LAB Assignment #4 for ECE 495/595

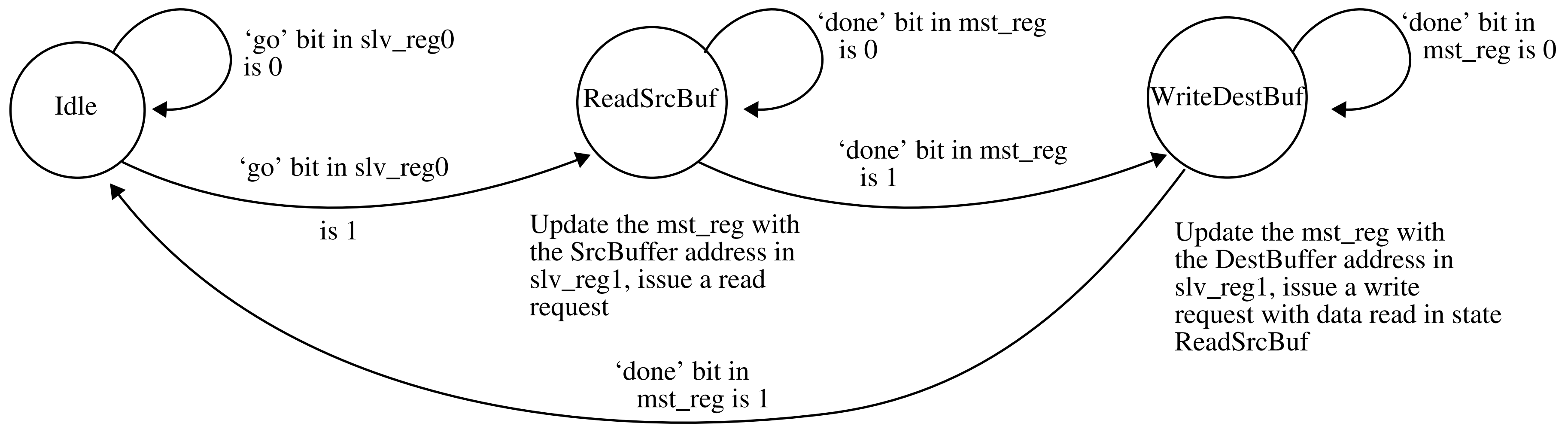
Assigned: Wed., Oct 26, 2011
Due: Wed., Nov. 2, 2011

Description: Implement a PLB bus master single beat controller

In this lab, you'll investigate a **bus master** interface of the PLB that is in the true spirit of hardware/software codesign. Bus master's can initiate memory read and write commands to the bus independently. They also enable the microprocessor(s) and hardware accelerators to 'share' memory, an extremely efficient way to have cooperating processes, both software and hardware, work together on some processing goal. Communication and synchronization can be accomplished using a set of software registers embedded embedded in the bus master interface.

You need to modify the user logic VHDL code for the bus master IP module and write a C program as follows (see drivers/bus_master_v1_00_a/src/bus_master_selftest.c for example C code):
C Code Sequence:
1) Write SrcBuffer with a value (any value will do -- use the selftest for loop if you want). Initialize the DestBuffer to 0.
2) Write the address of the SrcBuffer to slv_reg1 and the address of the DestBuffer to slv_reg2
3) Write a 'go' value into slv_reg0
4) Loop and continuously check for a match between SrcBuffer and DestBuffer. They will be different until the VHDL state machine has completed it's update (see below). When they become identical, print a message and exit the C program.
(NOTE: Be sure to declare the SrcBuffer and DestBuffer as 'volatile'. The default is declaration uses 'static').

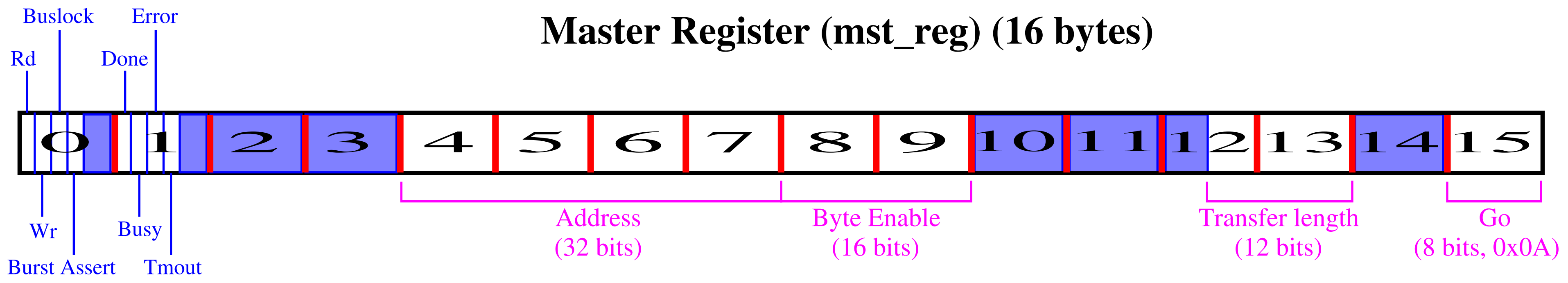
VHDL Requirements
Implement the following state machine:


Idle
'go' bit in slv_reg0 is 0
'go' bit in slv_reg0
is 1
ReadSrcBuf
'done' bit in mst_reg is 0
'done' bit in mst_reg is 1
Update the mst_reg with the SrcBuffer address in slv_reg1, issue a read request
WriteDestBuf
'done' bit in mst_reg is 0
Update the mst_reg with the DestBuffer address in slv_reg1, issue a write request with data read in state ReadSrcBuf
'done' bit in mst_reg is 1

Adding a Bus Master IP module:
The following describes the sequence of steps needed to add a bus master interface.
1) Run BSB and create a base project called bus_master.

2) Run Create Peripheral. At the **IP Interfaces Services** screen, click 'User Logic Software Registers' and 'Bus Master' (at the bottom) checkboxes. I added 3 registers under the **User S/W Registers** screen to allow the microblaze and the bus master to exchange information (e.g., the base address of an array).

3) Similar to the single beat PLB slave, the default template provided allows only the microblaze to read/write the master control register. (You will need to modify this if you want a hardware engine to able to issue commands.) The bus master template is somewhat more complicated than the slave module (which is also included in user_logic.vhd because you indicated software registers). The following describes the basic components and operations.



The 'center piece' of the bus master logic is the *mst_reg*, the **master register**. The layout of the register is shown above. It is 16 bytes in length and contains the following fields:

Byte 0: Control register (only most significant 4 bits defined)
Rd: read bit, when set, the control logic will initiate a read operation to a slave on the bus
Wr: write bit, when set, the control logic will initiate a write operation to a slave on the bus
Buslock: locks the bus when set (not needed but preserve the contents)
Bust Assert: indicates a burst transfer when set (not needed in a single beat master)

Byte 1: Status register (only most significant 4 bits defined)
Done: Transaction completed
Busy: Transaction in progress
Error: Error in transaction
Tmout: Timeout error

Bytes 2 and 3: Unused

Bytes 4-7: Address register
Memory address to read or write

Bytes 8 and 9: **Byte Enables**

Upto 16 byte enables are provided.

Bytes 10 and 11: Unused

Bytes 12 and 13: Transfer length in bytes

For single beat masters, always set to 4

Byte 14: Unused

Byte 15: Go register

Initiates a bus master operation, i.e., starts the state machine described below. This register is write-only (can not be read by microblaze).

The example code in ‘drivers/bus_master_v1_00_a/src/bus_master_selftest.c’ and ‘drivers/bus_master_v1_00_a/src/bus_master.c’ shows how microblaze can control this register for use as a *pseudo-DMA* controller. The self test module sets up two 4-byte buffers (aligned at 128) *SrcBuffer* and *DstBuffer*, loads them with default values using the loop:

```
for ( Index = 0; Index < BUS_MASTER_CORE_SELFTEST_BUFSIZE; Index++ )
{
  SrcBuffer[Index] = Index;
  DstBuffer[Index] = 0;
}
```

It then instructs the master to read a word into the internal FIFO embedded in the master, and then ‘busy-waits’ on the ‘done’ bit in the mst_reg.

```
BUS_MASTER_MasterRecvByte(baseaddr, (Xuint32) SrcBuffer, BUS_MASTER_CORE_SELFTEST_BUFSIZE);
while ( ! BUS_MASTER_mMasterDone(baseaddr) ) {}
```

It then instructs the bus master to write the 4-bytes read into it’s FIFO to the destination buffer.

```
BUS_MASTER_CORE_MasterSendByte(baseaddr, (Xuint32) DstBuffer, BUS_MASTER_CORE_SELFTEST_BUFSIZE);
while ( ! BUS_MASTER_mMasterDone(baseaddr) ) {}
```

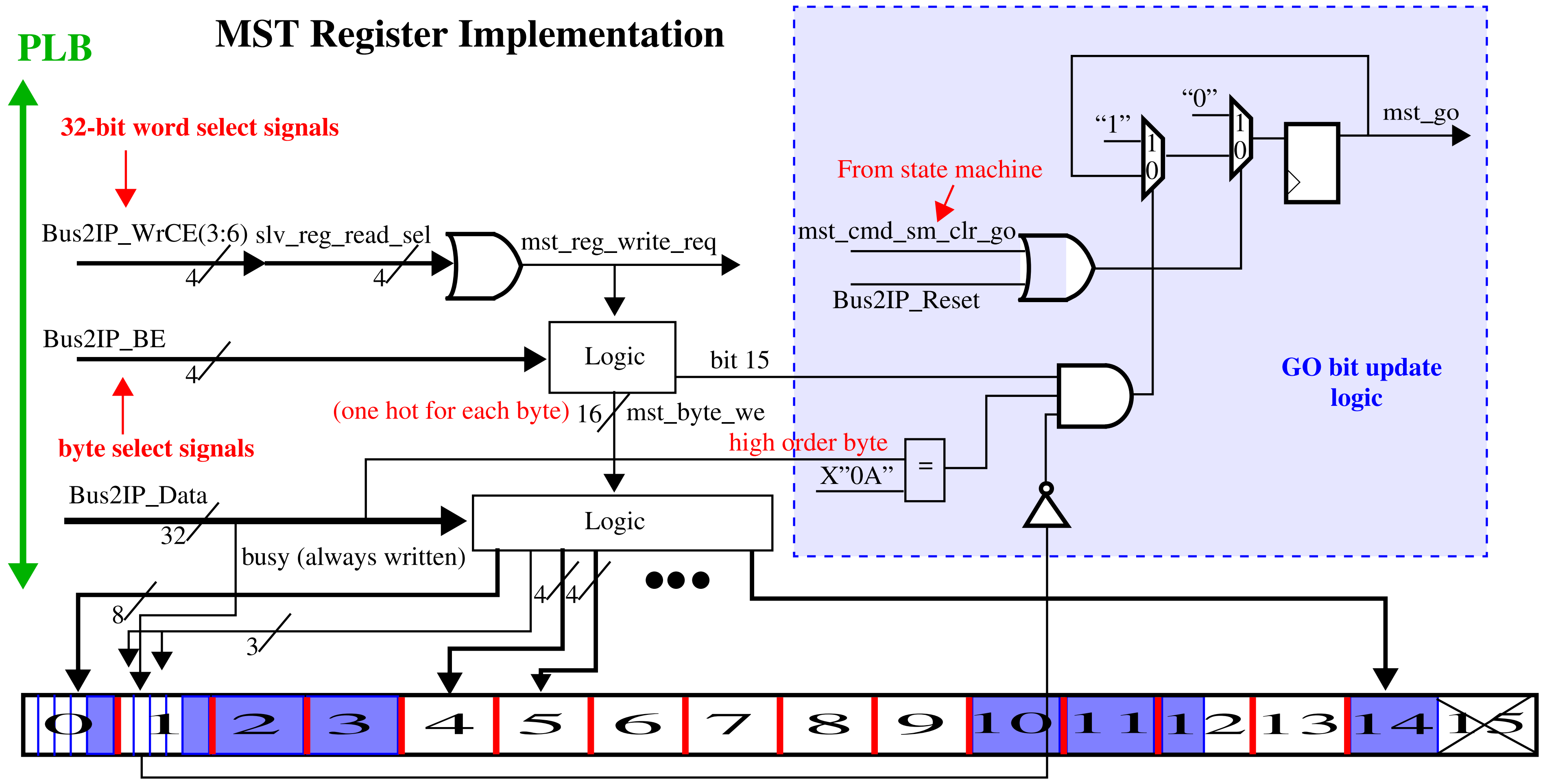
The ‘RecvByte’ code performs 5 operations:

a) Write ‘1’ into the ‘Rd’ bit of mst_reg.
b) Write SrcBuffer address into ‘addr’ of mst_reg.
c) Write 0xFFFF into ‘byte enable’ of mst_reg.
d) Write 4 into ‘transfer length’ of mst_reg.
e) Write ‘go’ byte.

The **MST Register Implementation** figure gives the schematic of the template code in ‘user_logic.vhd’ for implementing microblaze updates to the master register. You’ll need to modify this if you want to enable your hardware engine to issue bus commands.

Although you will **not** need to modify the generated **Bus Master’s State Machine**, it is instructive to understand how it works (see schematic above). The machine remains in IDLE until the ‘go’ signal. It sets the busy bit to ‘1’ and the go bit to ‘0’ on the rising edge. If the complete bit (‘Cmplt’) is not set and the controller (parent) has not acknowledged a command, then issue a command based on the contents of the master register. If the complete bit is not set but the command is acknowledged, then go to the ‘wait for data’ state. If ‘complete’ bit is set and we are still in this state, set error/timeout and proceed to ‘done’, resetting ‘busy’ to 0. So to succeed, the command must be acknowledged before the controller sets the ‘complete’ bit.

MST Register Implementation

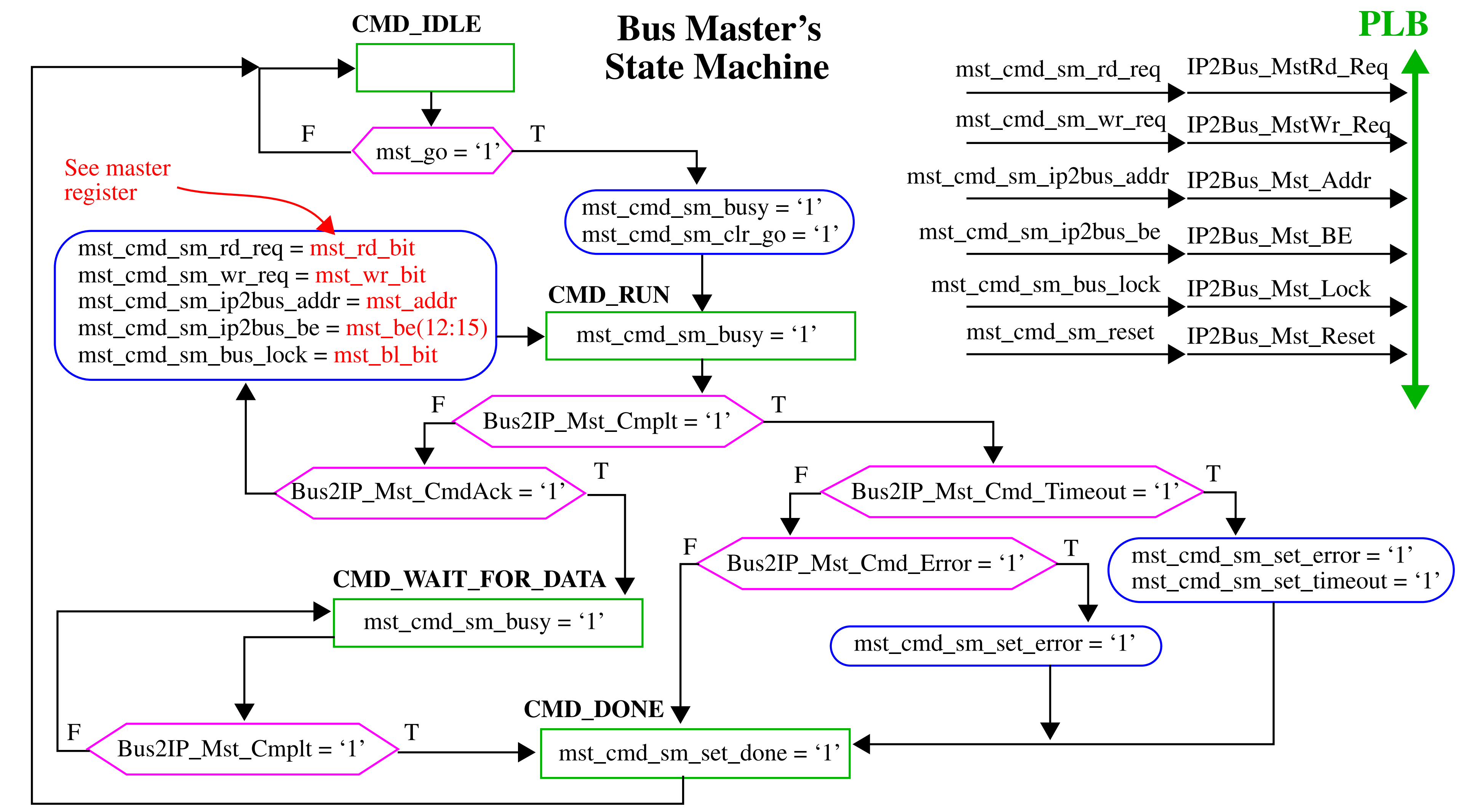


The last component of the VHDL is a FIFO buffer. Incoming and outgoing data are normally stored there (when the microblaze issues commands to the master reg). The incoming data is placed on *Bus2IP_MstRd_d* and is acknowledged on *Bus2IP_MstRd_src_rdy_n*. Outgoing data is placed on a separate bus, *Bus2IP_MstWr_d*, and acknowledged with *Bus2IP_MstWr_dst_rdy_n*. Again, you can choose to add to the functionality that exists or change it, i.e., eliminate the FIFO.

Laboratory Report Requirements:

1) No written report required for this laboratory. Be prepared to demonstrate your project in class on the due date.

Grading:



Proper operation: 100%