ESE5320: System-on-a-Chip Architecture

Day 20: November 8, 2023
Verification 1

Today

- Part 1:
  - Motivation
  - Challenge and Coverage
- Part 2:
  - Golden Model / Reference Specification
- Part 3:
  - Automation and Regression

Message

- If you don't test it, it doesn't work.

- Verification is important and challenging
- Demands careful thought
  - Tractable and adequate coverage
- Value to a simple functional reference
- **Must** be automated and rerun with changes
  - Often throughout lifecycle of design

Goal

- Assure design works correctly
  - Not fail and lose consumer confidence.
    - …or lose them money, privacy, service availability….
  - Not kill anyone
    - Ethical issue
  - Not lose points on your grade ☺

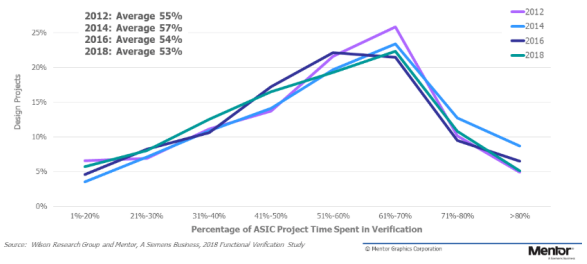
Challenge

- Designs are complex
  - Many ways things can go wrong
  - Many *subtle* ways things can go wrong
  - Many tricky interactions
- Designs are often poorly specified
  - Complex to completely specify

Verification

- Often dominant cost in product
  - Requires most manpower (cost)
  - Takes up most of schedule
    - In the critical path to making money

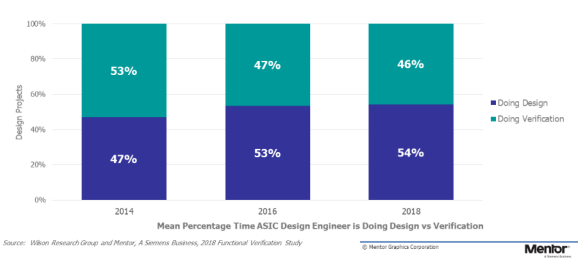
ASIC: Percentage of Project Time Spent in Verification



https://semiengineering.com/the-weather-report-2018-study-on-ic-asic-verification-trends/

7

ASIC: Mean % Time Design Engineer is Doing Design vs Verification



https://semiengineering.com/the-weather-report-2018-study-on-ic-asic-verification-trends/

8

Correctness?

- How do we define correctness for a design?
- How do we know the design is correct?
- How do we know the design remains correct when?
  - Add a some feature
  - Perform an optimization
  - Fix a bug

9

Life Cycle

- Design
  - specify what means to be correct
- Development
  - Implement and refine
  - Fix bugs
  - Optimize
- Operation and Maintenance
  - Discover bugs, new uses and interaction
  - Fix and provide updates
- Upgrade/revision

10

Testing and Coverage

11

Strawman Testing

Validate the design by testing it:

- Create a set of test inputs
- Apply test inputs
- Collect response outputs
- Check if outputs match expectations

12

Strawman: Inputs and Outputs

Validate the design by testing it:

- Create a set of test inputs
  - How do we generate an adequate set of inputs? (know if a set is adequate?)
- Apply test inputs
- Collect response outputs
- Check if outputs match expectations
  - How do we know if outputs are correct?

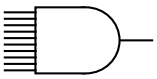
Try 1: Inputs and Outputs

- Create a set of test inputs
  - How do we generate an adequate set of inputs? (know if a set is adequate?)
    - All possible inputs
- Check if outputs match expectations
  - How do we know if outputs are correct?
    - Manually identify correct output

How many input cases?

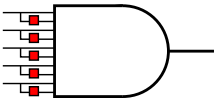
Combinational:

- 10-input AND gate?
- Any N-input combinational function?



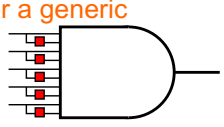
Add Pipelining

- The output doesn't correspond to the input on a single cycle
- Need to think about inputs sequences to output sequences
- How many input cases?



Add Pipelining

- The output doesn't correspond to the input on a single cycle
- Need to think about inputs sequences to output sequences
- How many input cases for a generic acyclic circuit?
  - Depth d
  - Inputs N
  - Simple case: just clock in inputs over d



Add Feedback State

- When have state
  - Different inputs can produce different outputs
- Behavior depends on state
- Need to reason about all states the design can be in

How many input cases?

- Function of 8b input
- Update of 32b checksum when given new 8b of input
  - Static int current; // internal state
  - Void CKSUM32(unsigned char input)
    {current=CKSUM(current,input);}
    Void CKSUM32reset()
    {current=0;}

How many input cases?

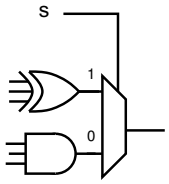
- Function of 8b input
- Update of 32b checksum when given new 8b of input
  - Void CKSUM32(unsigned char input)
    {current=CKSUM(current,input);}
- If only have access to input,
  - How long a sequence to get current into a potential value?

Observation

- Cannot afford
  - Exhaustively generate input cases
  - Manual write output expectations
- Will need to be smarter about test case selection

Structural Simplifications

- How many cases if treat as 7-input function?
- How many useful cases
  - If hold s at 0?
  - If hold s at 1?
  - Together total cases?



Useful Test Cases

```
int fun(int s,a,b,c,d) {
  if (s>20)
     if (s>100)
        return(a+b); else return(b+c);
  else
     if (s<0)
        return(c+d); else return(a+d);
}
```

What values of s will be interesting? --- likely to exhibit different behavior?

When s=10, what values of a, b, c, d interesting? – likely to help verify/debug?

Finite State Machine

- Logic depends on state
  - May have different logic in every state
  - May transition to different states based on state and input

```
int state;
while (true) {
  switch (state) {
    case (ST1): out=1; state=ST2; break;
    case (ST2):  if (in>0) {out=2; state=ST3;}
                 else {out=0; state=ST2;} break;
```

Finite State Machine

- What input cases should we try to exercise for an FSM? (goal for test cases)

```
int state;
while (true) {
   switch (state) {
      case (ST1): out=1; state=ST2; break;
      case (ST2):  if (in>0) {out=2; state=ST3;}
                   else {out=0; state=ST2;} break;
      case (ST3): ….
```


Coverage

- Do our tests execute every line of code?
  - What percentage of the code is exercised?
- Gate-level designs
  - Can we toggle every gate output?
- Necessary but not sufficient
  - Not exercised or not toggled, definitely not testing some functionality
    - Remember: If you don't test it, it doesn't work.
- Measurable

So far…

- Identifying test stimulus important and tricky
  - Cannot generally afford exhaustive
  - Need understand/exploit structure
- Coverage metrics a start
  - Not complete answer

Reference Specification (Golden Model)

Part 2

Strawman: Inputs and Outputs

Validate the design by testing it:

- Create a set of test inputs
  - How do we generate an adequate set of inputs? (know if a set is adequate?)
- Apply test inputs
- Collect response outputs
- Check if outputs match expectations
  - How do we know if outputs are correct?

Problem

- Manually writing down results for all input cases
  - Tedious
  - Error prone
  - …simply not viable for large number cases need to cover
    - Definitely not viable exhaustive
    - …and still not viable when select intelligently

Specification Model

- Ideally, have a function that can
  - compute the correct output
  - for any input sequence
- ``Gold Standard" – an oracle
  - Whatever the function says is truth
- Could be another program
  - Written in a different language? Same language?

Testing with Reference Specification

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

Test against Specification

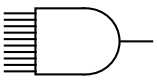
- Relieved ourselves of writing outputs
- Still have to select input cases
  - Can freely use larger set since not responsible for manually generating output match

Random Inputs

- Can use random inputs
  - Since can generate expected output for any case
- Use coverage metric to see how well random inputs are exercising the code
- Can be particularly good to identify interactions and corner cases didn't think of manually
- Still unlikely to generate very obscure cases

Random inputs

Combinational:

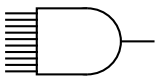


Expected number inputs to cause output to toggle?

- 10-input AND gate?

Random inputs

Combinational:



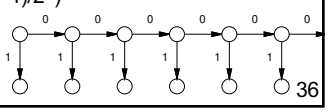
Expected number inputs to cause output to toggle?

- 10-input AND gate?

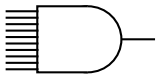
P(need more than m) = $((2^N-1)/2^N)^m$

$0.5 = (1023/1024)^m$

m~=709



Random inputs

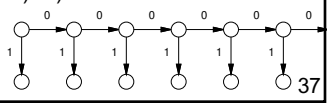


Combinational:

Want high probability of toggle?

P(need more than m) = $((2^N-1)/2^N)^m$

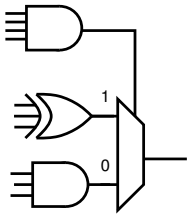
$P_{toggle} = (1023/1024)^m$



37

Random Inputs

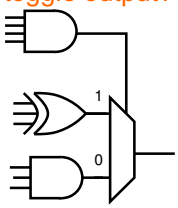
- Expected number of tests to toggle output?
  - Compare exhaustive



38

Random Inputs

- Expected number of tests to toggle output?
  - Compare exhaustive
- P(AND4 1)=1/16
- P(xor has 1)=1/2
- P(AND3 1)=1/8

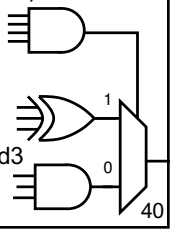


- P(get 1) = (1/16)*(1/2)+(15/16)*(1/8)~=0.15
  - 4 or 5 likely to generate a toggle

39

Random Inputs

- Expected number of tests to toggle output?
  - Compare exhaustive
- P(get 1) = (1/16)*(1/2)+(15/16)*(1/8)~=0.15
  - 4 or 5 likely to generate a toggle
- Still not guarantee test both
  - More to guarantee propagate toggle form xor3 and and3



40

Observation

- In many cases, random can find interesting cases quickly
  - Maybe not minimum, but small compared to exhaustive
- Some cases may be as bad as exhaustive
- Coverage metrics give us hints/guidance of which is which

41

Random Testing

- Completely random may be just as bad as exhaustive
  - Expected time to exercise interesting piece of code
  - Expected time to produce a legal input
    - E.g. – random packets will almost always have erroneous checksums
  - E.g. random bytes won't generate duplicate chunks, or much opportunity for LZW compression

42

Semi-Random

- How could we generate more useful but "randomized" inputs?
  - Focus on things we need to exercise

Biased Random

- Non-uniform random generation of inputs
  - Compute checksums correctly most of the time
    - Control rate and distribution of checksum errors
- Randomize properties of input:
  - What are some properties we might want to vary for our compression/deduplication?

Biased Random

- Non-uniform random generation of inputs
  - Compute checksums correctly most of the time
    - Control rate and distribution of checksum errors
- Randomize properties of input, E.g.
  - Lengths of repeated sequences
  - Distance between repeated sequences
  - Edit sequence applied to differentiate files

Testing with Reference Specification

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

Specification

- Where would we get a reference specification?
  - and why should we trust it?
  - Isn't this just another design that can be *equally* buggy?

Standard

- Many standards includes a reference implementation.

Existing Product

- Many times there's an existing product or open-source implementation…

Develop Specification

- Maybe develop a simple, functional implementation as part of early design

Specification Correct?

- How would we know the specification is correct? -- why should we trust it?
  - Simpler/smaller
    - Less opportunity for bugs
    - Written for function/clarity not performance
  - **Different**
    - Ok as long as reference and implementation don't have same bugs
      - Debug and test them against each other

Common Bugs

- Combinational (for simplicity)
- 5 input function, single output
- Assume two specifications have 1% error rate (1% of input cases wrong)
- Assume independent
  - (key assumption – weaker to extent wrong)
- Probability of both giving same wrong result?
  - For a particular input case?
  - Across all input cases?

Common Bugs

- Assuming Random, Independent errors
- P(not catch) = P1(bug)*P2(bug)
- P(not catch across all)
  ~= cases*P(not catch)
- Better:

  P(not catch across all) =
  $1-(1-P(\text{not catch}))^{\text{cases}}$

Day 13

Window Filter

- Compute based on neighbors
- for (y=0;y<YMAX;y++)
  for (x=0;x<XMAX;x++)

```
o[y][x]=F(d[y-1][x-1],d[y-1][x],d[y-1][x+1],
          d[y][x-1],d[y][x],d[y][x+1],
          d[y+1][x-1],d[y+1][x],d[y+1][x+1]);
```

Window Filter

Day 13

- Single read and write from dym, dy
- for (y=0;y<YMAX;y++)
  for (x=0;x<XMAX;x++) {
  dypxm=dypx; dypx=dnew; dnew=d[y+1][x+1];
  dyxm=dyx; dyx=dyxp; dyxp=dy[x+1];
  dymxm=dymx; dymx=dymxp; dymxp=dym[x+1];
  o[y][x]=F(dymxm,dymx,dymxp,
  dyxm,dyx,dyxp,
  dypxm,dypx,dnew);
  dym[x-1]=dyxm;dy[x-1]=dypxm; }

55

Simpler Functional

- Other examples of functional specification being simpler than implementation?

56

Simpler Functional

- Sequential vs. parallel
- Unpipelined vs. pipelined
- Simple algorithm
  - Brute force?
- No data movement optimizations
- Use robust, mature (well-tested) building blocks

57

Testing with Reference Specification

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

Refine inputs based on coverage metrics

58

Coverage

- Of specification or implementation?
  - Almost certainly both
- Specification may have a case split that implementation doesn't have
  - E.g. handle exceptional case
- Implementation typically have many more cases to handle in general

59

Automation and Regression

Part 3

60

Automated

- Testing suite **must** be automated
  - Single script or make build to run
  - Just start the script
  - Runs through all testing and comparison without manual interaction
  - Including scoring and reporting a single pass/fail result
    - Maybe a count of failing cases

61

Regression Test

- Regression Test -- Suite of tests to run and validate functionality
- To identify if your implementation has "regressed" – returned to a previously buggy state

62

Regression Tests

- One big test or many small tests?
- Benefit of many small tests?
- Benefit of big test(s)?

63

Automation Mandatory

- Will run regression suite repeatedly during Life Cycle
  - Every change
  - As optimize
  - Every bug fix

64

Life Cycle

- Design
  - specify what means to be correct
- Development
  - Implement and refine
  - Fix bugs
  - optimize
- Operation and Maintenance
  - Discover bugs, new uses and interaction
  - Fix and provide updates
- Upgrade/revision

65

Automation Value

- Engineer time is bottleneck
  - Expensive, limited resource
  - Esp. the engineer(s) that understand what the design should do
- Cannot spend that time evaluating/running tests
- Reserve it for debug, design, creating tests
- Capture knowledge in tools and tests

66

When find a bug

- If regression suite didn't originally find it
  - Add a test (expand regression suite) so will have a test to cover
- Make sure won't miss it again
- Test suite monotonically improving

When add a feature

- Add a test to validate that feature
  - And interaction with existing functionality
- Maybe add the test first…
  - See test identifies lack of feature before add functionality
  - …then see (correctly added) feature satisfies test

Continuous Integration

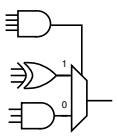
- When commit code to shared repo (git, svn)
  - Build and run regression suite
  - Perhaps before allow commit
  - Guarantee not break good version
    - Or, at least, know how functional/broken the current version is
- Alternately (complement), nightly regression
  - Automation to check out, build, run tests

Regression Test Size

- Want to be comprehensive
  - More tests better….
- Want to run in tractable time
  - Few minutes once make change or when checkin
  - Cannot run for weeks or months
  - Might want to at least run overnight
- Sometimes forced to subset
  - Small, focused subset for immediate test
  - Comprehensive test for full validation

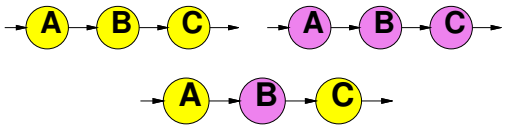
Unit Tests

- Regression for individual components
- Good to validate independently
- Lower complexity
  - Fewer tests
  - Complete quickly
- Make sure component(s) working before run top-level design tests
  - One strategy for long top-level regression



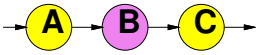
Functional Scaffolding

- If functional decomposed into components like implementation
- Replace individual components with implementation
  - Use reference/functional spec for rest



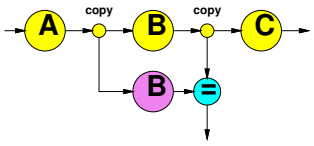
Functional Scaffolding

- If functional decomposed into components like implementation
- Replace individual components with implementation
  - Use reference/functional spec for rest
- Independent test of integration for that module



Functional Scaffolding

- If functional decomposed into components like implementation
- Run reference component and implementation together and check outputs



Decompose Specification

- Should specification decompose like implementation?
  - ultimate golden reference
    - Only if that decomposition is simplest
- But, worth refining
  - Golden reference simplest
  - Intermediate functional decomposed
    - Validate it versus golden
    - Still simpler than final implementation
    - Then use with implementation

Big Ideas

- Testing
  - Designs are complicated, need extensive validation – If you don't test it, it doesn't work.
  - Exhaustive testing not tractable
  - Demands care
  - Coverage one tool for helping identify
- Reference specification as "gold" standard
  - Simple, functional
- Must automate regression
  - Use regularly throughout life cycle

Admin

- Feedback
- P2 due Friday
- P3 out
- Next Week …