A SIMPLE SYSTEM WITH DIRECT MEMORY ACCESS MODULE IN VERILOG

Interface Circuits

Mohammad Parsa Bashari 400104812

Fall 2024

Contents

1 Problem Statement

In modern computing systems, efficient data transfer between memory and peripheral devices is critical for achieving high performance. Direct Memory Access (DMA) modules play a key role in enabling such efficient transfers without burdening the processor. In this section, we describe the problems with the traditional approach and explain how DMA addresses these issues.

1.1 Traditional Approach: Processor-Driven Data Transfer

In traditional systems, data transfer between memory and peripheral devices (I/O modules) is managed entirely by the processor. This approach involves the processor executing a series of instructions to read data from the source, temporarily store it in registers, and then write it to the destination. While simple to implement, this method has several drawbacks. High CPU utilization occurs because the processor is heavily involved in managing the data transfer, leaving fewer resources available for executing application logic. The data transfer speed is limited by the processor's instruction execution rate, resulting in inefficiencies for large-scale or frequent data transfers.

1.2 Direct Memory Access (DMA)

Direct Memory Access (DMA) is a hardware-based mechanism designed to overcome the limitations of processor-driven data transfer. A DMA controller operates independently of the processor, directly managing data transfers between memory and peripherals. This reduces processor involvement and improves system efficiency. The DMA controller offloads data transfer tasks from the processor, allowing it to focus on computational workloads.

2 Implementation

2.1 General Architecture

Figure 1 illustrates the general architecture of our system, along with the bus structure. The system contains four main modules (CPU, Memory, I/O, and DMA) that are connected by two buses: (1) System Bus, which connects Memory, CPU, and DMA; and (2) Peripheral Bus, which connects the I/O module to DMA.

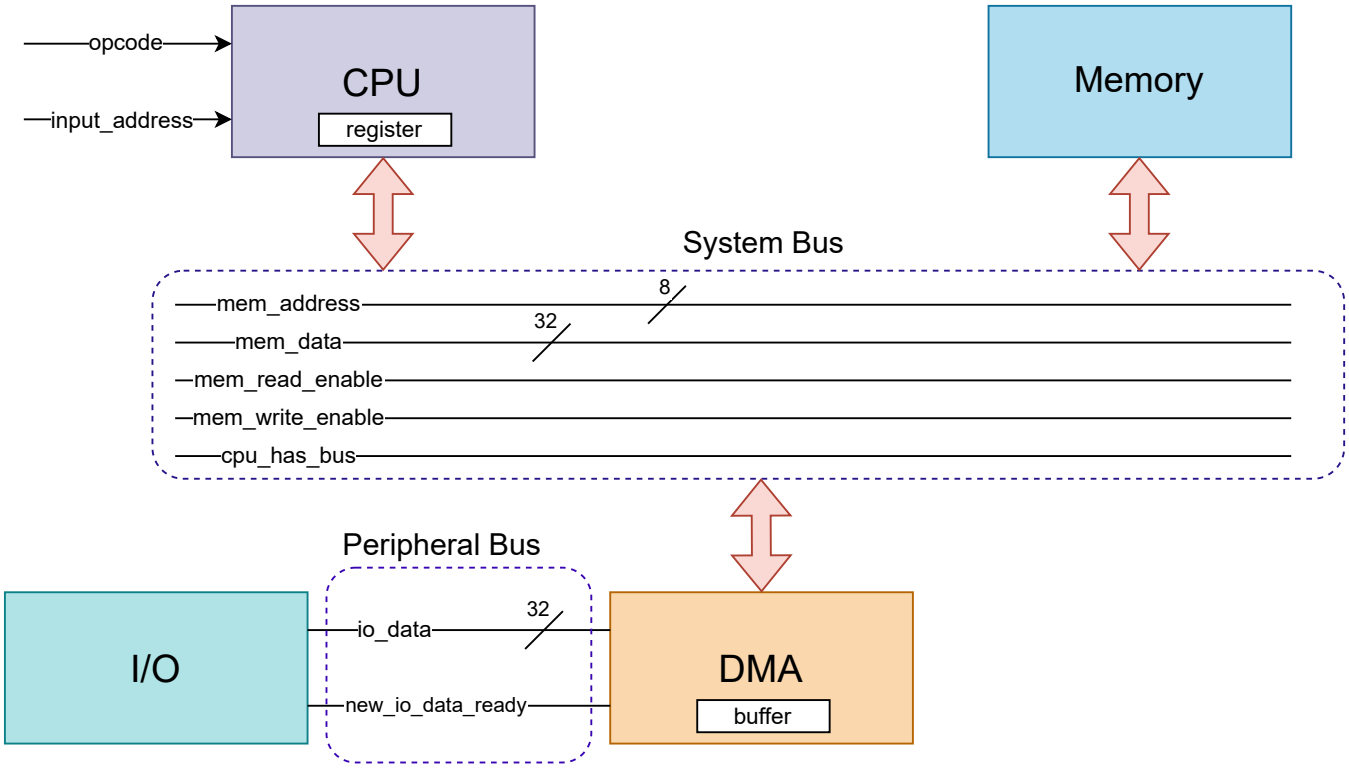

Figure 1: General system architecture.

Table 1 provides explanations for the signals in the system bus. Along with some basic signals, such as clock and reset, there are data and address lines (32 bits and 8 bits, respectively). There are also two control signals, `mem_read_enable` and `mem_write_enable`, to perform memory module operations. For bus arbitration purposes, there is a signal, `cpu_has_bus` that shows if the CPU is using the bus. DMA module will use this signal to perform cycle stealing when the CPU is not using the bus.

Signal	**Explanation**
`mem_address` (8 bits)	Address line.
`mem_data` (32 bits)	Data line.
`mem_read_enable`	Read enable signal for memory.
`mem_write_enable`	Write enable signal for memory.
`cpu_has_bus`	Shows if the CPU is using the bus.

Table 1: System bus signal lines.

Table 2 provides explanations for the signals in the peripheral bus. When there is new data ready in the I/O module, it sets the `new_io_data_ready` signal to one and puts the new data on the `io_data` line. DMA is then notified and stores the new data in its buffer. DMA then waits for the bus to be free and uses cycle stealing to transfer the data to memory.

Signal	Explanation
`io_data` (32 bits)	Data line for transferring from I/O module to DMA.
`new_io_data_ready`	Shows if I/O module has a new data ready.

Table 2: Peripheral bus signal lines.

This system is implemented using the hardware description language, Verilog. The following subsections provide the code along with an explanation for each module in the system.

2.2 I/O Module

The I/O module is implemented in a way that simulates a peripheral device such as a keyboard. The Verilog code for the I/O module is shown in Code Snippet 1.

```
1  module IO(
2      input clk,
3      input reset,
4      output reg new_data_ready,
5      output reg [31:0] io_data
6  );
7      reg [34:0] counter; // bits 34:3 is data, bits 2:0 for counting 8 cycles
8
9      always @(posedge clk or posedge reset) begin
10         if (reset) begin
11             counter <= 0;
12         end else begin
13             counter = counter + 1;
14             new_data_ready <= 0;
15             if (counter[2:0] == 3'b1) begin // generate a new data every 8 cycles
16                 io_data <= counter[34:3];
17                 new_data_ready <= 1;
18             end
19         end
20     end
21 endmodule
```

Code Snippet 1: I/O Module Verilog Code.

This I/O module uses a counter which generates new data every 8 cycles. The counter is 35 bits itself. The 3 rightmost bits are used to count every 8 cycles and the rest 32 bits are used as I/O data. So by setting the counter to zero at reset (line 11), it produces subsequent numbers (0, 1, 2, ...) every 8 cycles (lines 16 and 17).

2.3 Memory Module

Code Snippet 2 shows the Verilog code for the memory module. This memory contains 256 words, each 32 bits (1KB in total). The data line is defined as `inout` to enable a duplex pin, used for both write and read data. If the `read_enable` is active, this data line is set to the data in memory at the address in the address line (line 17). If `read_enable` is not active, this line is set to high-Z, enabling an outer drive of the data signal (for a possible write operation). Note that the write operation is synchronous (lines 11-15) when the read operation is not.

```verilog
module Memory(
    input clk,
    input reset,
    input [7:0] address,
    inout [31:0] data,
    input write_enable,
    input read_enable
);
    reg [31:0] mem [0:255]; // 256-word memory (each word 32 bits)

    always @(posedge clk) begin
        if (write_enable) begin
            mem[address] = data;
        end
    end

    assign data = (read_enable) ? mem[address] : 32'bz;
endmodule
```

Code Snippet 2: Memory Module Verilog Code.

2.4 CPU Module

Code Snippet 3 shows the Verilog code for the CPU module. As shown in Figure 1, the CPU module gets `opcode` (2 bits) and `input_address` (8 bits) as input. These inputs determine the instruction that the CPU has to execute and are used later in the testbench. The CPU is connected to the system bus (described in Table 1) using its ports. The `mem_data` port is defined as `inout` to enable both `LOAD` and `STORE` instructions to use the same data channel. In addition, the CPU has an internal 32-bit register (line 11) to enable load/store operations.

The CPU determines the control signals at the positive edge of each clock based on the value of `opcode`. In the case of 01, the CPU issues a `LOAD` instruction that reads from a memory location and stores the result in the CPU's internal register. In the case of 10, the CPU issues a `STORE` instruction that writes the value of the internal register to a memory location. In both cases above, the CPU activates the signal `get_bus` (which is connected to the `cpu_has_bus` signal in the system bus) to get control of the bus. A 00 or 11 opcode shows a `NOP`. In this case, the CPU disables the `get_bus` signal to let the DMA steal the cycles of `NOP`. In the end, the actual outputs of the CPU module are controlled by the signal `get_bus` (lines 17-20), setting them to high-Z when the CPU is not using the bus (like a tri-state buffer).

```verilog
module CPU(
    input clk,
    input reset,
    output reg get_bus,
    output [7:0] mem_address,
    inout [31:0] mem_data,
    output mem_write_enable,
    output mem_read_enable,
    input [1:0] opcode, // 2-bit opcode: 01 - LOAD, 10 - STORE, otherwise - NOP
    input [7:0] input_address, // Address for LOAD/STORE
    output reg [31:0] register // Internal register to simulate load/store operations
);
    reg [31:0] data;
    reg [7:0] address;
    reg write_enable;
    reg read_enable;
```

```
17      assign mem_data = (get_bus & write_enable) ? data : 32'bz;
18      assign mem_address = (get_bus) ? address : 8'bz;
19      assign mem_write_enable = (get_bus) ? write_enable : 1'bz;
20      assign mem_read_enable = (get_bus) ? read_enable : 1'bz;
21
22      always @(posedge clk or posedge reset) begin
23          if (reset) begin
24              get_bus <= 0;
25              write_enable <= 0;
26              read_enable <= 0;
27              address <= 8'b0;
28              register <= 32'b0;
29          end else begin
30            if (read_enable) register = mem_data;
31              case (opcode)
32                  2'b01: begin // LOAD
33                      get_bus = 1;
34                      read_enable = 1;
35                      write_enable = 0;
36                      address = input_address;
37                  end
38                  2'b10: begin // STORE
39                      get_bus = 1;
40                      read_enable = 0;
41                      write_enable = 1;
42                      address = input_address;
43                      data = register;
44                  end
45                  default: begin // NOP
46                      get_bus = 0;
47                  end
48              endcase
49          end
50      end
51  endmodule
```

Code Snippet 3: CPU Module Verilog Code.

2.5 DMA Module

Code Snippet 4 shows the Verilog code for the DMA module. According to the use of a memory-mapped I/O scheme, the memory location 0 is dedicated to the I/O module data. The DMA module checks if the I/O module has new data and if so, it stores this data in an internal buffer (defined in line 12) and sets the variable `new_data_in_buffer` to one, showing that a new data is in the buffer waiting for being written in memory. Based on the cycle-stealing approach, the DMA module then waits for the signal `cpu_has_bus` to get zero (bus become free), and then writes this data in the memory location 0.

It is worth noting that if the CPU does not free the bus until another data arrives from the I/O module, the previous data in the DMA's buffer will be overwritten by the new one. In addition, in the same way as in the CPU module, the actual outputs of the DMA module are controlled by the `cpu_has_bus` signal in a tri-state buffer style to free the bus (by setting high-Z values) when the CPU is driving the bus lines.

```
1   module DMA(
2       input clk,
3       input reset,
```

```
    input cpu_has_bus,
    input [31:0] io_data,
    input new_io_data_ready,
    output [7:0] mem_address,
    output [31:0] mem_data,
    output mem_write_enable,
    output mem_read_enable
);
    reg [31:0] buffer;
    reg new_data_in_buffer;

    assign mem_address = (!cpu_has_bus & new_data_in_buffer) ? 8'b0 : 8'bz;
    assign mem_data = (!cpu_has_bus & new_data_in_buffer) ? buffer : 32'bz;
    assign mem_read_enable = (!cpu_has_bus) ? 1'b0 : 1'bz;
    assign mem_write_enable = (!cpu_has_bus) ? new_data_in_buffer : 1'bz;

    always @(posedge clk or posedge new_io_data_ready) begin
        if (new_io_data_ready) begin
            buffer = io_data;
            new_data_in_buffer = 1;
        end else if (!cpu_has_bus & mem_write_enable)
            new_data_in_buffer = 0;  // if CPU released the bus and DMA wrote into Memory
    end

    always @(posedge reset) begin
        buffer = 0;
        new_data_in_buffer = 0;
    end
endmodule
```

Code Snippet 4: DMA Module Verilog Code

2.6 Top Module

In the top module (named `System`), which contains the whole system, the four main modules are instantiated and are connected through two buses as shown in Figure 1. The Verilog code of this module is shown in Code Snippet 5.

```
module System(
    input clk,
    input reset,
    input [1:0] opcode, // Opcode for CPU (01 - LOAD, 10 - STORE, otherwise - NOP)
    input [7:0] input_address, // Address for LOAD/STORE
    output [31:0] cpu_register
);

    // System Bus (CPU, Memory, DMA)
    wire [7:0] mem_address;
    wire [31:0] mem_data;
    wire mem_write_enable;
    wire mem_read_enable;
    wire cpu_has_bus;

    // Peripheral Bus (I/O Module <-> DMA)
    wire [31:0] io_data;
    wire new_io_data_ready;

```

```
    CPU cpu(
        .clk(clk),
        .reset(reset),
        .get_bus(cpu_has_bus),
        .mem_address(mem_address),
        .mem_data(mem_data),
        .mem_write_enable(mem_write_enable),
        .mem_read_enable(mem_read_enable),
        .opcode(opcode),
        .input_address(input_address),
        .register(cpu_register)
    );

    Memory memory(
        .clk(clk),
        .reset(reset),
        .address(mem_address),
        .data(mem_data),
        .write_enable(mem_write_enable),
        .read_enable(mem_read_enable)
    );

    DMA dma(
        .clk(clk),
        .reset(reset),
        .io_data(io_data),
        .new_io_data_ready(new_io_data_ready),
        .mem_address(mem_address),
        .mem_data(mem_data),
        .mem_write_enable(mem_write_enable),
        .mem_read_enable(mem_read_enable),
        .cpu_has_bus(cpu_has_bus)
    );

    IO io(
        .clk(clk),
        .reset(reset),
        .new_data_ready(new_io_data_ready),
        .io_data(io_data)
    );
endmodule
```

Code Snippet 5: Top Module Verilog Code

3 Evaluation and Results

3.1 Testbench Module

To evaluate the implementation described in the previous section, a testbench is used to perform some `LOAD`, `STORE`, and `NOP` instructions on the system. This testbench is written in a way to test different scenarios. For example, scenarios when the data is ready in the DMA, but the CPU does not free the bus for a few clock cycles. Because the print statements are clear enough, and for the sake of brevity, the Verilog code of the testbench module is provided in the Appendix.

3.2 Results

Figure 2 illustrates the simulation output in Modelsim. As can be seen in the figure, the I/O data is correctly written in the location 0 of memory.

```
VSIM 112> run
# <<time 10>> RESET
# <<time 20>> Instruction: NOP
# <<time 30>> Instruction: NOP
# <<time 40>> Instruction: NOP
# <<time 50>> Instruction: NOP
# <<time 60>> Intruction: LOAD MEM[0]   Result: cpu_register=00000000000000000000000000000000
# <<time 70>> Intruction: STORE MEM[1]
# <<time 80>> Instruction: NOP
# <<time 90>> Instruction: NOP
# <<time 100>> Instruction: NOP
# <<time 110>> Instruction: NOP
# <<time 120>> Instruction: NOP
# <<time 130>> Instruction: NOP
# <<time 140>> Instruction: NOP
# <<time 150>> Instruction: NOP
# <<time 160>> Instruction: NOP
# <<time 170>> Instruction: NOP
# <<time 180>> Intruction: LOAD MEM[0]   Result: cpu_register=00000000000000000000000000000001
# <<time 190>> Intruction: STORE MEM[2]
# <<time 200>> Intruction: LOAD MEM[1]   Result: cpu_register=00000000000000000000000000000000
# <<time 210>> Intruction: LOAD MEM[2]   Result: cpu_register=00000000000000000000000000000001
```

Figure 2: Simulation Output.

Figure 3 illustrates the waveform of the simulation in Modelsim. It can be seen that the system bus signals `mem_address` and `mem_data` are correctly managed because of the proper use of tri-state buffers in the DMA and CPU modules to perform bus arbitration.

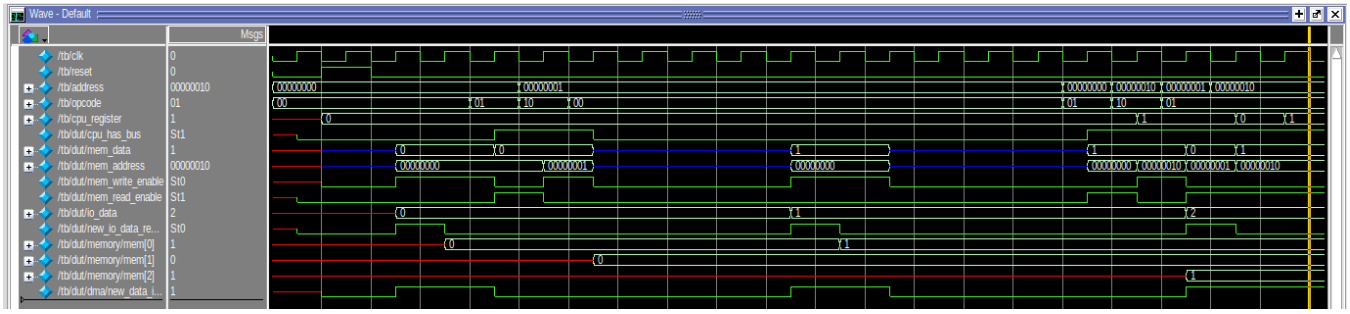

Figure 3: Waveform of Simulation.

Appendix

The Verilog code for the testbench module is provided in Code Snippet 6.

```
module tb;

reg clk, reset;
reg [7:0] address;
reg [1:0] opcode;
wire [31:0] cpu_register;

System dut(
  .clk(clk),
  .reset(reset),
  .opcode(opcode),
  .input_address(address),
  .cpu_register(cpu_register)
);

always #5 clk = ~clk;

initial begin

{clk, reset, address, opcode} = 0;
#10 reset = 1;
$display("<<time 10>> RESET");

#10 // start with NOP
reset = 0;
opcode = 2'b00;
$display("<<time 20>> Instruction: NOP");

#20 // LOAD CPU register with I/O data which is stored in 0x00 address
opcode = 2'b01;
address = 8'h00;
$display("<<time 30>> Instruction: NOP");
$display("<<time 40>> Instruction: NOP");

#10 $display("<<time 50>> Instruction: NOP");

// STORE CPU register to address 0x01
opcode = 2'b10;
address = 8'h01;

#10 $display("<<time 60>> Intruction: LOAD MEM[0] \t Result: cpu_register=%b", cpu_register);

// NOP for 10 cycles
opcode = 2'b00;

#10 $display("<<time 70>> Intruction: STORE MEM[1]");

#10 $display("<<time 80>> Instruction: NOP");
#10 $display("<<time 90>> Instruction: NOP");
#10 $display("<<time 100>> Instruction: NOP");
#10 $display("<<time 110>> Instruction: NOP");
#10 $display("<<time 120>> Instruction: NOP");
#10 $display("<<time 130>> Instruction: NOP");
```

```
#10 $display("<<time 140>> Instruction: NOP");
#10 $display("<<time 150>> Instruction: NOP");
#10 $display("<<time 160>> Instruction: NOP");


// LOAD CPU register with I/O data which is stored in 0x00 address
opcode = 2'b01;
address = 8'h00;

#10 $display("<<time 170>> Instruction: NOP");

// STORE CPU register to address 0x02
opcode = 2'b10;
address = 8'h02;

#10 $display("<<time 180>> Intruction: LOAD MEM[0] \t Result: cpu_register=%b", cpu_register);


// LOAD CPU register with Mem[1]
opcode = 2'b01;
address = 8'h01;
#10 $display("<<time 190>> Intruction: STORE MEM[2]");


// LOAD CPU register with Mem[2]
opcode = 2'b01;
address = 8'h02;
#10 $display("<<time 200>> Intruction: LOAD MEM[1] \t Result: cpu_register=%b", cpu_register);

#10 $display("<<time 210>> Intruction: LOAD MEM[2] \t Result: cpu_register=%b", cpu_register);
end

endmodule
```

Code Snippet 6: Testbench Module Verilog Code