structure.py

```
a = Signal()
b = Signal(2)
c = Signal(max=23)
d = Signal(reset=1)
e = Signal(4, reset_less=True)
f = Signal.like(c)
g = f.nbits
```

ClockSignal("sys")	→ Returns the clock signal of a given clock domain
ResetSignal("sys")	→ Returns the reset signal of a given clock domain
Mux(sel, val1, val0)	→ Multiplex between two values. Output val1 is sel asserted else val0.
Cat(a, b, ...)	→ Form a compound value from several smaller ones by concatenation. The first argument occupies the lower bits of the result.

If(a,
  b.eq(1)
).Elif(c,
  b.eq(0)
).Else(
  b.eq(d)
)
	→ Conditional execution of statements

Case(a, { 0 : b.eq(1), 3 : c.eq(0), 5 : b.eq(d), "default": b.eq(0), })	→ Dictionary of cases. Selector value used to decide which block to execute.
Array([a,b,c,...])	→ Represents lists of other objects that can be indexed by FHDL expressions
ClockDomain()	→ Creation of a synchronous domain
Constant()	→ Represents a constant, HDL literal integer. Thus, it supports slicing and can have a bit width different from what isimplied by the value it represents

misc.py

aa = Signal(8) bb = Signal(3) cc = Signal(64) **displacer(aa, bb, cc)** aa.eq(0xAA) bb.eq(2) → cc is 0x0000000000AA0000	→ Makes cc equal to aa << bb. bb is expressed as a multiple of aa size. Value aa is displaced at position bb in cc.
aa = Signal(64) bb = Signal(3) cc = Signal(8) **Chooser(aa, bb, cc)** aa.eq(0xAABBCCDD11223344) bb.eq(2) → cc is 0x33	→ Makes cc equal the value of it's size choosen in aa at index bb.

timeline(start, [
  (t0, [assignment1,assignment2,...]),
  (t1, [assignment3,assignment4,...]),
  …
])
	→ Execute statement in a timely manner. Sequencing starts when start condition is true. Then at each point in time (t0, t1,…) expressed as clock count, correspondings assignments are made.

WaitTimer(Module)	→ Take a number (n) of clock period as parameter. When signal wait is asserted, signal done become active after (n) period of clock.

bitcontainer.py

bits_for(n)	→ Returns how many bits are need to hold the value n
log2_int(n)	→ Returns the power to which the number 2 must be raised to obtain the value n
value_bits_sign(s)	→ Returns a tupple (nb_bits, sign) of given signal

decorator.py

@CEInserter CEInserter()(module)	→ Add Clock Enable signal to a module
@ResetInserter ResetInserter()(module)	→ Add Reset signal to a module (independant from clock domain reset)
ClockDomainsRenamer()(module)	→ Change clock domain of a module

resetsync.py

AsyncResetSynchronizer(Special)	→ Connects a synchronized reset signal to the provided ClockDomain

cdc.py

MultiReg(Special)	→ Signal synchronization
PulseSynchronizer(Module)	→ Signal synchronization of a pulse in a fast clock domain to a relatively slow clock domain
GrayCounter(Module)	→ The Gray code outputs differ in only one bit for every two successive values
ElasticBuffer(Module)	→ Implementation of an elastic buffer
Gearbox(Module)	→ Implementation of a Gearbox

coding.py

Encoder(Module)	→ Encode one-hot to binary
PriorityEncoder(Module)	→ Encode one-hot with priotity to binary
Decoder(Module)	→ Binary to one-hot
PriorityDecoder(Module)	→ Binary to one-hot

io.py

DifferentialInput(Special)	→ Instanciate a vendor specific differential input
DifferentialOutput(Special)	→ Instanciate a vendor specific differential output
DDRInput(Special)	→ Instanciate a vendor specific double data rate input
DDROutput(Special)	→ Instanciate a vendor specific double data rate output
CRG(Module)	→ Simple Clock and Reset Generator. Creates sys clock domain, assign given clk signal. Generates a Power On Reset

fsm.py

fsm = FSM(reset_state="START") self.submodules += fsm	→ Create a FSM (reset_state is optional)

fsm.act("MYSTATE",
  # assignment1,
  # assignment2,
)
	→ Add a state with statements

NextState("MYSTATE1")	→ Assign next state

fsm.act("MYSTATE",
  NextValue(a, b),
  ...,
)
	→ Synchronous statement inside a given state, equivalent to self.sync += a.eq(b) when the FSM is in the given state.

fsm.act("MYSTATE",
  a.eq(b),
  ...,
)
	→ Combinatorial statements of form a.eq(b) are equivalent to self.comb += a.eq(b) when the FSM is in the given state. Outside this state, if not stated, a.eq(0)

fsm.ongoing("MYSTATE")	→ Returns a signal that has the value 1 when the FSM is in the given state, and 0 otherwise
fsm.before_entering("MYSTATE")	→ Returns a signal that has the value 1 during the clock cycle before entering the given state
fsm.before_leaving("MYSTATE")	→ Returns a signal that has the value 1 during the clock cycle before leaving the given state
fsm.after_entering("MYSTATE")	→ Returns a signal that has the value 1 during the clock cycle after entering the given state
fsm.after_leaving("MYSTATE")	→ Returns a signal that has the value 1 during the clock cycle after leaving the given state
fsm.delayed_enter("STATE1", "STATE2", delay)	

specials.py

TSTriple(size)	→ A triplet (O, OE, I) defining a tri-state I/O port. Such objects are only containers for signals that are intended to be later connected to a tri-state I/O buffer, and cannot be used as module specials
Tristate(target, o, oe, i)	→ Instance of a tri-state I/O buffer. Signals target, o and i can have any width, while oe is 1-bit wide. The target signal should go to a port and not be used elsewhere in the design.

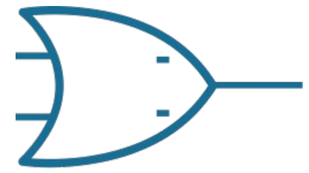
din = Signal(32)
dout = Signal(32)
dinout = Signal(32)
 	
self.specials += Instance("custom_core",
  p_DATA_WIDTH = 32,
  i_din = din,
  o_dout = dout,
  io_dinout = dinout
)
	→ Add an external Verilog or VHDL module to the design.
	
	The first parameters of the Instance is the Module's name followed by the parameters and ports of the Module.
	
	Prefixes are used to specify the type of interface:
	
	p_ for a Parameter
	i_ for an Input port
	o_ for an Output port
	io_ for a Bi-Directional port

Memory(width, depth, init) ---- mem = Memory(32, 128) port = mem.get_port() self.specials += mem, port	→ Instance of an on-chip SRAM. The width is the number of bits in each word, the depth is the number of words in the memory and an optional list of integers used to initialize the memory. To access the memory in hardware, ports can be obtained by calling the get_port method. A port always has an address signal a and a data read signal dat_r. Other signals may be available depending on the port's configuration.

fifo.py

SyncFIFO(Module)	→ Implementation of a synchronous FIFO. Read and write interfaces are accessed from the same clock domain. If different clock domains are needed, use AsyncFIFO
SyncFIFOBuffered(Module)	→ Registered FIFO output. Improves timing when it breaks due to sluggish clock-to-output delay. Increases latency by one cycle.
AsyncFIFO(Module)	→ Read and write interfaces are accessed from different clock domains, named `read` and `write`. Use ClockDomainsRenamer to rename to other names.
AsyncFIFOBuffered(Module)	→ Registered FIFO output. Improves timing when it breaks due to sluggish clock-to-output delay. Increases latency by one cycle.

Migen Cheat Sheet



https://m-labs.hk/gateware/migen/

record.py

Record(name, size)
Record(name, size, direction)
Record(name, sublayout)
 	
Example :
 	
layout = [
  ("a", 32, DIR_M_TO_S),
  ("b", 1),
]
 	
r = Record(layout)
self.comb += r.a.eq(25)
	→ Create a Record. Size is an int, sublayout must be a list. A layout is an equivalent of a C structure. It's a collection of signals.

layout_len(layout)	→ Gives the len (in bits) of the layout
layout_get(layout, name)	→ Get the layout of a named entry in a layout
r = Record(layout) r.flatten()	→ Returns a list of all signals contained in the record
r = Record(layout) r.raw_bits()	→ Returns a signal of len equals to layout_len. This signal is a flatten representation of all Record's bits.

layout = [
  ("a", 32, DIR_M_TO_S),
  ("b", 1, DIR_S_TO_M),
  ("c", 1, DIR_S_TO_M),
]
 	
m = Record(layout)
s = Record(layout)
m.connect(s, omit={"c"})
	→ Connect signals of a given records. Direction depend of direction defined in the layout.
	Here, this is equivalent to m.b.eq(s.b) and s.a.eq(m.a).
	Signal can be omitted during connection (omit) or all omitted except kept signals (keep).