EEE4120F

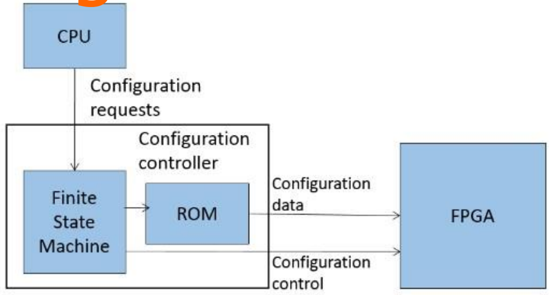
High Performance Embedded Systems

Lecture 20

More Verilog. Configuration Architectures. RC Building Blocks (IP Cores), basic handshaking, latches and other interface ingredients

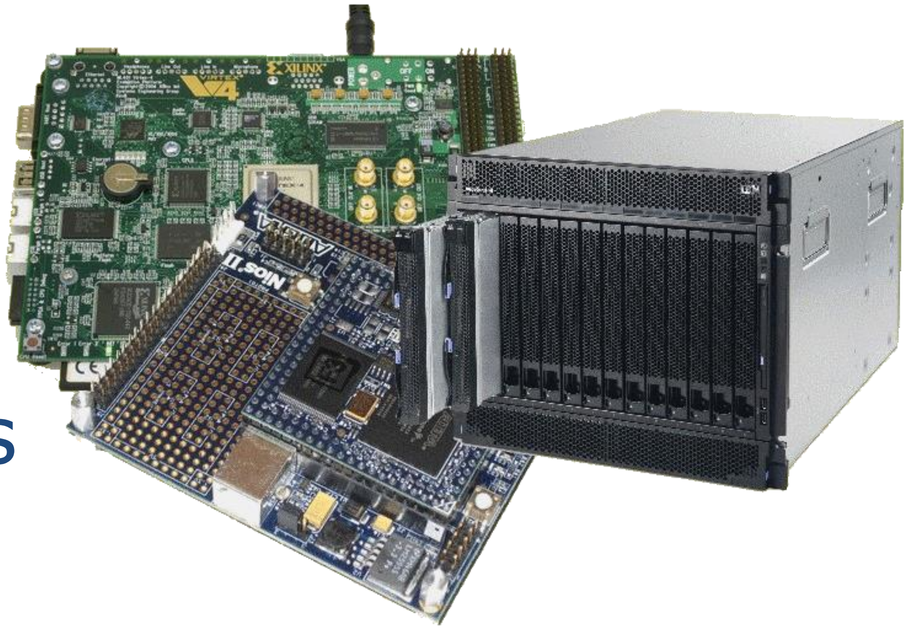
(slide 46 onwards not in syllabus)

Lecturer:
Simon Winberg



Lecture Overview

- When to use *assign*
- Blocking & non-blocking simulation and potential pitfalls
- The unconditional always
- Configuration architectures
- Digital signals, Interface basics, Using latches
- Large & Small RC platform case studies



Purpose of this lecture

- This lecture provides theories may assist you in the design of your YODA system, how subsystems you may be using in your design might cooperate in a robust and dependable way.
- (The configuration architectures section is largely theory you should know about, connecting with thoughts of designing an FPGA into a larger system, but which you aren't like to use further in this course)

(This lecture includes syllabus topics that is part of HPES, but understandably more immediate practical use of this would be more towards students who select to do a FPGA-based YODA project).

Towards sharpening your digital logic design lightsabre

When to use assign?

- Assign is a continuous driver, it essentially links some source on the RHS to a wire
- Assign is
  - Used outside an always blocks
  - Has a net or wire on the LHS
- The syntax is:

  always *netname* <= *RHS_expression*
- Assign is used for driving a source (on the RHS) to some wire or net type element on the LHS.
- The wire on the LHS changes value in response to the source driving it, so whenever any source on the RHS changes, the RHS expression is evaluated and assigned to LHS.

Using blocking and non-blocking assignments in simulation

- While one typically things to use blocking and non-blocking assignments in implementing cores, it is also useful (almost essential sometimes) for simulation. Consider these examples:

```
module blocking ();
 reg a, b, c, d , e, f ;
 // Blocking assignments
 initial begin
   $monitor("%g a=%b b=%b c=%b",
        $time,a,b,c);
   a = #10 1'b1;
   // sim assigns 1 to a at time 10
   b = #20 1'b0;
   // sim assigns 0 to b at time 30
   c = #30 1'b1;
   // sim assigns 1 to c at time 60
 end
endmodule
```

iverilog output:

```
0 a=x b=x c=x
10 a=1 b=x c=x
30 a=1 b=0 c=x
60 a=1 b=0 c=1
```

https://www.edaplayground.com/x/25Yx

```
module nonblocking();
reg a, b, c ;
// Nonblocking assignments
initial begin
  $monitor("%g a=%b b=%b c=%b",
    $time,a,b,c);
  a <= #10 1'b1;
  // sim assigns 1 to a at time 10
  b <= #20 1'b0;
  // sim assigns 0 to b at time 20
  c  <= #30 1'b1;
  // sim assigns 1 to c at time 30
end
endmodule
```

iverilog outpt

```
0 a=x b=x c=x
10 a=1 b=x c=x
20 a=1 b=0 c=x
30 a=1 b=0 c=1
```

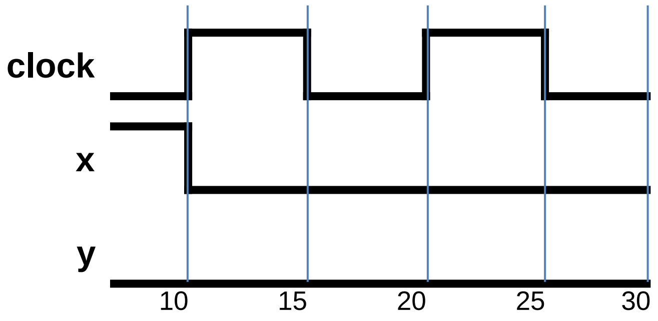
https://www.edaplayground.com/x/ars

Potential pitfalls

- If blocking assignments are not properly ordered, a race condition could occur
  - A race condition is an undesirable situation that may occur when a device attempts to perform two or more operations at the same time, but due to the nature of the device the operations must be done in the proper sequence to be completed correctly.
- When blocking assignments are scheduled to execute in the same time step (e.g. in different always blocks), then order of execution is unknown

Example:
```
always @(posedge clock)
    x = y;

always @(posedge clock)
    y = x;
```



Case where the first always block execute first… but maybe just luck.

The unconditional always

- The always block does not necessarily need to always have a sensitivity list. In such a case the always block is activated continuously
- This is commonly used for simulation, e.g. to generate a clock…

  let’s see a quick example

Example: simulating a clock

let's first implement a module that we want to send a clock to

```
// Implement a simple 4-bit counter
module counter4 (clock,reset,count);
  input clock;
  input reset;
  output reg[3:0] count;
  always @(posedge clock)
    begin
      if (reset)
         count = 0;
      else
         count = count + 1;
    end
endmodule
```

Run example at: https://www.edaplayground.com/x/28dF

Example: simulating a clock

Now let’s generate some code that first initializes the counter and then sends a clock to it. The clock will have a period of 100 simulation units, and the simulation will terminate after 1000 simulation units.

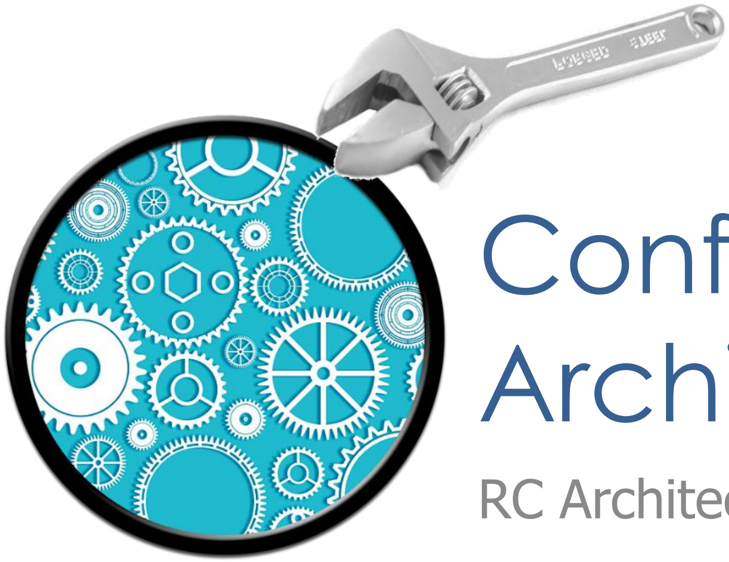
```
// generate a simulated clock that is connected to a 4-bit counter
module sim_clockgen (output clock);
  parameter half_cycle = 50;  // specify 1/2 clock period
  parameter max_time = 1000;  // max sim units to run for
  reg clock;        // the simulated clock
  reg reset;        // the simulated reset line
  reg[3:0] count; // counter value to link to counter
 initial  // let's first reset the clock
  begin
   // tell the simulator what wires we want to monitor
   $monitor("%g clock=%b count=%d",$time,clock,count);
   clock <= 0;
   reset <= 1; #200
   reset <= 0;
  end
  … // …
```

Example: simulating a clock

In this simulation we use an unconditional always to generate a clock

```
// generate a simulated clock that is connected to a 4-bit counter
module sim_clockgen (output clock);
… // …
 // here is an unconditional always that continuously runs
 always
  begin
   #half_cycle clock = ~clock;
  end
 // instantiate the 4-bit counter
 counter4 mycounter (clock,reset,count);

 initial // tell the simulator when to end
  #max_time $finish;
endmodule
```



Configuration Architectures

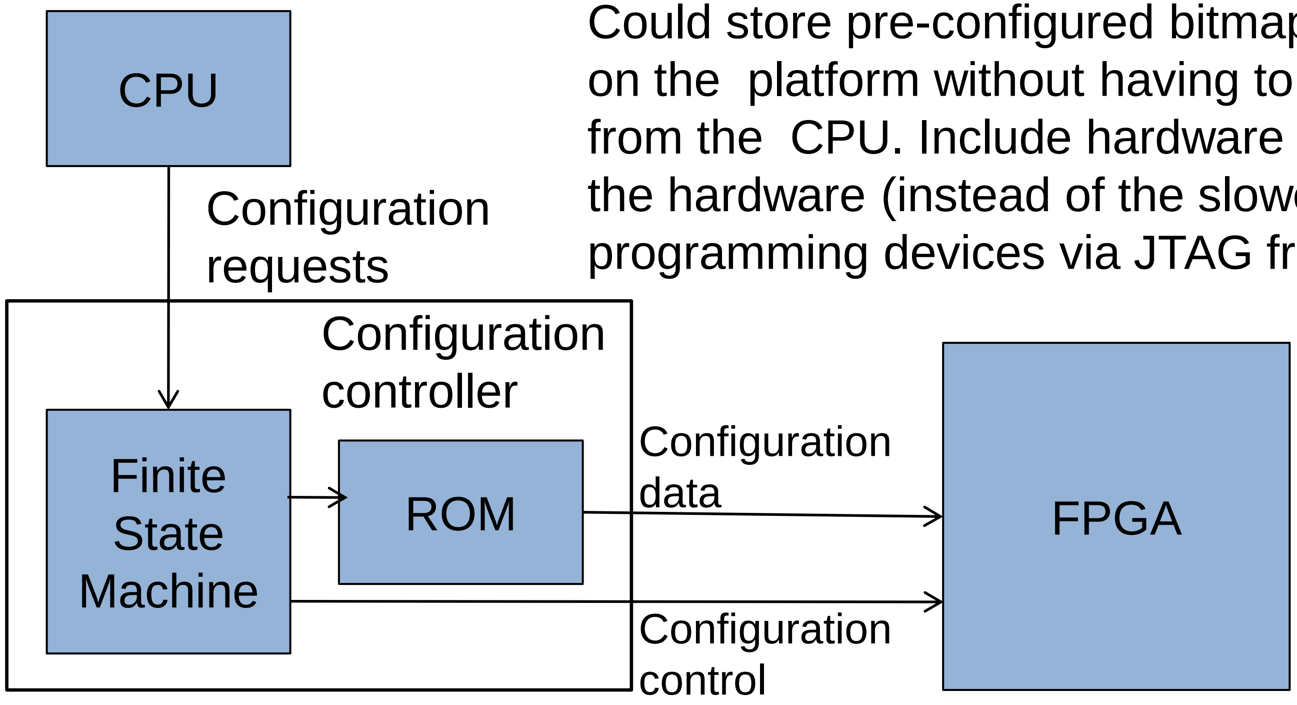
RC Architecture

Why cover this topic?

This topic is covered so that you have some understanding of what architectural aspects, outside of the FPGA, is needed in order to enable the programming of an FPGA. Generally, when an FPGA gets turned on, it has no ‘program’, or more accurately, the configuration of its interconnects is not set up; at startup all its interconnects are linked to high impedance to ensure the FPGA doesn’t blow up when you first turn it on. The configuration hardware connects to the FPGA in order to get the FPGA set up and then to start running useful combinational logic.

Configuration Architectures

- Configuration architecture =
  - Underlying circuitry that loads configuration data and keeps it at the correct locations



Could store pre-configured bitmaps in memory on the platform without having to send it each time from the CPU. Include hardware for programming the hardware (instead of the slower process of e.g., programming devices via JTAG from the host)

Adapted from Hauck and Dehon Ch4 (2008)

Configuration Architectures

- Larger systems (e.g., the VCC*) may have many FPGAs to be programmed)
- Models:
  - Sequentially programming FPGAs by shifting in data
  - Multi-context – having a MUX choose which FPGA to program

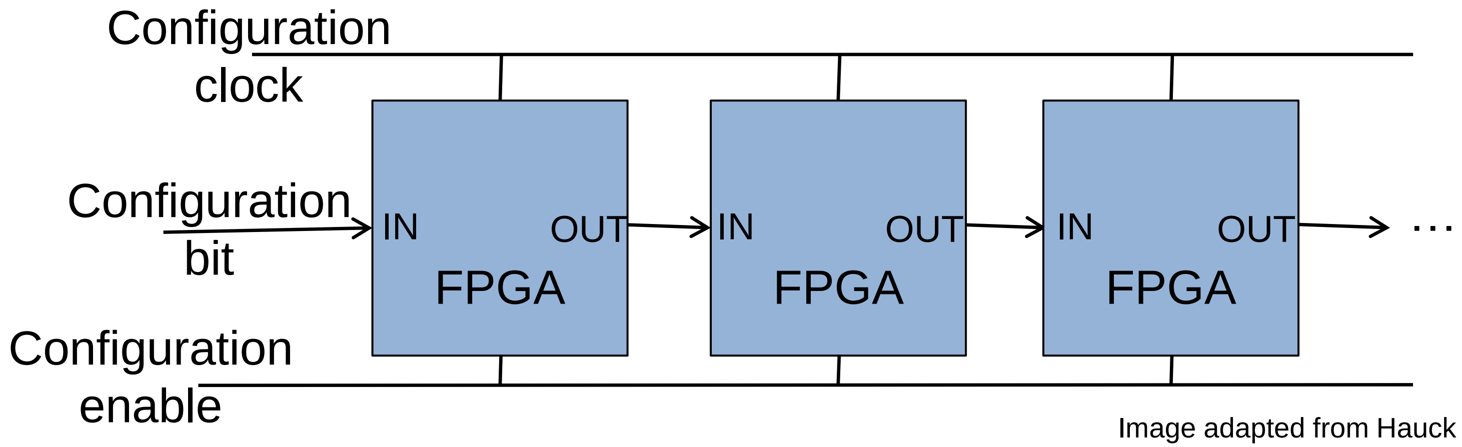
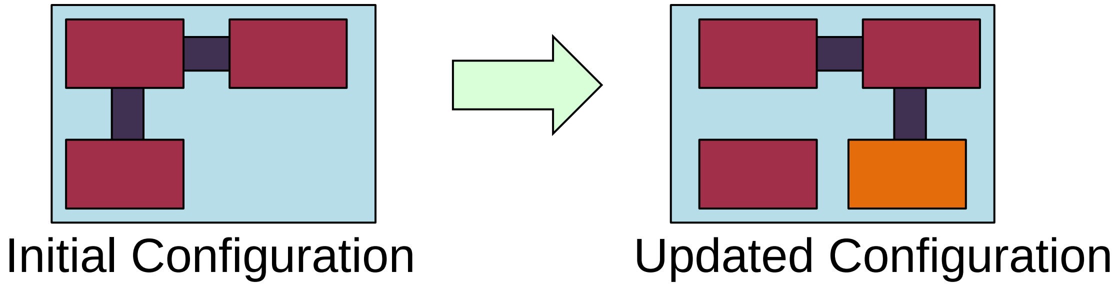


Image adapted from Hauck and Dehon Ch4 (2008)

* As in the Virtual Computer Corporation's (VCC) Virtual Computer (or Virtual Compacted Computer) architecture. See Case Study slides at end of slideshow.

Configuration Architectures

- Partially reconfigurable systems
  - Not all configurations may need entire chip
  - Could leave parts of chips unallocated
  - Partial configuration decreases configuration time
  - Modifying part of a previously configured system
    - E.g., a placement and routing configuration based on a currently configured state



Configuration Architectures

- Block configurable architecture
  - Not the same as “logical blocks” in an FPGA
  - Relocating configurations to different blocks at run time also referred to as “swappable logic units” (SLUs)
- Example: SCORE* relocatable architecture in which configurable blocks are handled in the same way as a virtual memory system

* Capsi & DeHon and Wawrzynek. “A streaming multithreaded model” In Third workshop on media and stream processors. 2001

Optional Additional Reading
(for configuration architectures)

- Reading
  - Hauck, Scott (1998). “The Roles of FPGAs in Reprogrammable Systems” *In Proceedings of the IEEE.* 86(4) pp. 615-639.

This optional reading will not examined

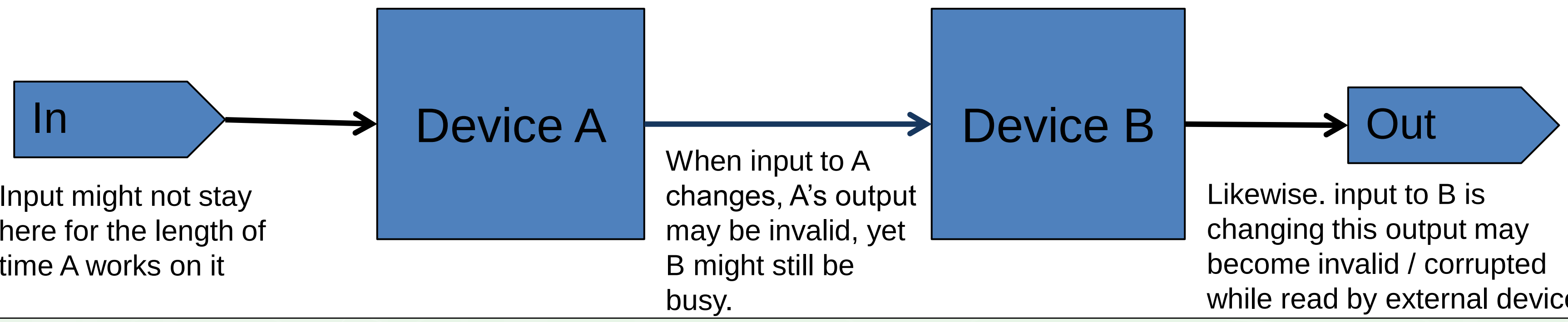
RC Building Blocks: Digital Signals and Data Transfers

Reconfigurable Computing

Developing reusable gateware components / IP Cores

Overview of digital signals

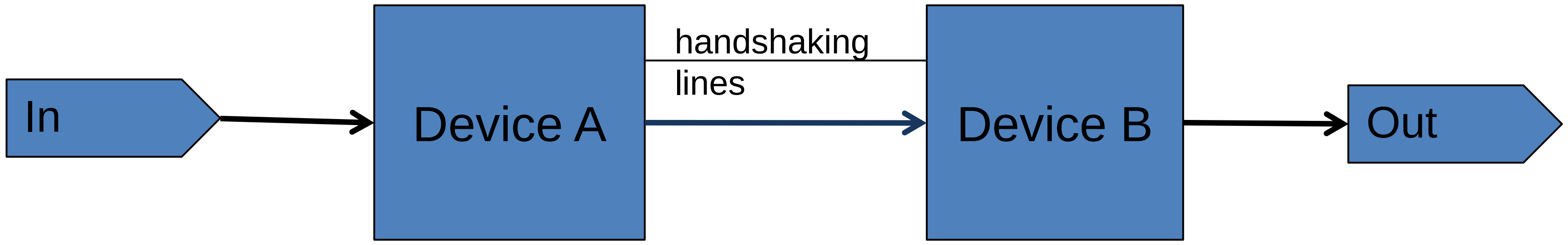
- Although one of our main objective is High Performance and parallelized operation, there are still sequential issues involved… for, example a device B waiting for a device A to provide input.
- Furthermore the input to device A might disappear (become invalid) before device A has completed its computations.


In
Device A
Device B
Out
Input might not stay here for the length of time A works on it
When input to A changes, A's output may be invalid, yet B might still be busy.
Likewise. input to B is changing this output may become invalid / corrupted while read by external device

Overview of digital signals

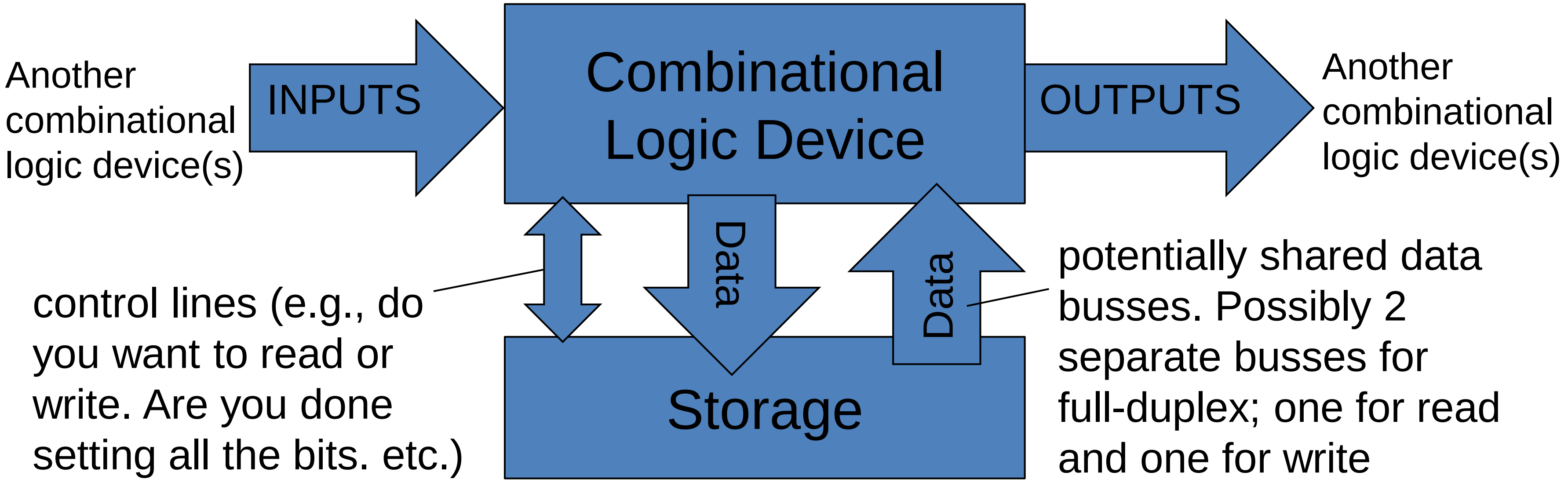
- There are various issues involved such as:
  - How does device A know when new data has arrived?
  - How does device B know when device A has completed?
  - What if both devices need to be clocked, but aren't active all the time?
  - What if you want to share address and data lines?

.. That is where 'handshaking' can come to the rescue



Digital logic modular design issues

- A sequential logic system typically involves two parts:
  - Storage (aka "bistable" device)
  - Combinational logic (OR, AND, etc. gates)


Another combinational logic device(s)
INPUTS
Combinational Logic Device
OUTPUTS
Another combinational logic device(s)
Data
Data
Storage
control lines (e.g., do you want to read or write. Are you done setting all the bits. etc.)
potentially shared data busses. Possibly 2 separate busses for full-duplex; one for read and one for write

Handshaking – making sure the data gets there

Reconfigurable Computing

Interface Handshaking basics

- Handshaking is a means to ensure reliable transfer of data and/or control between two devices that could be far apart (long latencies between pins) or could be operating at different clock frequencies.
- Essentially, handshaking provides additional pins to synchronize the transfer of data, so that data is sent when the receiver is ready.

Handshaking is essentially a form of blocking or synchronous communication (i.e. leading from the lecture series on parallel system design but now in the context of combinational logic designs).

Handshaking

- For effective data exchange:
  - Sender needs to first know when receiver is able to receive data, *and* when the data has been successfully received.
  - Receiver needs to know when the data is ready to be sent to it, *and* when the sender has determined that the receiver has acquired the data.

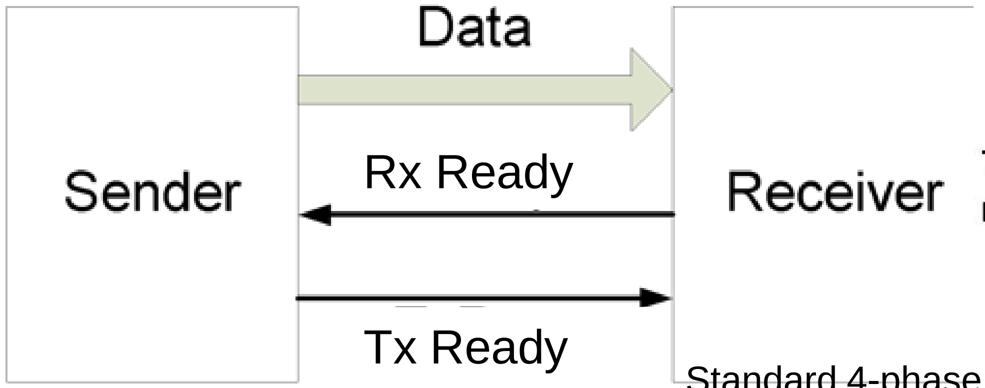
Handshaking

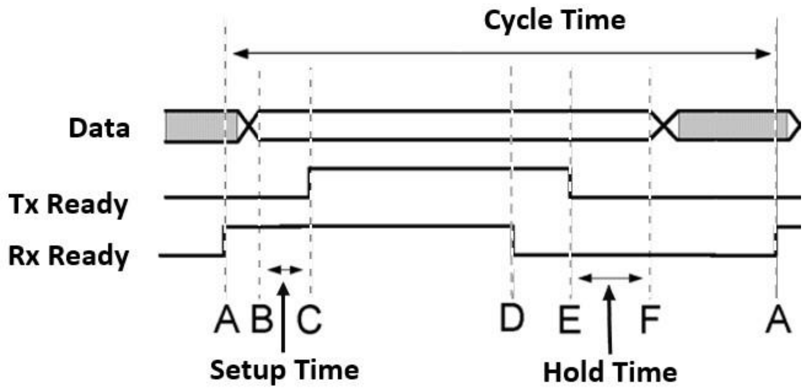
- There are many types of handshaking, but they are generally either:
  - Explicit handshaking
  - Implicit handshaking

Information based on Digilent training resources available at: https://reference.digilentinc.com/learn/courses/unit-3/start

Explicit Handshaking

- Explicit handshaking, basic 4-phase handshake:
  - This form of handshaking uses dedicated control signals (e.g. Rx Ready and Tx Ready) to indicate the impending action from the sender and readiness of the receiving device.
  - The receiving device must indicate when it is ready to receive data, and when it is done.
  - The sending device must strobe (tell) the receiving device when new data is ready for it, and when the data is no longer available.

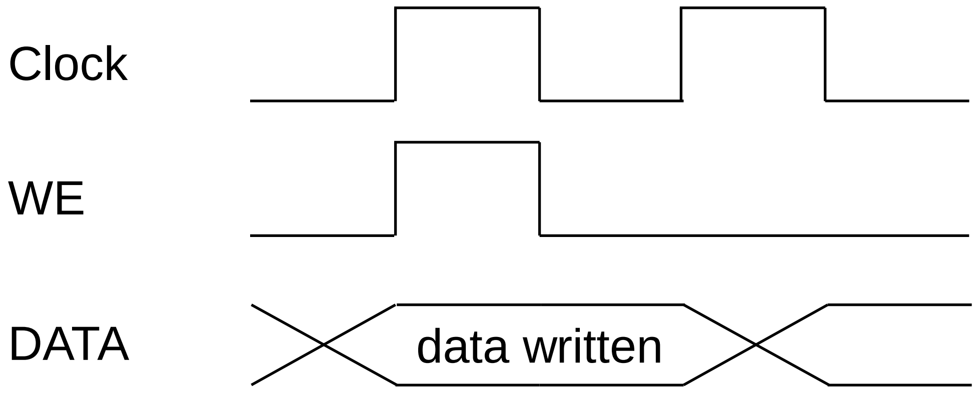




Standard 4-phase handshaking

Implicit Handshaking

- In terms of implicit handshaking it usually an assumption that the receiver is ready to receive provided timing constraints (such as synchronization with a clock) is met.



E.g. devise sending data assume receover will be ready. Data must remain on the bus for at least one clock

Digital Signal Interfaces

- Generally need the following
  - Address bus
  - Data bus
  - Control lines
    - Chip / Device select lines (CS)
    - Write enable lines (WE)
    - Read enable lines (RE)

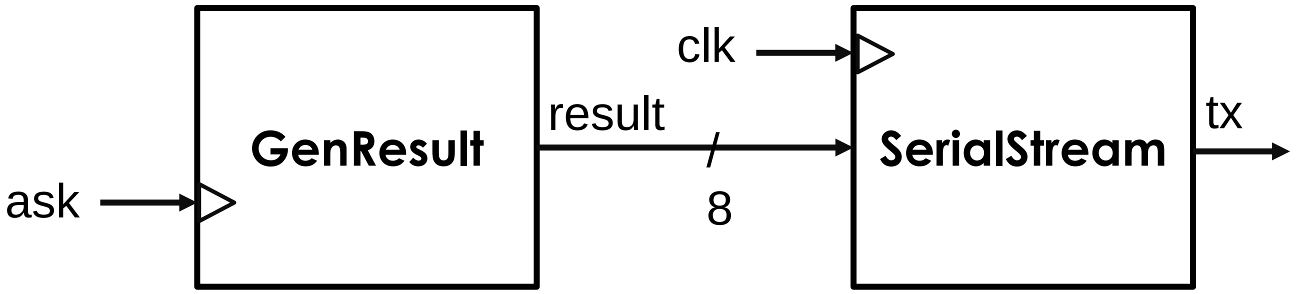
Looks familiar to Embedded Eystems I/O issues?
Well it is! But this time *you* are deciding the control lines. ☺

Interface / Handshaking Standards

- Next lecture we look at a selection of interfacing standards for developing your own reusable IP cores.
- We also look at standard memory interfaces and DMA transfers
- See SerialStream example to reinforce the handshaking topics presented here
- For now though we continue on with some latching and signal capture which are essential ingredients in reliable gateware interfaces

Example: Handshaking in Verilog

- Consider that you have system in which a module produces an 8-bit result. But this result, provided as a bus output, needs to stream this byte out as a bit stream (that could be sent to an UART).
- The block diagram of this system is provided below illustrating what is needed



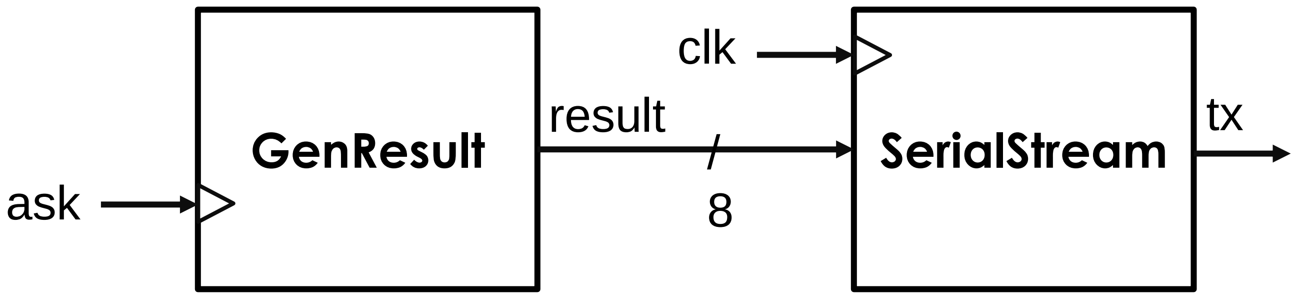
A positive edge on the 'ask' input causes GenResult to produce a result which is then sent to SerialStream

The 8-bit result of GenResult is sent to SrialStream to convert to serial data.

When SerialStream receives the data it sends bit 0 .. 7 of result, each bit on a positive edge of clk. SerialStream should not receive any other input until is has completed this task.

Example: Handshaking in Verilog

- BUT there are problems posed in the initial design as is:



A positive edge on the 'ask' input causes GenResult to produce a result which is then sent to SerialStream

The 8-bit result of GenResult is sent to SrialStream to convert to serial data.

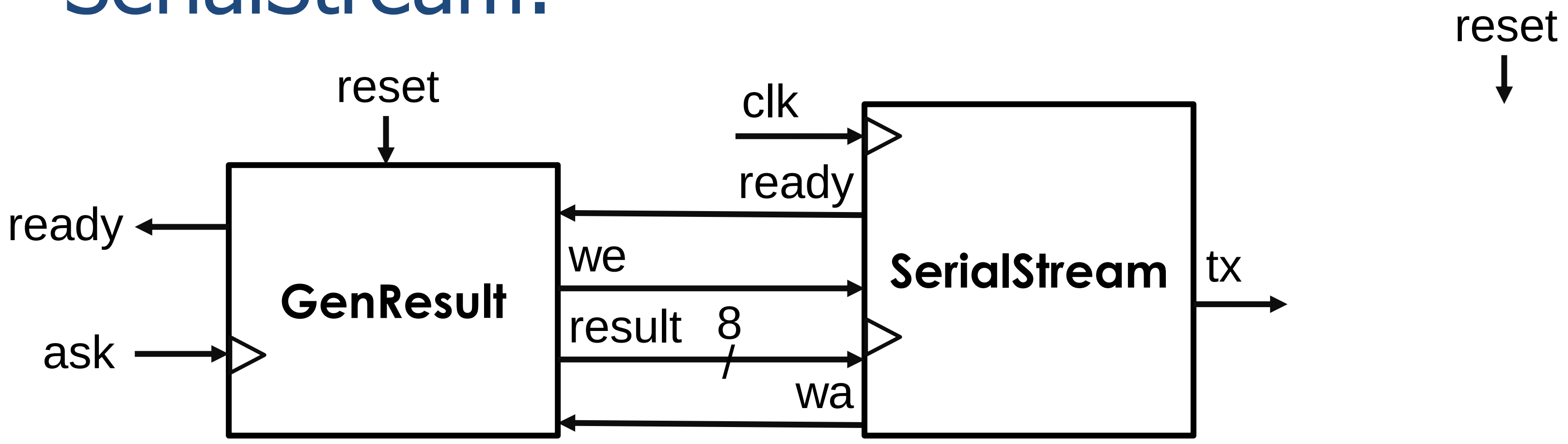
When SerialStream receives the data it sends bit 0 .. 7 of result, each bit on a positive edge of clk. SerialStream should not receive any other input until is has completed this task.

This is, at present, not a very robust design what happens if GenResult is send an 'ask' while it is still busy? Does the asker need to know that GenResult has send on the result? What of SerialStream; it should now allow a result to be sent before it is ready to send it. And, how will SerialStream know when it has received a new input, e.g. if you want to send 0 and another 0, how will it know that you want to send two 0s and not just one? No, in summary this initial version of the design is not robust.

Propose a better design that uses Handshaking to make it more robust….

Example: Handshaking in Verilog

- More robust design for GenResult and SerialStream:



The ready line is high when GenResult is ready to be sent an 'ask'. A positive edge on the 'ask' input causes GenResult to set ready low, and later produces a result which is then sent to SerialStream

GenResult waits until SerialStream has its ready line high before it sends result. To send a result, write enable (we) is set low. Then result is set. Then we is set high. When serial stream sets ready low and send a write ack (wa), GenResults sets we low (which confirms that the result has been sent to SertialStream). GenResult only sets its ready back to high when SerialStream sets its ready to high.

Initially SerialStream has its ready line high. When we it receives a we it reads data from the result input and then sets ready low. It then sends bit 0 .. 7 of result, each bit on a positive edge of clk. SerialStream sets ready back to high after the last bit has been sent.

Understandably, there are still some shortcomings to this design (e.g. GenResult could do with a timeout incase SertialStream gets stuck), but it is a lot more robust than the earlier design.

GenResult

```
module GenResult (reset, ask, ss_ready, ss_wa, ss_we, ready, result);
  input  rese, ask;
  input  ss_ready, ss_wa;
  output reg ss_we;
  output reg ready;
  output reg [7:0] result;

  // private registers
  reg handle_ask;

  // wait for a reset
  always@ (reset)
    begin
      result <= 10;
      ss_we  <= 0;
      ready  <= 1;
      handle_ask <= 0;
    end

  always@ (ask)
    begin
      if (handle_ask == 0)
        begin
          if (ss_ready == 1)
            begin
              handle_ask <= 1;
              ready  <= 0;
              result <= result + 1;
              ss_we  <= 1;
            end
        end
    end

  always@ (posedge ss_wa)
    begin
      // assume that SerialStream is now busy with
      // the request and will raise ready once ready again.
      ss_we <= 0; // handshaking, that the write enable has been done
    end

  always@ (posedge ss_ready)
    begin
      ss_we      <= 0; // just in case we missed the ack
      handle_ask <= 0;
      ready      <= 1;
    end
endmodule
```

SerialStream

```
module SerialStream (clk, reset, we, result, ready, wa, tx);
  input clk, reset;
  input we;
  input [7:0] result; output reg ready; output reg wa; output reg tx;

  // internal registers
  reg start; reg endoff;
  reg [8:0] mask; reg [7:0] sendb;
  reg lower_wa; // used to set wa back to 0 after a clock

  always@ (reset)
    begin
      start  <= 0;   ready  <= 1;
      endoff <= 0; tx     <= 0;
      wa     <= 0;  lower_wa <= 0;
      mask  <= 9'b0;
    end

  // wait for a we to arrive...
  always@ (posedge we)
    begin
      ready <= 0;   start <= 1;
      mask  <= 1;  sendb <= result;
      wa    <= 1;    lower_wa <= 1;
    end

  // wait for a we to arrive...
  always@ (posedge clk)
    begin
      if (endoff == 1)
        begin
          endoff <= 0;
          ready  <= 1;
        end else
        begin // send the next bit
          if (sendb & mask) tx = 1; else tx = 0; // not blocking assign
          mask = mask << 1;
          if (mask == 9'd256) endoff <= 1;
        end

      if (lower_wa == 1)
        begin
          wa       <= 0;
          lower_wa <= 0;
        end
    end
endmodule
```

You can access these files at: https://www.edaplayground.com/x/68ZH

Testbench for GenResult and SerialStream

```
// EEE4120F Example for demonstrating use of handshaking in Verilog

module toplevel_tb ();

  // Define registers that will be inputs and outputs to the modules being tested
  reg  clk;     reg  reset;    reg  ask;
  wire ss_ready;    wire ss_wa;
  wire ss_we;        wire ready;
  wire tx; // wire that connects to some transmit line
  wire [7:0] result;

  GenResult uut_gr    (reset, ask, ss_ready, ss_wa, ss_we, ready, result);
  SerialStream uut_ss (clk, reset, ss_we, result, ss_ready, ss_wa, tx);

  initial
    begin
      $display("reset ask ss_ready ss_wa ss_we ready result tx");
      $monitor("%b   %b     %b   %b       %d     %d     %d     %02d %b",
                clk, reset, ask, ss_ready, ss_wa, ss_we, ready, result, tx);

      // set up default values for data and control lines
      clk      = 0;
      reset    = 1;
      ask      = 0;

      // generate a clock pulse to make sure reset takes
      #5 clk = ~clk;

      // lower reset
      reset  = 0;
      #5 clk = ~clk;

      // ask to generate and set a value
      ask <= 1;
      #5 clk = ~clk;
      ask <= 0;
      #5 clk = ~clk;

      // run the clock for a while
      repeat (30)
      begin
        #5 clk <= ~clk;
      end

    end
endmodule
```

You can access these files at: https://www.edaplayground.com/x/68ZH

Testbench example output (when GenResult sends 11)

```
reset ask ss_ready ss_wa ss_we ready result tx
0     1     0     1       0     0     1     10  0
1     0     0     1       0     0     1     10  0
0     0     1     0       1     0     0     11  0
1     0     0     0       0     0     0     11  1
0     0     0     0       0     0     0     11  1
1     0     0     0       0     0     0     11  1
0     0     0     0       0     0     0     11  1
1     0     0     0       0     0     0     11  0
0     0     0     0       0     0     0     11  0
1     0     0     0       0     0     0     11  1
0     0     0     0       0     0     0     11  1
1     0     0     0       0     0     0     11  0
0     0     0     0       0     0     0     11  0
1     0     0     0       0     0     0     11  0
0     0     0     0       0     0     0     11  0
1     0     0     0       0     0     0     11  0
0     0     0     0       0     0     0     11  0
1     0     0     0       0     0     0     11  0
0     0     0     0       0     0     0     11  0
1     0     0     1       0     0     1     11  0
0     0     0     1       0     0     1     11  0
```

SerialStream starts sending here. It's sending in little endian, i.e. the lowest bit is sent first (which is 1 in the byte 00001011_2 that is sent.

As you can see ss_ready gets set when the 8 bits have been sent.

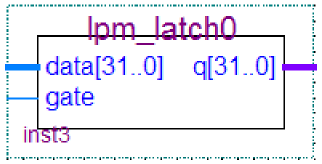
RC Building Blocks: Latching (capturing Signals)

Reconfigurable Computing

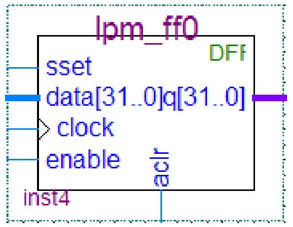
Digital Signal Capture and Storage

- In order to capture the signals, you need some storage
- Two basic types of storage:

Latches

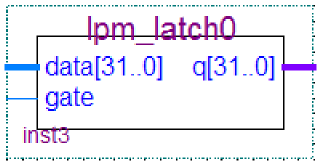


Flip-flops

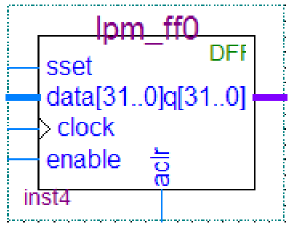


Difference between latch and flip-flop

- Latches Q = D
  - Changes state when the input states change (referred to as "transparency")
  - Can include an *enable input* bit – in which case the output (Q) is set to D only when the enable input is set.



- Flip-flop Q = D (Q changes when clocked)
  - A flip-flop only change state when the clock is pulsed.

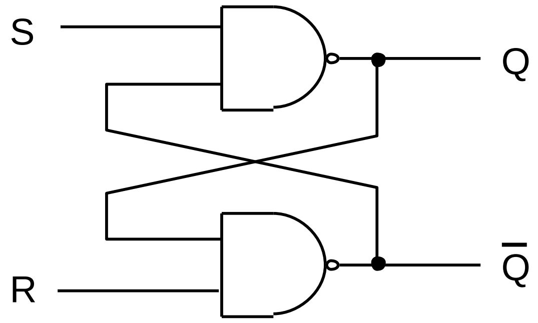


When to use a latch or a flip-flop

- Latches are used more in asynchronous designs `wire X, Y; assign X <= A;  assign Y <= B;`
- Flip-flips are used in synchronous designs `reg X, Y; X = A;  Y = B;`
- A "synchronous design" is a system that contains a clock

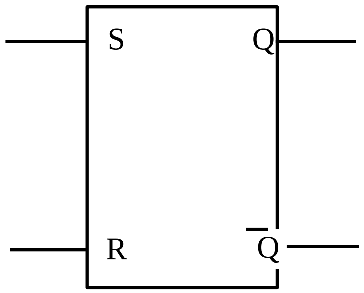
You can of course mix synchronous and asynchronous, and this is particularly applicable to parallel systems in which different parts of the system may run at different speeds (e.g., the main processor working at 1GHz and specialized hardware possibly operating asynchronously as fast as their composite pipelined operations are able to complete)

SR Latch



S-R Latch (set / reset latch)

S	R	$\overline{Q}$	Q
0	0	1	1
0	1	0	1
1	0	1	0
1	1	x	x



Symbol

S=R=1 is often called the memory or no-change state, by x I mean the line doesn't change from what it was before.

A basic NAND latch has two stable states:

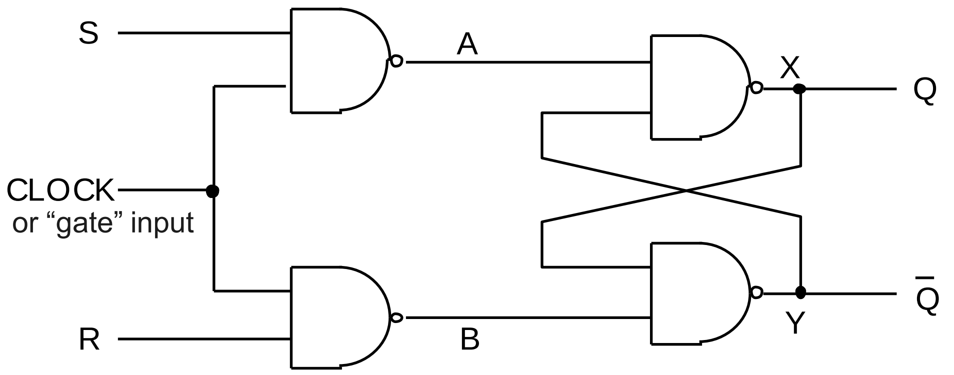
State 1 Q = 1 not_Q = 0

State 2 Q = 0 not_Q = 1

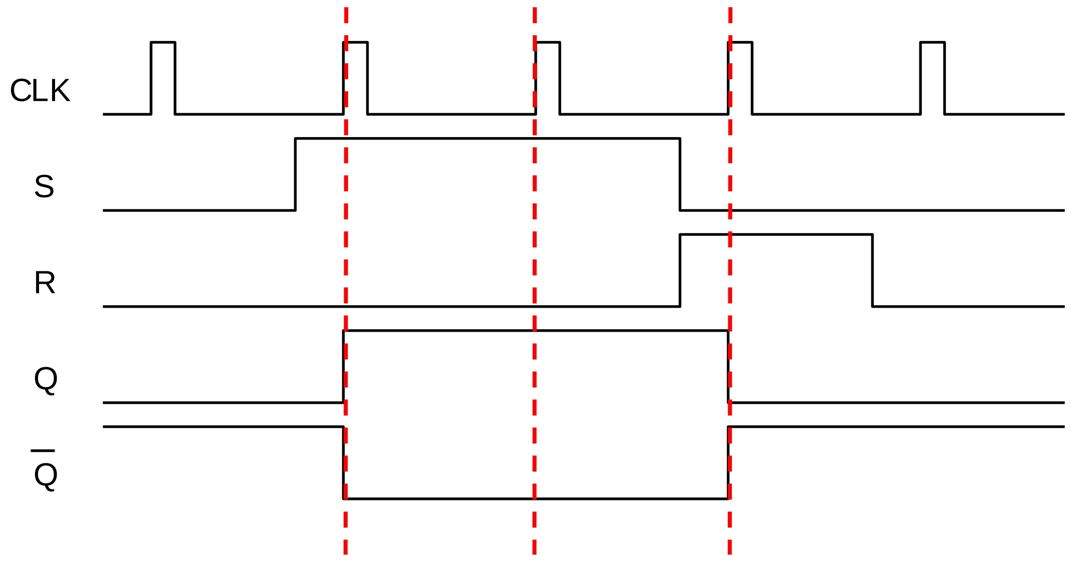
And an unstable state in which both S and R are set (which can cause the Q and not Q lines to toggle)

Example circuit available at: https://www.edaplayground.com/x/P4Sw

Gated SR Latch: a latch with enable

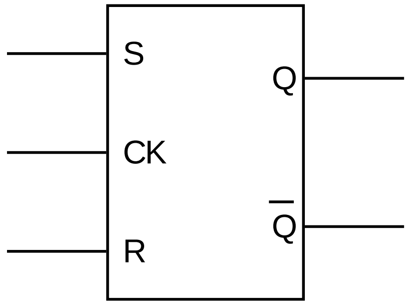


Combinational logic circuit with a clock (or enable) input connected.
Usually, this type used in digital systems.
It of course costs more in transistors!!



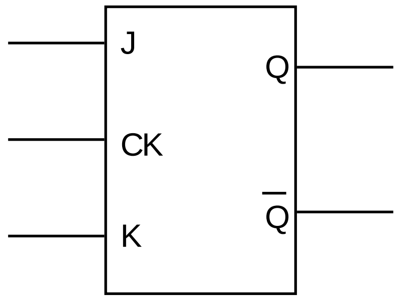
Only changed on clock pulse

Example signals



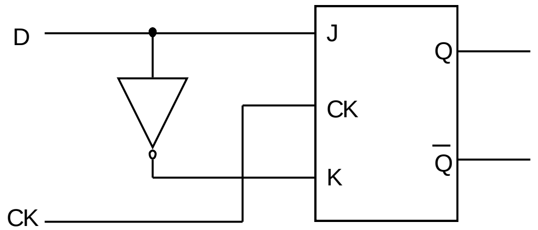
Gated SR-Latch Symbol

The JK and D Flip-flops



JK flip-flop

The standard JK flip-flop is much the same as a gated SR latch, modified so that Q toggles when J = K = 1

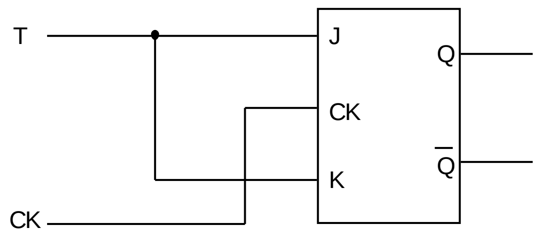


D flip-flop

The D-type flip flop (which you may want to use in Prac5 to store data) is a JK flop flop modified (see left) to hold the state of input D at each clock pulse.

clock	D	Q
0	0	X
1	1	0
2	1	1
3	0	1
…	…	…

T-type Flip-flop

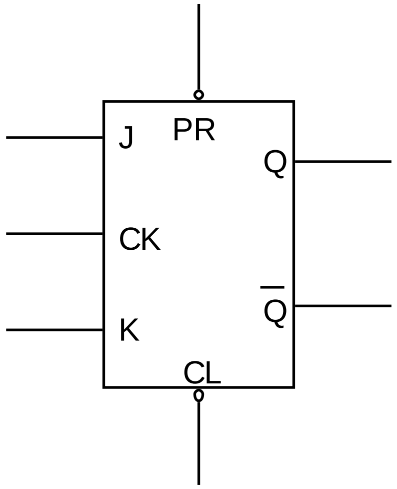


T flip-flop

The T-type flip-flops toggle the input. Q = not Q each time T is set to 1 when the clock pulses

Clock	T	Q
0	1	0
1	0	1
2	1	1
3	0	0
…	…	…

Preset and Clocking

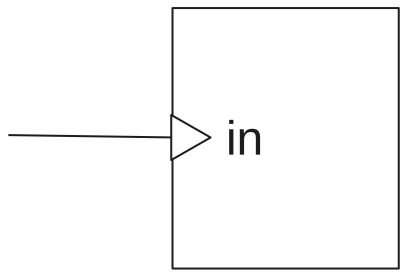


Preset line (PR) and clear line (CL) are asynchronous inputs used to set (to 1) or clear the value stored by the flip-flop.

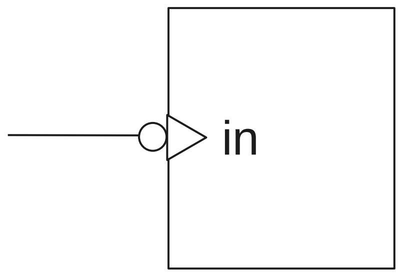
This is a type of structure one may need to use if init blocks were synthesised to initialize register values.

Edge triggered devices

A note on notation:
Edge-triggered inputs are shown using a triangle.
Negative edges triggered inputs are shown without a circle on the incoming line.



Positive edge triggered



Negative edge triggered

End of Lecture

Any Question??

NOT IN EXAM

RC Platform Case Studies Large & small FPGA-based RC systems

EEE4120F

(These slides, 46- onwards, planned to be a separate lecture and discussion of these innovative designs)

NOT IN EXAM

Why should we look at some RC Platform Case Studies?

This provides some interesting examples of what various companies and research organizations have done with FPGA platforms.

It’s to broaden your perspective on how FPGAs might be used…

List of Case Studies

NOT IN EXAM

- Large-scale FPGA-based RC system examples
  - PAM, VCC, Splash
- Small-scale FPGA-based RC system examples
  - PRISM
  - Algotronix CAL, XC620,
  - Cray Research XD1,
  - SRC
  - Silicon Graphics RASP

NOT IN EXAM

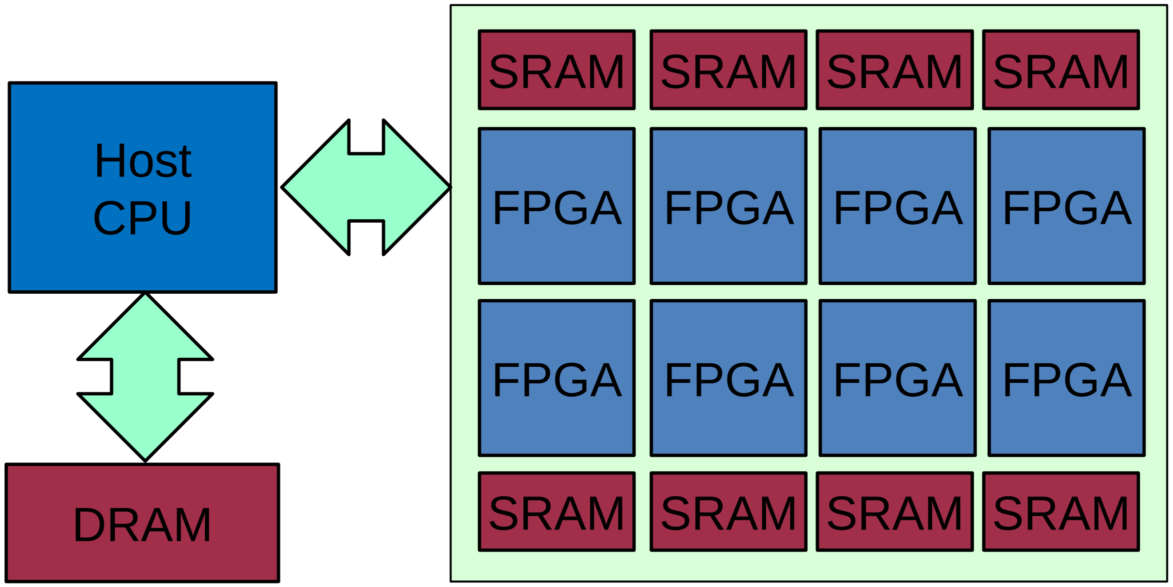
Large-scale RC Systems

A look at platforms architectures

NOT IN EXAM

Large RC System - PAM

- Programmable Active Memories (PAM)
  - Produced by Digital Equipment Corp (DEC)
  - Used Xilinx XC3000 FPGAs
  - Independent banks of fast static RAM



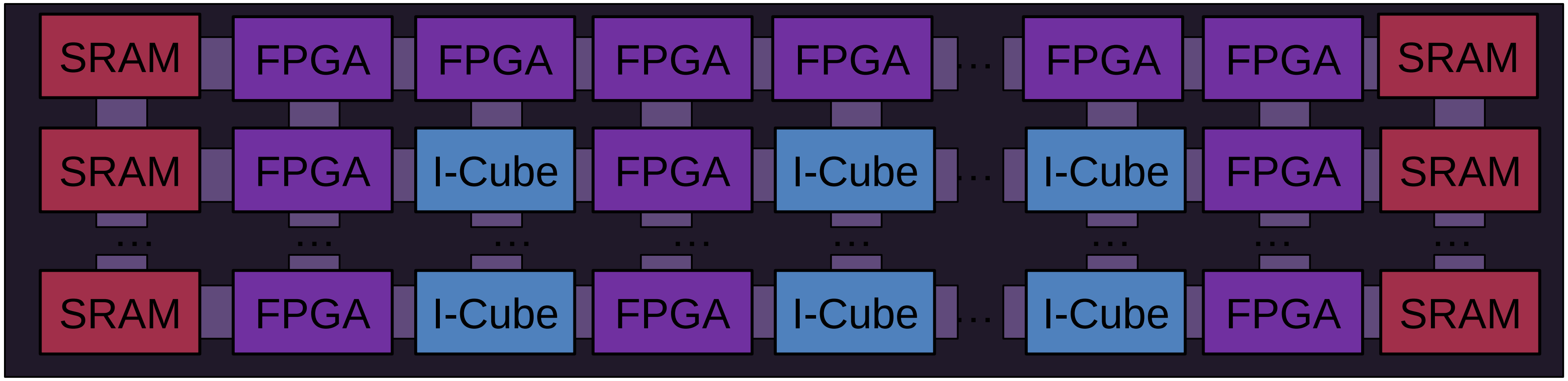
Digital Equipment Corp. PAM system (1980s)

Image adapted from Hauck and Dehon (2008) Ch3

Large RC System - VCC

NOT IN EXAM

- Virtual Computer Corporation (VCC)*
- First commercially commercial RC platform**
- Checkerboard layout of
  - Xilinx XC4010 devices and
  - I-Cube programmable interconnection devices
  - SRAM modules on the edges



VCC Virtual Computer

** Hauck and Dehon (2008)

* The Virtual Computer Corporation's (VCC) Virtual Computer (or Virtual Compacted Computer) architecture. http://commacorp.com/Virtual_Computer.pdf

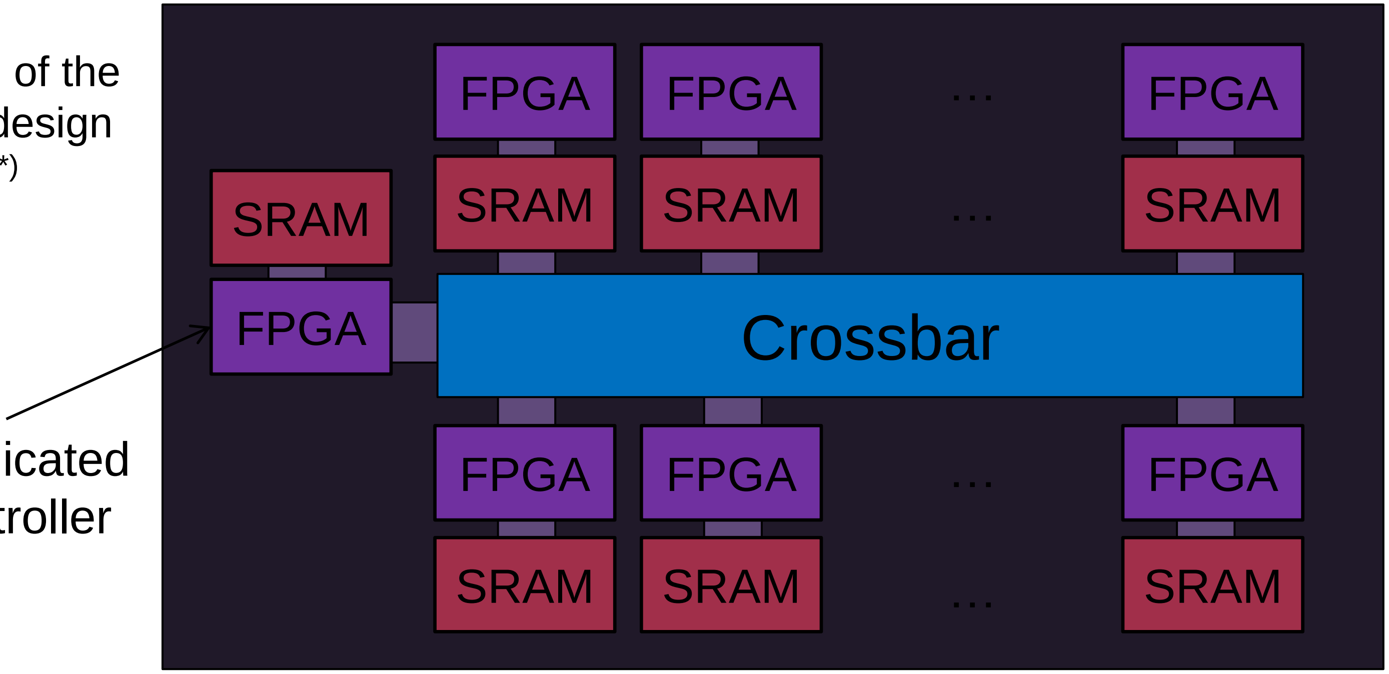
Large RC System - Splash

NOT IN EXAM

- Dev. by Super Computer Research (SCR) Center ~1990
- Well utilized (compared to previous systems).
- Comprised linear array of FPGAs each with own SRAM *

Summary of the Splash system
Developed initially to solve the problem of mapping the human genome and other similar problems. Design follows a reconfigurable linear logic array. The SPLASH aimed to give a Sun computer *better than supercomputer performance* for a certain types of problems. At the time, the performance of SPLASH was shown to *outperform a Cray 2 by a factor of 325*. FPGAs were used to build SPLASH, a cross between a specialized hardware board but more flexible like a supercomputer. The SPLASH system consists of software and hardware which plugs into two slots of a Sun workstation. **

Illustration of the SPLASH design
(adapted from *)



* Hauck and Dehon (2008)

SRC Splash version 2

**Adapted from: Waugh, T.C., "Field programmable gate array key to reconfigurable array outperforming supercomputers," Custom Integrated Circuits Conference, 1991., Proceedings of the IEEE 1991 , vol., no., pp.6.6/1,6.6/4, 12-15 May 1991 doi: 10.1109/CICC.1991.164051

NOT IN EXAM

Small-scale RC Systems

A look at platforms architectures

NOT IN EXAM

Small RC Systems

- Brown University's PRISM
  - Single FPGA co-processor in each computer in a cluster
  - Main CPUs offloading parallelized functions to FPGA
- Algotronix
  - Configurable Array Logic (CAL) – FPGA featuring very simple logic cells (compared to other FPGAs)
  - Later become XC6200 (when CAL bought by Xilinx)

* Hauck and Dehon (2008)

Conclusion & Suggested Reading

NOT IN EXAM

- Reading
  - Hauck, Scott (1998). "The Roles of FPGAs in Reprogrammable Systems" *In Proceedings of the IEEE.* 86(4) pp. 615-639.

This paper by Hauck gives a view on where FPGAs where expected to be utilized and how they could significantly change the way systems are designed. It's a fairly early paper, considering the short history of FPGAs, and indeed the Hauck's views are accurate but also underestimates the versatility and the breadth of domains that these technologies have reached.

Disclaimers and copyright/licensing details

Image sources:

man working on laptop – flickr

scroll, video reel, big question mark – Pixabay http://pixabay.com/ (public domain)

Processing Gears and Spanner – MaxPixel https://www.maxpixel.net (CC0)

some diagrammatic elements are from Xilinx ISE screenshots

References: Verilog code adapted from
http://www.asic-world.com/examples/verilog