Memory in SystemVerilog

Prof. Stephen A. Edwards

Columbia University

Spring 2020
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
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

Z: “not connected”

2-to-4 Decoder

Add. Data

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Wordline 0

Wordline 1

Wordline 2

Wordline 3

Bitline 0

Bitline 1

Bitline 2

D₀  D₁  D₂
Implementing ROMs

Z: “not connected”

Add. Data

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

2-to-4 Decoder

Bitline 2 Bitline 1 Bitline 0

Wordline 0

Wordline 1

Wordline 2

Wordline 3

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”

2-to-4 Decoder

Add. Data

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</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

- **0/1**
- **Z: “not connected”**

<table>
<thead>
<tr>
<th>Add. Data</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>011</td>
<td>110</td>
<td>100</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 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
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
SRAM Timing

![SRAM Timing Diagram]

**Diagram Details:**
- **Inputs:**
  - A12, A11, ..., A0
  - CS1, CS2, WE, OE
- **Outputs:**
  - D7, D6, ..., D0
- **Description:**
  - The diagram illustrates the timing of a 6264 8K x 8 SRAM.
  - Key signals include CS1, CS2, WE, and OE.
  - Address and data signals for write 1 and read 2 operations.
6264 SRAM Block Diagram
Toshiba TC55V16256J 256K × 16
Dynamic RAM Cell

Column

Row
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
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
Using Memory in SystemVerilog
Synchronous SRAM
Synchronous SRAM

Clock

Address

Data In

Write

Data Out

Memory

A0 A1

Write A1

D1

D0 old D1
Synchronous SRAM

Clock

Address

Data In

Write

Data Out

Memory

A0 A1 A1

D1

Read A1

D0 old D1 D1
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
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
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 5CSEMA5 has 397 of these blocks = 496 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_I2C
  - 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/Clocks/Aclrs**
  - **Output1**
  - **Output2**
  - **Mem Init**

**Diagram: Memory Block**

- **Inputs:**
  - `data_a[0]`
  - `address_a[12..0]`
  - `wren_a`
  - `data_b[15..0]`
  - `address_b[8..0]`
  - `wren_b`
  - `clock`

- **Outputs:**
  - `q_a[0]`
  - `q_b[15..0]`

**Questions:**

- **How many bits of memory?**
  - 8192

- **Use different data widths on different ports**

- **Read/Write Ports:**
  - **How wide should the 'q_a' output bus be?**
    - 1
  - **How wide should the 'data_a' input bus be?**
    - 1
  - **How wide should the 'q_b' output bus be?**
    - 16

**Note:** You can enter arbitrary values for width and depth.

**What should the memory block type be?**

- Auto
- MLAB
- M10K
- M144K
- LCs

**Options...**

**Set the maximum block depth to Auto**

**words**
Memory: One or Two Clocks

What clocking method do you want to use?

- Single clock
- Dual clock: use separate 'read' and 'write' clocks
- Dual clock: use separate 'input' and 'output' clocks
- No clock (fully asynchronous)
- Dual clock: use separate clocks for A and B ports

Create 'rden_a' and 'rden_b' read enable signals

Byte Enable Ports

- Create byte enable for port A
- Create byte enable for port B

What is the width of a byte for byte enables? 8 bits

Enable error checking and correcting (ECC) to check and correct single bit errors and detect double errors
Memory: Output Ports Need Not Be Registered
Memory: Wizard-Generated Verilog Module

This generates the following SystemVerilog module:

```verilog
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:
The Perils of Memory Inference

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.