Amaranth (2, lecture)

Hardware System Design

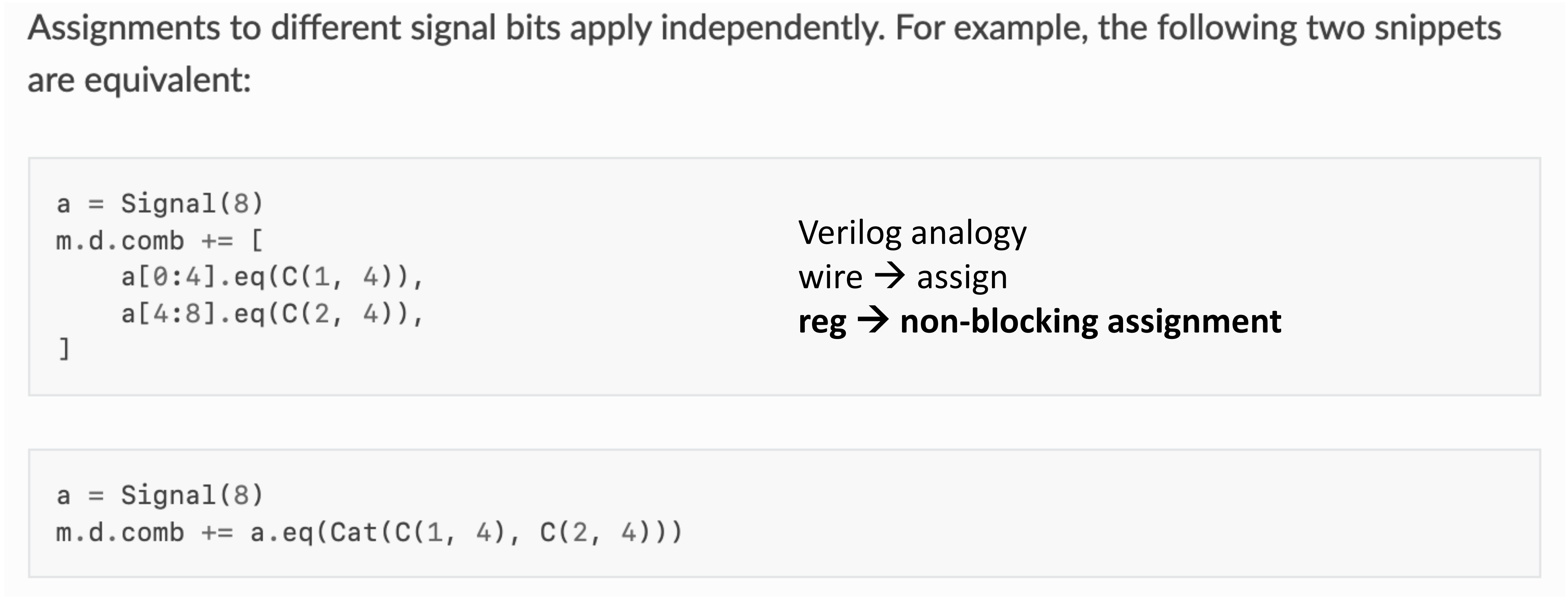
Spring, 2023

Amaranth examples

- Assignment order
- Memory
- FIFO
- FSM

- Tutorial code
  https://www.notion.so/tutorial-code-f5ba421763394b56968822fc6af7a7f0?pvs=4

Assignment order



Assignments to different signal bits apply independently. For example, the following two snippets are equivalent:

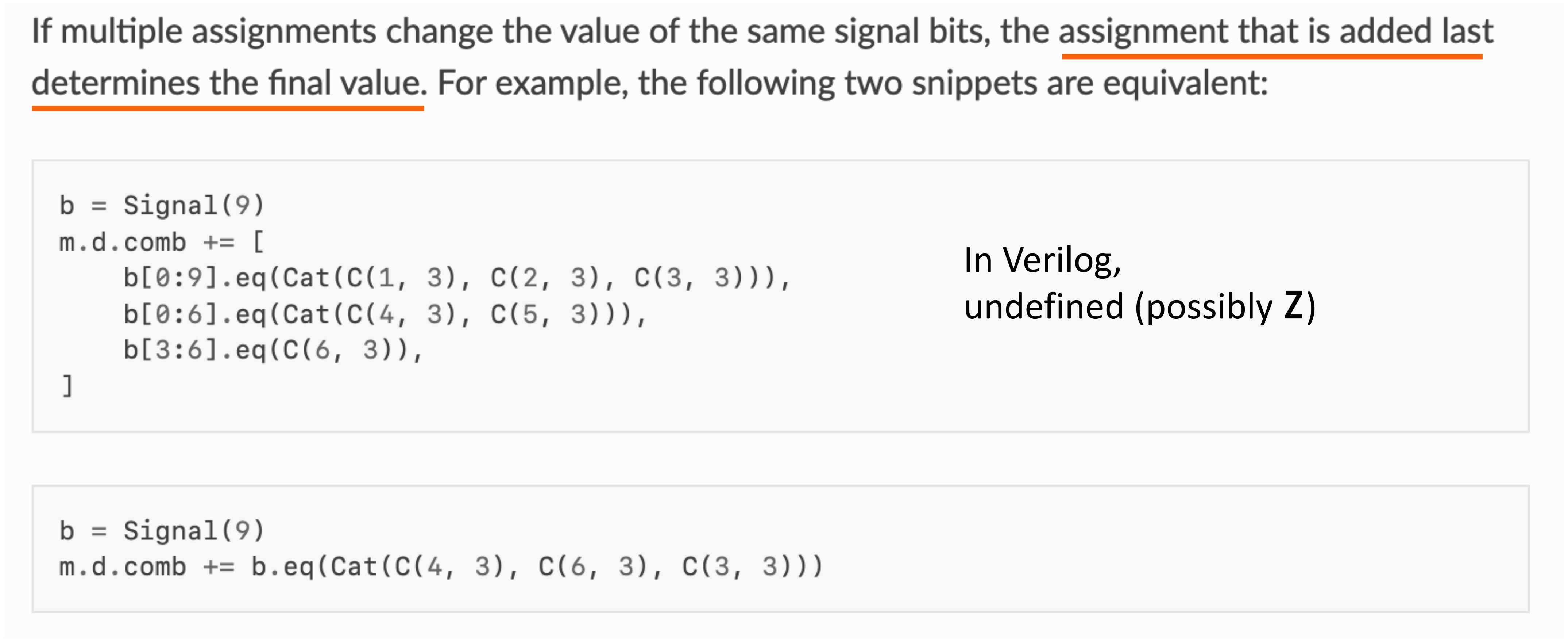
```
a = Signal(8)
m.d.comb += [
    a[0:4].eq(C(1, 4)),
    a[4:8].eq(C(2, 4)),
]
```

Verilog analogy
wire → assign
reg → non-blocking assignment

```
a = Signal(8)
m.d.comb += a.eq(Cat(C(1, 4), C(2, 4)))
```

[Language guide — Assignment order (amaranth-lang.org)](amaranth-lang.org)

Assignment order



If multiple assignments change the value of the same signal bits, the assignment that is added last determines the final value. For example, the following two snippets are equivalent:

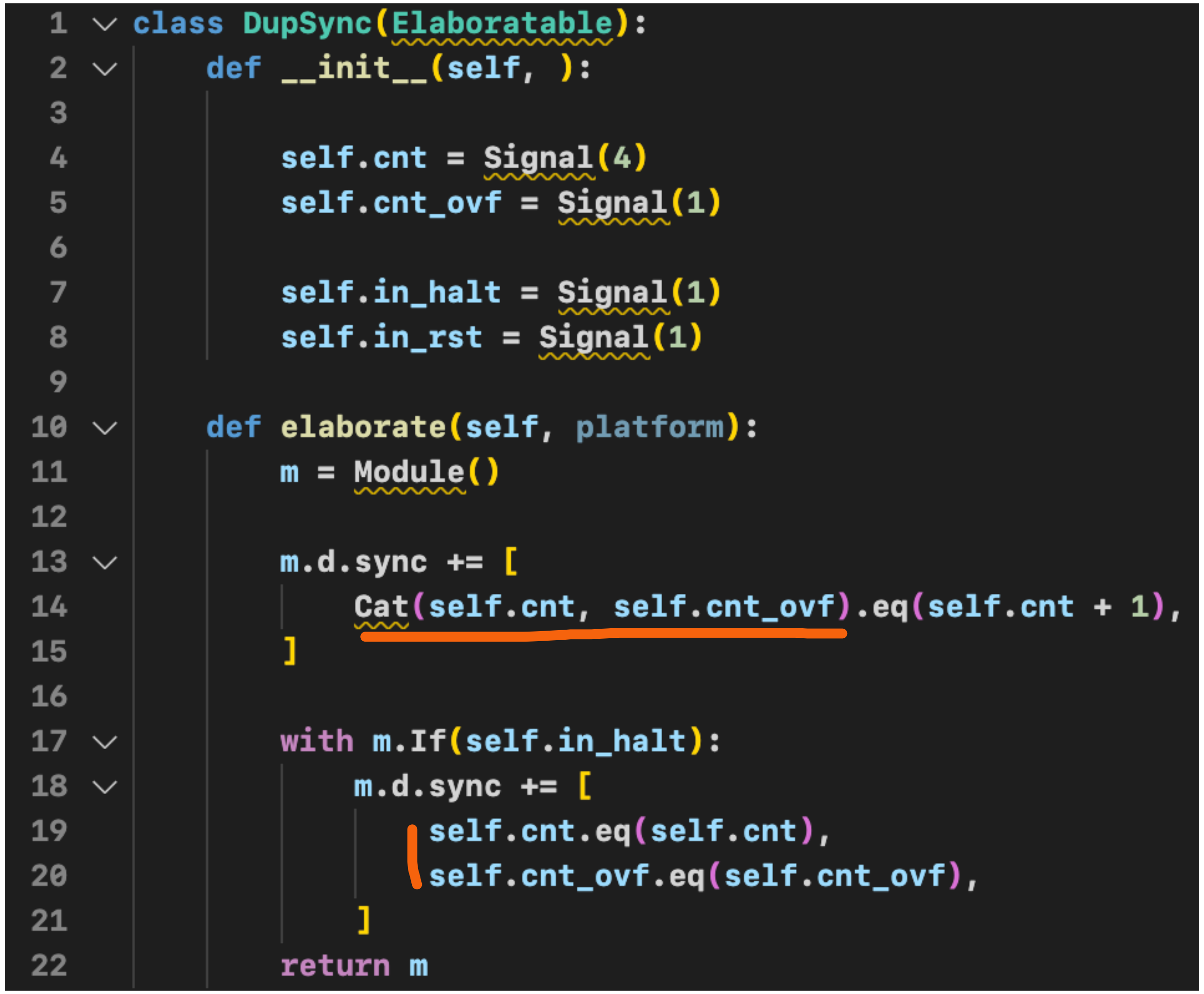
```
b = Signal(9)
m.d.comb += [
    b[0:9].eq(Cat(C(1, 3), C(2, 3), C(3, 3))),
    b[0:6].eq(Cat(C(4, 3), C(5, 3))),
    b[3:6].eq(C(6, 3)),
]
```

In Verilog,
undefined (possibly **Z**)

```
b = Signal(9)
m.d.comb += b.eq(Cat(C(4, 3), C(6, 3), C(3, 3)))
```

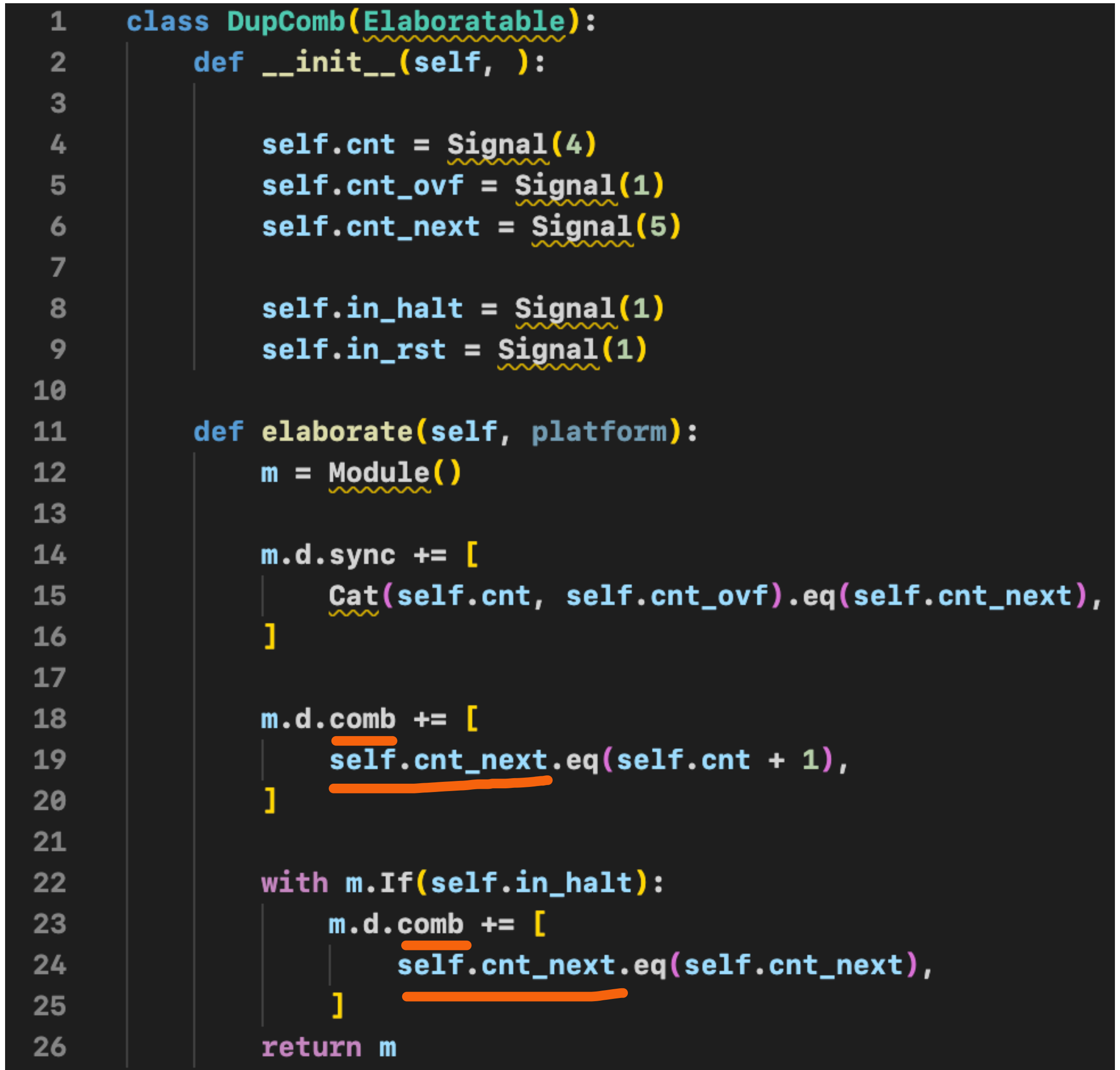
Language guide — Assignment order (amaranth-lang.org)

Assignment order



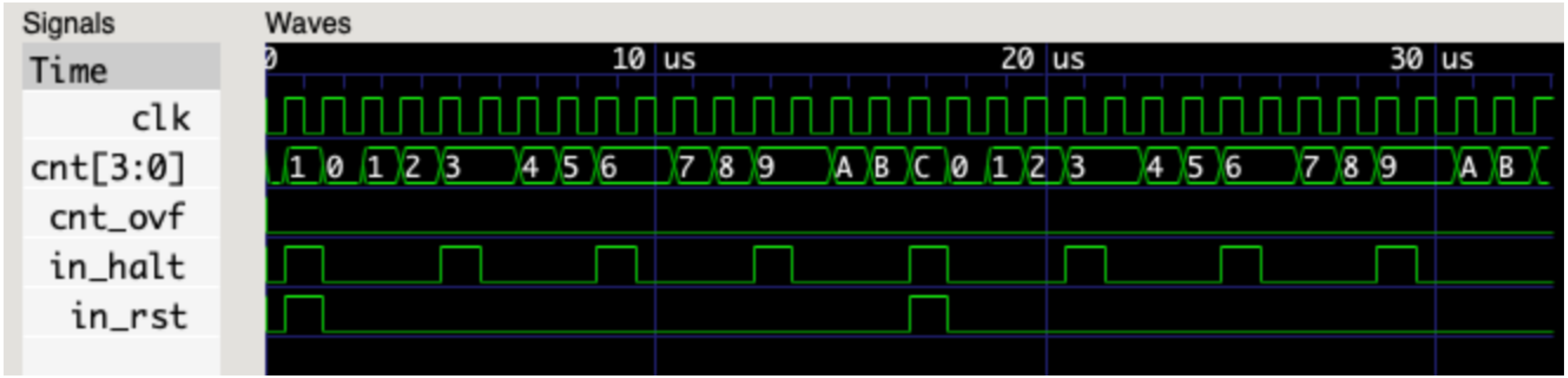
```python
class DupSync(Elaboratable):
    def __init__(self, ):

        self.cnt = Signal(4)
        self.cnt_ovf = Signal(1)

        self.in_halt = Signal(1)
        self.in_rst = Signal(1)

    def elaborate(self, platform):
        m = Module()

        m.d.sync += [
            Cat(self.cnt, self.cnt_ovf).eq(self.cnt + 1),
        ]

        with m.If(self.in_halt):
            m.d.sync += [
                self.cnt.eq(self.cnt),
                self.cnt_ovf.eq(self.cnt_ovf),
            ]
        return m
```

Assignment order

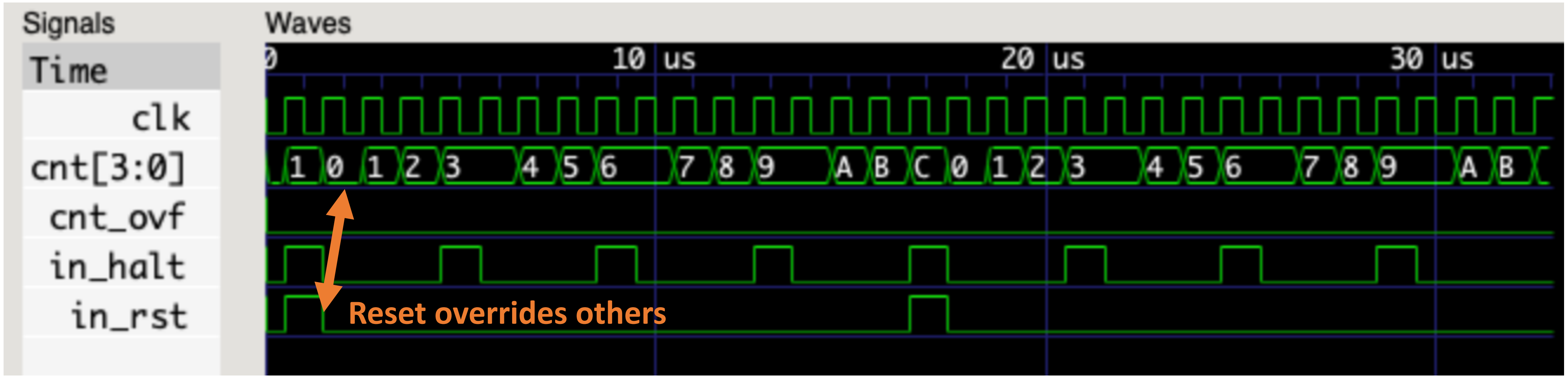


```python
class DupComb(Elaboratable):
    def __init__(self, ):

        self.cnt = Signal(4)
        self.cnt_ovf = Signal(1)
        self.cnt_next = Signal(5)

        self.in_halt = Signal(1)
        self.in_rst = Signal(1)

    def elaborate(self, platform):
        m = Module()

        m.d.sync += [
            Cat(self.cnt, self.cnt_ovf).eq(self.cnt_next),
        ]

        m.d.comb += [
            self.cnt_next.eq(self.cnt + 1),
        ]

        with m.If(self.in_halt):
            m.d.comb += [
                self.cnt_next.eq(self.cnt_next),
            ]
        return m
```

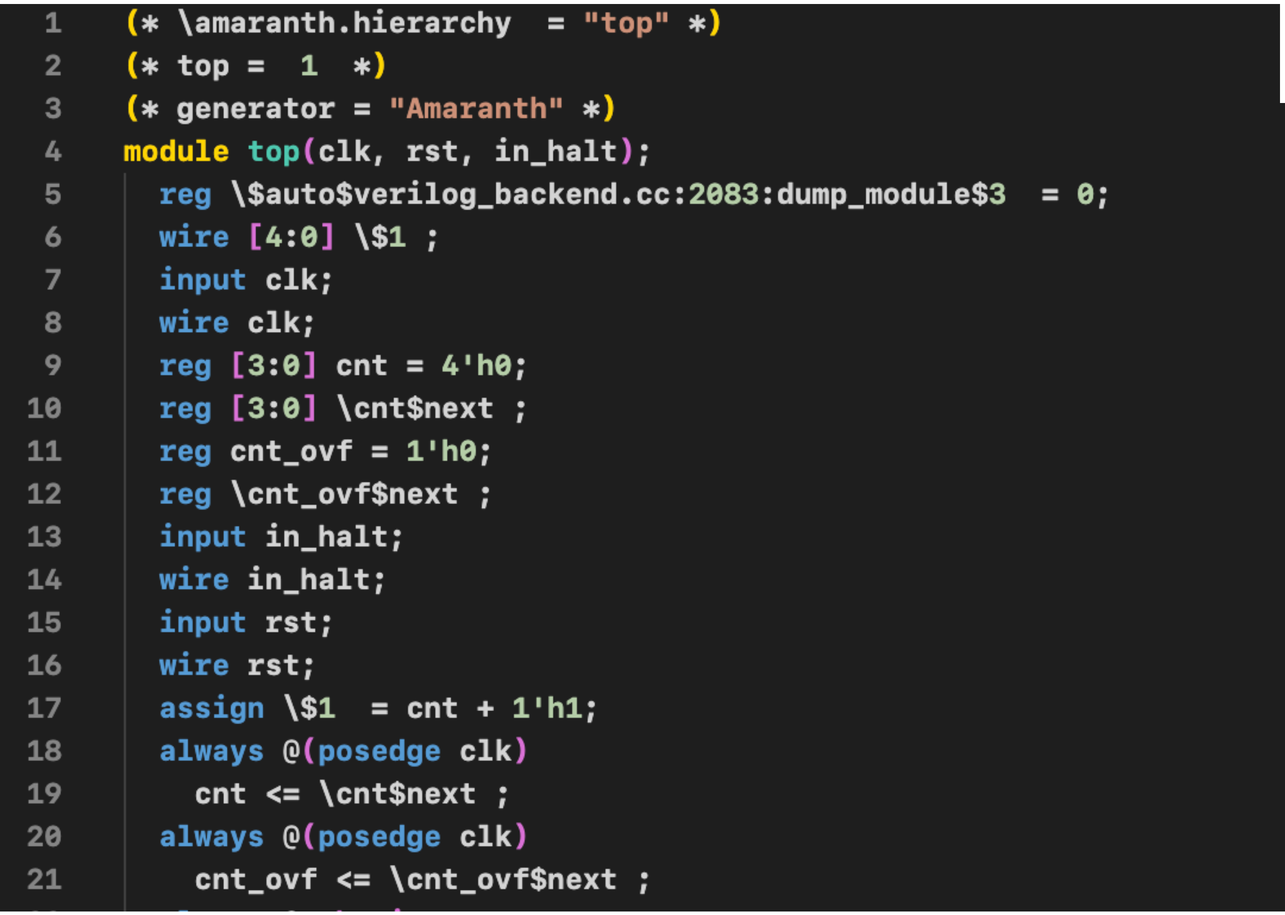
Assignment order



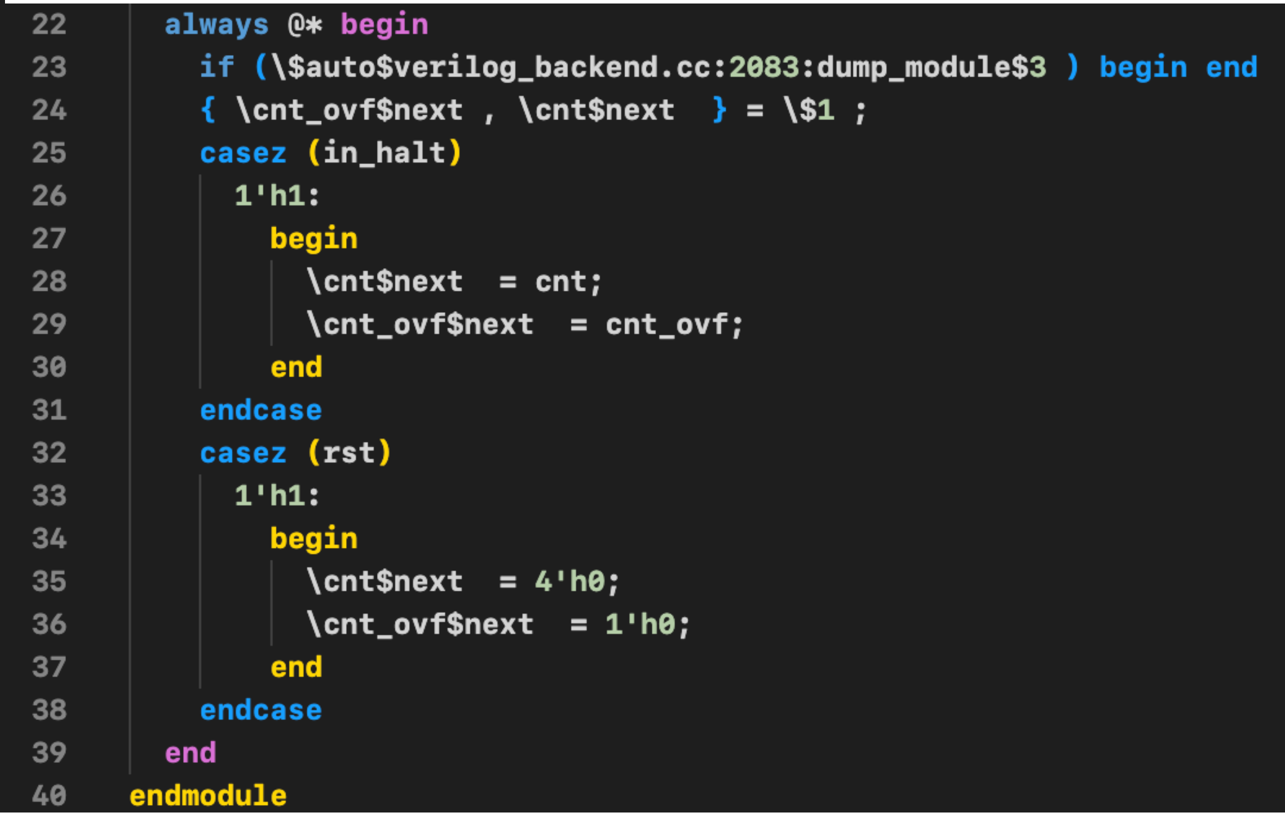
Assignment order



Assignment order

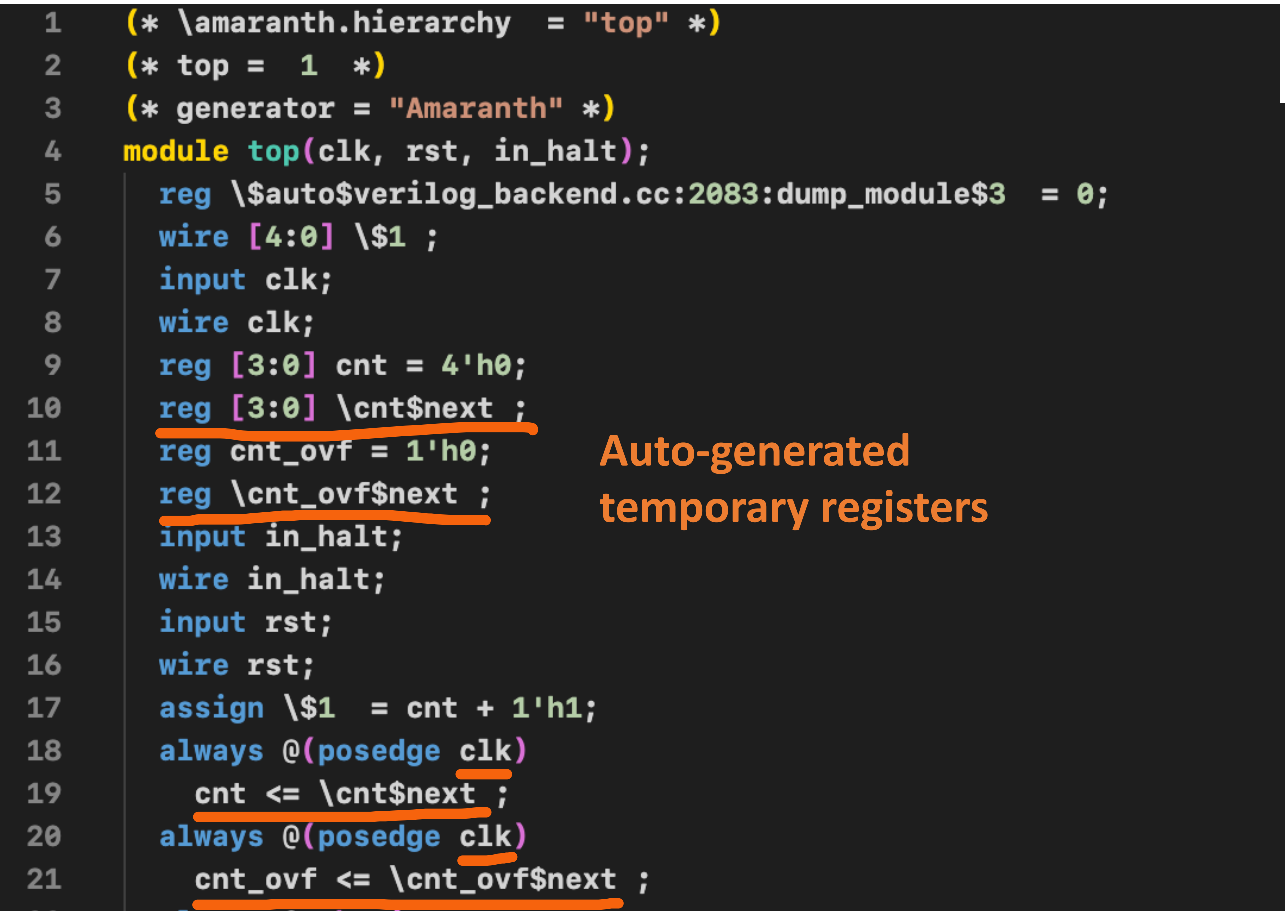


```
(* \amaranth.hierarchy  = "top" *)
(* top =  1  *)
(* generator = "Amaranth" *)
module top(clk, rst, in_halt);
  reg \$auto$verilog_backend.cc:2083:dump_module$3  = 0;
  wire [4:0] \$1 ;
  input clk;
  wire clk;
  reg [3:0] cnt = 4'h0;
  reg [3:0] \cnt$next ;
  reg cnt_ovf = 1'h0;
  reg \cnt_ovf$next ;
  input in_halt;
  wire in_halt;
  input rst;
  wire rst;
  assign \$1  = cnt + 1'h1;
  always @(posedge clk)
    cnt <= \cnt$next ;
  always @(posedge clk)
    cnt_ovf <= \cnt_ovf$next ;
```



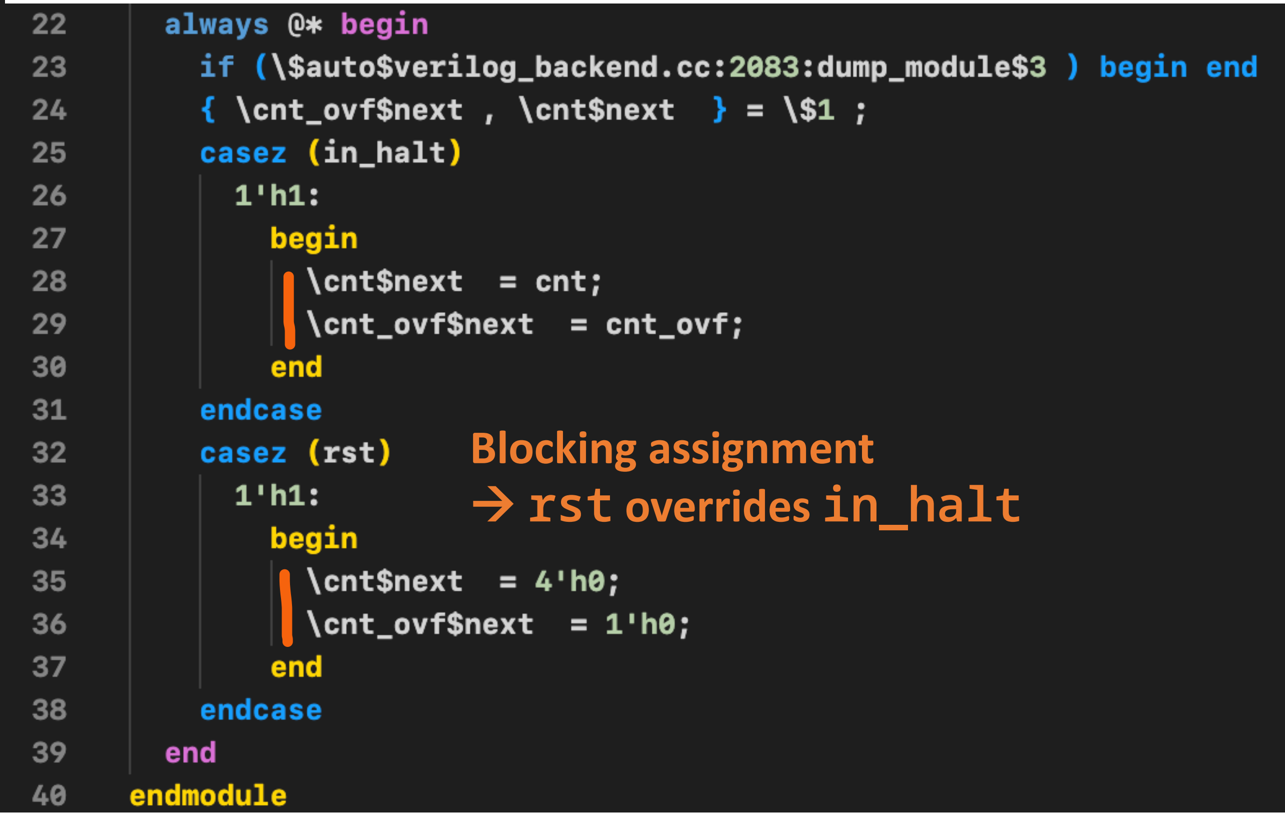
```
  always @* begin
    if (\$auto$verilog_backend.cc:2083:dump_module$3 ) begin end
    { \cnt_ovf$next , \cnt$next  } = \$1 ;
    casez (in_halt)
      1'h1:
        begin
          \cnt$next  = cnt;
          \cnt_ovf$next  = cnt_ovf;
        end
    endcase
    casez (rst)
      1'h1:
        begin
          \cnt$next  = 4'h0;
          \cnt_ovf$next  = 1'h0;
        end
    endcase
  end
endmodule
```

Assignment order



```
(* \amaranth.hierarchy  = "top" *)
(* top =  1  *)
(* generator = "Amaranth" *)
module top(clk, rst, in_halt);
  reg \$auto$verilog_backend.cc:2083:dump_module$3  = 0;
  wire [4:0] \$1 ;
  input clk;
  wire clk;
  reg [3:0] cnt = 4'h0;
  reg [3:0] \cnt$next ;
  reg cnt_ovf = 1'h0;
  reg \cnt_ovf$next ;
  input in_halt;
  wire in_halt;
  input rst;
  wire rst;
  assign \$1  = cnt + 1'h1;
  always @(posedge clk)
    cnt <= \cnt$next ;
  always @(posedge clk)
    cnt_ovf <= \cnt_ovf$next ;
```

Auto-generated temporary registers



```
  always @* begin
    if (\$auto$verilog_backend.cc:2083:dump_module$3 ) begin end
    { \cnt_ovf$next , \cnt$next  } = \$1 ;
    casez (in_halt)
      1'h1:
        begin
          \cnt$next  = cnt;
          \cnt_ovf$next  = cnt_ovf;
        end
    endcase
    casez (rst)
      1'h1:
        begin
          \cnt$next  = 4'h0;
          \cnt_ovf$next  = 1'h0;
        end
    endcase
  end
endmodule
```

Blocking assignment
→ rst overrides in_halt

Q&A on assignment order

Memory

- Memory implementation
  - Verilog → follow guideline & implement your own
  - Amaranth → battery included

- Hardware memory is usually single-port (seldom dual-port)
  - **`reg [31:0] example[0:8191]`** is not

    If your Verilog code use multiple ports

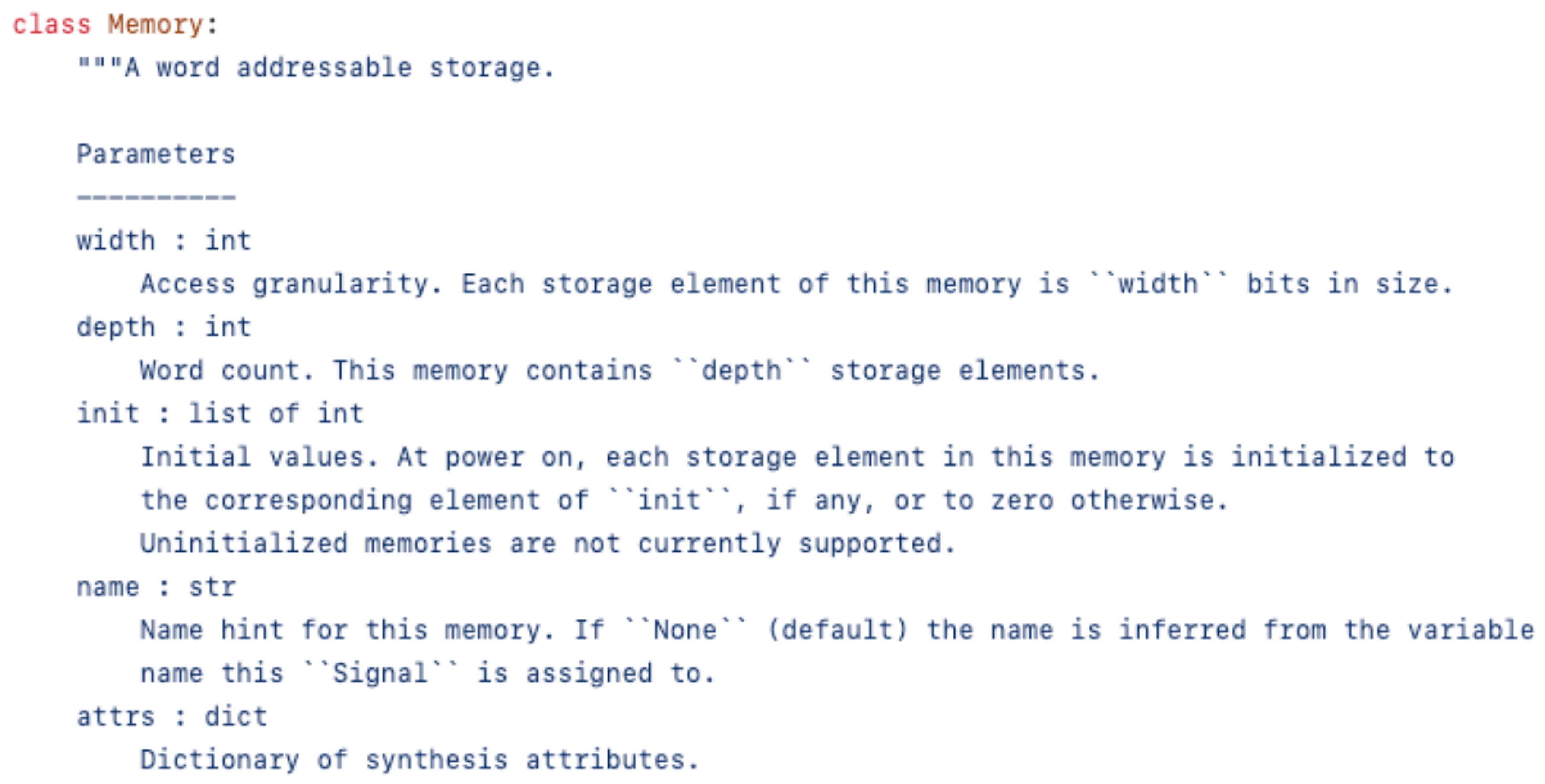
    → memory cannot be mapped to BRAM (physical memory construct)

    This convention is error-prone
  - Much better to abstract memory!

https://docs.xilinx.com/r/en-US/ug901-vivado-synthesis/RAM-HDL-Coding-Guidelines

amaranth/mem.py (github.com)

Memory

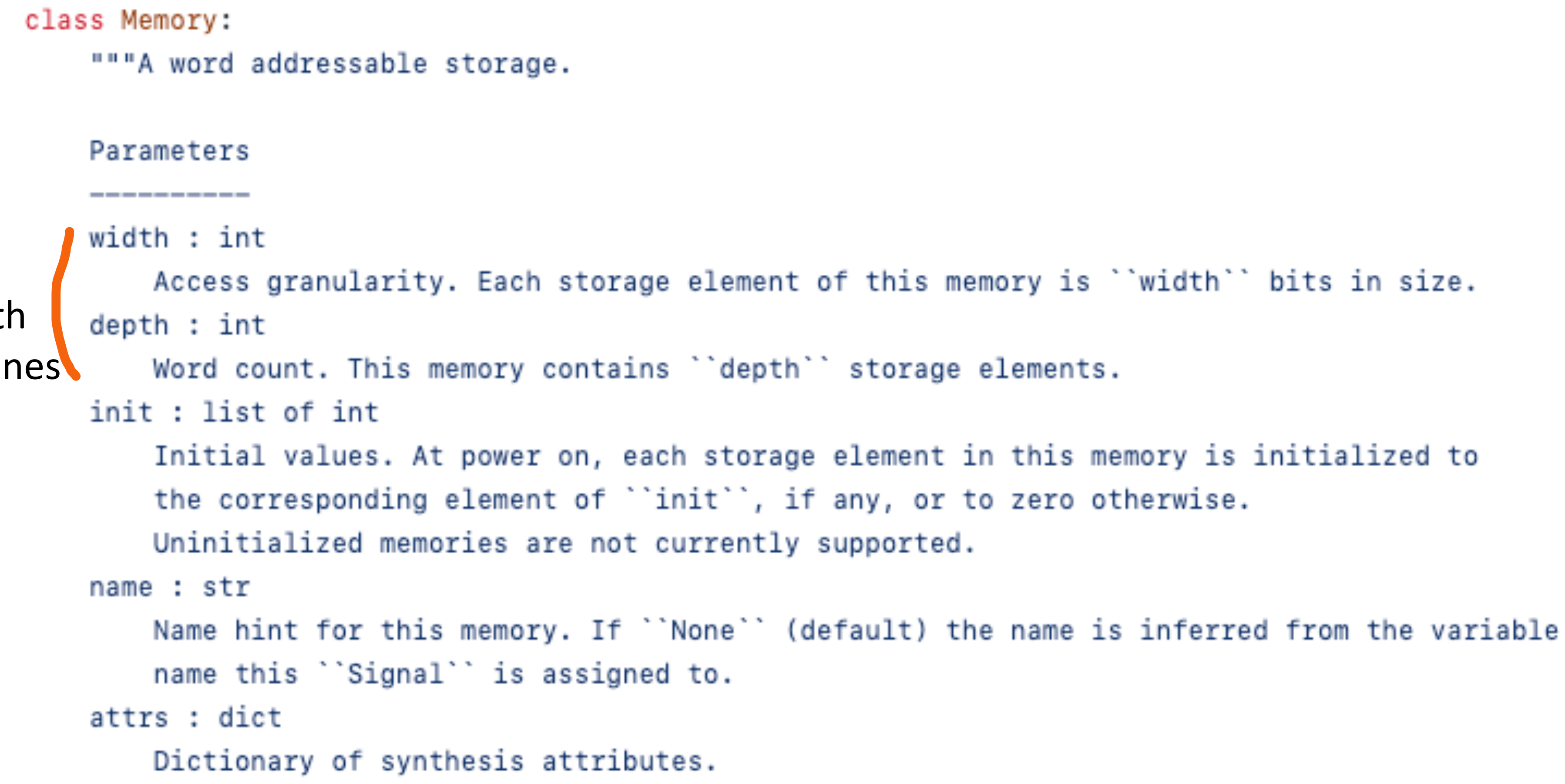


```
class Memory:
    """A word addressable storage.

    Parameters
    ----------
    width : int
        Access granularity. Each storage element of this memory is ``width`` bits in size.
    depth : int
        Word count. This memory contains ``depth`` storage elements.
    init : list of int
        Initial values. At power on, each storage element in this memory is initialized to
        the corresponding element of ``init``, if any, or to zero otherwise.
        Uninitialized memories are not currently supported.
    name : str
        Name hint for this memory. If ``None`` (default) the name is inferred from the variable
        name this ``Signal`` is assigned to.
    attrs : dict
        Dictionary of synthesis attributes.
```

[amaranth/mem.py (github.com)](https://github.com)

Memory

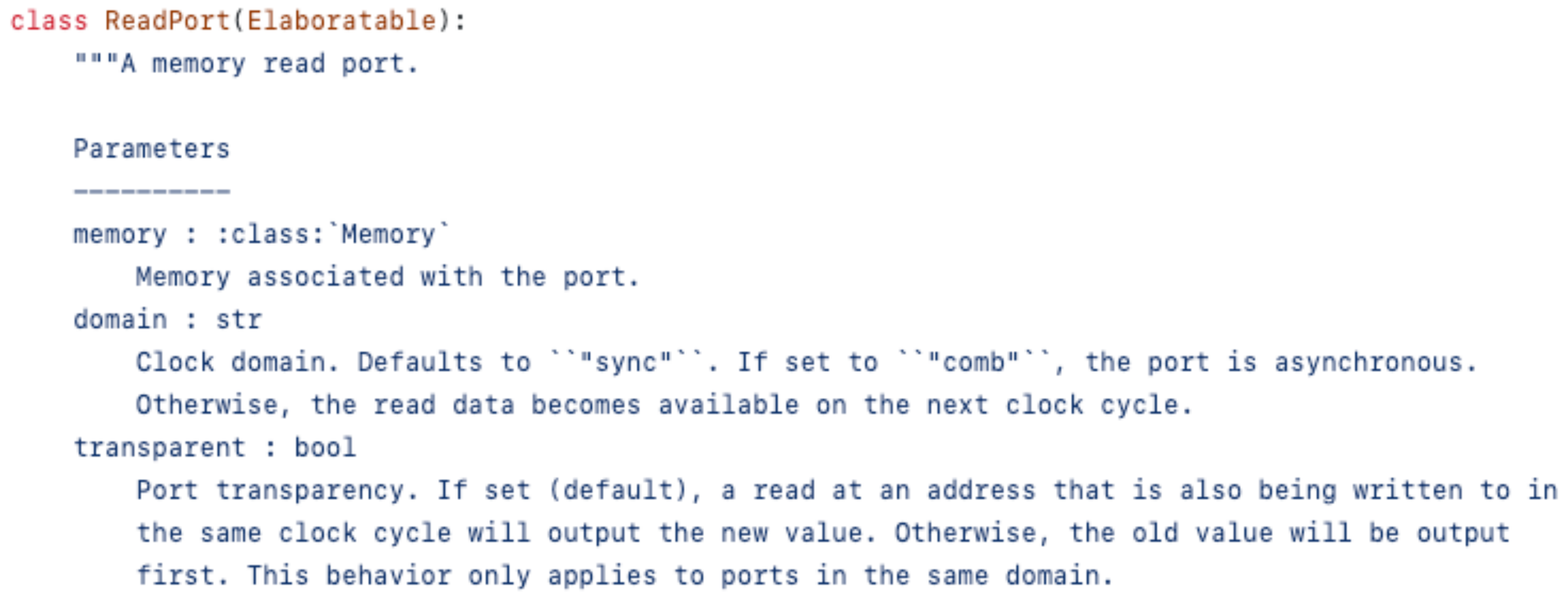
Memory size = width (bits) * depth line * number of lines



```
class Memory:
    """A word addressable storage.

    Parameters
    ----------
    width : int
        Access granularity. Each storage element of this memory is ``width`` bits in size.
    depth : int
        Word count. This memory contains ``depth`` storage elements.
    init : list of int
        Initial values. At power on, each storage element in this memory is initialized to
        the corresponding element of ``init``, if any, or to zero otherwise.
        Uninitialized memories are not currently supported.
    name : str
        Name hint for this memory. If ``None`` (default) the name is inferred from the variable
        name this ``Signal`` is assigned to.
    attrs : dict
        Dictionary of synthesis attributes.
```

[amaranth/mem.py (github.com)](https://github.com)

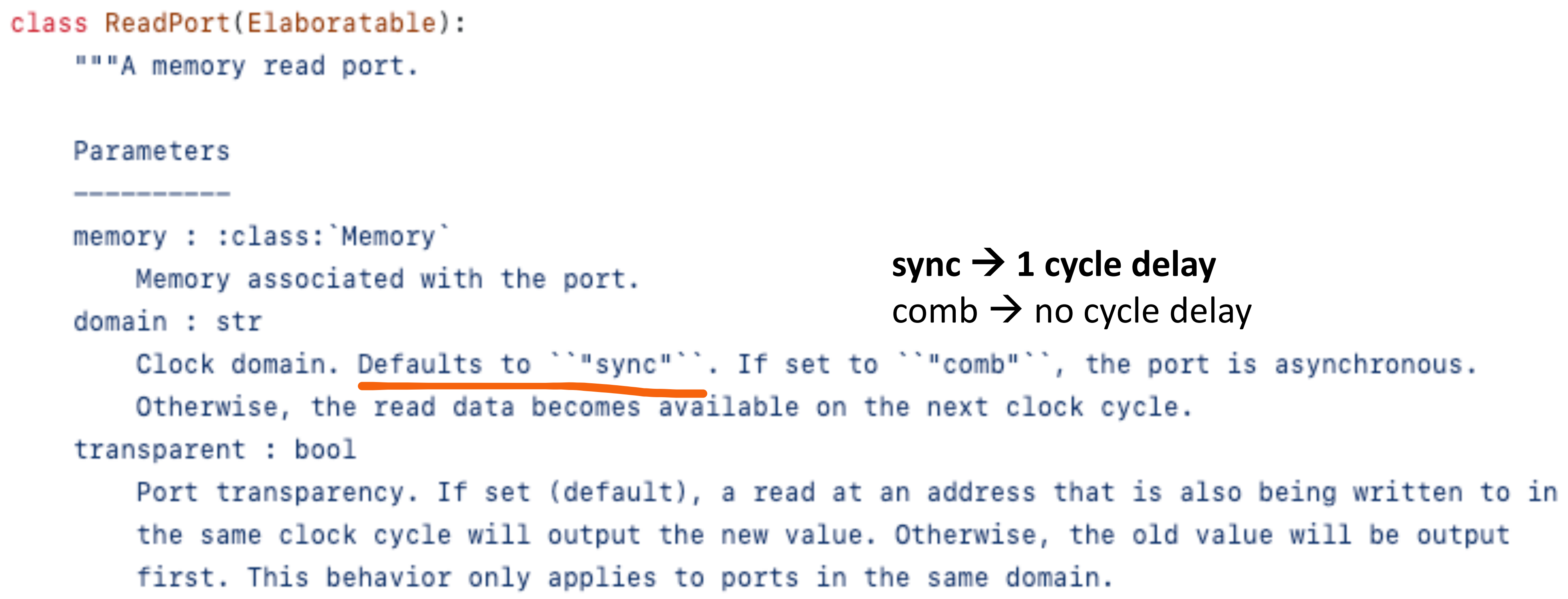
Memory



```
class ReadPort(Elaboratable):
    """A memory read port.

    Parameters
    ----------
    memory : :class:`Memory`
        Memory associated with the port.
    domain : str
        Clock domain. Defaults to ``"sync"``. If set to ``"comb"``, the port is asynchronous.
        Otherwise, the read data becomes available on the next clock cycle.
    transparent : bool
        Port transparency. If set (default), a read at an address that is also being written to in
        the same clock cycle will output the new value. Otherwise, the old value will be output
        first. This behavior only applies to ports in the same domain.
```

amaranth/mem.py (github.com)

Memory

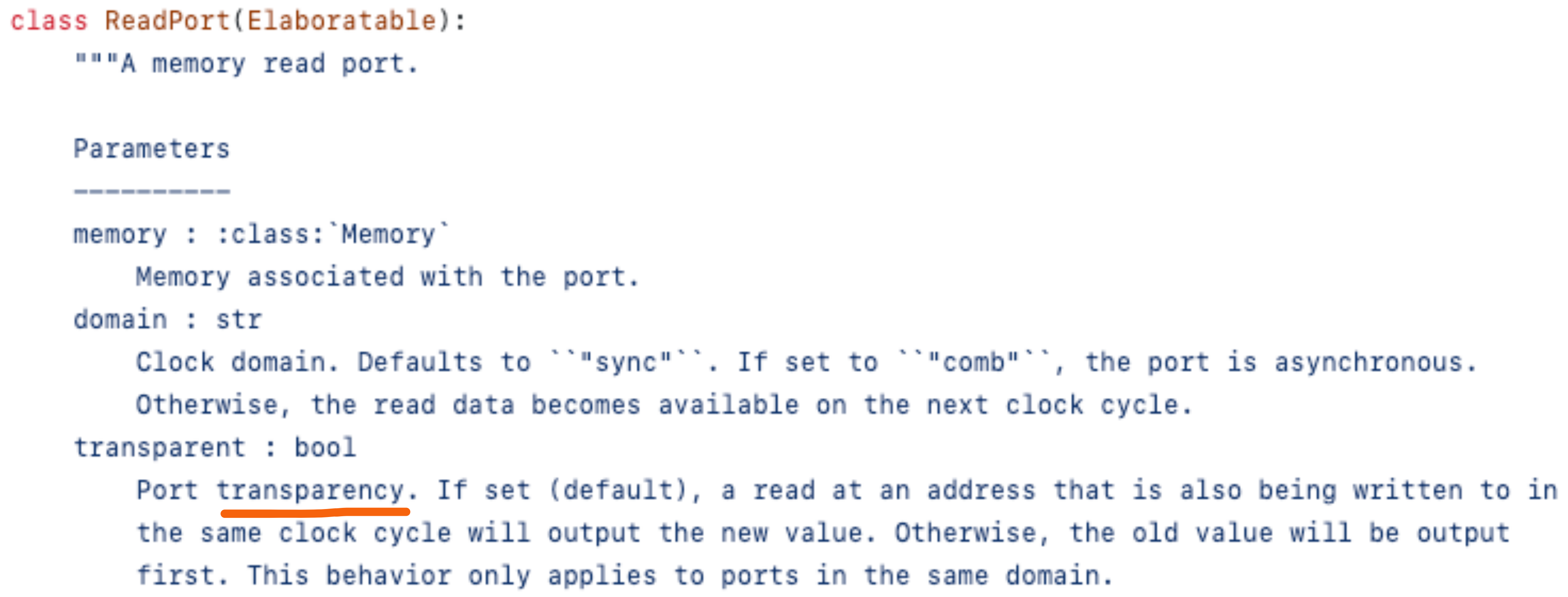


```
class ReadPort(Elaboratable):
    """A memory read port.

    Parameters
    ----------
    memory : :class:`Memory`
        Memory associated with the port.
    domain : str
        Clock domain. Defaults to ``"sync"``. If set to ``"comb"``, the port is asynchronous.
        Otherwise, the read data becomes available on the next clock cycle.
    transparent : bool
        Port transparency. If set (default), a read at an address that is also being written to in
        the same clock cycle will output the new value. Otherwise, the old value will be output
        first. This behavior only applies to ports in the same domain.
```

sync → 1 cycle delay
comb → no cycle delay

[amaranth/mem.py (github.com)](#)

Memory



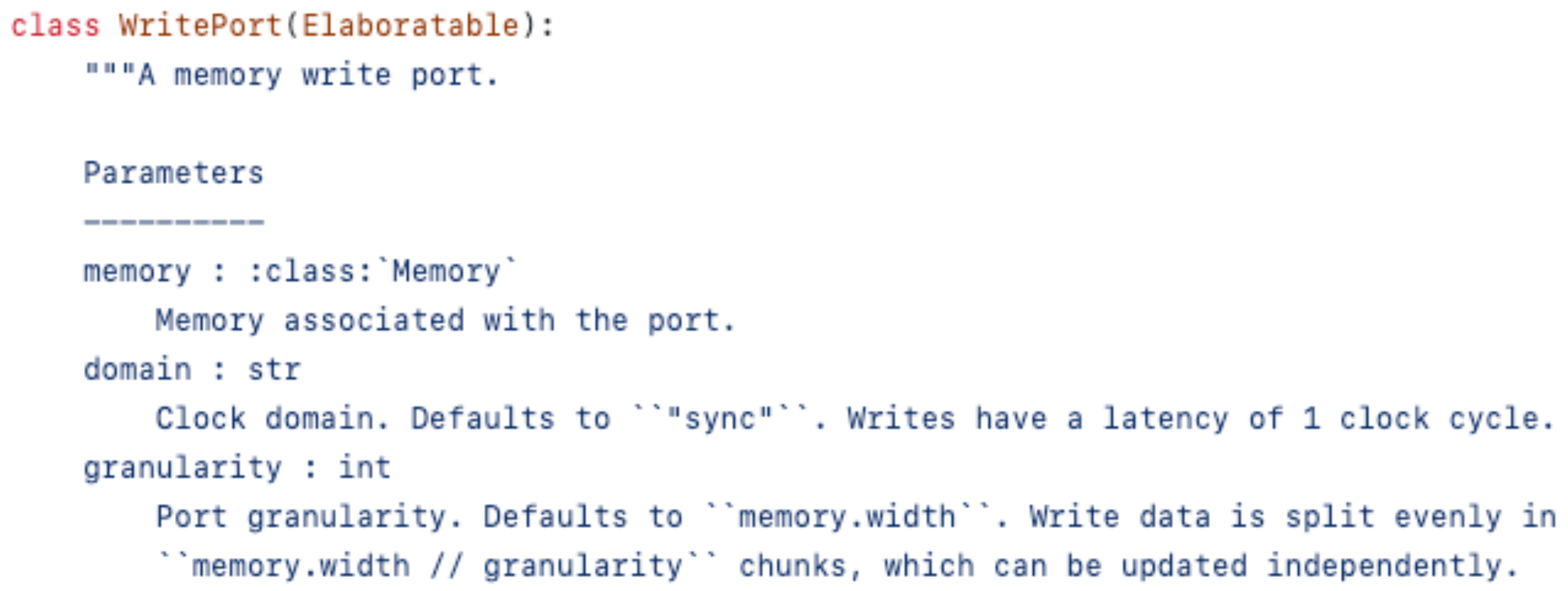
```
class ReadPort(Elaboratable):
    """A memory read port.

    Parameters
    ----------
    memory : :class:`Memory`
        Memory associated with the port.
    domain : str
        Clock domain. Defaults to ``"sync"``. If set to ``"comb"``, the port is asynchronous.
        Otherwise, the read data becomes available on the next clock cycle.
    transparent : bool
        Port transparency. If set (default), a read at an address that is also being written to in
        the same clock cycle will output the new value. Otherwise, the old value will be output
        first. This behavior only applies to ports in the same domain.
```

True → writing input passed to output

False → output the value that will be overwritten

[amaranth/mem.py (github.com)]

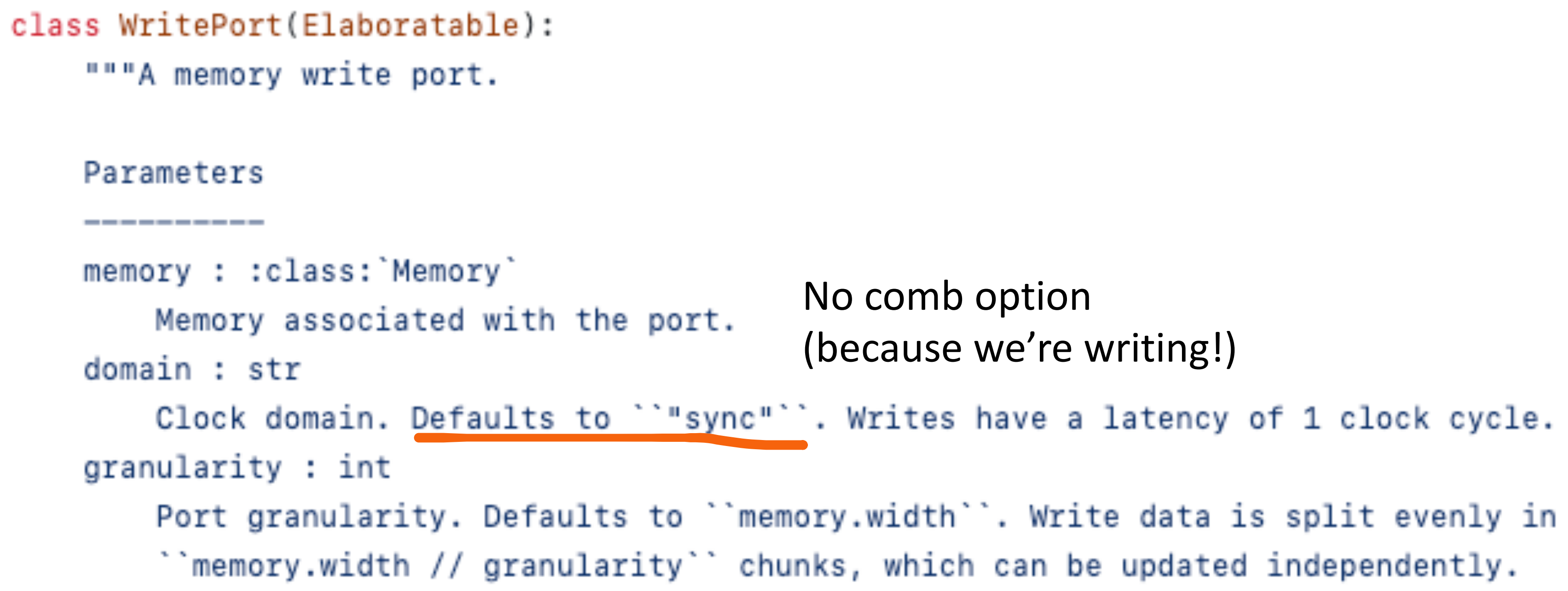
Memory



```
class WritePort(Elaboratable):
    """A memory write port.

    Parameters
    ----------
    memory : :class:`Memory`
        Memory associated with the port.
    domain : str
        Clock domain. Defaults to ``"sync"``. Writes have a latency of 1 clock cycle.
    granularity : int
        Port granularity. Defaults to ``memory.width``. Write data is split evenly in
        ``memory.width // granularity`` chunks, which can be updated independently.
```

[amaranth/mem.py (github.com)](https://github.com)

Memory

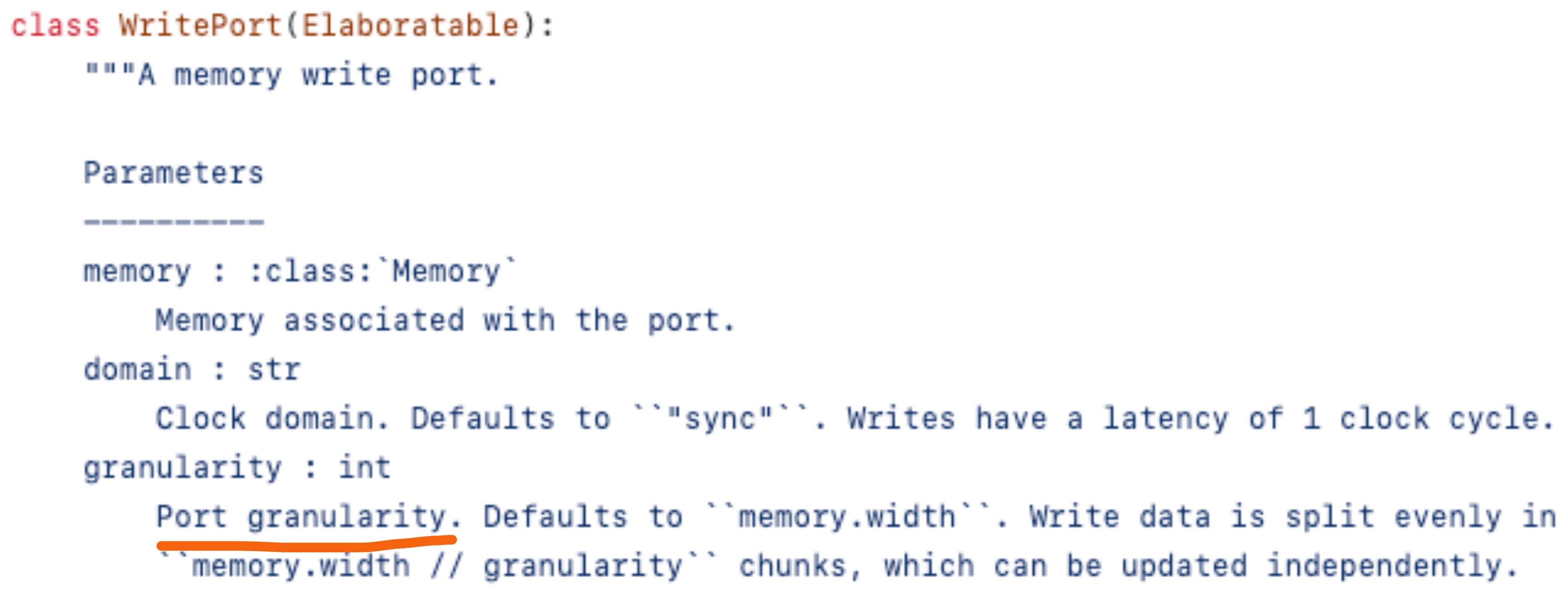


```python
class WritePort(Elaboratable):
    """A memory write port.

    Parameters
    ----------
    memory : :class:`Memory`
        Memory associated with the port.
    domain : str
        Clock domain. Defaults to ``"sync"``. Writes have a latency of 1 clock cycle.
    granularity : int
        Port granularity. Defaults to ``memory.width``. Write data is split evenly in
        ``memory.width // granularity`` chunks, which can be updated independently.
```

No comb option
(because we're writing!)

[amaranth/mem.py (github.com)](https://github.com)

Memory

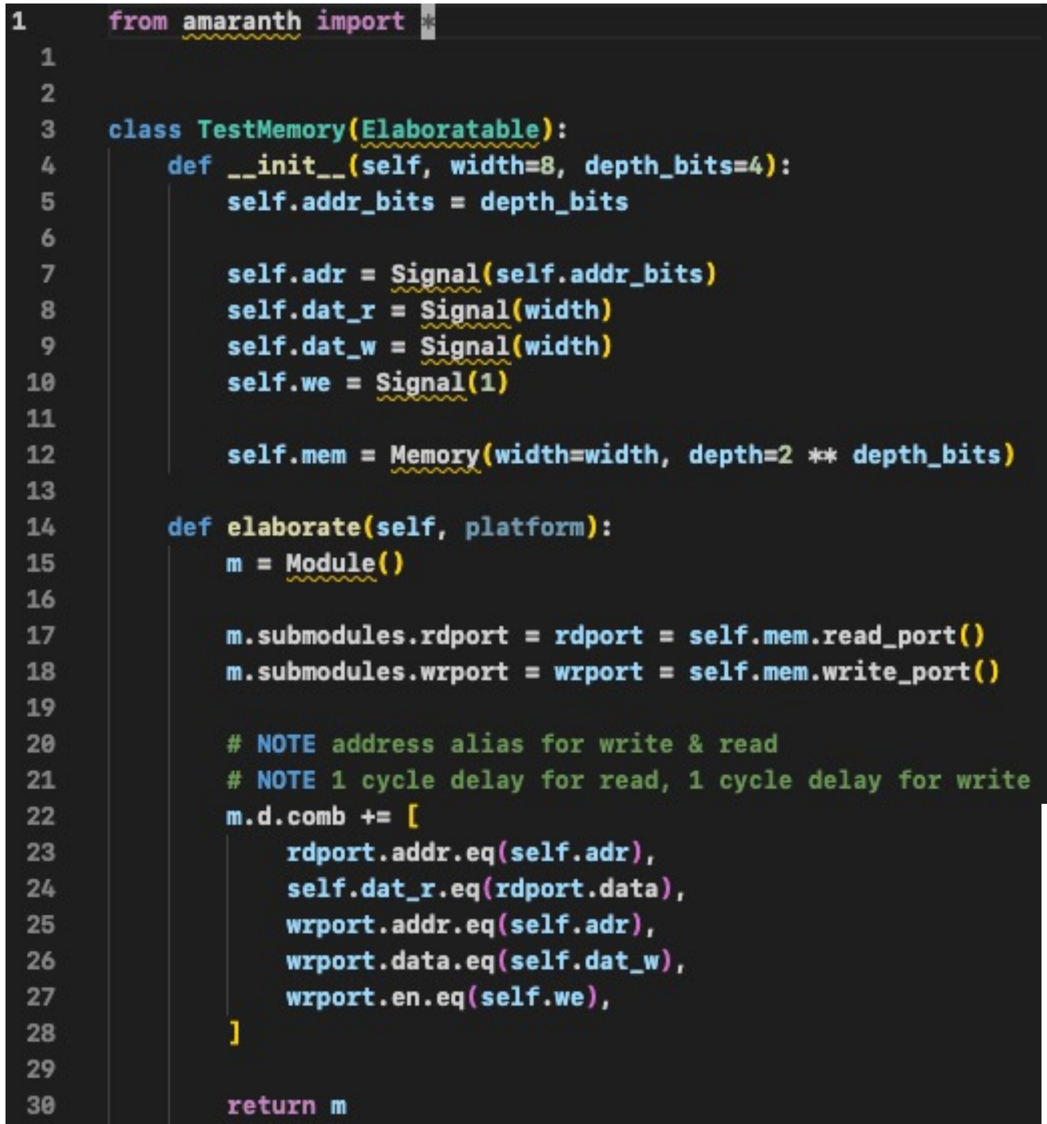


```
class WritePort(Elaboratable):
    """A memory write port.

    Parameters
    ----------
    memory : :class:`Memory`
        Memory associated with the port.
    domain : str
        Clock domain. Defaults to ``"sync"``. Writes have a latency of 1 clock cycle.
    granularity : int
        Port granularity. Defaults to ``memory.width``. Write data is split evenly in
        ``memory.width // granularity`` chunks, which can be updated independently.
```

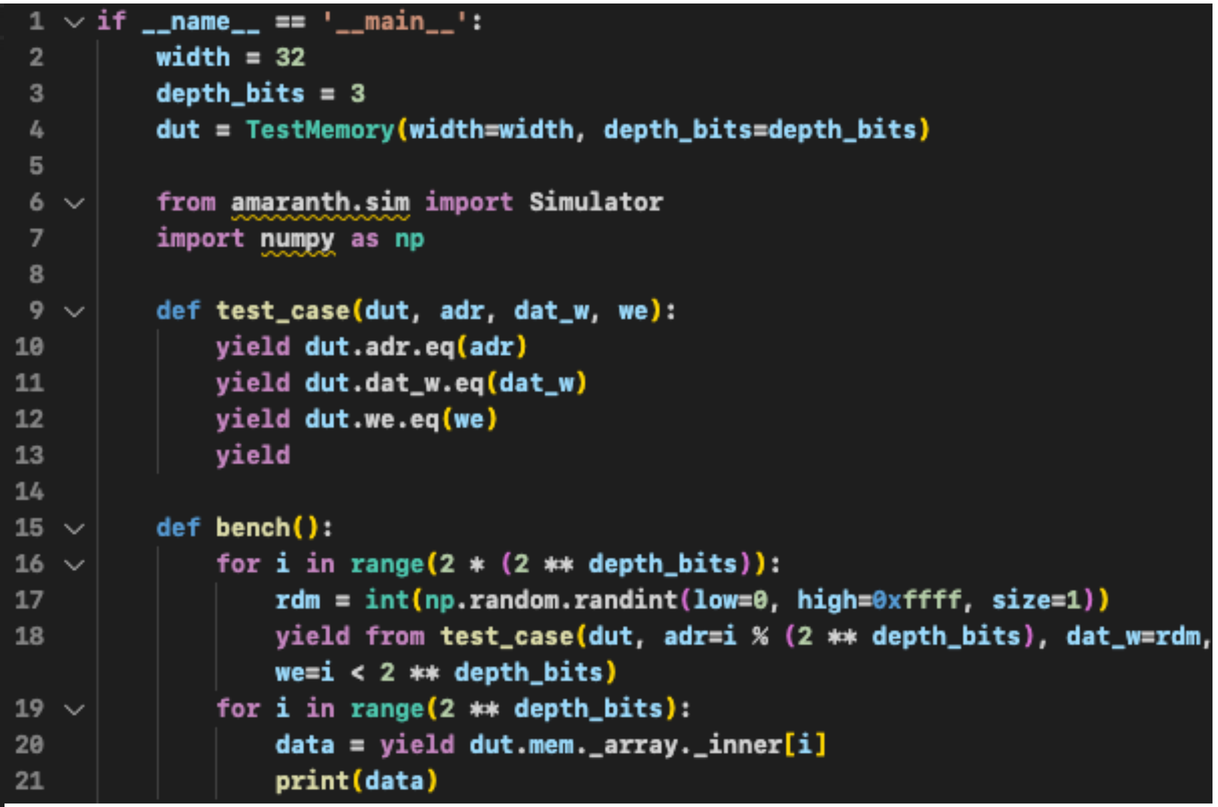
Default : write whole line or "storage element"
For byte-level write, `granularity=8`

amaranth/mem.py (github.com)

Memory

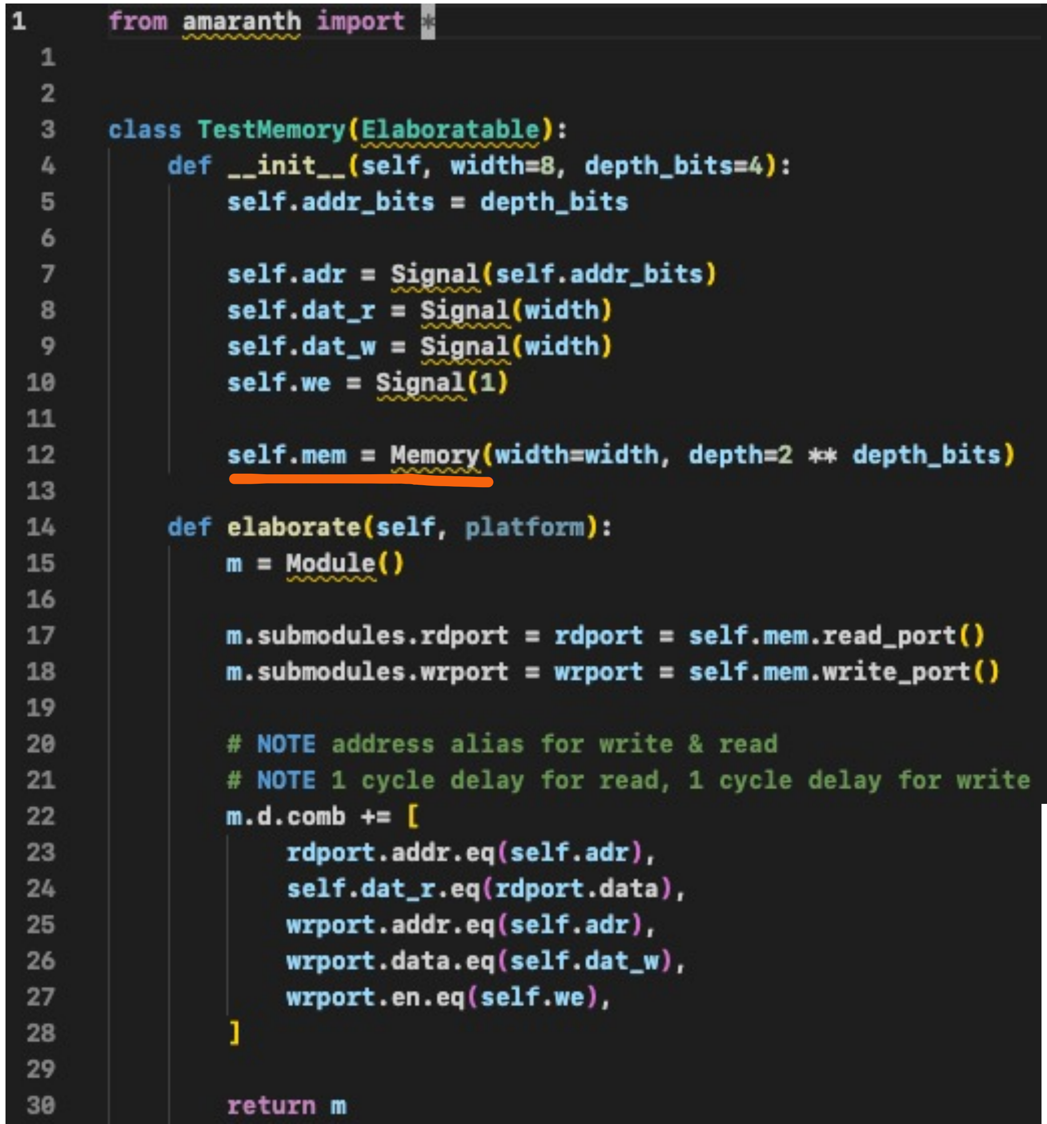


```python
from amaranth import *


class TestMemory(Elaboratable):
    def __init__(self, width=8, depth_bits=4):
        self.addr_bits = depth_bits

        self.adr = Signal(self.addr_bits)
        self.dat_r = Signal(width)
        self.dat_w = Signal(width)
        self.we = Signal(1)

        self.mem = Memory(width=width, depth=2 ** depth_bits)

    def elaborate(self, platform):
        m = Module()

        m.submodules.rdport = rdport = self.mem.read_port()
        m.submodules.wrport = wrport = self.mem.write_port()

        # NOTE address alias for write & read
        # NOTE 1 cycle delay for read, 1 cycle delay for write
        m.d.comb += [
            rdport.addr.eq(self.adr),
            self.dat_r.eq(rdport.data),
            wrport.addr.eq(self.adr),
            wrport.data.eq(self.dat_w),
            wrport.en.eq(self.we),
        ]

        return m
```

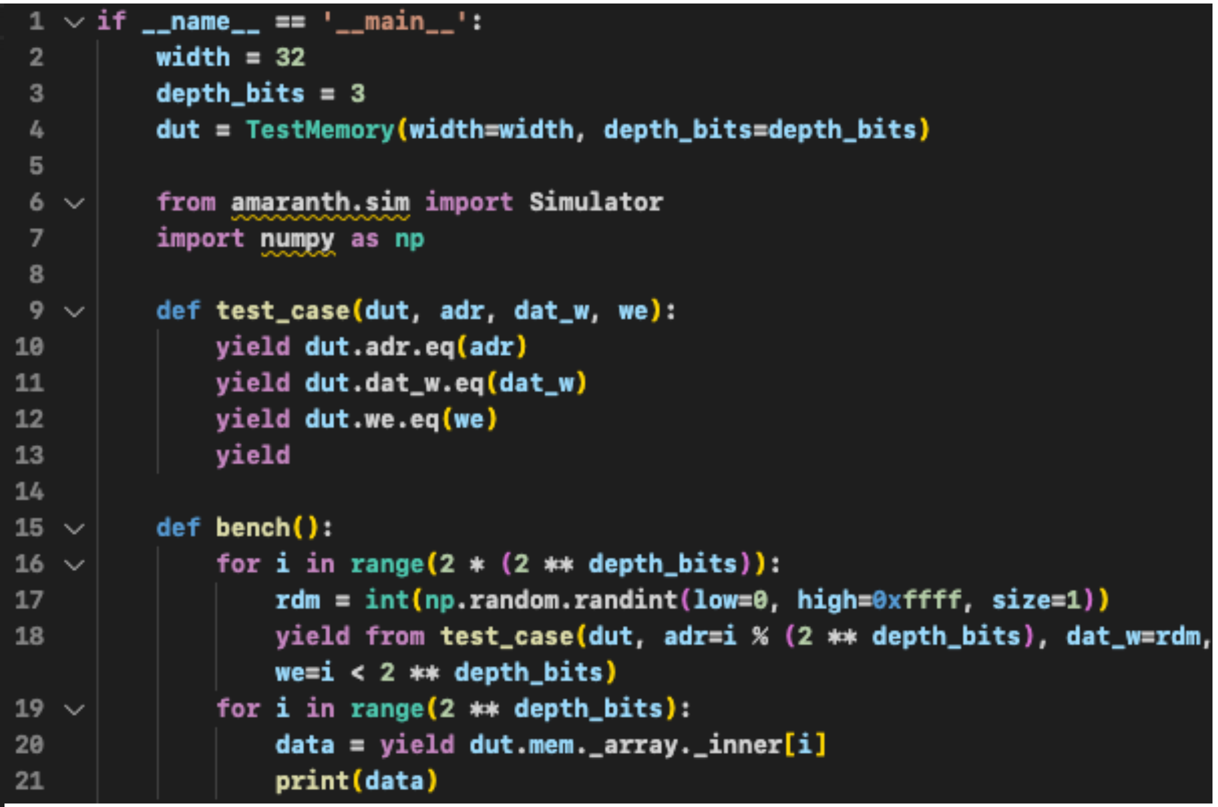


```python
if __name__ == '__main__':
    width = 32
    depth_bits = 3
    dut = TestMemory(width=width, depth_bits=depth_bits)

    from amaranth.sim import Simulator
    import numpy as np

    def test_case(dut, adr, dat_w, we):
        yield dut.adr.eq(adr)
        yield dut.dat_w.eq(dat_w)
        yield dut.we.eq(we)
        yield

    def bench():
        for i in range(2 * (2 ** depth_bits)):
            rdm = int(np.random.randint(low=0, high=0xffff, size=1))
            yield from test_case(dut, adr=i % (2 ** depth_bits), dat_w=rdm,
                we=i < 2 ** depth_bits)
        for i in range(2 ** depth_bits):
            data = yield dut.mem._array._inner[i]
            print(data)
```

Memory

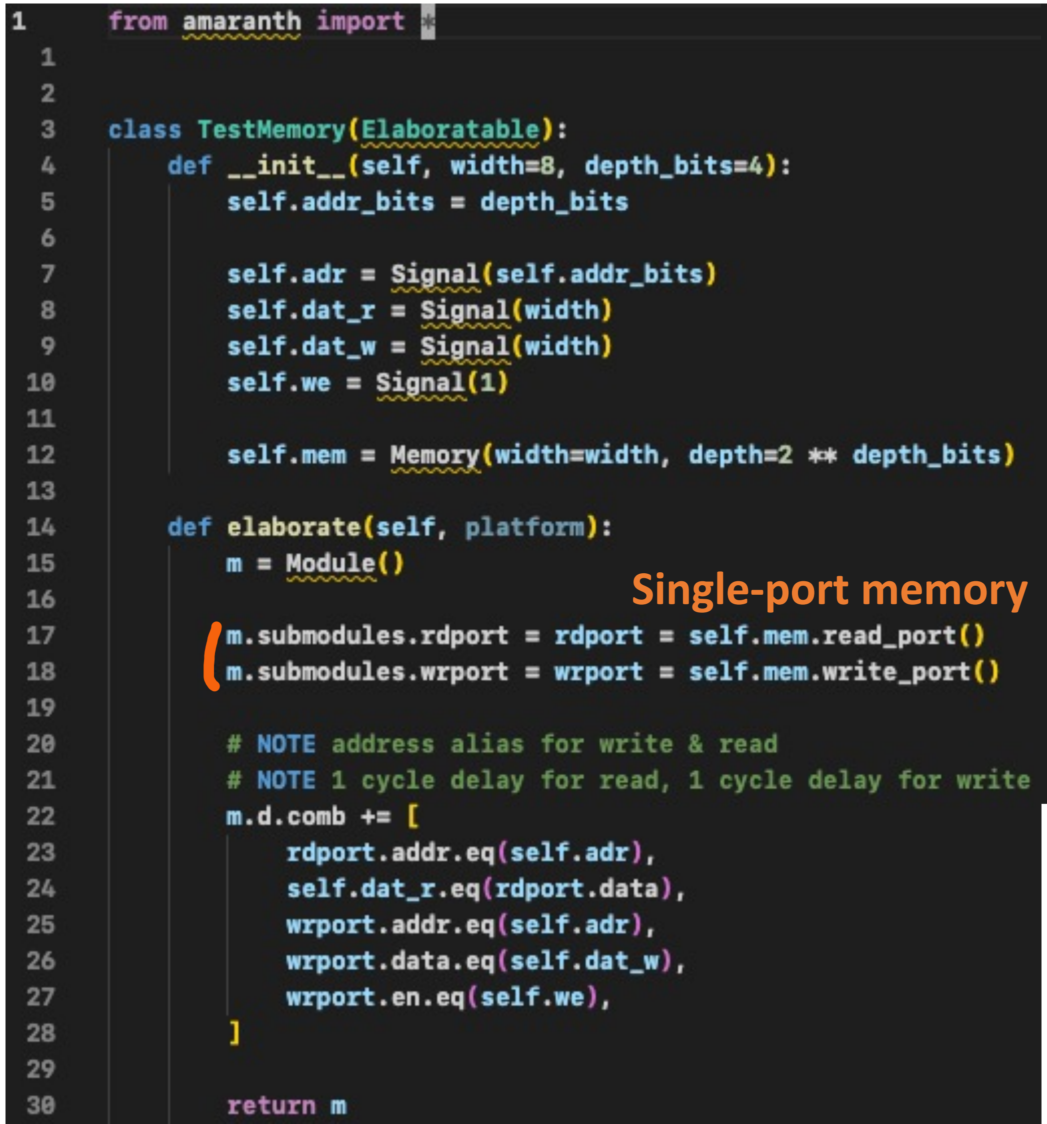


```python
from amaranth import *


class TestMemory(Elaboratable):
    def __init__(self, width=8, depth_bits=4):
        self.addr_bits = depth_bits

        self.adr = Signal(self.addr_bits)
        self.dat_r = Signal(width)
        self.dat_w = Signal(width)
        self.we = Signal(1)

        self.mem = Memory(width=width, depth=2 ** depth_bits)

    def elaborate(self, platform):
        m = Module()

        m.submodules.rdport = rdport = self.mem.read_port()
        m.submodules.wrport = wrport = self.mem.write_port()

        # NOTE address alias for write & read
        # NOTE 1 cycle delay for read, 1 cycle delay for write
        m.d.comb += [
            rdport.addr.eq(self.adr),
            self.dat_r.eq(rdport.data),
            wrport.addr.eq(self.adr),
            wrport.data.eq(self.dat_w),
            wrport.en.eq(self.we),
        ]

        return m
```



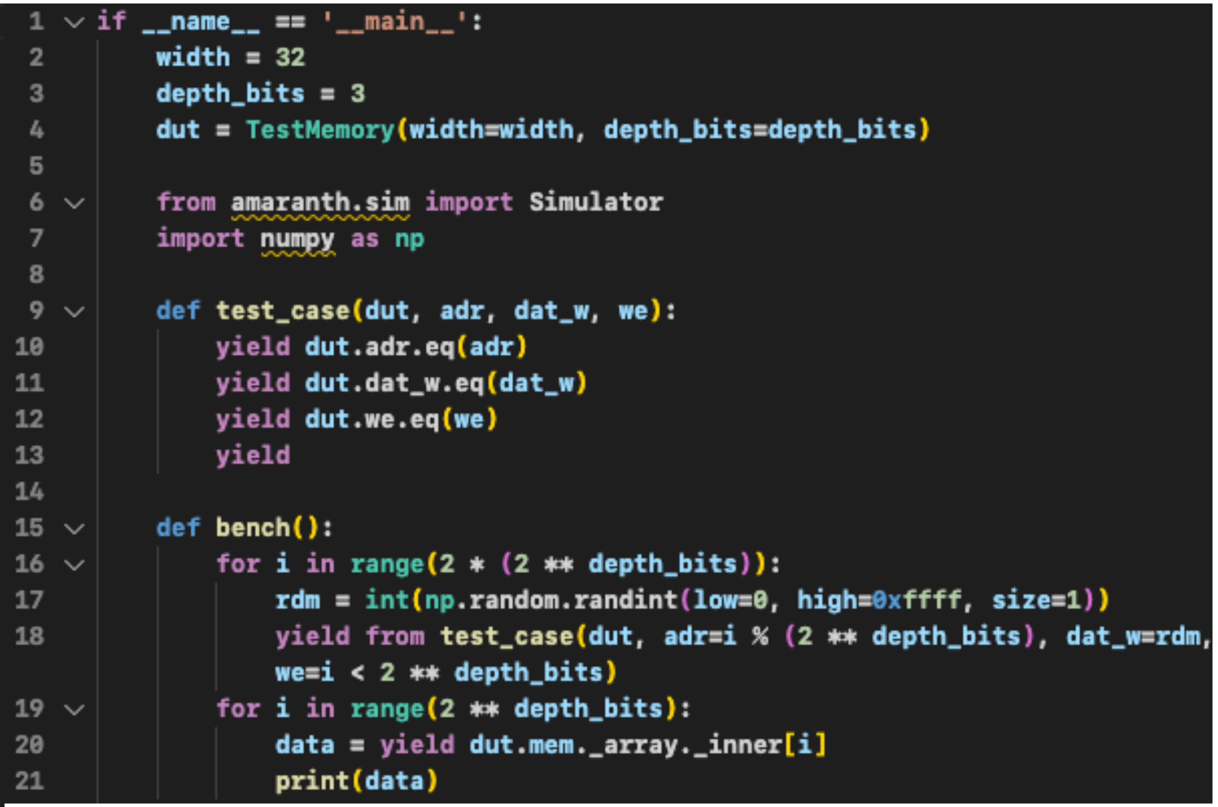
```python
if __name__ == '__main__':
    width = 32
    depth_bits = 3
    dut = TestMemory(width=width, depth_bits=depth_bits)

    from amaranth.sim import Simulator
    import numpy as np

    def test_case(dut, adr, dat_w, we):
        yield dut.adr.eq(adr)
        yield dut.dat_w.eq(dat_w)
        yield dut.we.eq(we)
        yield

    def bench():
        for i in range(2 * (2 ** depth_bits)):
            rdm = int(np.random.randint(low=0, high=0xffff, size=1))
            yield from test_case(dut, adr=i % (2 ** depth_bits), dat_w=rdm,
            we=i < 2 ** depth_bits)
        for i in range(2 ** depth_bits):
            data = yield dut.mem._array._inner[i]
            print(data)
```

Memory



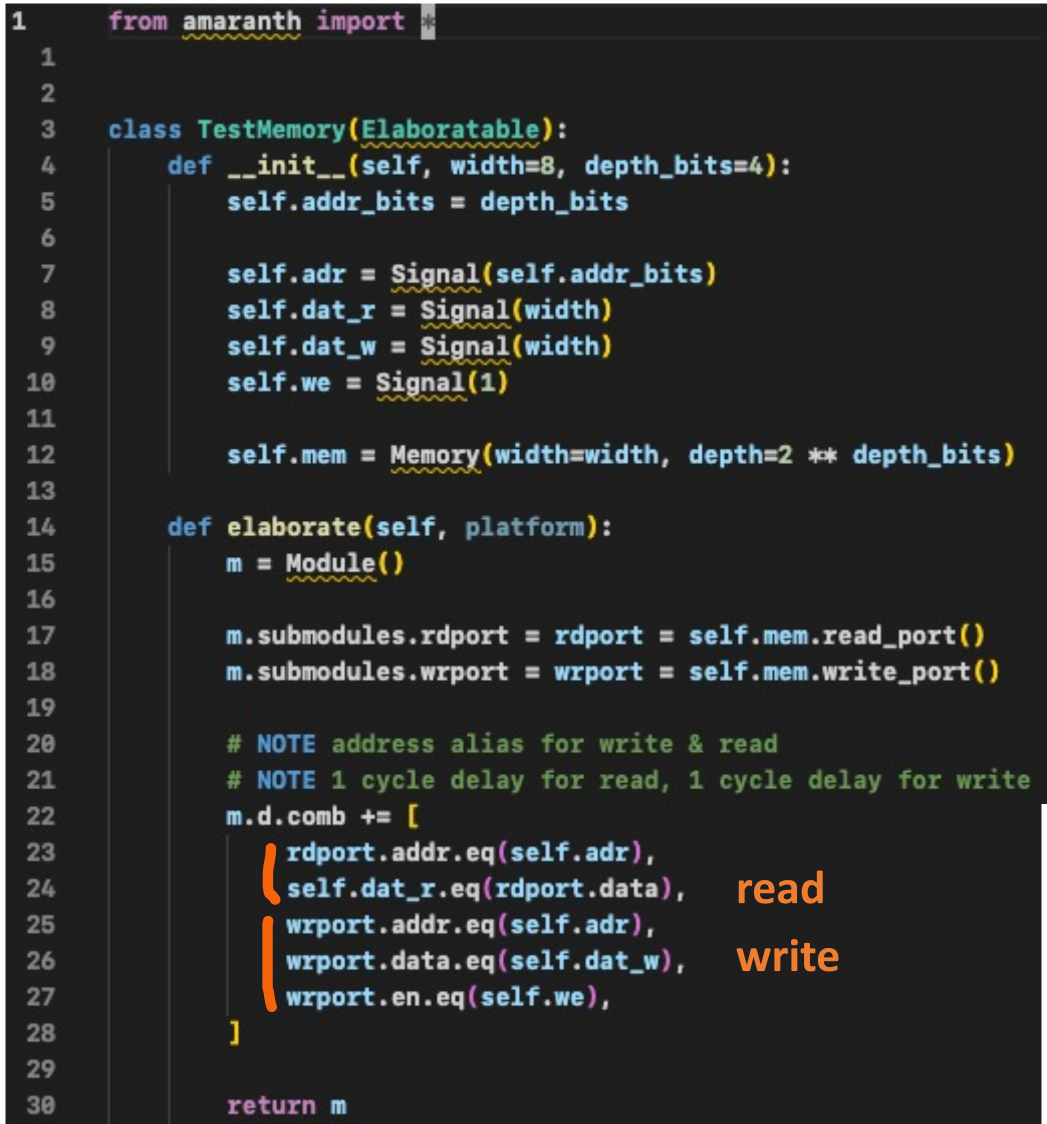
```python
from amaranth import *


class TestMemory(Elaboratable):
    def __init__(self, width=8, depth_bits=4):
        self.addr_bits = depth_bits

        self.adr = Signal(self.addr_bits)
        self.dat_r = Signal(width)
        self.dat_w = Signal(width)
        self.we = Signal(1)

        self.mem = Memory(width=width, depth=2 ** depth_bits)

    def elaborate(self, platform):
        m = Module()

        m.submodules.rdport = rdport = self.mem.read_port()
        m.submodules.wrport = wrport = self.mem.write_port()

        # NOTE address alias for write & read
        # NOTE 1 cycle delay for read, 1 cycle delay for write
        m.d.comb += [
            rdport.addr.eq(self.adr),
            self.dat_r.eq(rdport.data),
            wrport.addr.eq(self.adr),
            wrport.data.eq(self.dat_w),
            wrport.en.eq(self.we),
        ]

        return m
```

Single-port memory

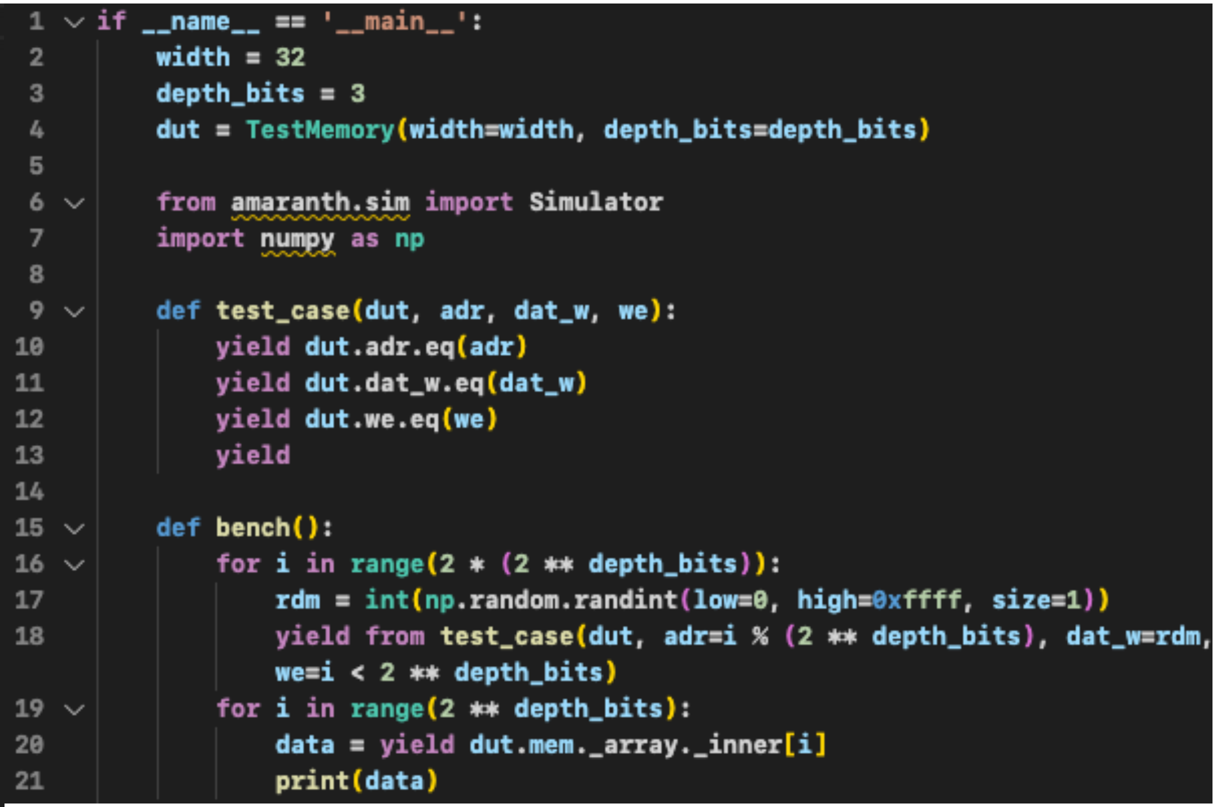


```python
if __name__ == '__main__':
    width = 32
    depth_bits = 3
    dut = TestMemory(width=width, depth_bits=depth_bits)

    from amaranth.sim import Simulator
    import numpy as np

    def test_case(dut, adr, dat_w, we):
        yield dut.adr.eq(adr)
        yield dut.dat_w.eq(dat_w)
        yield dut.we.eq(we)
        yield

    def bench():
        for i in range(2 * (2 ** depth_bits)):
            rdm = int(np.random.randint(low=0, high=0xffff, size=1))
            yield from test_case(dut, adr=i % (2 ** depth_bits), dat_w=rdm,
                we=i < 2 ** depth_bits)
        for i in range(2 ** depth_bits):
            data = yield dut.mem._array._inner[i]
            print(data)
```

Memory

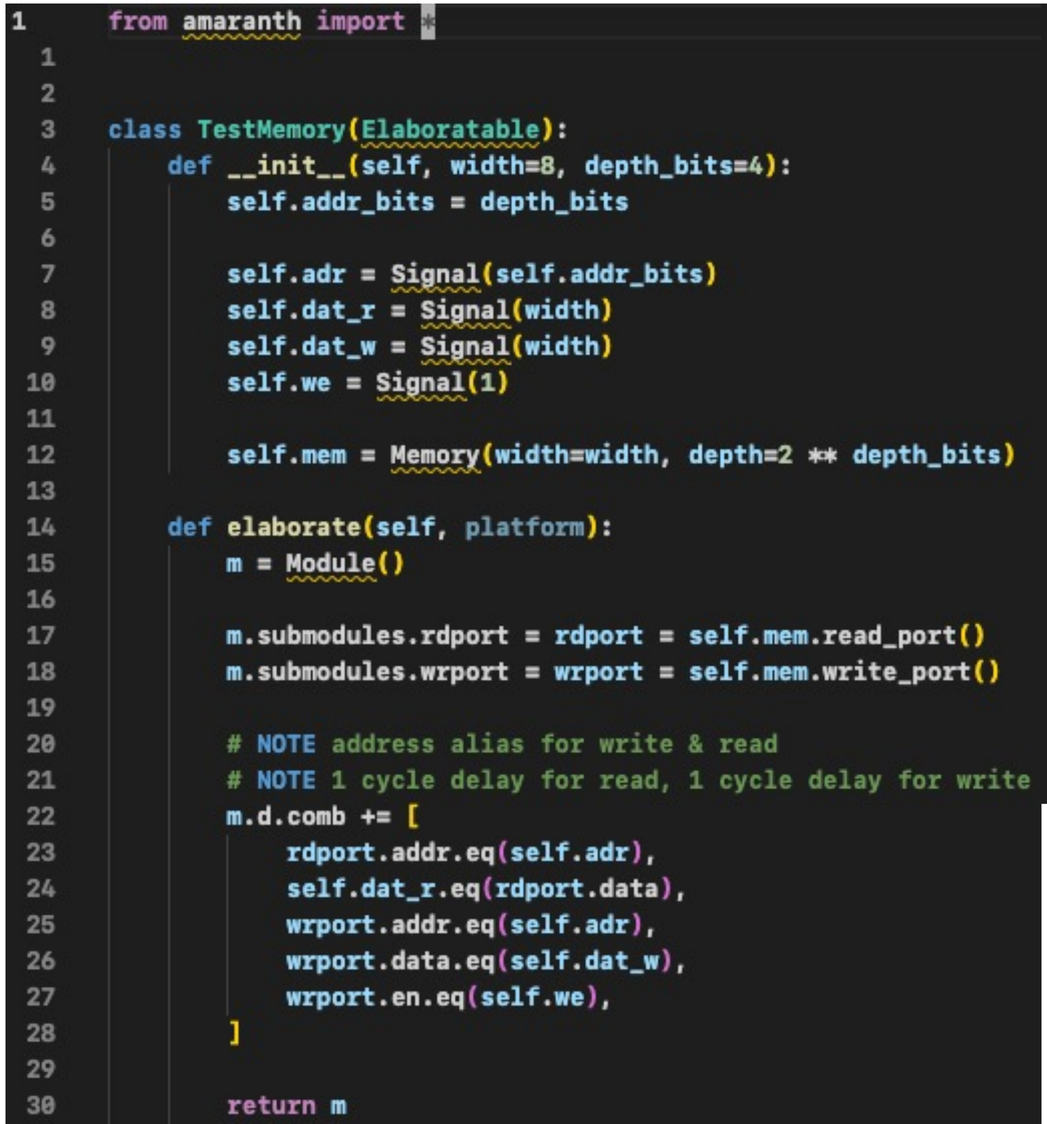


```python
from amaranth import *


class TestMemory(Elaboratable):
    def __init__(self, width=8, depth_bits=4):
        self.addr_bits = depth_bits

        self.adr = Signal(self.addr_bits)
        self.dat_r = Signal(width)
        self.dat_w = Signal(width)
        self.we = Signal(1)

        self.mem = Memory(width=width, depth=2 ** depth_bits)

    def elaborate(self, platform):
        m = Module()

        m.submodules.rdport = rdport = self.mem.read_port()
        m.submodules.wrport = wrport = self.mem.write_port()

        # NOTE address alias for write & read
        # NOTE 1 cycle delay for read, 1 cycle delay for write
        m.d.comb += [
            rdport.addr.eq(self.adr),
            self.dat_r.eq(rdport.data),
            wrport.addr.eq(self.adr),
            wrport.data.eq(self.dat_w),
            wrport.en.eq(self.we),
        ]

        return m
```

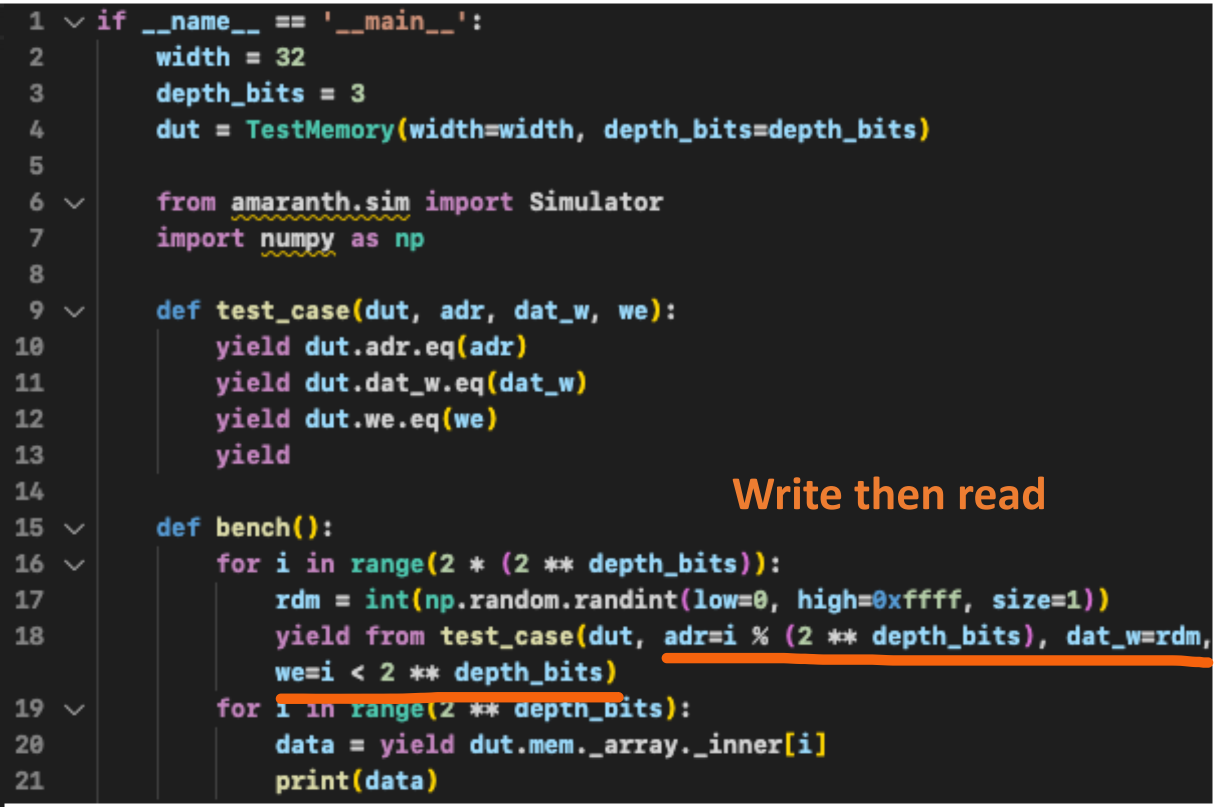


```python
if __name__ == '__main__':
    width = 32
    depth_bits = 3
    dut = TestMemory(width=width, depth_bits=depth_bits)

    from amaranth.sim import Simulator
    import numpy as np

    def test_case(dut, adr, dat_w, we):
        yield dut.adr.eq(adr)
        yield dut.dat_w.eq(dat_w)
        yield dut.we.eq(we)
        yield

    def bench():
        for i in range(2 * (2 ** depth_bits)):
            rdm = int(np.random.randint(low=0, high=0xffff, size=1))
            yield from test_case(dut, adr=i % (2 ** depth_bits), dat_w=rdm,
                we=i < 2 ** depth_bits)
        for i in range(2 ** depth_bits):
            data = yield dut.mem._array._inner[i]
            print(data)
```

Memory



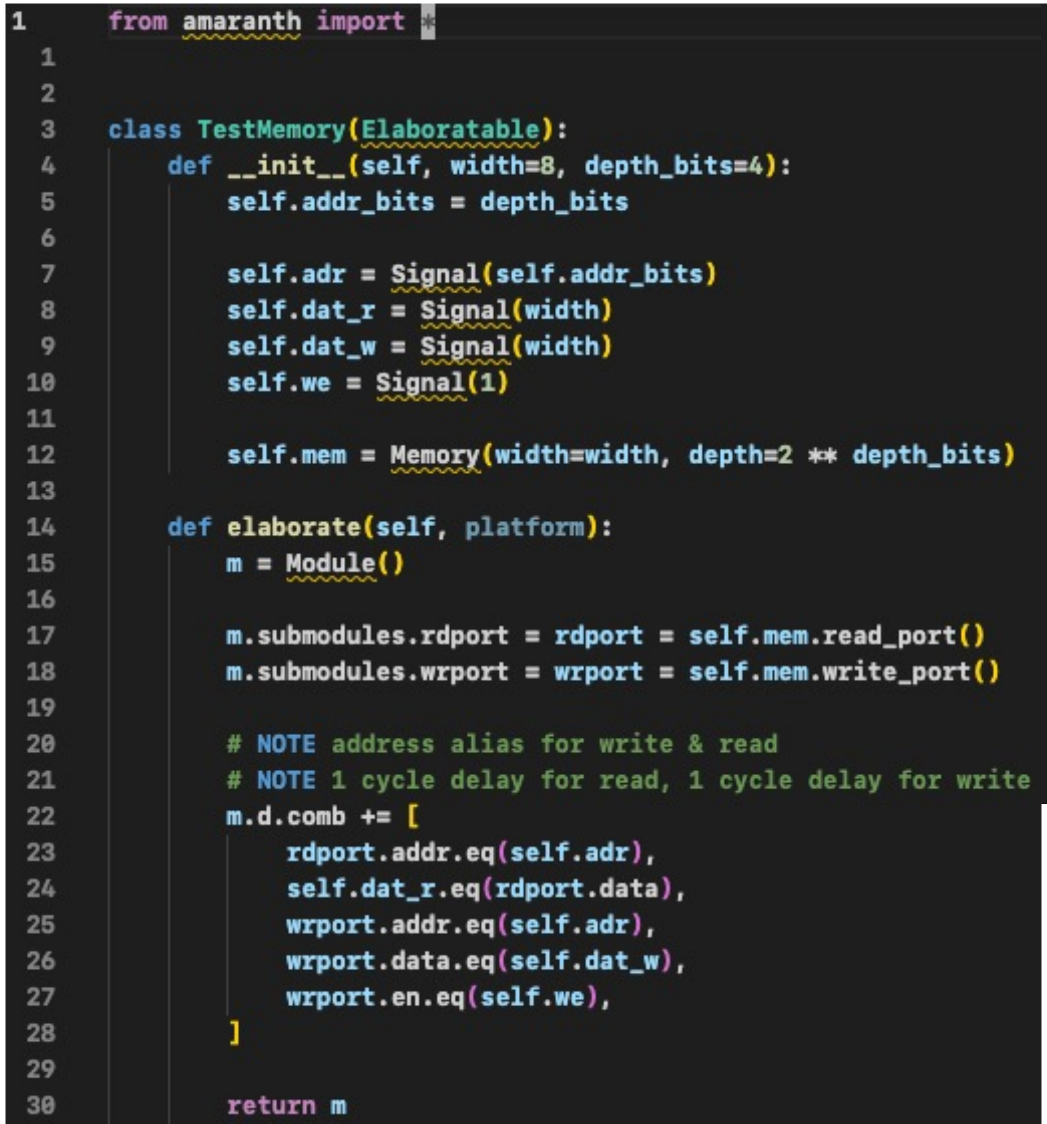
```python
from amaranth import *


class TestMemory(Elaboratable):
    def __init__(self, width=8, depth_bits=4):
        self.addr_bits = depth_bits

        self.adr = Signal(self.addr_bits)
        self.dat_r = Signal(width)
        self.dat_w = Signal(width)
        self.we = Signal(1)

        self.mem = Memory(width=width, depth=2 ** depth_bits)

    def elaborate(self, platform):
        m = Module()

        m.submodules.rdport = rdport = self.mem.read_port()
        m.submodules.wrport = wrport = self.mem.write_port()

        # NOTE address alias for write & read
        # NOTE 1 cycle delay for read, 1 cycle delay for write
        m.d.comb += [
            rdport.addr.eq(self.adr),
            self.dat_r.eq(rdport.data),
            wrport.addr.eq(self.adr),
            wrport.data.eq(self.dat_w),
            wrport.en.eq(self.we),
        ]

        return m
```



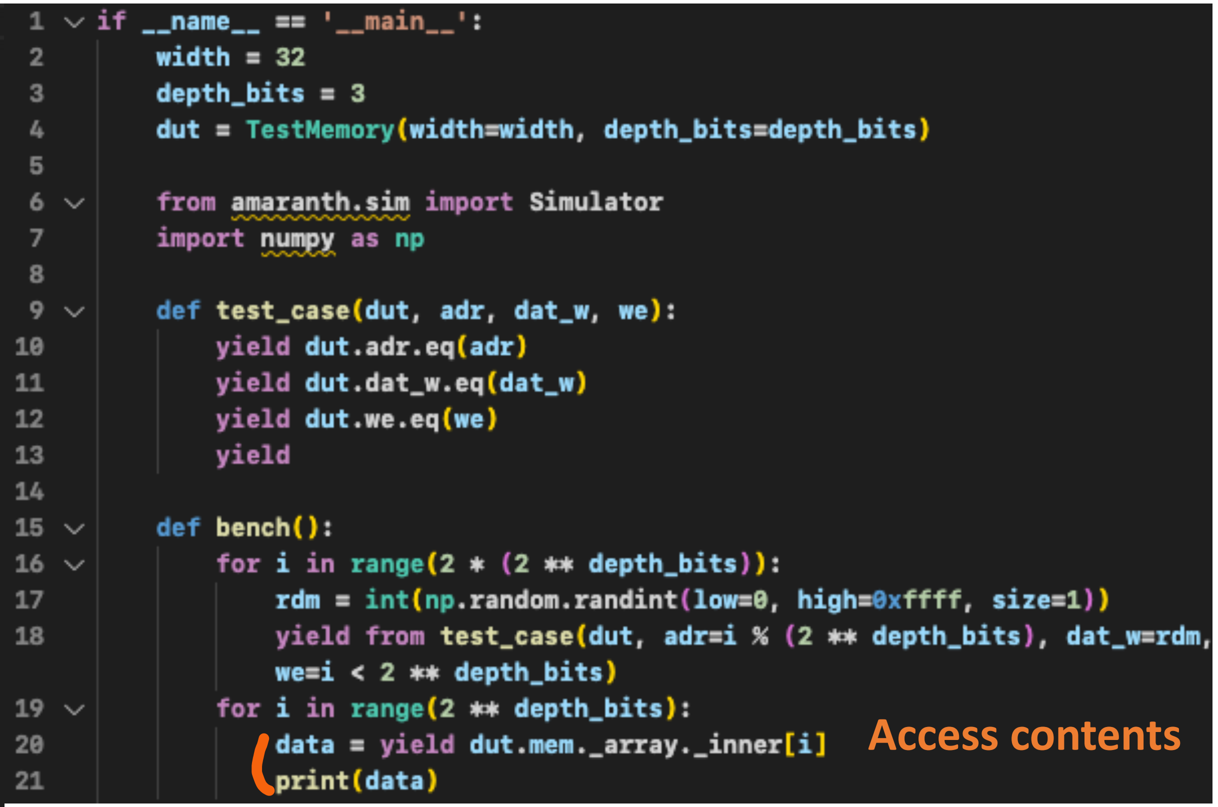
```python
if __name__ == '__main__':
    width = 32
    depth_bits = 3
    dut = TestMemory(width=width, depth_bits=depth_bits)

    from amaranth.sim import Simulator
    import numpy as np

    def test_case(dut, adr, dat_w, we):
        yield dut.adr.eq(adr)
        yield dut.dat_w.eq(dat_w)
        yield dut.we.eq(we)
        yield

    def bench():
        for i in range(2 * (2 ** depth_bits)):
            rdm = int(np.random.randint(low=0, high=0xffff, size=1))
            yield from test_case(dut, adr=i % (2 ** depth_bits), dat_w=rdm,
            we=i < 2 ** depth_bits)
        for i in range(2 ** depth_bits):
            data = yield dut.mem._array._inner[i]
            print(data)
```

Write then read

Memory

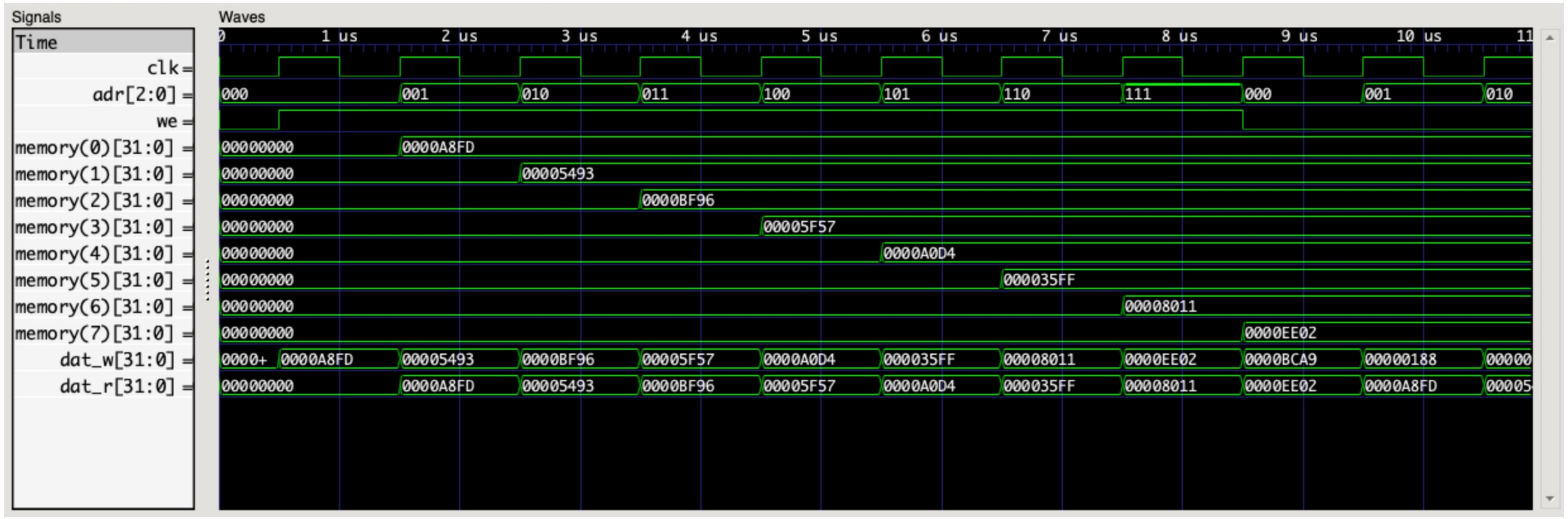


```python
from amaranth import *


class TestMemory(Elaboratable):
    def __init__(self, width=8, depth_bits=4):
        self.addr_bits = depth_bits

        self.adr = Signal(self.addr_bits)
        self.dat_r = Signal(width)
        self.dat_w = Signal(width)
        self.we = Signal(1)

        self.mem = Memory(width=width, depth=2 ** depth_bits)

    def elaborate(self, platform):
        m = Module()

        m.submodules.rdport = rdport = self.mem.read_port()
        m.submodules.wrport = wrport = self.mem.write_port()

        # NOTE address alias for write & read
        # NOTE 1 cycle delay for read, 1 cycle delay for write
        m.d.comb += [
            rdport.addr.eq(self.adr),
            self.dat_r.eq(rdport.data),
            wrport.addr.eq(self.adr),
            wrport.data.eq(self.dat_w),
            wrport.en.eq(self.we),
        ]

        return m
```



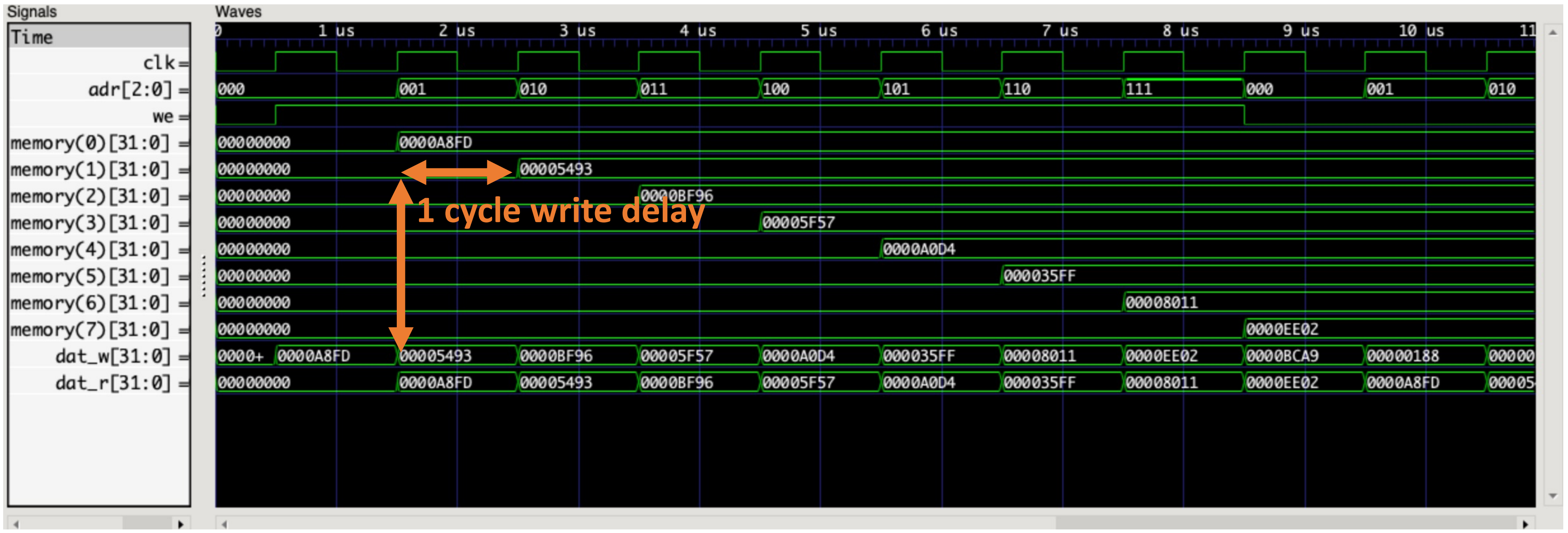
```python
if __name__ == '__main__':
    width = 32
    depth_bits = 3
    dut = TestMemory(width=width, depth_bits=depth_bits)

    from amaranth.sim import Simulator
    import numpy as np

    def test_case(dut, adr, dat_w, we):
        yield dut.adr.eq(adr)
        yield dut.dat_w.eq(dat_w)
        yield dut.we.eq(we)
        yield

    def bench():
        for i in range(2 * (2 ** depth_bits)):
            rdm = int(np.random.randint(low=0, high=0xffff, size=1))
            yield from test_case(dut, adr=i % (2 ** depth_bits), dat_w=rdm,
            we=i < 2 ** depth_bits)
        for i in range(2 ** depth_bits):
            data = yield dut.mem._array._inner[i]
            print(data)
```

Memory

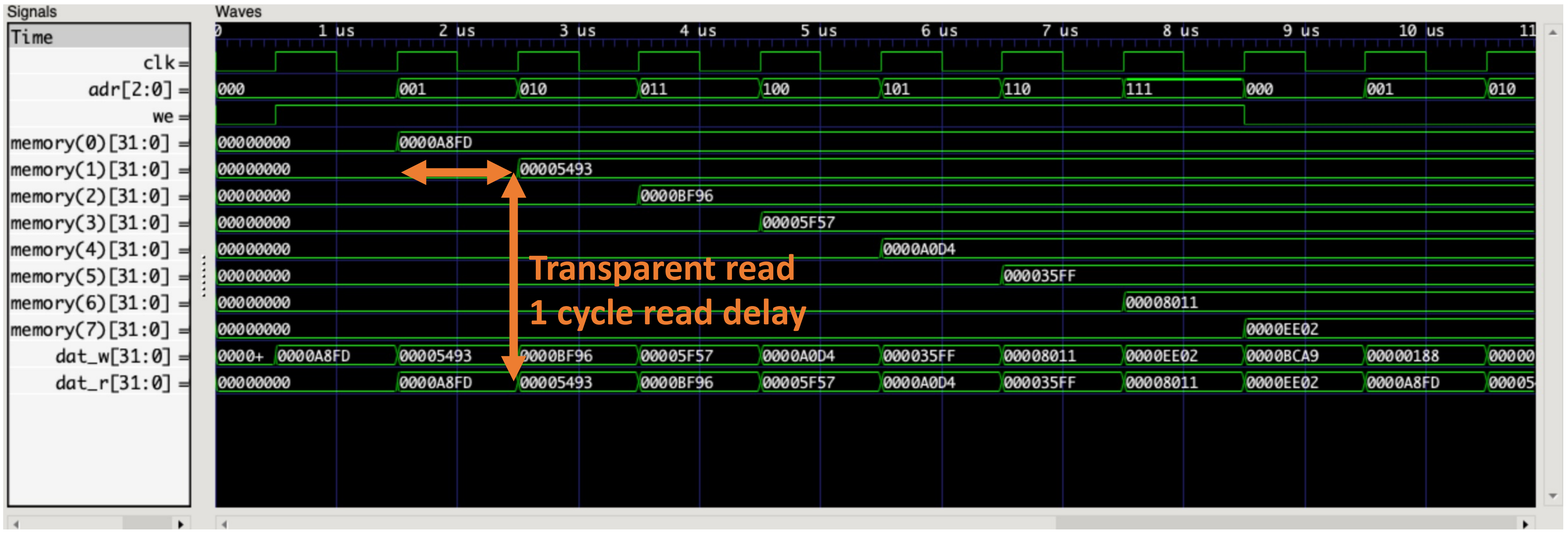


```
adr[2:0]          000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 | 000 | 001 | 010
memory(0)[31:0]   00000000 | 0000A8FD
memory(1)[31:0]   00000000 | 00005493
memory(2)[31:0]   00000000 | 0000BF96
memory(3)[31:0]   00000000 | 00005F57
memory(4)[31:0]   00000000 | 0000A0D4
memory(5)[31:0]   00000000 | 000035FF
memory(6)[31:0]   00000000 | 00008011
memory(7)[31:0]   00000000 | 0000EE02
dat_w[31:0]       0000+ | 0000A8FD | 00005493 | 0000BF96 | 00005F57 | 0000A0D4 | 000035FF | 00008011 | 0000EE02 | 0000BCA9 | 00000188 | 00000
dat_r[31:0]       00000000 | 0000A8FD | 00005493 | 0000BF96 | 00005F57 | 0000A0D4 | 000035FF | 00008011 | 0000EE02 | 0000A8FD | 00005
```

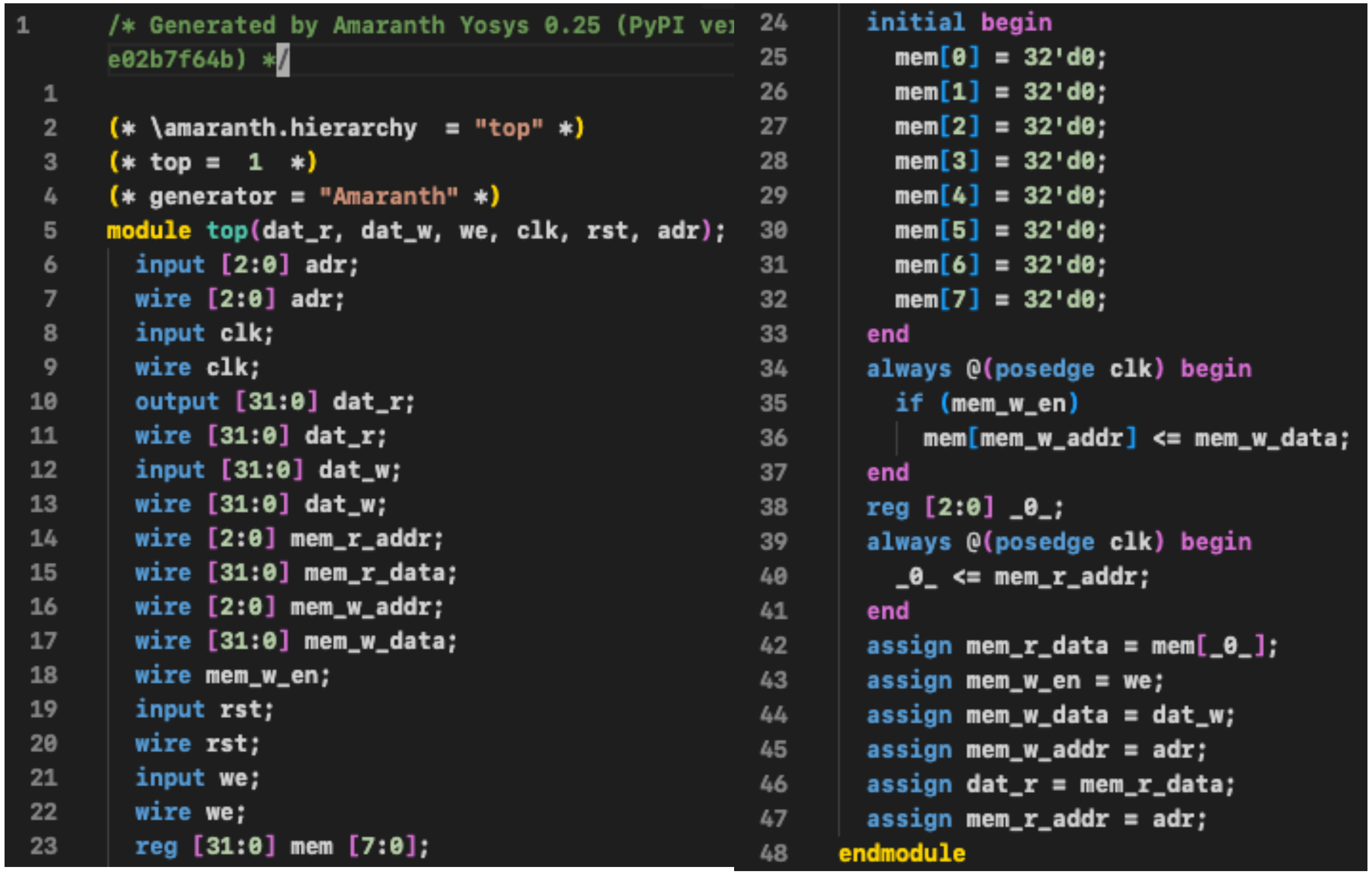
Memory


Signals
Waves
Time
clk
adr[2:0]
we
memory(0)[31:0]
memory(1)[31:0]
memory(2)[31:0]
memory(3)[31:0]
memory(4)[31:0]
memory(5)[31:0]
memory(6)[31:0]
memory(7)[31:0]
dat_w[31:0]
dat_r[31:0]
1 us
2 us
3 us
4 us
5 us
6 us
7 us
8 us
9 us
10 us
000
001
010
011
100
101
110
111
000
001
010
00000000
0000A8FD
00005493
0000BF96
00005F57
0000A0D4
000035FF
00008011
0000EE02
0000+
0000A8FD
00005493
0000BF96
00005F57
0000A0D4
000035FF
00008011
0000EE02
0000BCA9
00000188
00000000
0000A8FD
00005493
0000BF96
00005F57
0000A0D4
000035FF
00008011
0000EE02
0000A8FD
1 cycle write delay

Memory

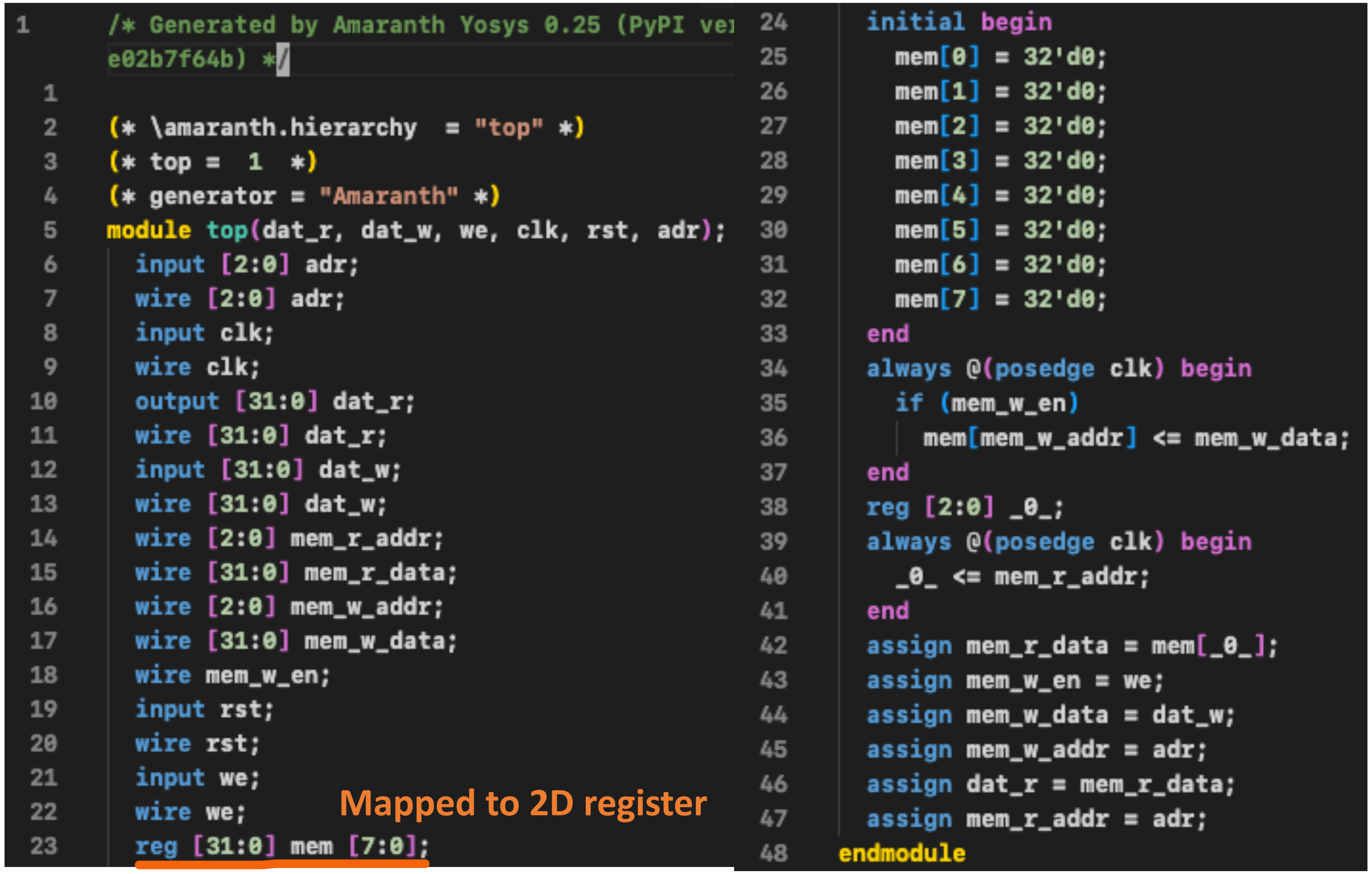

Signals
Waves
Time
0
1 us
2 us
3 us
4 us
5 us
6 us
7 us
8 us
9 us
10 us
11
clk =
adr[2:0] =
000
001
010
011
100
101
110
111
000
001
010
we =
memory(0)[31:0] =
00000000
0000A8FD
memory(1)[31:0] =
00000000
00005493
memory(2)[31:0] =
00000000
0000BF96
memory(3)[31:0] =
00000000
00005F57
memory(4)[31:0] =
00000000
0000A0D4
memory(5)[31:0] =
00000000
000035FF
memory(6)[31:0] =
00000000
00008011
memory(7)[31:0] =
00000000
0000EE02
dat_w[31:0] =
0000+
0000A8FD
00005493
0000BF96
00005F57
0000A0D4
000035FF
00008011
0000EE02
0000BCA9
00000188
00000
dat_r[31:0] =
00000000
0000A8FD
00005493
0000BF96
00005F57
0000A0D4
000035FF
00008011
0000EE02
0000A8FD
00005
Transparent read
1 cycle read delay

Memory



```
/* Generated by Amaranth Yosys 0.25 (PyPI ve
e02b7f64b) */

(* \amaranth.hierarchy  = "top" *)
(* top =  1  *)
(* generator = "Amaranth" *)
module top(dat_r, dat_w, we, clk, rst, adr);
  input [2:0] adr;
  wire [2:0] adr;
  input clk;
  wire clk;
  output [31:0] dat_r;
  wire [31:0] dat_r;
  input [31:0] dat_w;
  wire [31:0] dat_w;
  wire [2:0] mem_r_addr;
  wire [31:0] mem_r_data;
  wire [2:0] mem_w_addr;
  wire [31:0] mem_w_data;
  wire mem_w_en;
  input rst;
  wire rst;
  input we;
  wire we;
  reg [31:0] mem [7:0];
  initial begin
    mem[0] = 32'd0;
    mem[1] = 32'd0;
    mem[2] = 32'd0;
    mem[3] = 32'd0;
    mem[4] = 32'd0;
    mem[5] = 32'd0;
    mem[6] = 32'd0;
    mem[7] = 32'd0;
  end
  always @(posedge clk) begin
    if (mem_w_en)
      mem[mem_w_addr] <= mem_w_data;
  end
  reg [2:0] _0_;
  always @(posedge clk) begin
    _0_ <= mem_r_addr;
  end
  assign mem_r_data = mem[_0_];
  assign mem_w_en = we;
  assign mem_w_data = dat_w;
  assign mem_w_addr = adr;
  assign dat_r = mem_r_data;
  assign mem_r_addr = adr;
endmodule
```

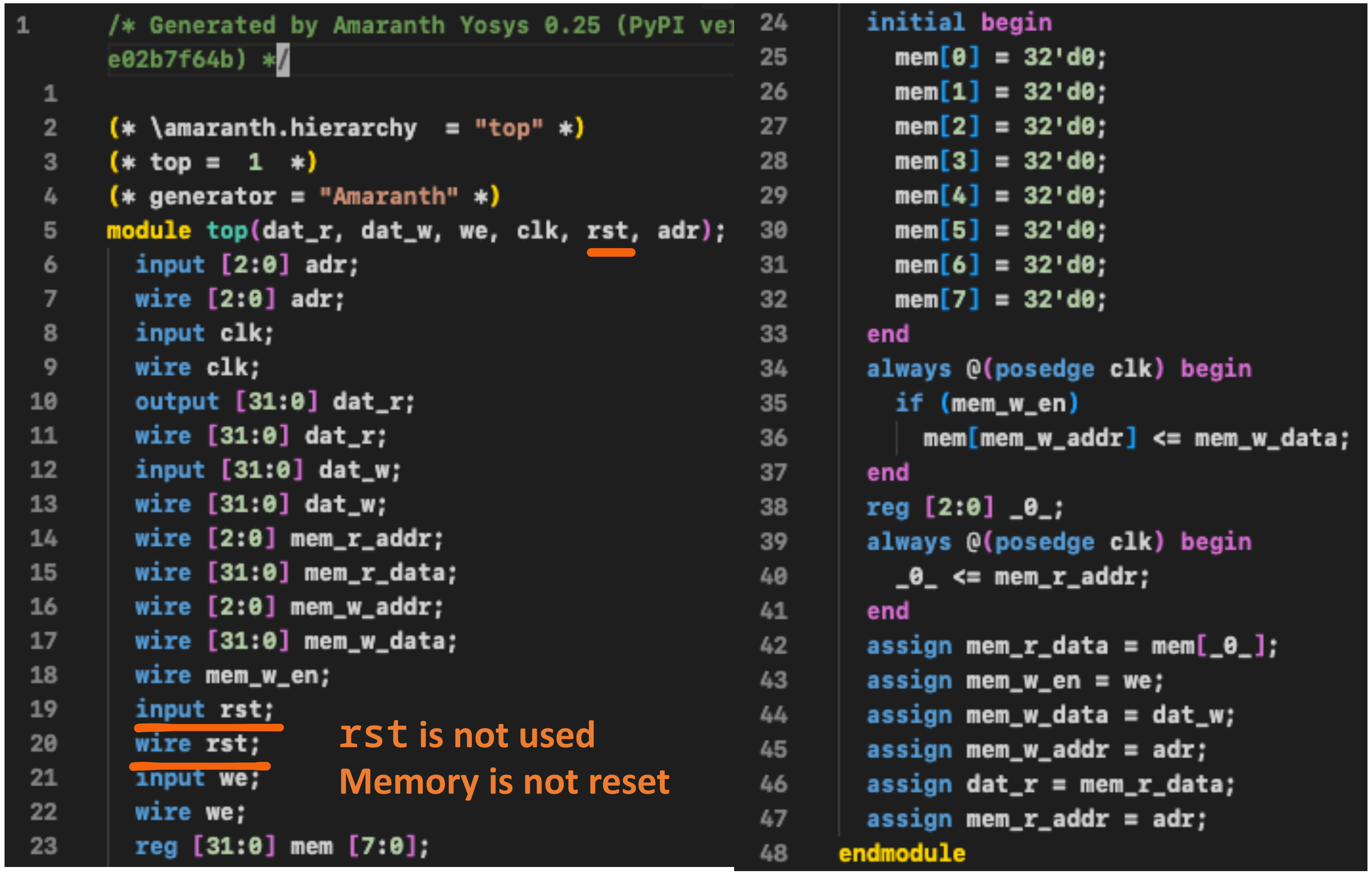
Memory



```
/* Generated by Amaranth Yosys 0.25 (PyPI ve
e02b7f64b) */

(* \amaranth.hierarchy  = "top" *)
(* top =  1  *)
(* generator = "Amaranth" *)
module top(dat_r, dat_w, we, clk, rst, adr);
  input [2:0] adr;
  wire [2:0] adr;
  input clk;
  wire clk;
  output [31:0] dat_r;
  wire [31:0] dat_r;
  input [31:0] dat_w;
  wire [31:0] dat_w;
  wire [2:0] mem_r_addr;
  wire [31:0] mem_r_data;
  wire [2:0] mem_w_addr;
  wire [31:0] mem_w_data;
  wire mem_w_en;
  input rst;
  wire rst;
  input we;
  wire we;
  reg [31:0] mem [7:0];
  initial begin
    mem[0] = 32'd0;
    mem[1] = 32'd0;
    mem[2] = 32'd0;
    mem[3] = 32'd0;
    mem[4] = 32'd0;
    mem[5] = 32'd0;
    mem[6] = 32'd0;
    mem[7] = 32'd0;
  end
  always @(posedge clk) begin
    if (mem_w_en)
      mem[mem_w_addr] <= mem_w_data;
  end
  reg [2:0] _0_;
  always @(posedge clk) begin
    _0_ <= mem_r_addr;
  end
  assign mem_r_data = mem[_0_];
  assign mem_w_en = we;
  assign mem_w_data = dat_w;
  assign mem_w_addr = adr;
  assign dat_r = mem_r_data;
  assign mem_r_addr = adr;
endmodule
```

Mapped to 2D register

Memory



```
/* Generated by Amaranth Yosys 0.25 (PyPI ve
e02b7f64b) */

(* \amaranth.hierarchy  = "top" *)
(* top =  1  *)
(* generator = "Amaranth" *)
module top(dat_r, dat_w, we, clk, rst, adr);
  input [2:0] adr;
  wire [2:0] adr;
  input clk;
  wire clk;
  output [31:0] dat_r;
  wire [31:0] dat_r;
  input [31:0] dat_w;
  wire [31:0] dat_w;
  wire [2:0] mem_r_addr;
  wire [31:0] mem_r_data;
  wire [2:0] mem_w_addr;
  wire [31:0] mem_w_data;
  wire mem_w_en;
  input rst;
  wire rst;
  input we;
  wire we;
  reg [31:0] mem [7:0];
  initial begin
    mem[0] = 32'd0;
    mem[1] = 32'd0;
    mem[2] = 32'd0;
    mem[3] = 32'd0;
    mem[4] = 32'd0;
    mem[5] = 32'd0;
    mem[6] = 32'd0;
    mem[7] = 32'd0;
  end
  always @(posedge clk) begin
    if (mem_w_en)
      mem[mem_w_addr] <= mem_w_data;
  end
  reg [2:0] _0_;
  always @(posedge clk) begin
    _0_ <= mem_r_addr;
  end
  assign mem_r_data = mem[_0_];
  assign mem_w_en = we;
  assign mem_w_data = dat_w;
  assign mem_w_addr = adr;
  assign dat_r = mem_r_data;
  assign mem_r_addr = adr;
endmodule
```

`rst` is not used
Memory is not reset

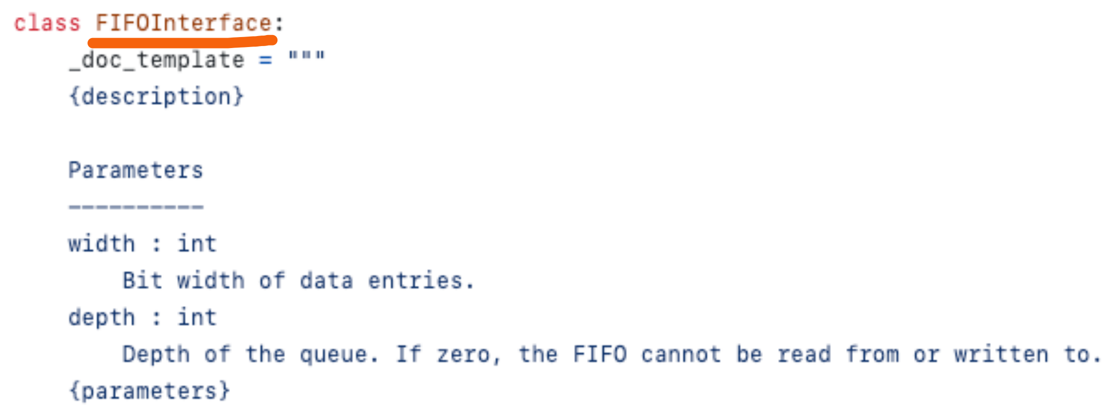
Q&A on Memory

FIFO

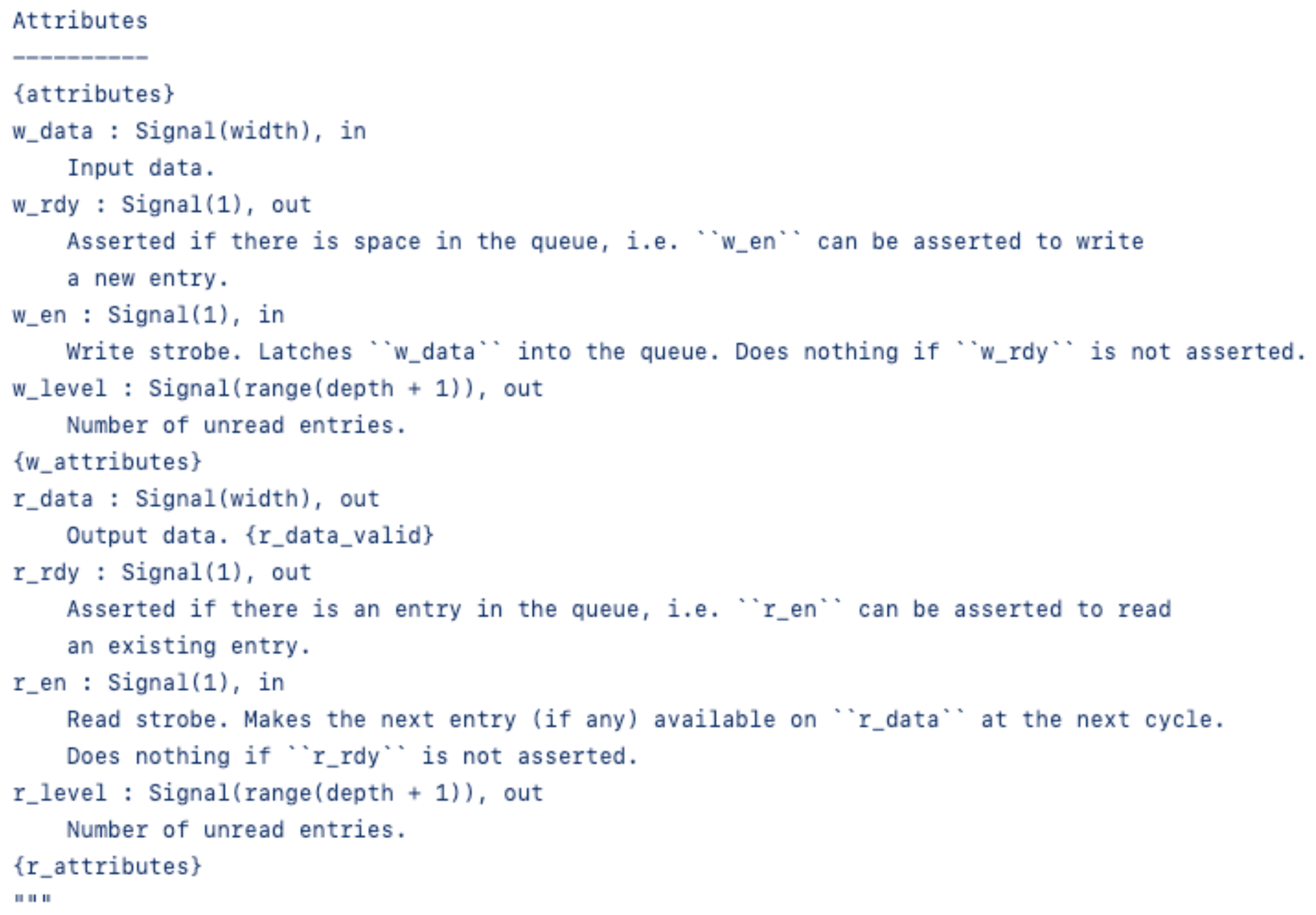
- **`SyncFIFO, SyncFIFOBuffered`**
- **`AsyncFIFO, AsyncFIFOBuffered`**
  - For CDC(Clock Domain Crossing)
- Use **`Memory`** as construct

amaranth/fifo.py (github.com)

FIFO



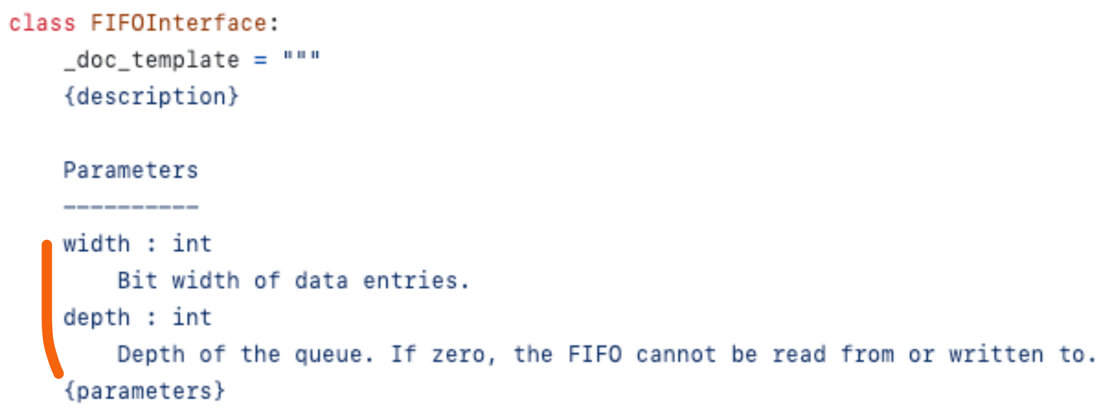
```
class FIFOInterface:
    _doc_template = """
    {description}

    Parameters
    ----------
    width : int
        Bit width of data entries.
    depth : int
        Depth of the queue. If zero, the FIFO cannot be read from or written to.
    {parameters}
```



```
Attributes
----------
{attributes}
w_data : Signal(width), in
    Input data.
w_rdy : Signal(1), out
    Asserted if there is space in the queue, i.e. ``w_en`` can be asserted to write
    a new entry.
w_en : Signal(1), in
    Write strobe. Latches ``w_data`` into the queue. Does nothing if ``w_rdy`` is not asserted.
w_level : Signal(range(depth + 1)), out
    Number of unread entries.
{w_attributes}
r_data : Signal(width), out
    Output data. {r_data_valid}
r_rdy : Signal(1), out
    Asserted if there is an entry in the queue, i.e. ``r_en`` can be asserted to read
    an existing entry.
r_en : Signal(1), in
    Read strobe. Makes the next entry (if any) available on ``r_data`` at the next cycle.
    Does nothing if ``r_rdy`` is not asserted.
r_level : Signal(range(depth + 1)), out
    Number of unread entries.
{r_attributes}
"""
```

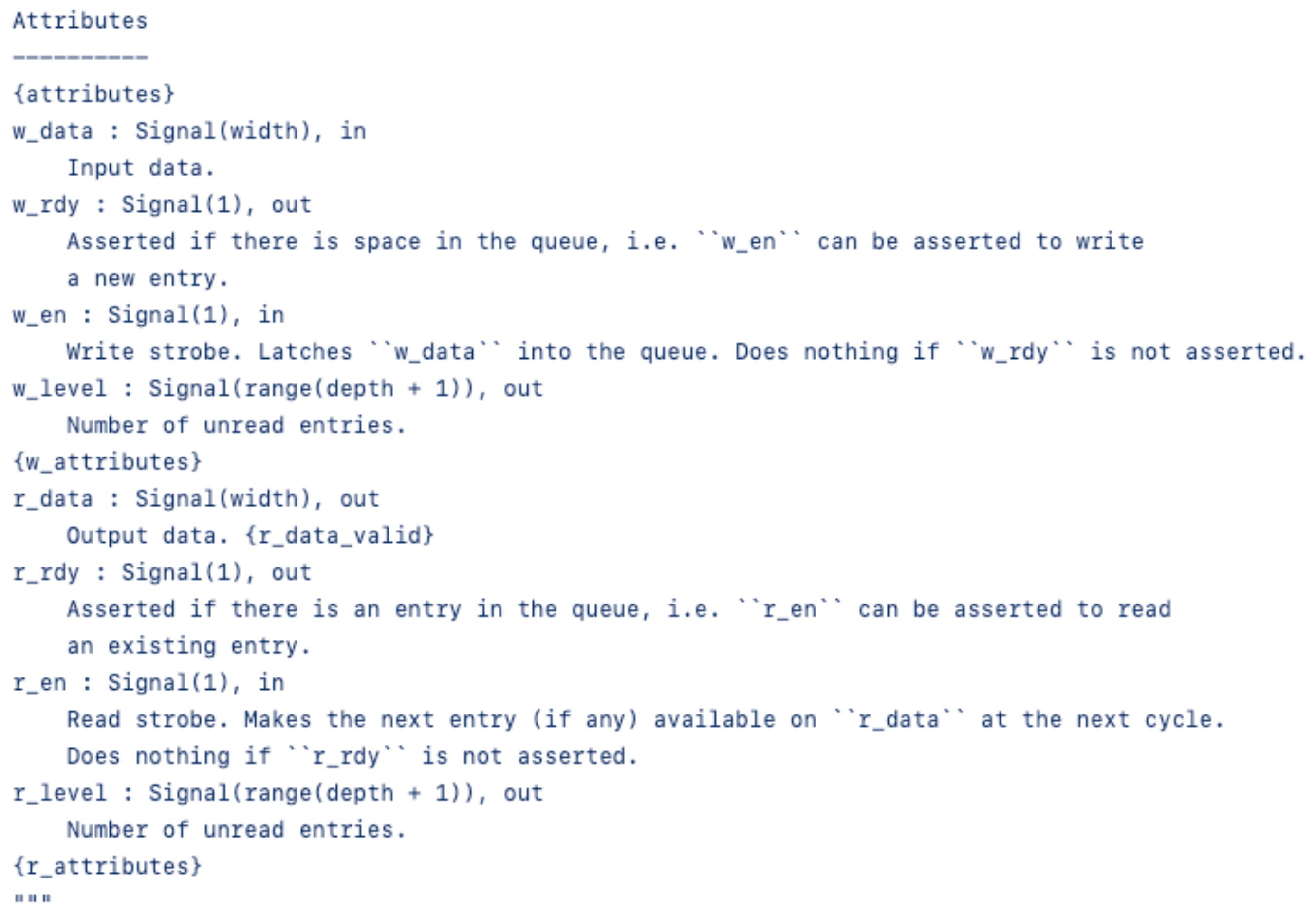
amaranth/fifo.py (github.com)

FIFO



```
class FIFOInterface:
    _doc_template = """
    {description}

    Parameters
    ----------
    width : int
        Bit width of data entries.
    depth : int
        Depth of the queue. If zero, the FIFO cannot be read from or written to.
    {parameters}
```

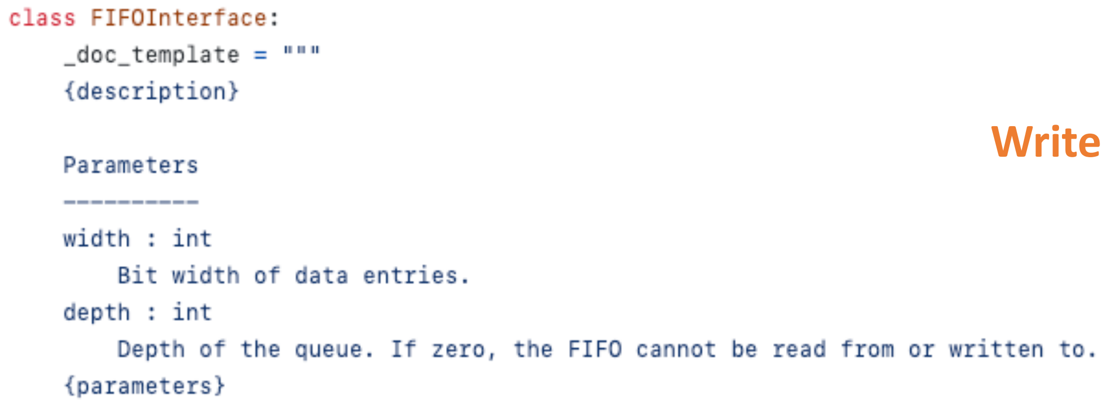
Same as Memory



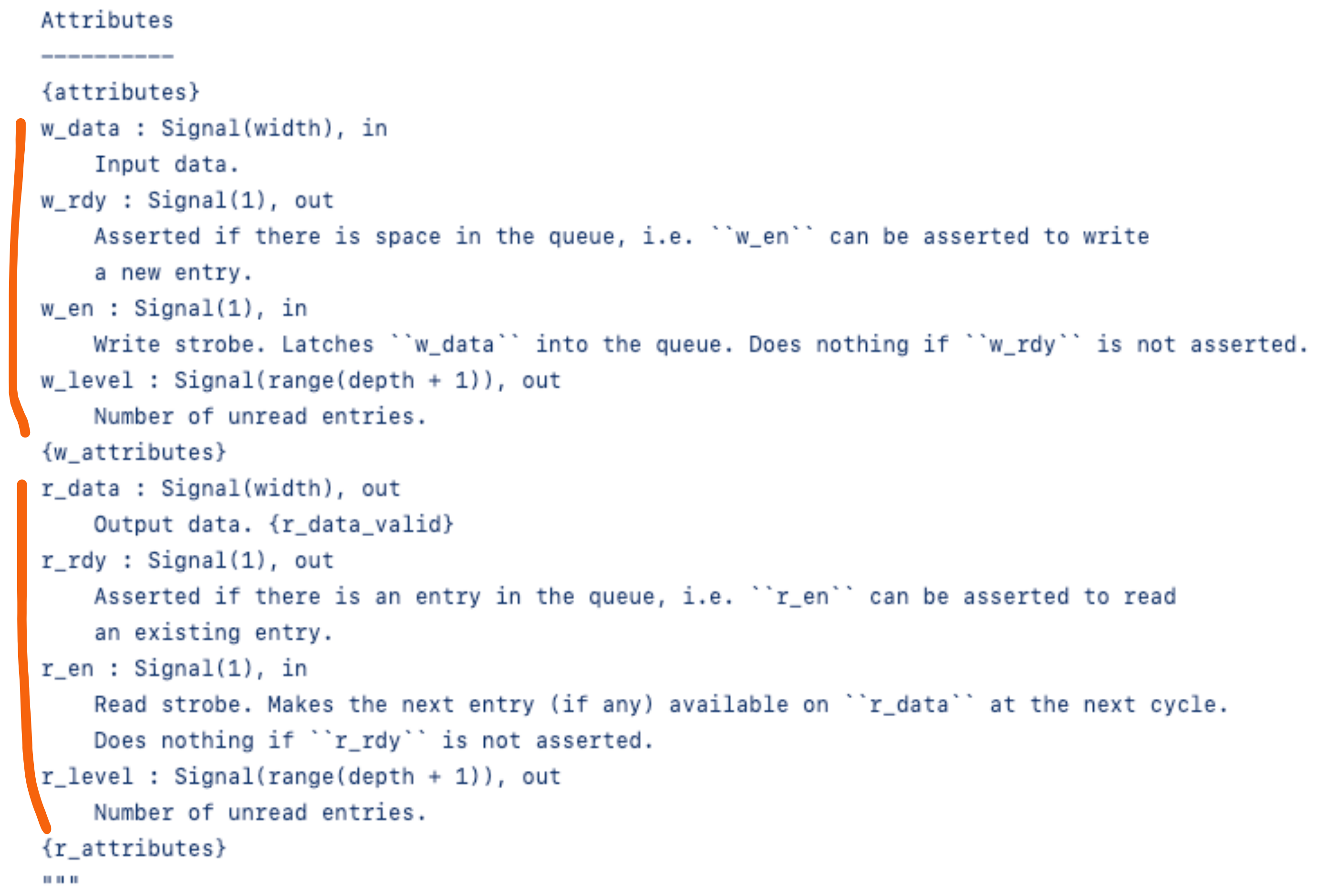
```
Attributes
----------
{attributes}
w_data : Signal(width), in
    Input data.
w_rdy : Signal(1), out
    Asserted if there is space in the queue, i.e. ``w_en`` can be asserted to write
    a new entry.
w_en : Signal(1), in
    Write strobe. Latches ``w_data`` into the queue. Does nothing if ``w_rdy`` is not asserted.
w_level : Signal(range(depth + 1)), out
    Number of unread entries.
{w_attributes}
r_data : Signal(width), out
    Output data. {r_data_valid}
r_rdy : Signal(1), out
    Asserted if there is an entry in the queue, i.e. ``r_en`` can be asserted to read
    an existing entry.
r_en : Signal(1), in
    Read strobe. Makes the next entry (if any) available on ``r_data`` at the next cycle.
    Does nothing if ``r_rdy`` is not asserted.
r_level : Signal(range(depth + 1)), out
    Number of unread entries.
{r_attributes}
"""
```

amaranth/fifo.py (github.com)

FIFO



```
class FIFOInterface:
    _doc_template = """
    {description}

    Parameters
    ----------
    width : int
        Bit width of data entries.
    depth : int
        Depth of the queue. If zero, the FIFO cannot be read from or written to.
    {parameters}
```



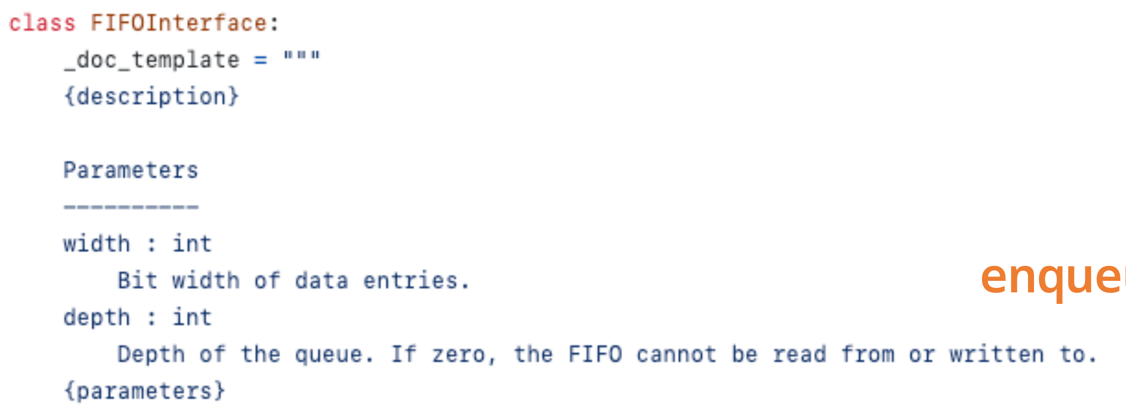
Write

Read

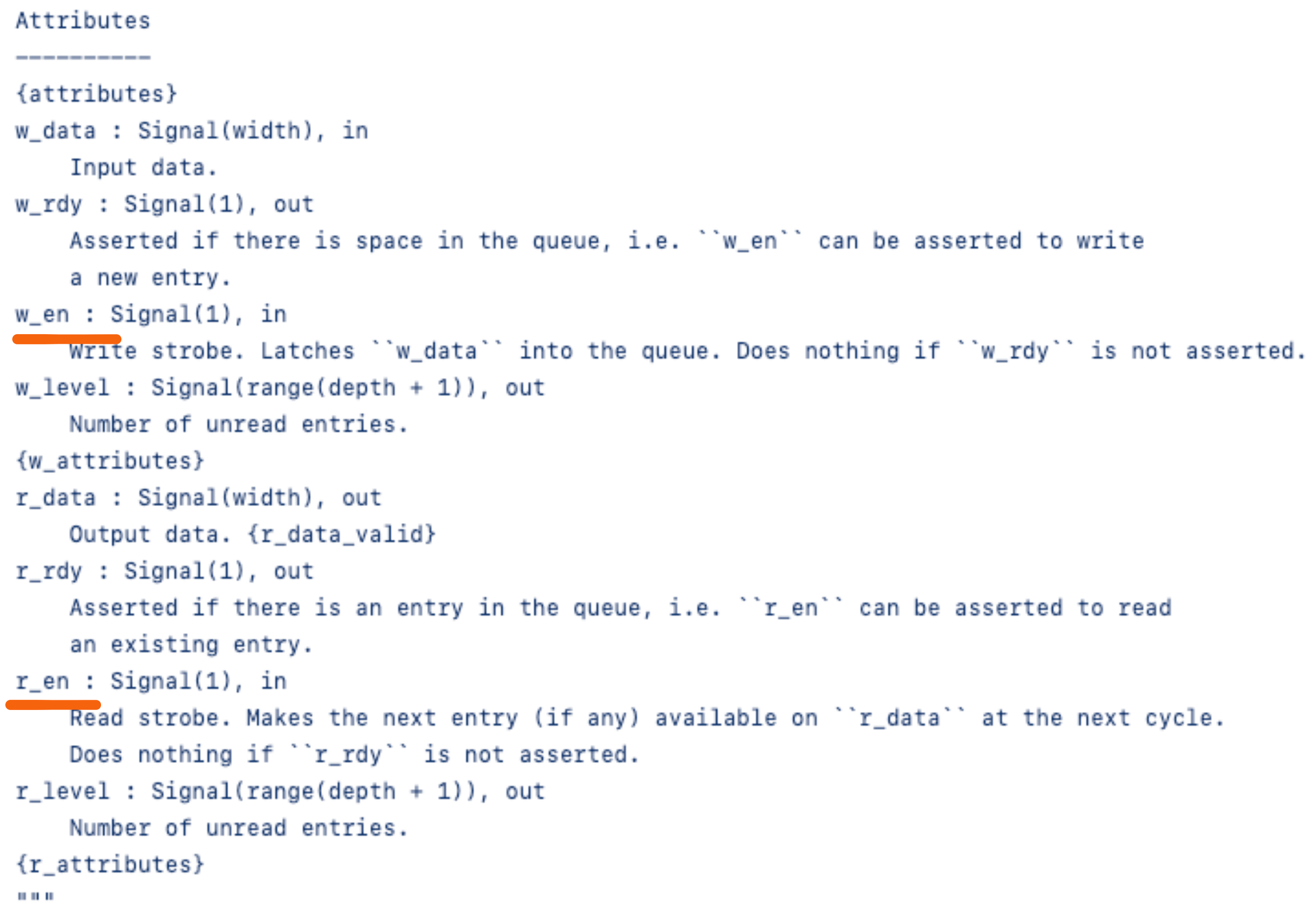
```
Attributes
----------
{attributes}
w_data : Signal(width), in
    Input data.
w_rdy : Signal(1), out
    Asserted if there is space in the queue, i.e. ``w_en`` can be asserted to write
    a new entry.
w_en : Signal(1), in
    Write strobe. Latches ``w_data`` into the queue. Does nothing if ``w_rdy`` is not asserted.
w_level : Signal(range(depth + 1)), out
    Number of unread entries.
{w_attributes}
r_data : Signal(width), out
    Output data. {r_data_valid}
r_rdy : Signal(1), out
    Asserted if there is an entry in the queue, i.e. ``r_en`` can be asserted to read
    an existing entry.
r_en : Signal(1), in
    Read strobe. Makes the next entry (if any) available on ``r_data`` at the next cycle.
    Does nothing if ``r_rdy`` is not asserted.
r_level : Signal(range(depth + 1)), out
    Number of unread entries.
{r_attributes}
"""
```

amaranth/fifo.py (github.com)

FIFO



```
class FIFOInterface:
    _doc_template = """
    {description}

    Parameters
    ----------
    width : int
        Bit width of data entries.
    depth : int
        Depth of the queue. If zero, the FIFO cannot be read from or written to.
    {parameters}
```



```
Attributes
----------
{attributes}
w_data : Signal(width), in
    Input data.
w_rdy : Signal(1), out
    Asserted if there is space in the queue, i.e. ``w_en`` can be asserted to write
    a new entry.
```

enqueue

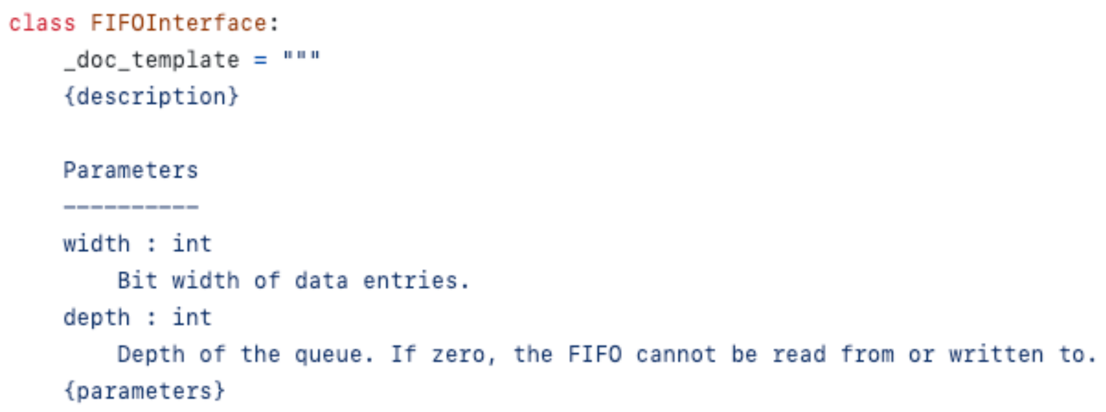
```
w_en : Signal(1), in
    Write strobe. Latches ``w_data`` into the queue. Does nothing if ``w_rdy`` is not asserted.
w_level : Signal(range(depth + 1)), out
    Number of unread entries.
{w_attributes}
r_data : Signal(width), out
    Output data. {r_data_valid}
r_rdy : Signal(1), out
    Asserted if there is an entry in the queue, i.e. ``r_en`` can be asserted to read
    an existing entry.
```

dequeue

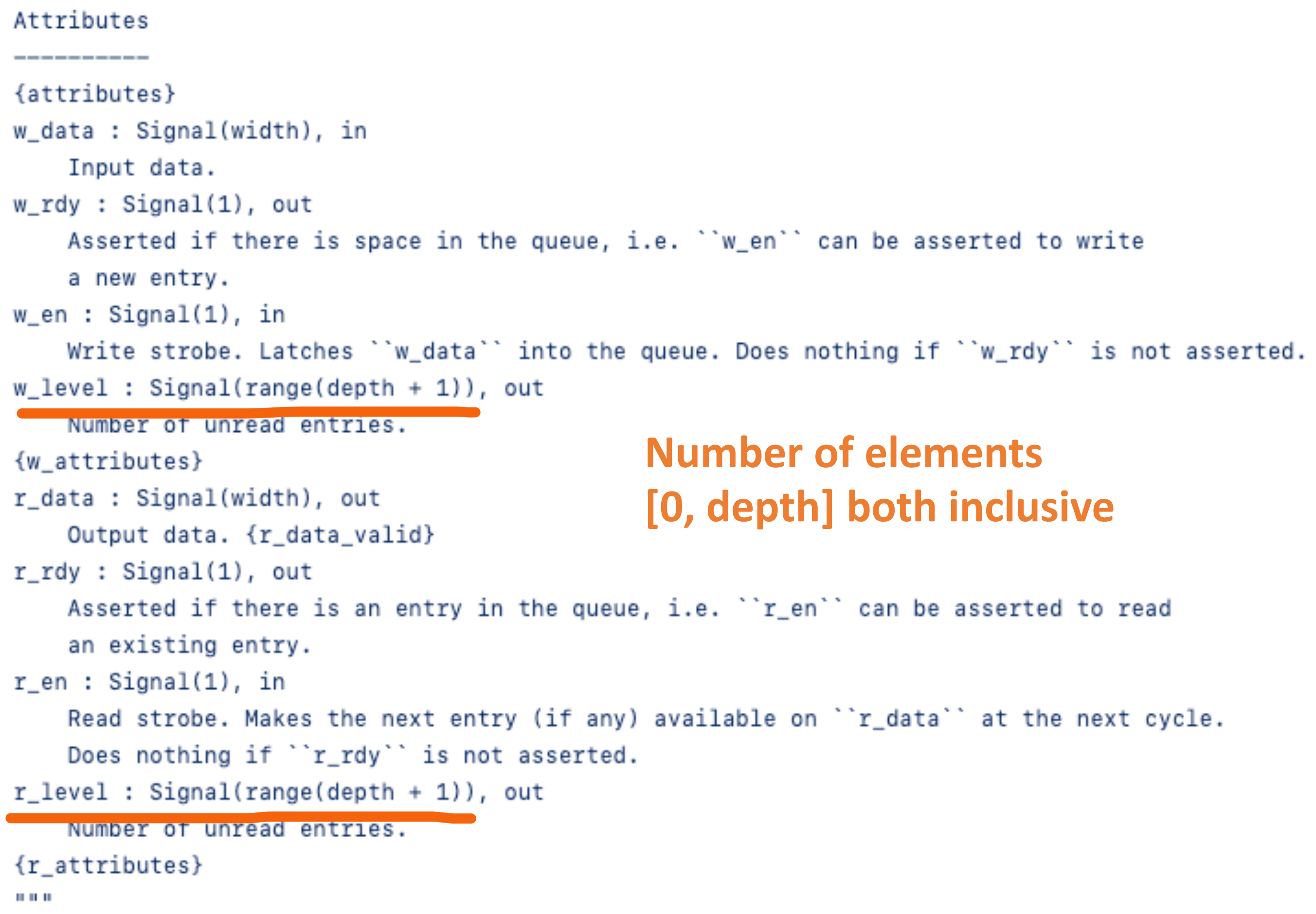
```
r_en : Signal(1), in
    Read strobe. Makes the next entry (if any) available on ``r_data`` at the next cycle.
    Does nothing if ``r_rdy`` is not asserted.
r_level : Signal(range(depth + 1)), out
    Number of unread entries.
{r_attributes}
"""
```

amaranth/fifo.py (github.com)

FIFO



```
class FIFOInterface:
    _doc_template = """
    {description}

    Parameters
    ----------
    width : int
        Bit width of data entries.
    depth : int
        Depth of the queue. If zero, the FIFO cannot be read from or written to.
    {parameters}
```

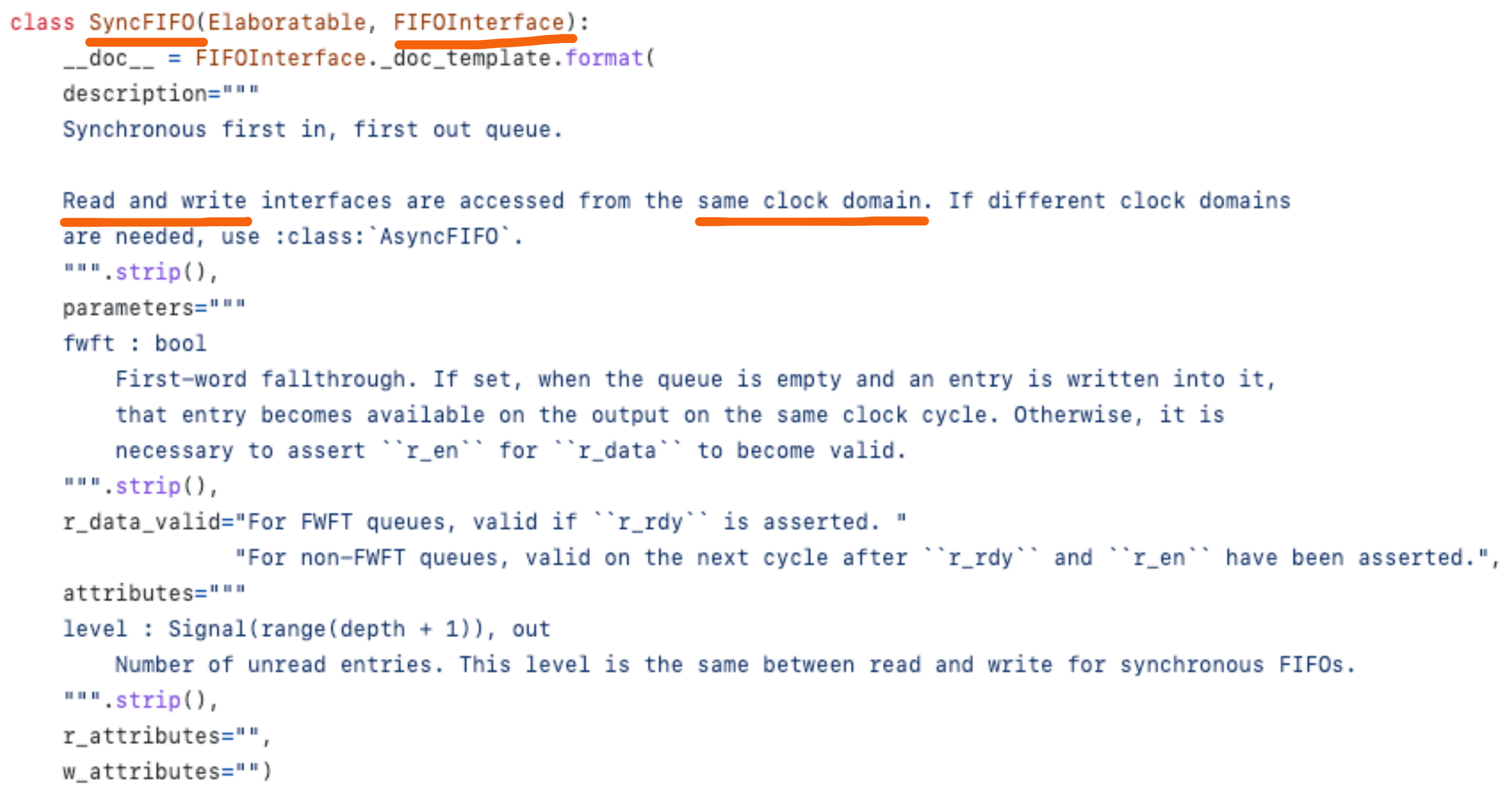


```
Attributes
----------
{attributes}
w_data : Signal(width), in
    Input data.
w_rdy : Signal(1), out
    Asserted if there is space in the queue, i.e. ``w_en`` can be asserted to write
    a new entry.
w_en : Signal(1), in
    Write strobe. Latches ``w_data`` into the queue. Does nothing if ``w_rdy`` is not asserted.
w_level : Signal(range(depth + 1)), out
    Number of unread entries.
{w_attributes}
r_data : Signal(width), out
    Output data. {r_data_valid}
r_rdy : Signal(1), out
    Asserted if there is an entry in the queue, i.e. ``r_en`` can be asserted to read
    an existing entry.
r_en : Signal(1), in
    Read strobe. Makes the next entry (if any) available on ``r_data`` at the next cycle.
    Does nothing if ``r_rdy`` is not asserted.
r_level : Signal(range(depth + 1)), out
    Number of unread entries.
{r_attributes}
"""
```

Number of elements
[0, depth] both inclusive

amaranth/fifo.py (github.com)

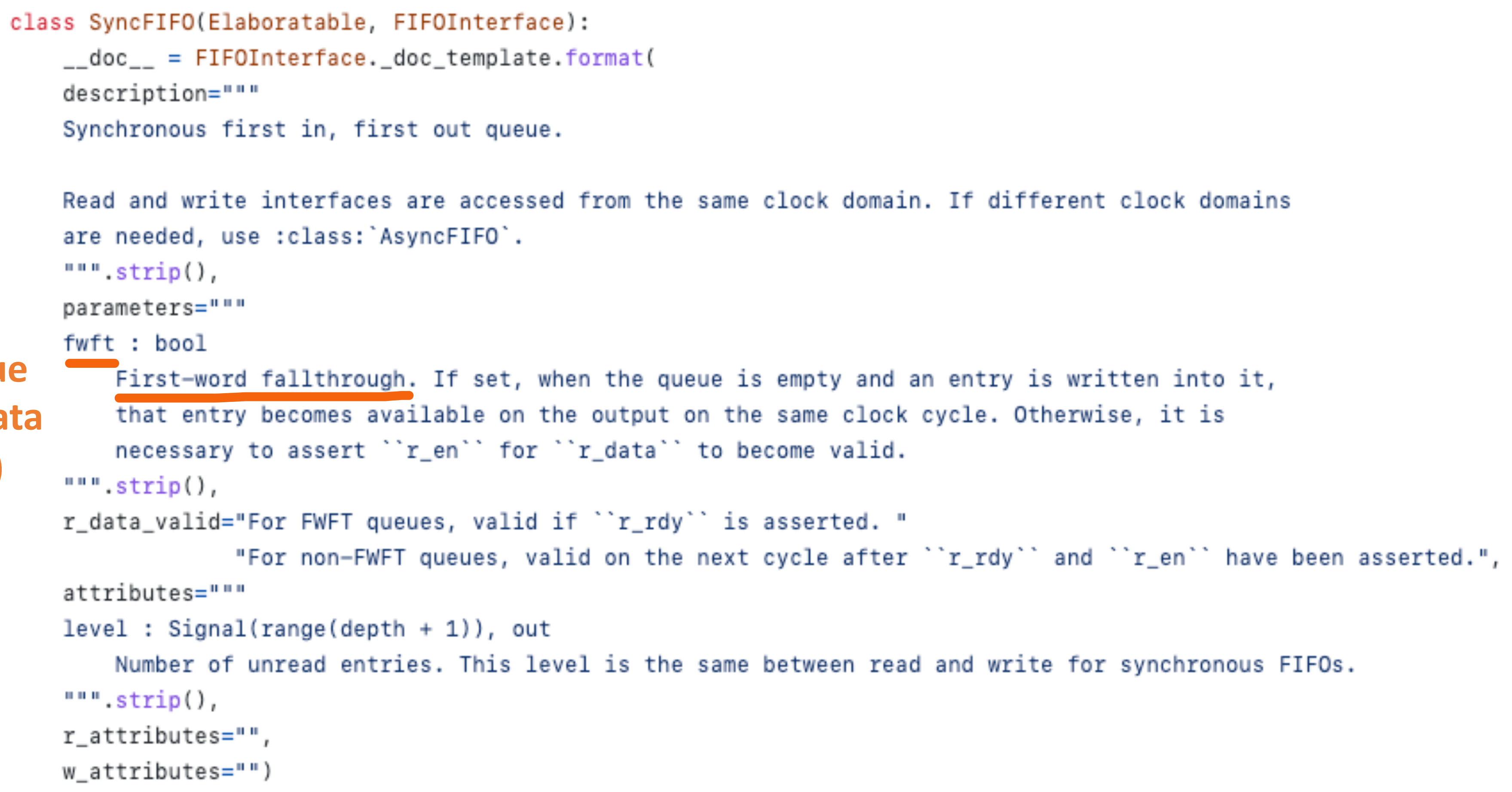
FIFO



```
class SyncFIFO(Elaboratable, FIFOInterface):
    __doc__ = FIFOInterface._doc_template.format(
    description="""
    Synchronous first in, first out queue.

    Read and write interfaces are accessed from the same clock domain. If different clock domains
    are needed, use :class:`AsyncFIFO`.
    """.strip(),
    parameters="""
    fwft : bool
        First-word fallthrough. If set, when the queue is empty and an entry is written into it,
        that entry becomes available on the output on the same clock cycle. Otherwise, it is
        necessary to assert ``r_en`` for ``r_data`` to become valid.
    """.strip(),
    r_data_valid="For FWFT queues, valid if ``r_rdy`` is asserted. "
                 "For non-FWFT queues, valid on the next cycle after ``r_rdy`` and ``r_en`` have been asserted.",
    attributes="""
    level : Signal(range(depth + 1)), out
        Number of unread entries. This level is the same between read and write for synchronous FIFOs.
    """.strip(),
    r_attributes="",
    w_attributes="")
```

amaranth/fifo.py (github.com)

FIFO

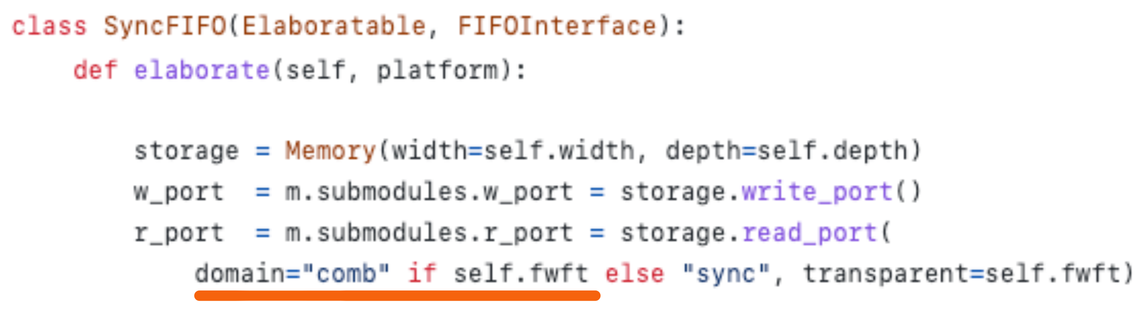


```
class SyncFIFO(Elaboratable, FIFOInterface):
    __doc__ = FIFOInterface._doc_template.format(
    description="""
    Synchronous first in, first out queue.

    Read and write interfaces are accessed from the same clock domain. If different clock domains
    are needed, use :class:`AsyncFIFO`.
    """.strip(),
    parameters="""
    fwft : bool
        First-word fallthrough. If set, when the queue is empty and an entry is written into it,
        that entry becomes available on the output on the same clock cycle. Otherwise, it is
        necessary to assert ``r_en`` for ``r_data`` to become valid.
    """.strip(),
    r_data_valid="For FWFT queues, valid if ``r_rdy`` is asserted. "
                 "For non-FWFT queues, valid on the next cycle after ``r_rdy`` and ``r_en`` have been asserted.",
    attributes="""
    level : Signal(range(depth + 1)), out
        Number of unread entries. This level is the same between read and write for synchronous FIFOs.
    """.strip(),
    r_attributes="",
    w_attributes="")
```

Empty & enqueue → ready on r_data (True by default)

[amaranth/fifo.py (github.com)](https://github.com)

FIFO

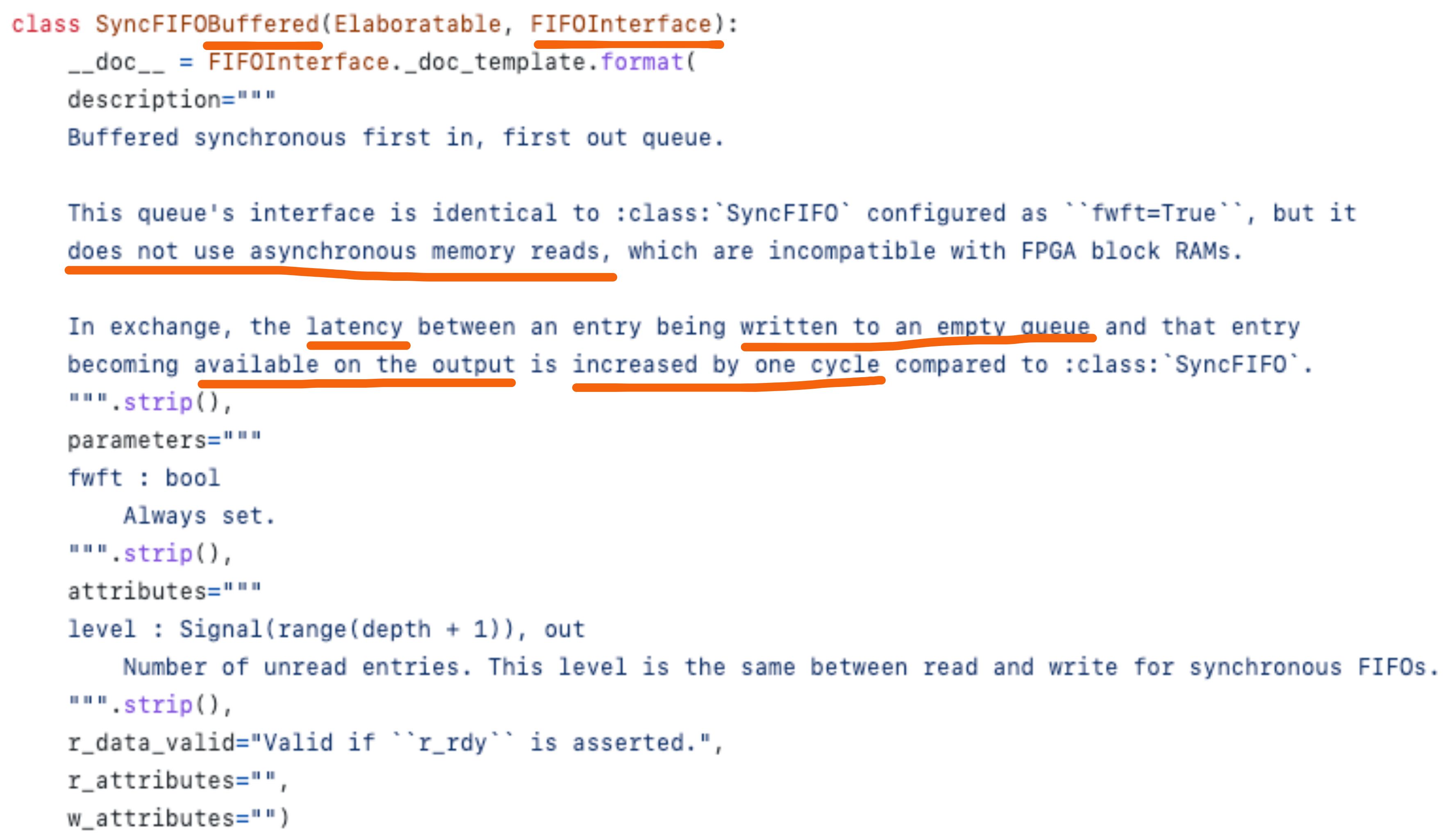


```
class SyncFIFO(Elaboratable, FIFOInterface):
    def elaborate(self, platform):

        storage = Memory(width=self.width, depth=self.depth)
        w_port  = m.submodules.w_port = storage.write_port()
        r_port  = m.submodules.r_port = storage.read_port(
            domain="comb" if self.fwft else "sync", transparent=self.fwft)
```

Async read by default

amaranth/fifo.py (github.com)

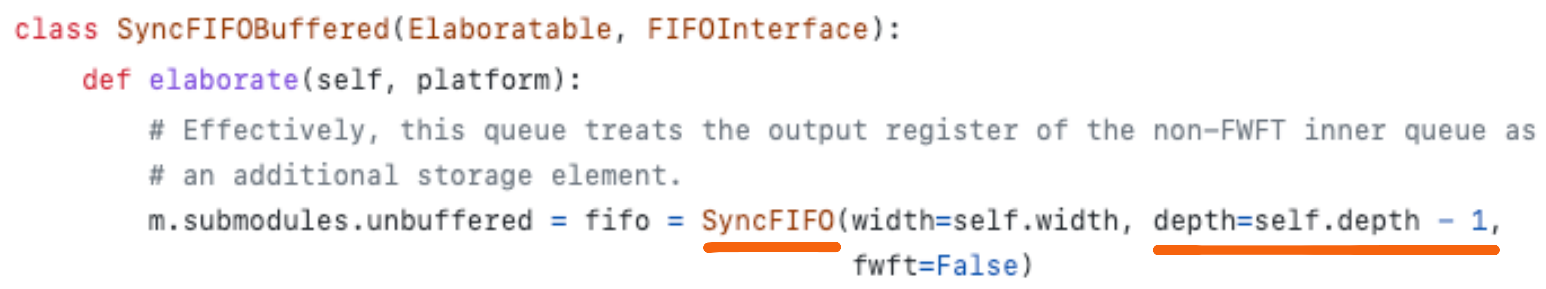
FIFO



```
class SyncFIFOBuffered(Elaboratable, FIFOInterface):
    __doc__ = FIFOInterface._doc_template.format(
    description="""
    Buffered synchronous first in, first out queue.

    This queue's interface is identical to :class:`SyncFIFO` configured as ``fwft=True``, but it
    does not use asynchronous memory reads, which are incompatible with FPGA block RAMs.

    In exchange, the latency between an entry being written to an empty queue and that entry
    becoming available on the output is increased by one cycle compared to :class:`SyncFIFO`.
    """.strip(),
    parameters="""
    fwft : bool
        Always set.
    """.strip(),
    attributes="""
    level : Signal(range(depth + 1)), out
        Number of unread entries. This level is the same between read and write for synchronous FIFOs.
    """.strip(),
    r_data_valid="Valid if ``r_rdy`` is asserted.",
    r_attributes="",
    w_attributes="")
```

[amaranth/fifo.py (github.com)](https://github.com)

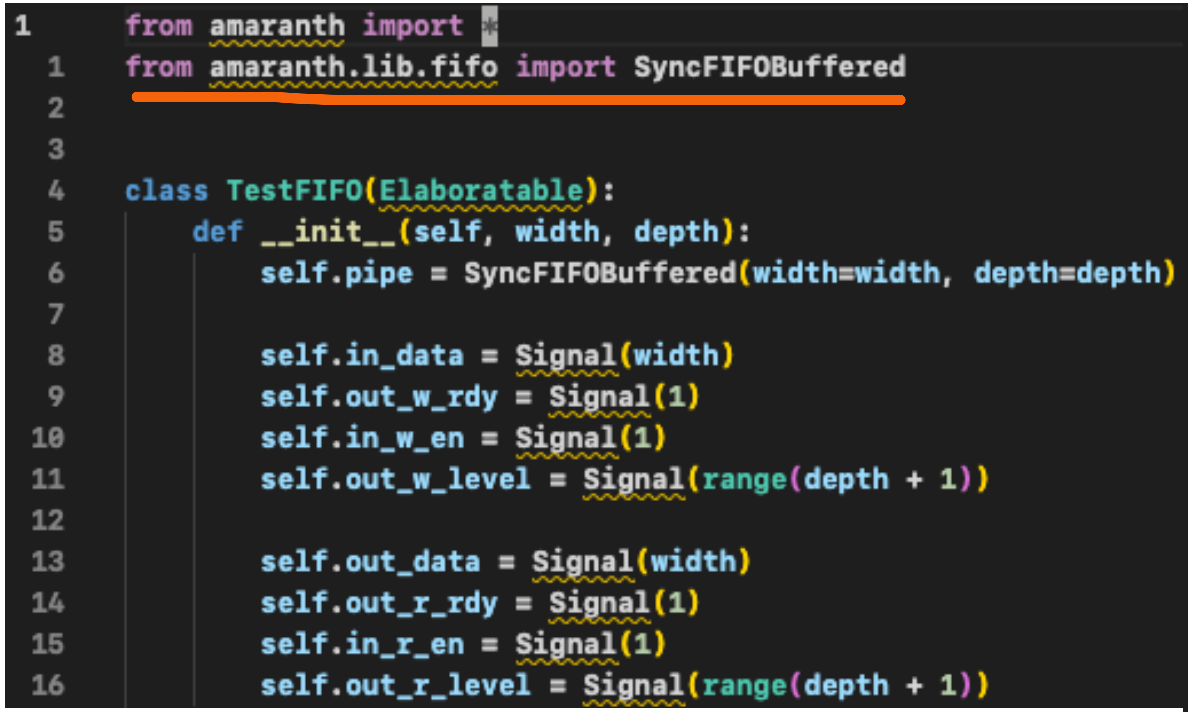
FIFO



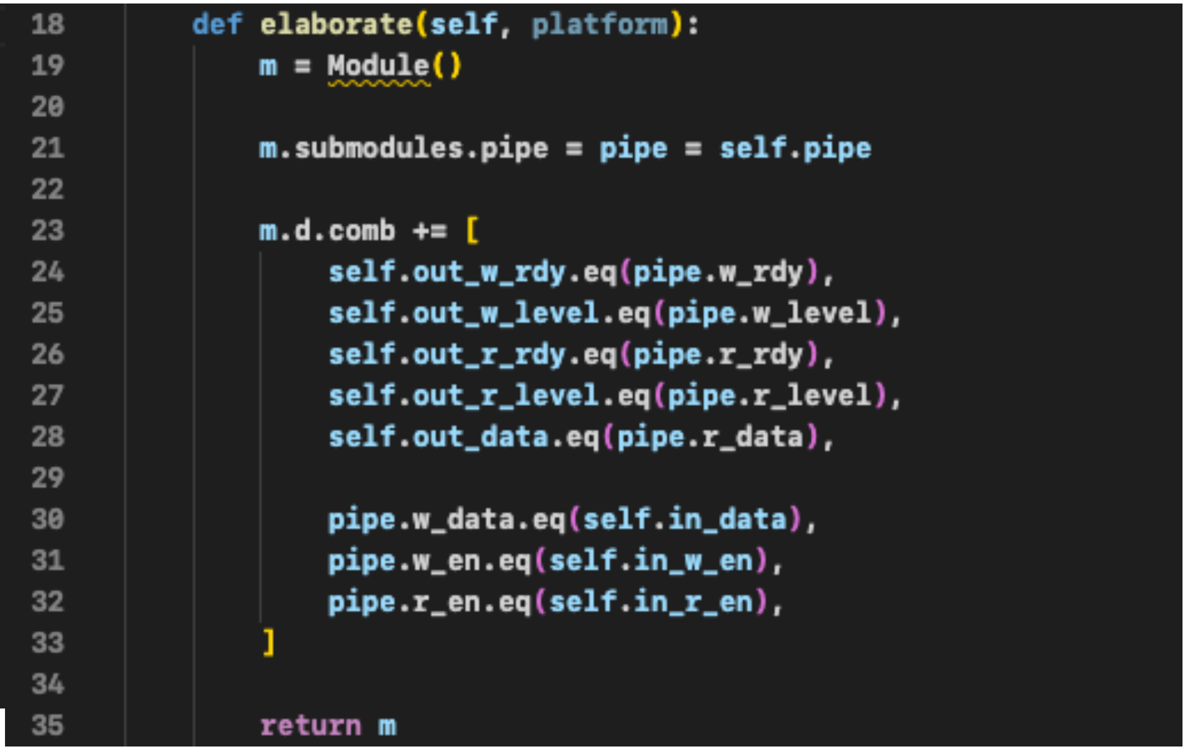
```
class SyncFIFOBuffered(Elaboratable, FIFOInterface):
    def elaborate(self, platform):
        # Effectively, this queue treats the output register of the non-FWFT inner queue as
        # an additional storage element.
        m.submodules.unbuffered = fifo = SyncFIFO(width=self.width, depth=self.depth - 1,
                                                  fwft=False)
```

[amaranth/fifo.py (github.com)](https://github.com)

FIFO

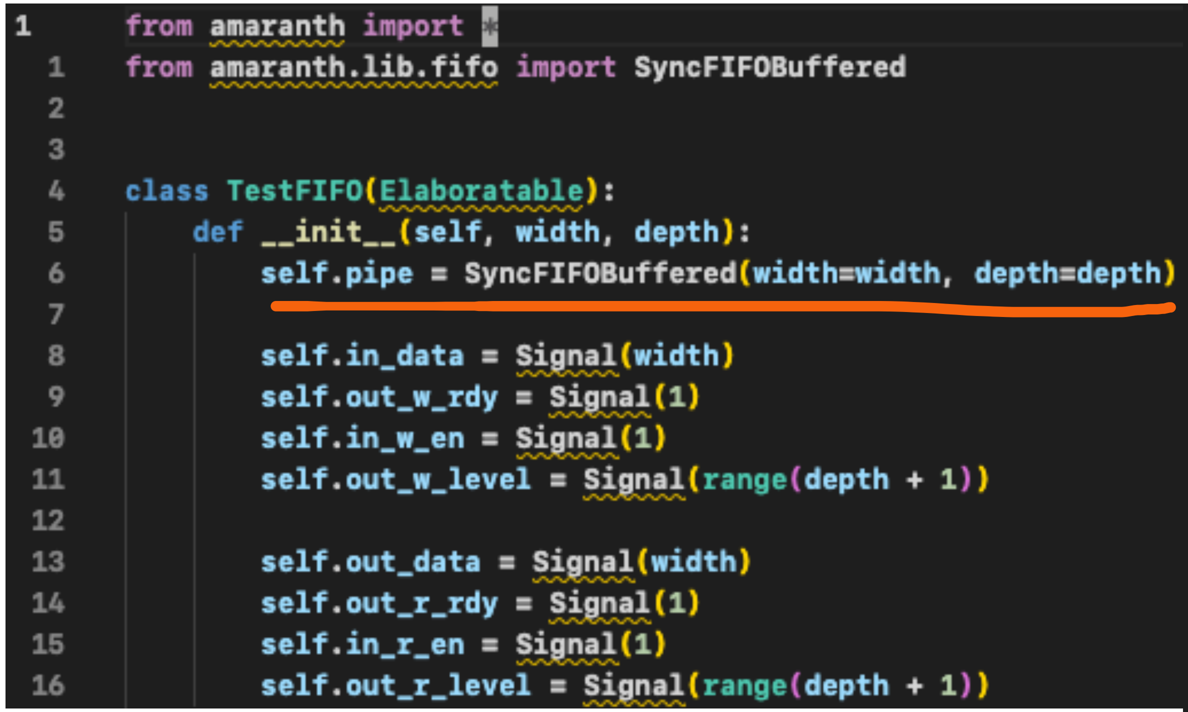


```python
from amaranth import *
from amaranth.lib.fifo import SyncFIFOBuffered


class TestFIFO(Elaboratable):
    def __init__(self, width, depth):
        self.pipe = SyncFIFOBuffered(width=width, depth=depth)

        self.in_data = Signal(width)
        self.out_w_rdy = Signal(1)
        self.in_w_en = Signal(1)
        self.out_w_level = Signal(range(depth + 1))

        self.out_data = Signal(width)
        self.out_r_rdy = Signal(1)
        self.in_r_en = Signal(1)
        self.out_r_level = Signal(range(depth + 1))
```

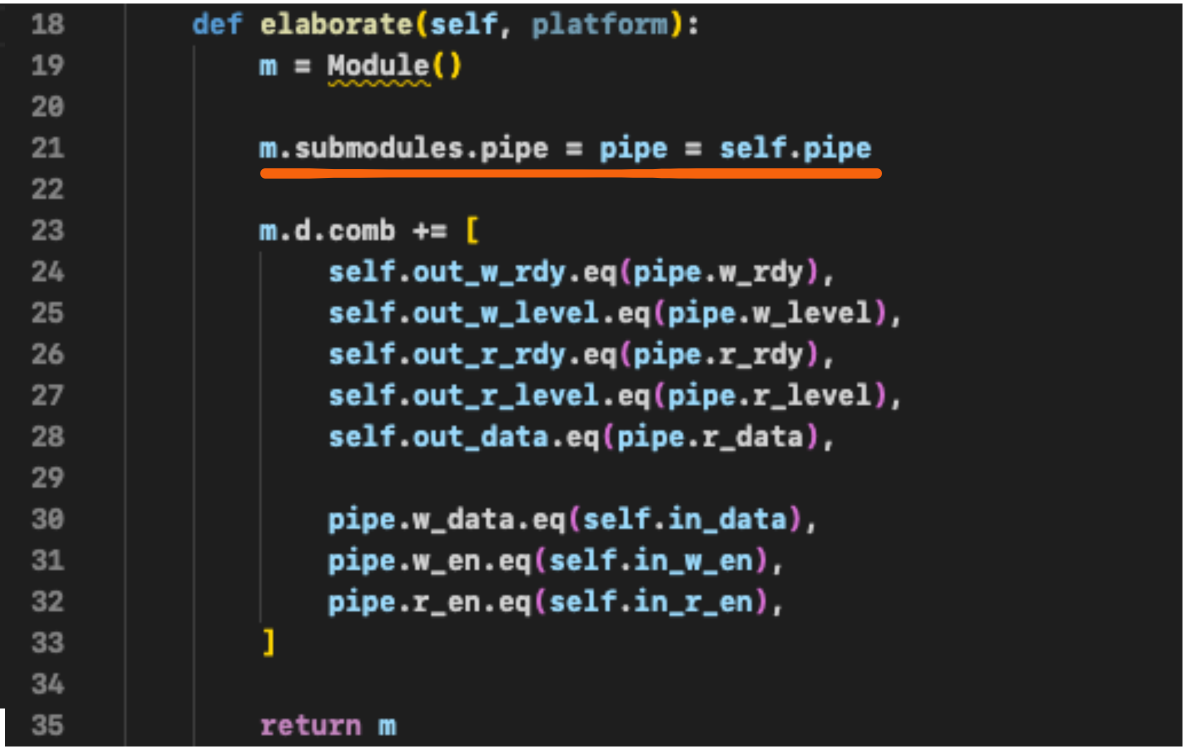


```python
    def elaborate(self, platform):
        m = Module()

        m.submodules.pipe = pipe = self.pipe

        m.d.comb += [
            self.out_w_rdy.eq(pipe.w_rdy),
            self.out_w_level.eq(pipe.w_level),
            self.out_r_rdy.eq(pipe.r_rdy),
            self.out_r_level.eq(pipe.r_level),
            self.out_data.eq(pipe.r_data),

            pipe.w_data.eq(self.in_data),
            pipe.w_en.eq(self.in_w_en),
            pipe.r_en.eq(self.in_r_en),
        ]

        return m
```

FIFO

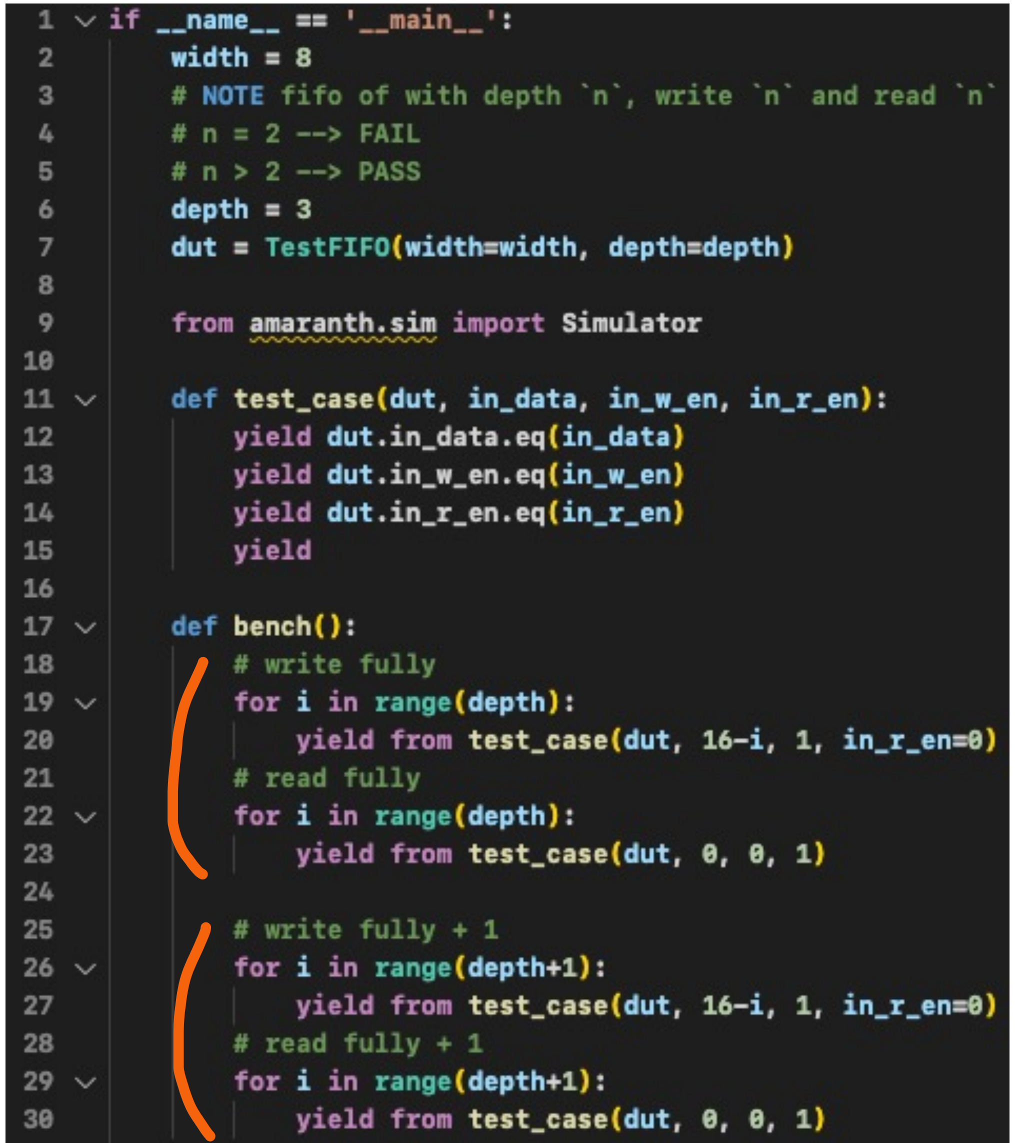


```python
from amaranth import *
from amaranth.lib.fifo import SyncFIFOBuffered


class TestFIFO(Elaboratable):
    def __init__(self, width, depth):
        self.pipe = SyncFIFOBuffered(width=width, depth=depth)

        self.in_data = Signal(width)
        self.out_w_rdy = Signal(1)
        self.in_w_en = Signal(1)
        self.out_w_level = Signal(range(depth + 1))

        self.out_data = Signal(width)
        self.out_r_rdy = Signal(1)
        self.in_r_en = Signal(1)
        self.out_r_level = Signal(range(depth + 1))
```



```python
    def elaborate(self, platform):
        m = Module()

        m.submodules.pipe = pipe = self.pipe

        m.d.comb += [
            self.out_w_rdy.eq(pipe.w_rdy),
            self.out_w_level.eq(pipe.w_level),
            self.out_r_rdy.eq(pipe.r_rdy),
            self.out_r_level.eq(pipe.r_level),
            self.out_data.eq(pipe.r_data),

            pipe.w_data.eq(self.in_data),
            pipe.w_en.eq(self.in_w_en),
            pipe.r_en.eq(self.in_r_en),
        ]

        return m
```

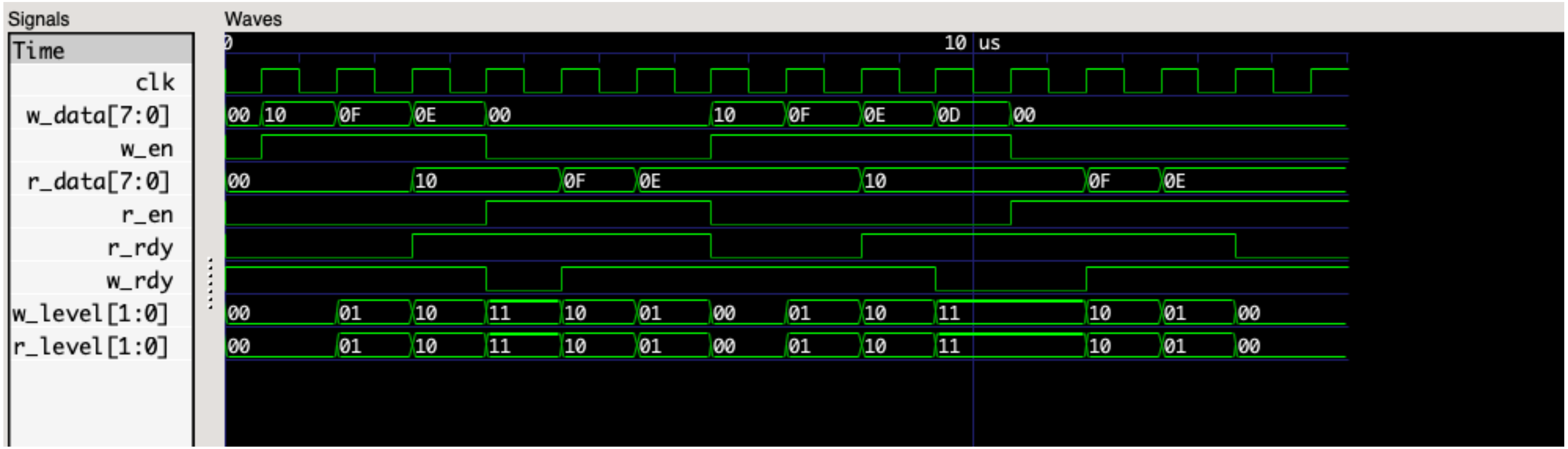
FIFO



```python
if __name__ == '__main__':
    width = 8
    # NOTE fifo of with depth `n`, write `n` and read `n`
    # n = 2 --> FAIL
    # n > 2 --> PASS
    depth = 3
    dut = TestFIFO(width=width, depth=depth)

    from amaranth.sim import Simulator

    def test_case(dut, in_data, in_w_en, in_r_en):
        yield dut.in_data.eq(in_data)
        yield dut.in_w_en.eq(in_w_en)
        yield dut.in_r_en.eq(in_r_en)
        yield

    def bench():
        # write fully
        for i in range(depth):
            yield from test_case(dut, 16-i, 1, in_r_en=0)
        # read fully
        for i in range(depth):
            yield from test_case(dut, 0, 0, 1)

        # write fully + 1
        for i in range(depth+1):
            yield from test_case(dut, 16-i, 1, in_r_en=0)
        # read fully + 1
        for i in range(depth+1):
            yield from test_case(dut, 0, 0, 1)
```

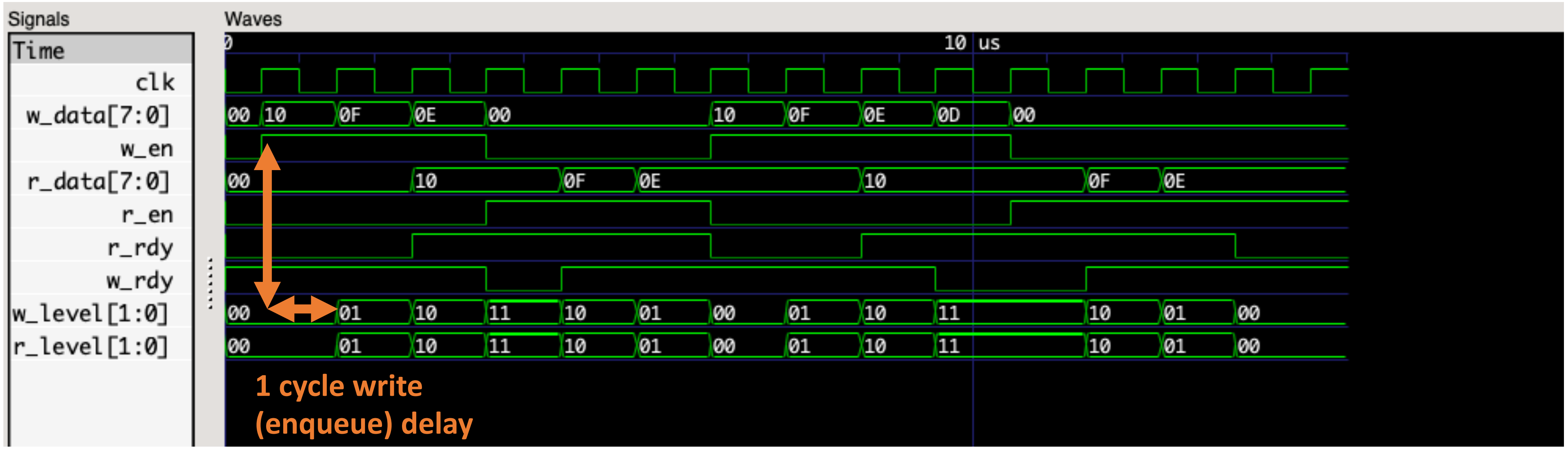
FIFO

- SyncFIFOBuffered (**depth=3**)



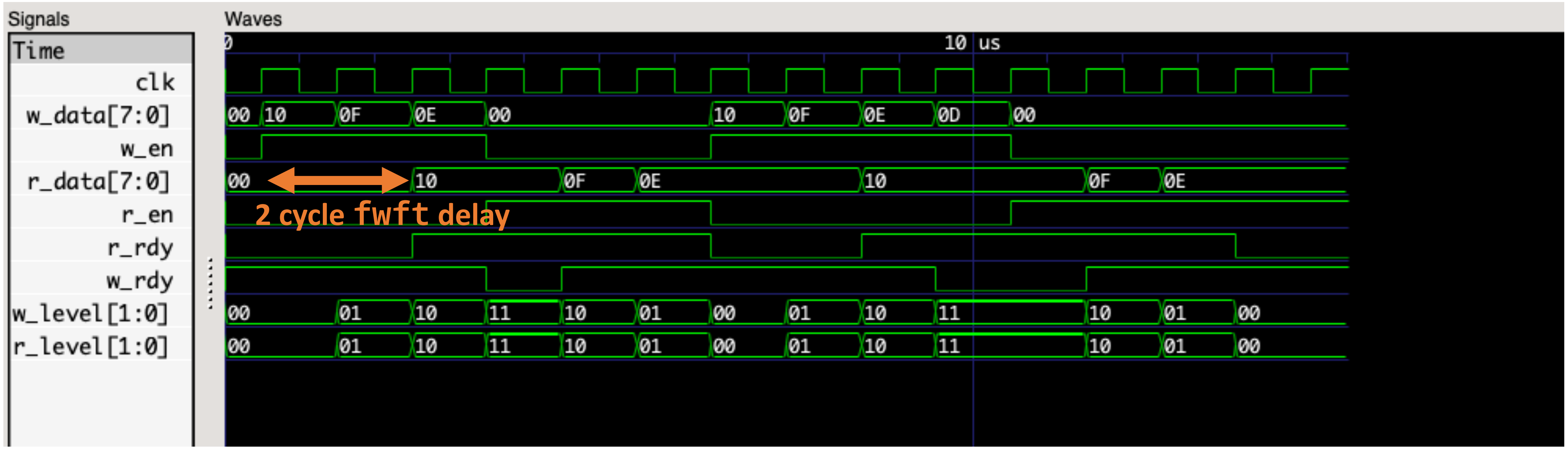
FIFO

- SyncFIFOBuffered (**depth=3**)



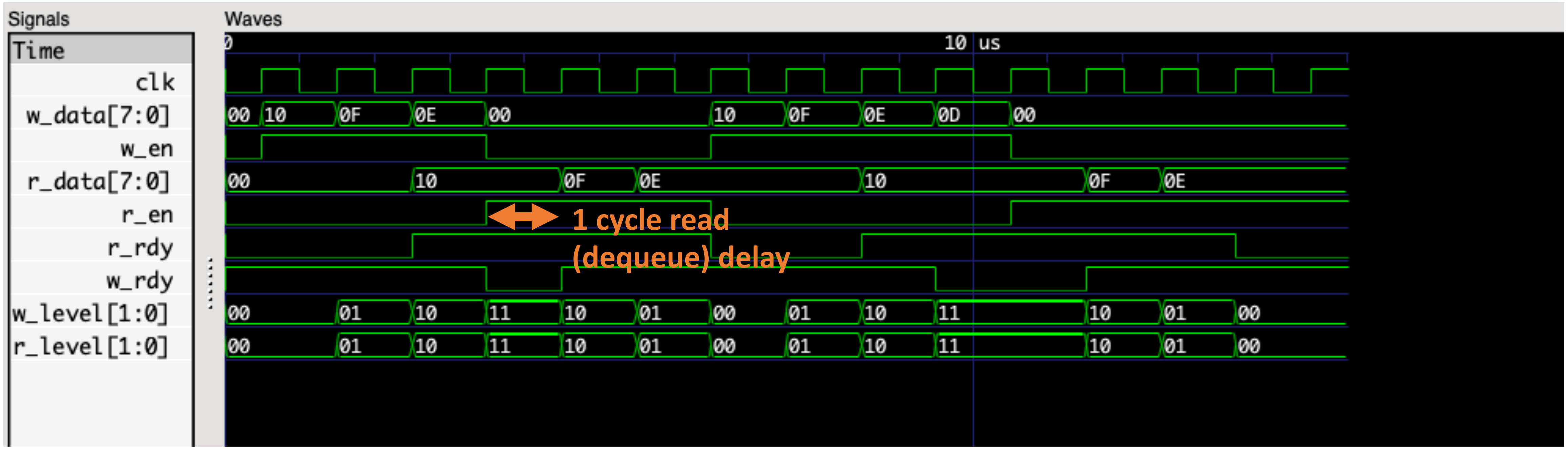
FIFO

- SyncFIFOBuffered (**depth=3**)



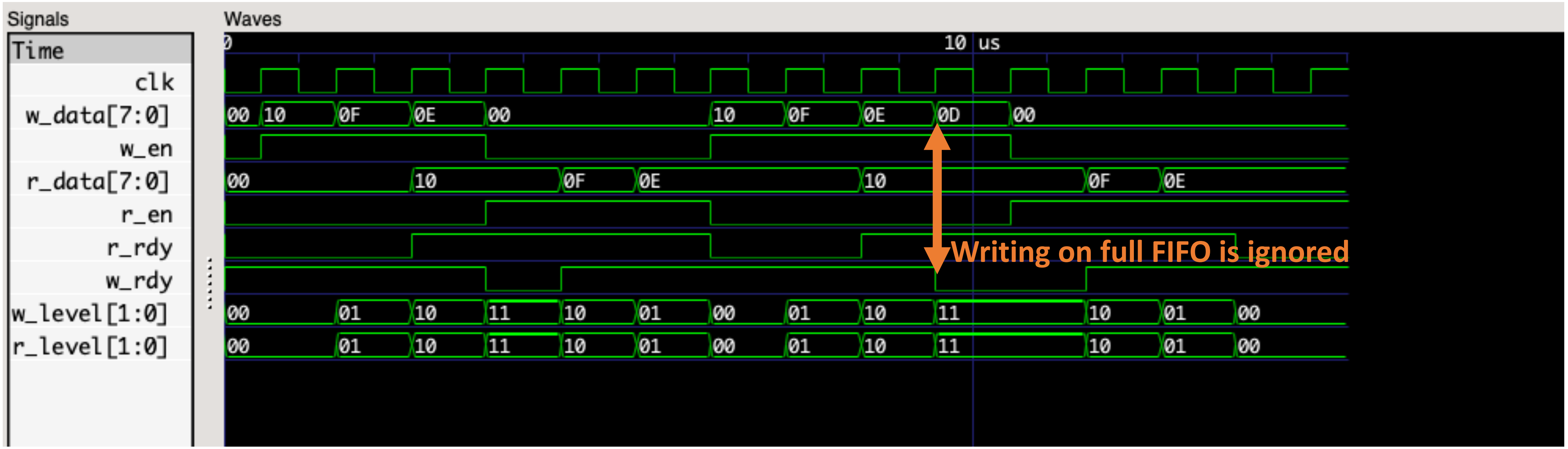
FIFO

- SyncFIFOBuffered (**depth=3**)



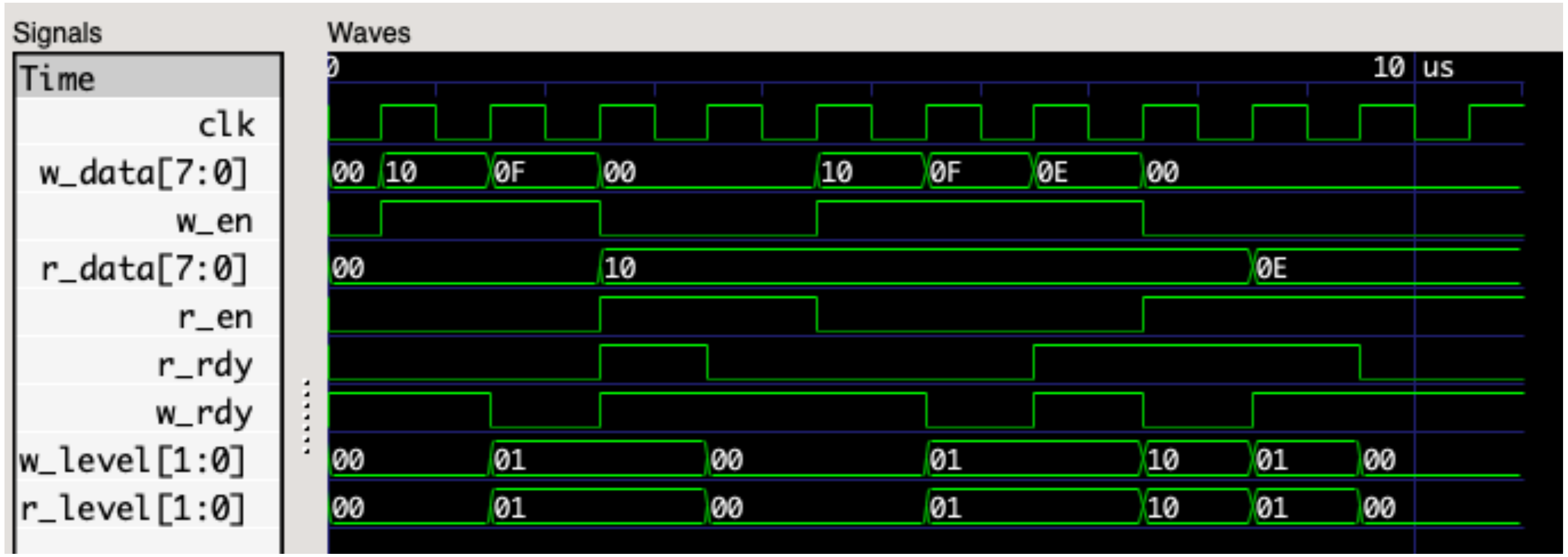
FIFO

- SyncFIFOBuffered (**depth=3**)



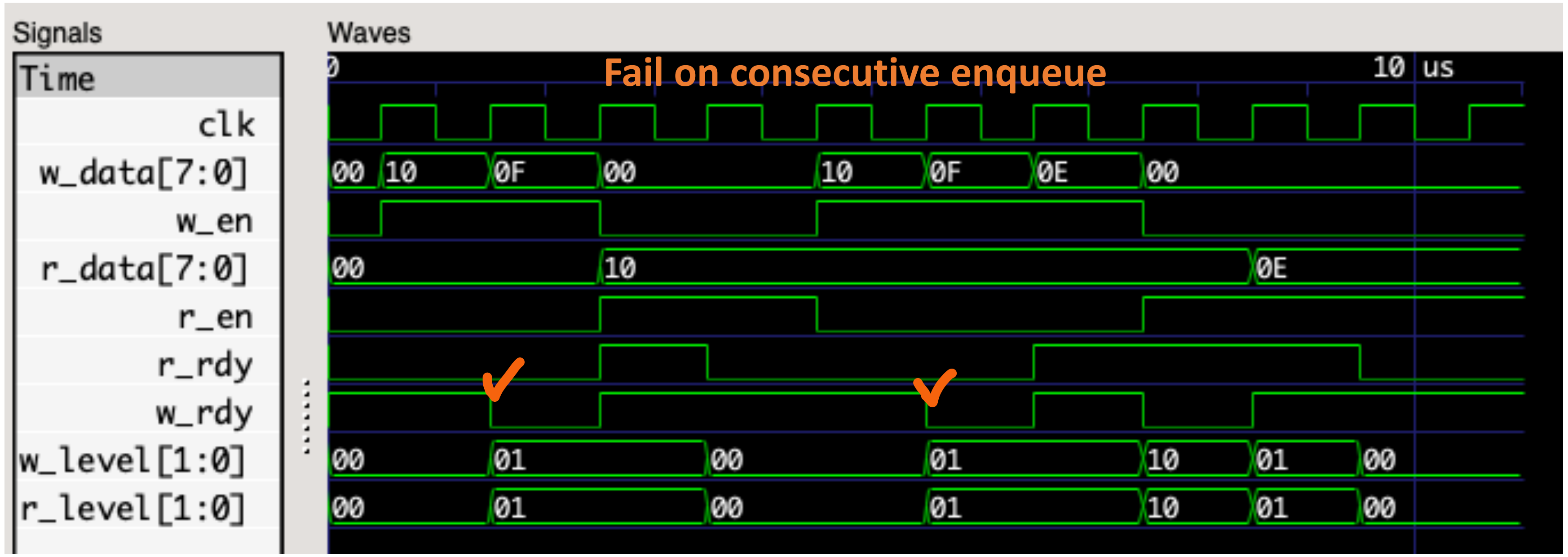
FIFO

- SyncFIFOBuffered (**depth=2**)



FIFO

- SyncFIFOBuffered (**depth=2**)

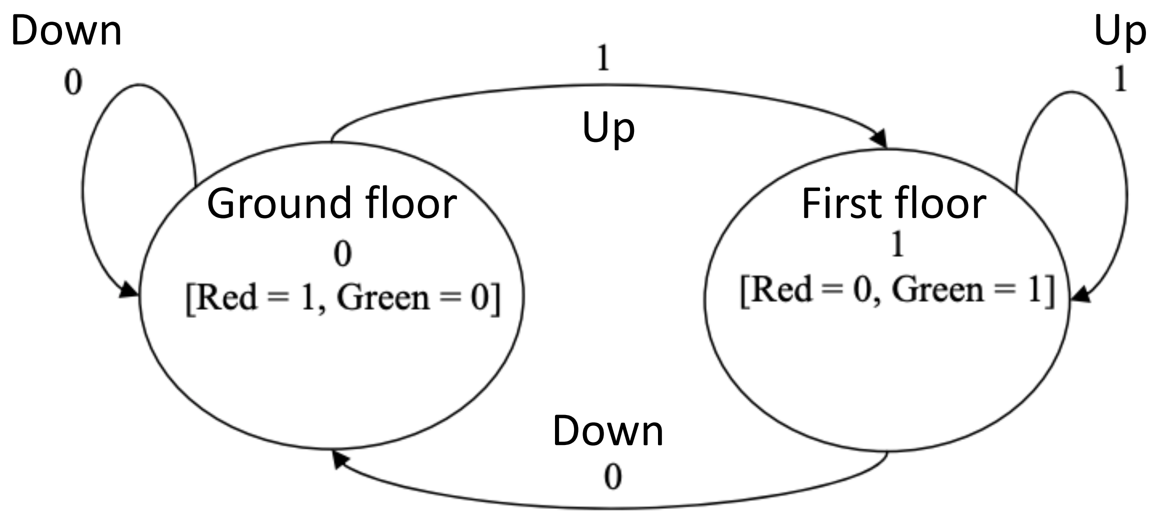


Q&A on FIFO

FSM

- Finite State Machine

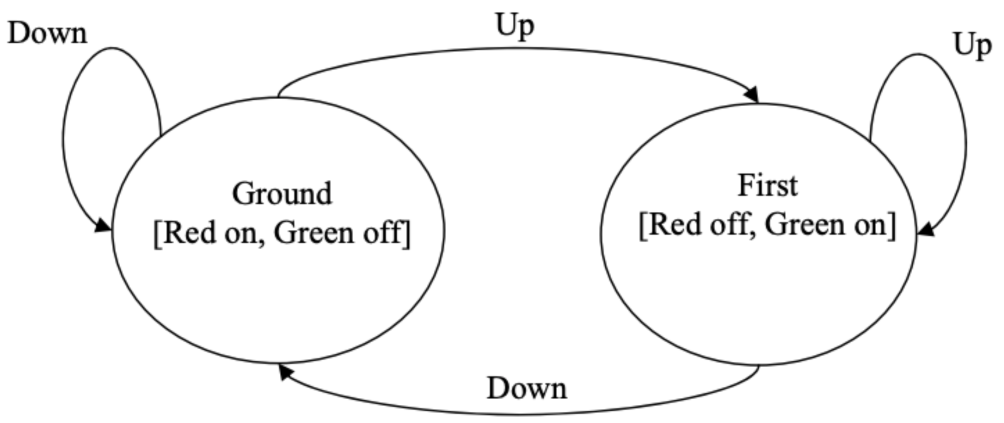
- Consider 2-floor elevator
  - Red light if ground floor
  - Green light if first floor

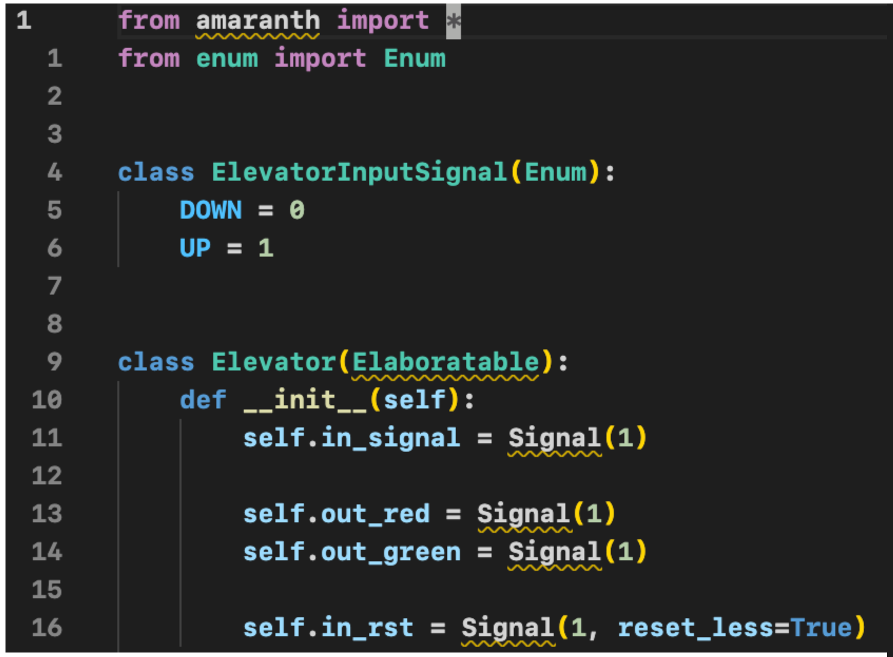


amaranth/dsl.py (github.com)
Example finite state machine (princeton.edu)

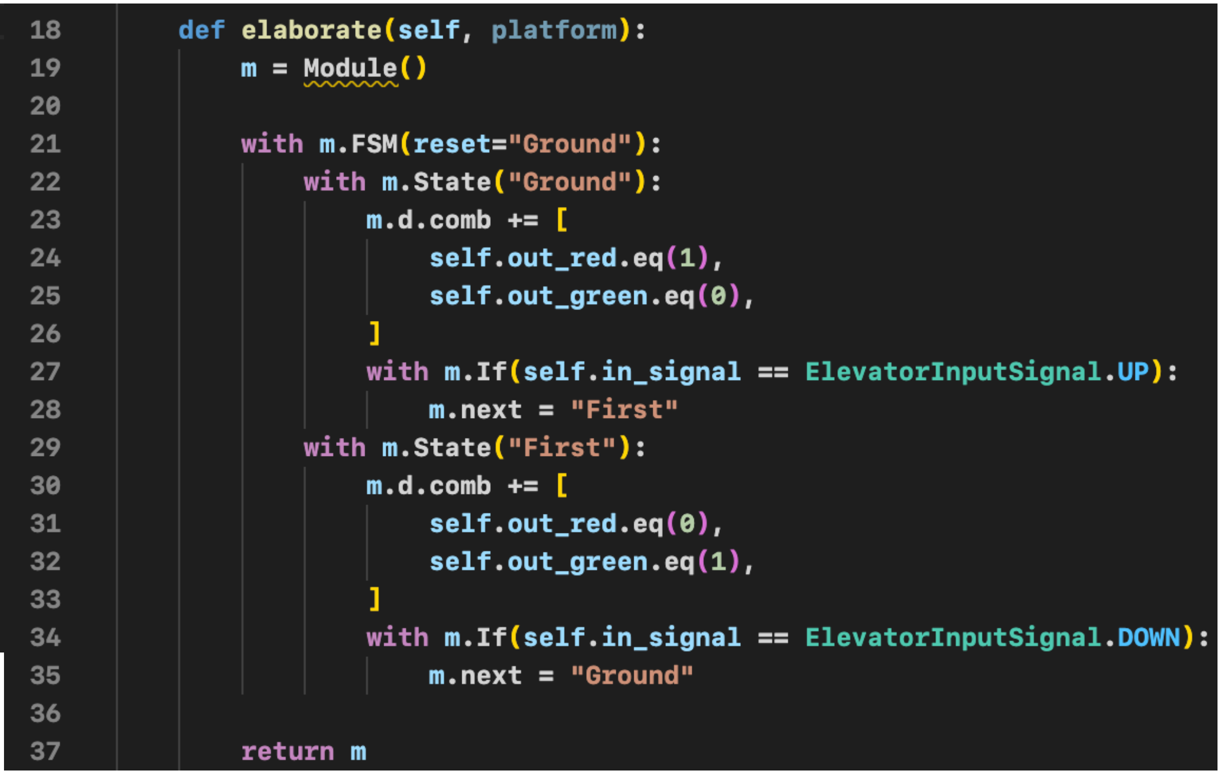
Example finite state machine (princeton.edu)

FSM





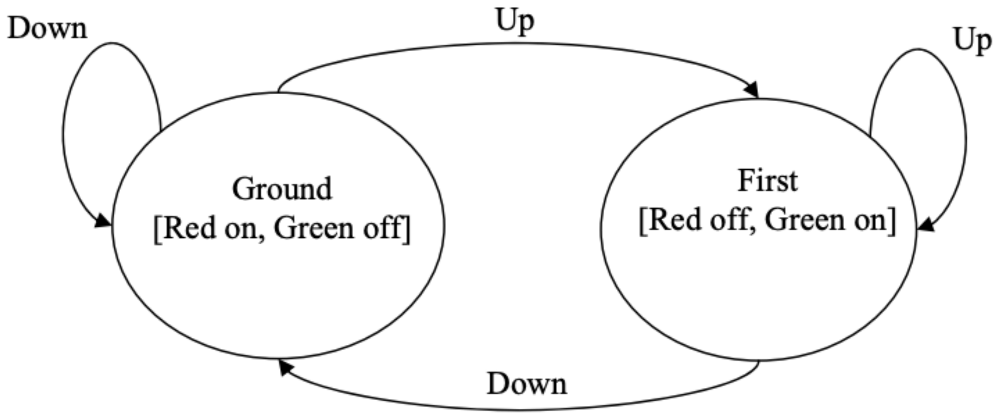
```python
from amaranth import *
from enum import Enum


class ElevatorInputSignal(Enum):
    DOWN = 0
    UP = 1


class Elevator(Elaboratable):
    def __init__(self):
        self.in_signal = Signal(1)

        self.out_red = Signal(1)
        self.out_green = Signal(1)

        self.in_rst = Signal(1, reset_less=True)
```

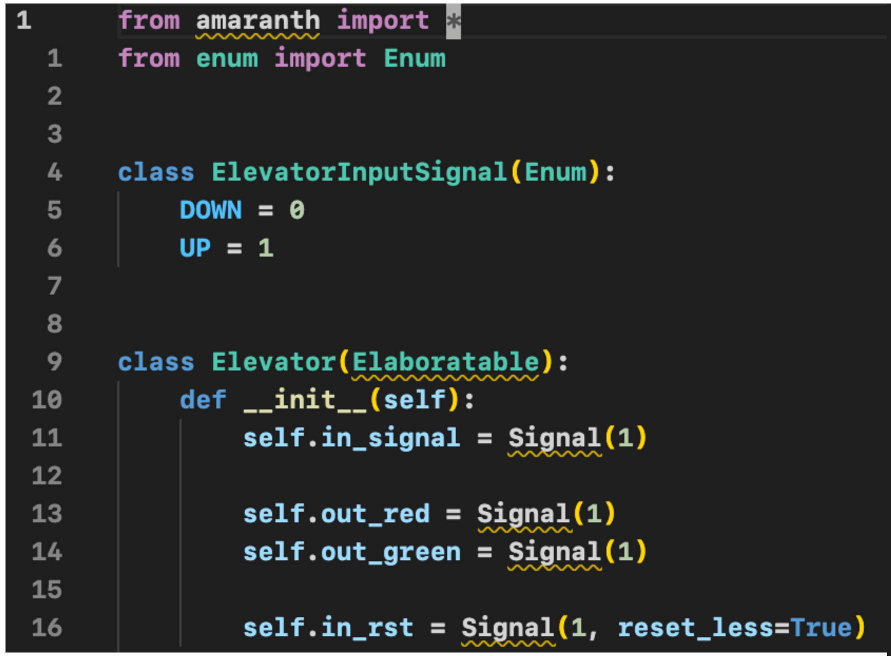


```python
    def elaborate(self, platform):
        m = Module()

        with m.FSM(reset="Ground"):
            with m.State("Ground"):
                m.d.comb += [
                    self.out_red.eq(1),
                    self.out_green.eq(0),
                ]
                with m.If(self.in_signal == ElevatorInputSignal.UP):
                    m.next = "First"
            with m.State("First"):
                m.d.comb += [
                    self.out_red.eq(0),
                    self.out_green.eq(1),
                ]
                with m.If(self.in_signal == ElevatorInputSignal.DOWN):
                    m.next = "Ground"

        return m
```

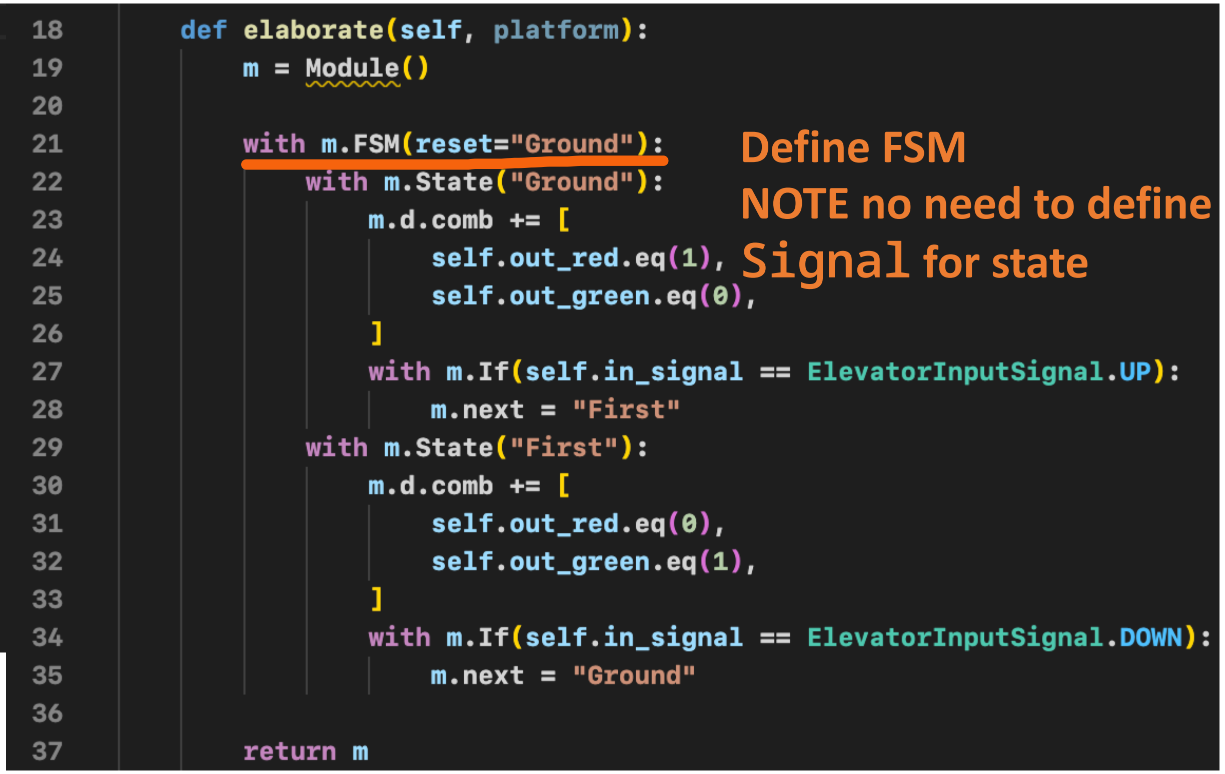
Example finite state machine (princeton.edu)

FSM





```python
from amaranth import *
from enum import Enum


class ElevatorInputSignal(Enum):
    DOWN = 0
    UP = 1


class Elevator(Elaboratable):
    def __init__(self):
        self.in_signal = Signal(1)

        self.out_red = Signal(1)
        self.out_green = Signal(1)

        self.in_rst = Signal(1, reset_less=True)
```



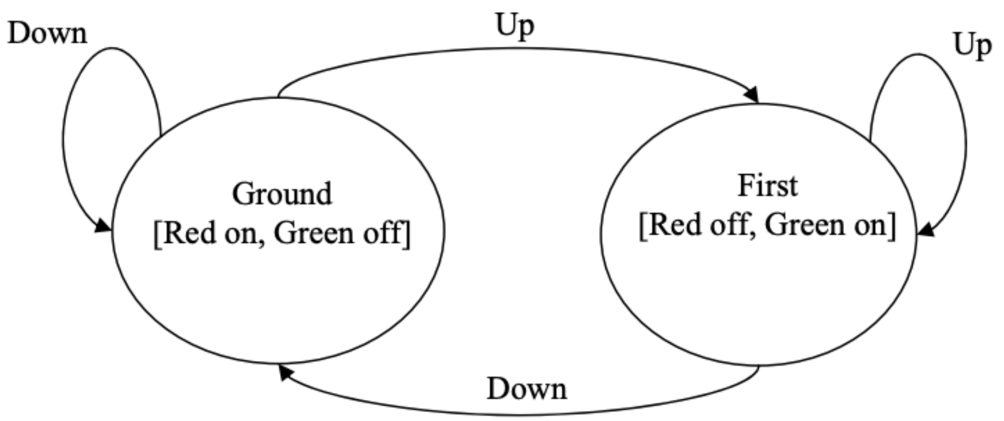
```python
    def elaborate(self, platform):
        m = Module()

        with m.FSM(reset="Ground"):
            with m.State("Ground"):
                m.d.comb += [
                    self.out_red.eq(1),
                    self.out_green.eq(0),
                ]
                with m.If(self.in_signal == ElevatorInputSignal.UP):
                    m.next = "First"
            with m.State("First"):
                m.d.comb += [
                    self.out_red.eq(0),
                    self.out_green.eq(1),
                ]
                with m.If(self.in_signal == ElevatorInputSignal.DOWN):
                    m.next = "Ground"

        return m
```

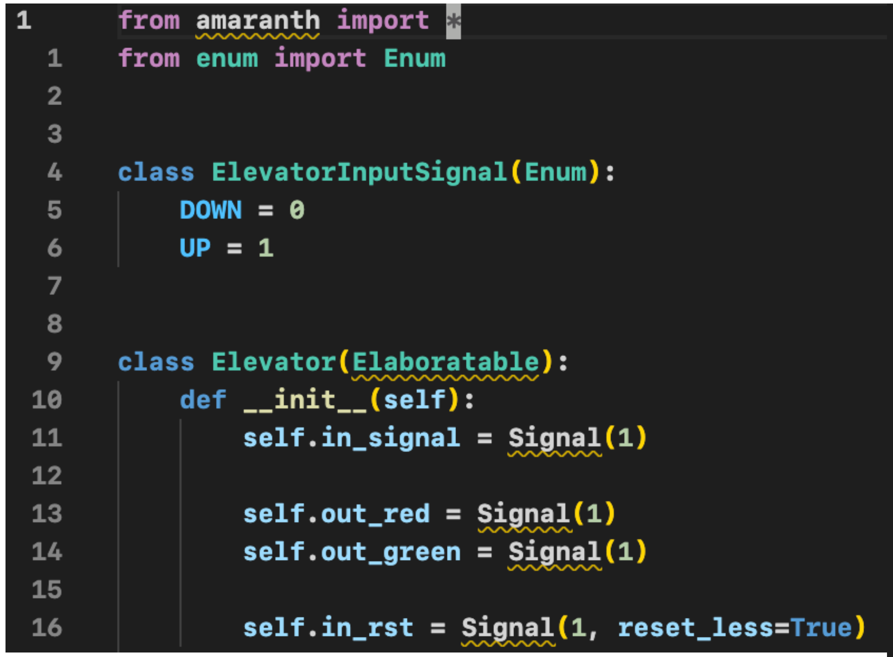
Define FSM

NOTE no need to define Signal for state

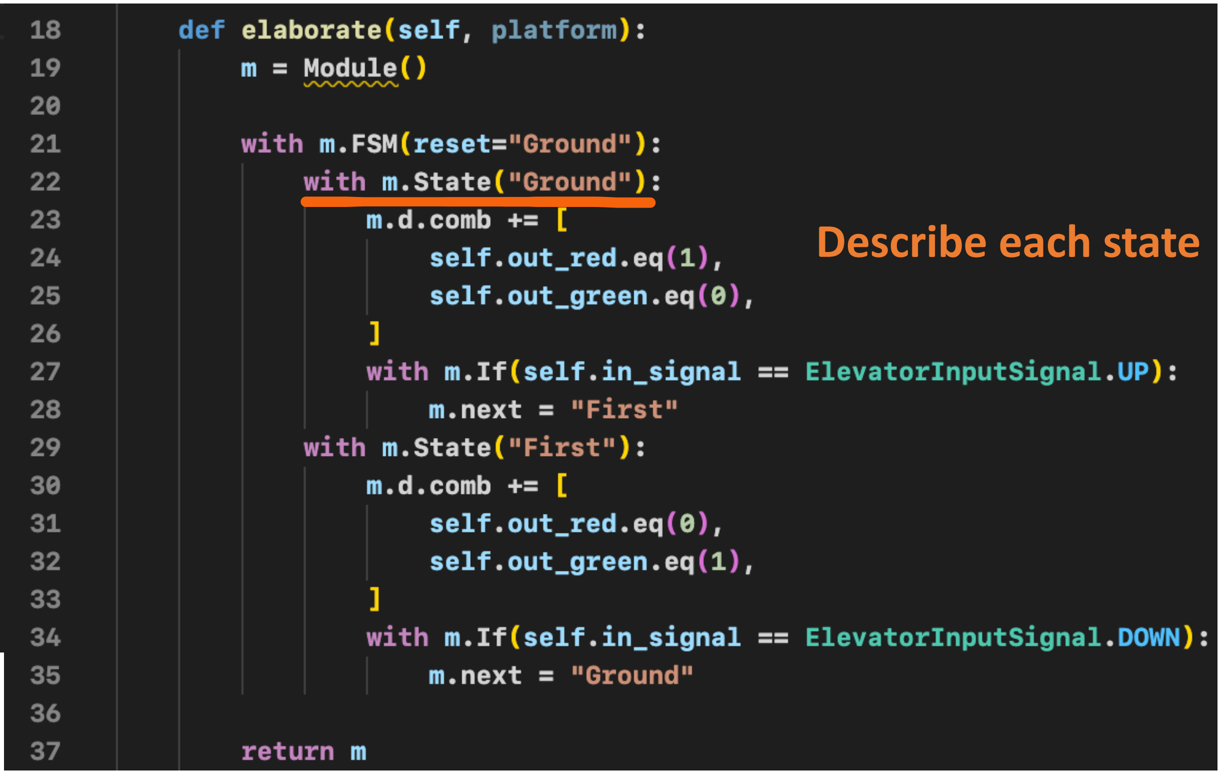
Example finite state machine (princeton.edu)

FSM





```python
from amaranth import *
from enum import Enum


class ElevatorInputSignal(Enum):
    DOWN = 0
    UP = 1


class Elevator(Elaboratable):
    def __init__(self):
        self.in_signal = Signal(1)

        self.out_red = Signal(1)
        self.out_green = Signal(1)

        self.in_rst = Signal(1, reset_less=True)
```

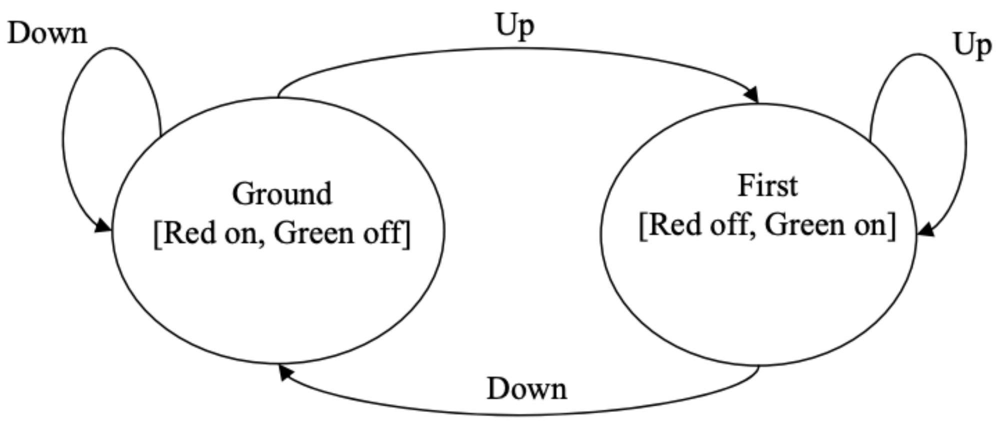


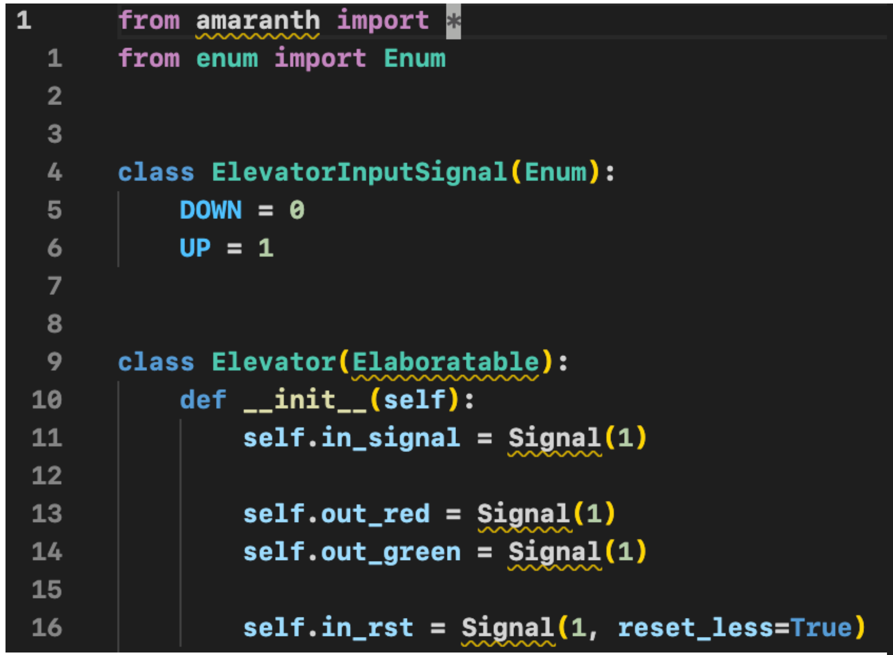
```python
    def elaborate(self, platform):
        m = Module()

        with m.FSM(reset="Ground"):
            with m.State("Ground"):
                m.d.comb += [
                    self.out_red.eq(1),
                    self.out_green.eq(0),
                ]
                with m.If(self.in_signal == ElevatorInputSignal.UP):
                    m.next = "First"
            with m.State("First"):
                m.d.comb += [
                    self.out_red.eq(0),
                    self.out_green.eq(1),
                ]
                with m.If(self.in_signal == ElevatorInputSignal.DOWN):
                    m.next = "Ground"

        return m
```

Describe each state

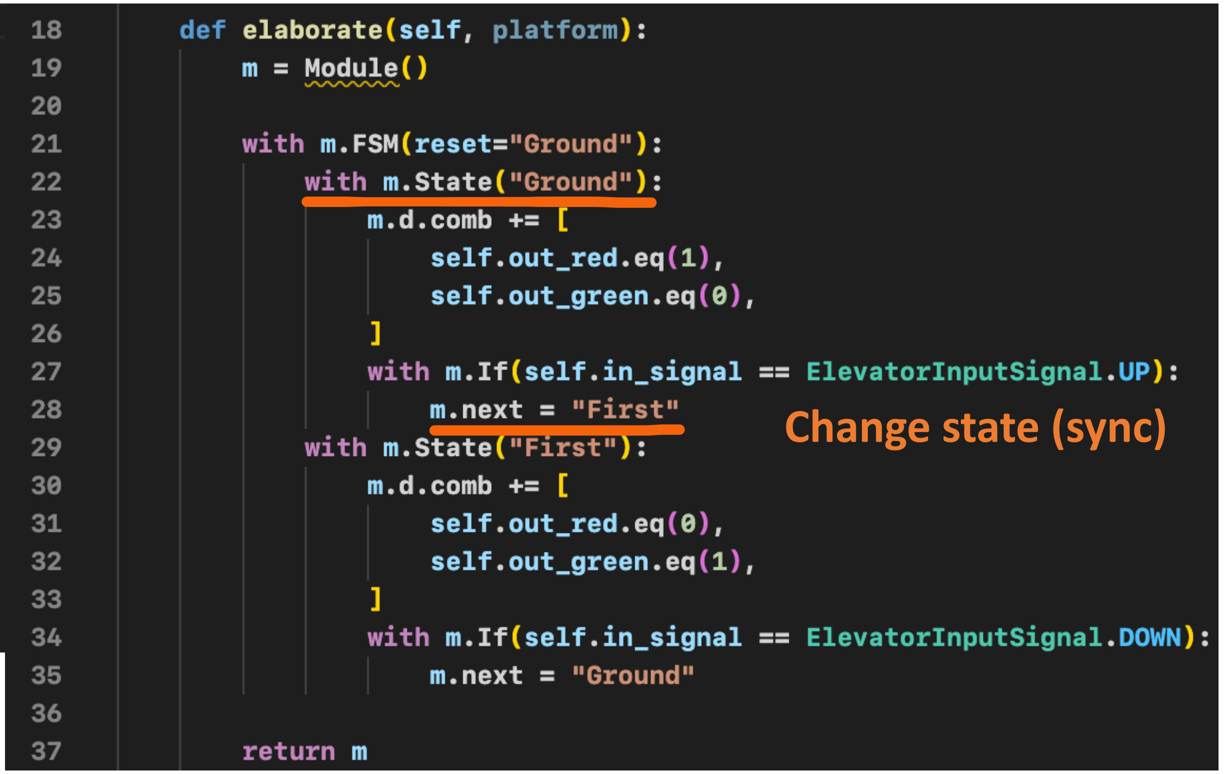
Example finite state machine (princeton.edu)

FSM





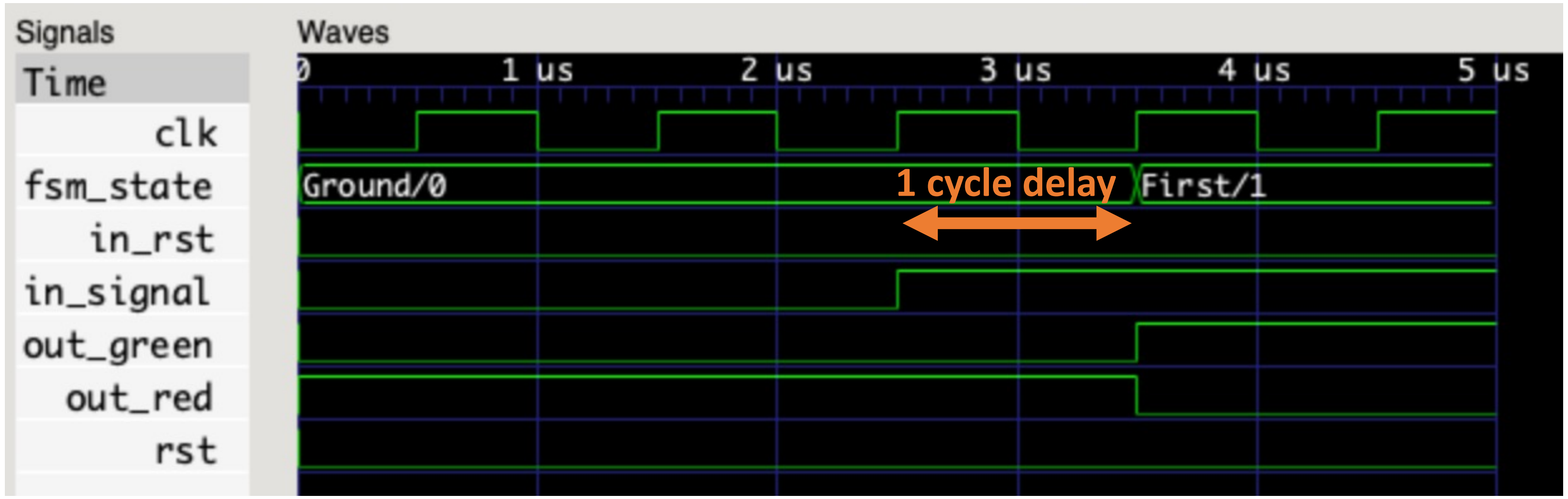
```python
from amaranth import *
from enum import Enum


class ElevatorInputSignal(Enum):
    DOWN = 0
    UP = 1


class Elevator(Elaboratable):
    def __init__(self):
        self.in_signal = Signal(1)

        self.out_red = Signal(1)
        self.out_green = Signal(1)

        self.in_rst = Signal(1, reset_less=True)
```



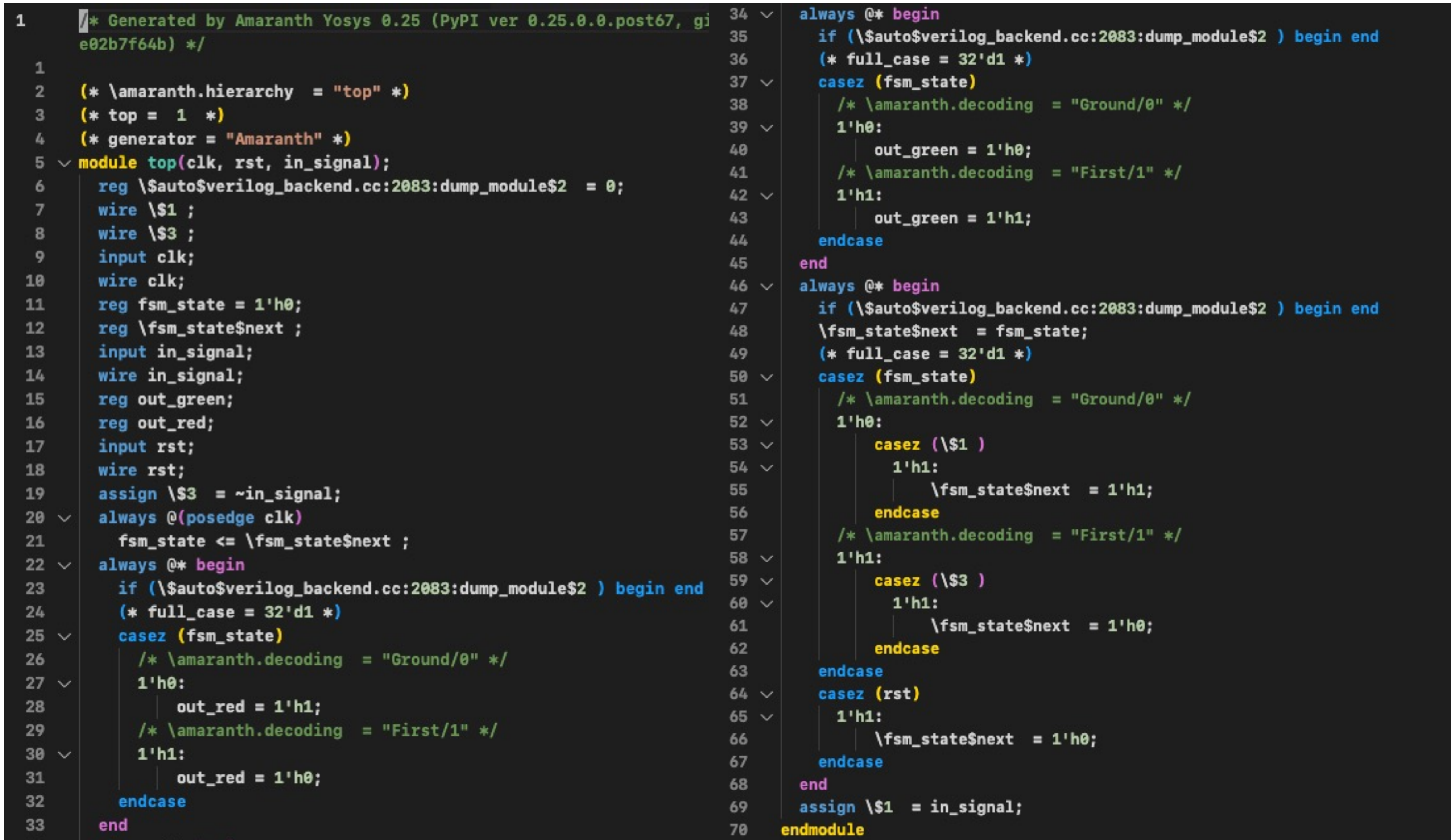
```python
    def elaborate(self, platform):
        m = Module()

        with m.FSM(reset="Ground"):
            with m.State("Ground"):
                m.d.comb += [
                    self.out_red.eq(1),
                    self.out_green.eq(0),
                ]
                with m.If(self.in_signal == ElevatorInputSignal.UP):
                    m.next = "First"
            with m.State("First"):
                m.d.comb += [
                    self.out_red.eq(0),
                    self.out_green.eq(1),
                ]
                with m.If(self.in_signal == ElevatorInputSignal.DOWN):
                    m.next = "Ground"

        return m
```

Change state (sync)

FSM

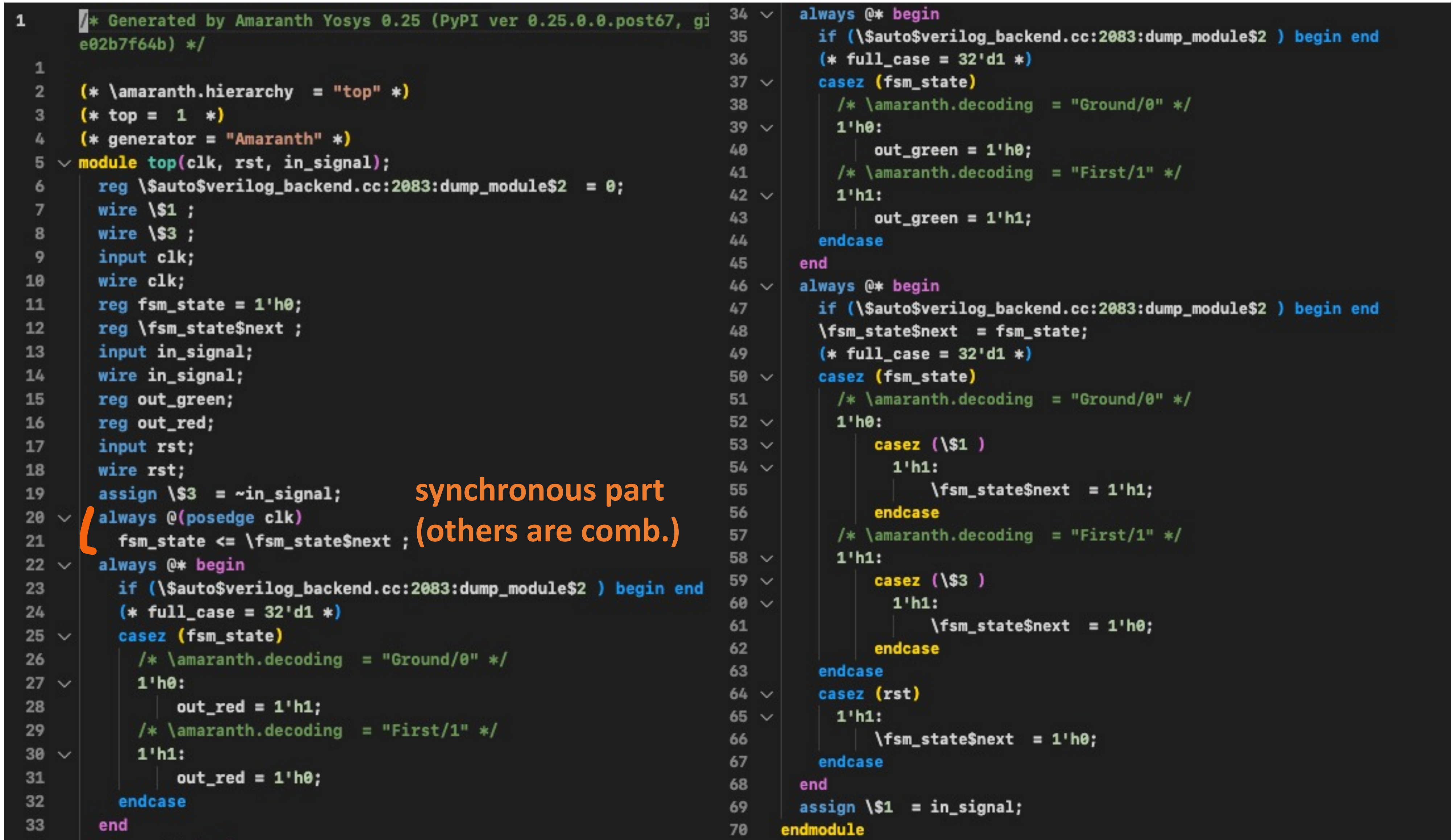

Signals
Waves
Time
clk
fsm_state
in_rst
in_signal
out_green
out_red
rst
0
1 us
2 us
3 us
4 us
5 us
Ground/0
First/1
1 cycle delay

FSM



```
/* Generated by Amaranth Yosys 0.25 (PyPI ver 0.25.0.0.post67, gi
e02b7f64b) */

(* \amaranth.hierarchy  = "top" *)
(* top =  1  *)
(* generator = "Amaranth" *)
module top(clk, rst, in_signal);
  reg \$auto$verilog_backend.cc:2083:dump_module$2  = 0;
  wire \$1 ;
  wire \$3 ;
  input clk;
  wire clk;
  reg fsm_state = 1'h0;
  reg \fsm_state$next ;
  input in_signal;
  wire in_signal;
  reg out_green;
  reg out_red;
  input rst;
  wire rst;
  assign \$3  = ~in_signal;
  always @(posedge clk)
    fsm_state <= \fsm_state$next ;
  always @* begin
    if (\$auto$verilog_backend.cc:2083:dump_module$2 ) begin end
    (* full_case = 32'd1 *)
    casez (fsm_state)
      /* \amaranth.decoding  = "Ground/0" */
      1'h0:
          out_red = 1'h1;
      /* \amaranth.decoding  = "First/1" */
      1'h1:
          out_red = 1'h0;
    endcase
  end
  always @* begin
    if (\$auto$verilog_backend.cc:2083:dump_module$2 ) begin end
    (* full_case = 32'd1 *)
    casez (fsm_state)
      /* \amaranth.decoding  = "Ground/0" */
      1'h0:
          out_green = 1'h0;
      /* \amaranth.decoding  = "First/1" */
      1'h1:
          out_green = 1'h1;
    endcase
  end
  always @* begin
    if (\$auto$verilog_backend.cc:2083:dump_module$2 ) begin end
    \fsm_state$next  = fsm_state;
    (* full_case = 32'd1 *)
    casez (fsm_state)
      /* \amaranth.decoding  = "Ground/0" */
      1'h0:
          casez (\$1 )
            1'h1:
                \fsm_state$next  = 1'h1;
          endcase
      /* \amaranth.decoding  = "First/1" */
      1'h1:
          casez (\$3 )
            1'h1:
                \fsm_state$next  = 1'h0;
          endcase
    endcase
    casez (rst)
      1'h1:
          \fsm_state$next  = 1'h0;
    endcase
  end
  assign \$1  = in_signal;
endmodule
```

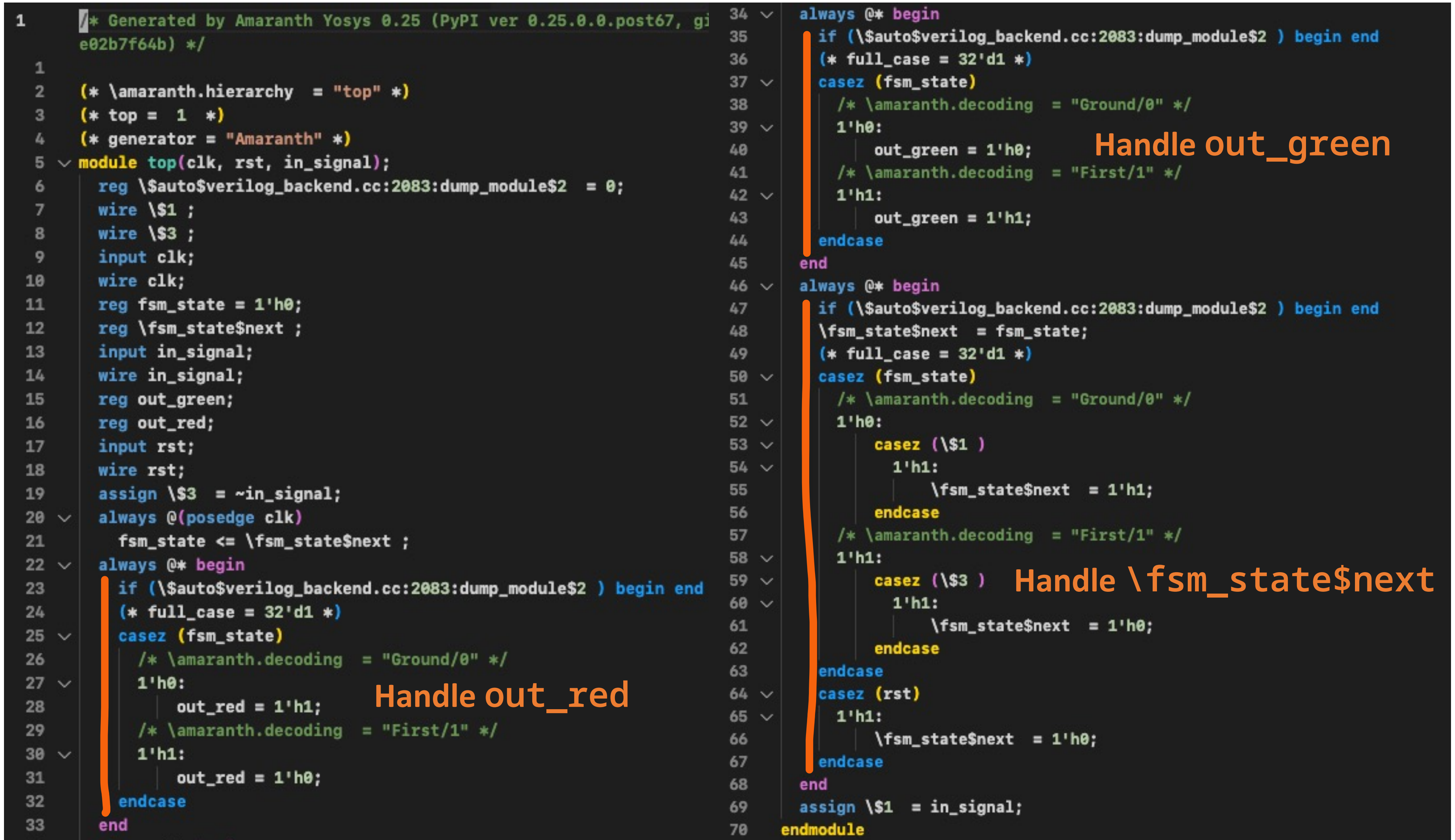
FSM



```
/* Generated by Amaranth Yosys 0.25 (PyPI ver 0.25.0.0.post67, gi
e02b7f64b) */

(* \amaranth.hierarchy  = "top" *)
(* top =  1  *)
(* generator = "Amaranth" *)
module top(clk, rst, in_signal);
  reg \$auto$verilog_backend.cc:2083:dump_module$2  = 0;
  wire \$1 ;
  wire \$3 ;
  input clk;
  wire clk;
  reg fsm_state = 1'h0;
  reg \fsm_state$next ;
  input in_signal;
  wire in_signal;
  reg out_green;
  reg out_red;
  input rst;
  wire rst;
  assign \$3  = ~in_signal;
  always @(posedge clk)
    fsm_state <= \fsm_state$next ;
  always @* begin
    if (\$auto$verilog_backend.cc:2083:dump_module$2 ) begin end
    (* full_case = 32'd1 *)
    casez (fsm_state)
      /* \amaranth.decoding  = "Ground/0" */
      1'h0:
          out_red = 1'h1;
      /* \amaranth.decoding  = "First/1" */
      1'h1:
          out_red = 1'h0;
    endcase
  end
  always @* begin
    if (\$auto$verilog_backend.cc:2083:dump_module$2 ) begin end
    (* full_case = 32'd1 *)
    casez (fsm_state)
      /* \amaranth.decoding  = "Ground/0" */
      1'h0:
          out_green = 1'h0;
      /* \amaranth.decoding  = "First/1" */
      1'h1:
          out_green = 1'h1;
    endcase
  end
  always @* begin
    if (\$auto$verilog_backend.cc:2083:dump_module$2 ) begin end
    \fsm_state$next  = fsm_state;
    (* full_case = 32'd1 *)
    casez (fsm_state)
      /* \amaranth.decoding  = "Ground/0" */
      1'h0:
          casez (\$1 )
            1'h1:
                \fsm_state$next  = 1'h1;
          endcase
      /* \amaranth.decoding  = "First/1" */
      1'h1:
          casez (\$3 )
            1'h1:
                \fsm_state$next  = 1'h0;
          endcase
    endcase
    casez (rst)
      1'h1:
          \fsm_state$next  = 1'h0;
    endcase
  end
  assign \$1  = in_signal;
endmodule
```

synchronous part
(others are comb.)

FSM



```verilog
/* Generated by Amaranth Yosys 0.25 (PyPI ver 0.25.0.0.post67, gi
e02b7f64b) */

(* \amaranth.hierarchy  = "top" *)
(* top =  1  *)
(* generator = "Amaranth" *)
module top(clk, rst, in_signal);
  reg \$auto$verilog_backend.cc:2083:dump_module$2  = 0;
  wire \$1 ;
  wire \$3 ;
  input clk;
  wire clk;
  reg fsm_state = 1'h0;
  reg \fsm_state$next ;
  input in_signal;
  wire in_signal;
  reg out_green;
  reg out_red;
  input rst;
  wire rst;
  assign \$3  = ~in_signal;
  always @(posedge clk)
    fsm_state <= \fsm_state$next ;
  always @* begin
    if (\$auto$verilog_backend.cc:2083:dump_module$2 ) begin end
    (* full_case = 32'd1 *)
    casez (fsm_state)
      /* \amaranth.decoding  = "Ground/0" */
      1'h0:
          out_red = 1'h1;
      /* \amaranth.decoding  = "First/1" */
      1'h1:
          out_red = 1'h0;
    endcase
  end
always @* begin
  if (\$auto$verilog_backend.cc:2083:dump_module$2 ) begin end
  (* full_case = 32'd1 *)
  casez (fsm_state)
    /* \amaranth.decoding  = "Ground/0" */
    1'h0:
        out_green = 1'h0;
    /* \amaranth.decoding  = "First/1" */
    1'h1:
        out_green = 1'h1;
  endcase
end
always @* begin
  if (\$auto$verilog_backend.cc:2083:dump_module$2 ) begin end
  \fsm_state$next  = fsm_state;
  (* full_case = 32'd1 *)
  casez (fsm_state)
    /* \amaranth.decoding  = "Ground/0" */
    1'h0:
        casez (\$1 )
          1'h1:
              \fsm_state$next  = 1'h1;
        endcase
    /* \amaranth.decoding  = "First/1" */
    1'h1:
        casez (\$3 )
          1'h1:
              \fsm_state$next  = 1'h0;
        endcase
  endcase
  casez (rst)
    1'h1:
        \fsm_state$next  = 1'h0;
  endcase
end
assign \$1  = in_signal;
endmodule
```

Q&A on FSM