PYNQ™

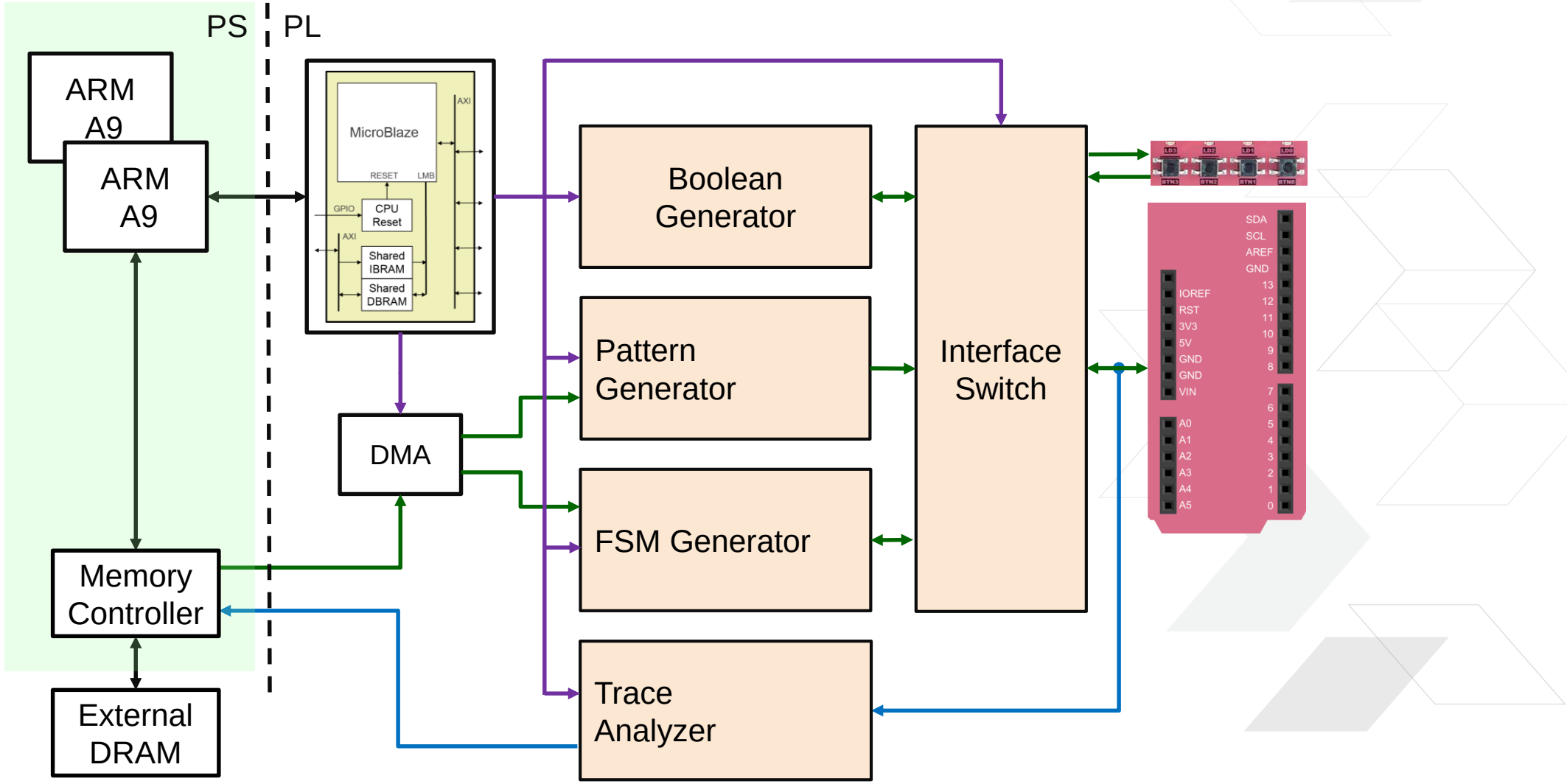
logictools Overlay

XILINX.

Agenda

- logictools Overlay
- Boolean Generator
- API state machine
- Wavedrom
- Pattern Generator
- FSM Generator
- Trace Analyzer

logictools Overlay



XILINX.

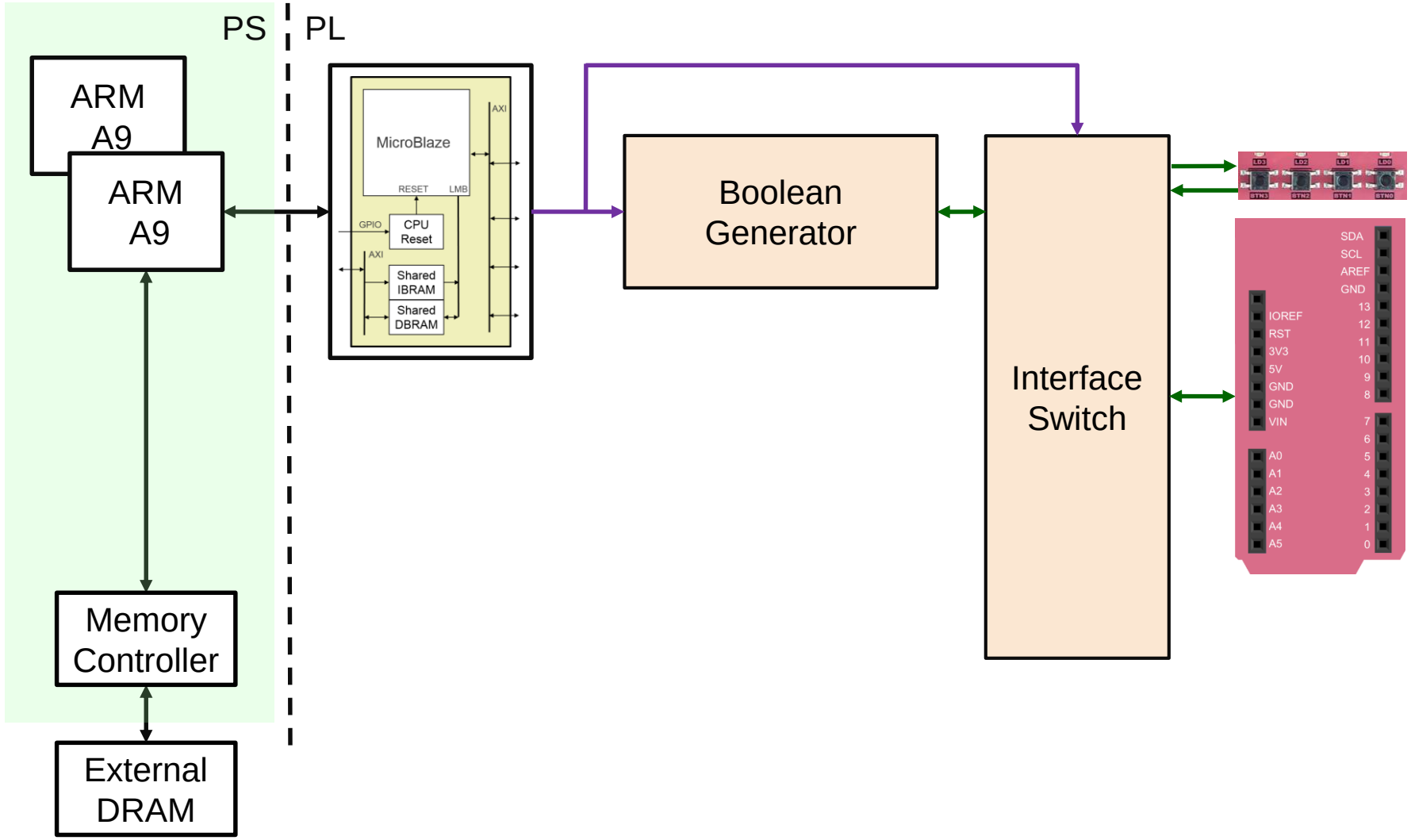
logictools: digital instrument on a chip

- **Controlled from web browser served from ARM A9 CPUs**
- **Declarative**
  - You specify the functionality you want, and where you want it applied
    - Boolean functions, state machines, digital patterns, trace analysis
- **Instantaneous realization**
  - *logictools* creates it for you … instantaneously
- **Reusable open-source software and hardware libraries**
  - Library of novel IP available for reuse by overlay developers
    - Useful also to non-Pynq users

logictools: digital instrument on a chip

- Create a digital circuit by specifying its function and IOs with a simple, declarative API in Python
- Generate Boolean functions, finite state machines & digital patterns on external interface IOs
- Specify and display real-time waveforms in Jupyter Notebooks, in any web browser
- Reusable, open-source implementation of hardware and software libraries

Boolean Generator



Boolean Generator

- **Up to 24, independent Boolean functions of up to 5 inputs each, can be specified simultaneously**

- **Their inputs and outputs can be connected to the 20 digital pins of the Arduino shield interface, plus**
  - PYNQ-Z1's 4 input pushbuttons (PB0 – PB3)
  - PYNQ-Z1's 4 output LEDs (LD0 – LD3)

- **Boolean functions are specified, as strings. For example:** `'LD2 = PB3 ^ PB0'`

  specifies that the values of pushbuttons 3 and 0 are ex-or'ed to produce the value on LED2

Boolean Generator

- **The Boolean Generator can be set up and run, as follows:**

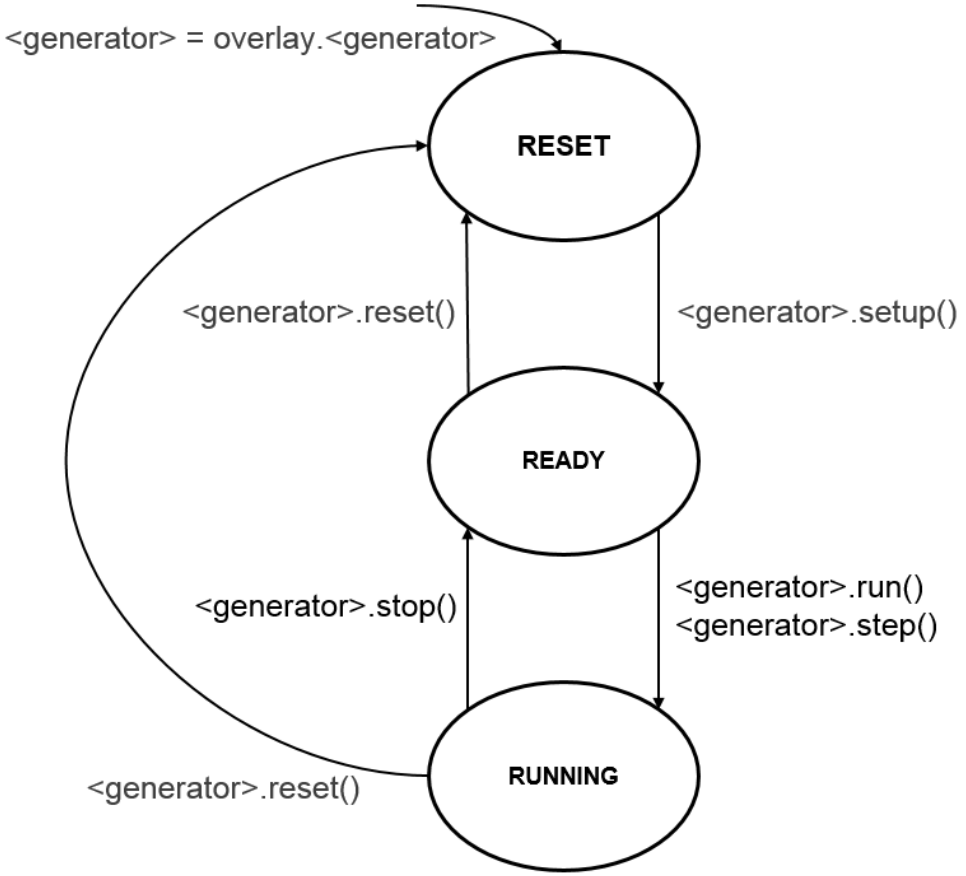
```
from pynq.overlays.logictools import LogicToolsOverlay

logictools_olay = LogicToolsOverlay('logictools.bit')
boolean_generator = logictools_olay.boolean_generator

# Define and set-up a single Boolean function
function = ['LD2 = PB3 ^ PB0']
boolean_generator.setup(function)

# Run the function generator
boolean_generator.run()
```

- **Once complete, the generator is shutdown, as follows:**

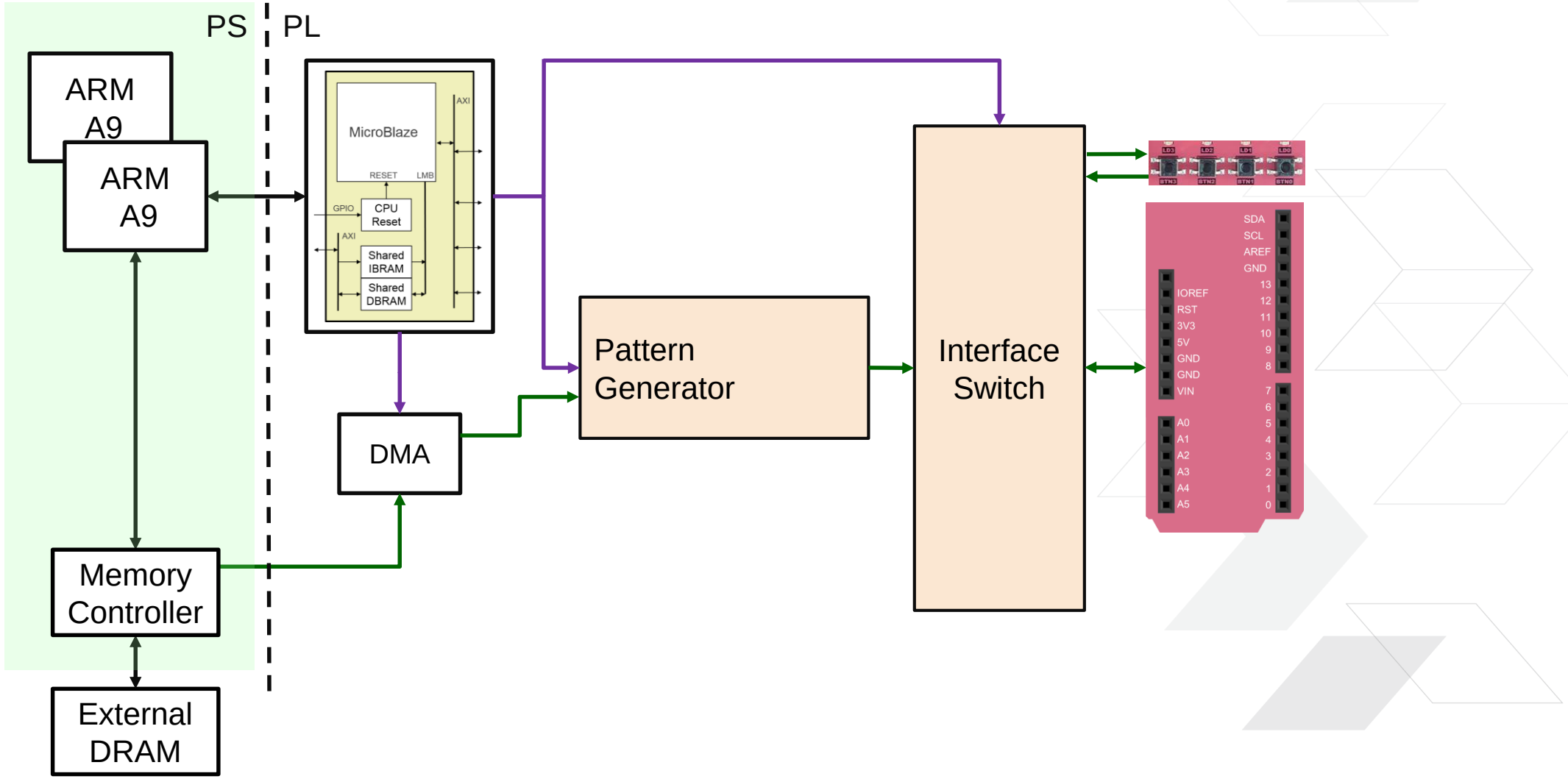
```
# Stop generator when finished
boolean_generator.stop()
```

logictools: generator API state diagram

- **The operating sequences for any generator is summarized by the state diagram below:**



Pattern Generator



XILINX.

Wavedrom

- **Wavedrom is an open source JavaScript application for rendering timing waveforms:**

  "WaveDrom draws your Timing Diagram or Waveform from simple textual description.
  It comes with description language, rendering engine and the editor.
  WaveDrom editor works in the browser or can be installed on your system.
  Rendering engine can be embedded into any webpage."

  http://wavedrom.com/

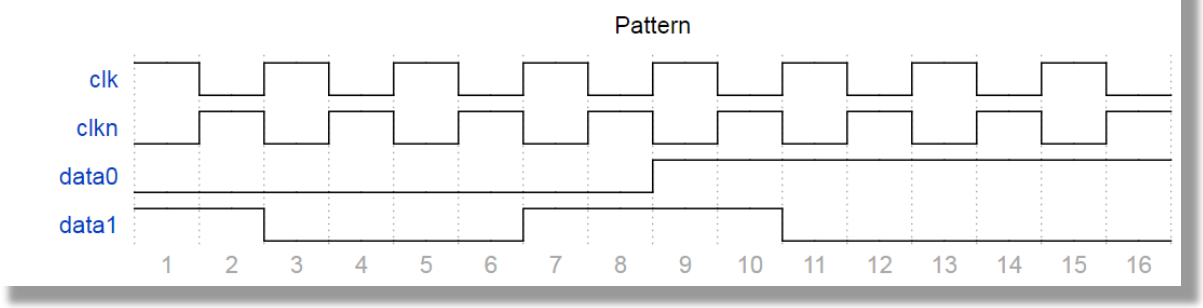
- **We use Wavedrom to show the waveforms of signals captured with the Trace Analyzer**
- **We have extended the Wavedrom description language to specify signals that we want to generate, using the Pattern Generator**

Sample waveform description and rendering

- **Timing waveforms are specified as WaveJSON strings and rendered with the draw_waveform() method in PYNQ**

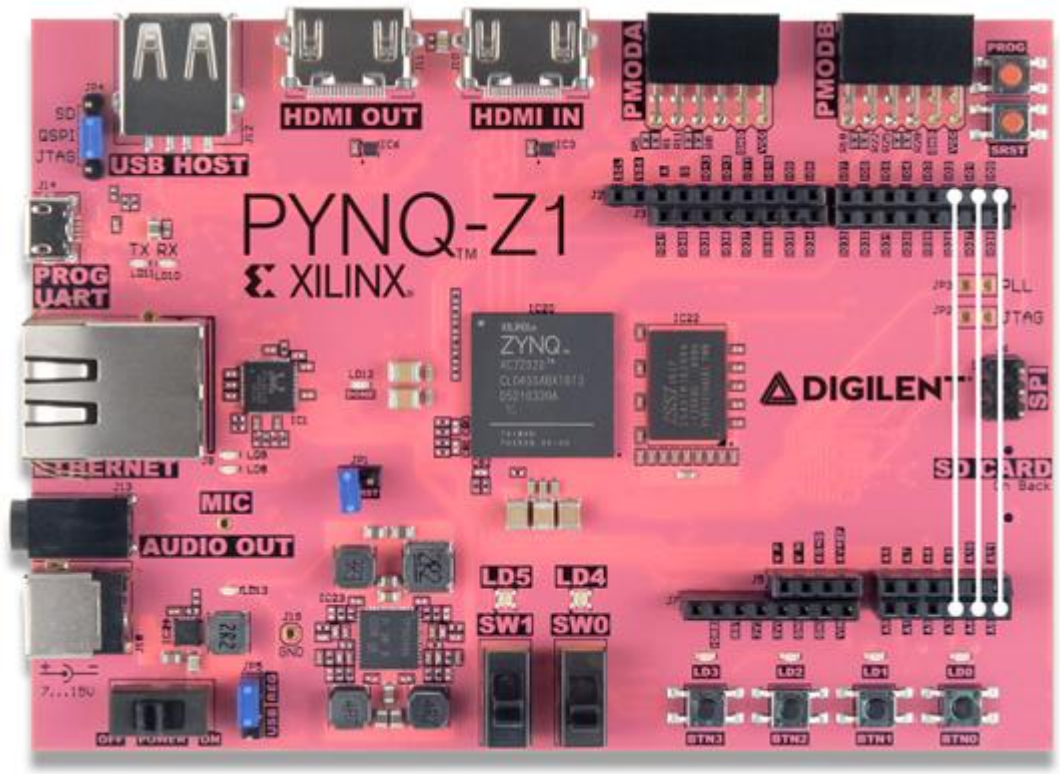
```
from pynq.lib.logictools.waveform import draw_wavedrom

pattern = {'signal': [{'name': 'clk', 'wave': 'hl' * 8},
                      {'name': 'clkn', 'wave': 'lh' * 8},
                      {'name': 'data0', 'wave': 'l.......h.......'},
                      {'name': 'data1', 'wave': 'h.l...h...l.....'}],
           'head': {'text': 'Pattern'},
           'foot': {'tock': 1}}

draw_wavedrom(pattern)
```

WaveJSON strings



- **The WaveJSON format is defined at www.wavedrom.com and explained further in the accompanying notebooks**

Pattern Generator External Loopback Example



← Stimulus: D0, D1, D2

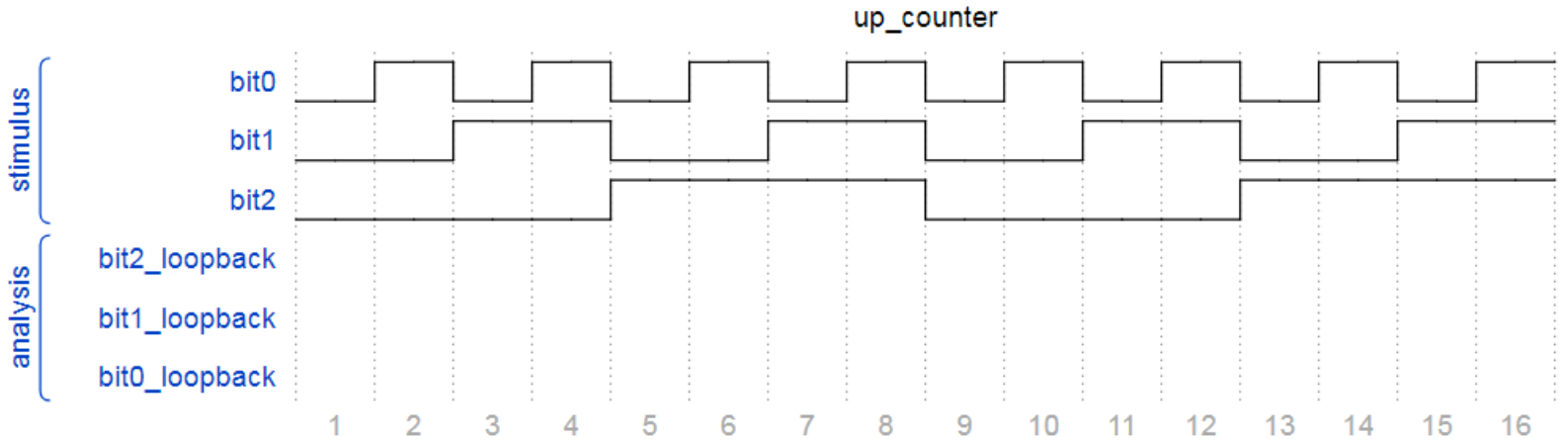
← Response: D17, D18, D19

XILINX

Pattern Generator Example

```
from pynq.overlays.logictools import LogicToolsOverlay
from pynq.lib.logictools import PatternGenerator

logictools_olay = LogicToolsOverlay('logictools.bit')

from pynq.lib.logictools import Waveform

up_counter = {'signal': [
    ['stimulus',
        {'name': 'bit0', 'pin': 'D0', 'wave': 'lh' * 8},
        {'name': 'bit1', 'pin': 'D1', 'wave': 'l.h.' * 4},
        {'name': 'bit2', 'pin': 'D2', 'wave': 'l...h...' * 2}],

    ['analysis',
        {'name': 'bit2_loopback', 'pin': 'D17'},
        {'name': 'bit1_loopback', 'pin': 'D18'},
        {'name': 'bit0_loopback', 'pin': 'D19'}]],

    'foot': {'tock': 1},
    'head': {'text': 'up_counter'}}

waveform = Waveform(up_counter)
waveform.display()
```

Displaying the stimulus waveforms



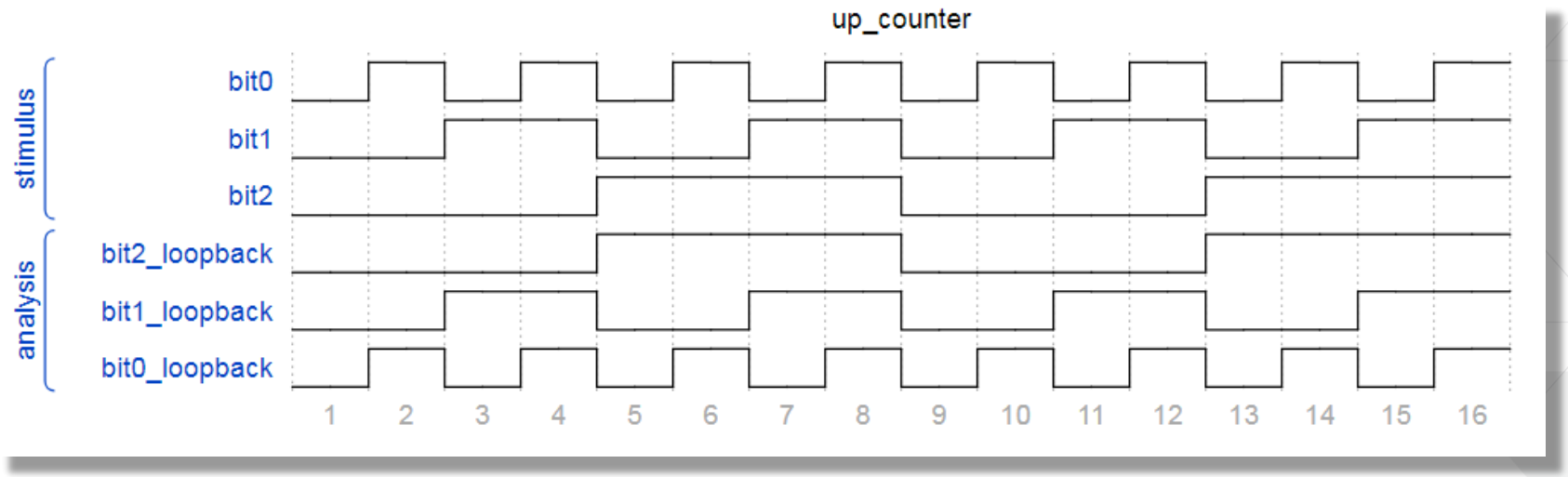
WaveDrom Extensions

```
from pynq.overlays.logictools import LogicToolsOverlay
from pynq.lib.logictools import PatternGenerator

logictools_olay = LogicToolsOverlay('logictools.bit')

from pynq.lib.logictools import Waveform

up_counter = {'signal': [
    ['stimulus',
        {'name': 'bit0', 'pin': 'D0', 'wave': 'lh' * 8},
        {'name': 'bit1', 'pin': 'D1', 'wave': 'l.h.' * 4},
        {'name': 'bit2', 'pin': 'D2', 'wave': 'l...h...' * 2}],

    ['analysis',
        {'name': 'bit2_loopback', 'pin': 'D17'},
        {'name': 'bit1_loopback', 'pin': 'D18'},
        {'name': 'bit0_loopback', 'pin': 'D19'}]],

    'foot': {'tock': 1},
    'head': {'text': 'up_counter'}}

waveform = Waveform(up_counter)
waveform.display()
```

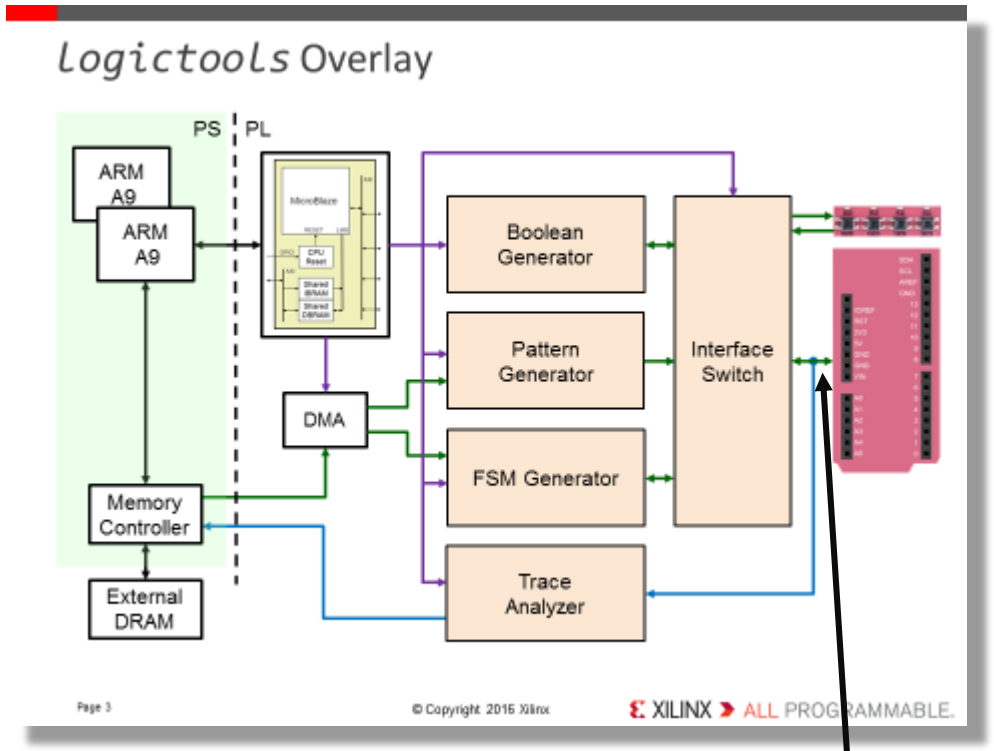
Setting up and running the pattern generator

```
pattern_generator = logictools_olay.pattern_generator
pattern_generator.trace(num_analyzer_samples=16)

pattern_generator.setup(up_counter,
                        stimulus_group_name='stimulus',
                        analysis_group_name='analysis')

pattern_generator.run()
pattern_generator.show_waveform()

# Stop the pattern generator when finished
pattern_generator.stop()
```

Displaying the stimulus and analysis waveforms



Internal Signal Loopback



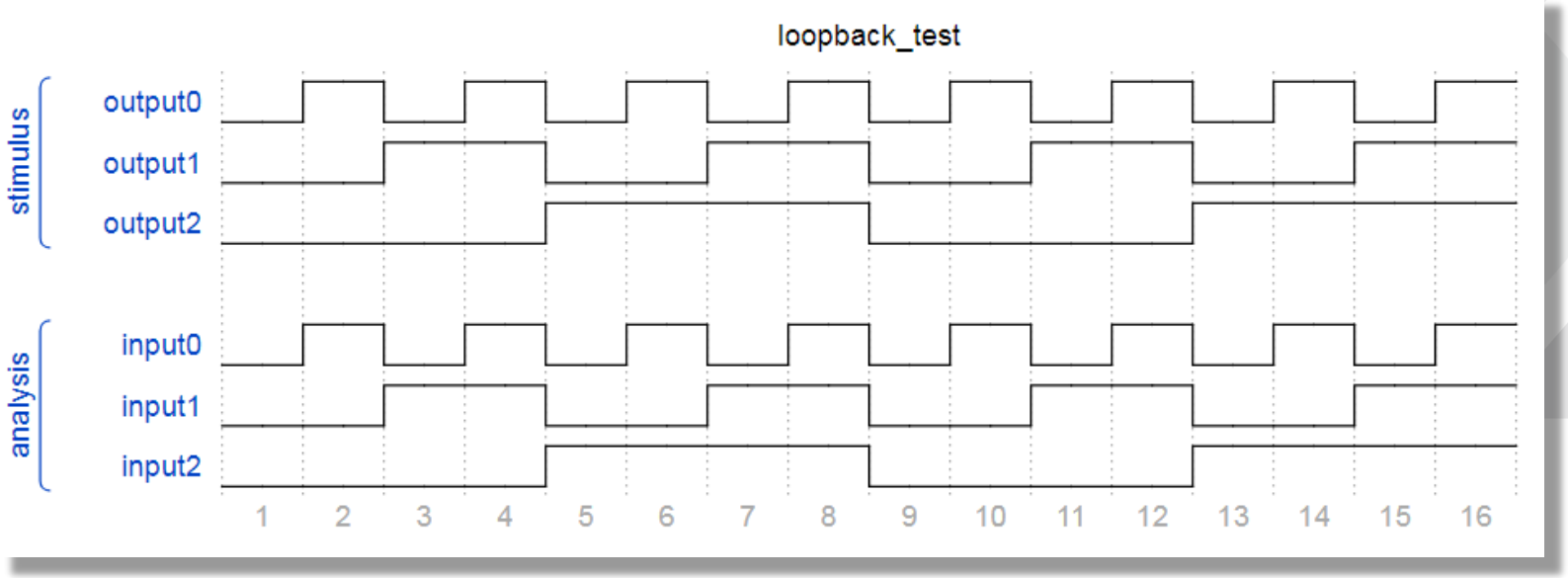
Internal visibility of all IO signals

So we can monitor stimulus signals without external loopback wires

XILINX.

Internal loopback simplifies analysis group spec

```
loopback_test = {'signal': [
    ['stimulus',
     {'name': 'output0', 'pin': 'D0', 'wave': 'lh' * 8},
     {'name': 'output1', 'pin': 'D1', 'wave': 'l.h.' * 4},
     {'name': 'output2', 'pin': 'D2', 'wave': 'l...h...' * 2}],
    {},
    ['analysis',
     {'name': 'input0', 'pin': 'D0'},
     {'name': 'input1', 'pin': 'D1'},
     {'name': 'input2', 'pin': 'D2'}]],

    'foot': {'tock': 1},
    'head': {'text': 'loopback_test'}}
```



XILINX.

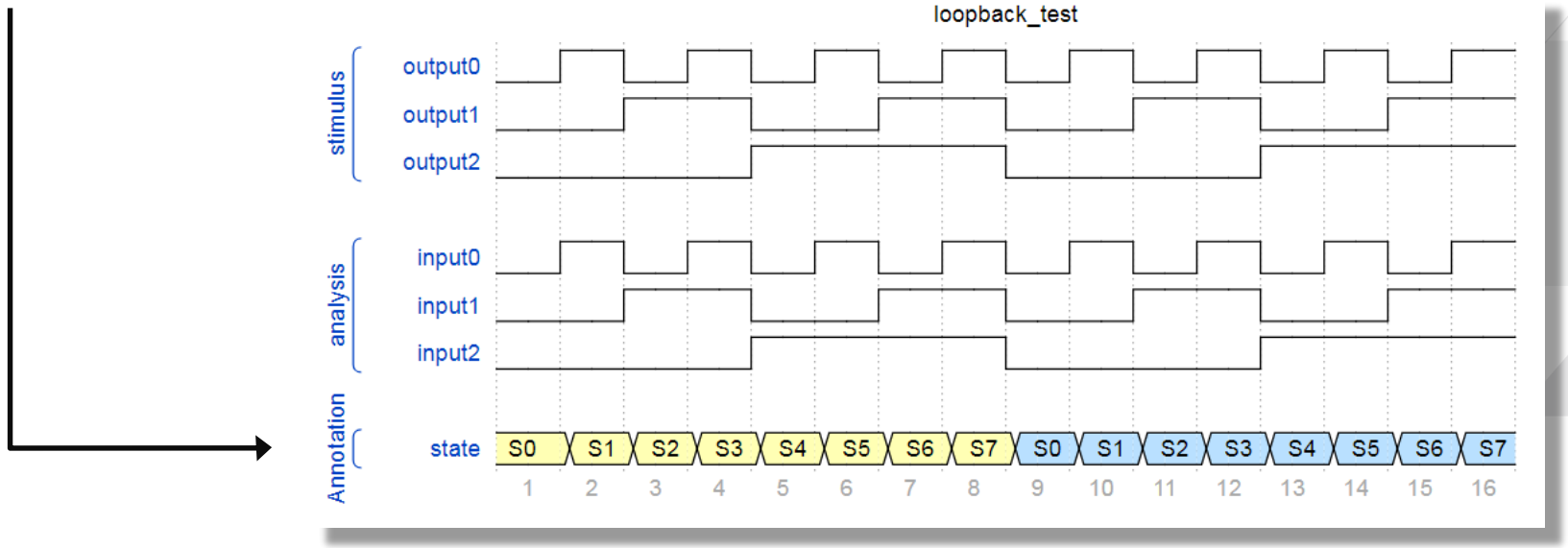
WaveJSON of captured waveforms

```
import pprint

output_wavejson = pattern_generator.waveform.waveform_dict
pprint.pprint(output_wavejson)
```

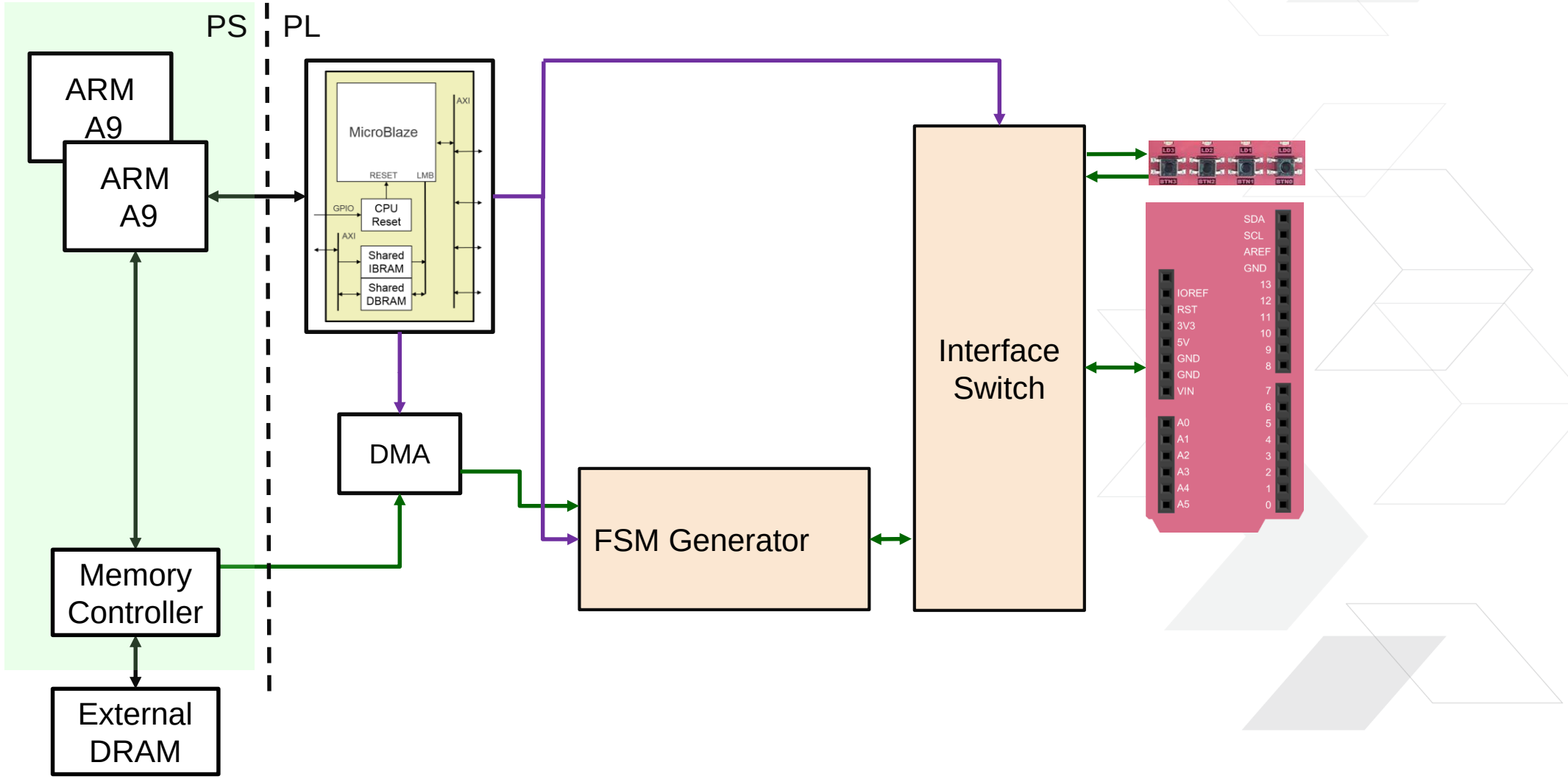
```
{'foot': {'tock': 1},
 'head': {'text': 'loopback_test'},
 'signal': [['stimulus',
             {'name': 'output0', 'pin': 'D0', 'wave': 'lhlhlhlhlhlhlhlh'},
             {'name': 'output1', 'pin': 'D1', 'wave': 'l.h.l.h.l.h.l.h.'},
             {'name': 'output2', 'pin': 'D2', 'wave': 'l...h...l...h...'}],
            {},
            ['analysis',
             {'name': 'input0', 'pin': 'D0', 'wave': 'lhlhlhlhlhlhlhlh'},
             {'name': 'input1', 'pin': 'D1', 'wave': 'l.h.l.h.l.h.l.h.'},
             {'name': 'input2', 'pin': 'D2', 'wave': 'l...h...l...h...'}]]}
```

Annotating captured waveforms

```
state_list = ['S0', 'S1', 'S2', 'S3', 'S4', 'S5', 'S6', 'S7',
              'S0', 'S1', 'S2', 'S3', 'S4', 'S5', 'S6', 'S7']

color_dict = {'white': '2', 'yellow': '3', 'orange': '4', 'blue': '5'}

output_wavejson['signal'].extend([{}, ['Annotation',
                                       {'name': 'state',
                                        'wave': color_dict['yellow'] * 8 +
                                                color_dict['blue'] * 8,
                                        'data': state_list}]])

draw_wavedrom(output_wavejson)
```



FSM Generator



XILINX.

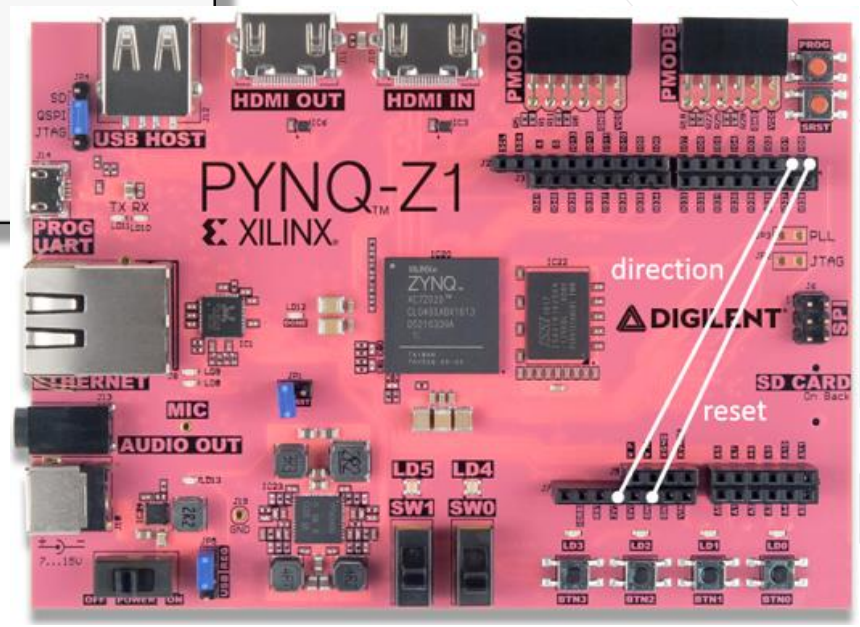
FSM Generator Max Configurations

Max # Inputs	Max # States	Max # Outputs
8	31	12
7	63	13
6	127	14
5	255	15
4	511	16

- For example, we can specify a FSM with up to 127 states, up to 6 inputs, and up to 14 outputs
- Since there are only 20 GPIO pins on an Arduino shield, the total number of inputs and outputs cannot exceed 20
- The maximum number of inputs is 8, and the maximum number of outputs is 19

FSM Generator Example

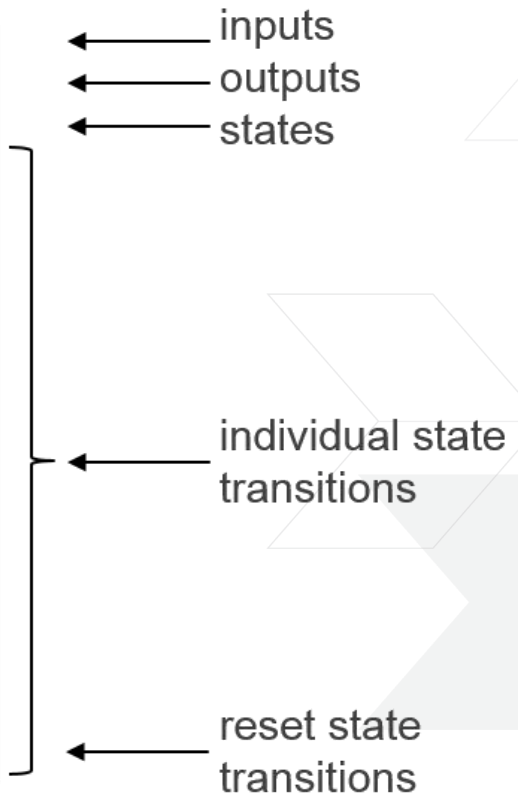
```
fsm_spec = {'inputs': [('reset','D0'), ('direction','D1')],
            'outputs': [('bit2','D3'), ('bit1','D4'), ('bit0','D5')],
            'states': ['S0', 'S1', 'S2', 'S3', 'S4', 'S5', 'S6', 'S7'],
            'transitions': [['01', 'S0', 'S1', '000'],
                            ['00', 'S0', 'S7', '000'],
                            ['01', 'S1', 'S2', '001'],
                            ['00', 'S1', 'S0', '001'],
                            ['01', 'S2', 'S3', '011'],
                            ['00', 'S2', 'S1', '011'],
                            ['01', 'S3', 'S4', '010'],
                            ['00', 'S3', 'S2', '010'],
                            ['01', 'S4', 'S5', '110'],
                            ['00', 'S4', 'S3', '110'],
                            ['01', 'S5', 'S6', '111'],
                            ['00', 'S5', 'S4', '111'],
                            ['01', 'S6', 'S7', '101'],
                            ['00', 'S6', 'S5', '101'],
                            ['01', 'S7', 'S0', '100'],
                            ['00', 'S7', 'S6', '100'],
                            ['1-', '*',  'S0', '']]}
```



- ‘reset’ is tied low (inactive)
- ‘direction’ is tied high
  - implying an ‘up’ count

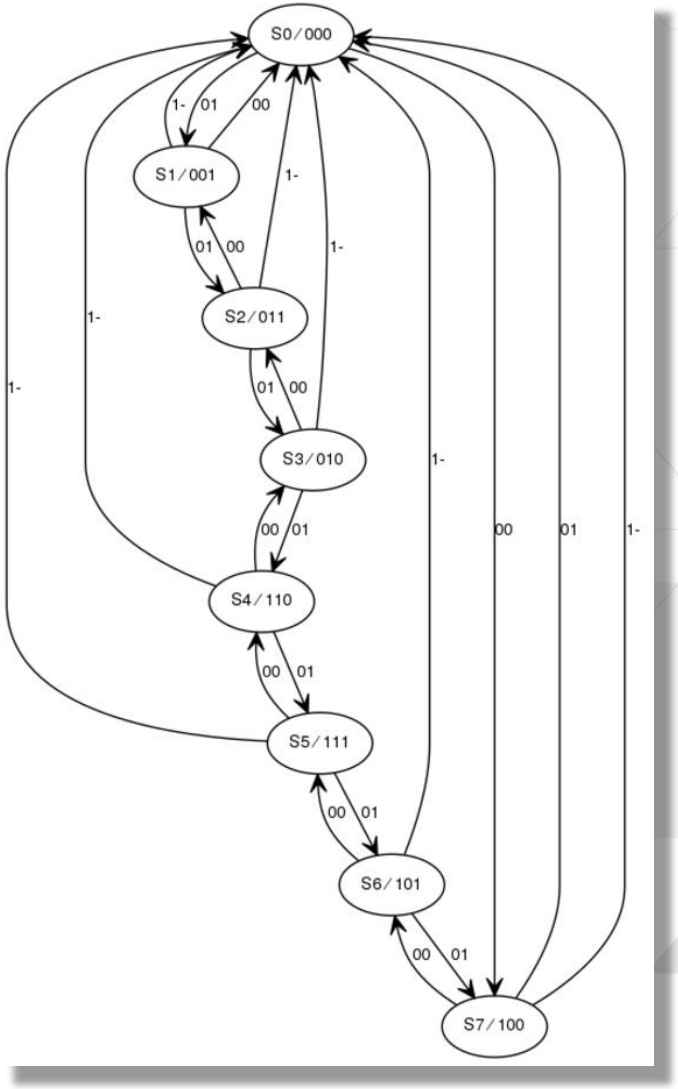
Specifying a 3-bit, up/down, Gray code counter

```
fsm_spec = {'inputs': [('reset','D0'), ('direction','D1')],
            'outputs': [('bit2','D3'), ('bit1','D4'), ('bit0','D5')],
            'states': ['S0', 'S1', 'S2', 'S3', 'S4', 'S5', 'S6', 'S7'],
            'transitions': [['01', 'S0', 'S1', '000'],
                            ['00', 'S0', 'S7', '000'],
                            ['01', 'S1', 'S2', '001'],
                            ['00', 'S1', 'S0', '001'],
                            ['01', 'S2', 'S3', '011'],
                            ['00', 'S2', 'S1', '011'],
                            ['01', 'S3', 'S4', '010'],
                            ['00', 'S3', 'S2', '010'],
                            ['01', 'S4', 'S5', '110'],
                            ['00', 'S4', 'S3', '110'],
                            ['01', 'S5', 'S6', '111'],
                            ['00', 'S5', 'S4', '111'],
                            ['01', 'S6', 'S7', '101'],
                            ['00', 'S6', 'S5', '101'],
                            ['01', 'S7', 'S0', '100'],
                            ['00', 'S7', 'S6', '100'],
                            ['1-', '*',  'S0', '']]}
```



Showing the state diagram for a FSM

```
fsm_generator.setup(fsm_spec)

fsm_generator.show_state_diagram()
```

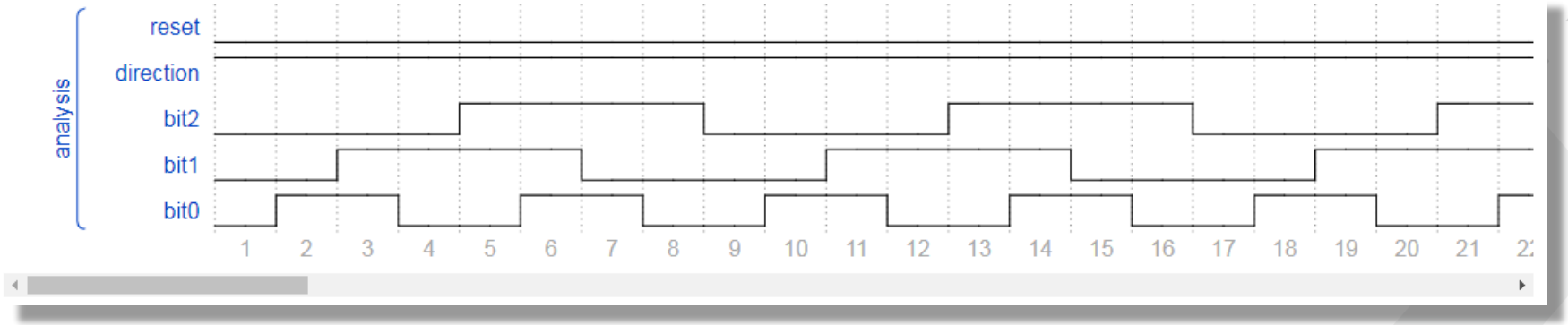


3-bit Gray code counter state diagram

Automatically generated from FSM specification

Running FSM and displaying the waveform

```
fsm_generator.run()
fsm_generator.show_waveform()
```



Generator API command summary

BooleanGenerator

- reset()
- run()
- setup()
- show_waveform()
- status()
- step()
- stop()
- trace()

PatternGenerator

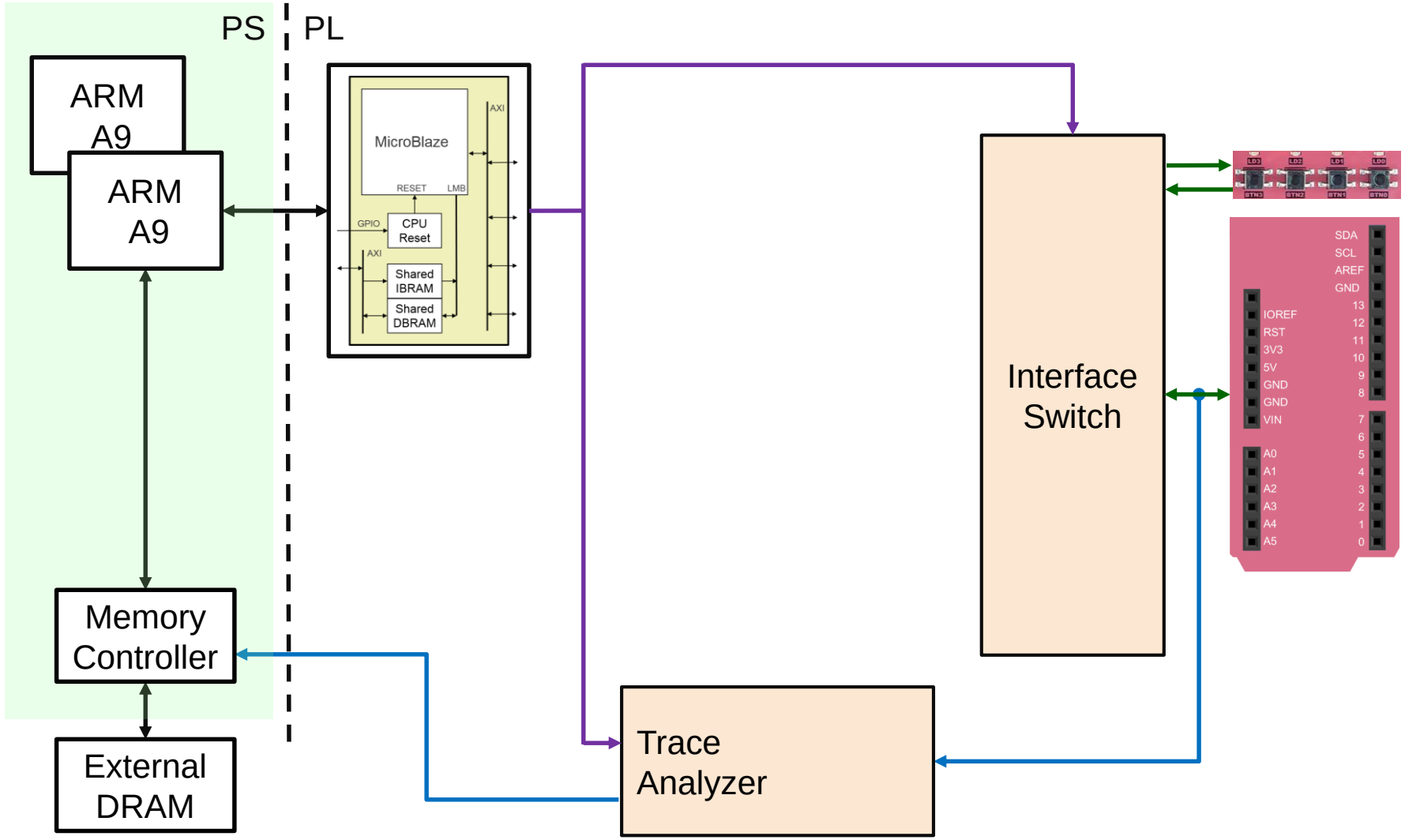
- reset()
- run()
- setup()
- show_waveform()
- status()
- step()
- stop()
- trace()

FSMGenerator

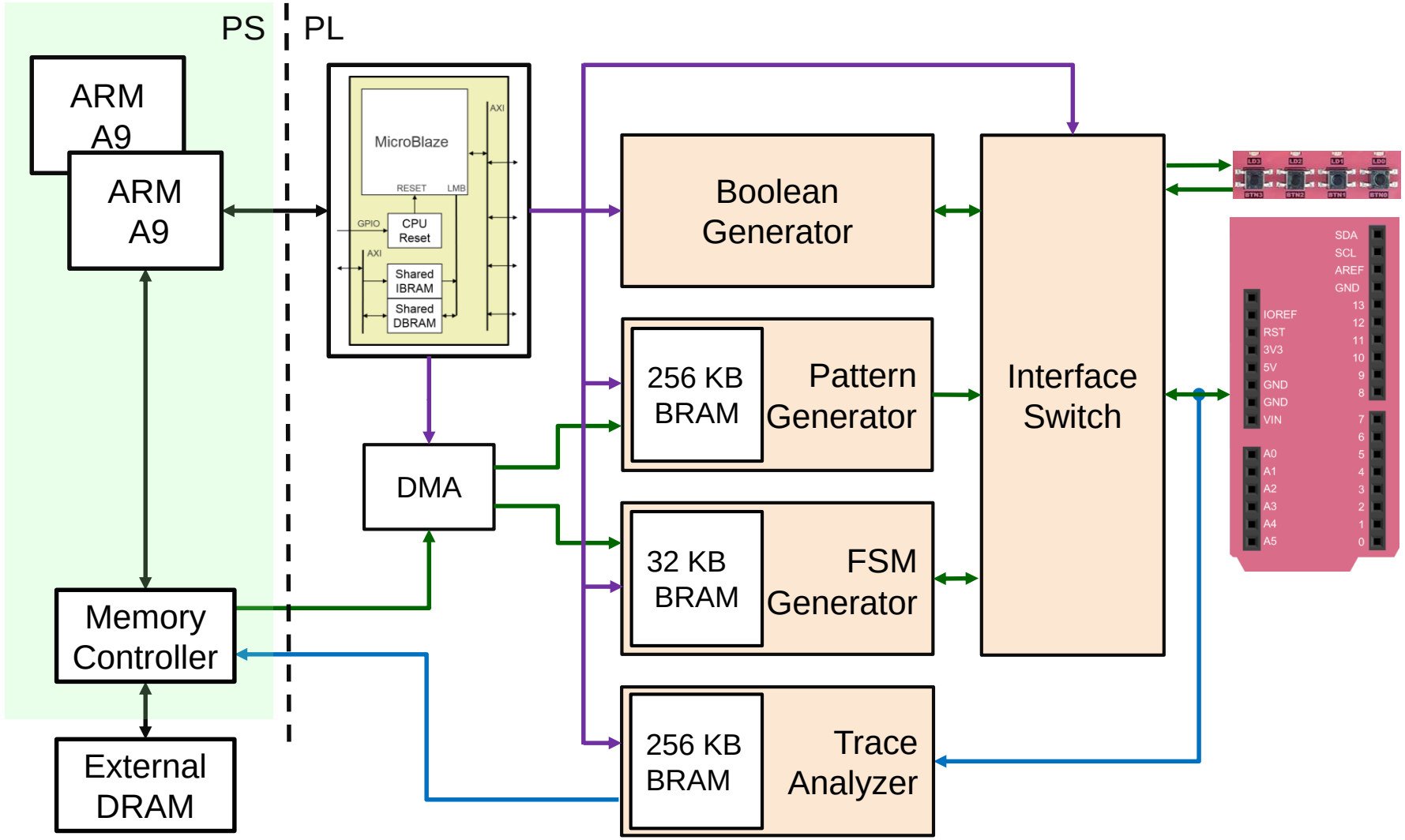
- reset()
- run()
- setup()
- show_waveform()
- status()
- step()
- stop()
- trace()

- show_state_diagram()

Trace Analyzer

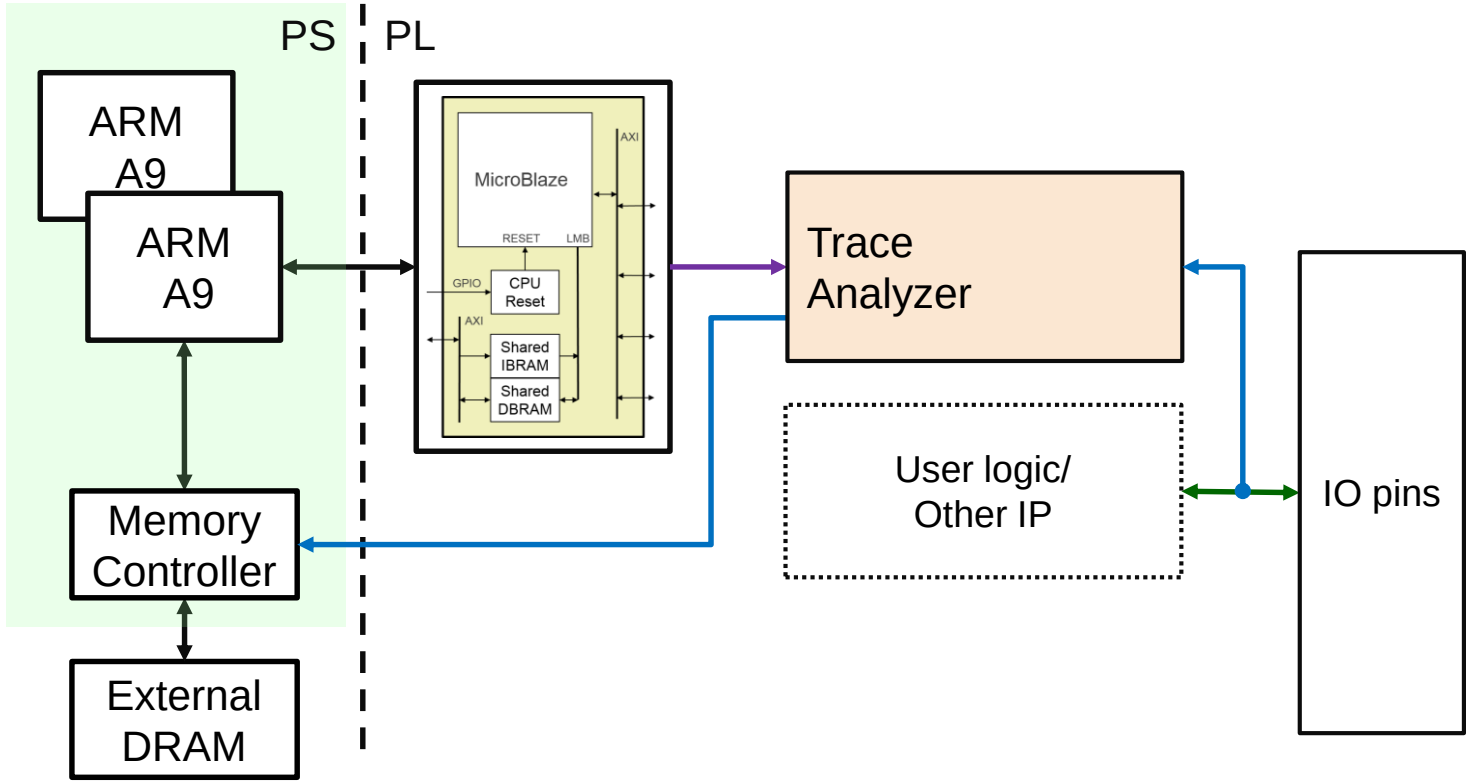


Pynq-Z1 *logictools* BRAM allocation



XILINX.

Trace Analyzer used independently



logictools: software-defined instrumentation

- **Controlled from web browser**
- **Declarative**
  - Specify the functionality that you want, and where you want it connected
    - Boolean functions, state machines, digital patterns, trace analysis
- **Instantaneous realization**
  - *logictools* creates it for you … instantaneously
- **Reusable software and hardware libraries**
  - Library of novel IP available for reuse by overlay developers
    - Useful also to non-PYNQ users

Adaptable.
Intelligent.

XILINX

PYNQ