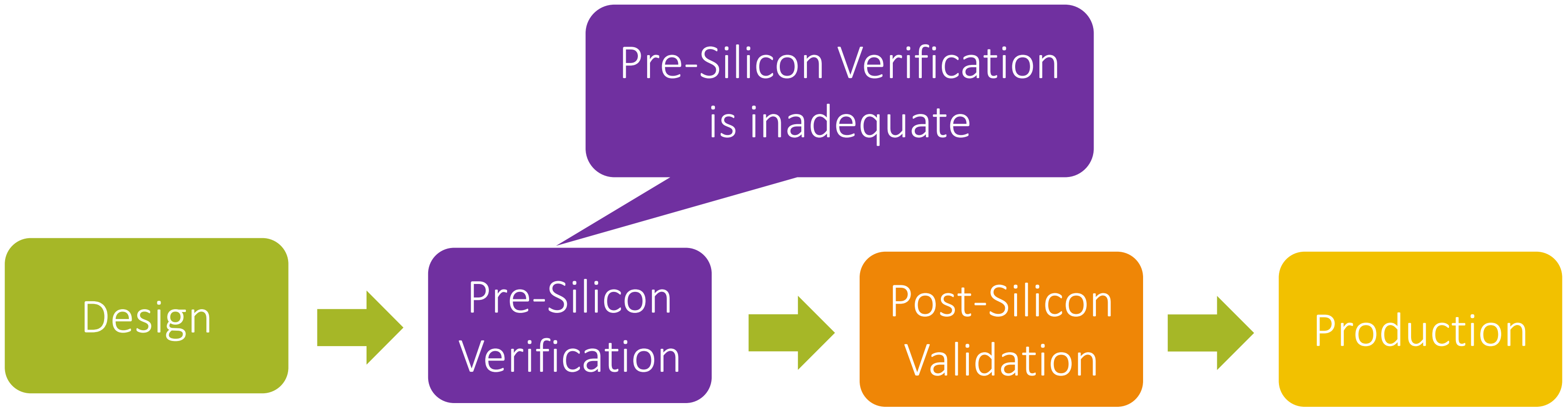
Threadmill: A Post-Silicon Exerciser for Multi-Threaded Processors

Authors: *Allon Adir, Maxim Golubev, Shimon Landa, Amir Nahir, Gil Shurek, Vitali Sokhin, Avi Ziv*

Presenters: Abraham Addisie and Dong-hyeon Park

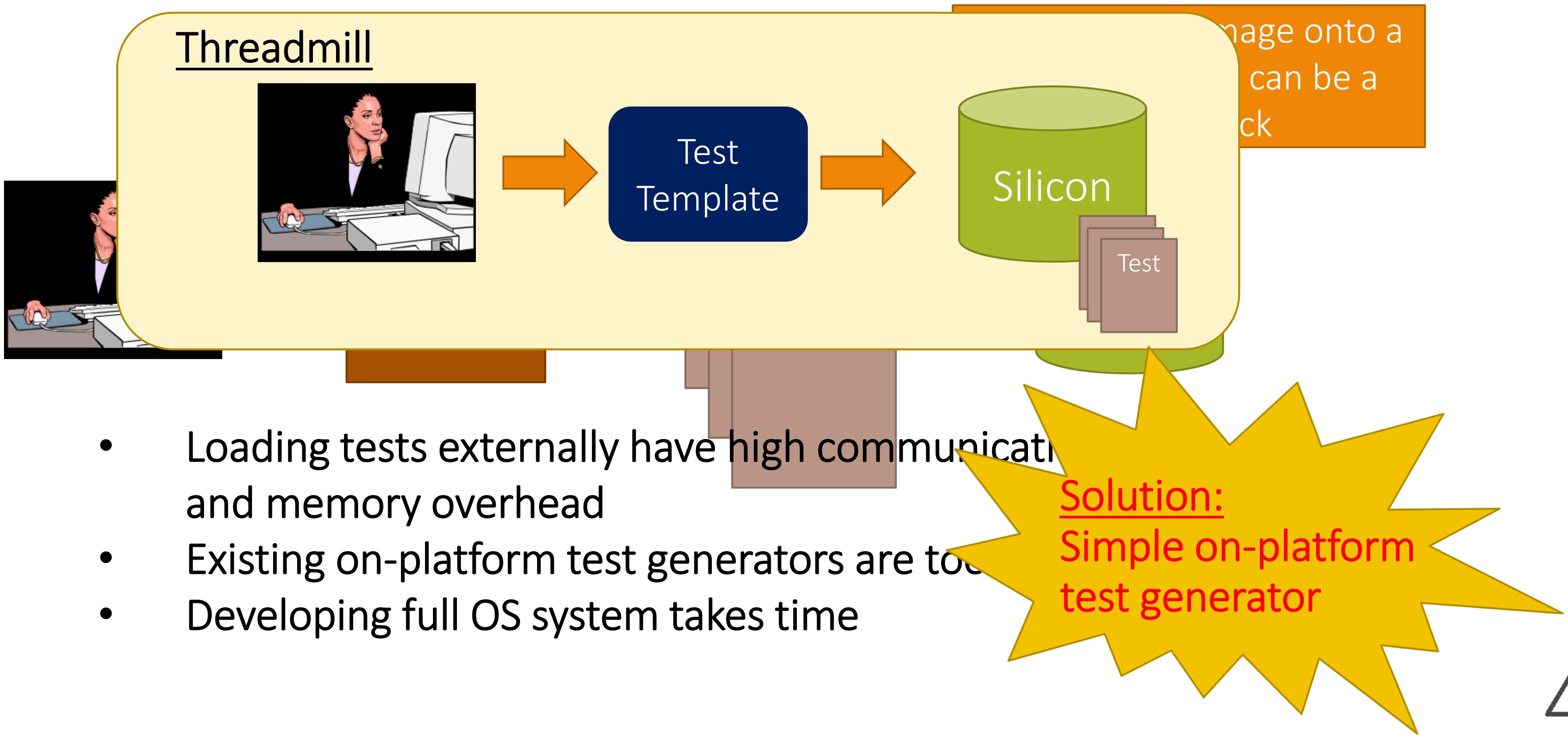
Outline

Motivation – Why Post Silicon?



- Post-Silicon validation is becoming the next-level vehicle for functional verification

Problems of Post-Silicon Validation



- Loading tests externally have high communicati
  and memory overhead
- Existing on-platform test generators are too
- Developing full OS system takes time

Outline

Motivation

Overview of Threadmill

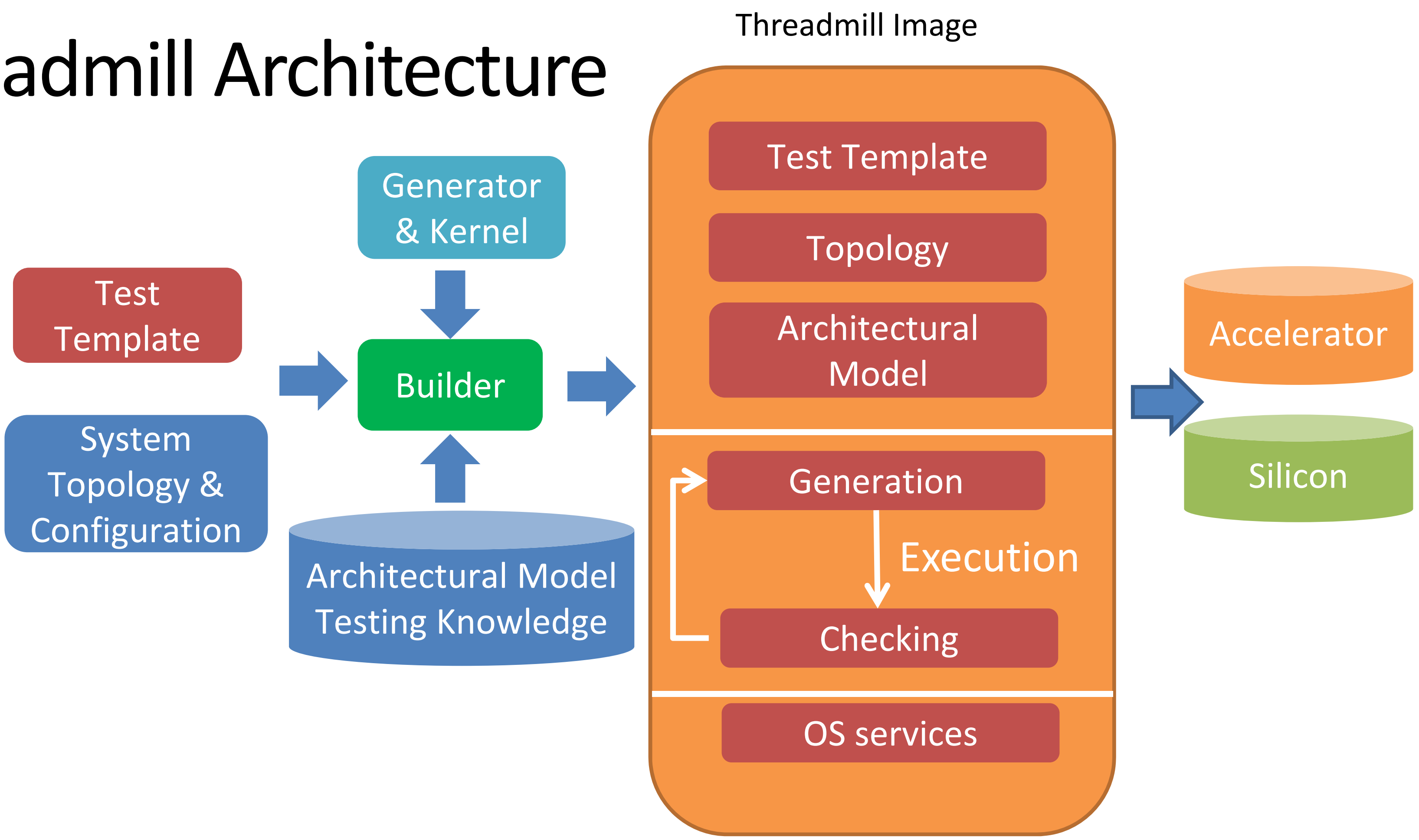
Key Techniques of Threadmill

Conclusion

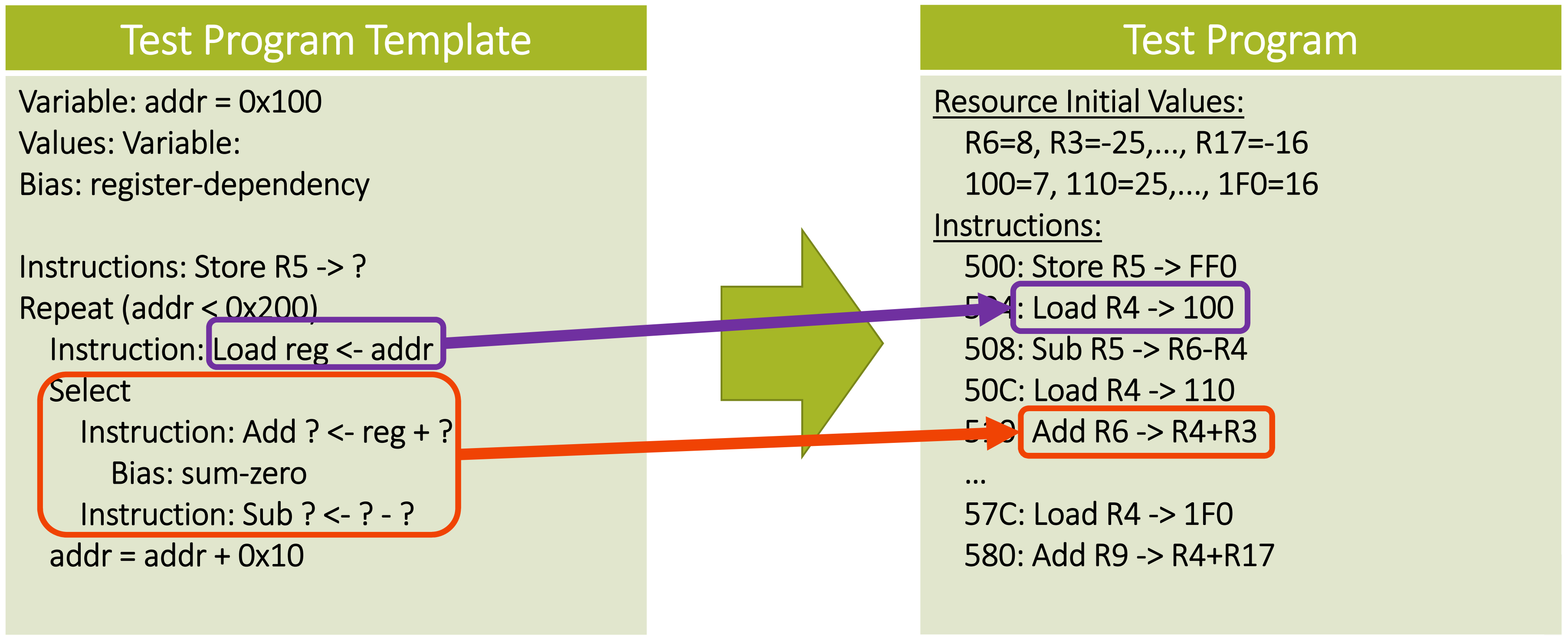
Questions

Debate!

Threadmill Architecture



Test-Template Language



Test Program Template

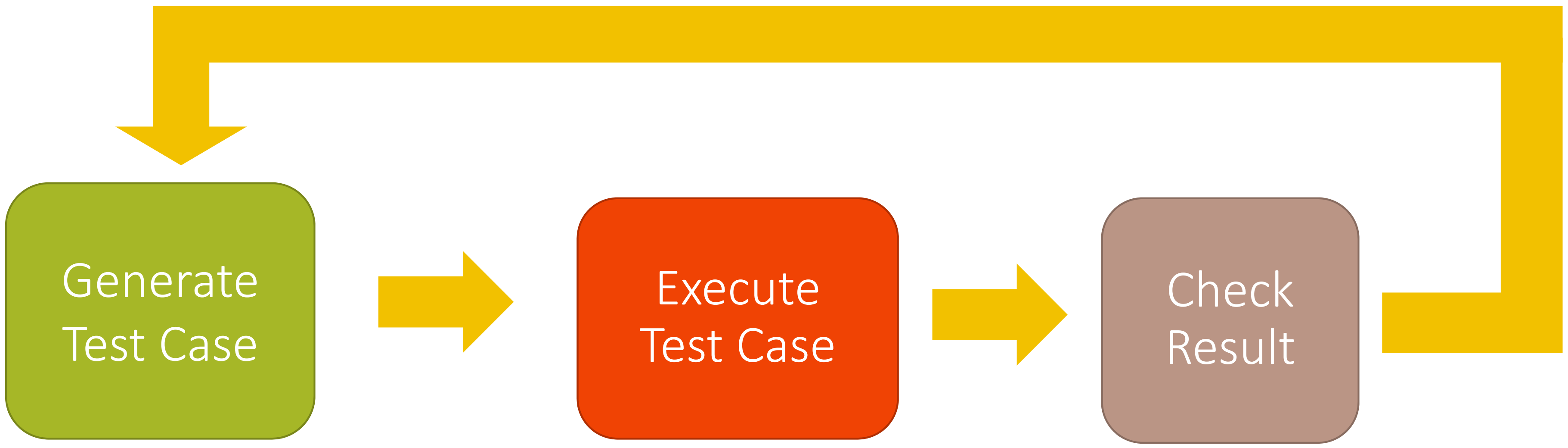
```
Variable: addr = 0x100
Values: Variable:
Bias: register-dependency

Instructions: Store R5 -> ?
Repeat (addr < 0x200)
   Instruction: Load reg <- addr
   Select
      Instruction: Add ? <- reg + ?
         Bias: sum-zero
      Instruction: Sub ? <- ? - ?
   addr = addr + 0x10
```

Test Program

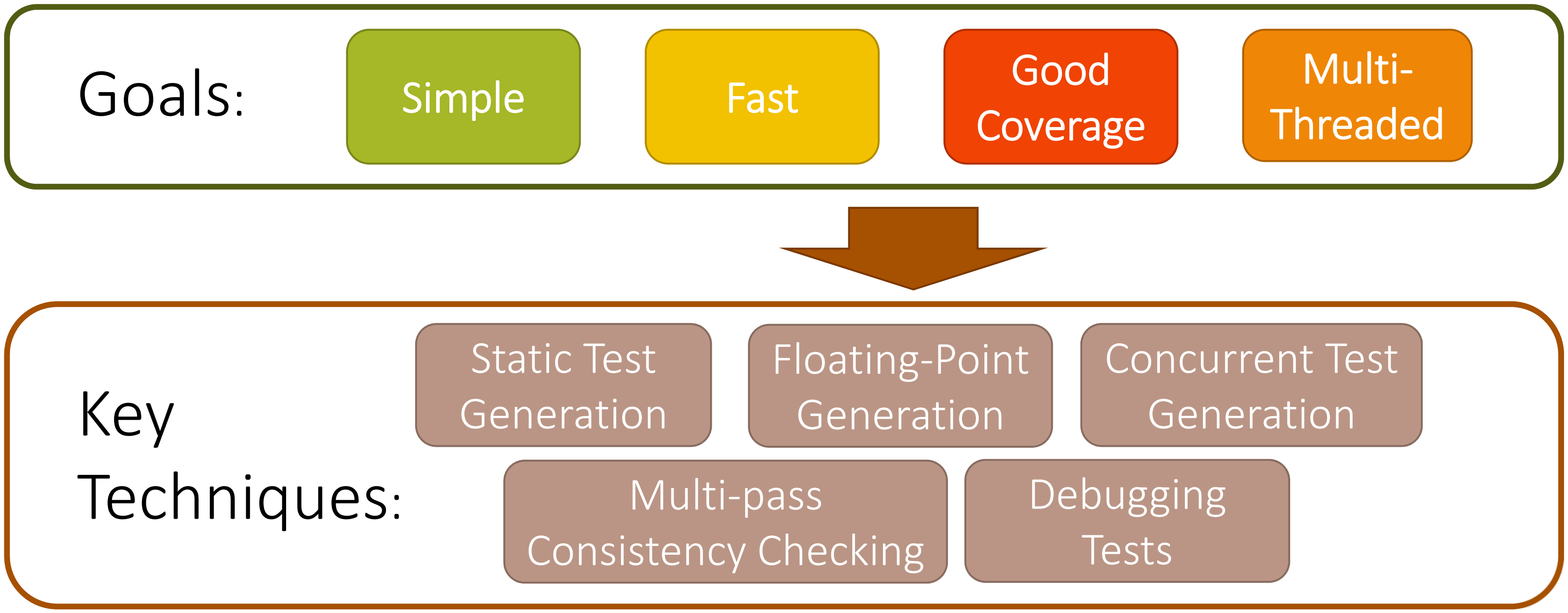
```
Resource Initial Values:
   R6=8, R3=-25,..., R17=-16
   100=7, 110=25,..., 1F0=16
Instructions:
   500: Store R5 -> FF0
   504: Load R4 -> 100
   508: Sub R5 -> R6-R4
   50C: Load R4 -> 110
   510: Add R6 -> R4+R3
   ...
   57C: Load R4 -> 1F0
   580: Add R9 -> R4+R17
```

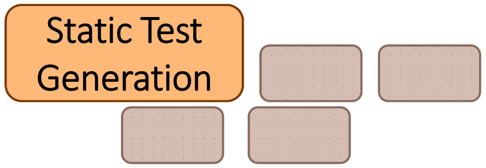
Execution Process



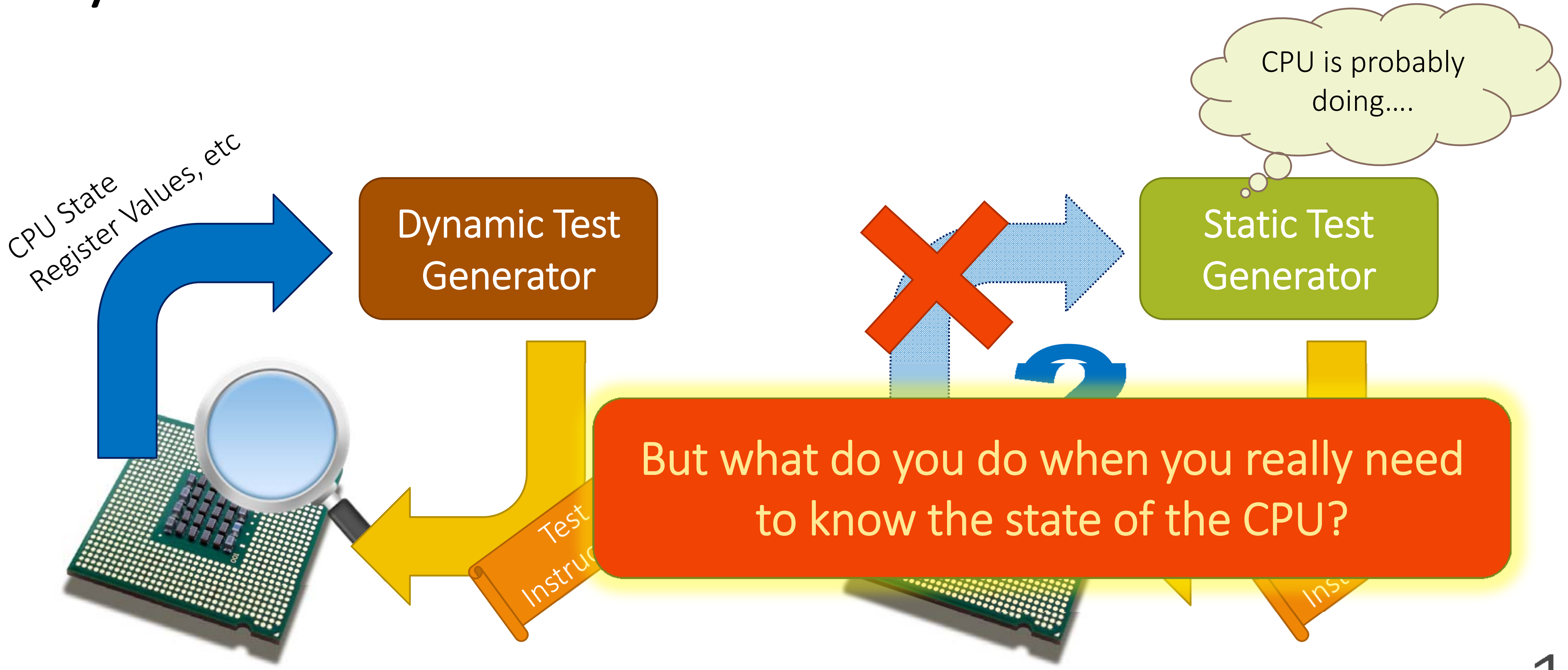
Outline

Threadmill Design

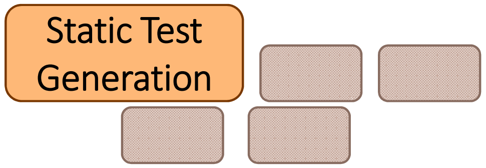




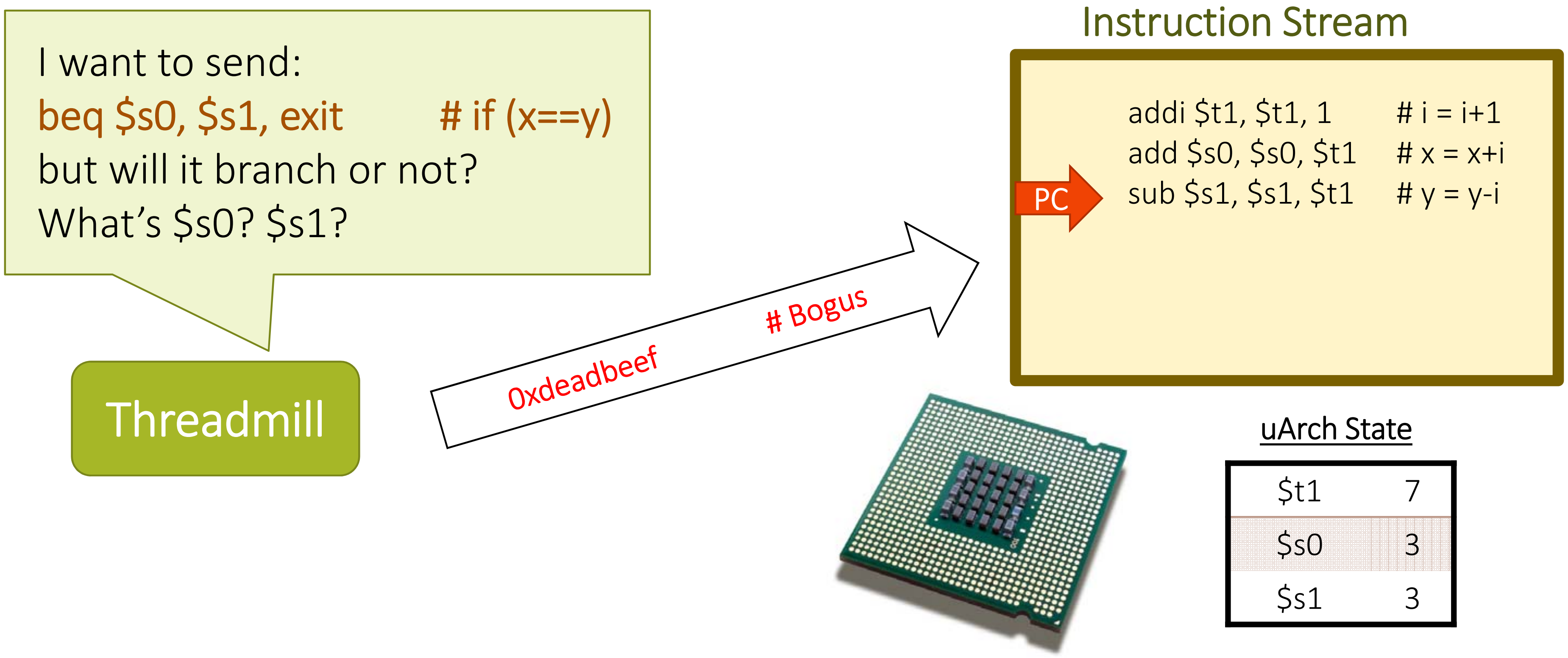
Dynamic vs Static Generation



But what do you do when you really need to know the state of the CPU?



Observing Machine State

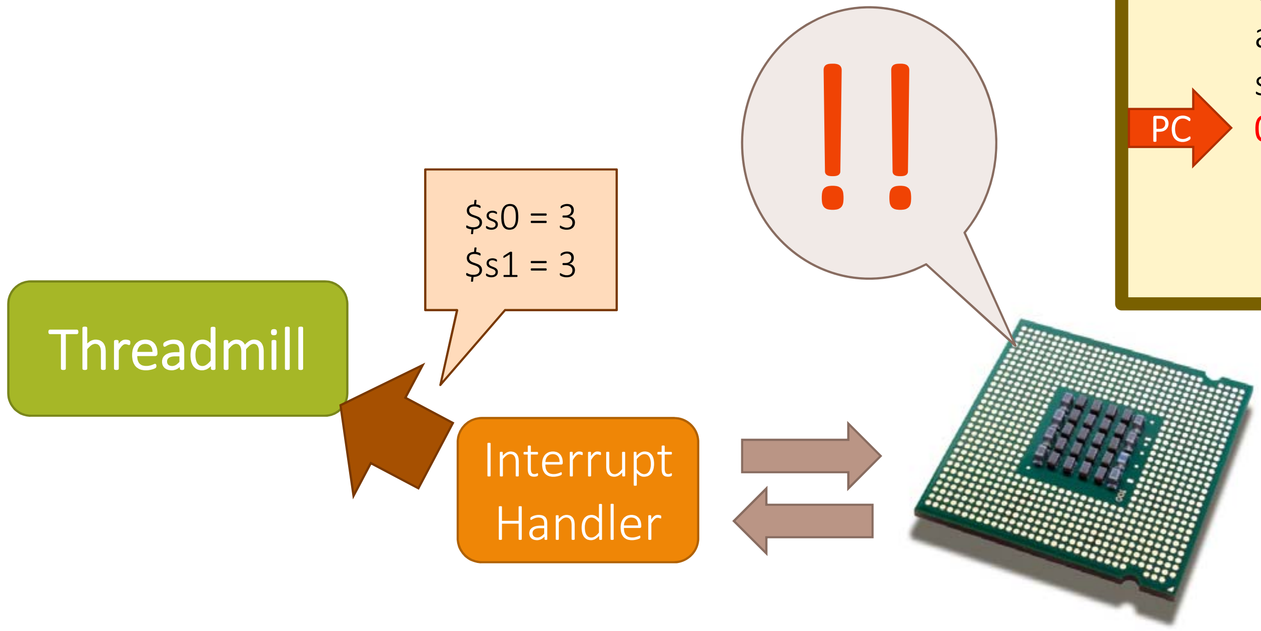


```
addi $t1, $t1, 1      # i = i+1
add $s0, $s0, $t1     # x = x+i
sub $s1, $s1, $t1     # y = y-i
```

uArch State

$t1	7
$s0	3
$s1	3

Observing Machine State



Instructions Executed

```
addi $t1, $t1, 1        # i = i+1
add $s0, $s0, $t1       # x = x+i
sub $s1, $s1, $t1       # y = y-i
PC → 0xdeadbeef         # Bogus
```

uArch State

$t1	7
$s0	3
$s1	3

Static Test Generation

Observing Machine State

Now I know $s0=$s1, so
beq $s0, $s1, exit # if (x==y)
will branch to exit.

Threadmill

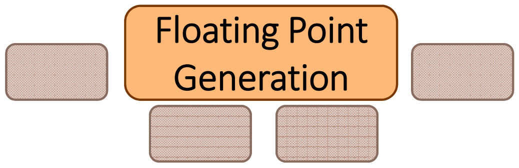
beq $s0, $s1, exit # if (x==y)

Instructions Executed

```
     addi $t1, $t1, 1      # i = i+1
     add $s0, $s0, $t1     # x = x+i
     sub $s1, $s1, $t1     # y = y-l
PC → beq $s0, $s1, exit    # if (x==y)
```

uArch State

$t1	7
$s0	3
$s1	3

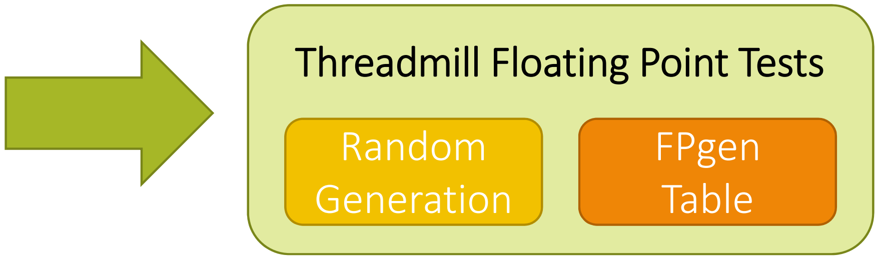


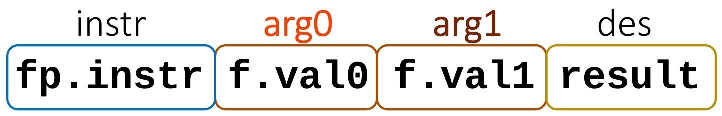
Floating-Point Instructions

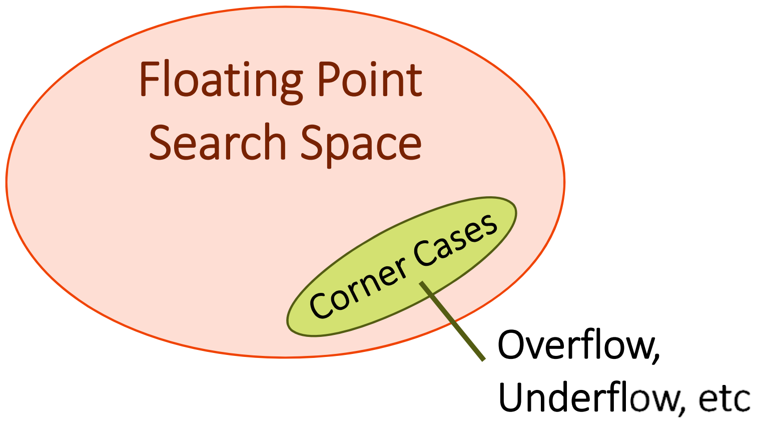
How to choose argument values?

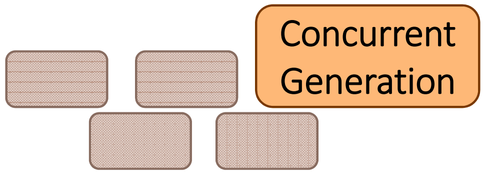
random → inefficient

FPgen: Generate table of interesting test cases “off-line”

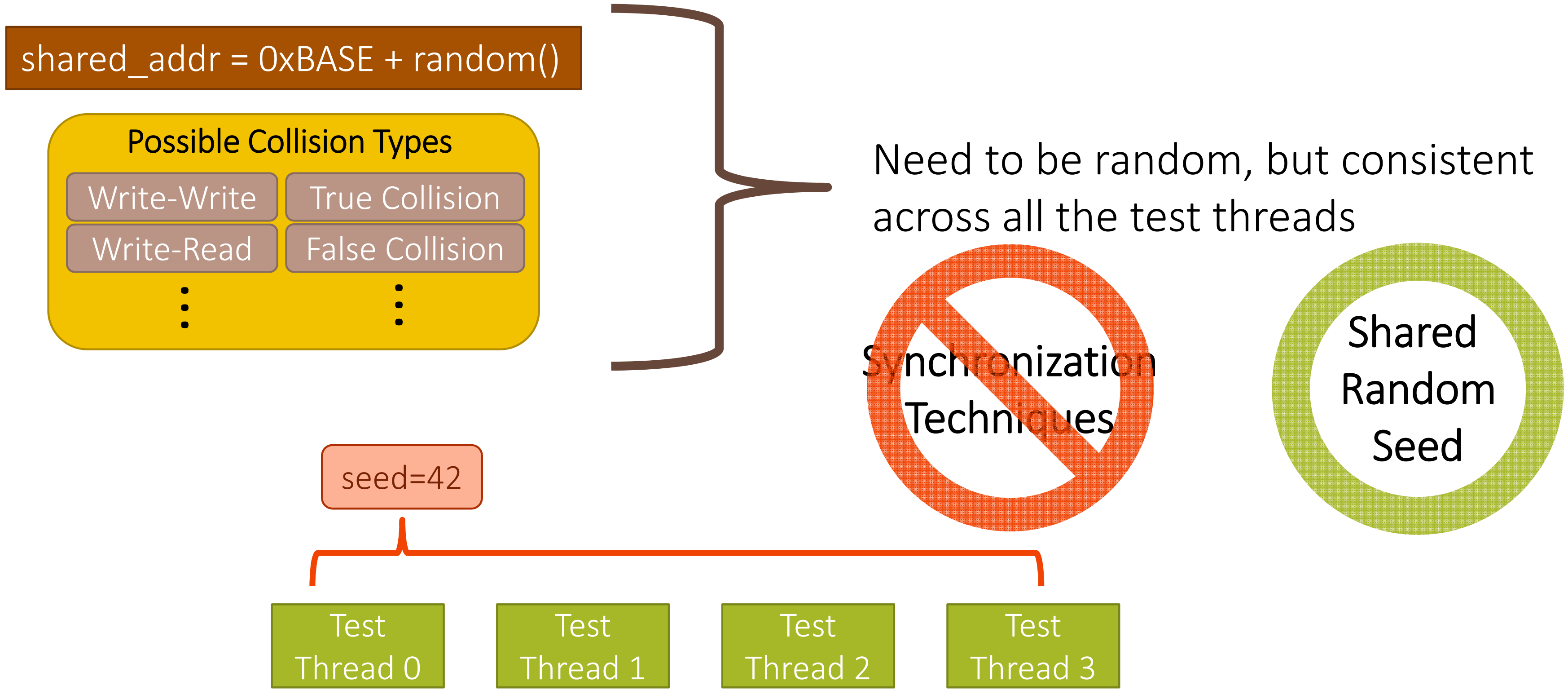


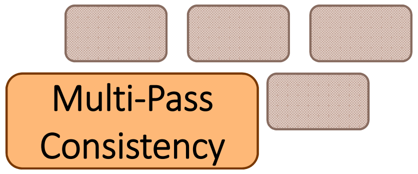






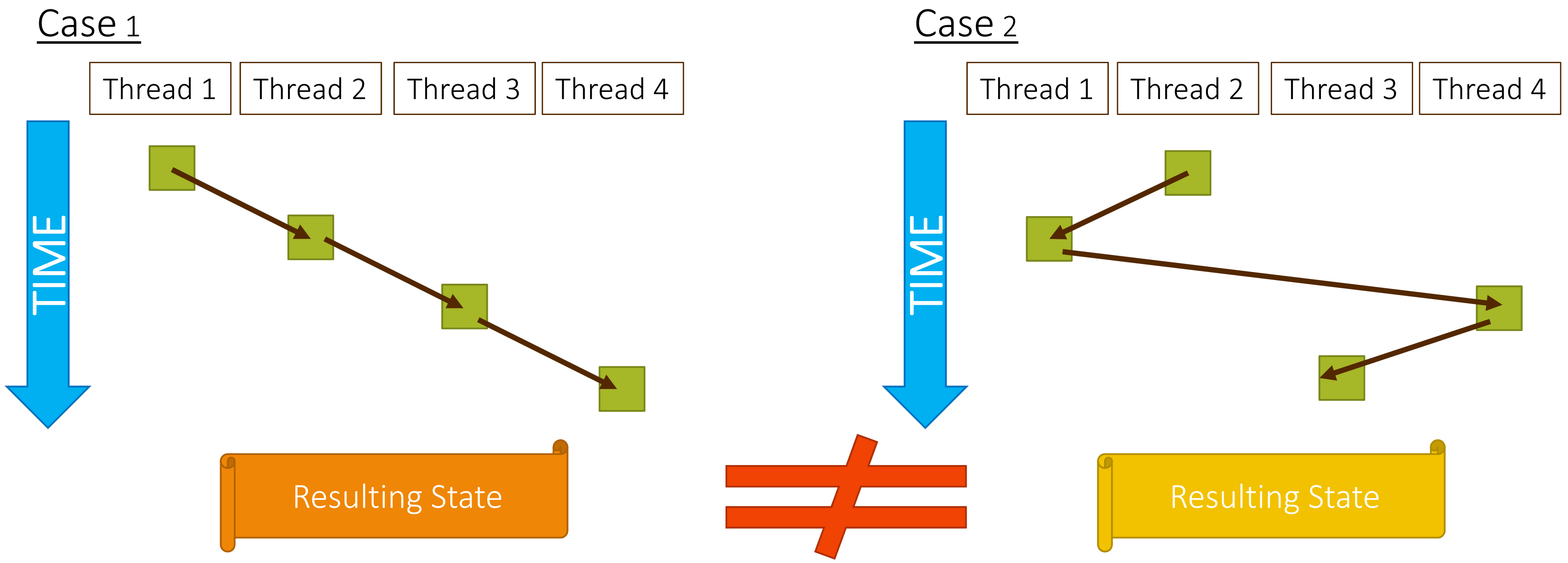
Concurrent Test Generation

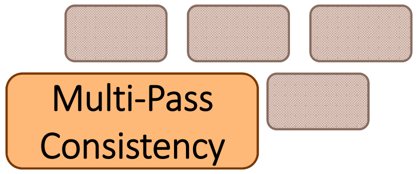




Multi-Pass Consistency Checking

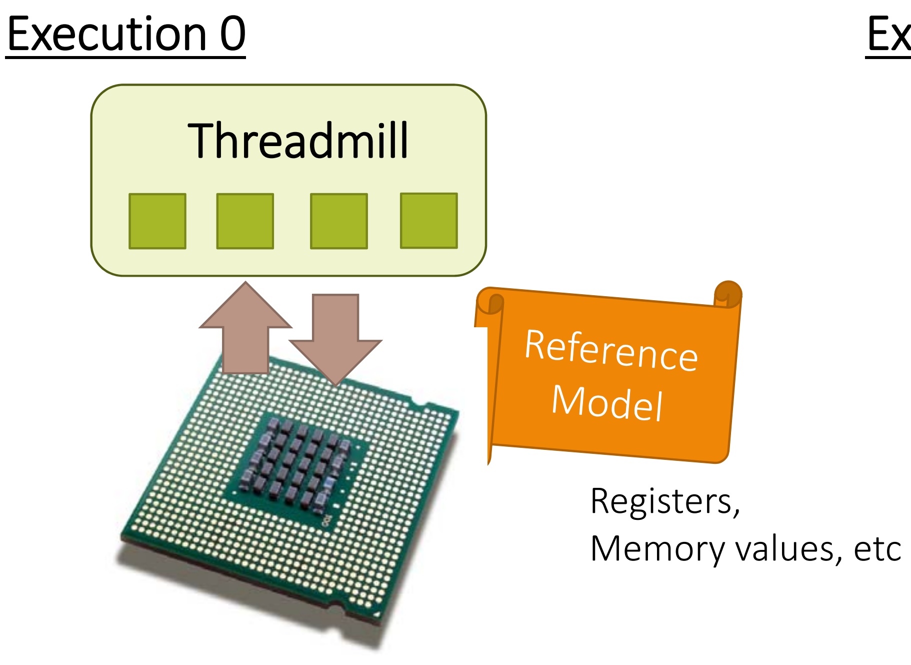
Main Focus: Detecting Bugs in Multi-Threaded Consistency

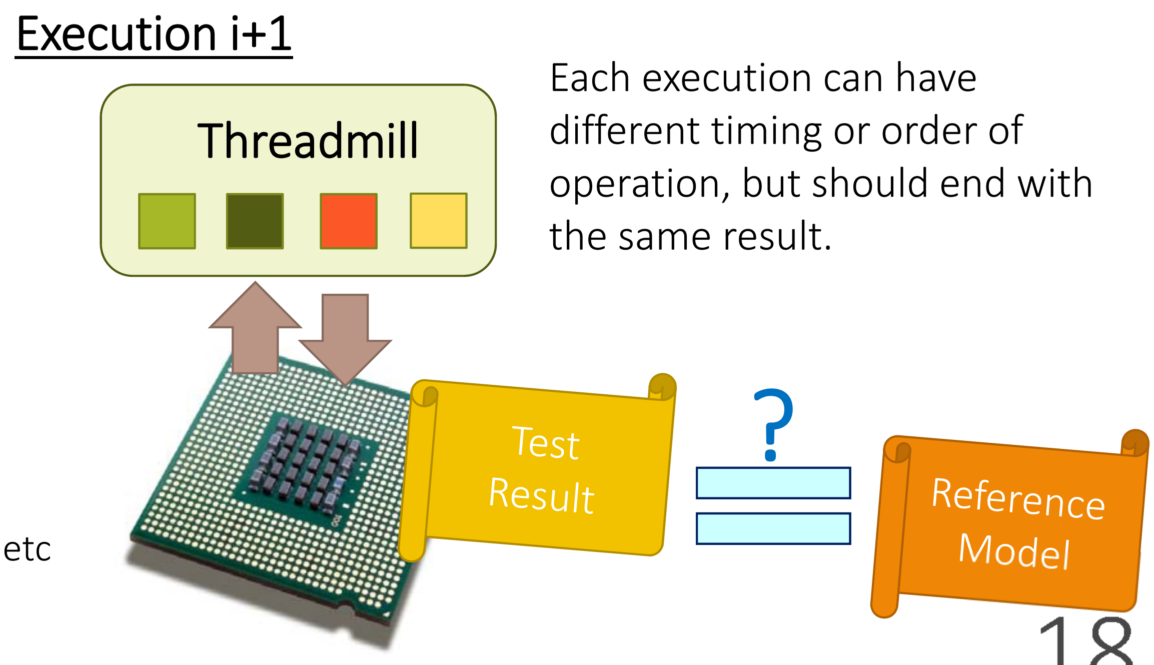




Multi-Pass Consistency Checking

Main Focus: Detecting Bugs in Multi-Threaded Consistency

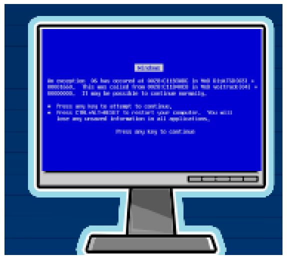


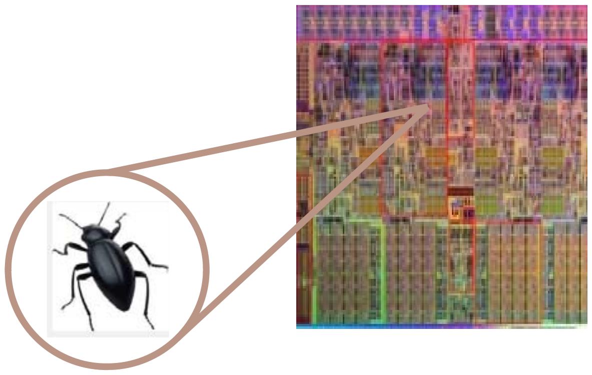


Each execution can have different timing or order of operation, but should end with the same result.

Debugging Tests

- Restart the failed exerciser image a few test-cases before the failure
- Take the test-template that causes the bug and run it on a pre-silicon test generation





Conclusion

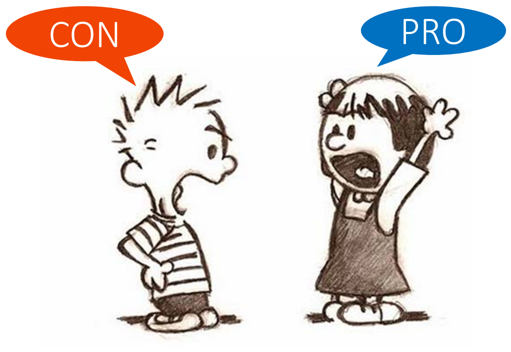
Strengths

- ✓ Fast and Light-weight.
- ✓ Automatic testing and checking of multi-threaded execution.

Weakness

- ❑ Doesn’t check datapath or permanent bugs.
- ❑ Little evaluation of performance or coverage of the platform (paper).

Questions?



Debate

Threadmill's failure detection mechanism only checks for bugs in multi-threaded interactions, and do not check bugs in the datapath. Is this adequate?

Instead of generating tests on-chip, isn't it better to simply load pre-generated tests?