Memory in SystemVerilog

Prof. Stephen A. Edwards

Columbia University

Spring 2022
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.
<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 erase</td>
<td>10 years</td>
</tr>
<tr>
<td>FLASH</td>
<td>1000s, block erase</td>
<td>10 years</td>
</tr>
<tr>
<td>EEPROM</td>
<td>1000s, byte erase</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>01</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>01</td>
</tr>
</tbody>
</table>

Z: “not connected”

2-to-4 Decoder

Wordline 0

Wordline 1

Wordline 2

Wordline 3
Implementing ROMs

A 2-to-4 decoder is used to select one of four outputs based on the input addresses. The address lines $A_1$ and $A_0$ are used to select one of the four wordlines, and the output lines are connected to the bitlines.

The add. data table shows the output values for each input combination:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>011</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
</tr>
</tbody>
</table>
Implementing ROMs

Z: “not connected”

Add. Data

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A0</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

2-to-4 Decoder

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

D1, D2, D0
Implementing ROMs

<table>
<thead>
<tr>
<th>Add. Data</th>
<th>00</th>
<th>011</th>
</tr>
</thead>
<tbody>
<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”

2-to-4 Decoder

D0 0
D1 0
D2 1
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
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

A12
A11
A2  6264
A1  8K × 8
A0  SRAM
D7
D6
D1
D0
CS1
CS2
WE
OE
Addr
Data

write 1
read 2
6264 SRAM Block Diagram
Toshiba TC55V16256J 256K × 16

Memory Cell Array
1,024 × 256 × 16
(4,194,304)

Data Input Buffer
Row Decoder
Sense Amplifier
Column Decoder
Clock Generator
Address Buffer

VDD
GND
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

Clk
RAS
CAS
WE
Addr
BA
DQ

Load  Active  Write  Read  Refresh

Op  R  C  C  

B  B  B

W  W

R  R
Using Memory in SystemVerilog
Synchronous SRAM

Clock

Address A0 Read A0

Data In

Write

Data Out D0

Memory

Address

Data In

Write

Clock

Data Out
Synchronous SRAM

Clock
Address
Data In
Write
Data Out

Memory

Clock
Address
Data In
Write
Data Out

Write A1
D1
old D1
Synchronous SRAM

*Clock*  
*Address*  
*Data In*  
*Write*  
*Data Out*  

*Memory*

- **Address**: A0, A1, A1
- **Data In**: D1
- **Write**: old D1, D1
- **Data Out**: D0, old D1, D1

Read A1
Memory: A Fundamental 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
In hardware, you must schedule memory accesses
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 per block  Dual ported: two addresses, write enable signals
Data busses can be 1–20 bits wide  Our Cyclone 5CSEMA5 has 397 = 496 KB
Memory in Quartus: the Megafuction 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?
- ALCHDL
- VHDL
- Verilog HDL

What name do you want for the output file?

/output/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 in 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

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

[Diagram of 2-PORT RAM memory with parameters and options for port widths, data_a[12..0] and data_b[15..0], wren_a and wren_b, clock, q_a[0] and q_b[15..0].]

- **Block Type**: M10K

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

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

**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 could enter arbitrary values for width and depth

**What should the memory block type be?**
- M10K
- Auto
- MLAB
- M144K
- LCs

**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:

```
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,

  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 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
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)
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

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

Use the Megafunction Wizard for anything else.