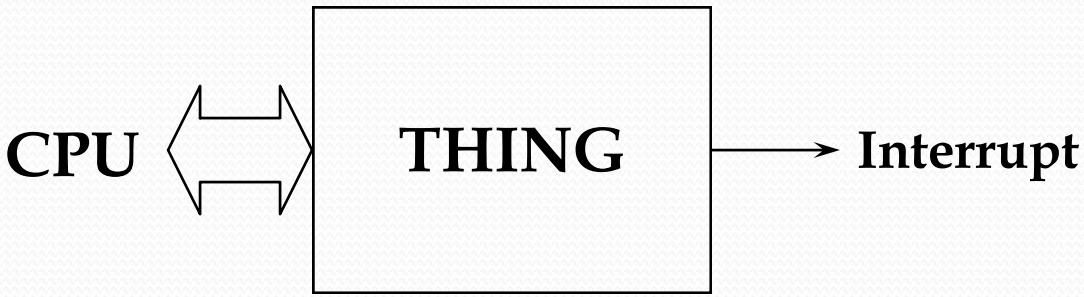
Lab 8.1

Lab 8.1

- In this lab, you will:
  - Learn how to write CPU bus-functional models
  - Learn how to perform read and write cycles using CPU bus-functional models

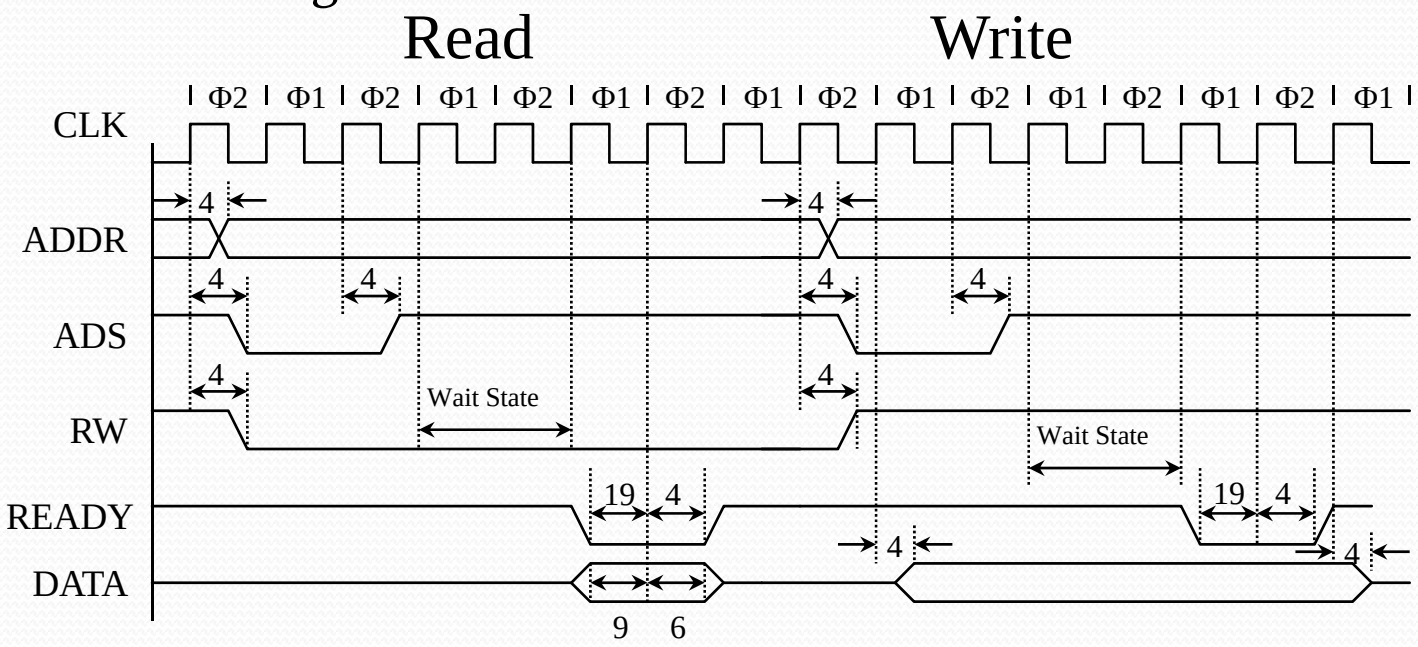
Lab 8.1

- In directory ~/lab8.1, you will find the following files:
  - thing.v Model for the "thing"
  - testthng.v Partial testbench for the "thing"
- The testbench module instantiates a THING that has a CPU interface and an interrupt output line.



Lab 8.1

- The CPU interface uses a 16MHz i386sx bus:
  - There can be 0 to any number of wait states
  - Timing values are in nanoseconds



Lab 8.1

- Pin description from the THING's perspective

Name	Direction	Width	Description
CLK	input	1	Clock
ADDR	input	8	Address
ADS	input	1	Address strobe
RW	input	1	Read when asserted low
READY	output	1	Data ready or data accepted. Introduces wait cycles.
DATA	inout	8	Data

- Register address map

Address	Register
0	Status
1	Mask

Lab 8.1

- Interrupts are generated whenever some condition occurs. In the real world, the THING would have other I/Os to detect these conditions. However, in this case, conditions occur spontaneously.
- The THING has three status bits in a read-only register at address 0 indicating which condition (A, B, or C) occurred.

Status register:

7	6	5	4	3	2	1	0
-	-	-	-	-	A	B	C

- Whenever one of the status bits is set, the interrupt output is asserted to '1'.

Lab 8.1

- The THING has a second register at address 1 to mask interrupts. If the mask bit is set, the corresponding status bit does not cause the interrupt output to be asserted.

Mask register:

7	6	5	4	3	2	1	0
-	-	-	-	-	A	B	C

Lab 8.1

- The test procedure follows these steps:
  - While not all conditions have occurred:
    - Verify that interrupt is cleared
    - Wait for the interrupt
    - Identify which condition caused the interrupt
    - Mask the interrupt for that condition
      - The interrupt should go away
- Review that the test procedure implements this algorithm.
  - Add any steps that may be missing.
- Your task is to complete the READ and WRITE tasks corresponding to the timing diagram.
- In what order and what time did the conditions occur?

Lab 8.1 : Optional

- Write a model of a 256x8bit RAM that could interface directly with the CPU.
- Using the procedures defined in the previous exercise, test your memory model by instantiating it in place of the THING.

Lab 8.2

Lab 8.2

- In this lab, you will:
  - Learn how to use hierarchical names
  - Learn how to bypass interfaces for additional visibility

Lab 8.2

- In directory ~/lab8.2, you will find the following files:
  - testfsm.v Testbench for FSM
  - fsm.v Model of an FSM
- The FSM implements a simple synchronizer with four states: out-of-sync, syncing, in-sync and losing-sync. How the FSM transitions between these states is unimportant currently.
- Your task is to have the testbench display the current state of the FSM at every rising edge of the clock using a hierarchical name.

Lab 8.2

- Add an *always* block in the testbench module that displays the current value of the state register at every rising edge of the clock.
- Simulate your model. What is the state transition sequence of the FSM?
- Modify the testbench to force the state machine to a different state during its execution.
- Verify that the state transition sequence has been successfully modified.

Lab 8.2 : Optional

- The state encoding for the FSM is as follows:
  - 00Out of sync
  - 01 Syncing
  - 11 In sync
  - 10 Sync loss
- Write a function that translates a state code into a meaningful string.
- Using this function, modify the trace message to display the name of the state instead of the code.
- Simulate to verify that the names are properly translated then displayed.

Lab 8.3

Lab 8.3

- In this lab, you will:
  - Learn how to back-annotate a gate-level simulation using an SDF file
  - Learn how to interface to a 3rd-party FPGA model
  - Learn how to program an LMG *SmartCircuit* FPGA model
  - Learn how to alternate between simulation configurations

Lab 8.3

- In directory ~/lab8.3, you will find the following files:
  - thing.rtl.v — RTL model for the THING
  - thing.gate.v — Gate-level model for the THING
  - thing.sdf — Post-PPR SDF file for gate-level model
  - thing.xnf — P&R XNF netlist for Xilinx XC4005-84
  - thing.rpt — Xilinx PPR report for "thing.xnf"
  - xc4005_84.pl — PERL script to produce pin connections
- The gate-level model uses Xilinx XC4000 primitives (or "gates"). An ASIC implementation would simply use a different set of gates - the full-timing gate-level simulation process remains the same.

Lab 8.3

- Simulate the RTL (synthesizable) model of the THING using the testbench previously written in Lab 8.1. Note that the interrupts may occur in a different order.
  - Ignore any hold violation error messages you may have.
  - At what time(s) is the interrupt asserted?

```
% ... ../lab8.1/testthng.v thing.rtl.v
```

- Edit the testbench (you may choose to copy the file ../lab8.1/testthng.v in this directory), and add the following statements at an appropriate location:

```
`ifdef SDF
initial $sdf_annotate("thing.sdf", DUT);
`endif
```

Lab 8.3

- Try to simulate the RTL model with SDF back-annotated delays using the command below.
  - Why is an error produced?

```
% ... +define+SDF ../lab8.1/testthng.v thing.rtl.v
```

- Simulate the gate-level model without SDF back-annotated delays using the following command (ignore any timing violation for the time being):
  - At what times are the interrupts asserted?
  - How does it differ from the RTL model? Why?

```
% ... ../lab8.1/testthng.v thing.gate.v \
        -y ../xc4000e +libext+.v
```

Lab 8.3: Optional

- LMG *SmartCircuit* FPGA models provide programmable models of the physical FPGA part. All physical pins on the device can be found on the module interface. It is necessary to correlate logical pins (in your design) with the physical pin (on the device).
- The file `thing.rpt` contains the pin assignment from the P&R tool. However, the assignments are described with respect to the physical pin number while the FPGA model pins are named to describe their functionality.
- Refer to the LMG documentation for the FPGA model found in the following file to identify the name of the pin that corresponds to a physical pad.

```
../lmc/templates/xc4005e_84/xc4005e_84.txt
```

Lab 8.3: Optional

- For example, RST was assigned to pin #60 which is named "CS0" in the LMG model. Therefore, the proper connection for the RST pin would be:

```
.CS0(RST)
```

- Manually completing the pin connections between the FPGA and the access module is a tedious and error- prone process that would have to be repeated after every FPGA P&R. Writing a script to perform the job is a good time-saving investment.
- The file "xc4005_84.pl" contains a PERL script that parses a PPR report file and produces the corresponding Verilog interface model
  for the LMG xc4005_84 model.

Lab 8.3: Optional

- Generate the interface model by executing the following command:

```
% xc4005_84.pl thing.rpt >thing.fpga.v
```

- The file `thing.fpga.v` contains an instantiation of a LMG *SmartCircuit* 84-pin XC4005E FPGA. This model is a generic model that needs to be programmed using an XNF file containing P&R and delay information.
- To program the FPGA model, create a Model Command File named `thing.mcf` containing the following line:

```
load -source thing.xnf
```

Lab 8.3: Optional

- To select the proper timing version of the FPGA and to identify the MCF file for configuring the FPGA model, add the following *defparam* statement to the FPGA thing model (or the PERL script).

```
defparam
   LMC.TimingVersion = "XC4005E_84-3",
   LMC.SCFFile       = "thing.mcf";
```

- Simulate the FPGA model using this info:

```
% ... ../lab8.2/testthng.v thing.fpga.v -y \
      ../lmc/special/cds/verilog/swift +libext+.v
```

  - At what time(s) is the interrupt asserted?
  - How does it differ from the gate-level+SDF model?

Lab 8.3: Optional

- Benchmark the simulation performance of the various THING models:
  - Behavioral (from Lab 8.1)
  - RTL
  - Verilog gate
  - LMG *SmartCircuit* model
- Which one would be best to debug a testbench?