Memory in SystemVerilog

Prof. Stephen A. Edwards

Columbia University

Spring 2016
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>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>011</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td></td>
</tr>
</tbody>
</table>

Z: “not connected”
Implementing ROMs

Add. Data

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Z: "not connected"
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>011</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td></td>
</tr>
</tbody>
</table>
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>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>00</td>
<td>01</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 labeled bit lines and 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
SRAM Timing

A12
A11
  .
  .
A2  6264
A1  8K × 8
A0  SRAM
CS1
CS2
WE
OE
Addr
Data

write 1
read 2
6264 SRAM Block Diagram
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
## 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

Load  Active  Write  Read  Refresh

Clk
RAS
CAS
WE
Addr
BA
DQ
Using Memory in SystemVerilog
Basic Memory Model

![Memory Diagram]

- **Address**: A0
- **Data In**: D1 (old D1)
- **Write**: Read A0
- **Clock**:
- **Data Out**: D0

Legend:
- **Address**: A1
- **Data In**: D1
- **Write**: Read A0
- **Data Out**: D0
Basic Memory Model

Clock
Address A0 A1 Write A1
Data In D1
Write
Data Out D0 old D1
Basic Memory Model

<table>
<thead>
<tr>
<th>Address</th>
<th>Data In</th>
<th>Write</th>
<th>Clock</th>
<th>Data Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>D1</td>
<td></td>
<td></td>
<td>D0</td>
</tr>
<tr>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td>old D1</td>
</tr>
<tr>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td>D1</td>
</tr>
</tbody>
</table>

Clock: \( \uparrow \downarrow \uparrow \downarrow \)  
Address: \( A_0 \ A_1 \ A_1 \)  
Read A1
Data In: \( \boxed{D_1} \)  
Write
Data Out: \( D_0 \ old \ D_1 \ D_1 \)
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 megafuction from the list below

- DSP
- Gates
- I/O
- Interfaces
- JTAG-accessible Extensions
- Memory Compiler
  - ALTOTP
  - ALTUFM_J2C
  - ALTUFM_NONE
  - ALTUFM_PARALLEL
  - ALTUFM_SPI
  - FIFO
  - LPM_SHIFTREG
  - RAM initializer
  - RAM: 1-PORT
  - RAM: 2-PORT
  - ROM: 1-PORT
  - ROM: 2-PORT
  - Shift register (RAM-based)
- 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
Memory: Select Port Widths

RAM: 2-PORT

Parameter Settings

General

Widths/Blk Type

Clks/Rd, Byte En

Regs/Cikens/Aclrs

Output1

Output2

Mem Init

How many bits of memory?

- Use different data widths on different ports

Read/Write Ports

- How wide should the 'q_a' output bus be?
- How wide should the 'data_a' input bus be?
- How wide should the 'q_b' output bus be?

Note: You could enter arbitrary values for width and depth

What should the memory block type be?

- Auto
- MLAB
- M10K
- M144K
- LCs

Set the maximum block depth to Auto words
Memory: One or Two Clocks
Memory: Output Ports Need Not Be Registered
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 Megafunction 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 inadvertently 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
The Perils of Memory Inference

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.