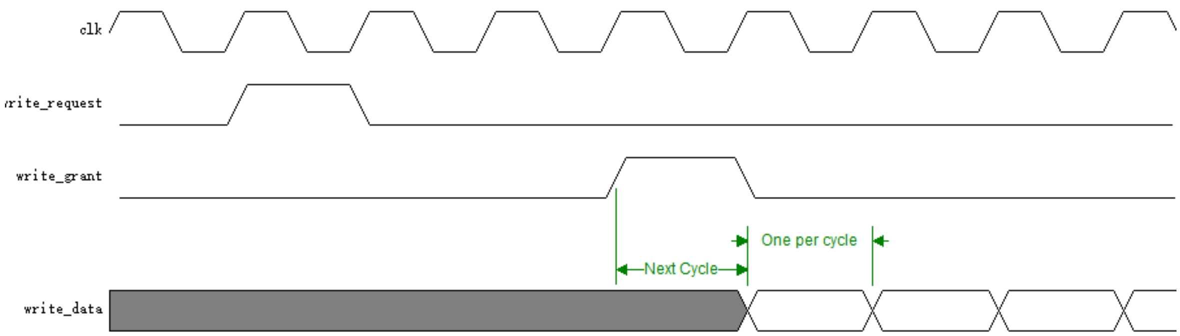
Common Instruction-Level Abstraction Issues

Draft Working Document: February 19, 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 sends the data on the data port one per cycle. So there is 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 charateristic 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.

```
Accelerator = ila.Abstraction('ExampleILA')          # Define the Abstraction
                                                     # Define the Interface
write_grant = Accelerator.inp('write_grant', 1)      #   - request grant
write_data  = Accelerator.reg('write_data' , 32)     #   - data port
write_length= Accelerator.reg('write_len'  , 16)     #   - burst length
...                                                  #   - possibly more ports
writing     = Accelerator.reg('writing'    , 1)      # 1 bit flag indicating if
                                                     # it is writing to the bus
Accerlator.decode_exprs += [ write_grant == 1 ]
# The grant operation is considered as an instruction to the accelerator

writing_nxt = ila.ite( write_grant == 1,  # The effect of the instruction above
                   b0,                    # This is the complete function
                   writing )              # without holes. However, you can
                                          # use synthesis to create this
                                          # function
#Note: b0 = ila.const(0,1) (1-bit-wide constant 0)

Accelerator.set_next('writing', writing_nxt )

write_fsm    = Accelerator.add_microabstraction('WriteFsm', writing == 1 )
# Bus Write logic.
```

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

Instruction Ordering

Interrupt

For general purpose processors, the instruction-level abstraction (ILA) can not only capture the operations defined by its instruction-set architecture, but also the behavior of interrupts. As defined in the ILA definition, an instruction can be fetched from both the input port and the architectural states.

Take a processor with interrupt service routine (ISR) stored in a certain range of instruction memory, such as 8051 micro-controller, as an example. When the interrupt signal is raised, the processor stores the current program counter into the stack and updates the program counter to the entry point of the ISR. This can be modeled with the following strategies:

1. Include the interrupt signal as part of the instruction.
2. Evaluate the decode functions of normal instructions to *false* if any interrupt signal is raised.
3. Model the interrupt handling with a new instruction whose decode function is evaluated to *true* when the interrupt signal is raised.
4. The next state function of the new interrupt instruction updates the program counter to the ISR entry point.

Note that, in this example, instructions in the ISR are also normal instructions. The below shows part of an ILA template of the above example in modeling interrupts.

```
Processor   = ila.Abstraction('ExampleILA')      # Define the abstraction
                                                 # Define arch-states
intrpt_sig  = Processor.inp('intrpt_sig', 1)     #   - interrupt signal
instr_mem   = Processor.mem('instr_mem', 32, 8)  #   - instruction memory
pc          = Processor.reg('pc', 32)            #   - program counter
...                                              #   - registers, flags, ...

Processor.decode_exprs  = [(intrpt_sig == False) & (ISA_DECODEs)]
    # ISA_DECODEs: the decode functions of normal instructions (defined in ISA)
Processor.decode_exprs += [intrpt_sig == True]
    # Add a new instruction to handle the interrupt

pc_nxt_interrupt = ISR_entry
pc_nxt = ila.choice('pc_nxt', pc_nxt_interrupt, pc_nxt_normal)
Processor.set_next('pc', pc_nxt)
    # In addition to next state functions defined in the ISA (pc_nxt_normal),
    # the program counter now can be updated to the ISR entry point (ISA_entry)
    # when the interrupt signal is raised.
```

Specification and Micro-architecture