General Concepts

- A memory is an array of storage locations
  - Each with a unique address
  - Like a collection of registers, but with optimized implementation
- Address is unsigned-binary encoded
  - \( n \) address bits \( \Rightarrow 2^n \) locations
- All locations the same size
  - \( 2^n \times m \) bit memory
Memory Sizes

- Use power-of-2 multipliers
  - Kilo (K): $2^{10} = 1,024 \approx 10^3$
  - Mega (M): $2^{20} = 1,048,576 \approx 10^6$
  - Giga (G): $2^{30} = 1,073,741,824 \approx 10^9$
- Example
  - 32K x 32-bit memory
  - Capacity = 1,025K = 1Mbit
  - Requires 15 address bits
- Size is determined by application requirements

Basic Memory Operations

- a inputs: unsigned address
- d_in and d_out
  - Type depends on application
- Write operation
  - en = 1, wr = 1
  - d_in value stored in location given by address inputs
- Read operation
  - en = 1, wr = 0
  - d_out driven with value of location given by address inputs
- Idle: en = 0
Example: Audio Delay Unit

- System clock: 1MHz
- Audio samples: 8-bit signed, at 50kHz
  - New sample arrives when audio_in_en = 1
- Delay control: 8-bit unsigned ⇒ ms to delay
- Output: audio_out_en = 1 when output ready

Audio Delay Datapath

- Max delay = 255ms
  - Need to store $255 \times 50 = 12,750$ samples
  - Use a $16K \times 8$-bit memory (14 address bits)
Audio Delay Control Section

Step 1: (idle state)
- audio\_in\_en = 0 ⇒ do nothing
- audio\_in\_en = 1 ⇒ write memory using counter value as address

Step 2:
- Read memory using subtracter output as address, increment counter

<table>
<thead>
<tr>
<th>State</th>
<th>audio_in_en</th>
<th>Next state</th>
<th>addr_sel</th>
<th>mem_en</th>
<th>mem_wr</th>
<th>count_en</th>
<th>audio_out_en</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>0</td>
<td>Step 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 1</td>
<td>1</td>
<td>Step 2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 2</td>
<td>-</td>
<td>Step 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Wider Memories

- Memory components have a fixed width
  - E.g., \(\times 1, \times 4, \times 8, \times 16, \ldots\)

- Use memory components in parallel to make a wider memory
  - E.g, three 16K\(\times 16\) components for a 16K\(\times 48\) memory
More Locations

To provide \(2^n\) locations with \(2^k\)-location components

- Use \(2^n/2^k\) components

Address A

- at offset \(A \mod 2^k\)
  - least-significant \(k\) bits of \(A\)
  - in component \(\lfloor A/2^k \rfloor\)
  - most-significant \(n-k\) bits of \(A\)
- decode to select component

Example:

64K×8 memory composed of 16K×8 components
Tristate Drivers

- Allow multiple outputs to be connected together
  - Only one active at a time
  - Remaining outputs are high-impedance
    - Both output transistors turned off
- Allow bidirectional input/output ports

Memories with Tristate Ports

- During write
  - memory d drivers hi-Z
  - memory senses d
- During read
  - selected memory drives d
- Fewer pins and wires
  - Reduced cost of PCB
- Usually not available within ASICs or FPGAs
Memory Types

- Random-Access Memory (RAM)
  - Can read and write
- Static RAM (SRAM)
  - Stores data so long as power is supplied
  - Asynchronous SRAM: not clocked
  - Synchronous SRAM (SSRAM): clocked
- Dynamic RAM (DRAM)
  - Needs to be periodically refreshed
- Read-Only Memory (ROM)
  - Combinational
  - Programmable and Flash rewritable
  - Volatile and non-volatile

Asynchronous SRAM

- Data stored in 1-bit latch cells
  - Address decoded to enable a given cell
  - Usually use active-low control inputs
  - Not available as components in ASICs or FPGAs
Asynch SRAM Timing

- Timing parameters published in data sheets
- Access time
  - From address/enable valid to data-out valid
- Cycle time
  - From start to end of access
- Data setup and hold
  - Before/after end of WE pulse
  - Makes asynch SRAMs hard to use in clocked synchronous designs

Example Data Sheet

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>18W</th>
<th>18W</th>
<th>18W</th>
<th>18W</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ CYCLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tRC</td>
<td>Read Cycle Time</td>
<td>13</td>
<td>15</td>
<td>17</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>tAD</td>
<td>Address Set to Valid</td>
<td>12</td>
<td>15</td>
<td>17</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>tDH</td>
<td>Data Hold</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>tCSS</td>
<td>CE LOW to Data Valid</td>
<td>12</td>
<td>15</td>
<td>17</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>tDHQ</td>
<td>CE LOW to Data Valid</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td>ns</td>
</tr>
</tbody>
</table>

| WRITE CYCLE |            |     |     |     |     |      |
| tWC | Write Cycle Time | 12 | 15 | 17 |     | ns   |
| tAC | CE LOW to Write End | 10 | 15 | 15 |     | ns   |
| tAS | Address Set to Write End | 10 | 15 | 15 |     | ns   |
| tAH | Address Hold from Write End | 0 | 0 | 0 |     | ns   |
| tDW | Address Set Up to Write Start | 0 | 15 | 15 |     | ns   |
| tPH | Pulse Width | 15 | 15 | 15 |     | ns   |
| tDHQ | Data Hold | 7 | 9 | 9 |     | ns   |
| tCSS | CE LOW to Chip Enable | 0 | 0 | 3 |     | ns   |
| tDEE | Chip Enable to End of Write | 10 | 15 | 15 |     | ns   |
Synchronous SRAM (SSRAM)

- Clocked storage registers for inputs
  - address, data and control inputs
  - stored on a clock edge
  - held for read/write cycle

- Flow-through SSRAM
  - no register on data output

Example: Coefficient Multiplier

- Compute function \( y = c_i \times x^2 \)
  - Coefficient stored in flow-through SSRAM
    - 12-bit unsigned integer index for \( i \)
  - \( x, y, c_i \) 20-bit signed fixed-point
    - 8 pre- and 8 post-binary point bits
  - Use a single multiplier
    - Multiply \( c_i \times x \times x \)
Multiplier Datapath

Multiplier Timing and Control
Pipelined SSRAM

- Data output also has a register
  - More suitable for high-speed systems
  - Access RAM in one cycle, use the data in the next cycle

Memories in VHDL

- RAM storage represented by an array signal

```vhdl
type RAM_4Kx16 is array (0 to 4095) of std_logic_vector(15 downto 0);
signal data_RAM : RAM_4Kx16;
...
data_RAM_flow_through : process (clk) is
begin
  if rising_edge(clk) then
    if en = '1' then
      if wr = '1' then
        data_RAM(to_integer(a)) <= d_in;  d_out <= d_in;
      else
        d_out <= RAM(to_integer(a));
      end if;
    end if;
  end if;
end process data_RAM_flow_through;
```
Example: Coefficient Multiplier

```vhdl
library ieee; use ieee.std_logic_1164.all,
    ieee.numeric_std.all, ieee.fixed_pkg.all;
entity scaled_square is
port ( clk, reset : in std_logic;
    start : in std_logic;
    i : in unsigned(11 downto 0);
    c_in, x : in sfixed(7 downto -12);
    y : out sfixed(7 downto -12) );
end entity scaled_square;
architecture rtl of scaled_square is
signal c_ram_en, c_ram_wr, x_ce, mult_sel, y_ce : std_logic;
signal c_out, x_out : sfixed(7 downto -12);
signal y_out : sfixed(7 downto -12);
type c_array is array (0 to 4095) of sfixed(7 downto -12);
signal c_RAM : c_array;
type state is (step1, step2, step3);
signal current_state, next_state : state;
begin
    c_ram_wr <= '0';
c_RAM_flow_through : process (clk) is
    begin
        if rising_edge(clk) then
            if c_ram_en = '1' then
                if c_ram_wr = '1' then
                    c_RAM(to_integer(i)) <= c_in;
                    c_out <= c_in;
                else
                    c_out <= c_RAM(to_integer(i));
                end if;
            end if;
        end if;
    end process c_RAM_flow_through;
end;
```
Example: Coefficient Multiplier

```vhdl
y_reg : process (clk) is
    variable operand1, operand2 : sfixed(7 downto -12);
begin
    if rising_edge(clk) then
        if y_ce = '1' then
            if mult_sel = '0' then
                operand1 := c_out;  operand2 := x_out;
            else
                operand1 := x_out;  operand2 := y_out;
            end if;
            y_out <= operand1 * operand2;
        end if;
    end if;
end process y_reg;

y <= y_out;
state_reg : process ...
next_state_logic : process ...
output_logic : process ...
end architecture rtl;
```

Pipelined SSRAM in VHDL

```vhdl
data_RAM_pipelined : process (clk) is
    variable pipeline_en : std_logic;
    variable pipeline_d_out : std_logic_vector(15 downto 0);
begin
    if rising_edge(clk) then
        if pipelined_en = '1' then
            d_out <= pipelined_d_out;
        end if;
        pipeline_en := en;
    end if;
    if en = '1' then
        if wr = '1' then
            data_RAM(to_integer(a)) <= d_in; pipeline_d_out := d_in;
        else
            pipeline_d_out := RAM(to_integer(a));
        end if;
    end if;
end process data_RAM_pipelined;
```
Generating SSRAM Components

- Variations on SSRAM behavior
  - E.g., write-first, read-first or no-change on write cycle
  - Burst accesses to successive locations
- Not all synthesis tools recognize the same templates
- Use a RAM core generator tool

Example: RAM Core Generator
Multiport Memories

- Multiple address, data and control connections to the storage locations
- Allows concurrent accesses
  - Avoids multiplexing and sequencing
- Scenario
  - Data producer and data consumer
- What if two writes to a location occur concurrently?
  - Result may be unpredictable
  - Some multi-port memories include an arbiter

FIFO Memories

- First-In/First-Out buffer
  - Connecting producer and consumer
  - Decouples rates of production/consumption

- Implementation using dual-port RAM
  - Circular buffer
  - Full: write-addr = read-addr
  - Empty: write-addr = read-addr
Example: FIFO Datapath

- Equal = full or empty
  - Need to distinguish between these states — How?

Example: FIFO Control

- Control FSM
  - → filling when write without concurrent read
  - → emptying when without concurrent write
  - Unchanged when concurrent write and read

<table>
<thead>
<tr>
<th>State</th>
<th>wr_en</th>
<th>rd_en</th>
<th>transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>filling</td>
<td>0</td>
<td>0</td>
<td>full = filling and equal</td>
</tr>
<tr>
<td>emptying</td>
<td>0</td>
<td>1</td>
<td>empty = emptying and equal</td>
</tr>
<tr>
<td>full</td>
<td>1</td>
<td>0</td>
<td>filling</td>
</tr>
</tbody>
</table>
Multiple Clock Domains

- Need to resynchronize data that traverses clock domains
  - Use resynchronizing registers
- May overrun if sender's clock is faster than receiver's clock
- FIFO smooths out differences in data flow rates
  - Latch cells inside FIFO RAM written with sender's clock, read with receiver's clock

Dynamic RAM (DRAM)

- Data stored in a 1-transistor/1-capacitor cell
  - Smaller cell than SRAM, so more per chip
  - But longer access time
- Write operation
  - pull bit-line high or low (0 or 1)
  - activate word line
- Read operation
  - precharge bit-line to intermediate voltage
  - activate word line, and sense charge equalization
  - rewrite to restore charge
DRAM Refresh

- Charge on capacitor decays over time
  - Need to sense and rewrite periodically
    - Typically every cell every 64ms
  - Refresh each location
- DRAMs organized into banks of rows
  - Refresh whole row at a time
- Can’t access while refreshing
  - Interleave refresh among accesses
  - Or burst refresh every 64ms

Read-Only Memory (ROM)

- For constant data, or CPU programs
- Masked ROM
  - Data manufactured into the ROM
- Programmable ROM (PROM)
  - Use a PROM programmer
- Erasable PROM (EPROM)
  - UV erasable
  - Electrically erasable (EEPROM)
  - Flash RAM
### Combinational ROM

- A ROM maps address input to data output
  - This is a combinational function!
  - Specify using a table
- Example: 7-segment decoder

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0111111</td>
<td>6</td>
<td>1111101</td>
</tr>
<tr>
<td>1</td>
<td>0000110</td>
<td>7</td>
<td>0000111</td>
</tr>
<tr>
<td>2</td>
<td>1011011</td>
<td>8</td>
<td>111111</td>
</tr>
<tr>
<td>3</td>
<td>1001111</td>
<td>9</td>
<td>1101111</td>
</tr>
<tr>
<td>4</td>
<td>1100110</td>
<td>10–15</td>
<td>1000000</td>
</tr>
<tr>
<td>5</td>
<td>1101101</td>
<td>16–31</td>
<td>0000000</td>
</tr>
</tbody>
</table>

### Example: ROM in VHDL

```vhdl
library ieee; use ieee.numeric_std.all;
architecture ROM_based of seven_seg_decoder is
    type ROM_array is
        array (0 to 31) of std_logic_vector ( 7 downto 1 );
    constant ROM_content : ROM_array := ( 0 => "0111111", 1 => "0000110", 2 => "1011011", 3 => "1001111", 4 => "1100110", 5 => "1101101", 6 => "1111101", 7 => "0000111", 8 => "1111111", 9 => "1101111", 10 to 15 => "1000000", 16 to 31 => "0000000" );
begin
    seg <= ROM_content(to_integer(unsigned(blank & bcd)));
end architecture ROM_based;
```
Flash RAM

- Non-volatile, readable (relatively fast), writable (relatively slow)
- Storage partitioned into blocks
  - Erase a whole block at a time, then write/read
  - Once a location is written, can't rewrite until erased
- NOR Flash
  - Can write and read individual locations
  - Used for program storage, random-access data
- NAND Flash
  - Denser, but can only write and read block at a time
  - Used for bulk data, e.g., cameras, memory sticks

Memory Errors

- Bits in memory can be flipped
- Hard error
  - The chip is broken
  - E.g., manufacturing defect, wear (in Flash)
- Soft error
  - Stored data corrupted, but cell still works
  - E.g., from atmospheric neutrons
- Soft-error rate
  - Frequency of occurrence
Error Detection using Parity

- Add a parity bit to each location
- On write access
  - compute data parity and store with data
- On read access
  - check parity, take exception on error
- If we could tell which bit flipped
  - correct by flipping it back, then write back to memory location
  - Can’t do this with parity

Error-Correcting Codes (ECC)

- Allow identification of the flipped bit
- Hamming Codes
  - E.g., for single-bit-error correction of \( N \)-bit word, need \( \log_2 N + 1 \) extra bits
- Example: 8-bit word, \( d_1...d_8 \)
  - 12-bit ECC code, \( e_1...e_{12} \)
  - \( e_1, e_2, e_4, e_8 \) are check bits, the rest data
Hamming Code Example

On write: Compute check bits and store with data

Every data bit covered by two or more check bits

On read: Recompute check bits and XOR with read check bits
result called the syndrome

0000 => no error

If data bit flipped
  covering bits of syndrome are 1
  = binary code of flipped ECC bit

If stored check bit flipped
  that bit of syndrome is 1

On error, unflip bit and rewrite memory location
Multiple-Error Detection

- What if two bits flip
  - syndrome identifies wrong bit, or is invalid
- One extra check bit allows
  - single-error correction, double-error detection

<table>
<thead>
<tr>
<th>$N$</th>
<th>Single-bit correction</th>
<th>Double-bit detection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Check bits</td>
<td>Overhead</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>50%</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>31%</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>19%</td>
</tr>
<tr>
<td>64</td>
<td>7</td>
<td>11%</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
<td>6.3%</td>
</tr>
<tr>
<td>256</td>
<td>9</td>
<td>3.5%</td>
</tr>
</tbody>
</table>

Summary

- Memory: addressable storage locations
- Read and Write operations
- Asynchronous RAM
- Synchronous RAM (SSRAM)
- Dynamic RAM (DRAM)
- Read-Only Memory (ROM) and Flash
- Multiport RAM and FIFOs
- Error Detection and Correction
  - Hamming Codes