You only use 10% of your FPGA - Optimisation techniques for high FPGA utilisation

Quentin Schibler
Department of Computer Science and Technology, University of Cambridge
École Normale Supérieure Paris - Saclay.
qlas2@cl.cam.ac.uk

Background

Performance of RISC-V softcores

We see a use for softcore processors primarily as a robust replacement for complex state machines often needed in FPGA designs. Such a CPU needs to be quite **small** and **utilise the FPGA well**, as it trades off performance for ease of use. RISC-V is a well-suited ISA for such CPUs thanks to its modularity and support for compressed instructions. It is no surprise that FPGA vendors as well as the open-source community built RISC-V soft cores. Our **Rope** core increases utilisation using a higher clock frequency.

	Stratix 10	Ice 40	UltraScale
Intel NIOS-V	323 MHz / 1800 ALMs	Not known	Not known
Neorv32	311 MHz / 744 ALMs	92 MHz / 1130 LCs	347MHz / 751 LUTs
Picorv32	434 MHz / 724 ALMs	75 MHz / 1300 LCs	500MHz / 761 LUTs
Rope	745 MHz / 620 ALMs	180 MHz / 1200 LCs **	TBD

Table 1. Implementation performance of a variety of RISC-V softcores. ** experimental results, might change significantly.

Barrel processor architecture

Our **Rope** core exploits barrel processing to achieve high clock frequencies. Barrel processors are a type of multithreaded processors sharing a single pipeline between harts. Each stage executes a different hart every clock cycle. The pipeline can be simplified as **no data hazard** can occur, and **no branch prediction** is needed. This helps to reduce the area and doesn't penalise deep pipelines as much. The logic utilisation is also better as the pipeline is fully utilised on every clock, there are no bubbles, and no flushes are needed. The downside is that serial execution speed is poor, software needs to utilise every thread efficiently.

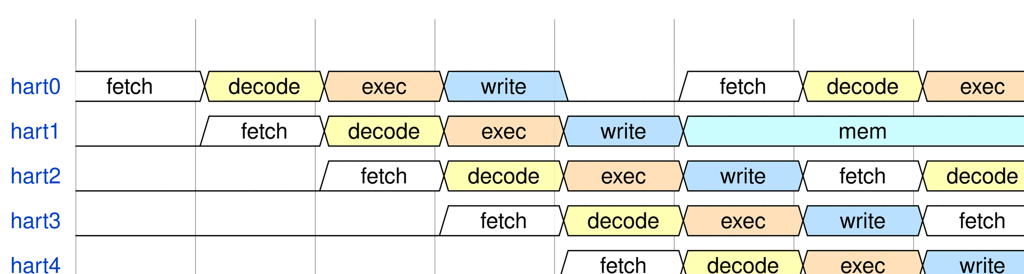

Figure 1. Execution of 5 harts in a 4 stage pipeline. As long as only one hart is waiting for memory, utilisation is maximised.

Diversity of FPGA architectures

FPGA vendors have different strategies to achieve high-performance designs on their architectures. Intel has a huge number of registers inside of the interconnect, encouraging heavy use of pipelining. On the contrary, small FPGAs like the ICE40 don't have any registers outside of the LUTs, long pipelines will greatly increase area usage. AMD has an hybrid approach with some register inside of the interconnect, and a NoC managing the routing. A barrel architecture has the added benefit that it is easier to adapt to a variety of platforms, thanks to its simplicity. Rope has a concept of a **physical pipeline** stage number that can be parameterized depending on the FPGA used to improve timing/area. The logical representation of the pipeline stays the same

Generic optimisation techniques

Timing analysis strategies

Timing analysis is a complex process, especially as the design grows larger. Place and route algorithms often **hide optimisation opportunities** as they sometimes spend lots of effort on one poorly optimised path, at the expense of others. They then show up as critical paths. Finding the real culprit becomes harder the more paths you have. Reasoning about a **smaller part of your design** is thus important, and brings added benefits, such as reducing compilation time and speeding up iterations. To avoid getting optimistic results from CAD tools, a framework for timing analysis is needed.

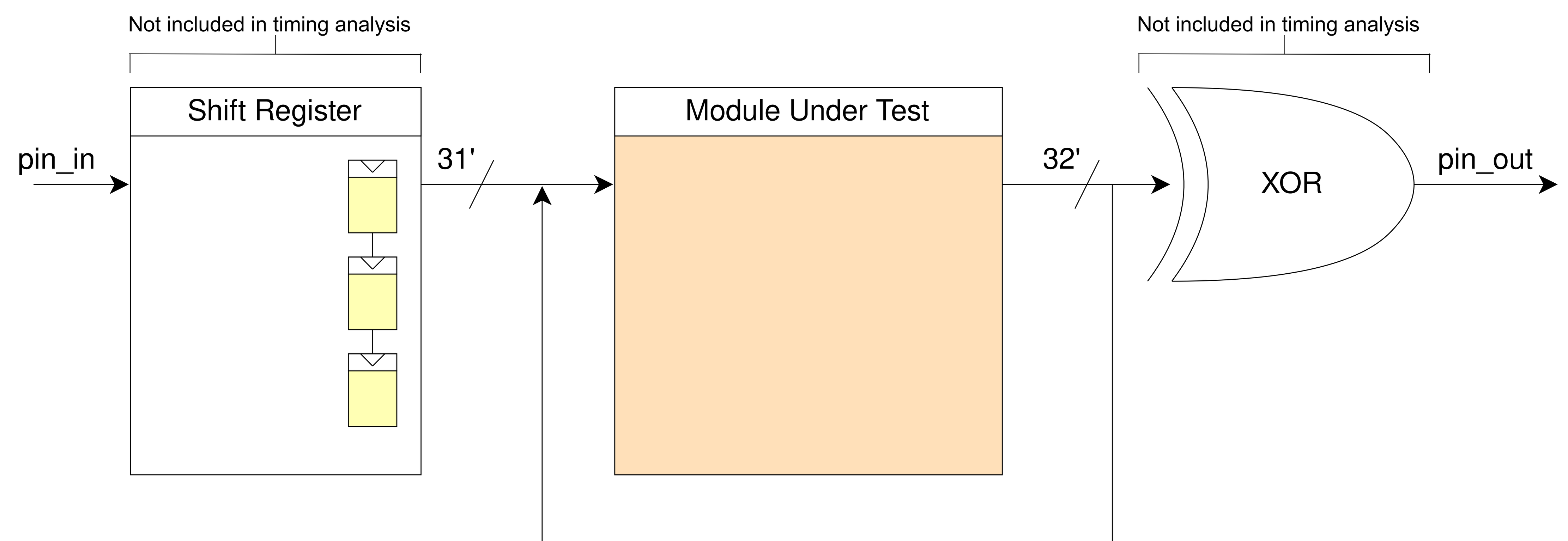

Figure 2. Framework for analysing timing. The shift register and the XOR gate makes it possible to use only two physical FPGA pins regardless of the module I/O width. The loopback around the module makes for a harder P&R problem.

Managing resets

Resets should be kept to a minimum as they tend to be high fanout signals, that increase the place and route problem complexity. On some FPGA architectures like Intel's hyperflex, resets prevent CAD tools from performing crucial optimisations. Various strategies can be used to eliminate the need for resets. Resetting data (as opposed to control) registers is usually not needed, as a valid bit can be used instead, or a handshake mechanism. Resetting a pipe doesn't require resetting every register in the pipe, but only the first one, and then clocking the design to propagate the reset value down. When a reset signal is truly needed, a **reset tree** can be used instead to limit high fanouts. This can be done automatically by some CAD tools, but better results can be obtained manually, and the design will work as expected even using tools that do not support that feature.

Platform dependent optimisation techniques

Retiming into Intel's hyper-registers

Intel FPGAs benefit from splitting combinatorial logic into lots of pipeline stages, to make use of **hyper-registers**. Unfortunately, other architectures will suffer from that design choice. Changing the number of stages would require a major redesign for every platform. Instead, a buffer with a FIFO-like interface can be placed after large chunks of combinatorial logic.

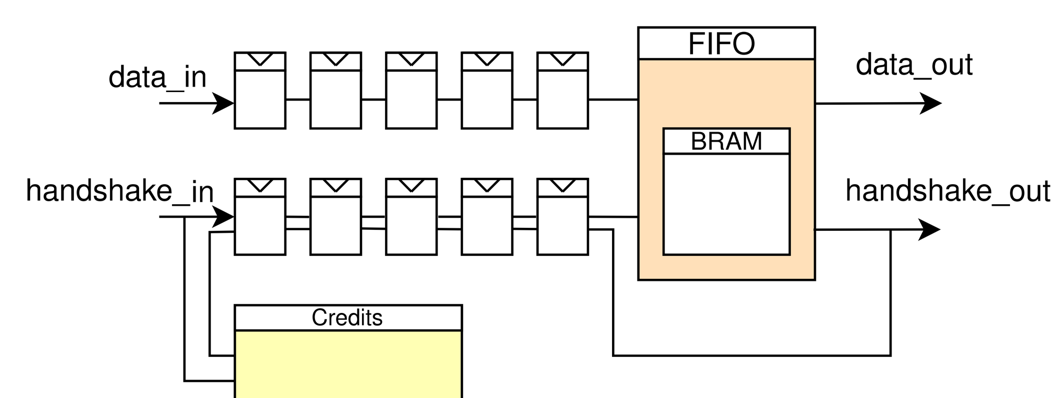

Figure 3. Credit buffer, handshakes can be implemented using valid/ready signals

The inputs to that buffer are sent to a register pipe without reset nor enable signals, and a credit buffer counts the difference between input and output handshakes. A FIFO absorbs any data left on the pipe while the output is stalled. The register pipe will be implemented as hyper-registers, and they will be retimed into the preceding logic. Optimising the ALU for **Rope** did not require changing the design, only changing the depth of the buffer. On the timing framework, it reached a frequency of 910MHz with an 8-depth buffer, from 330MHz with a 0-depth buffer.

Using adder for fast carry computation

On architectures where deep pipelining doesn't make sense, a carry-select adder can improve timing significantly. The hard part is computing the carry-in for any block, as it requires the carry-in from the previous block. This requires routing the signal to multiple LUTs chained together. On most FPGAs, LUTs have dedicated **fast routing logic for carry lines** when used as adders. The carry computation can be expressed as an addition to make sure the CAD tool uses the fast carry lines.

Figure 4. 4-bit carry-select adder. The Carry Module computes the carry for the next block using the following equation : $C_i = C_{i-1} + C_i^0 + C_i^1 + 2$
Where C_i is the carry in for block i, and C_i^0, C_i^1 the precomputed carries.