Designing with Memory

Prof. Stephen A. Edwards

Columbia University

Spring 2014
Using Memory
Basic Memory Model

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

Clock
Address A0
Data In
Write
Data Out D0

Read A0
Basic Memory Model
Basic Memory Model

![Basic Memory Model Diagram]

Clock
Address
Data In
Write
Clock
Data Out

Read A1

A0 A1 A1

D1

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 V 5CSXFC6 has 557 of these blocks (696 KB)
Memory in Quartus: the Megafunction Wizard

Which megafuncon would you like to customize?
Select a megafuncon 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/Bk Type > Clks/Rd, Byte En > Regs/Ckens/Acirs > 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? 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?

- [ ] Auto
- [ ] MLAB
- [ ] M10K
- [ ] 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

RAM: 2-PORT

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

Block Type: M10K
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 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
The Perils of Memory Inference

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)
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.
Implementing Memory
Early Memories

Williams Tube CRT-based random access memory, 1946. Used on the Manchester Mark I. 2048 bits.
Early Memories

Mercury acoustic delay line.

Used in the EDASC, 1947.

32 × 17 bits
Early Memories

Magnetic core memory, 1952. IBM.
Early Memories

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”

Add. Data

<table>
<thead>
<tr>
<th>00</th>
<th>011</th>
</tr>
</thead>
<tbody>
<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

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

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>

2-to-4 Decoder
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 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, $1024 \times 1$ bit, 1972
SRAM Timing

- CS1
- CS2
- WE
- OE
- Addr
- Data

Write 1
Read 2
Toshiba TC55V16256J 256K × 16

Diagram of the Toshiba TC55V16256J 256K × 16 SRAM chip, showing pin connections and block diagram layout.
Dynamic RAM Cell

Column

Row

![SEM image of a Dynamic RAM Cell](image)
Ancient (c. 1982) DRAM: 4164 64K × 1
Basic DRAM read and write cycles

- **RAS**
- **CAS**
- **Addr**: Row, Col, Row, Col
- **WE**
- **Din**: to write
- **Dout**: read
Page Mode DRAM read cycle

- **RAS**: Row Address Strobe
- **CAS**: Column Address Strobe
- **Addr**: Address
- **WE**: Write Enable
- **Din**: Data In
- **Dout**: Data Out

The diagram illustrates the timing of the read cycle with corresponding signals for RAS, CAS, Addr, WE, Din, and Dout.
Samsung 8M × 16 SDRAM

- BA1
- BA0
- A11
- A10
- A2
- A1
- A0
- UDQM
- LDQM
- DQ15
- DQ14
- DQ1
- DQ0
- WE
- CAS
- RAS
- CS
- CKE
- CLK
- LRAS
- LCBR
- LWCE
- LCAS
- LWCBR
- L(U)DQM
- CLK
- CKE
- CS
- RAS
- CAS
- WE
- L(U)DQM
- Data Input Register
- Bank Select
- Row Buffer
- Refresh Counter
- Row Decoder
- Column Decoder
- LWE
- LDQM
- Timing Register
**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