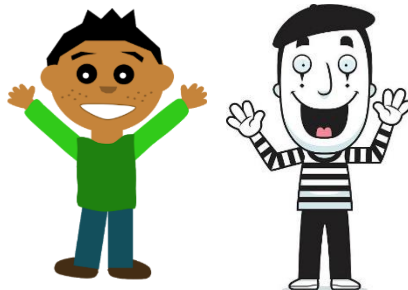
EEE4120F

High Performance Embedded Systems

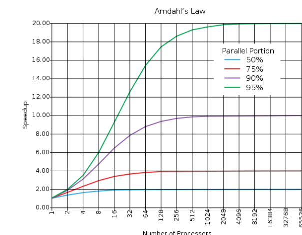
Lecture 19

HDL Imitation Method, Benchmarking and Amdahl's for FPGAs

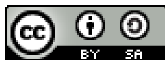


HDL HDL Imitation

Lecturer:
Simon Winberg



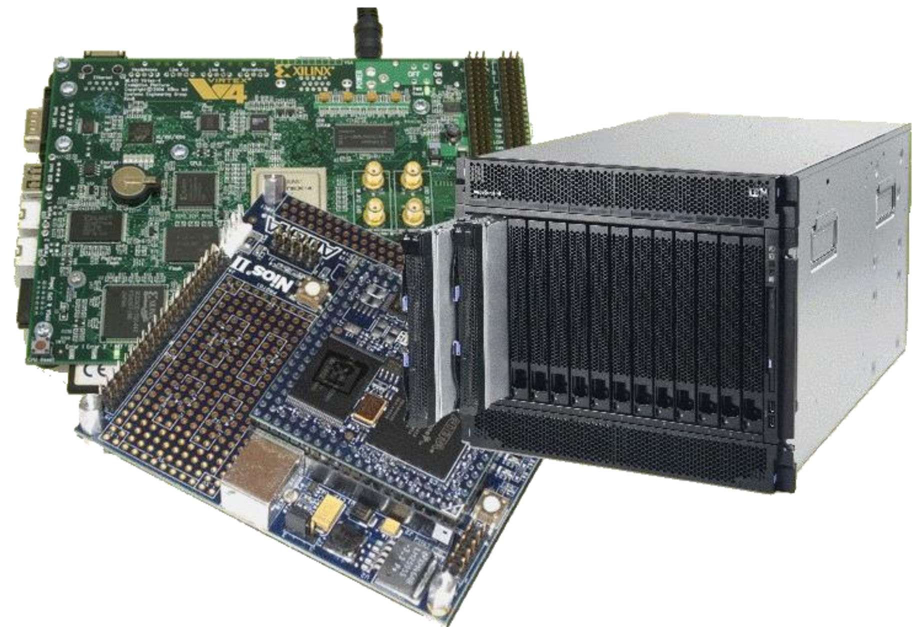
Amdahl's for FPGA



Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

Lecture Overview

- HDL Imitation Method
- Using Standard Benchmarks for FPGAs
- Amdahl’s Law and FPGA



HDL Imitation Method

or ‘C-before-HDL approach to starting HDL designs.

An approach to ‘golden measures’ & quicker development

C

```
void mymod
  (char* out, char* in)
{
  out[0] = in[0]^1;
}
```

HDL

```
module mymod
  (output out, input in)
{
  out = in^1;
}
```

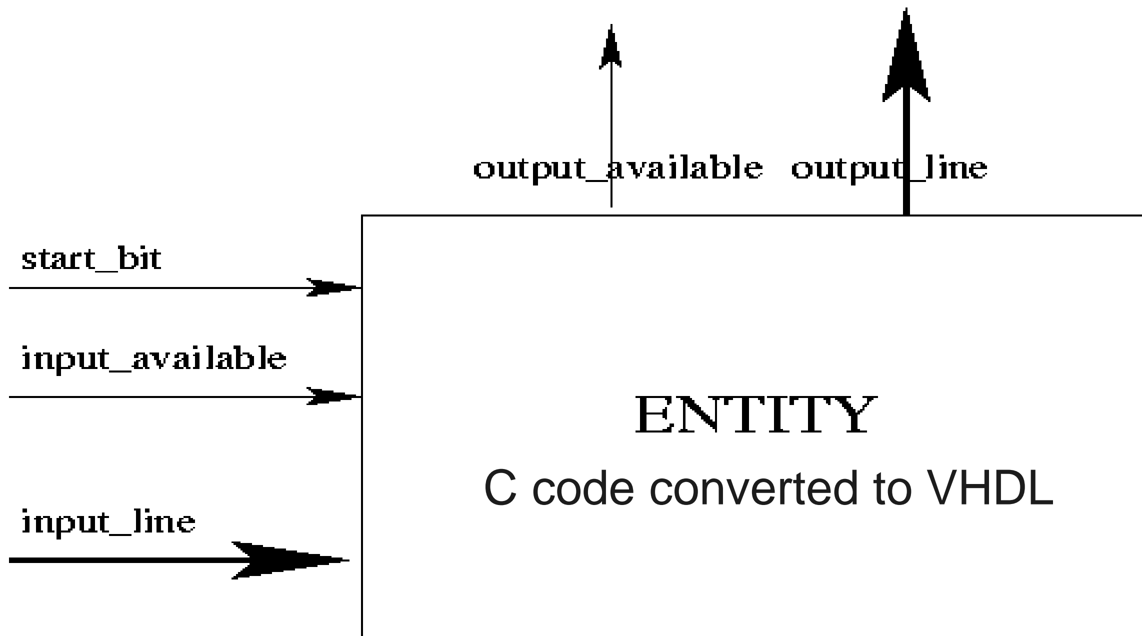
The same method can work with Python, but C is better suited due to its typical use of pointer.

HDL Imitation Method

- This method can be useful in designing both golden measures and HDL modules in (almost) one go …
- It is mainly a means to validate that you algorithm is working properly, and to help get into a 'thinking space' suited for HDL.
- This method is loosely based on approaches for C→HDL automatic conversion (discussed later in the course)

HDL Imitation approach using C

- C program: functions; variables;
  - based on sequence (start to end) and the use of memory/registers operations
- VHDL / Verilog HDL:
  - Implements an entity/module for the procedure



HDL Imitation in C

& good references to review in using this method

- Standard C characteristics
  - Memory-based
  - Variables (registers) used in performing computation
  - Normal C and C programs are sequential
- Specialized C flavours for parallel description & FPGA programming:
  - Mitrion-C , SystemC , pC (IBM Parallel C) System Crafter, Impulse C , OpenCL
  - FpgaC Open-source (http://fpgac.sourceforge.net/) – does generate VHDL/Verilog but directly to bit file

HDL Imitation: where it's useful

- Best to simplify this approach, where possible, to just one module at a time
- When you're confident the HDL works, you could just leave the C version behind
- Getting a whole complex design together as both a C-imitating-HDL program and a true HDL implementation is likely not viable (as it may be too much overhead to maintain)

HDL Imitation: Example

Example Task:
Implement an countup module that counts up on target value, increasing its a counter value on each positive clock edge. When the target value is reached set the counter_done flag and stop counting.

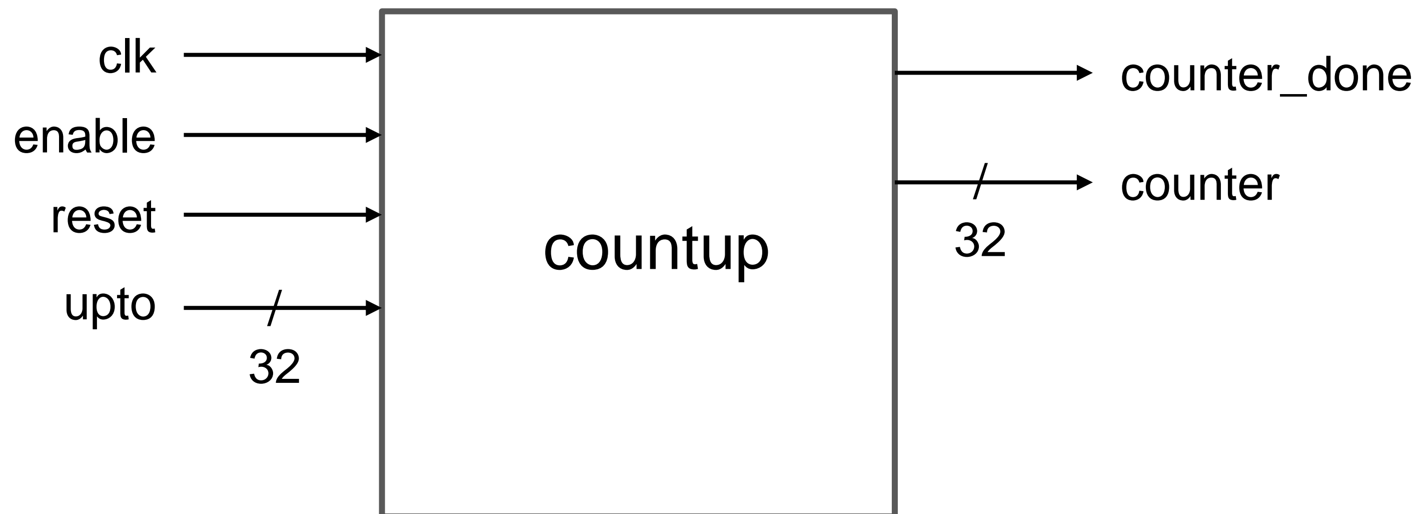
Approach:
1. Sketch the design of the needed module and its interface.
2. Think what registers are needed, including any regs to test the module.
3. Write a quick C implementation that can act as both a quick starting point and test of the plan, and which can then be easily converted to HDL.
4. Test the C program, make sure it is behaving as anticipated.
5. Covert the program to HDL
6. Test the HDL and make sure it is also working.

Note: obviously this is a very simple example for illustrative purposes. You are unlikely to use this approach for such simple situations especially once you are feeling confident in HDL coding; but for more complex problems that can be a valuable time-saver and 'sanity-check' for your HDL code.

HDL Imitation: Example

Example Task:
Implement an countup module that counts up on target value, increasing its a counter value on each positive clock edge. When the target value is reached set the counter_done flag and stop counting.

1. Sketch the design of the needed module and its interface….



Design note: In standard Verilog you cannot have global signals. Each module needs to be quite stand-alone, it can only be connected to via its ports; you cannot somehow link to a global register without connecting through a port. You can have a 'tristate' register that you can either read or write (like a variable parameter).

HDL Imitation: Example

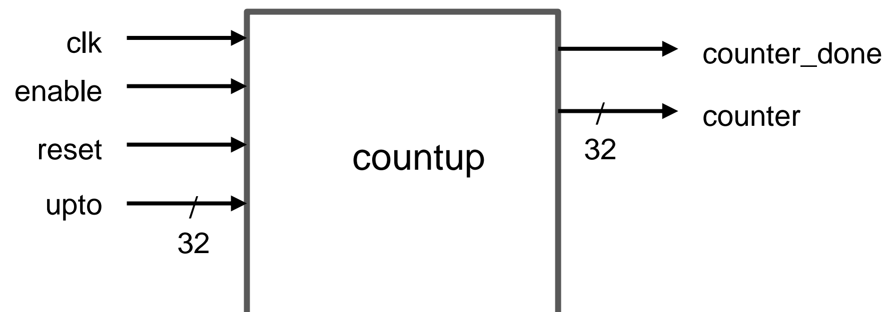
2. Think what registers are needed, including any regs to test the module.

Looking at the module interface design (copied below) it is clear that we will need to have registers for:

- clk : a bit
- enable : a bit
- reset : a bit
- upto   : a bus (the same size as counter)
- counter : a bus (of 32 bits, could consider it an unsigned int)
- counter_done : a bit

The inputs to the module are: clk, enable, reset and upto.
The outputs are: counter and counter_done

Note that here counter is considered an output, the value to be stored within countup.



HDL Imitation: Example

3. Write a quick C implementation that can act as both a quick starting point and test of the plan, and which can then be easily converted to HDL.

We can start with implementing the module… then we can think about implementing the testbench, essentially same approach as using Verilog.

```
// Modules to test ///////////////////////////////////

void countup (
    // inputs:
    bit clk, bit enabled, bit reset, UNSIGNED_BUS upto,
    // outputs
    UNSIGNED_BUS& counter,  // note would be defined as reg in countup, i.e. stores value
    bit& counter_done )
{
    static bit reached;
    // check if reset
    if (reset.now == 1) {
      counter = 0;
      CLR(reached); // not yey reached the upto target
    } else
    // this would be an always@ in Verilog....
    if (POSEDGE(clk)) {
      if ((enabled.now == 1) && (reached.now==0)) {
        counter = counter + 1;
        if (counter==upto) {
          SET(counter_done);
          SET(reached);
          }
      } // end if enabled==0
    }
}
```

Note the .now is explained in a moment, as is CLR, SET and POSEDGE. As well as why ‘&’ is there.

main.cpp

(explaining the C macros used in previous HDL imitation code)

```
// define a bit type
typedef struct bit_struct {
    unsigned char pre, now;
  } bit;

// define a unsigned bus type, for unsigned values
typedef unsigned UNSIGNED_BUS;

// define a bus type
typedef int      SIGNED_BUS;

#define SET(x)      {x.pre=x.now; x.now=1;}
#define CLR(x)      {x.pre=x.now; x.now=0;}
#define TOGGLE(x)   {x.pre=x.now; x.now=!x.now;}

#define POSEDGE(x)  (x.now>x.pre)
#define NEGEDGE(x)  (x.pre<x.now)
```

For bits we usually want to know if it has changed, if there was a posedge or negedge so we need the previous value, thus using a struct.

This is to more remind us that we need to implement this as a bus, e.g. input [31:0] bus;

Since we defined a bit type we need some operations for that. I we used proper C++, a bit class could have been implemented to do the same thing more elegantly. In Verilog you define a function for each of these so that the code looks the same.

This is the equivalent of a positive and negative edge, since we know the previous value of a bit. Again, if we used C++ this could become a function that receives a bit as input

HDL Imitation: Example

4. Test the C program, make sure it is behaving as anticipated.

For this we essentially need to write a testbench for the imitated module.

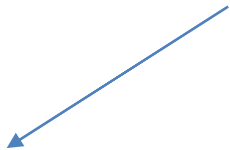
```
int main() {
    // Define output and inputs for top-level module
    unsigned n_clk; // for iterating clock pulses
    bit      clk;   // clock bit

    // registers to be used to pass to 'toplevel' module to test
    UNSIGNED_BUS counter, upto;
    bit          enable, reset, counter_done;

    // initialize values, this is kind of equivalent initial block in Verilog...
    CLR(clk); // remember we defined CLR to do equivalent of Verilog clk=0
    CLR(counter_done); SET(reset); SET(enable);
    counter = 0; upto    = 10;  // set target to count up to

    // print tables of register log
    printf("clk,counter,counter_done\n");

    // clock iterator
    for (n_clk=0; n_clk<CLOCKS; n_clk++) {
        // this is somewhat like a monitor statement
        printf("  %d,%07d,%01d\n",clk.now,counter,counter_done.now);
        // call te top-level module to be tested
        countup(clk,enable,reset,upto,counter,counter_done);
        // toggle the clock
        TOGGLE(clk);
        // see if a few clocks have passed to lower reset
        if (n_clk == 2) CLR(reset);
    }
    return 0;
}
```

Note here we are kind of setting the simulation duration by having a counter for the number of clocks (n_clk) to iterate through.

HDL Imitation: Example

Running the result

```
$ CasHDL
Test C-like-HDL Module!
clk,counter,counter_done
  0,0000000,0
  1,0000000,0
  0,0000000,0
  1,0000000,0
  0,0000001,0
  1,0000001,0
  0,0000002,0
  1,0000002,0
…
  0,0000009,0
  1,0000009,0
  0,0000010,1
  1,0000010,1
  0,0000010,1
…
  0,0000010,1
  1,0000010,1
```



And you can see from this log that the program words as anticipated, after counter reaches 10 (the upto value) it stops counting up.

So this basically means that you C program is working properly. It would then be a matter of translating the C into Verilog….

HDL Imitation: Example

```
// Countup module counts up to 'upto' value
module countup (
    // inputs:
    clk, enabled, reset, upto,
    // outputs
    counter, counter_done );
// toplevel module to test
input clk, enabled, reset;
input [31:0] upto;
output reg counter_done;
// local regsiters
reg reached;
output reg [31:0] counter;
always@(reset or posedge(clk))
  begin
   // check if reset
    if (reset == 1) begin
      counter = 31'b0;
      reached = 0; // not yet reached the upto target
      counter_done = 0;
    end else
    // this would be an always@ in Verilog....
      if ((enabled==1) & (reached==0)) begin
        counter = counter + 1;
        if (counter==upto) begin
          counter_done <= 1;
          reached=1;
        end
      end // end if done==0
  end // always
endmodule
```

5. Covert the program to HDL

Try on: https://www.edaplayground.com/x/4ELg

HDL Imitation: Example

```
// countup_tb testbench
module countup_tb ();
wire  [31:0]counter;
reg   [31:0] upto;
reg   enable;
reg   reset;
reg   clk;
wire  counter_done;

  // instantiate the module
  countup uut (clk,enable,reset,upto,counter,counter_done);

initial
  begin
   $monitor("%b %d %b",clk,counter,counter_done); // Print the welcome message
   clk   = 0;
   reset = 1;
   enable = 1;
   upto    = 10;  // set target to count up to

   #5 clk = ~clk; // apply the reset
   reset = 0;
   #5 clk = ~clk; // apply the dropped reset

   repeat (20)    // print tables of register log
     begin
      #5 clk = ~clk;
     end

  end
endmodule
```

6. Test the HDL and make sure it is also working.

Let's see what happens when run…

Try on: https://www.edaplayground.com/x/4ELg

HDL Imitation: Example

```
iverilog '-Wall' '-g2012' design.sv testbench.sv && unbuffer vvp a.out
0  0  0
1  1  0
0  1  0
1  2  0
0  2  0
1  3  0
0  3  0
1  4  0
0  4  0
1  5  0
0  5  0
1  6  0
0  6  0
1  7  0
0  7  0
1  8  0
0  8  0
1  9  0
0  9  0
1  10  1
0  10  1
1  10  1
0  10  1
Done
```

6. Test the HDL and make sure it is also working. (run the Verilog version to see same result as for C version)

```
$ CasHDL
Test C-like-HDL Module!
  0,0000000,0
  1,0000000,0
...
  0,0000002,0
  1,0000002,0
...
  0,0000009,0
  1,0000009,0
  0,0000010,1
  1,0000010,1
  0,0000010,1
...
  0,0000010,1
  1,0000010,1
```

Can see that they both deliver the same result, equivalent operation.

Benchmarking HDL
&
Amdahl’s for FPGA

But First …

Terminology buffs?

- Every heard of DMIPS?
- In relation to a VAX?
- How bizarre... how is that possibly of any relevance to HPEC or FPGAs?...

Well, let's find out in the next slide ...

Why MIPS and FLOPS are not enough

- Limitations of MIPS and FLOPS
  - MIPS alone are not all that meaningful for benchmarking because 1 CISC instruction may be worth many RISC instructions (but the CISC might still complete the task *faster*)
  - Similarly MFLOPS alone, while a bit more useful, do not give a sufficiently full picture, the processor could do lots of FLOPS but be low on other things (e.g. memory operations)
- DMIPS =
  - Dhrystone MIPS (Million Instructions Per Second). Shows number of iterations of the Dhrystone loop repeated per second. More holistic performance measure aligned to likely processing needs
  - DMIPS = Dhrystone_score / 1,757
  - The value 1,757 is the number of Dhrystones per second obtained on the VAX 11/780, nominally a 1 MIPS machine

Whetstone, Dhrystone and CoreMark

I'll explain each of these …

Whetstone, Dhrystone and CoreMark

- Whetstone is a collection of commonly used computation tasks, repeated in a loop, and the time the loop takes to complete equates to the Whetstone rating.

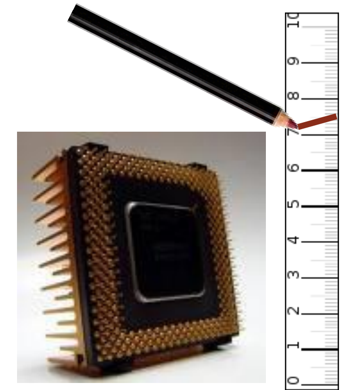
For further details see: http://www.coremark.org/home.php

Whetstone, Dhrystone and CoreMark

- The Dhrystone benchmark contains no floating point operations; it is works similarly to the Whetstone, but uses computations appropriate for fixed-point or integer based applications.

For further details see: http://www.coremark.org/home.php

Whetstone, Dhrystone and CoreMark



- CoreMark is a smaller benchmark
- Developed by the Embedded Microprocessor Benchmark Consortium (EEMBC)
- Focuses on the CPU core, similar to Dhrystone.
- CoreMark is intended to
  - Execute on any processor, incl. small micro-controllers.
  - Avoid issues such as the compiler computing the work during compile time
  - Use real algorithms rather than being mostly synthetic.
  - CoreMark has established rules for running the benchmark and for reporting the results.

For further details see: http://www.coremark.org/home.php

Relevance to FPGA

- Clearly Whetstone, Dhrystone and CoreMark are relevant to HPC generally (and were originally developed with microprocessors in mind)
- HOWEVER: These techniques apply to FPGAs as well, especially nowadays where you may want to use an FPGA for e.g. intensive signal processing and want to compare your FPGA implementation to a more standard CPU implementation.

Using Dhrystone with an FPGA

(a case study)

Hint: if you want to be rather ambitious and fancy you might consider using a benchmark approach similar to this in your YODA project!

Case Study

Using Dhrystone with an FPGA

Example of when & why you might use a benchmark such as Dhrystone on a FPGA.

Clock Increase	Clock Speed (MHz)	DMIPS	Perfect linear
1	100	135	-
2	200	271	270
3	300	407	406.5
4	400	542	542.531

The below right presents results on an investigation by Glover (2005) on running the PowerPC softcore processor on an FPGA using different configurations. As you can see, the Dhrystone performance of the platform in response to increased clock speed was pretty much 1:1. This was not necessarily expected as increasing the clock could cause higher temperatures and greater resistance in gate delays.

One common representation of the Dhrystone benchmark is DMIPS, which is obtained when the Dhrystone score is divided by 1757, as follows:

DMIPS = 833333.3/1757 = 474

DMIPS/MHz = 1.56

Table 2 contains the DMIPS values for various PowerPC clock frequencies when running 333 million iterations.

Table 2: **DMIPS Values**

CPU Clock Frequency	Inlining Disabled		Inlining Enabled	
	DMIPS/MHz	Dhrystone MIPS	DMIPS/MHz	Dhrystone MIPS
100 MHz	1.35	135	1.56	156
200 MHz	1.35	271	1.56	312
300 MHz	1.35	407	1.56	468
400 MHz	1.35	542	1.56	628

Conclusion

The Dhrystone benchmark is a general-performance benchmark used to evaluate processor execution time. As illustrated in this application note, the embedded PowerPC core in the Virtex-II Pro delivers 600+ DMIPs with a processor clock rate of 400 MHz.

Ref source: “Running the Dhrystone 2.1 Benchmark on a Virtex-II Pro PowerPC Processor” by Paul Glover, 2005.
Available: https://www.xilinx.com/support/documentation/application_notes/xapp507.pdf

Applying Amdahl to FPGAs

(a case study)

<u>Hint:</u> you might want to consider using this sort of approach in your YODA project!

Another hint …. Your lecture might well be infatuated with this topic of applying Amdahl to FPGAs, so it might well appear in a test or exam ☺

Applying Amdahl to FPGAs

When contrasting FPGA-based solutions to CPU-based solutions, in considering speedup of an operation, the comparison is likely around a multicore perspective, i.e. looking at both the FPGA side and CPU side fitting in with Flynn's MIMD model (i.e. multiple instructions on multiple different data source – see diagram on right).

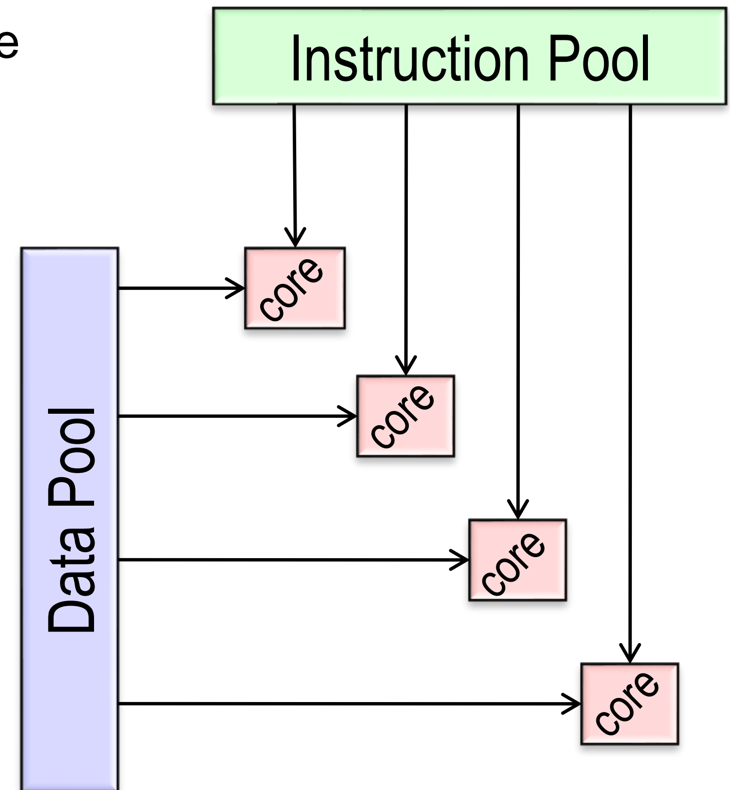
$$\text{Speedup} = T_{p1} / T_{p2}$$

Where
T_{p1} = Run-time of original (or non-optimized) program
T_{p2} = Run-time of optimised program

<u>Amdahl's Law</u>

$$\text{Speedup}_{\text{parallel}} = \frac{1}{(1 - f) + \frac{f}{n}}$$

f = fraction of computation that can be parallelized
1-f = fraction that cannot be parallelized / startup



Applying Amdahl to FPGAs

CPU-based

- Initialization
  - Loading in the data
  - Creating the threads
  - Data partitioning
  - Starting the threads
- Parallel Work
  - Threads working on tasks
  - Possible comms/IO blocks
- Join / finalizing (if need)
  - Waiting for threads to complete
  - Combining results etc.

FPGA-based

- Initialization / start-up
  - Programming the FPGA
  - Reset operations
  - Host->FPGA comms; configuring cores (setting parameters / regs)
- Parallel Work
  - Cores doing processing
  - Possible IO/synch blocks
- Finalizing (if need)
  - Doing clean-up operations
  - FPGA->Host comms; e.g. writing results back to host.

Applicability of Amdahl to FPGAs

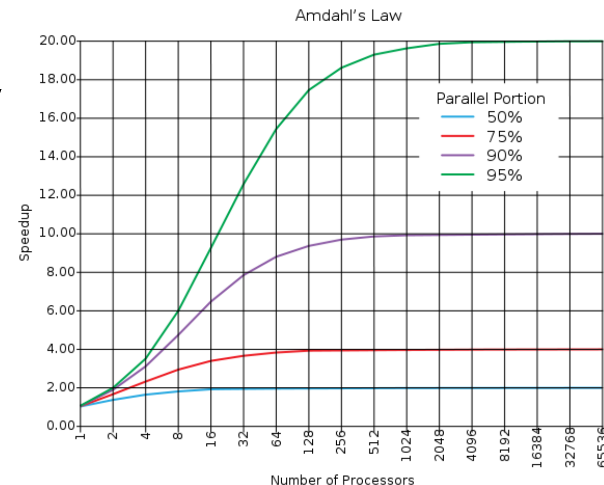
<u>(Basic) Amdahl's Law for FPGA-based processors</u>

$$Speedup_{parallel} = \frac{1}{(1-f) + \frac{f}{n}}$$

f = fraction of computation running on the cores
1-f = fraction of start-up and configuration time
n = number of parallel processors / cores

BUT a major problem with this is that 'n' is a potentially faulty component for CPU vs FPGA or even FPGA vs FPGA performance predictions. It assumes an approach of using multiple of the same cores to boost the parallel performance.

So, applying Amdahl in this case is not necessarily fair or logical… although one often does want a means to compare speedup between a CPU-based and a FPGA-accelerated system. Some parts of Amdahl's basic formula given here is useable, but you are comparing potentially very different systems, e.g. a bit like comparing a rocket to an airplane for getting a payload from A to B; they can both get the job done but they have different loading and other mechanisms to do so.. ultimately it is the speedup T1/T2 that you want out at the end.



Still going to have a similar view *if* you assume that the parallel section provides acceleration as doing the parallel part 'n' times as fast.

End of Lecture

Back to some Verilog…

(next learning set)

Disclaimers and copyright/licensing details

I have tried to follow the correct practices concerning copyright and licensing of material, particularly image sources that have been used in this presentation. I have put much effort into trying to make this material open access so that it can be of benefit to others in their teaching and learning practice. Any mistakes or omissions with regards to these issues I will correct when notified. To the best of my understanding the material in these slides can be shared according to the Creative Commons "Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)" license, and that is why I selected that license to apply to this presentation (it's not because I particulate want my slides referenced but more to acknowledge the sources and generosity of others who have provided free material such as the images I have used).

Image sources:
Flickr
Pixabay http://pixabay.com/ (public domain)
Product logos/icons from applications concerned
Chip image – Wikipedia open commons
Desert photo snippet – segment from photo on flickr
Ruler – Open Clipart www.openclipart.org (public domain)
ImpulseC – images from http://www.impulsec.com/products.htm