

Module Declaration

Verilog 95:

Verilog 95 is good to know about for historical reasons.

```
module first_module(A,B,C,D);
  input A,B,C;
  output D;

  ...
endmodule
```

Verilog-2001

Verilog has certain conveniences and is less error-prone.

```
module first_module(input A,
                    input B,
                    input C,
                    output D);

  ...
endmodule
```

Declaring module ports

<input|output|inout> Variable_declaration

inout: value can be read and written. Avoid them.

reg: for FF output

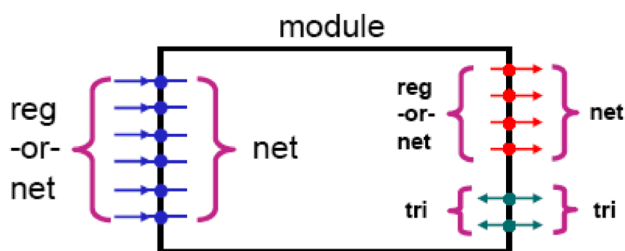
wire: implicit type of a signal. value of any input/output

```
module first_module(input A,
                    input B,
                    input C,
                    output reg D, // registered
                    output F,    // non registered
                    inout E);

  ...
endmodule
```

Port Connection Rules

Inputs are always of type net (superset of wire and tristate bus).



Inputs:

- Internally must be of net data type (e.g. wire)
- Externally the inputs may be connected to a reg or net data type

Outputs:

- Internally may be of net or reg data type
- Externally must be connected to a net data type

Inouts:

- Internally must be of net data type (tri recommended)
- Externally must be connected to a net data type (tri recommended)

Module Parameters

Parameters make modules more flexible and configurable. Parameters can be provided arguments. If they are not, default values are used in module instantiation.

[Example: how parameters are used]

At compile time, this module computes the logarithm at runtime.

- parameters precede the input/output list so they can define widths of inputs/outputs
- 4, 2, 7, and 64 specify default parameter values.
- parameters can be accessed outside the module
- localparams can only be accessed inside the module.

```
module memory_bank #(parameter Rd_Ports = 4, Wr_Ports = 2,
Size_Log = 7, Width = 64)
(
  input          clk          ,
  input [Size_Log*Wr_Ports-1:0] in_pos  ,
  input [Wr_Ports-1:0]         in_enable ,
  input [Wr_Ports*Width-1:0]    in_data  ,
  input [sclog2(Size)*Rd_Ports-1:0] out_pos ,
  // output
  output [Rd_Ports*Width-1:0]    out_data );

  localparam Size = 1<<Size_Log;
  //...
endmodule
```

[Example of Parameters]

Verilog-95 style. We need to know the order that i/o is defined in order to use it.

DO NOT USE IMPLICIT NAMES

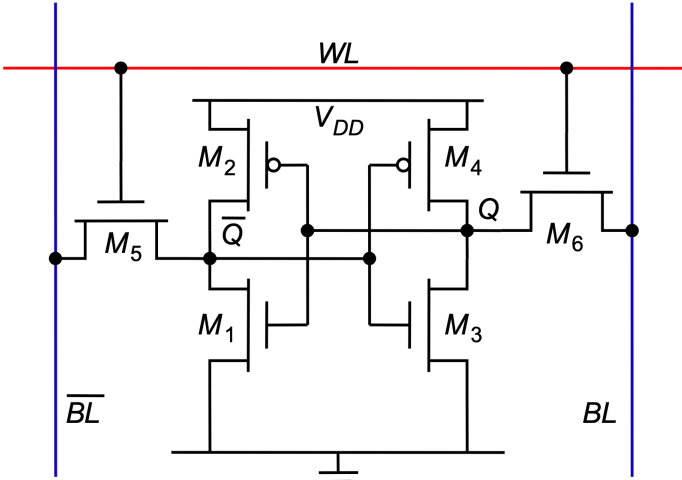
```
memory_bank #(1,1,8,64)  iname (clk, next_state_PC, ~next_state_N,
next_state, pc_tmp, out_data_variable);
```

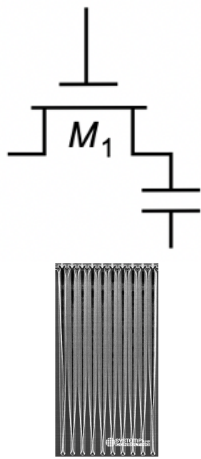
```
memory_bank  #(.Rd_Ports(1), .Wr_Ports(1), .Size_Log(8), .Width(64))
iname      (
  .clk      (clk),
  .in_pos   (next_state_PC),
  .in_enable (~next_state_N),
  .in_data  (next_state),
  .out_pos  (pc_tmp),
  .out_data (out_data_variable)
);
```

Memories

Professor Litz usually means SRAM when he talks about memory. Both SRAM and DRAM lose their data when power is cut.

Recap: How is memory built?

6-transistary CMOS SRAM Cell	
 <p>red line: represents the word line. SRAM cells are lined up on the word line.</p> <p>blue lines: bit line. In each word there are some number of bits (SRAM cells).</p>	<p>SRAM cells are the standard way to store data in FPGAs and ASICs. An SRAM cell has 6 transistors. 2 are used as read/write ports and 4 are used for a cross-coupled inverter.</p> <p>cross-coupled inverter: output of one is connected to input of the other. A bit can be stored in a stable way, for a virtually infinite amount of time.</p> <p>To invert the stored bit, both gates are opened, one is driven high and the other low. To read a bit, either M5 or M6 is opened.</p> <p>M5: read/write port M6: read/write port</p>

DRAM Cell	
<p>DRAM cells leverage a very different implementation than SRAM cells – DRAM is about 10x cheaper! DRAM cells are comprised of 1 transistor. They are not available in FPGA and ASIC designs.</p> <p>A bit's value is stored in a deep-trench capacitor. When the NMOS transistor receives a 1 it is open, and charge accumulates on the capacitor. When it is closed, the charge stays on the capacitor.</p> <p>caveat: reads are destructive. When a bit is read, electrons stored are lost. The old value must be re-written to the DRAM cell.</p>	 <p>deep trench capacitors are small from a 2-D perspective but are very deep.</p>

Memories I: Simplest Memory you can Build

```
reg [7:0] synRAM [1023:0];

always @(posedge clk) begin
    if (we) begin
        synRAM[addr] <= data_in;
        data_out <= 8b'x; // don't care
    end else
        data_out <= synRAM[addr];
end

end
```

Define 2D array where there are some number of n-bit wide # elements

- **reg [7:0]** is for the width of each element
- **synRAM [1023:0]** is # of elements in the memory block

A synchronous always block is triggered by the positive clock edge. If the write enable signal **we** is high, a memory element is written to. If it is low, the memory element is read from it.

- `synRAM[addr] <= data_in` to write
 - `data_out <= synRAM[addr]` to read
- `data_in` and `data_out` must also be 8 bits.

The model above prototypes a memory block in Verilog. It is synthesized as a memory block from a simulation perspective.

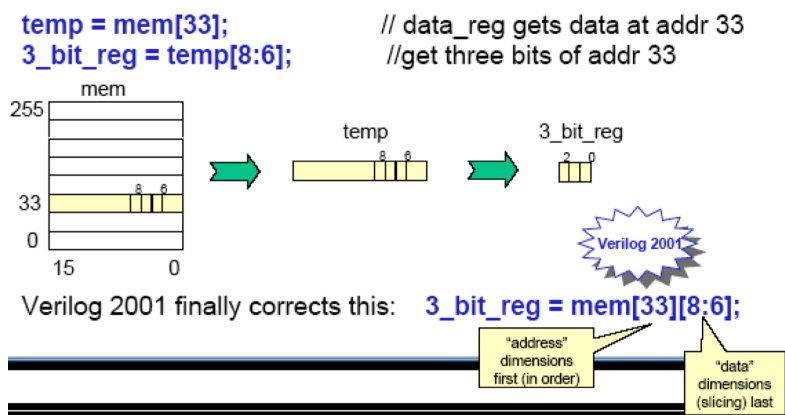
The **\$readmemh** command initializes the `synRAM` data block with a set of given data values in the `memory.list` file. The data in the file must be formatted properly. Not synthesizable.

```
$readmemh("memory.list", synRAM);
```

- Memory synthesized for an FPGA is undefined at beginning of runtime. An FSM would need to generate writes to each address to put values in memory.

Memories II: Accessing Memories

In this example we have 256 elements that are 16 bits wide. Memory can be indexed to get 16-bit values. Then individual bits can be selected from that. Verilog-2001 allows this to be done in 1 line of code.



Memories III: Make Memory Configurable

Parameters are used in this SRAM module to make them **configurable**. Note WORDS, ABITS, DBITS, and DELAY. They have default values and can be used to customize the size/specification of a memory block.

```
module sram(clk,addr,oe_n,we_n,cs_n,din,dout);
    parameter      WORDS = 256;
    parameter      ABITS=8;
    parameter      DBITS = 32;
    localparam     DELAY=25;

    input          clk;
    input [ABITS-1:0] addr;
    input          we_n;
    input          cs_n;
    input          oe_n;
    input [DBITS-1:0] din;
    output reg [DBITS-1:0] dout;
```

[Example: synchronous sram.v]

chip select cs_n enables the read and write path of the memory module.

write enable we_n enables writing to a memory block.

out enable oe_n enables reading from a memory block.

```
reg [DBITS-1:0] mem[0:WORDS-1];
reg [ABITS-1:0] addr_reg;

integer i,mcd;

always @(posedge clk)
begin
    addr_reg = addr;
    if (!cs_n) begin
        if (!we_n) begin
            mem[addr_reg] <= din;
            dout <= #(DELAY) 'bx;
            $display("%d Write A=%h D=%h", $realtime, addr_reg, din);
        end else if (!oe_n) begin
            dout <= #(DELAY) mem[addr_reg];
            $display("%d Read A=%h D=%h", $realtime, addr_reg, mem[addr_reg]);
        end
    end
end
```

"Synthesizing" a memory

So how is this all actually synthesized?

main solution: instantiate a specific SRAM module from the Xilinx library (FPGAs) or third-party memory library (ASICs).

- **If you synthesize a behavioral SRAM model you get many, many flip-flops.**
- **The memory model you use for simulation is often different than what you will actually use for the hardware. It is crucial that simulation model behaves like the synthesized model.**
 - in Verilog, comb logic is used to drive sim and synthesis
 - <3 a beauty of Verilog

Turning "off" synthesis

Xilinx and Altera both have custom RAM devices. The pragmas `translate_off` and `translate_on` are used to indicate parts of code that are not to be synthesized.

```
// synthesis translate_off
`define sim_only
// synthesis translate_on

`ifdef sim_only
// Simulation model
`else
// Synthesis instance
`endif
```

FPGA Memories

for ASICs: 3rd party IP

for FPGAS: 3 options

1. use code you wrote to describe a memory module and synthesize it to generate a large sea of flip-flops or
2. make use of block RAMs
3. make use of distributed RAM

2 and 3 are custom hardware blocks/explicit structures that are a better option than 1.

There are custom IP blocks in FPGAs. Synthesis tools either directly infer that memory blocks should be synthesized (strict coding style) or manual instantiation is leveraged.

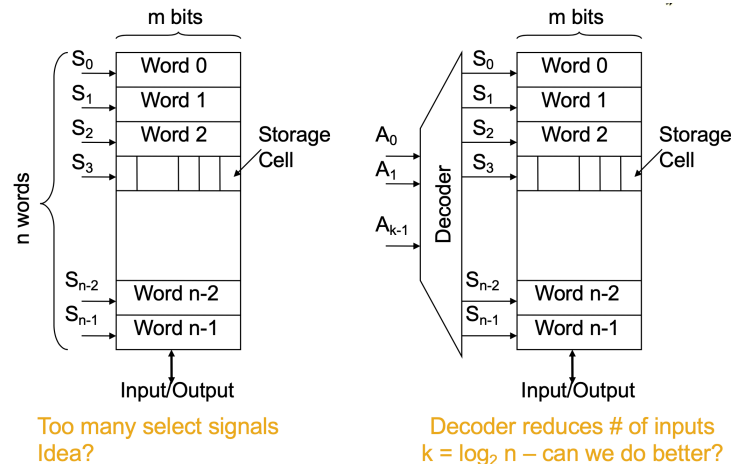
How can we make sure when generating a bit file for a Verilog design the FPGA will do it?
either

1. synthesis tool is smart enough to infer
2. manual instantiation (safe side)

1-D Memory Architecture

A 1-D memory architecture is structured as follows: a row-column approach

- each row is m bits wide and is referred to as a word
- there are n words in the memory block.
- 1 input/output wire is used per word to increase memory density.
- the capacity of the SRAM block can be scaled independently of i/o signals.



- The # of select times scales with # of words in design to the left (one-hot select signal)
- logarithmic address $\log_2(\# \text{ words})$ to the right (binary encoded select signal) for a more compact design.

2-D (Array) Memory Architecture

A 2-D memory architecture is structured as follows:

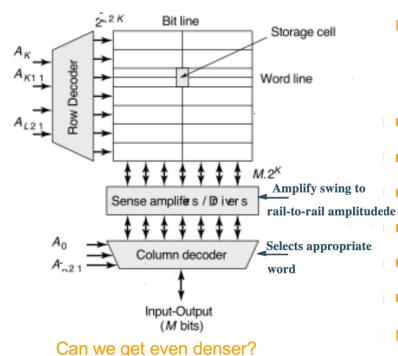
- each row is multiple words wide.
- there are n rows, fewer rows than the 1-D memory architecture

row decoder: used to select a row from the block.

column decoder: used to select a word from a row,

word line drivers:

sense amplifiers: strengthen signal from input/output data. Increase storage density by allowing DRAM cells to be as small as possible (store as few electrons as possible). They make tiny amounts of electrons readable.



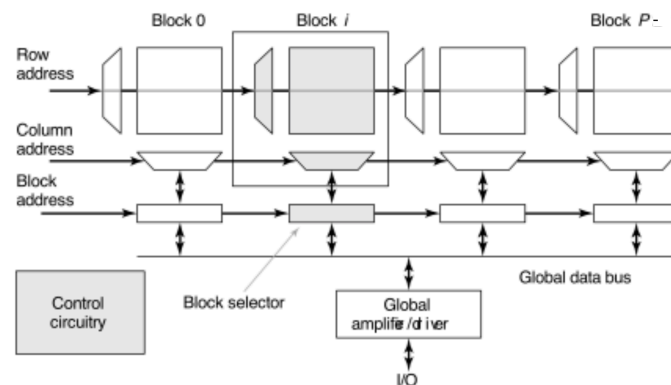
- Total memory access time consists of many components
- Row decode
- Word line drivers
- Cell driving bit line cap
- Sense amp delay
- Column decode
- Output driving circuitry
- Which dominate?

Less flexibility. Individual bits cannot be written. Efficient when using multiple bits from a row. Increases memory density.

3-D Memory Architecture

A 3-D memory architecture is structured as follows: have rows of 2-D memory blocks.

- each row address has multiple blocks of memory (each block has its own row/column decoder)
- scale row address to #rows/# blocks. Each block uses the same row address. Each block uses the same column address. s
- reduce flexibility, reduce bandwidth to get higher density



Advantages:

1. Shorter wires within blocks
2. Block address activates only 1 block => power savings

Very large blocks can result in low clocking frequency. Larger density memory blocks are often multi-cycle to access