Lecture 2

Embedded system design

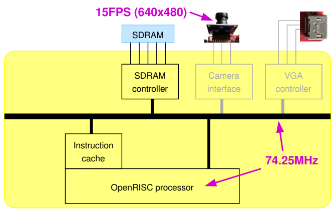
Memories

CS476 - ESD
February 29, 2024

Dr. Theo Kluter
EPFL

Notes

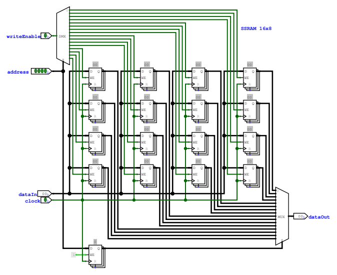
Acceleration



- Last week we have seen that our system cannot calculate Sobel in real time.
- We can accelerate the system by moving parts of the software to hardware.
- Ways to do this are custom instructions, accelerators, stream processing, ...
- We will visit all these methods later on. But all have something in common: they often need memory for temporal storage.

Notes

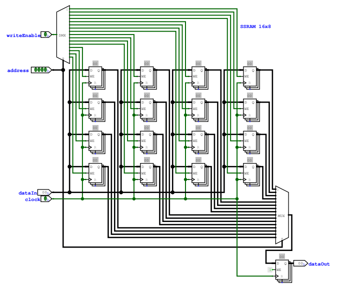
Synchronous Static Random Access Memories (SSRAM's)



- In digital technology nodes (ASIC and FPGA) we only find SSRAM's. Of course they are not build-up with flipflops as shown here.
- Typical for on-chip SSRAM's is that they have uni-directional data-buses, hence `dataIn` and `dataOut`.
- The signal `address` selects the memory cell and the signal `writeEnable` indicates if the cell should be written.
- There are two distinct behaviors in case of a write:
  - *Write before read*: The value written to the memory cell is also available on the output.
  -

Notes

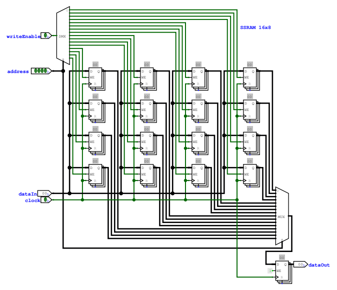
Synchronous Static Random Access Memories (SSRAM's)



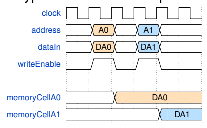
- In digital technology nodes (ASIC and FPGA) we only find SSRAM's. Of course they are not build-up with flipflops as shown here.
- Typical for on-chip SSRAM's is that they have uni-directional data-buses, hence `dataIn` and `dataOut`.
- The signal `address` selects the memory cell and the signal `writeEnable` indicates if the cell should be written.
- There are two distinct behaviors in case of a write:
  - *Write before read*: The value written to the memory cell is also available on the output.
  - *Read before write*: The value in the memory cell prior to the write operation is available on the output.

Notes

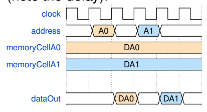
Synchronous Static Random Access Memories (SSRAM's)



- A typical SSRAM write operation is given by:



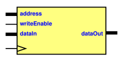
- A typical SSRAM read operation is given by (note the delay):



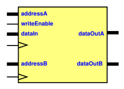
Notes

Synchronous Static Random Access Memories (SSRAM's)

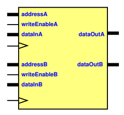
- SSRAM's can be found in different configurations, namely:
  - single-ported



  - This is the smallest memory and arguably most used.
  - semi-dual-ported



  - Here we have two *read-ports*, but we can only write on the *A-port*.
  - true-dual-ported



  - Here we have two complete ports that access the same memory array.
- And we can easily describe them in Verilog.

Notes

Single ported SSRAM

```
module singlePortSSRAM #( parameter bitwidth = 8,
                          parameter nrOfEntries = 512,
                          parameter readAfterWrite = 0 )
                        ( input wire                              clock,
                                                                  writeEnable,
                          input wire [$clog2(nrOfEntries)-1 : 0] address,
                          input wire [bitwidth-1 : 0]            dataIn,
                          output reg [bitwidth-1 : 0]            dataOut);

  reg [bitwidth-1 : 0] memoryContent [nrOfEntries-1 : 0];

  always @(posedge clock)
    begin
      if (readAfterWrite != 0) dataOut = memoryContent[address];
      if (writeEnable == 1'b1) memoryContent[address] = dataIn;
      if (readAfterWrite == 0) dataOut = memoryContent[address];
    end

endmodule
```

Notes

Semi dual-ported SSRAM

```
module semiDualPortSSRAM #( parameter bitwidth = 8,
                            parameter nrOfEntries = 512,
                            parameter readAfterWrite = 0 )
                          ( input wire                              clockA, clockB,
                                                                    writeEnable,
                            input wire [$clog2(nrOfEntries)-1 : 0] addressA, addressB,
                            input wire [bitwidth-1 : 0]            dataIn,
                            output reg [bitwidth-1 : 0]            dataOutA, dataOutB);

  reg [bitwidth-1 : 0] memoryContent [nrOfEntries-1 : 0];

  always @(posedge clockA)
    begin
      if (readAfterWrite != 0) dataOutA = memoryContent[addressA];
      if (writeEnable == 1'b1) memoryContent[addressA] = dataIn;
      if (readAfterWrite == 0) dataOutA = memoryContent[addressA];
    end

  always @(posedge clockB)
    dataOutB = memoryContent[addressB];

endmodule
```

Notes

Dual-ported SSRAM

```
module dualPortSSRAM #( parameter bitwidth = 8,
                        parameter nrOfEntries = 512,
                        parameter readAfterWrite = 0 )
                      ( input wire                              clockA, clockB,
                                                                writeEnableA, writeEnableB,
                        input wire [$clog2(nrOfEntries)-1 : 0] addressA, addressB,
                        input wire [bitwidth-1 : 0]            dataInA, dataInB,
                        output reg [bitwidth-1 : 0]            dataOutA, dataOutB);

  reg [bitwidth-1 : 0] memoryContent [nrOfEntries-1 : 0];

  always @(posedge clockA)
    begin
      if (readAfterWrite != 0) dataOutA = memoryContent[addressA];
      if (writeEnableA == 1'b1) memoryContent[addressA] = dataInA;
      if (readAfterWrite == 0) dataOutA = memoryContent[addressA];
    end

  always @(posedge clockB)
    begin
      if (readAfterWrite != 0) dataOutB = memoryContent[addressB];
      if (writeEnableB == 1'b1) memoryContent[addressB] = dataInB;
      if (readAfterWrite == 0) dataOutB = memoryContent[addressB];
    end

endmodule
```

Notes

Notes

SSRAM's in ASIC and FPGA

- In ASIC-design the size of the SSRAM's is dependent on the memory-generator and the *area* you have available.
- In FPGA-design it is more restricted, as the memories are already implemented. You can only use what you have:

Table 1–1. Resources for the Cyclone IV E Device Family

Resources	EP4CE6	EP4CE10	EP4CE15	EP4CE22	EP4CE30	EP4CE40	EP4CE55	EP4CE75	EP4CE115
Logic elements (LEs)	6,272	10,320	15,408	22,320	28,848	39,600	55,856	75,408	114,480
Embedded memory (Kbits)	270	414	504	594	594	1,134	2,340	2,745	3,888
Embedded 18 × 18 multipliers	15	23	56	66	66	116	154	200	266
General-purpose PLLs	2	2	4	4	4	4	4	4	4
Global Clock Networks	10	10	20	20	20	20	20	20	20
User I/O Banks	8	8	8	8	8	8	8	8	8
Maximum user I/O [(1)]	179	179	343	153	532	532	374	426	528

Note to Table 1–1:

(1) The user I/Os count from pin-out files includes all general purpose I/O, dedicated clock pins, and dual purpose configuration pins. Transceiver pins and dedicated configuration pins are not included in the pin count.

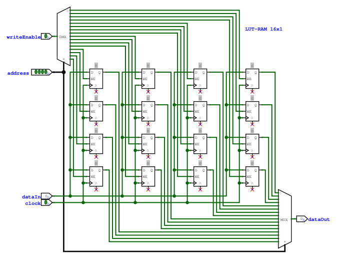
Notes

SSRAM's in ASIC and FPGA

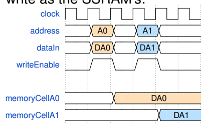
- For the FPGA we are using, following are the permissible `nrOfEntries x bitwidth` configurations:
  - `8192 x 1 bit`
  - `4092 x 2 bit`
  - `2048 x 4 bit`
  - `1024 x 8 bit` or `1024 x 9 bit`
  - `512 x 16 bit` or `512 x 18 bit`
  - `256 x 32 bit` or `256 x 36 bit`
- Other configurations are possible by using partially/multiple of these SSRAM's.
- By using the earlier seen Verilog descriptions, the synthesis tool will map to these SSRAM's.
- **Warning:** If your design uses more SSRAM memory bits as available on your FPGA, the synthesis tool will implement parts of the memory bits as flipflops and multiplexers. This will:
  - Explode the size of your design (often it cannot be mapped any more on the FPGA).
  - Have a severe impact on the critical path of your design (read the speed you can operate your design).
- For small memories, most FPGA's provide also the so-called LUT-RAM's. These have most of the time a `16 x 1 bit` configuration in a single-port or semi dual-port architecture.
- *Note:* the FPGA on our platform does not support LUT-RAM's.

Notes

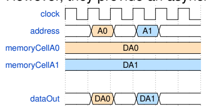
LUT-RAM's



- The LUT-RAM's have the same synchronous write as the SSRAM's:



- However, they provide an asynchronous read:



- Also LUT-RAMs can be easily described in Verilog:

Notes

Single ported LUT-RAM

```
module singlePortLUTRAM #( parameter bitwidth = 8,
                           parameter nrOfEntries = 32)
                         ( input wire                            clock,
                                                                 writeEnable,
                           input wire [$clog2(nrOfEntries)-1 : 0] address,
                           input wire [bitwidth-1 : 0]           dataIn,
                           output wire [bitwidth-1 : 0]          dataOut);

  reg [bitwidth-1 : 0] memoryContent [nrOfEntries-1 : 0];

  assign dataOut = memoryContent[address];

  always @(posedge clock)
    if (writeEnable == 1'b1) memoryContent[address] = dataIn;

endmodule
```

EPFL

Embedded system design

Dr. Theo Kluter

Notes

Semi dual-ported LUT-RAM

```
module semiDualPortLUTRAM #( parameter bitwidth = 8,
                             parameter nrOfEntries = 32)
                           ( input wire                            clock,
                                                                   writeEnable,
                             input wire [$clog2(nrOfEntries)-1 : 0] addressA, addressB,
                             input wire [bitwidth-1 : 0]           dataIn,
                             output wire [bitwidth-1 : 0]          dataOutA, dataOutB);

  reg [bitwidth-1 : 0] memoryContent [nrOfEntries-1 : 0];

  assign dataOutA = memoryContent[addressA];
  assign dataOutB = memoryContent[addressB];

  always @(posedge clock)
    if (writeEnable == 1'b1) memoryContent[addressA] = dataIn;

endmodule
```

EPFL

Embedded system design

Dr. Theo Kluter

Notes

But how to use those memories?

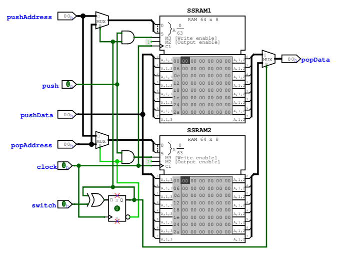
- We now have seen the on-chip memory architectures.
- We also have seen how to instantiate them in Verilog.
- We are now going to concentrate on how to use them, namely:
  - Ping-Pong buffers.
  - Last-in First-out (LIFO) buffers.
  - First-in First-out (FIFO) buffers.
- Each of these buffers are used for particular data-accesses in our system.
- Before starting with the buffers, some definitions:
  - *Producer*: a producer is an entity that generates data.
  - *Consumer*: a consumer is an entity that reads the data and does something with it.
  - *Push*: a push is a write of a datum by a producer.
  - *Pop*: a pop is a read of a datum by a consumer.

EPFL

Embedded system design

Dr. Theo Kluter

Ping-pong buffers



- In ping-pong buffers the producer writes it's data in one memory, whilst the consumer reads from the other memory. The moment both are done, the memories are switched.
- Typical applications for these kind of buffers are:
  - Data-transfer calculation overlap.
  - The access pattern of the producer on the data is different from the consumer.
  - The push/pop frequency is different, hence the producer/consumer have other timely accesses.
  - The consumer needs to access certain data multiple times, whilst the producer only provides it once.
- Of course, this only works if the consumer can consume the data in the time-slot that the producer requires to produce one block of data!

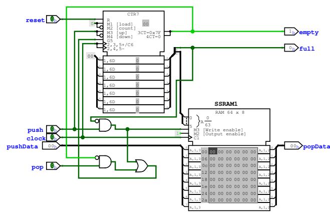
Notes

Ping-pong buffers

- The ping-pong buffers are arguably the most versatile kind of buffers.
- But how to determine the size of them?
- What about the inferred delay, as the consumer always performs the calculations when already one set of data is provided by the producer. Otherwise formulated: the consumer always lacks one time-slot behind.
- What is the influence on *area*, *performance*, and *power consumption*?
- Does it make sense.....
- All questions for which there is no simple answer, as it depends the requirements and trade-offs.

Notes

Last-in First-out (LIFO) buffers

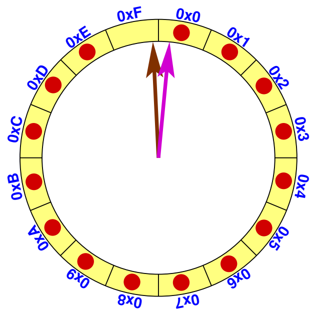


- In LIFO buffers, the last value *pushed* is the first that is *popped.* This can easily be realized to use an up/down counter that generates the address for the SSRAM.
- Typical applications for LIFO-buffers are:
  - Data reordering.
  - Temporal storage of values (think of the stack).
- In practice, the LIFO-buffers are not often used, more appropriate are the FIFO-buffers.

Notes

First-in First-out (FIFO) buffers

- In a FIFO-buffer we transform the SSRAM into a circular buffer.
- At the beginning the FIFO is *empty*. Hence the *push-pointer* equals the *pop-pointer*.
- When the producer pushes a datum, the *push-pointer* will increment.
- When the consumer does not pop, at a certain moment the producer filled the FIFO. The FIFO is full.
- The consumer makes again place by poping.
- Of course in normal circumstances the producer and consumer have both actions, such that the state of the FIFO changes continuously.
- And the FIFO can even become empty again.



Notes

First-in First-out (FIFO) buffers

- FIFO-buffers are arguably the most used buffers in hardware.
- Typical applications of FIFO-buffers are:
  - Timely access pattern buffering (e.g. the producer generates the data in another timely manner as the consumer can handle them).
  - Save clock-boundary crossings.
  - ...
- As you can imagine, we would like to have a generic description of a FIFO-buffer, something we are going to do in today's practical work.
- But there is one part that is missing, how to test?

Notes

Testing a unit by using a testbench

- We begin with our design. We call this the *Device Under Test (DUT)*.
- The first component of a testbench is the *input stimuli generator*, which provides the various *test vectors*.
- Then we have to ensure correct "output values" of the DUT. This is done by the *Output reaction checker*.
- The *Input stimuli generator* and the *Output reaction checker* form the *test-harnas*.
- Whereas the DUT only uses *synthesizable* Verilog descriptions, the *test-harnas* uses *non-synthesizable* Verilog descriptions.
- The *test-harnas* is described in a new **module**, where the DUT is used as a **component**. This **module** is called the testbench.

Notes

Our device under test

- Let's take a FIFO as example for how to make a testbench. The FIFO is defined by:

```
module fifo #(parameter nrOfEntries = 16,
              parameter bitWidth = 32)
             (input wire                 clock,
                                         reset,
                                         push,
                                         pop,
              input wire [bitWidth-1:0]  pushData,
              output wire                full,
                                         empty,
              output wire [bitWidth-1:0] popData);
endmodule
```

- We have 2 parameters, and several connections.
- Note that we require a clock and a reset.
- We can now build-up our basic testbench:

Notes

testbench

```
/* set the time-units for simulation */
`timescale 1ps/1ps

module fifoTestbench;

  reg reset, clock;
  initial
    begin
      reset = 1'b1;
      clock = 1'b0;                 /* set the initial values */
      repeat (4) #5 clock = ~clock; /* generate 2 clock periods */
      reset = 1'b0;                 /* de-activate the reset */
      forever #5 clock = ~clock;    /* generate a clock with a period of 10 time-units */
    end

  reg s_push, s_pop;
  wire s_full, s_empty; /* define the signals for the DUT */
  reg [7:0] s_pushData;
  wire [7:0] s_popData;

  fifo #(.nrOfEntries(32), /* instantiate the DUT as component */
         .bitWidth(8)) DUT
        (.clock(clock),
         .reset(reset),
         .push(s_push),
         .pop(s_pop),
         .pushData(s_pushData),
         .full(s_full),
         .empty(s_empty),
         .popData(s_popData));

  initial
    begin
      $dumpfile("fifoSignals.vcd"); /* define the name of the .vcd file that can be viewed by GTKWAVE */
      $dumpvars(1,DUT);             /* dump all signals inside the DUT-component in the .vcd file */
    end

endmodule
```

Notes

testbench

- Next we have to create the *input stimuli generator*, there are various ways to do this, namely:
  - A finite state machine that generates the required input values.
  - An initial block that generates the stimuli.
  - A model/files that contain the various values.
  - ...
- This time we will restrict ourselves to an initial block, like:

```
initial
  begin
    s_push = 1'b0;
    s_pop = 1'b0;
    s_pushData = 8'd0;
    @(negedge reset);              /* wait for the reset period to end */
    repeat(2) @(negedge clock);    /* wait for 2 clock cycles */
    s_push = 1'b1;
    repeat(32) @(negedge clock) s_pushData = s_pushData + 8'd1;
    s_push = 1'b0;
    s_pop = 1'b1;
    repeat(32) @(negedge clock); /* wait for 32 clock cycles */
    s_pop = 1'b0;
    $finish;                       /* finish the simulation */
  end
```

- The checker we leave for the moment and just look at the wave-files.

Notes