Gisselquist
Technology, LLC

1. Wires

Daniel E. Gisselquist, Ph.D.

Lesson Overview

- What is a **wire**?
- What can I do with it?
- How do I build a design?

Objectives

- To get an initial, basic familiarization with combinational logic
- To learn how to run the tools to build a design
- To get an initial design running on an FPGA board

First design

Let's build a simple Verilog design

```
module thruwire(i_sw, o_led);
        input   wire    i_sw;
        output  wire    o_led;

        assign  o_led = i_sw;
endmodule
```

First design

Let's build a simple Verilog design

```
module thruwire(i_sw, o_led);
        input   wire    i_sw;
        output  wire    o_led;

        assign  o_led = i_sw;
endmodule
```

- Verilog files contain modules
- This module is named `thruwire`
- While Verilog allows more than one module per file, I recommend only one module per file.

First design

Let's build a simple Verilog design

```
module thruwire(i_sw, o_led);
        input   wire    i_sw;
        output  wire    o_led;

        assign  o_led = i_sw;
endmodule
```

- The **module** keyword marks the beginning
- **endmodule** marks the end of the module

First design

Let's build a simple Verilog design

```
module thruwire(i_sw, o_led);
        input   wire    i_sw;
        output  wire    o_led;

        assign  o_led = i_sw;
endmodule
```

- This module declare two ports, `i_sw` and `o_led`
- The first is declared to be an **input**
- The second an **output**
- Both are **wire**'s, but we'll get to that later

First design

Let's build a simple Verilog design

```
module thruwire(i_sw, o_led);
        input   wire     i_sw;
        output  wire     o_led;

        assign  o_led = i_sw;
endmodule
```

- Our one piece of logic sets `o_led` to be the same as `i_sw`

First design

Let's build a simple Verilog design

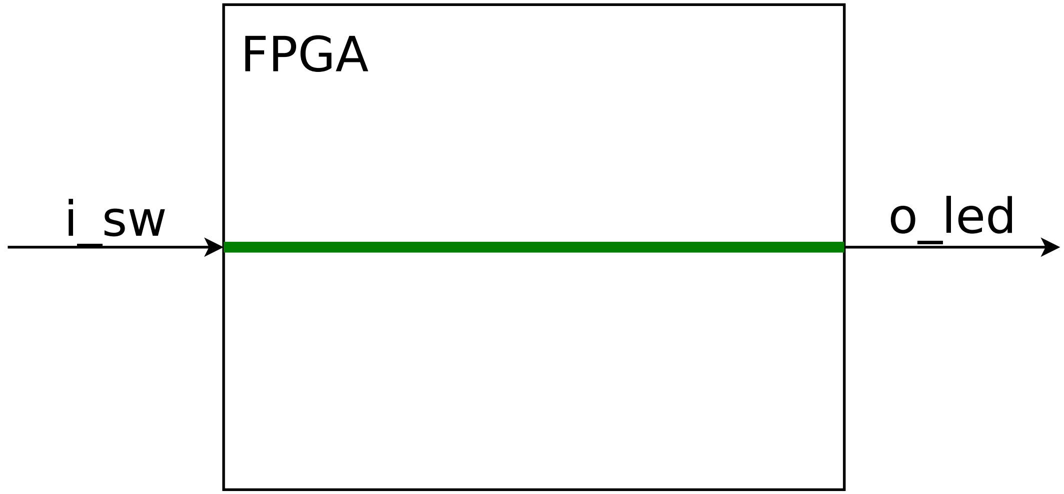
```
module thruwire(i_sw, o_led);
        input   wire    i_sw;
        output  wire    o_led;

        assign  o_led = i_sw;
endmodule
```

FPGA's are commonly used as:

- Traffic cops
  A programmable/adjustable wire fabric
- Voltage level shifters
- This logic would be appropriate for each
  ... it generates a simple "wire" through the chip

Schematic

Here's what a schematic of this design would look like

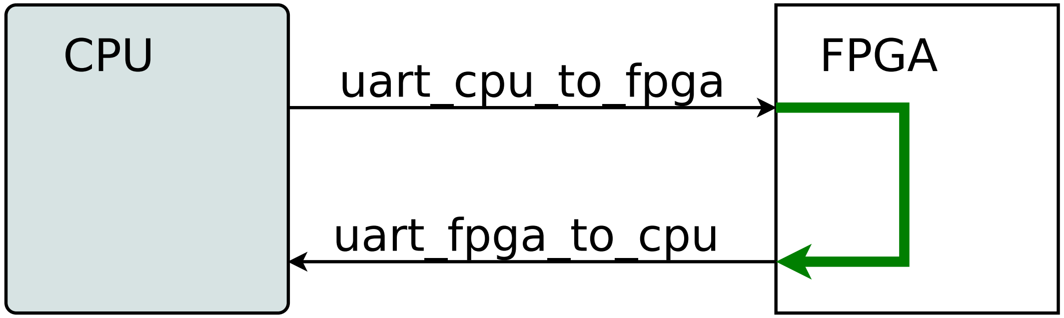


All from this assign statemnt

```
assign o_led = i_sw;
```

Schematic

A very similar design would make a good first serial port test



- Your circuit board should pass this test before you try to implement your own serial port within it

Constraints

A fundamental part of any FPGA design maps your ports to the pins

- This is the purpose of a *Constraint File*
- Different vendors use different forms for their constraint files
  - PCF: Used by Arachne-PNR and NextPNR
  - UCF: Used by ISE for older Xilinx designs
  - XDC: Used by Vivado for newer Xilinx designs
  - QSF: Used by Quartus for ~~Altera~~ Intel chips
- Your board vendor should provide you with a master constraint file
- You'll still need to
  - Comment-out pins you aren't using
  - Rename pins to match your Verilog

PCF File

If you are using `nextpnr`, you'll need a PCF file

```
set_io   i_sw     P13
set_io   o_led    C8
```

- Maps top-level ports to pins
- You'll find `P13` and `C8` on the schematic
  - Find the FPGA pins connected to the switch
    ... and the LED output
  - If your design has no switches, you can use buttons (for now)
    Buttons also bounce, but we'll get to that later

UCF File

If you are using `ISE`, you'll need a UCF file

```
NET "i_sw"  LOC = "P9" | IOSTANDARD = LVCMOS33;
NET "o_led" LOC = "N3" | IOSTANDARD = LVCMOS33;
```

- This would be for the older Xilinx FPGA's
- Make sure you actually look up the correct pins
  - `P13` for one board might be something else on another

  On this board, the switch is on pin `P9`
- Most development boards use the 3.3V LVCMOS standard
  - Pins are typically grouped in banks
  - All pins in a bank use the same voltage
  - This voltage is usually fixed
  - The master constraint file will help here

XDC File

If you are using `Vivado`, you'll need a XDC file

```
set_property -dict {PACKAGE_PIN E22
      IOSTANDARD LVCMOS12} [get_ports {i_sw}]
set_property -dict {PACKAGE_PIN T14
      IOSTANDARD LVCMOS25} [get_ports {o_led}]
```

- This would be for the newer Xilinx FPGA's
- Usually, the vendor will provide a "master XDC" file
- From there, you should be able to
  - Rename the appropriate ports to `i_sw` and `o_led`
  - Comment out every other I/O port

Build the design

For an iCE40 design, this will look like:

```
% yosys -p 'synth_ice40 -json thruwire.json' \
      thruwire.v
% nextpnr-ice40 --hx8k --package ct256 \
      --pcf thruwire.pcf --json thruwire.json
% icepack thruwire.asc thruwire.bin
```

You'll need to do this for every project–get used to this flow.

- A makefile can drastically simplify this process

You should now have a file `thruwire.bin` that you can load onto your board.

- If you aren't using an iCE40, follow your chip vendor's instructions

First Success!

Follow your board vendor's instructions for loading this file onto your board.
Notice now that every time you flip the switch, the LED responds

First Success!

Follow your board vendor's instructions for loading this file onto your board.
Notice now that every time you flip the switch, the LED responds
Yaaaayyyyyy!!! Your first FPGA design.

Simulation

Simulation is an important part of design

Simulation	Hardware
Can trace all signals	Can only see some signals
Extended tests cost GB	Extended tests are simple
Easy to debug	*Very hard* to debug

Because hardware is so hard to debug, simulation is vital

- A successful complex project
  . . . *requires simulation*!

Simulation

Simulation is an important part of design

Simulation	Hardware
Can trace all signals	Can only see some signals
Extended tests cost GB	Extended tests are simple
Easy to debug	*Very hard* to debug

Because hardware is so hard to debug, simulation is vital

- A successful complex project
  . . . *requires simulation*!

Do it the easy way:

Simulation

Simulation is an important part of design

Simulation	Hardware
Can trace all signals	Can only see some signals
Extended tests cost GB	Extended tests are simple
Easy to debug	*Very hard* to debug

Because hardware is so hard to debug, simulation is vital

- A successful complex project
  . . . *requires simulation*!

Do it the easy way: *use the simulator!*

Verilator

Let's now build our design using Verilator

```
% verilator -Wall -cc thruwire.v
% cd obj_dir/
% make -f Vthruwire.mk
```

- Verilator compiles Verilog into C++ placed into `obj_dir/`
- The make command then builds this converted C++ file into a shared object file we can now use

Verilator Driver

You'll need a main simulation driver too.

```
#include <stdio.h>
#include <stdlib.h>
#include "Vthruwire.h"
#include "verilated.h"

int main(int argc, char **argv) {
        // Your logic here
}
```

Verilator Driver

You'll need a main simulation driver too.

```
// ...
int main(int argc, char **argv) {
        // Call commandArgs first!
        Verilated::commandArgs(argc, argv);

        // Instantiate our design
        Vthruwire *tb = new Vthruwire;

        // ...
}
```

Verilator Driver

You'll need a main simulation driver too.

```
int main(int argc, char **argv) {
        // ...

        // Now run the design thru 20 timesteps
        for(int k=0; k<20; k++) {
                // We'll set the switch input
                // to the LSB of our step
                tb->i_sw = k&1;

                tb->eval();

                // ...
        }
}
```

Verilator Driver

You'll need a main simulation driver too.

```
int main(int argc, char **argv) {
        // ...
        for(int k=0; k<20; k++) {
                // We'll set the switch input
                // to the LSB of our counter
                tb->i_sw = k&1;

                tb->eval();

                // Now let's print our results
                printf("k = %2d, ", k);
                printf("sw = %d, ", tb->i_sw);
                printf("led = %d\n", tb->o_led);
        }
}
```

Building it all

Last step, let's put it all together:

```
% g++ -I /usr/share/verilator/include                    \
    -I obj_dir/                                          \
    /usr/share/verilator/include/verilated.cpp           \
    thruwire.cpp obj_dir/Vthruwire__ALL.a                \
    -o thruwire
```

(Double check the location of Verilator in your own installation, it might be located in another directory.)

Wow, that's pretty complicated.

You should have a `Makefile` in your `ex-01-thruwire` directory with both the code and the build instructions.

```
% cd ex-01-thruwire/
% make
# (Make output skipped for brevity)
%
```

Simulation

We can now run our simulator!

```
% thruwire
k =  0, sw = 0, led = 0
k =  1, sw = 1, led = 1
k =  2, sw = 0, led = 0
k =  3, sw = 1, led = 1
k =  4, sw = 0, led = 0
k =  5, sw = 1, led = 1
k =  6, sw = 0, led = 0
k =  7, sw = 1, led = 1
k =  8, sw = 0, led = 0
k =  9, sw = 1, led = 1
# .... (Lines skipped for brevity)
%
```

Good habits

Many Verilog problems can be avoided by some simple steps

1. Make **‘default_nettype** `none` the first line of your Verilog file
   - Before your **module** declaration
   - Otherwise mis-spelled identifiers will be quietly turned into wires

```
module thruwire(i_sw, o_led);
        input   wire    i_sw;
        output  wire    o_led;

        assign  o_led = sw;
endmodule
```

Without **‘default_nettype** `none`, this design would pass without error

Good habits

Many Verilog problems can be avoided by some simple steps

1. Make **‘default_nettype** `none` the first line of your Verilog file
2. Fix any errors when you `verilator -Wall` your design
3. Run your design in a simulator
   - Attempt to recreate any hardware bugs . . . *in the simulator*

These three rules will save you a lot of heartache!
. . . *Get in the habit of using them!*

Bus Signals

That was one single wire. We can also declare values consisting of many bits.

```
input   wire    [8:0]   i_sw;
output  wire    [8:0]   o_led;
```

This defines

- `i_sw` to be 9-input wires, and
- `o_led` to be 9-output wires

Bit Select

- Select bits of interest from a bus

```
assign  o_led[7] = i_sw[0];
assign  o_led[6:5] = i_sw[5:4];
```

  - Bit 7 of `o_led` is set to bit 0 of `i_sw`
  - Bits 5 and 6 of `o_led` are set to bits 4 and 5 of `i_sw`

- Concatenate bits together

```
assign  o_led[4:0] = { i_sw[2:0], i_sw[7:6] };
```

  - The $\{\cdot, \cdot\}$ operator composes a new bit vector from other vectors

Internal Signals

You can also declare and work with internal wires

```
wire    [8:0]   w_internal;
```

- Internal wires are neither **input** nor **output**
- These wires can now be used in logic

```
assign  w_internal = 9'h87;
assign  o_led = i_sw ^ w_internal;
```

Literals

A Verilog literal is defined as

- A width
- An apostrophe
- An optional sign indication, s

  Defaults to unsigned

- A numeric type: h (hex), d (decimal), o (octal), b (binary), sd (signed decimal)
- The value: a series of digits, possibly containing underscores
  Underscores can be *very* useful for longer numbers

Examples include:
1'b0 1'b1 2'b01 4'b0101 4'h5 -7'sd124
32'hdead_beef 32'd100_000_000
Place a '-' in front of the width for negative numbers

Sign Extension

If the literal is smaller than the context . . .

- If there is no 's', the number is unsigned and it is zero extended
- Any literal with an 's' is sign extended
- . . . to fit the width

If the literal is too big for the context . . .

- It is truncated to fit the context

Many tools will create a warning for width mismatches

Operators

The Verilog language supports the following operators

<table>
<tr><td>+</td><td>Addition</td><td>−</td><td>Subtraction</td></tr>
<tr><td><<</td><td>Left Shift</td><td>>></td><td>Right shift</td></tr>
<tr><td>−</td><td>Unary negation</td><td>?:</td><td>Tertiary operator</td></tr>
<tr><td>~</td><td>Bit-wise negation</td><td>^</td><td>Bit-wise XOR</td></tr>
<tr><td>|</td><td>Bitwise OR</td><td>&</td><td>Bitwise AND</td></tr>
<tr><td>||</td><td>Logical OR</td><td>&&</td><td>Logical and</td></tr>
<tr><td>!</td><td>Logical negation</td><td>>>></td><td>Arithmetic right shift</td></tr>
<tr><td>==</td><td>Equality</td><td>! =</td><td>Inequality</td></tr>
<tr><td><, <=</td><td>Less than (Equal)</td><td>>, >=</td><td>Greater than (Equal)</td></tr>
<tr><td colspan="2">Limited, use with care</td><td colspan="2">Avoid within logic</td></tr>
<tr><td>*</td><td>Multiplication</td><td>/</td><td>Division</td></tr>
<tr><td></td><td></td><td>%</td><td>Remainder</td></tr>
</table>

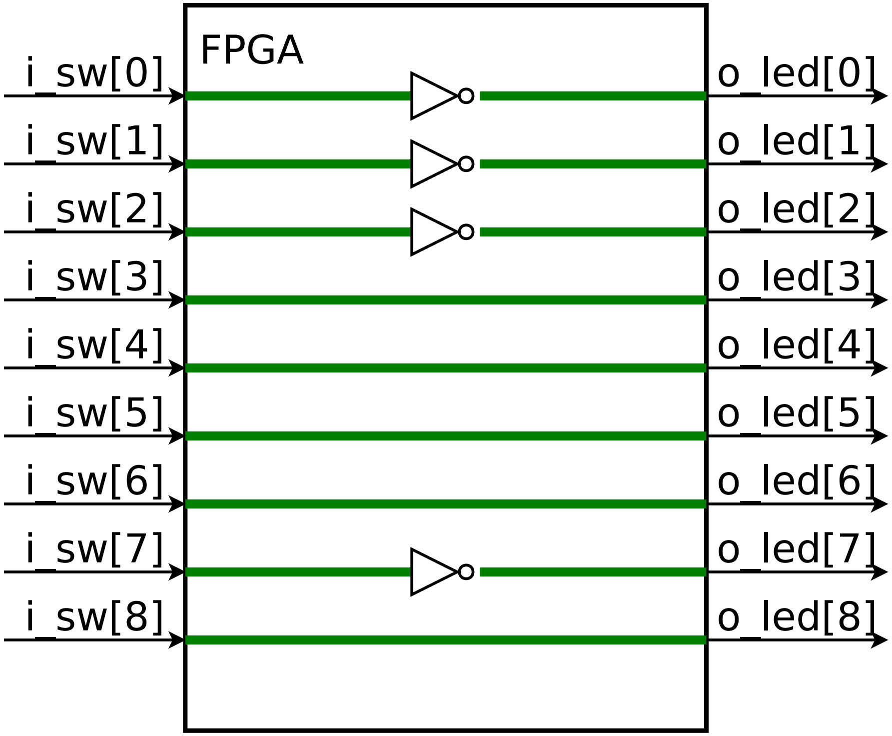
- Some FPGA's support native multiplication
- None support a single clock divide or remainder

Schematic

From this code:

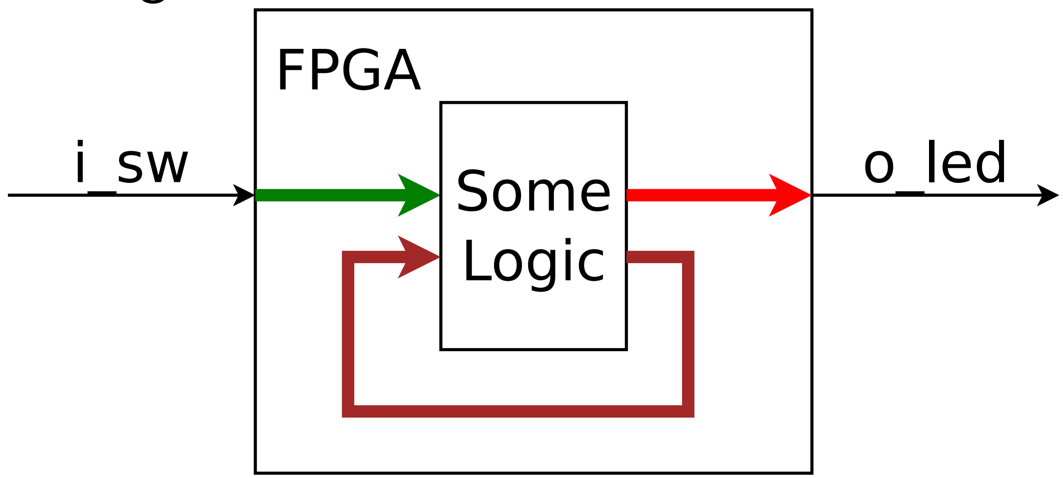
```
assign  w_internal = 9'h87;
assign  o_led = i_sw ^ w_internal;
```

Get this internal structure:



Circular Logic

Avoid circular logic!



Example:

```
assign o_led = i_sw + o_led;
```

- This doesn't work in hardware like it might in software
- This is roughly equivalent to creating a short circuit
- Most tools will fail to build such designs
  This include Verilator

Dual Assignment

You are designing hardware: A value can only be set once

This is an error:

```
assign  o_led = i_sw + o_led;
assign  o_led = i_sw + 1;
```

Let's build it

Let's build this design:

```
`default_nettype        none

module maskbus(i_sw, o_led);
        input   [8:0]   i_sw;
        output  [8:0]   o_led;

        wire    [8:0]   w_internal;

        assign  w_internal = 9'h87;
        assign  o_led = i_sw ^ w_internal;
endmodule
```

. . . using Verilator

Updated Driver

Let's update our driver for this wire bus design

```
int main(int argc, char **argv) {
        // ...
        for(int k=0; k<20; k++) {
            // ...
            // Bottom 9 bits of counter
            tb->i_sw = k & 0x1ff;

            tb->eval();

            // Now let's print our results
            printf("k = %2d, ", k);
            printf("sw = %3x, ", tb->i_sw);
            printf("led = %3x\n", tb->o_led);
        }
}
```

Sim Result

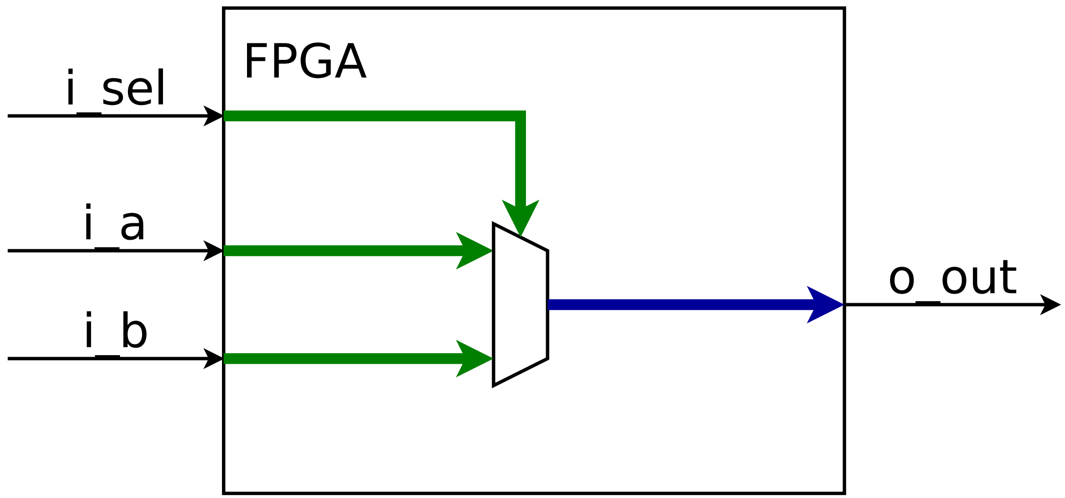

```
% ./maskbus
k =  0, sw =   0, led =  87
k =  1, sw =   1, led =  86
k =  2, sw =   2, led =  85
k =  3, sw =   3, led =  84
k =  4, sw =   4, led =  83
k =  5, sw =   5, led =  82
k =  6, sw =   6, led =  81
k =  7, sw =   7, led =  80
k =  8, sw =   8, led =  8f
k =  9, sw =   9, led =  8e
# .... (Lines skipped for brevity)
%
```

Examples

What can you do with wires and wire logic?

Example: Multiplexer

```
input   wire    i_a, i_b, i_sel;
output  wire    o_out;

assign  o_out = (i_sel) ? i_a : i_b;
```



Examples

What can you do with wires and wire logic?
Example: Multiplexer

```
input   wire    i_a, i_b, i_sel;
output  wire    o_out;

assign  o_out = (i_sel) ? i_a : i_b;
```

- This is a good example of the tertiary operator
- Interested in making a connection to one of two serial ports?
- How about connecting one of two bus masters to an interconnect?

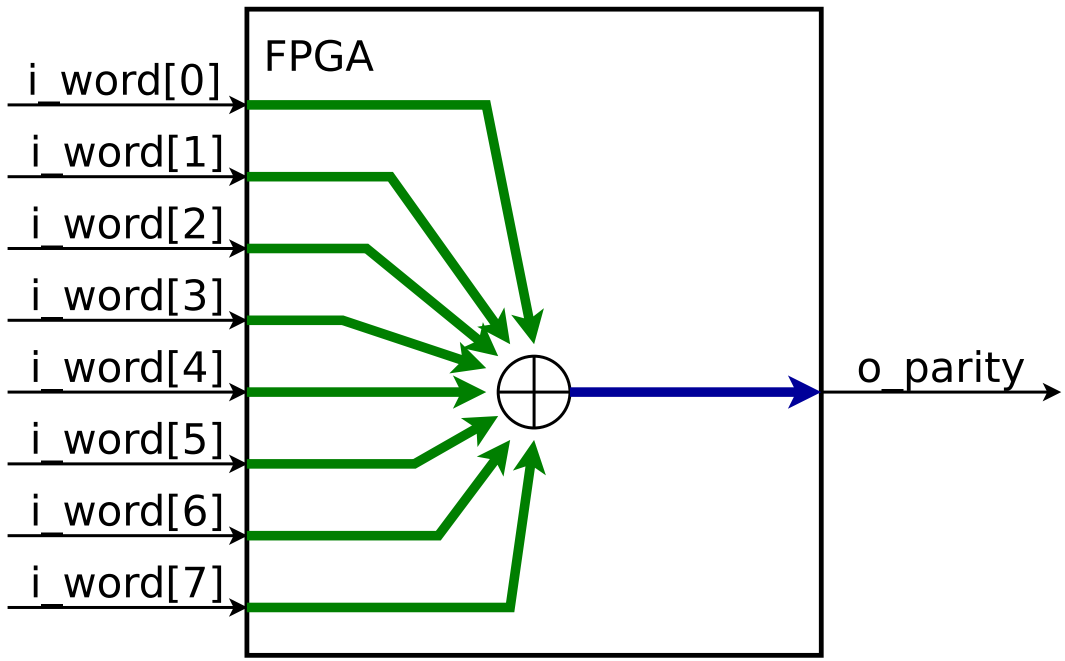
We'll get to these examples later.

Examples

What can you do with wires and wire logic?

Example: Parity check

```
input   wire    [7:0]   i_word;
output  wire            o_parity;

assign  o_parity = ^i_word;
```



Examples

What can you do with wires and wire logic?

Example: Parity check

```
input   wire    [7:0]   i_word;
output  wire            o_parity;

assign  o_parity = ^i_word;
```

This form of XOR is a *reduction operator*

- It XORs all the word's bits together
- Other reduction operators include | and &

Error Correction Code (ECC) creation logic is very similar

Examples

What can you do with wires and wire logic?

Example: Interrupt detector

```
input   wire   [7:0]   i_irq_source;
output  wire           o_irq;

assign  o_irq = |i_irq_source;
```

- `i_irq_source` contains eight interrupt sources
- `o_irq` is true if any interrupt source is true

Examples

What can you do with wires and wire logic?

Example: CPU stall determination

```
assign  dcd_stall = (dcd_valid)&&(op_stall);
```

From the ZipCPU, the decode stage must stall if

- It has produced a valid result, and
- The next stage, read operands, is stalled for some reason

  These stalls can back up through the CPU

  Ex. Read operands might be stalled if the ALU is stalled

Examples

What can you do with wires and wire logic?

Example: Determining if there's a phase error in a phase lock loop

```
assign  phase_err = (output_phase != input_phase);
```

In this case, the loop will adjust if there are any errors

Exercise

This section has two exercises:

1. Build and try the `thruwire` demo.
   - Toggle the switch.
   - Verify that toggling your switch will toggle the LED
   - Build and run the Verilator simulation
2. Create a test of your serial port connection
   - Connecting the input serial port wire to the output
     Beware: These wires are often marked "TX" and "RX", but not always from the perspective of the FPGA
   - Turn off any 'local echo'
   - Turn off any hardware flow control
   - Verify that characters typed into your terminal program show up on the screen

Conclusion

- Wires represent connections within the design
- Wires can also represent the outputs of combinatorial logic
- Wires have no memory, circular logic or feedback is illegal
- You know how to create constraints for your project!

You can now build and load a design onto an FPGA!