokens

- Use data presence to indicate when producing a value
- Only compare corresponding outputs
 - Only store present outputs from computations, since that's all comparing
- Relevant non-Real-Time
- Examples?
 - (not want to match cycle-by-cycle)

Penn ESE5320 Fall 2024 -- DeHon

43

43

Timing

- Record timestamp from implementation
- Allow reference specification to specify its time stamps
 - “Model this as taking one cycle”
 - Or requirements on its timestamps
 - This must occur before cycle 63
 - This must occur between cycle 60 and 65
- Compare values and times
- More relevant Real Time
- Example Real Time where exact cycle **not matter?**

Penn ESE5320 Fall 2024 -- DeHon

44

44

Challenge

- Cannot record at full implementation rate
 - Inadequate bandwidth to
 - Store off to disk
 - Get out of chip
- Cannot record all the data you might want to compare at full rate

Penn ESE5320 Fall 2024 -- DeHon

45

45

At Speed Testing

- Compiled assertions might help
 - Perform the check at full rate so don't need to record
- Capture bursts to on-chip memory
 - Higher bandwidth
 - ...but limited capacity, so cannot operate continuously

Penn ESE5320 Fall 2024 -- DeHon

46

46

Bursts to Memory

- Run in bursts
- Repeat
 - Enable computation
 - Run at full rate storing to memory buffer
 - Stall computation
 - Offload memory buffer at (lower) available bandwidth
 - (possibly check against golden model)

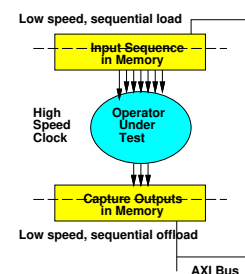
Penn ESE5320 Fall 2024 -- DeHon

47

47

Generalize

- Generalize to input and output
- Feed from memories
- Compute full rate
- Write into memory
- Can run at high rate for number of cycles can store inputs and outputs



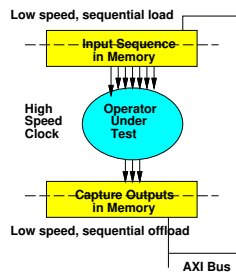
Penn ESE5320 Fall 2024 -- DeHon

48

48

Generalize

- Generalize to input and output
- Feed from memories
- Compute full rate
- Write into memory



- What might this fail to test?

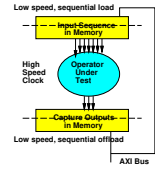
Penn ESE5320 Fall 2024 -- DeHon

49

49

Burst Testing

- Issue
 - May only see high speed for computation/interactions that occur within a burst period
 - May miss interaction at burst boundaries
- Mitigation
 - Rerun with multiple burst boundary offsets
 - So all interactions occur within some burst
 - Decorrelate interaction and burst boundary



Penn ESE5320 Fall 2024 -- DeHon

50

50

Timing Validation

- Doesn't need to be all testing either
- Static Timing Analysis to determine viable clock frequency
 - As Vivado is providing for you
- Cycle estimates as get from Vivado
 - II, to evaluate a function
- Worst-Case Execution Time for software

Penn ESE5320 Fall 2024 -- DeHon

51

51

Decompose Verification

Breaks into two pieces:

1. Does it function correctly?
2. What speed does it operate it?
 - Does it continue to work correctly at that speed?

Penn ESE5320 Fall 2024 -- DeHon

52

52

Learn More

- CIS6730 – Computer Aided Verification
- CIS5410 – includes verification for real-time system properties
- CIS5000 – Software Foundations
 - Has mechanized proofs, proof checkers

Penn ESE5320 Fall 2024 -- DeHon

53

53

Big Ideas

- Assertions valuable
 - Reason about requirements and invariants
 - Explicitly validate
- Formally validate equivalence when possible
- Valuable to decompose testing
 - Functionality
 - Functionality at performance
- ...we can extend techniques to address timing and support at-speed tests

Penn ESE5320 Fall 2024 -- DeHon

54

54

Admin

- Feedback
- Reading for Wednesday on Canvas
- P3 due Friday