

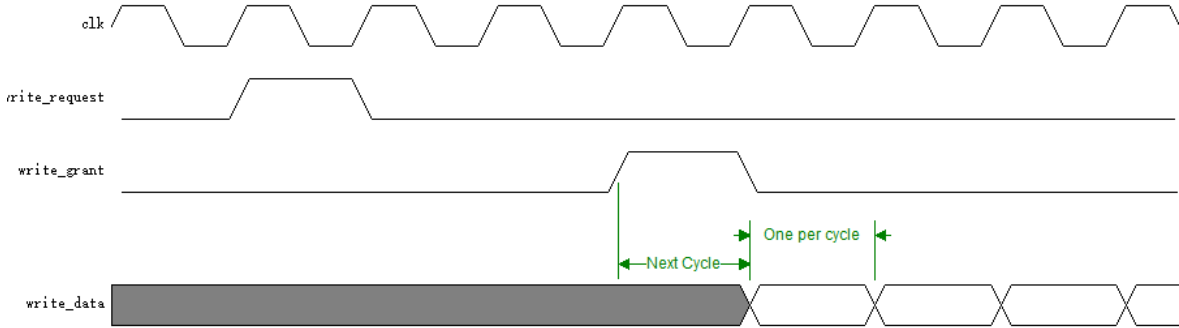
Common Instruction-Level Abstraction Issues

Draft Working Document: February 18, 2017

Timing

Instruction-Level Abstraction can be used to abstract away timing information if they are not intended by the specification. However if timing requirements are part of specification, ILA can also be used to capture the timing on the interface at a scale of clock cycles.

Figure 1: Timing diagram of burst write in an example bus



Let's take a simple bus interface as an example. An accelerator can initiate a burst write of an arbitrary length, it first sends a request to the bus arbiter and waits for the grant. The grant can come any time after the request. But right after the grant the accelerator should send the data on the data port one per cycle. So there is a timing requirement between grant and data write. The timing diagram is shown in Figure 1. An example template of the write interface is shown below. The key idea in modeling timing characteristics is to use a counter to count the cycle. And when writing the refinement relations, define the matching between the behavior of the implementation in each cycle with each step in the sub-instructions in ILA.

```

1 Accelerator = ila.Abstraction('ExampleILA')           # Define the Abstraction
2                                                         # Define the Interface
3 write_grant = Accelerator.inp('write_grant', 1)        # - request grant
4 write_data  = Accelerator.reg('write_data', 32)        # - data port
5 write_length = Accelerator.reg('write_len', 16)       # - burst length
6 ...                                                # - possibly more ports
7 writing      = Accelerator.reg('writing', 1)           # 1 bit flag indicating if
8                                                         # it is writing to the bus
9 Accelerator.decode_exps += [ write_grant == 1 ]
10 # The grant operation is considered as an instruction to the accelerator
11
12 writing_nxt = ila.ite( write_grant == 1,              # The effect of the instruction above
13                       b0,                             # This is the complete function
14                       writing )                        # without holes. However, you can
15                                                         # use synthesis to create this
16                                                         # function
17 #Note: b0 = ila.const(0,1) (1-bit-wide constant 0)
18
19 Accelerator.set_next('writing', writing_nxt )
20
21 write_fsm   = Accelerator.add_microabstraction('WriteFsm', writing == 1 )
22 # Bus Write logic.
```

```

23 counter      = write_fsm.reg('counter', 32) # The counter here is used
24                                                    # to count cycles
25 write_fsm.set_init('counter', h0_16 )         # b0_16 = ila.const(0,16)
26 write_fsm.set_next('counter', counter + 1)
27 write_fsm.set_next('writing',
28     ila.ite(counter == write_length-1, b0, writing) ) # turn the flag off,
29                                                         # when necessary
30 write_fsm.set_next('write_data', ?? )           # write some values
31                                                         # each cycle

```

Instruction Ordering

Interrupt

Specification and Micro-architecture