Memory in SystemVerilog

Prof. Stephen A. Edwards

Columbia University

Spring 2015
Implementing Memory
Memory = Storage Element Array + Addressing

Bits are expensive
They should dumb, cheap, small, and tightly packed

Bits are numerous
Can’t just connect a long wire to each one
Williams Tube

CRT-based random access memory, 1946. Used on the Manchester Mark I. 2048 bits.
Mercury acoustic delay line

Used in the EDASC, 1947.

32 × 17 bits
Selectron Tube

RCA, 1948.

2 × 128 bits

Four-dimensional addressing

A four-input AND gate at each bit for selection
Magnetic Core

IBM, 1952.
Magnetic Drum Memory

1950s & 60s. Secondary storage.
# Modern Memory Choices

<table>
<thead>
<tr>
<th>Family</th>
<th>Programmed</th>
<th>Persistence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask ROM</td>
<td>at fabrication</td>
<td>∞</td>
</tr>
<tr>
<td>PROM</td>
<td>once</td>
<td>∞</td>
</tr>
<tr>
<td>EPROM</td>
<td>1000s, UV</td>
<td>10 years</td>
</tr>
<tr>
<td>FLASH</td>
<td>1000s, block</td>
<td>10 years</td>
</tr>
<tr>
<td>EEPROM</td>
<td>1000s, byte</td>
<td>10 years</td>
</tr>
<tr>
<td>NVRAM</td>
<td>∞</td>
<td>5 years</td>
</tr>
<tr>
<td>SRAM</td>
<td>∞</td>
<td>while powered</td>
</tr>
<tr>
<td>DRAM</td>
<td>∞</td>
<td>64 ms</td>
</tr>
</tbody>
</table>
Implementing ROMs

Add. Data

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>10</td>
<td>00</td>
</tr>
</tbody>
</table>

Z: “not connected”

2-to-4 Decoder

Wordline 0

Wordline 1

Wordline 2

Wordline 3

Bitline 0

Bitline 1

Bitline 2

Bitline 3

$A_0 \quad A_1$
Implementing ROMs

Z: “not connected”

Add. Data

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>00</td>
</tr>
</tbody>
</table>

2-to-4 Decoder

Bitline 0
Wordline 0

Bitline 1
Wordline 1

Bitline 2
Wordline 2

Bitline 3
Wordline 3

D0 0
D1 0
D2 1

Implementing ROMs

- Z: “not connected”

Add. Data

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2-to-4 Decoder

\[
\begin{array}{cccc}
A_1 & A_0 & D_2 & D_1 & D_0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 \\
\end{array}
\]
Implementing ROMs

Z: “not connected”

<table>
<thead>
<tr>
<th>Add. Data</th>
<th>00</th>
<th>011</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>010</td>
</tr>
</tbody>
</table>
Mask ROM Die Photo
A Floating Gate MOSFET

Cross section of a NOR FLASH transistor. Kawai et al., ISSCC 2008 (Renesas)
Floating Gate n-channel MOSFET

Floating gate uncharged; Control gate at 0V: Off
Floating Gate n-channel MOSFET

Floating gate uncharged; Control gate positive: On
Floating Gate n-channel MOSFET

Floating gate negative; Control gate at 0V: Off
Floating Gate n-channel MOSFET

Floating gate negative; Control gate positive: Off
EPROMs and FLASH use Floating-Gate MOSFETs
Static Random-Access Memory Cell

[Diagram of a static random-access memory cell with bit lines and a word line.]
Layout of a 6T SRAM Cell

Weste and Harris. *Introduction to CMOS VLSI Design*. Addison-Wesley, 2010.
Intel’s 2102 SRAM, 1024 × 1 bit, 1972
2102 Block Diagram
6264 SRAM Block Diagram

- **Inputs:**
  - CE1, CE2 (Chip Enable)
  - WE, OE (Write Enable, Output Enable)
  - A0, A1, A2, A3, A4, A5, A6, A7, A8 (Address Inputs)

- **Outputs:**
  - I/O0, I/O1, I/O2, I/O3, I/O4, I/O5, I/O6, I/O7 (Input/Output)

- **Components:**
  - Input Buffer
  - Column Decoder
  - Row Decoder
  - Sense Amps
  - 256 x 32 x 8 Array
  - Power Down
Toshiba TC55V16256J 256K × 16
Dynamic RAM Cell
Ancient (c. 1982) DRAM: 4164 64K × 1
Basic DRAM read and write cycles
Page Mode DRAM read cycle

RAS

CAS

Addr

WE

Din

Dout

read

read

read
Samsung 8M × 16 SDRAM

- BA1
- BA0
- A11
- A10
- A2
- A1
- A0
- UDQM
- LDQM
- DQ15
- DQ14
- DQ1
- DQ0
- WE
- CAS
- RAS
- CS
- CKE
- CLK

Bank Select
Data Input Register
8M x 4 / 4M x 8 / 2M x 16
8M x 4 / 4M x 8 / 2M x 16
8M x 4 / 4M x 8 / 2M x 16
Sense AMP
Output Buffer I/O Control
Column Decoder
Latency & Burst Length
Programming Register
Timing Register

I/O Control
LWE
LDQM
Output Buffer
DQi

Address Register
Row Buffer
Refresh Counter
Row Decoder
Col. Buffer

Bank Select
Data Input Register
8M x 4 / 4M x 8 / 2M x 16
8M x 4 / 4M x 8 / 2M x 16
8M x 4 / 4M x 8 / 2M x 16
Sense AMP
Output Buffer I/O Control
Column Decoder
Latency & Burst Length
Programming Register
Timing Register

Timing Register
CLK CKE CS RAS CAS WE L(U)DQM
# SDRAM: Control Signals

<table>
<thead>
<tr>
<th>RAS</th>
<th>CAS</th>
<th>WE</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>NOP</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Load mode register</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Active (select row)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Read (select column, start burst)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Write (select column, start burst)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Terminate Burst</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Precharge (deselect row)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Auto Refresh</td>
</tr>
</tbody>
</table>

Mode register: selects 1/2/4/8-word bursts, CAS latency, burst on write
SDRAM: Timing with 2-word bursts

The diagram illustrates the timing sequence for SDRAM operations with 2-word bursts. The signals include:

- **Clk**: Clock signal.
- **RAS**: Row Address Strobe.
- **CAS**: Column Address Strobe.
- **WE**: Write Enable.

The diagram shows the timing for:

- **Load**: The sequence of operations starting with the load operation.
- **Active**: Transition to active mode.
- **Write**: Data write operations.
- **Read**: Data read operations.
- **Refresh**: Refresh operations.

The diagram also highlights the waveforms for **Addr** (Address) and **BA** (Bank Address) along with the data signals **DQ** (Data Out).
Using Memory in SystemVerilog
Basic Memory Model
**Basic Memory Model**

![Diagram of a basic memory model](image)

- **Address** → Memory
- **Data In** → Memory
- **Write** → Memory
- **Clock** → Memory
- **Data Out** → Memory

Clock

Address: A0 A1

Data In: D1

Write: 

Data Out: D0 old D1
Basic Memory Model

Clock

Address

Data In

Write

Data Out

Memory

A0

A1

A1

D1

Data Out

D0

old D1

D1

Read A1
Memory Is Fundamentally a Bottleneck

Plenty of bits, but

You can only see a small window each clock cycle

Using memory = scheduling memory accesses

Software hides this from you: sequential programs naturally schedule accesses

You must schedule memory accesses in a hardware design
Modeling Synchronous Memory in SystemVerilog

```verilog
module memory(
    input logic clk ,
    input logic write ,
    input logic [3:0] address ,
    input logic [7:0] data_in ,
    output logic [7:0] data_out);

logic [7:0] mem [15:0];

always_ff @(posedge clk)
begin
    if (write)
        mem[address] <= data_in;
    data_out <= mem[address];
end
endmodule
```

- **Write enable**
- **4-bit address**
- **8-bit input bus**
- **8-bit output bus**
- **The memory array: 16 8-bit bytes**
- **Clocked**
- **Write to array when asked**
- **Always read (old) value from array**
M10K Blocks in the Cyclone V

10 kilobits (10240 bits) per block
Dual ported: two addresses, write enable signals
Data busses can be 1–20 bits wide
Our Cyclone V 5CSXFC6 has 557 of these blocks (696 KB)
Memory in Quartus: the Megafunction Wizard

Which megafuction would you like to customize?
Select a megafunction from the list below

- DSP
- Gates
- I/O
- Interfaces
- JTAG-accessible Extensions
- Memory Compiler
  - ALTOTP
  - ALTUFM_I2C
  - ALTUFM_NONE
  - ALTUFM_PARALLEL
  - ALTUFM_SPI
  - FIFO
  - LPM_SHIFTREG
  - RAM initializer
  - RAM: 1-PORT
  - RAM: 2-PORT
- PLL

Which device family will you be using? Cyclone V

Which type of output file do you want to create?
- AHDL
- VHDL
- Verilog HDL

What name do you want for the output file?
/home/sedwards/svn/classes/2014/4840/dummy/memory

Output files will be generated using the classic file structure

Return to this page for another create operation

Note: To compile a project successfully in the Quartus II software, your design files must be in the project directory, in a library specified in the Libraries page of the Options dialog box (Tools menu), or a library specified in the Libraries page of the Settings dialog box (Assignments menu).

Your current user library directories are:
Memory: Single- or Dual-Ported

RAM: 2-PORT

Parameter Settings: General, Widths/Blk Type, Ckls/Rd, Byte En, Regs/Ckns/Aclrs, Output1, Output2, Mem Init

Currently selected device family: Cyclone V

How will you be using the dual port RAM?
- With one read port and one write port
- With two read/write ports

How do you want to specify the memory size?
- As a number of words
- As a number of bits
Memory: Select Port Widths

- **Memory**: 8192 bits
- **Data Widths**
  - Use different data widths on different ports
  - q_a[0] width: 1 bit
  - q_b[15..0] width: 16 bits
- **Block Type**: M10K
- **Options**: Set the maximum block depth to **Auto** for words.
Memory: One or Two Clocks
Memory: Output Ports Need Not Be Registered

Which ports should be registered?

- Write input ports
  - `data_a`, `wraddress_a`, and `wren_a`
- Read input ports
  - `rdaddress` and `rden`
- Read output port(s)
  - `q_a` and `q_b`

- Create one clock enable signal for each clock signal
- Use different clock enables for registers
- Create an 'aclr' asynchronous clear for the registered ports
Memory: Wizard-Generated Verilog Module

This generates the following SystemVerilog module:

```systemverilog
module memory ( // Port A:
    input logic [12:0] address_a, // 8192 1-bit words
    input logic clock_a,
    input logic [0:0] data_a,
    input logic wren_a, // Write enable
    output logic [0:0] q_a,

    // Port B:
    input logic [8:0] address_b, // 512 16-bit words
    input logic clock_b,
    input logic [15:0] data_b,
    input logic wren_b, // Write enable
    output logic [15:0] q_b);
```

Instantiate like any module; Quartus treats specially
Two Ways to Ask for Memory

1. Use the Megafuction Wizard
   + Warns you in advance about resource usage
     – Awkward to change

2. Let Quartus infer memory from your code
   + Better integrated with your code
     – Easy to inadvertantly ask for garbage
module twoport(
    input logic clk,
    input logic [8:0] aa, ab,
    input logic [19:0] da, db,
    input logic wa, wb,
    output logic [19:0] qa, qb);

logic [19:0] mem [511:0];

always_ff @(posedge clk) begin
    if (wa) mem[aa] <= da;
    qa <= mem[aa];
    if (wb) mem[ab] <= db;
    qb <= mem[ab];
end
endmodule

Failure: Exploded!
Synthesized to an 854-page schematic with 10280 registers (no M10K blocks)
Page 1 looked like this:
module twoport2(
    input logic clk,
    input logic [8:0] aa, ab,
    input logic [19:0] da, db,
    input logic wa, wb,
    output logic [19:0] qa, qb);

logic [19:0] mem [511:0];

always_ff @(posedge clk) begin
    if (wa) mem[aa] <= da;
    qa <= mem[aa];
end

always_ff @(posedge clk) begin
    if (wb) mem[ab] <= db;
    qb <= mem[ab];
end
endmodule

Failure

Still didn’t work:

RAM logic “mem” is uninferred due to unsupported read-during-write behavior
module twoport3(
    input logic clk,
    input logic [8:0] aa, ab,
    input logic [19:0] da, db,
    input logic wa, wb,
    output logic [19:0] qa, qb);

logic [19:0] mem [511:0];

always_ff @(posedge clk) begin
    if (wa) begin
        mem[aa] <= da;
        qa <= da;
    end else qa <= mem[aa];
end

always_ff @(posedge clk) begin
    if (wb) begin
        mem[ab] <= db;
        qb <= db;
    end else qb <= mem[ab];
end
endmodule

Finally!

Took this structure from a template: Edit → Insert Template → Verilog HDL → Full Designs → RAMs and ROMs → True Dual-Port RAM (single clock)
The Perils of Memory Inference

module twoport4(
    input logic clk,
    input logic [8:0] ra, wa,
    input logic write,
    input logic [19:0] d,
    output logic [19:0] q);

logic [19:0] mem [511:0];

always_ff @(posedge clk) begin
    if (write) mem[wa] <= d;
    q <= mem[ra];
end
endmodule

Also works: separate read and write addresses

Conclusion:

Inference is fine for single port or one read and one write port.

Use the Megafunction Wizard for anything else.