Myf.v

```
module myf;
        integer   f, index, r;
        reg [7:0] memory [0:15];
        initial begin
          $dumpfile("fdem.vcd");
          $dumpvars(0, myf);
        end
        initial begin
           f = $fopen("mem2.dat");
           $display("Generating contents of file mem2.dat");
           for(index = 0; index < 16; index = index + 1) begin
           r = $random;
           $fdisplay(f, "%b", r[12:5]);
           end
           $fclose(f);
           $readmemb("mem2.dat", memory);
           $display("\nContents of memory array: binary format");
           for(index = 0; index < 16; index = index + 1)
           $displayb(memory[index]);
           repeat(2)
           begin
             $display("\nContents of memory array: hexdecimal format");
             for(index = 0; index < 6; index = index + 1)
             $displayh(memory[index]);
           end
        end endmodule
```

Results:

Cir.v

```
`timescale 1 ns / 100 ps
module cir(a, b, f);
 input  a, b;
 output f;
 assign f = a | b;
endmodule
```

mycir_tb.v

```
`timescale 1 ns / 100 ps
module cir_tb;
 reg  a, b;
 wire f;
 cir g1 ( a, b, f);
 initial  begin
   a = 0; b = 0;
   #2 check_res( f, 0 );
   a = 0; b = 1;
   #2 check_res( f, 0 );
   a = 1; b = 0;
   #2 check_res( f, 0 );
   a = 1; b = 1;
   #2 check_res( f, 0 );
   #10 $stop;
 end
 task check_res;
 input data;
 input res;
 begin
   if( data != res) $display($time, "ns, Error: a=%b, b=%b, Expected value = %d, Actual value = %d \n", a, b,  res, data);
 end
 endtask
endmodule
```

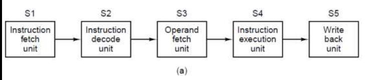
Demo.v

```
`timescale 1 ns / 100 ps
module demo (count, count_tri, clk, rst_l, load_l, enable_l, cnt_in, oe_l);
 output [3:0] count;
 output [3:0] count_tri;
 input clk;
 input rst_l;
 input load_l;
 input enable_l;
 input [3:0] cnt_in;
 input oe_l;
 reg [3:0] count;
 // tri-state buffers
 assign count_tri = (!oe_l) ? count : 4'bZZZZ;
 // synchronous 4 bit counter
 always @ (posedge clk or negedge rst_l)
 begin
  if (!rst_l) begin
    count <= #1 4'b0000;
  end
  else if (!load_l) begin
    count <= #1 cnt_in;
  end
  else if (!enable_l) begin
    count <= #1 count + 1;
  end
 end endmodule
```

Demo_tb.v

```
`timescale 1 ns / 100 ps
module demo_tb;
 reg clk_50;
 reg rst_l, load_l, enable_l;
 reg [3:0] count_in;
 reg oe_l;
 wire [3:0] cnt_out;
 wire [3:0] count_tri;
 demo U1 ( .count(cnt_out),
.count_tri(count_tri),  .clk(clk_50),
         .rst_l(rst_l),   .load_l(load_l),
.cnt_in(count_in),
         .enable_l(enable_l), .oe_l(oe_l)
 );
 // create a 50Mhz clock
 always
 #10 clk_50 = ~clk_50; // every ten nanoseconds invert
 initial   begin
   $display($time, " << Starting the Simulation >>");
   clk_50 = 1'b0;
    rst_l = 0;  enable_l = 1'b1; load_l = 1'b1; count_in = 4'h0;  oe_l = 4'b0;
   #20 rst_l = 1'b1;
    $display($time, " << Coming out of reset >>");
    @(negedge clk_50); // wait till the negedge of
    load_count(4'hA);
    @(negedge clk_50);
    $display($time, " << Turning ON the count enable >>");
    enable_l = 1'b0;
     wait (cnt_out == 4'b0001);
    $display($time, " << count = %d - Turning OFF the count enable >>", cnt_out);
    enable_l = 1'b1;
    #40;
    // let the simulation run for 40ns
    // the counter should not count
    $display($time, " << Turning OFF the OE >>");
    oe_l = 1'b1;

    // disable OE, the outputs of count_tri should go high Z.
    #20;
    $display($time, " << Simulation Complete >>");
    $stop;
 end
 initial begin
 // $monitor will print whenever a signal changes in the design
 $monitor($time, " clk_50=%b, rst_l=%b, enable_l=%b, load_l=%b, count_in=%h, cnt_out=%h, oe_l=%b, count_tri=%h", clk_50, rst_l,enable_l, load_l, count_in, cnt_out, oe_l, count_tri);
 end
 // The load_count task loads the counter with the value passed
 task load_count;
 input [3:0] load_value;
 begin
   @(negedge clk_50);
   $display($time, " << Loading the counter with %h >>", load_value);
   load_l = 1'b0;
```

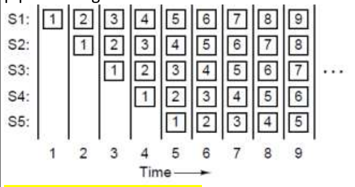


```
   count_in = load_value;
   @(negedge clk_50);
   load_l = 1'b1;
 end
 endtask
endmodule
```

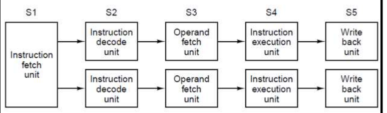
Design Principles for Modern Computers

All instructions directly executed
Maximize rate at which instructions are issued
Instructions should be easy to decode
Only loads and stores reference memory
Provide plenty of registers

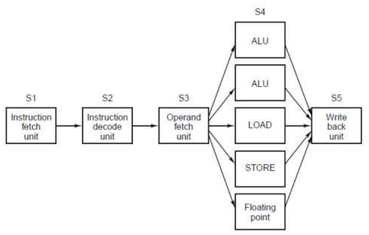
Pipelining: Speed depends on the slowest pipeline stage



Superscalar Architectures: speeds up processing of data but takes up more HW space



Dual five-stage pipelines with a common instruction fetch unit



A superscalar processor with five functional units

PCI Properties

Bus Type	Bus Width	Bus Speed	MB/Sec
PCI	32 bits	33 MHz	132 MBps
PCI	64 bits	33 MHz	264 MBps
PCI	64 bits	66 MHZ	512 MBps
PCI	64 bits	133 MHz	1 GBps

PCI Signals # = low active (can also be a bar)

AD[31:0] – I/O: 32-bit address/data bus, PCI is little endian (lowest # index is LSB). Carries the start address.

C/BE[3:0] – I/O: 4-bit command/byte enable bus, defines the PCI command during address phase and bytes to be transferred, indicates enable during data phase (each bit corresponds to a “byte lane” in AD[31:0] – example: C/BE[0] is the byte enable for AD[7:0]

PAR – I/O: Parity bit, used to verify correct transmittal of address/data and command/byte-enable. The XOR of AD[31:0], C/BE#[3:0], and PAR should return zero (even parity), in other words, the # of 1’s across these 37 signals should be even

FRAME# - I/O: Signals the start and end of a transaction. Indicates the start and duration.

TRDY# - I/O: When the target asserts this signal, it tells the initiator that its ready to send or receive data (TRDY depends on FRAME#, IRDY, TRDY)

IRDY# - I/O: Assertion by initiator indicates that it is ready to send receive data

STOP# - I/O: Used by target to indicate that it needs to terminate the transaction

DEVSEL# - I/O: Device select, part of the PCI distributed address decoding (each target is responsible for decoding the address associated with each transaction, when a target recognizes its address, it asserts DEVSEL# to claim the corresponding transaction)

GNT# - I: Asserted by system arbiter to grant bus ownership to the initiator. Point to point connection from arbiter – each initiator has its own GNT# line

CLK: PCI input clock, all signal sampled on rising edge, this clock can vary from 0-33 MHz, 33 MHz is really 33.333 MHz (30ns clk period). Rev 2.1 supports clk feq up to 66 MHz

RST#: When asserted, the reset signal forces all PCI configuration registers, masters and target state machines and output drivers to an initialized state. RST# may be asserted or deasserted asynchronously to the PCI clock edge PCI device must tri-state all I/Os during reset.

PCI Bus Operation

Initiator: Or master, owns the bus and initiates the data transfer, every initiator must also be a target

Target: or slave, target of the data transfer (read or write)

Agent: any initiator/target or target on the PCI Bus

Single-Function PCI: a package containing one function

Multi-function PCI: a package containing two or more PCI functions

Address Phase: Every PCI transaction starts off with an address phase one PCI clock period in duration. An exception is a transaction of 64-bit addressing wherin an initiator uses 54-bit addressing delivered in two address phases and consuming two PCI clock periods. During the address phase, the initiator identifies the target device (via the address) and the type of transaction. The target device is identified by driving a start address within its assigned range onto the PCI address/data bus. At the same time, the initiator identifies the type of transaction by driving the command type onto the 4-bit wide PCI Command/Byte Enable Bus The initiator also asserts the FRAME# signal to indicate the presence of a valid start address and transaction type of the bus. Since the initiator only presents the start address and command for one PCI clock cycle, it is the responsibility of every PCI target device to latch the address and command on the next rising-edge of the clock so that it may subsequently be decoded. Now, a target device can determine if it is being addressed and the type of transaction in progress. Upon completion of the address phase. the address data bus becomes the data bus for the duration of the transaction and is used to transfer data in each of the data phases. It is the responsibility of the target to latch the start address and to auto-increment it (assuming that the target supports burst transfers) to point to the next group of locations (a dword or a quadword) during each subsequent data transfer.

Claiming the Transaction: When a PCI target determines that it is the target of a transaction, it must claim the transaction by asserting DEVSEL#. If the initiator does not sample DEVSEL# asserted within the predetermined amount of time, it aborts the transaction

Data Phase: The # of bytes to be transferred during a data phase is determined by the # of Command/Byte Enable signals that are asserted by the initiator during the data phase. Each data phase is at least one PCI clock period in duration. Both initiator and the target must indicate that they are ready to complete a data phase, or the data phase is extended by a wait state one PCI CLK in period duration. The PCI bus defines ready signal lines used by both the initiator (IRDY) and the target (TRDY#) for this purpose

Burst Transfer: one consisting of a single address phase followed by 2 or more data phases. The bus master only has to arbitrate for bus ownership one time. Each burst consists of an address phase and data phase.

Transaction: FRAME# is asserted at the start of the address phase and remains asserted until the initiator is ready (asserts IRDY#) to complete the final data phase. When the target samples IRDY# asserted and FRAME# deasserted in a data phase, it realizes that this is the final data phase. However, the data phase will not complete until the target has also asserted the TRDY# signal.

PCI Bus Arbitration

Arbiter: Refers to the process by which one or more PCI bus master devices may require use of the PCI bus to perform a data transfer with another PCI device. Each requesting master asserts its REQ# output to inform the bus arbiter of its pending request of the bus. Uses

Centralized Arbitration: independent grant and request lines, does not specify a particular policy (mandates the use of fair policy). Arbiter has independent grant and request lines for each device. If a master generates a request, is subsequently granted the bus and does not initiate a transaction (assert FRAME#) within 16 PC1 clocks after the bus goes idle, the arbiter may assume that the master is malfunctioning

Fairness: The central arbiter is required to implement a fairness algorithm to avoid deadlocks. Fairness means that each potential bus master must be granted access to the bus independent of other requests. Fairness is defined as a policy that ensures that high-priority masters will not dominate the bus to the exclusion of lower-priority masters when they are continually requesting the bus. However, this does not mean that all agents are required to have equal access to the bus.

PCI Bus State

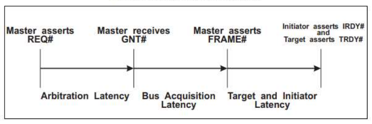
FRAME#	IRDY#	Description
deasserted	deasserted	Bus Idle.
deasserted	asserted	Initiator is ready to complete the last data transfer transaction, but it has not yet completed.
asserted	deasserted	A transaction is in progress and the initiator is not ready to complete the current data phase.
asserted	asserted	A transaction is in progress and the initiator is read complete the current data phase.

PCI Bus Parking: A master must only assert its REQ# to signal a current need for the bus. If a systems designer implements a bus parking scheme, a default bus owner should be defined when no other masters request bus. The choice of which master to park the bus on is defined by the by designer of the bus arbiter

Hidden Arbitration: Unlike some arbitration schemes, the PCI scheme allows bus arbitration to take place while the current PCI bus master is performing a data transfer. No bus time is wasted on a dedicated period to perform an arbitration bus cycle. This is referred to as hidden arbitration.

Latency

Figure 6-1: Access Latency Components



Maximum Latency Configuration Register: Ideally, the bus arbiter should be programmable by the system. If it is, the start up config SW can determine the priority to be assigned to each member of the bus master community by reading from the maximum latency config register associated with each bus master. The bus master designer hardwires this register to indicate, in increments of 250 ns, how quickly the master requires the access to the bus in order to achieve adequate performance.

Bus Access Latency: the amount of time that expires from the moment a bus master requests the use of the PCI bus until it completes the first data transfer of the transaction. In other words, it is the sum of arbitration, bus acquisition and target latency.

Arbitration Latency: the period from the bus master's assertion of REQ# until the bus arbiter asserts the bus master's GNT#.
This period is a function of the arbitration algorithm, the master's priority and whether any other masters are requesting access to the bus.

Bus Acquisition Latency: the period time from the reception of GNT# by the requesting bus master until the current bus master surrenders the bus. The requesting bus master can then initiate its transaction by asserting FRAME#. The duration of the period is a function of how long the current bus master's transaction-in progress takes to complete. This parameter is the larger of either the current master's LT value (in other words, its time slice) or the longest latency to first data phase completion in the system (which is limited to a maximum of 16clocks).

Initiator & Target Latency: the period from the start of a transaction until the master and the currently-addressed target are ready to complete the first data transfer of the transaction. This period is a function how fast the master can transfer the first data item, as well as the access time for the currently addressed target device (and is limited to a maximum of 8 clocks for the master and 16docks for the target)

Timing Requirement for Master: It is a rule that the initiator must not keep IRDY# deasserted for more than 7 PCI clock cycles during any data phase. In other words, it must be prepared to transfer a data item within 8 clock cycles after entry into any data phase.

Master Latency Timer (LT): It is either initialized by the configuration software at startup time or contains a hardwired value. The value contained in the LT defines the minimum amount of time (in PCI clock periods) that the bus master is permitted to retain ownership of the bus each time that it acquires bus ownership and initiates a transaction

Preventing Master from Monopolizing Bus: **1.** Inclusion of LT associated with each master **2.** The rule that required the initiator to keep IRDY deasserted for no longer than 8 PCI clock cycles during any data phase

Preventing Target from Monopolizing Bus: **1.** The target cannot transfer the first data item within 16 clock cycles from the assertion of FRAME#. **2.** Although the target can transfer the first data item within 16 clocks from the assertion of FRAME#, it cannot transfer one of the subsequent data items within 8 clock cycles from the start of the data phase. Any data phase other than the first one is referred to as a *subsequent data phase*.

Case 1: asserts DEVSEL# to claim the transaction, does not assert TRDY#, thereby indicating its unwillingness to transfer the first data item, asserts STOP# to indicate that it wants to terminate the transaction with no data transferred in the 1st data phase and therefore no data transferred in the transaction. Master Response: terminate the transaction with no data transferred, thus freeing up the bus for other masters to use. After 2 PCI clocks have elapsed, the master can reassert its request and when it receives GNT#, reinitiate its transaction again

Can LT Value Be Hardwired? Yes, for a master that performs one or two data phases per transaction, but the hardwired value may not exceed 16 (and it could be zero). The implication is your master has a timeslice of zero! In other words, if you Mtiate a transaction and the arbiter immediately removes your GNT# (because another master is requesting the bus), your master can perform one (and only one) data phase and must relinquish the bus.

Master/Target Abort Handling: the transaction on the destination bus ended in a Master Abort: bc no target responded, or in a Target Abort bc the target is broken, or the target does not support the byte enable combination

Read/Write Data Phase

Table 7-1: PCI Command Types

C/BE[3:0]# (binary)	Command Type
0000	Interrupt Acknowledge
0001	Special Cycle
0010	I/O Read
0011	I/O Write
0100	Reserved
0101	
0110	Memory Read
0111	Memory Write
1000	Reserved
1001	
1010	Configuration Read
1011	Configuration Write
1100	Memory Read Multiple
1101	Dual Address Cycle
1110	Memory Read Line
1111	Memory Write-and-Invalidate

Byte Enables: they may change in each data phase: PCI permits burst transactions where the byte enables change from one data phase to the next. The initiator may use any byte enable setting, consisting contiguous or noncontiguous byte enables.
During a read transaction, the initiator will typically assert all of the byte enables during each data phase (bc burst reads are typically reading a steam of dwords or quadwords), but it may use any combination. During burst transfer, if all byte enables are deasserted during data phase, then a dword will be "skipped".

Target with limited byte enable support: I/O and memory targets may support restricted byte enable settings and may respond with Target Abort for any other pattern. All devices must support any byte enable combo during config transactions.

Performance During Read Transactions: PCI permits burst transactions where the byte enables change from one data phase to the next. The initiator may use any byte enable setting, consisting of contiguous or noncontiguous byte enables.
During a read transaction, the initiator will typically assert all of the byte enables during each data phase (because burst reads are typically reading a stream of dwords or quadwords), but it may use any combination. During a burst transfer, If all byte enables are deasserted during the data phase, then a dword will be "skipped". In actual practice, most read transactions Involve a burst transfer of multiple objects (dwords or quadwords) between the initiator and the currently addressed target. The read transaction involving multiple data phases only requires the turnaround cycle during the first data phase. The second through the last data phases can each be accomplished in a single clock cycle (if both the initiator and the currently-addressed target are capable of zero wait state transfers). The achievable transfer rate during the second through the last data phases is thus one transfer every 30ns (at a PCI bus speed of 33MHz) or 33 million transfers per second. If each data phase involves the transfer of four bytes, the resultant data transfer rate is 132Mbytes per second.

Performance During Write Transactions: The first through the last data transfer of a burst write transaction can each be accomplished in a single clock cycle (if both the initiator and the currently addressed target are capable of zero wait state data phases).The achievable transaction rate during the first through the last data phases is thus one transfer every 30ns (at a PCI bus speed of 33MHz), or 33 million transfers per second. If each data transfer involves the transfer of four bytes, the data transfer rate Is132Mbytes per second

Request/Grant Timing

When the arbiter determines that it is a master's turn to use the bus, it asserts the master's GNT# line. The arbiter may deassert a master's GNT# on any PCI clock. A master must ensure that its GNT# is asserted on the rising clock edge on which it wishes to start a transaction. If GNT# is deasserted, the transaction must not proceed. Once asserted by the arbiter, GNT# may be deasserted under the following circumstances:

- If GNT# is deasserted and FRAME# is asserted the transfer is valid and will continue. The deassertion of GNT# by the arbiter indicates that the master will no longer own the bus at the completion of the transaction currently in progress. The master keeps FRAME# asserted while the current transaction is still in progress. It deasserts FRAME# when it is ready to complete the final data phase.
- The GNT# to one master can be deasserted simultaneously with the assertion of another master's GNT# **if the bus isn't in the idle state**. The idle state is defined as a clock cycle during which both FRAME# and IRDY# are deasserted. If the bus appears to be idle, the master whose GNT# is being removed may be using stepping to drive the bus (even though it hasn't asserted FRAME# yet; stepping is covered in "Address/Data Stepping" on page 162). The coincidental deassertion of its GNT# along with the assertion of another master's GNT# could result in contention on the AD bus. The other master could immediately start a transaction (because the bus is technically idle). The problem is prevented by delaying grant to the other master by one cycle. Table 5-1 on page 67 defines the bus state as indicated by the current state of FRAME# and IRDY#.
- GNT# may be deasserted during the final data phase (FRAME# is deasserted) in response to the current bus master's REQ# being deasserted.

How LT Works

When the bus master detects bus idle (FRAME# and IRDY# deasserted) and its GNT# asserted, it has bus acquisition and may initiate a transaction. Upon initiation of the transaction, the master's LT is initialized to the value written to the LT by the configuration software at startup time (or its hardwired value). Starting on the next rising-edge of the PCI clock and on every subsequent rising-edge, the master decrements its LT by one.

If the master is in the midst of a burst transaction and the arbiter removes its GNT# (in other words, it is preempted), this indicates that the arbiter has detected a request from another master and is granting ownership of the bus for the next transaction to the other master. In other words, the current master has been preempted.

If the current master's LT has not yet been exhausted (decremented all the way down), it has not yet used up its timeslice and may retain ownership of the bus until either:

- it completes its burst transaction or
- its LT expires,

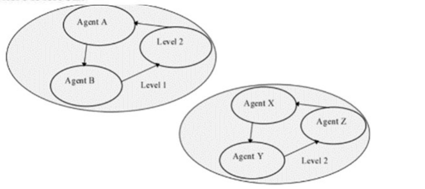
whichever comes first. If it is able to complete its burst before expiration of its LT, the other master that has its GNT# may assume bus ownership when it detects that the current master has returned the bus to the idle state. If the current master is not able to complete its burst transfer before expiration of its LT, it is permitted to complete one more data transfer and must then yield the bus.

If the current master has exhausted its LT, still has its GNT# and has not yet completed it burst transfer, it may retain ownership of the bus and continue to burst data until either:

- it completes its overall burst transfer or
- its GNT# is removed by the arbiter.

In the latter case, the current master is permitted to complete one more data transfer and must then yield the bus.

It should be noted that, when forced to prematurely terminate a data transfer, the bus master must "remember" where it was in the transfer. After a brief period, it may then reassert its REQ# to request bus ownership again so that it may continue where it left off.



Assume the following conditions:

- Master A is the next to receive the bus in the first group.
- Master X is the next to receive it in the second group.
- A master in the first group is the next to receive the bus.

If all agents on level 1 and 2 have their **REQ#** lines asserted and continue to assert them, and if Agent A is the next to receive the bus for Level 1 and Agent X is the next for Level 2, then the order of the agents accessing the bus would be:

A, B, Level 2 (this time it is X)

A, B, Level 2 (this time it is Y)

A, B, Level 2 (this time it is Z)

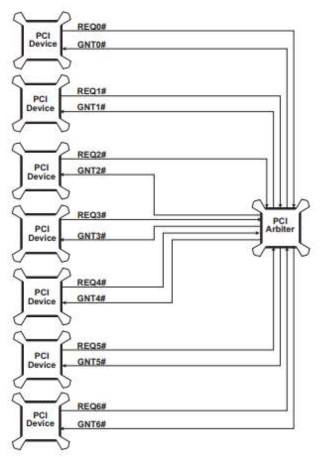
and so forth.

If only Agent B and Agent Y had their **REQ#**s asserted and continued to assert them, the order would be:

B, Level 2 (Y),

B, Level 2 (Y).

Figure 5-1: The PCI Bus Arbiter


PCI Device
REQ0#
GNT0#
PCI Device
REQ1#
GNT1#
PCI Device
REQ2#
GNT2#
PCI Device
REQ3#
GNT3#
PCI Device
REQ4#
GNT4#
PCI Device
REQ5#
GNT5#
PCI Device
REQ6#
GNT6#
PCI Arbiter