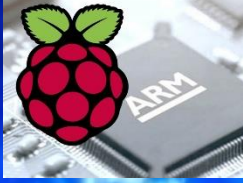


EEE3096S



Hardware Description Languages

a taste in 5-lectures Part4

Embedded Systems II

L40

Dr Simon Winberg



Electrical Engineering
University of Cape Town

Outline of Lecture

- Busses and endianness
- Bit-order and Bit-shifting in Verilog
- Blocking vs. non-blocking
- Arrays and Memory

Busses & endian

- Busses or bit signal vectors are specified as follows:
 - `reg [20:0] dataA; // little endian LSB in bit 0`
 - `reg [0:20] dataB; // big endian MSB in bit 0`

It doesn't really matter if you are using them just as busses, it is only relevant when applying operations such as add.

Question: Can you say `dataA <= dataB` without an error?

A: Yes, because they are the same size

More on busses and arrays...

Selecting a bit [] and bit sequences [:]

- Consider the following bus:

```
wire bus [7:0]; // 8 bit buss
```

- As you probably assumed select a bit with:

```
bus[n] : select bit number n (from 0)
```

- You can also select a sequence:

```
bus[4:6] - select 3 bits: bus[4],bus[5],bus[6]
```

```
bus[6:4] - select 3 bits: bus[6],bus[5],bus[4]
```

Concatenating bits { }

- The curly brackets { } are used for concatenating bits (the [] are used for selecting bits outside declarations)

- **Example:**

```
input [15:0] a;           // 16-bit input
```

```
output [16:0] res1;      // 17-bit output
```

```
assign res1 = { {a[15]}, {a[15:0]}}; // 17 bits
```

```
assign resbig = {{16{a[15]}}, {a[15:0]}};
```

Question: What do you think resbig might become?

A: The notation {4{X}} will duplicate X 4 times, i.e. XXXX, so in the above resbig will actually be a 32 bit quantity with the first 16 bites equal to a[15] followed by a.

Rearranging bits

- Now that you know that { } concatenates bits and [] selects bits from a bus you can easily write code to rearrange bits, for example ...
 - Send part of one big bus to smaller busses:
 `small1 = big[15:0];`
 `small2 = big[31:16];`
 - Combine smaller busses into a big bus:
 `big = {small2, small1};`

Named Port Bindings

- As you have seen in previous code snippets you can ...

Just list wires or other components in sequence to connect them to ports of a module instance

```
module add2 (  
    input [1:0] a, input [1:0] b,  
    output [2:0] sum);  
    assign sum = in0 + in1;  
endmodule
```

```
module toplevel ();  
    var [1:0] first, second;  
    var [2:0] result;  
    add2(first,second,result);  
endmodule
```

You can explicitly use port names when binding the instance (code example right)

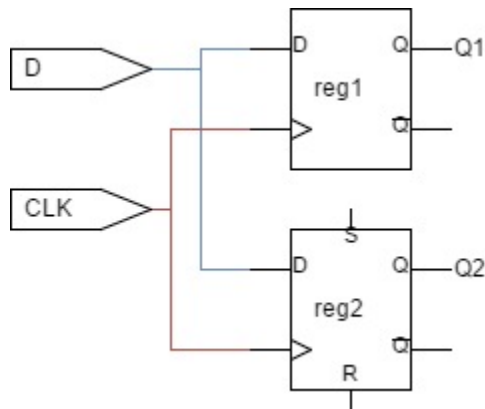
```
module add2 (  
    input [1:0] a, input [1:0] b,  
    output [2:0] sum);  
    assign sum = in0 + in1;  
endmodule
```

```
module toplevel ();  
    var [1:0] first, second;  
    var [2:0] result;  
    add2(.a(first),.b(second),.sum(result));  
endmodule
```

Blocking/Non-blocking statements

- Blocking

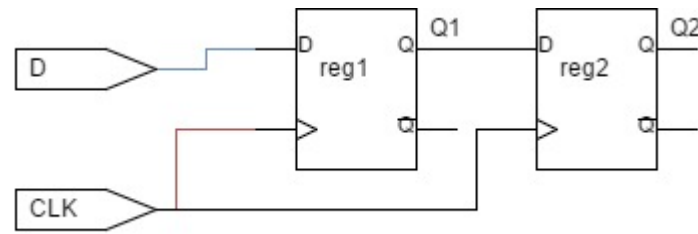
```
module blockFFs (  
    input D, input clk,  
    output reg Q1, output reg Q2)  
always @ (posedge clk)  
begin  
    Q1 = D;  
    Q2 = Q1;  
end  
endmodule
```



Note: can only have blocking in always section


- Non-blocking

```
module nonblockFFs (  
    input D, input clk,  
    output reg Q1, output reg Q2)  
always @ (posedge clk)  
begin  
    Q1 <= D;  
    Q2 <= Q1;  
end  
endmodule
```



always@ Example : D-type Flip Flop

```
module flipflop (din, clk, rst,q);  
  input din, clk, rst;  
  output q;  
  reg q; // q is a registered output  
  always @ (posedge clk) // whenever clk  
  begin  
    if (rst == 1) q = 0; // keep q low in reset  
    else q = din;  
  end  
endmodule
```



: the inline if

- The same C notation for using the : symbol for implementing an if as part of an expression can be used in Verilog

Example: these two modules do the same thing...

```
module mymod (sela,a,b,out);  
  input sela, a, b;  
  output out;  
  assign out = sela? a : b;  
endmodule
```

```
module mymod (sela,a,b,out);  
  input sela, a, b;  
  output out;  
  always @(*)  
  begin  
    if (sela)  
      assign out = a;  
    else  
      assign out = b;  
    end  
  end  
endmodule
```

Functions in Verilog

functions are not
modules in Verilog

- Functions can be used as **macros** within the body of a Verilog module.
 - These can effectively save typing.
 - They work differently to module instantiations.
 - These are defined inside a module.
 - Can only have input parameters

Example function declaration:

```
function [31:0] negate;  
  input [31:0] a;  
  negate = ~a;  
endfunction
```

Example using the function:

```
reg [15:0] a;  
wire [15:0] x;  
assign b = negate (a);  
initial begin  
  a=10;  
  a = add(1, a);  
  $display(" a=%b -a=%b", a, b);  
end
```

Example function: Converting endianness

- If need be you can construct a function to convert endianness, e.g.:

```
function [31:0] toBigEndian;  
    // transform data from little-endian to big-endian  
    input [31:0] x;  
    toBigEndian = {x[7:0], x[15:8], x[23:16], x[31:24]};  
endfunction
```

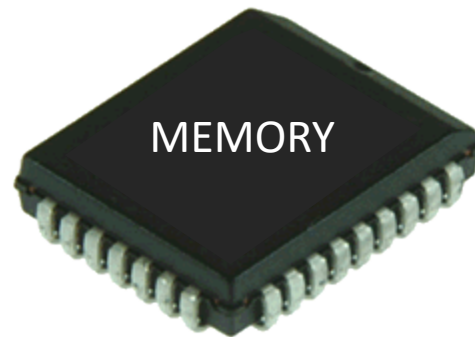

Vectors & Signal concatenation { }

Syntax for concatenating wires: **{ x1, x2, ... xn }** the collective can be used just the same as any other variable.

```
module adder4 (a, b, cin, sum, cout);  
    input [3:0] a, b;    // 2x 4 bit vector inputs  
    input cin;           // carry input  
    output [3:0] sum;    // 4-bit little endian vector  
    output cout;         // carry out  
    // perform the adder operation  
    assign {cout,sum} = a + b + cin;  
    // the leftmost is MSB since it is little endian  
endmodule
```

*i.e. if the input is little endian so is the output
If you said {sum,cout}=A*

Memory in HDL



Memories / Arrays of Busses

- C / C++ programmers are used to declaring arrays as:

datatype ary [size];

- You could do the same in Verilog and it would essentially be an array of bits or datatypes you are declaring, not a bus.

- Busses are declared as **wire [N:0] bus;**
note here that you need to choose to use little or big endian:
use **high_bit : low_bit** for little endian,
or **low_bit : high_bit** for big endian

Remember that busses don't store data, they are just wires, hence using a wire datatype. If you want instead to use registers that store the data written to them then use reg.

- **Example:**

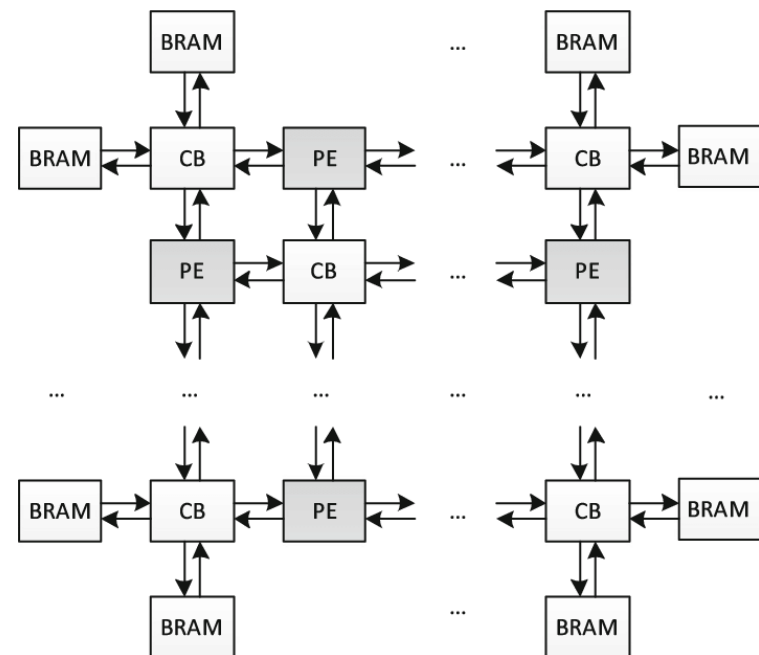
If I need to use 4 x 8-bit registers (i.e. each register storing 8 bits), I would declare them following reg as follows:

reg [7:0] numbers [3:0];

This structure essentially defines a memory, in this case 4 memory addresses of 8-bits.

BRAM Memory

- A FPGA generally has
 - **BRAM** – this is the most versatile, connects individual memory cells to your logic
 - It is often used for registers and caches, or small arrays to run DSP operations.
 - BRAM is the most flexible but tends to be rather limited (e.g. a few Kb to Mb)



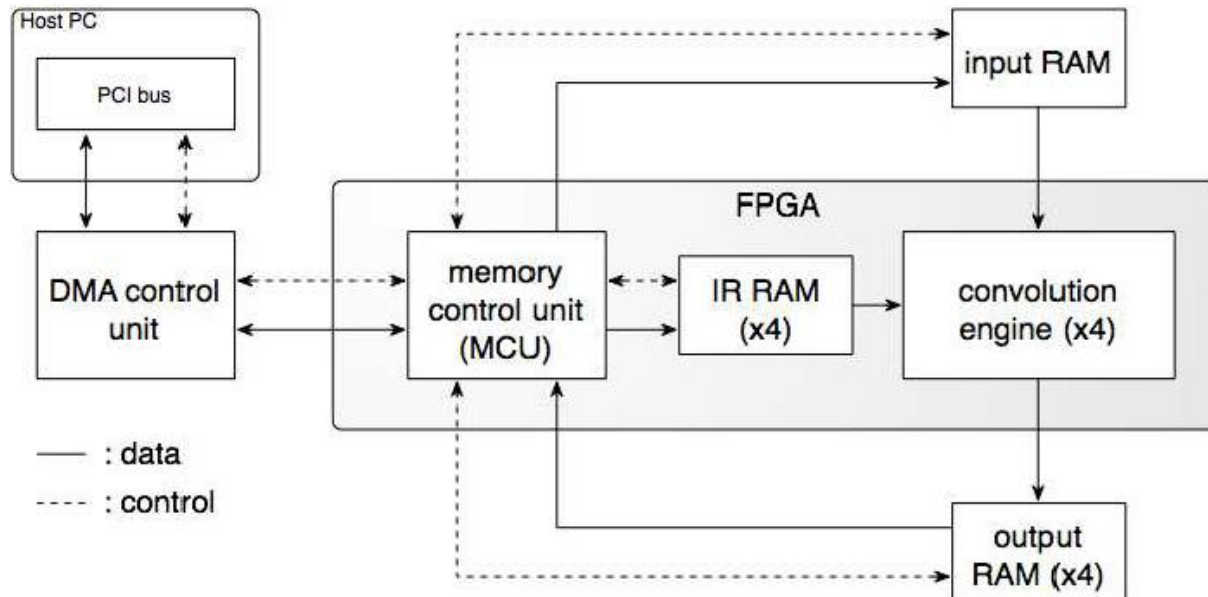
How BRAM fits in to an FPGA

SRAM, DRAM and others

- An FPGA may have SRAM integrated into the chip, this is fairly common nowadays (e.g. a few 100 MB). There may also be DRAM but usually on off chip, has to go through external address and data lines connected to FPGA.
- Usually accessing the SRAM needs to go through a memory control unit (MCU)

MCU – Connecting to memories

- The Memory Control Unit (MCU) is usually a module (typically combination logic on the FPGA) to integrates to external memory or shares memory among other modules (external hardware implementations may be used).
- A MCU is typically implemented as a state machine. Usually there is a cache or FIFO structure where you send address ranges to write or read and the MCU will send a signal to say when the requested operation has completed (if reading will put the data read into an output buffer, if writing will have send data in the input buffer to the specified addresses).



Where a MCU fits in to a example FPGA design

Single-port and Dual-port memories

- These can be defined in code. Often this would be done more to simulate the operation of a memory chip, or as a means for multiple modules to share the same memory

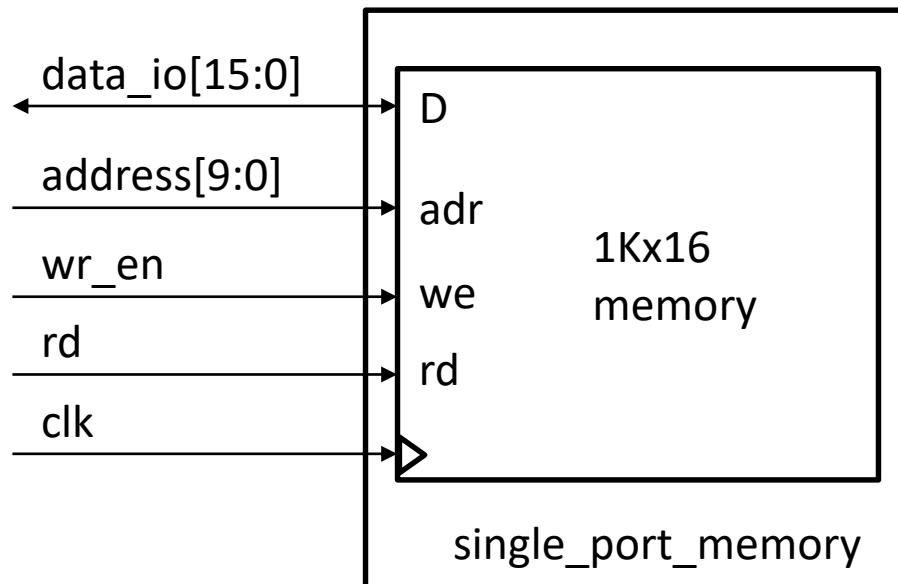
For time constraints in this course we will only consider the design of a single port memory...

.... and that prompts the next class activity!

Class Activity: HDL Memory

Verilog Activity 2:

For this class activity consider this memory component below...



Operation: As you know from previous lectures, with this type of memory design you can only read or write at one time. The design is such that it does not do anything unless the `wr_en` is set (causing data to be sent to memory) or `rd` is set and data is read from the given address. If both `we_en` and `rd` are low or they are both high then nothing happens.

Hint: a *inout* defines a tristate, can be either input or output.



Steps to undertake for class activity #2

- Design your module (you already know the ports but write out the interface)
- Understand how it is going to work
- Implement the one functionality to write (then maybe test it, e.g. dry run on paper)
- Implement the other functionality to read (then test it on paper)
- Consider handling exception/illegal states
- If there is time plan and develop test benches

Later, once you've had a change to try this on your own I'll release my sample solution and we will discuss it in next lecture

EEE3096S Verilog limit... towards EEE4120F

- I recommend thinking about how to implement a state machine in Verilog but we will probably not have time to cover this
- Next year the course EEE4120F “High Performance Digital Embedded Systems” will
 - Take these Verilog basics further to develop statemachines and signal processing chains / DFG structures
 - Introduce OpenCL and develop solutions for heterogeneous platforms
 - Cover high performance parallel embedded computer design and development

The Next Episode...

Lecture L41

- Memory solution
- Statemachine preview
- Summary of EEE3096S and study advice

