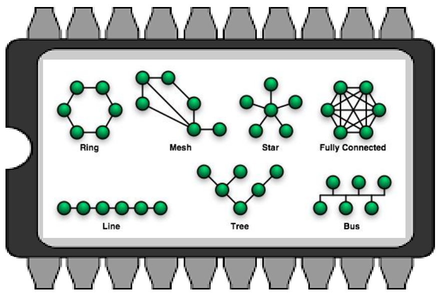
EEE4120F

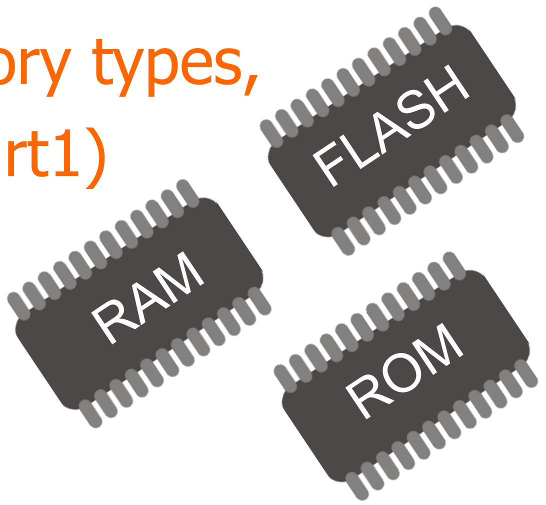
High Performance Embedded Systems

Lecture 21

On-chip Interconnection, Memory types, Memory Control Units (part1)

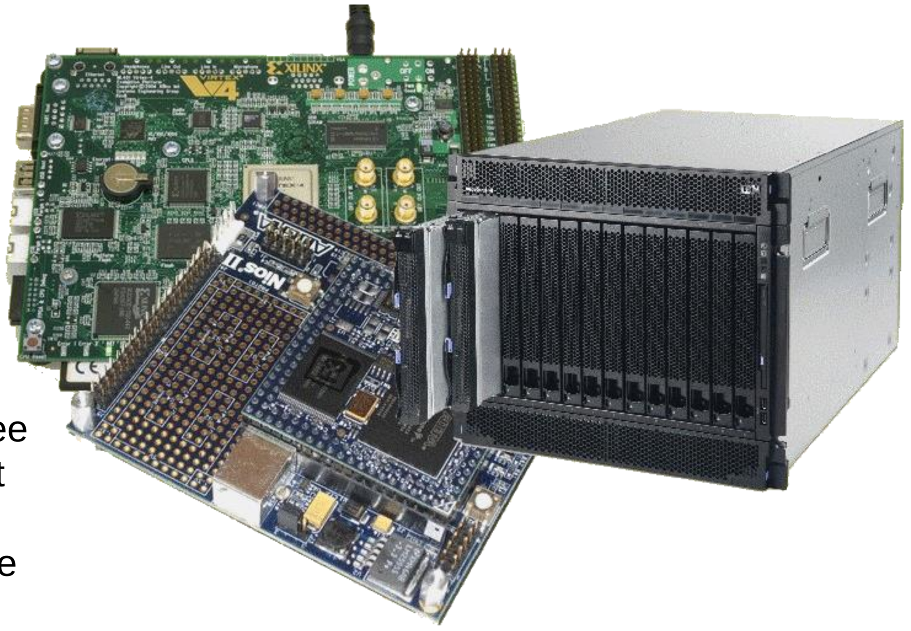


Lecturer:
Simon Winberg



Lecture Overview

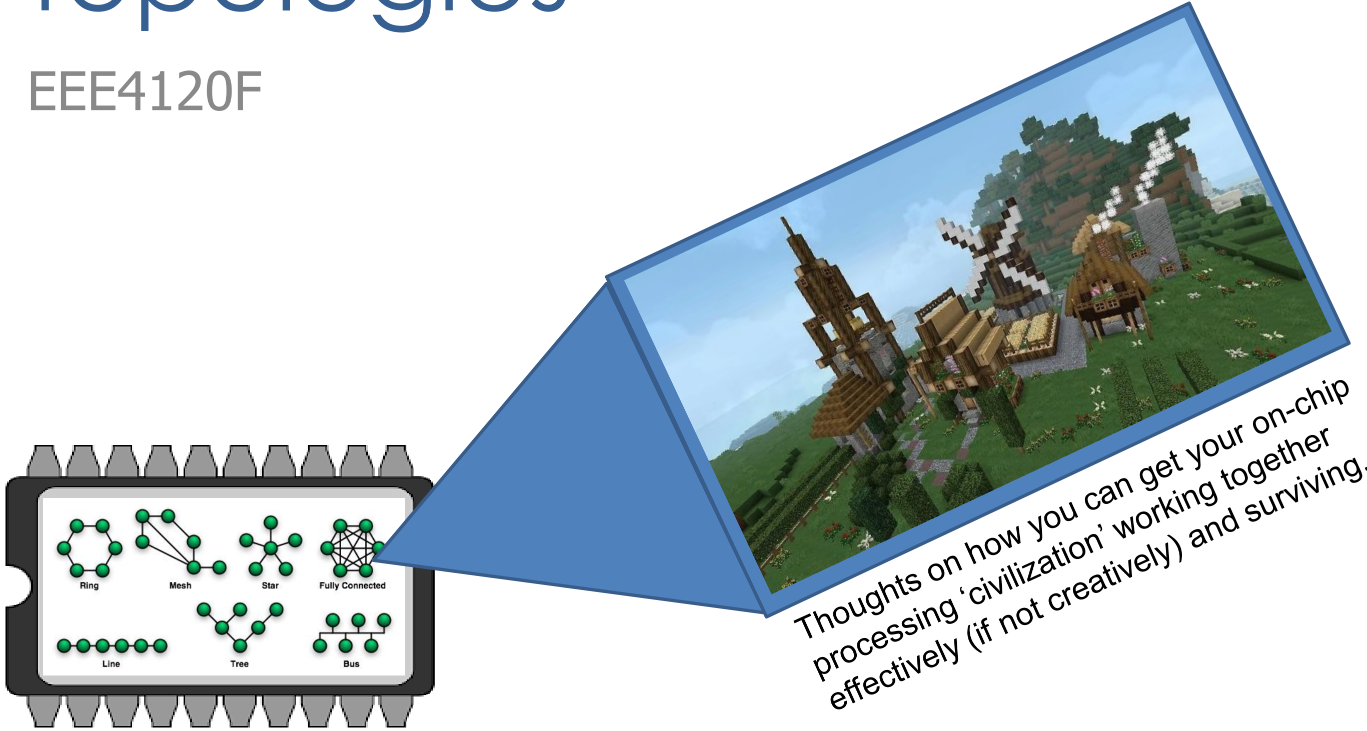
- On-chip interconnection bus topologies
- Interfacing standards
- Memory types
- Memory Control Units (part 1 of 2)
- Using Memory & MCU in Verilog



Comments on this topic:
These are aspects a computer engineer or an embedded systems developer should know about. Regards of which YODA project chosen, the first three parts are relevant to both OpenCL or FPGA. The last three parts (covering the bulk of this lecture) is more closely relevant to FPGA-based, and special-purpose computer, development techniques.

On-chip communication topologies

EEE4120F

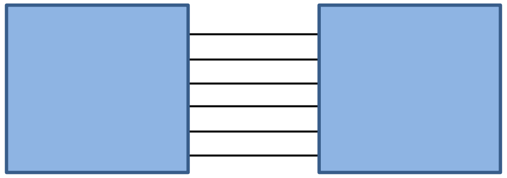


Thoughts on how you can get your on-chip processing ‘civilization’ working together effectively (if not creatively) and surviving.

On-chip interconnection topologies

- Point-to-point direct coupling (or dedicated peer interface)
  - One-to-one connection, or one to many
  - *Benefits:* simple (simplest, isn't really a bus), fast, allows high bandwidth, low area, quick & easy, low power (closely coupled), efficient to implement (no need for driver lines, separate arbitrator etc.).
  - *Drawbacks:* only direct connections. Often non-standard control lines.

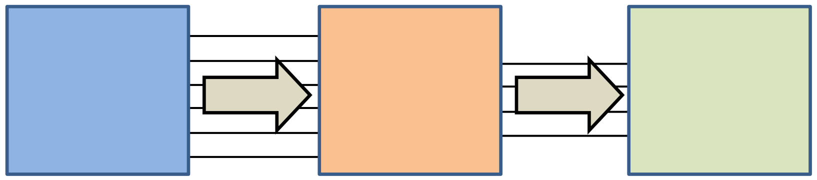
(may lead to daisy chaining)



On-chip daisy chain topologies / dataflow interconnects

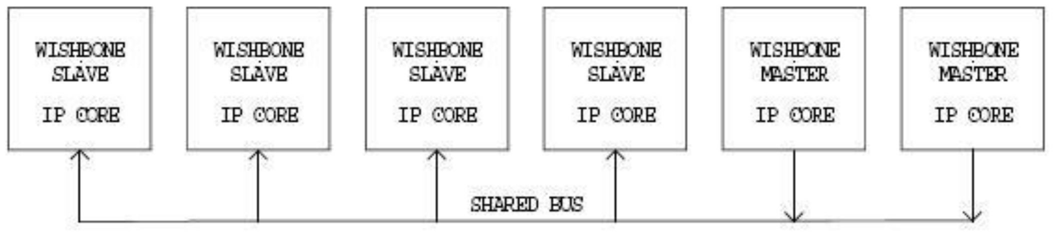
- Chain of point-to-point directly coupled links
  - A sequence of one-to-one connections
  - *Benefits:* (fairly) simple, natural approach for much signal processing (feed forward data flow), supports pipelining, fast neighbour links allows high bandwidth, low area, typically no need for separate arbitrator (if sequenced right).
  - *Drawbacks:* Comprise only direct links between neighbours. Often non-standard control lines. Challenges in different processing speeds for each core (can lead to trickey buffering).

(may lead towards ring topology)



On-chip bus topologies

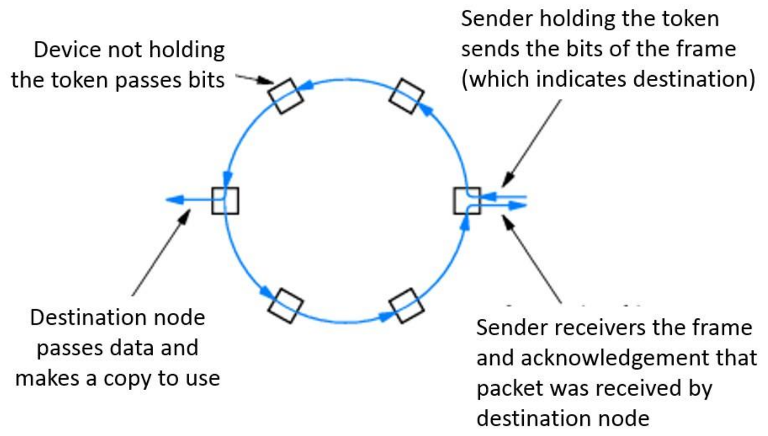
- Shared bus topology
  - Multiple masters and slaves on a shared bus.
  - Requires a bus arbiter (or bus arbitrator module, the BAM; 'master' could be this)
  - *Benefits:* simple, extensible, low area cost, easy to build & efficient to implement.
  - *Drawbacks:* Larger load per data bus line, longer delay for data transfer, larger energy consumption, and lower bandwidth.



(see next lecture for more on this)

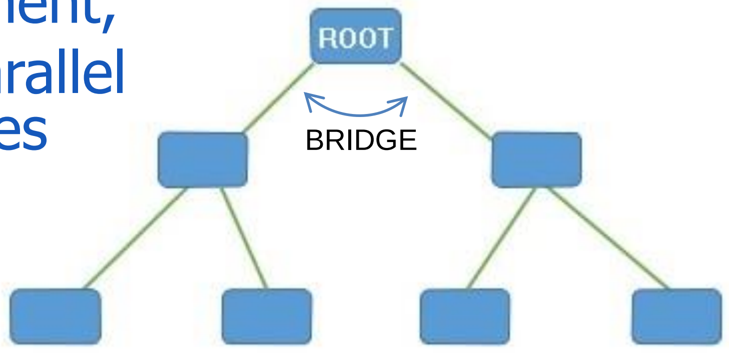
On-chip bus topologies

- Ring topology
  - Each node alternates between master/slave to receive and transmit over the ring interface.
  - Need a ring communication protocol, usually a token-pass protocol.
  - *Benefits:* somewhat scalable; comprises only closely-coupled connections between neighbouring devices (i.e., not all components needing to be coupled to the same bus lines); Low power for driving signals (not needing to drive signals far and to many receivers); Possibly higher bandwidths.
  - *Drawbacks:* limited scalability; potentially high latencies (long delay between getting token and forwarding data), and possibly lower bandwidth. May cause collectively higher power (due to having to relay signals, albeit each relay using little power)


Device not holding the token passes bits
Sender holding the token sends the bits of the frame (which indicates destination)
Destination node passes data and makes a copy to use
Sender receivers the frame and acknowledgement that packet was received by destination node

On-chip bus topologies

- Hierarchical bus:
  - Several shared buses interconnected by bridges forming a hierarchy / tree.
  - Components placed at an appropriate level in the hierarchy according to the performance level they require and which nodes they speak to.
  - Disadvantages: Transactions across the bridge involve additional overhead; during transfer both buses inaccessible to other nodes.
  - Advantages: Larger throughput using this model (than simple bus) because:
    a. Decreased load per bus segment,
    b. Potential for simultaneous parallel transactions on different buses
    c. Multiple transfers can be preceded bridges in a pipelined manner.



Interfacing Standards

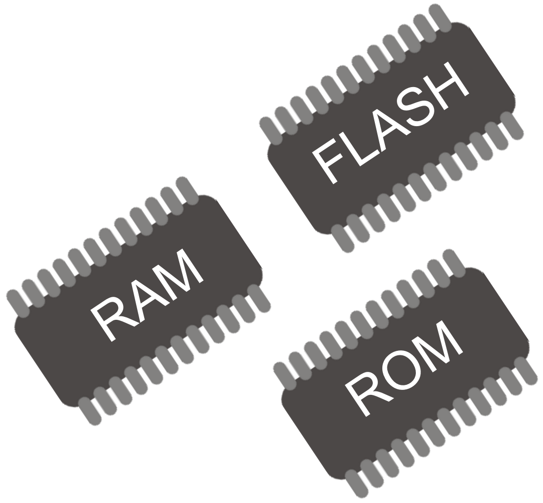
- The Avalon bus by Altera – Open Standard
- Advanced Microcontroller Bus Architecture (AMBA) by ARM – Open Standard
- On-chip Peripheral Bus (OPB) by Xilinx
- Wishbone bus (originally developed by Silicore Corporation) – Open Standard

(see next lecture for more on this)

Memories (recap)

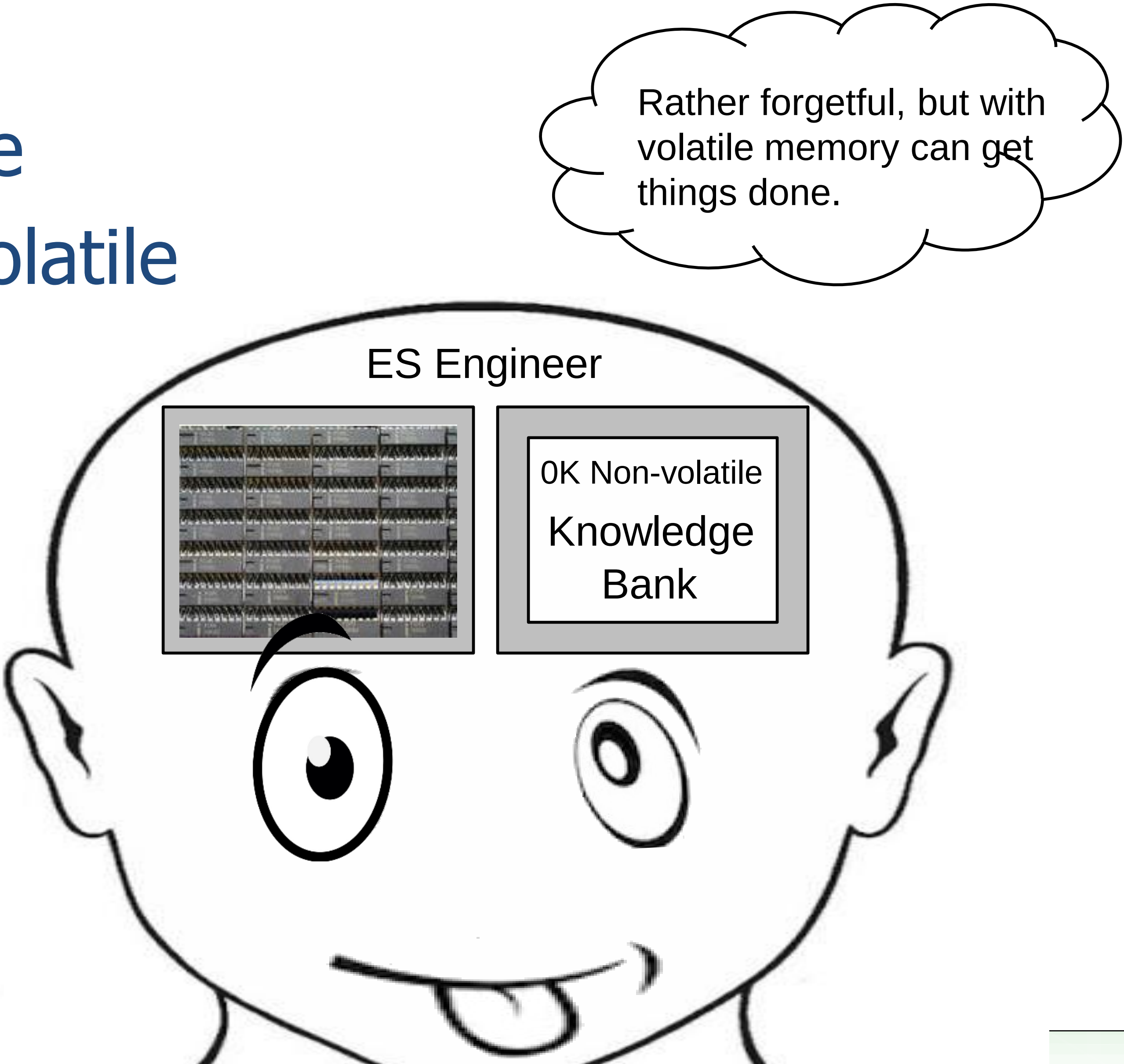
EEE4120F

What you learned previously about memories and memory types (e.g. in ES2) are of course relevant to FPGA-based embedded systems as well. A super fast recap of the essential points follows…



Memory types

- Volatile
- Non-volatile

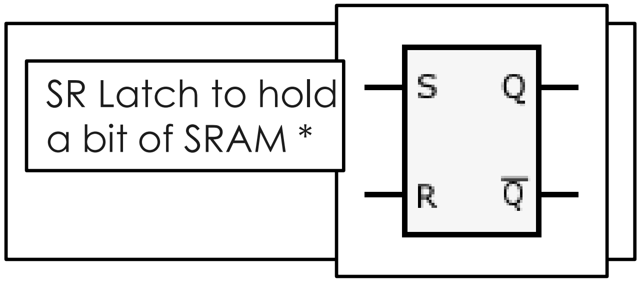

Rather forgetful, but with volatile memory can get things done.
ES Engineer
0K Non-volatile
Knowledge Bank

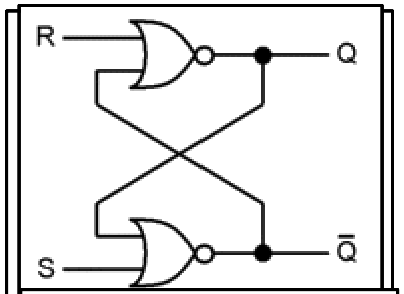
Volatile memory

- DRAM
  - Capacitor stores "memory" that leaks away and needs to be periodically refreshed
  - High memory capacity
- SDRAM = Synchronous DRAM
  - Runs in synch with system* clock
  - DDR SDRAM = Double-data rate SDRAM, runs at 2x the system clock

* Note the system clock in this case is closer to the "motherboard" clock. Usually considerably slower than the processor clock (standard DRAM may have its own even slower clock and synchronization hassles)

Volatile memory


SR Latch to hold a bit of SRAM *


SR Latch implemented using two NOR gates *

- SRAM
  - Static RAM
  - Does not need refreshing
  - Uses "bistable latching circuitry" (i.e. a flip flop) to store each bit
  - Can be very fast compared to DRAM
  - A small amount of SRAM (~16 Kb) is typically used within a microcontroller / FPGA to hold things such as a boot loader and interrupt vectors, and as CACHE

* Images from http://en.wikipedia.org/wiki/Latch_(electronics)

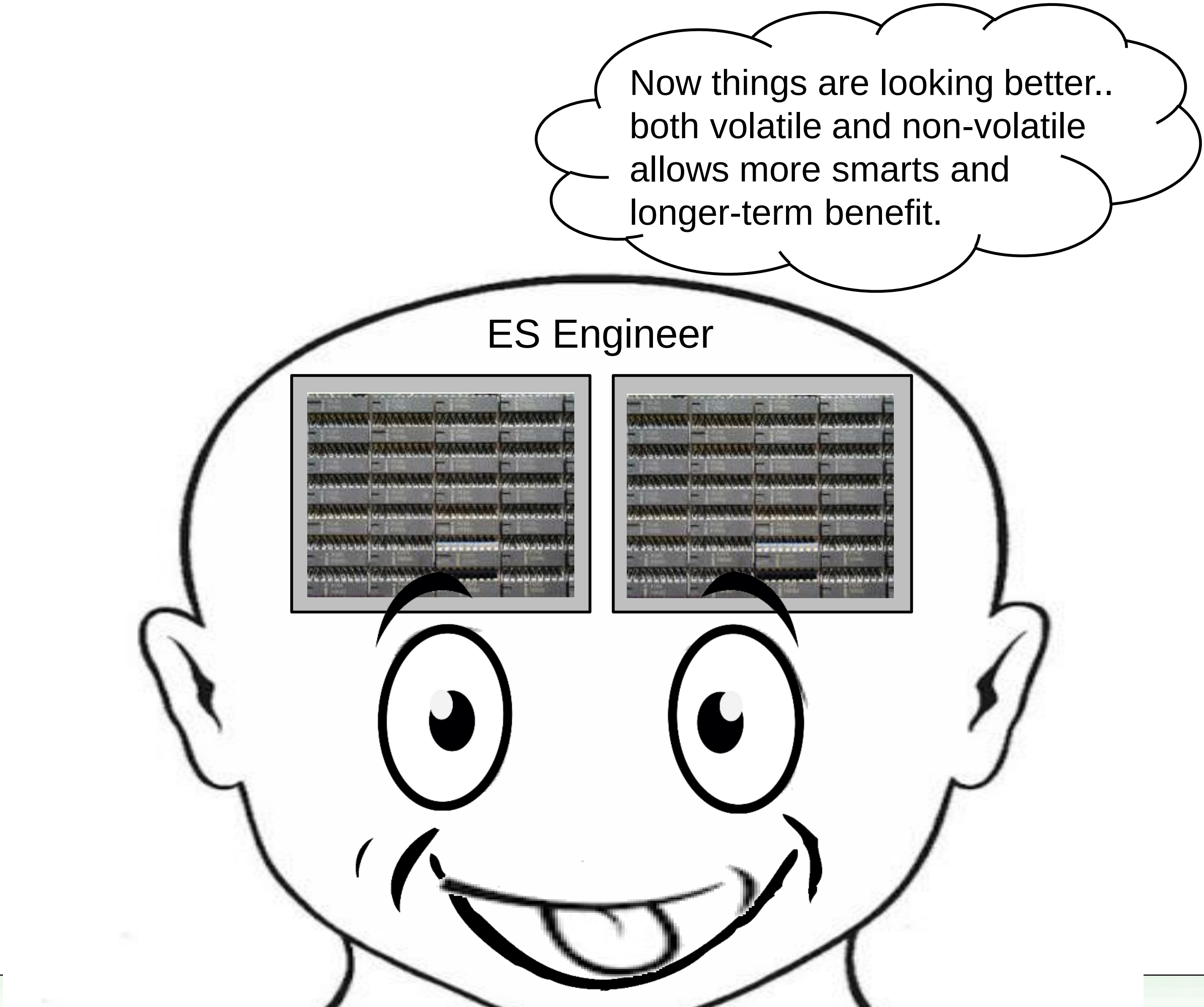
Volatile Memory

- BRAM or Block RAM
  - This refers to a small block of RAM (a few Kilobytes) integrated within the FPGA (connected some LBs)
  - Generally only found in higher-end FPGAs (e.g. 16Kb takes ~ 256K transistors if not more for connection and addressing logic)
  - Block SRAM is more common and easier to use; the FPGA may include Block DRAM
  - Generally can be set to RAM or ROM
  - As ROM it can be used as a (big) LUT
  - Usually not directly accessible form outside the FPGA (need to provide circuitry / softcore and comms protocol to access it from a PC)

Volatile memory

- Under development
  - Z-RAM : Zero-capacitor RAM
    - Single transistor
    - Higher density than DRAM
    - Although it is called zero-capacitor, the capacitor is actually there in the form of a “floating body effect” caused by the transistor substrate
    - See: http://www.innovativesilicon.com/

Non-Volatile Memory types


Now things are looking better.. both volatile and non-volatile allows more smarts and longer-term benefit.
ES Engineer

Non-Volatile memory

- Trusty old ROM and EEPROM
  - Still widely used as it is highly robust
  - Current versions store large amounts of data
  - Fairly simple technology (i.e. fused connections) and (in EEPROM ability to fuse and then program/un-fuse connections)
- Usually ROM is slower than RAM
- Shadowing ROM (i.e. copy to RAM) to make it faster – especially for EEPROMs
- EEPROM very slow write; faster read

Non-Volatile Memory

- Flash memory
  - Can be electrically erased and programmed
  - High capacity (e.g., millions of bytes/chip)
  - Needs to be programmed one block at a time (~8Kb / block)
    - Erased (all bits in block set to 1)
    - Programmed one block at a time
  - Memory wear
    - Limited to about 100,000 erase – write cycles
    - Usually a file system (e.g. ext3) will keep track of bad sectors (i.e., mark deteriorated blocks). But this deterioration might happen a certain time after the erase and write is complete and verified.

NAND Flash memory model

(don't need to know for examination)

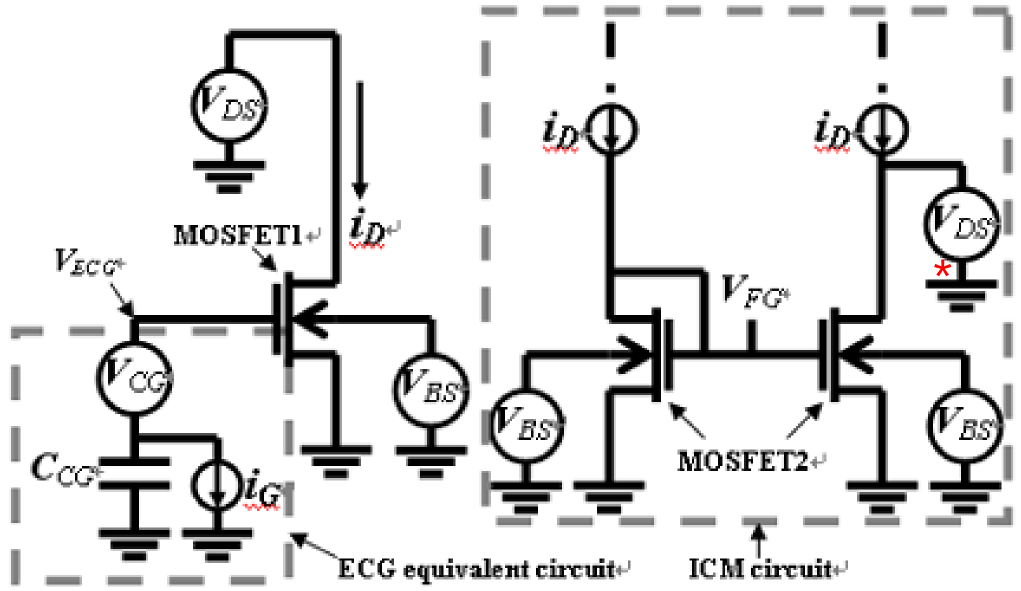


Image source: IEEE Electron Device Letters, Vol. 26, No. 8, AUGUST 2005, pg 564 Available at: http://koasas.kaist.ac.kr/bitstream/10203/1570/1/01468223.pdf

The above diagram provides a macro circuit model for a single flash memory cell, showing a *Effective-Control-Gate* (ECG) equivalent circuit and the *Ideal-Current-Mirror* (ICM) used to calculate the floating gate (FG*) voltage. MOSFET1 is the equivalent N-MOSFET model of a flash memory cell, and MOSFET2 is the model of a N-MOSFET test structure that is identical with the flash memory cell (excluding the short between FG and CG).

Using Memory in Verilog

EEE4120F

Using memory in Verilog

- Very simple approach:
  - (NB this isn't explicitly interfacing to a RAM chip or even emulating a memory chip)

```
parameter DATA_WIDTH = 8;   // word size of your memory
parameter RAM_DEPTH  = 128; // amount of words you want

reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1];
```

And there you go; you can now do operations such as:

```
mem[addr] = dataword;              dataword =  mem[addr];
```

(write to memory)   (read from memory)

Setting up a Memory module

- The array access technique used in the previous slide you probably already knew… and that's fine if you don't need to use much data (e.g. < 32Kb)
- That array will be more like cache memory; it will be fast, tightly coupled, and eat lots of your logic elements.
- What we often need is a memory module that may connect to external memory (i.e. on the platform but not in the FPGA) or large chunks of dedicated memory in the FPGA chip.

Memory Control Unit
(part 1 of 2)

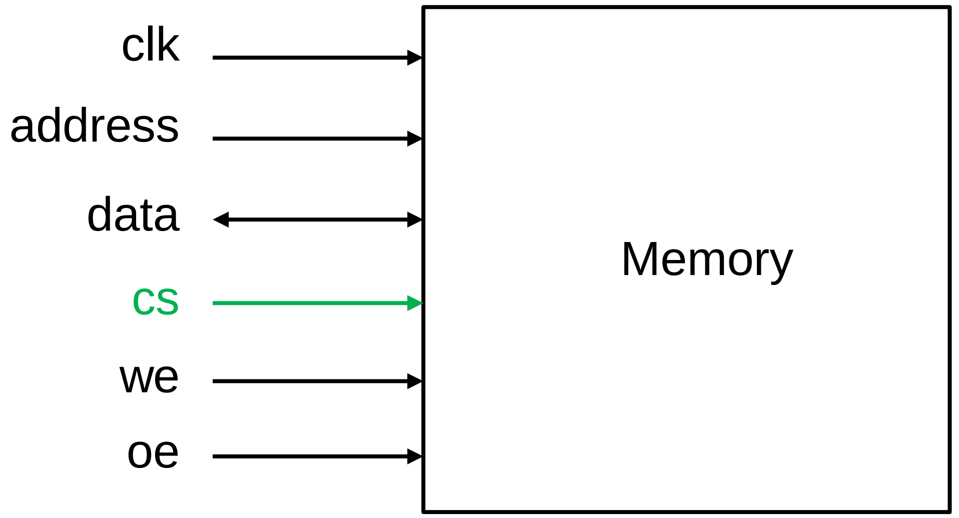
EEE4120F

The Memory Control Unit

A memory module or memory control unit (MCU) is used to wrap access to a memory device. It is usually going to use an implicit or explicit handshaking communication, as the access to the memory might not be done at the same speed as the FPGA is clocked. If you are using SRAM it can handle high speed, but, being possibly on a different chip, the memory and your FPGA logic might not be perfectly synchronized.
In these examples we are not actually using external memory, rather BRAM or BROM but we may want multiple modules accessing the same memory, and that is where a memory control unit is needed.

Memory Control Unit Interface

This is a usual (implicit handshaking) interface for a memory control unit



Note: The CS port isn't necessarily needed.

The oe line is to be able to decide if data is latched to the output data or not

Explanation of ports:

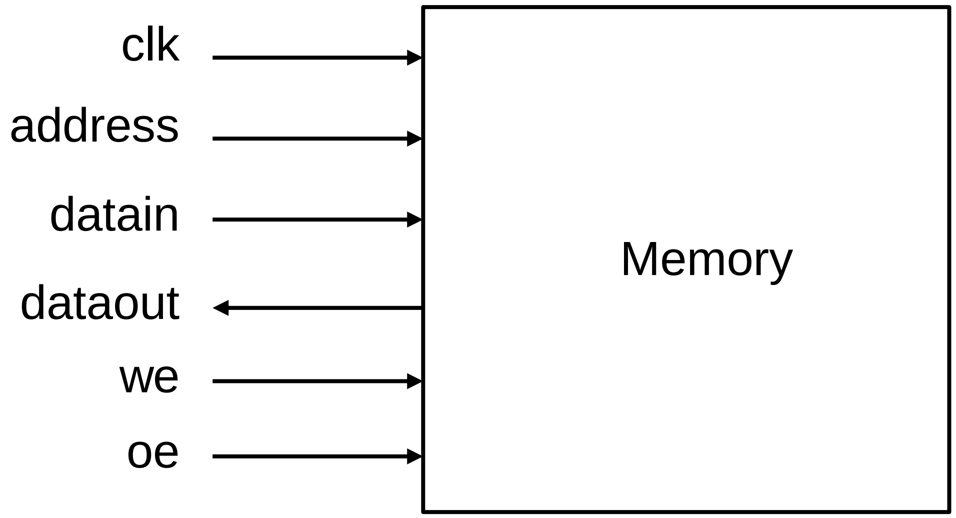
clk : clock input
address : address of memory to access
data : data word (bi-directional / inout)
cs : chip select (i.e. chip ignores inputs if cs=0)
we : write enable / read enable (if RAM chip)
oe : output enable (a safety precaution to prevent data from returned)

https://www.edaplayground.com/x/2gb7

Memory Control Unit Interface

MCU with dedicated input and output data lines (often easier & more reliable).

This is a usual (implicit handshaking) interface for a memory control unit



Explanation of ports:

clk : clock input
address : address of memory to access
datain : data input
dataout : data output
cs : chip select (i.e. chip ignores inputs if cs=0)
we : write enable / read enable (if RAM chip)
oe : output enable (a safety precaution to prevent data from returned)

Memory Control Unit in Verilog

```
// RAM control unit
module ramcu (
  clk     , // Clock Input
  address , // Address Input
  data    , // Data bi-directional
  cs      , // Chip Select
  we      , // Write Enable/Read Enable
  oe        // Output Enable
  );
  // Setup some parameters
  parameter DATA_WIDTH = 8;  // word size of the memory
  parameter ADDR_WIDTH = 8;  // number of memory words, e.g. 2^8-1
  parameter RAM_DEPTH = 1 << ADDR_WIDTH;  // i.e. RAM_DEPTH = 2^ADDR_WIDTH

  // Define inputs
  input clk, cs, we, oe;
  input [ADDR_WIDTH-1:0] address;

  // data is bidirectional
  inout [DATA_WIDTH-1:0]  data;
  // Private registers
  reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1]; // Set up the memory array
  reg [DATA_WIDTH-1:0] r_data;              // copy of data value to return
  reg r_oe;            // delayed oe, r_oe updates only when rdata updated

  // START OF OPERATION: ///////////////////////////////////////////////
  // Tri-State Buffer control:
  //  The data item is defined as an inout, only when oe, output enabled
  //  is high, should it send a value to data, otherwise it should keep
  //  the data port linked to high impedence (z) so as not to drive a value.
  //  i.e. output to data happens when oe = 1 & cs = 1 & we = 0

  // Write to memory when: we = 1 & cs = 1
  always@ (posedge clk)
  begin
   if (cs)
   begin
     if (we) mem[address] <= data;
     r_data <= mem[address];
   end
   r_oe   <= oe;
  end

  assign data = (oe && cs && !we)? r_data : 8'bz;

endmodule // end ramcu
```

Try it at: https://www.edaplayground.com/x/2gb7

Memory Control Unit Test Bench

```
// EEE4120F Memory Control Unit Example
// Testbench for the ramcu RAM control unit

module ramcu_tb ();
  wire [7:0] data;       // this is the connection to ramcu data port
  reg  [7:0] data_value; // need a register to store value to send on a write
  reg clk, cs, we, oe;
  reg [7:0] addr;
  // Instantiate the module to be tested
  ramcu ramcu_uut(clk,addr,data,cs,we,oe);

  assign data = !oe ? data_value : 'bz;

  initial begin
    // set up initial conditions
    clk   = 0; cs    = 0; we    = 0; oe    = 0; addr  = 0;

    $display("clk cs we oe addr data");
    $monitor("%b   %b  %b   %b %03d  %03d",
             clk,cs,we,oe,addr,data);

    // try doing a clock change just when nothing should happen (cs=0)
    #5 clk = ~clk; #5 clk = ~clk;

    // let is do a memory write...
    // note that you need to deselect the chip before setting up data
    $display("write 99 to mem[1]");
    oe   = 0; we   = 1; cs   = 1; addr = 8’d1; data_value = 8'd99;
    #5 clk = ~clk; #5 clk = ~clk; // do a clock pulse

    // now disconnect data_value from data line
    $display("write junk into toplevel data_value buffer");
    cs  = 0; we  = 0; oe = 0; data_value = 8'd2;
    #5 clk = ~clk; #5 clk = ~clk; // do a clock pulse

    // let's read the value back
    $display("check reading back data");
    addr = 8’d1; cs   = 1;  we   = 0; oe   = 1; // ask module to write out data
    #5 clk = ~clk; #5 clk = ~clk;    // do a clock pulse (reads the data)
    // you should now have back the value that was written to memory

    // disable chip select and write some more junk to local buffer
    $display("write more junk into toplevel data_value");
    cs = 0; data_value = 8'd123;

    #5 clk = ~clk; #5 clk = ~clk; // do a clock pulse to see what happens
  end // end initial
endmodule
```

```
// You could add this to the end of the code to see
// what it is memory:

// OPTIONAL: repeat loop to dump part of the memory
addr = 0;
#5
$display("dump some memory:");
repeat (5)
 begin
  $display("mem[%d]=%d",addr,ramcu_uut.mem[addr]);
  addr = addr + 1;
 end
```

Try it at: https://www.edaplayground.com/x/2gb7

End of Lecture

Any Question??

Disclaimers and copyright/licensing details

Image sources:

man working on laptop – flickr
scroll, video reel, big question mark – Pixabay http://pixabay.com/ (public domain)
some diagrammatic elements are from Xilinx ISE screenshots
https://commons.wikimedia.org/wiki/Category:Images (creative commons)