You only use 10% of your FPGA

Quentin Schibler[1,2]

[1]Computer Laboratory, University of Cambridge
[2]École Normale Supérieure Paris-Saclay

Abstract

A RISC-V processor design is usually expected to be clocked at most at 300MHz on the latest high-end FPGAs, 100MHz being a reasonable average guess. One day I opened up a data sheet and read that Intel's Stratix 10 routing fabric can be clocked at 1GHz. This is one order of magnitude higher, but can we reach such a high frequency, and if not, what is the practical limit? As I set out to explore how fast and resource-efficient a RISC-V CPU can be on FPGAs, I will share a few guidelines I discovered along the way. Topics will include various strategies to make timing analysis doable for large designs, how to take advantage of hyper-registers, reducing area using BRAMs without sacrificing performance, and managing resets and other high-fanout signals.

Timing Analysis

We developed a technique for doing timing analysis on subsets of a design, based on the work in [1]. This simplifies timing analysis by using a divide and conquer approach. This is because reasoning about a small part of your design is simpler, and brings compilation time down, speeding up iterations. We found that the timing for our designs was within 5% of the worst timing measurement for each subset. Thanks to that observation, we will use a subset of the CPU, for the sake of simplifying, as an example for exploring guidelines in the following sections. We chose a FIFO for that purpose, as it is a simple but significant component that motivated some significant design choices.

Hyper-registers and BRAM

A common optimization used by P&R algorithms is to retime registers to improve timing. We successfully did so to increase the frequency of a 32-bit adder from 300MHz to 450MHz, without noticing any significant improvements by adding more than two pipeline stages. However, by getting rid of register reset, we are able to retime into hyper-registers instead. Those were able to cut carry lines, improving timing significantly, almost reaching the 1GHz fabric limit with 3 pipeline stages. We are able to exploit the latency/frequency trade-off to our advantage.

Table 1: *Area/Frequency of BRAM and LUT 32-bit FIFO for different depth*

	BRAM	LUT
2-depth	5 LUT 417MHz	72 LUT 949MHz
8-depth	8 LUT 400MHz	173 LUT 772MHz
16-depth	17 LUT 389MHz	365 LUT 663MHz

We will now add one constraint to our CPU design: every component has to use elastic pipelines. This makes it easier to modify and extend the design. This requires sending every computation result into a FIFO. According to [2], the depth of the FIFO must be large enough to absorb a full pipeline flush, so 8 or 16-depth FIFOs are common in our design. Using a BRAM for those is the obvious choice, but as we see in Table 1, it can be beneficial area-wise even for 2-depth FIFO. The ratio of LUT to BRAM for Stratix 10 devices is around 80 LUT/BRAM. To keep a balanced design, we would ideally use one BRAM for every 80 LUT. To put that into perspective, the ALU we designed takes roughly 250 LUT.

Timing is an issue though, but it can easily be improved. First off, we can pipeline the outputs into hyper-registers, adding one cycle of latency between a read request and actually receiving the result. Using LUTs instead doesn't work well, as too much slack is needed to go from the BRAM

block to the LUT. The maximum frequency of BRAMs can only be achieved if we never read and write to the same address on the same clock cycle. This is because logic would need to be created to save the old data, or to forward the new data. For our FIFO, we can simply add a cycle of latency between a write and a read (but still can issue one read and one write every cycle). Using both techniques, we are able to improve the FIFO operating speed up to 800MHz. However, we added 2 cycles of latency, a cost we will pay on top of the latency of the rest of the design. As an example, to reach a frequency of 890MHz for our 32-bit ALU, we needed 8 cycles of latency. We can already expect the CPU pipeline to be extremely long. To mitigate this issue that could lead to costly pipeline flush later down the line, we decided to choose a barrel CPU design. This is because it does not suffer from hazard, and thus does not need pipeline flush.

Managing resets

As we have seen, resets should be limited to a minimum so that CAD tools have the most freedom to retime into hyper-registers. Various strategies can be used for that purpose. 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. Applying this technique to our FIFO, we can avoid resetting the head and tail pointer. Instead, we can discard the first few results from the FIFO until both pointers are equal again. This allows Quartus to retime them to hyper-registers, improving performance up to 993MHz. When you are forced to use a reset signal, use a reset tree instead to limit high fanout. In most CAD tools, register pipes can be duplicated automatically into trees, balancing the fanout as much as possible.

References

[1] Charles Eric LaForest. "High-speed soft-processor architecture for FPGA overlays". In: 2015.

[2] Mustafa Abbas and Vaughn Betz. "Latency Insensitive Design Styles for FPGAs". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 360–3607. DOI: 10.1109/FPL.2018.00068.