External SRAM

A common type of system RAM is asynchronous static RAM (SRAM).

Access is more complicated than internal memory -- here data, address and control signals must be asserted in a specific order and held for a specific time.

A memory controller is usually used to shield the synchronous system from SRAM. It is responsible for generating the properly timed signals and making the SRAM look 'synchronous'.

Its performance is measured by the number of memory accesses that can be completed in a given time period.

Designing a memory controller that is optimal is non-trivial.

This set of slides demonstrates the development of a memory controller. Note that the timing characteristics of an SRAM will differ depending on the manufacturer, but the same basic principles apply.
External SRAM

Your boards do NOT have external SRAM installed by default but it is available on an expansion card.

We have Digilent Memory Modules, C2: 1Mb SRAM, no Flash available.

These expansion cards have two banks of ISSI IS61LV5128AL 512K x 8 SRAM

High-performance, low-power CMOS process with 10 ns access times. Single 3.3V power supply, static operation: no clock or refresh required. Easy memory expansion with $\overline{CE}$ (power down) and $\overline{OE}$ options.

Functional Block Diagram

A0-A18 \rightarrow \text{Decoder} \rightarrow 512K \times 8 \text{ Mem array}

I/O0-I/O7 \leftrightarrow \text{I/O data circuit} \leftrightarrow \text{Column I/O}

\overline{CE} \quad \overline{OE} \quad \text{Control circuit} \rightarrow \text{Column I/O}
External SRAM

Truth Table:

<table>
<thead>
<tr>
<th>Mode</th>
<th>WE</th>
<th>CE</th>
<th>OE</th>
<th>I/O operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not selected (power-down)</td>
<td>X</td>
<td>H</td>
<td>X</td>
<td>High-Z</td>
</tr>
<tr>
<td>Output disabled</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>High-Z</td>
</tr>
<tr>
<td>Read</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>$D_{out}$</td>
</tr>
<tr>
<td>Write</td>
<td>L</td>
<td>L</td>
<td>X</td>
<td>$D_{in}$</td>
</tr>
</tbody>
</table>

This device has a 19-bit address bus (A0-A18), a bi-directional 8-bit data bus (I/O0-I/O7) and three active low control signals, $\overline{CE}$, $\overline{OE}$ and $\overline{WE}$ (see table above).

The timing characteristics of an asynchronouns SRAM are complex and involve more than 24 parameters -- here we focus on only a few key parameters.

For **reading**:

- $t_{RC}$: read cycle time, the min time between two read operations (min 10 ns).
- $t_{AA}$: address access time, the time required to obtain a stable output data after an address change (max 10 ns).
- $t_{OHA}$: output hold time, the time that the output data remains valid after the address changes (min 2 ns).
SRAM Read Timing Parameters

- $t_{DOE}$: output enable access time, the time required to obtain valid data after the $\overline{OE}$ is activated (max 4 ns).
- $t_{HZOE}$: output enable high-Z time, the time for the tri-state buffer to enter the high-impedance state after $\overline{OE}$ is deactivated (max 4 ns).
- $t_{LZOE}$: output enable to low-Z time, the time for the tri-state buffer to leave the high-impedance state after $\overline{OE}$ is activated (min 0 ns).

Reading can be accomplished in two ways, with $\overline{OE}$ activated, changing the address will change the data; second, $\overline{OE}$ can be used (for interleaved read/writes).

Timing diagram for **address-controlled read cycle**

```
Address

\[ \begin{array}{c}
\text{D}_{\text{out}} \\
\text{previous data valid} \\
\text{data valid}
\end{array} \]
```

- $t_{RC}$
- $t_{OHA}$
- $t_{AA}$
SRAM Read and Write Timing Parameters

Timing diagram for $\overline{OE}$ controlled read cycle time

For write operations, the following specs are important:

- $t_{WC}$: write cycle time, the minimal time between two write operations (min 10 ns).
- $t_{SA}$: address setup time, the minimal time that the address must be stable before $\overline{WE}$ is activated (min 0 ns).
- $t_{HA}$: address hold time, the minimal time that the address must be stable after $\overline{WE}$ is deactivated (min 0 ns).
SRAM Read and Write Timing Parameters

- $t_{PWE1}$: $\overline{WE}$ pulse width, the minimal time that $\overline{WE}$ must be asserted (min 8 ns).
- $t_{SD}$: data setup time, the minimal time that data must be stable before the latching edge (the edge in which $\overline{WE}$ changes from 0 to 1) (min 6 ns).
- $t_{HD}$: data hold time, the minimal time that data must be stable after the latching edge (min 0 ns).

Data sheet gives several timing diagrams for write, this one for $\overline{WE}$ controlled
Memory Controller
Block diagram of memory controller

- **mem**: asserted to 1 to initiate a memory operation.
- **rw**: specifies read (1) or write (0) operation
- **addr**: is a 19-bit address
- **data_f2s**: 8-bit data to be written to SRAM.
- **data_s2f_r**: 8-bit registered data retrieved from SRAM
- **data_s2f_ur**: 8-bit unregistered data retrieved from SRAM
- **ready**: status signal indicating whether the controller is ready to accept a new command -- needed b/c memory operation may take more than 1 clock cycle.
Memory Controller

The memory controller provides a 'synchronous' wrap around the SRAM.

When the main system wants to access memory, it places the address and data (for writes) on the bus and activates mem and rw signals.

On the rising edge of clock, all signals are sampled by the memory controller and the operation is performed.

For reads, the data becomes available after 1 or 2 clock cycles.
Memory Controller

The data path consists of one address register and two data registers, which store the data for each direction.

Since the data bus, \( dio \), is a bi-directional signal, a tri-state buffer is inserted.

The FSM defines the control path which is constrained by the timing specs.

Consider the control sequence for a read operation:

Here, \( \overline{WE} \) is deactivated during the entire operation.
- Place the address on the \( ad \) bus and activate the \( \overline{OE} \) signal.
- Wait for at least \( t_{AA} \). The data from the SRAM becomes available after this interval.
- Retrieve the data from \( dio \) and deactivate the \( \overline{OE} \) signal.

For a write operation:
- Place address on \( ad \) bus and data on \( dio \) bus and activate \( \overline{WE} \) signal.
- Wait for at least \( t_{PWE1} \).
- Deactivate \( \overline{WE} \) -- data is latched to SRAM on 0 to 1 transition.
- Remove data from the \( dio \) bus.
Memory Controller

Note that $t_{HD}$ (data hold time after write ends) is 0 ns for this SRAM.

This means that it is possible to remove the data and deactivate $\overline{WE}$ simultaneously.

However, it is unwise to do this because of variations in propagation delays and it is best to ensure $\overline{WE}$ is deactivated first.

We consider first a 'safe' design -- one in which the design of the memory controller provides large timing margins.

The controller uses two clock cycles (20 ns with a 100 MHz system clk) to complete the memory access and requires 3 clk cycles for back-to-back operations.

The FSMD has 5 states, initially is in **idle**, and starts a memory operation when $mem$ signal is activated.

The $rw$ signal determines if it is a read or write operation.
For read operation, FSM moves to \textbf{r1}. As it does so, the memory address, \textit{addr}, is sampled and stored in \textit{addr\_reg}.
Memory Controller

The \textit{oe\_n} signal is activated in both the \textit{r1} and \textit{r2} states.

The \textit{dio} data is sampled and stored in \textit{data\_s2f\_reg} on the edge returning the FSM to the \textbf{idle} state.

Note that the \textit{oe\_n} signal is deactivated \textbf{after} the state transition is made, and therefore after the latching event.

Note that the block diagram shown earlier has two read ports.

The \textit{data\_s2f\_r} signal is driven by the registers and becomes available AFTER the FSM exits the \textit{r2} state (and back to \textbf{idle}).

The data on this port remains unchanged until the end of the next read cycle.

The \textit{data\_s2f\_ur} signal is connected to the SRAM’s \textit{dio} bus, with the data becoming valid at the end of \textit{r2} but is REMOVED when the FSM enters \textbf{idle}.

This port allows the main system (for some apps) to sample/store the data in its own registers, making it available one cycle earlier (same time as \textit{data\_s2f\_r}).
Memory Controller

If it had to wait for data_s2f_r to be valid, then the main system would not be able to latch the data in its own registers until the NEXT clk cycle.

For the write operation, the FSM moves to the w1 state.

The memory address, addr, and the data, data_f2s, are sampled and stored in addr_reg and data_f2s_reg registers on the transition to w1.

The we_n and tri_n signals are both activated in state w1. tri_n enables the tri-state buffer to put the data on the SRAM’s dio bus.

On the transition to w2, we_n is deactivated but tri_n remains activated. This ensures the data is properly latched to the SRAM when we_n changes from 0 to 1.

At the end of the write cycle, the FSM returns to idle and tri_n is deactivated to remove the data from the dio bus.
Timing Analysis of Memory Controller
We must verify the memory controller meets the timing specs. for the SRAM chip.

With a 100 MHz system clock, the FSM stays in each state for 10 ns.

For a read cycle, oe_n is asserted for two states, totaling 20 ns.
This provides a 10 ns margin over the 10 ns $t_{AA}$.

Deactivating oe_n in r1 imposes a more stringent timing constraint.

The data is stored in the data_s2f_reg when the FSM moves to idle.
Although oe_n is deactivated at the transition, the data remains valid for a small interval b/c the FPGA’s pad delay and the $t_{HZOE}$ delay (4 ns) of the SRAM.

During the write cycle, we_n is asserted in w1 -- the 10 ns interval exceeds the $t_{PWE1}$ spec (min 8 ns).

tri_n remains activated in w2 and thus ensures the data is still stable during the 0 -> 1 transition of the we_n signal.
Timing Analysis of Memory Controller

In terms of performance, both read and write operations take two clock cycles to complete.

During the read operation, the unregistered data \(data_{s2f\_ur}\) is available at the end of the second cycle (right before the rising edge)

While the registered data is available after the clock edge.

Note that both operations require a return to idle and therefore the sustained rate is 3 clocks (not 2).

With regard to the HDL implementation, the memory controller must generate fast, glitch-free control signals.

The output logic can be modified to include Moore look-ahead output buffers. This scheme adds a buffer (DFF) for each output signal to remove glitches and reduce clock-to-output delay.
HDL of Memory Controller

To compensate for the added clock cycle, we ‘look ahead’ at the state’s future value
\((state\_next \text{ signal})\) and use it to replace the \(state\_reg\) signals in the output logic.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity sram_ctrl is
  port(
    clk, reset: in std_logic;
    mem: in std_logic;
    rw: in std_logic;
    addr: in std_logic_vector(18 downto 0);
    data_f2s: in std_logic_vector(7 downto 0);
    ready: out std_logic;
    data_s2f_r, data_s2f_ur:
      out std_logic_vector(7 downto 0);
    ad: out std_logic_vector(18 downto 0);
    we_n, oe_n: out std_logic;
    dio: inout std_logic_vector(7 downto 0);
    ce_n: out std_logic);
  end sram_ctrl;
```
HDL of Memory Controller

```vhdl
architecture arch of sram_ctrl is
  type state_type is (idle, r1, r2, w1, w2);
  signal state_reg, state_next: state_type;
  signal data_f2s_reg, data_f2s_next: std_logic_vector(7 downto 0);
  signal data_s2f_reg, data_s2f_next: std_logic_vector(7 downto 0);
  signal addr_reg, addr_next: std_logic_vector(18 downto 0);
  signal we_buf, oe_buf, tri_buf: std_logic;
  signal we_reg, oe_reg, tri_reg: std_logic;

  begin
    process (clk, reset)
      begin
        if (reset = '1') then
          state_reg <= idle;
          addr_reg <= (others => '0');
  ```
HDL of Memory Controller

data_f2s_reg <= (others => '0');
data_s2f_reg <= (others => '0');
tri_reg <= '1';
we_reg <= '1';
ue_reg <= '1';

elsif (clk'event and clk = '1') then
    state_reg <= state_next;
    addr_reg <= addr_next;
    data_f2s_reg <= data_f2s_next;
    data_s2f_reg <= data_s2f_next;
    tri_reg <= tri_buf;
    we_reg <= we_buf;
    oe_reg <= oe_buf;

end if;
end process;
HDL of Memory Controller

-- next state logic

process(state_reg, mem, rw, dio, addr, data_f2s,
        data_f2s_reg, data_s2f_reg, addr_reg)
begin
   addr_next <= addr_reg;
   data_f2s_next <= data_f2s_reg;
   data_s2f_next <= data_s2f_reg;
   ready <= '0';
   case state_reg is
      when idle =>
         if (mem = '0') then
            state_next <= idle;
         else
            addr_next <= addr;
            if (rw = '0') then -- write
               state_next <= w1;
               data_f2s_next <= data_f2s;
            else
HDL of Memory Controller

```vhdl
state_next <= r1;
end if;
end if;
ready <= '1';
when w1 =>
state_next <= w2;
when w2 =>
state_next <= idle;
when r1 =>
state_next <= r2;
when r2 =>
data_s2f_next <= dio;
state_next <= idle;
end case;
end process;
```
HDL of Memory Controller

-- look-ahead output logic

process (state_next)
begin
tri_buf <= '1';
we_buf <= '1';
oe_buf <= '1';
case state_next is
when idle =>
when w1 =>
    tri_buf <= '0';
    we_buf <= '0';
when w2 =>
    tri_buf <= '0';
when r1 =>
    oe_buf <= '0';
when r2 =>
    oe_buf <= '0';
end case; end process;
HDL of Memory Controller

-- to main system
  data_s2f_r <= data_s2f_reg;
  data_s2f_ur <= dio;

-- to SRAM
  we_n <= we_reg;
  oe_n <= oe_reg;
  ad <= addr_reg;

-- I/O for SRAM chip
  ce_n <= '0';
  dio <= data_f2s_reg when tri_reg = '0'
           else (others => 'Z');

end arch;
Testing the Memory Controller

Basic testing circuit allows testing of a single read and write operation.

The following signals are used to control the operation.

- \(sw\): 4 bits wide and used as data or address input.
- \(led\): 4 bits wide and used to display retrieved data.
- \(btn(0)\): When asserted, the current value of \(sw\) is loaded to a data register. The output of the register is used as the data input for the write operation.
- \(btn(1)\): When asserted, the controller uses the value of \(sw\) as a memory address and performs a write operation.
- \(btn(2)\): When asserted, the controller uses the value of \(sw\) as a memory address and performs a read operation. The readout is routed to the \(led\) signal.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ram_ctrl_test is
  port(
    clk, reset: in std_logic;
    sw: in std_logic_vector(3 downto 0);
```
Testing the Memory Controller

\begin{verbatim}
btn: in std_logic_vector(2 downto 0);
led: out std_logic_vector(3 downto 0);
ad: out std_logic_vector(18 downto 0);
we_n, oe_n: out std_logic;
dio: inout std_logic_vector(7 downto 0);
ce_n: out std_logic

end ram_ctrl_test;

architecture arch of ram_ctrl_test is
constant ADDR_W: integer:= 19;
constant DATA_W: integer:= 8;
signal addr: std_logic_vector(ADDR_W-1 downto 0);
signal data_f2s, data_s2f:
    std_logic_vector(DATA_W-1 downto 0);
signal mem, rw: std_logic;
signal data_reg: std_logic_vector(3 downto 0);
signal db_btn: std_logic_vector(2 downto 0);
\end{verbatim}
Testing the Memory Controller

begin
ctrl_unit: entity work.sram_ctrl
  port map(
    clk=>clk, reset=>reset, mem=>mem, rw=>rw,
    addr=>addr, data_f2s=>data_f2s,
    ready=>open, data_s2f_r=>data_s2f,
    data_s2f_ur=>open, ad=>ad, we_n=>we_n,
    oe_n=>oe_n, dio=>dio, ce_n=>ce_n);

debounce_unit0: entity work.debounce
  port map(
    clk=>clk, reset=>reset, sw=>btn(0),
    db_level=>open, db_tick=>db_btn(0));

debounce_unit1: entity work.debounce
  port map(
    clk=>clk, reset=>reset, sw=>btn(1),
    db_level=>open, db_tick=>db_btn(1));
Testing the Memory Controller

debounce_unit2: entity work.debounce

port map(
    clk=>clk, reset=>reset, sw=>btn(2),
    db_level=>open, db_tick=>db_btn(2));

--data registers
process(clk)
begin
    if (clk'event and clk = '1') then
        if (db_btn(0) = '1') then
            data_reg <= sw;
        end if;
    end if;
end process;

-- address
addr <= "0000000000000000" & sw;
Testing the Memory Controller

-- command

process (db_btn, data_reg)

begin

data_f2s <= (others => '0');

if (db_btn(1) = '1') then -- write
    mem <= '1';
    rw <= '0';
    data_f2s <= "0000" & data_reg;

elsif (db_btn(2) = '1') then -- read
    mem <= '1';
    rw <= '1';

else
    mem <= '0';
    rw <= '1';

end if;

end process;

led <= data_s2f(3 downto 0);

end arch;