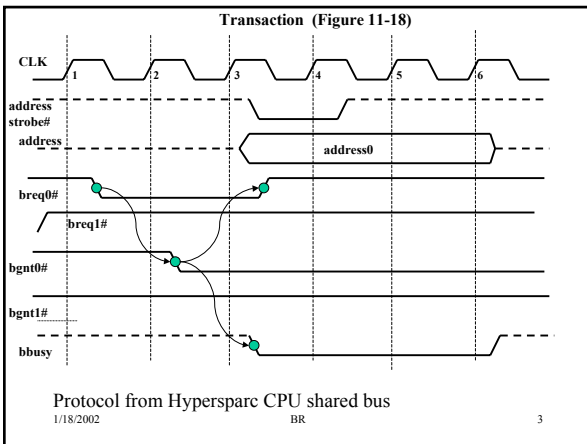
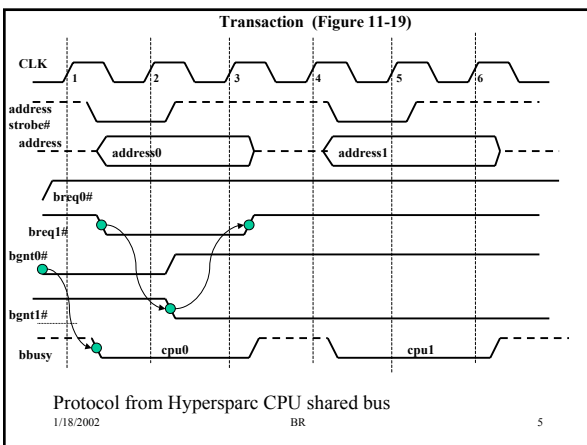


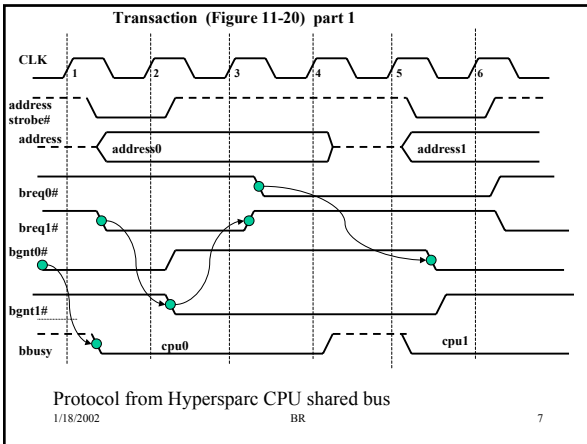
- ### Solving Bus Contention between Multiple Devices
- Central Arbitration - arbiter decides which device gets the bus via Bus Request/Bus Grant pairs
 - Single Bus Master – one device (the Root) initiates and controls all transfers (used by USB, IEEE Firewire)
 - Time Division Multiplexing (TDM) – give each device a scheduled time period in which to access the bus
 - Carrier Sense Multiple Access (CSMA)
 - Used by single-cable Ethernet
 - Each device listens to what it sends - if data is corrupted, then this means that a collision occurred (multiple devices tried to send)
 - On collision, wait a random amount of time ("backoff time"), then try again. On successive collisions, keep increasing the range of backoff time.
- 1/18/2002 BR 2



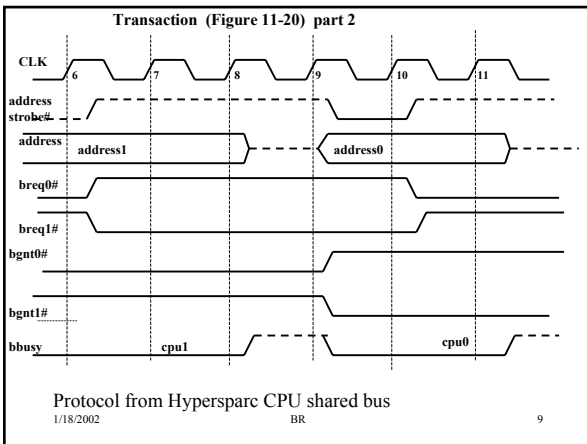
- ### Comments on Figure 11-18
- Note that all assertions occur after rising edge of clock
 - Dotted lines means that there is no active drive on line – may be pulled HIGH or may be at high impedance state
 - Transaction shows a single request for bus
 - CPU0 makes request in clk 1
 - Arbiter grants request in clk 2
 - CPU0 grabs bus in clk 3
- 1/18/2002 BR 4



- ### Comments on Figure 11-19
- Activity shows a bus handoff from CPU0 to CPU1
 - In clk1, arbiter has already granted bus to CPU0 and CPU0 grabs bus by asserting the *bbusy* line
 - In clk1, CPU1 also requests the bus
 - In clk2, Arbiter negates bus grant to CPU0, and asserts bus grant to CPU1
 - CPU1 has been granted the bus, but must wait until bus is free by monitoring the *bbusy* line. When *bbusy* goes high, then bus is free.
 - CPU0 releases bus in clock 3
 - CPU1 grabs bus in clock 4 by asserting *bbusy*
 - CPU1 releases bus in clock 6 by releasing *bbusy*
 - This is called an overlapped bus grant because arbiter granted bus to CPU1 while CPU0 still had the bus
 - minimizes handoff time between CPU0 and CPU1, only one clock cycle wasted (clock 3)
- 1/18/2002 BR 6



- Comments on Figure 11-20 (part 1)**
- This is similar to Figure 11-19 except that CPU0 makes a request for an additional transaction
 - In clk2, the arbiter grants the bus to CPU1 and negates the bus grant to CPU0
 - CPU0 makes an additional request for the bus by asserting bus grant in Clk 3. Note that CPU0 can do this even while it is busy with the first transaction!!! (*bbusy* is asserted by CPU0!!!). CPU0 is essentially asking for the bus ahead of time
 - CPU1 grabs the bus on clk5
 - The arbiter grants the bus to CPU0 in clk5 and negates the grant to CPU1
- 1/18/2002 BR 8



- Comments on Figure 11-20 (part 2)**
- This continues the transactions started in part 1
 - CPU1 releases the bus in Clk 8
 - CPU0 was granted the bus back in Clk 6 and has to wait until CPU1 releases the bus.
 - CPU0 grabs the bus in Clk 9 and releases the bus in Clk 11
- 1/18/2002 BR 10

- Simulation Goals**
- Want a VHDL simulation that can be used to test this protocol
 - You will be provided with a working Arbiter model
 - You will need to write a CPU model that can be used to ‘test’ the arbiter model
 - The CPU will read a data file that will control its behavior
 - By changing the CPU data file, we can test different situations
 - You will be provided with a test bench that instantiates two CPUs and the arbiter
 - Configurations will be used to test different situations
 - See the sim2 assignment page for a definition of the file format for the CPU data file.
- 1/18/2002 BR 11

- CPU Data File Format**
- Data file controls number of requests and duration of each request
 - First line: it contains a single integer that determines the clock cycle for the FIRST request
 - ‘-1’ means make NO requests, subsequent lines should be ignored
 - Subsequent lines contain 3 integers
 - The 1st integer is how many clock cycles to assert *bbusy* after the bus request has been granted
 - The 2nd integer is the number of clock cycles from the assertion of *bbusy* to the next request. A ‘-1’ means make no further requests.
 - The 3rd integer is the address to place on the address bus during the request.
- 1/18/2002 BR 12

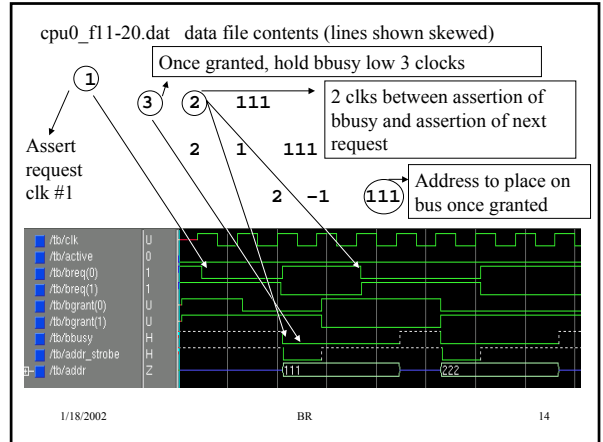
Test Cases

- Configurations used to specify data files that test the previous three waveforms plus two additional test cases
 - `cfg_tb_f11_20.vhd` specifies that `cpu#0` in the testbench use data file `cpu0_f11-20.dat`, and `cpu#1` use data file `cpu1_f11-20.dat`
 - All data files are already provided for you
- I have included data files/configurations for two other test cases labeled as Figure 11-21 and Figure 11-22 in the simulation WWW page.
- Figure 11-21 has CPU #0 make a series of requests while CPU#1 does nothing.
- Figure 11-22 has both CPU #0 and CPU#1 make a series of requests.

1/18/2002

BR

13



Modeling Approach

- You will need to use a FSM to model your CPU
 - Because all signals change after the rising clock edge and take at least one clock cycle to change, you can use either a two process model (Mealy) or a one process model (Moore)
 - Have your state registers be *rising edge triggered*
- You will need some initialization code that opens the file and reads the first line. One way is shown below:

```
myprocess (siga, sigb, whatever)
variable init: boolean;
begin
  if (init = FALSE) then
    -- do onetime code here...
    init = TRUE;
  end if;
  -- rest of code for process
  ....
end myprocess;
```

1/18/2002

BR

15

What States do you need?

- There are multiple correct solutions. Think about sequences that your CPU might have to do
 - Assert `Breq`, wait for bus grant, wait for bus busy to be negated, assert bus busy for #clks that defines the transaction, Negate bus busy when finished
- Use a defined type for your state definitions to make debugging easier.
- Figure 11-18 is the easiest, Fig 11-22 is the hardest.
- Some things to be aware of:
 - Will need to assert `Breq` in some situations while you are asserting `bbusy` (Figure 11-20)
 - After a bus grant, the bus may be free or it may not be free – must monitor the `bbusy` line
 - You will need counters to keep track of things like how long (in clock cycles) that you have asserted `bbusy`, when do you need to make the next bus request, etc.
 - The same CPU model is used for both CPU#0 and CPU#1 !!!

1/18/2002

BR

16

Other Hints

- Be careful about the use of variables in your processes
- May want to use signals to implement counter values
 - easier to control when they are updated
 - Can display them as waveforms for debugging purposes
- You do not have to duplicate the delays that are shown in the reference timing diagrams
 - I only care about what clock cycle the signal is asserted/negated in, not when in the clock cycle it is asserted/negated.
- The `addr`, `bbusy`, `addr_strobe` ports all have multiple drivers on them! Must drive with 'Z' if not actively driving the line.
- The 'active' output should be driven to '0' while a CPU is still active. It should be driven to 'Z' after the CPU has finished processing all entries in the data file.
 - This is used by the 'stim' entity to control generation of clock pulses.

1/18/2002

BR

17

Integer to std_logic_vector

- The address bus value in the CPU data files is an integer – need to convert this to a `std_logic_vector`

```
addr <= transport
  To_Std_Logic_vector (integer_val, length) after delay;
```

`to_std_logic_vector` converts `integer_val` to a `std_logic_vector` of `length` bits.

Contained in the `std_logic_1164_utils` package in `utilities` library.

```
Library utilities;
use utilities.standard_utils.all;
use utilities.std_logic_1164_utils.all;
```

Use both packages.

1/18/2002

BR

18