

Custom Processors in FPGAs

You can build a custom processor with the newest Xilinx FPGAs.



by Ilya Tarasov
Associate Professor
KSTA
ilya.tarasov@inlinegroup.ru

Processor cores, systems, and designs are gaining ground in the FPGA solution portfolio. Xilinx® MicroBlaze™ and PicoBlaze™ processors, along with the EDK development tool, have paved the way to processor-based systems on a single chip.

You are not, however, limited to these solutions only – you can always implement a soft processor with your desired features. In this article, which is devoted to soft processors in general, I'll attempt to determine when it is preferable to implement custom soft-core processors instead of proven Xilinx solutions.

How Custom Processor Design Works

Before discussing why you might need to design a custom processor, let's take a look at the design methodology. I'll use a very simple example of a processor to show the main steps and what problems you might encounter.

The heart of the processor is a finite state machine (FSM). An FSM is quite effective because the latest FPGAs are synchronous by nature; therefore, an FSM is a good base from which to describe processor behavior. Being an FSM, the processor must change its state at the rising edge of the clock signal.

Let's take a straightforward example where we have three 8-bit registers: PC (program counter); RA and RB (general-purpose registers); and an 18-bit command bus named "cmd." This scheme is very efficient for an FPGA with 1,024 x 18-bit organization. Figure 1 shows the behavioral code for this example.

This example illustrates the implementation of a fully synchronous digital design. If you can provide an appropriate sequence of commands for each new program counter value, you'll get a programmable state machine; in other words, a processor.

Of course, you may want to use on-chip FPGA resources for a complete system-on-chip solution without any external memory components. But looking at block RAM waveforms, you might find it necessary to spend at least one clock cycle to read command from memory. To correct this, let's implement a two-stage FSM where the first stage is "read" and the second stage is "execute," shown in Figure 2.

Why (and When) to Opt for a Custom Processor

Designers familiar with the MicroBlaze processor and EDK software may ask why we need yet another core, or how it could replace proven, high-performance products. Indeed, you don't need another processor in general if you can perform a common task using well-known approaches.

But because the MicroBlaze processor is a 32-bit RISC core, it is possible to achieve a significantly different solution. Let's analyze when these steps might lead to valuable results.

DSP Algorithms with High Parallelism

In this case, FPGA performance is determined by the structure of hardware multiplier blocks, which realize certain algorithms of data processing. There may be many such data paths with embedded

multipliers and XtremeDSP™ solution slices, and no processor core can achieve comparable benchmarks given the highly paralleled nature of the DSP block array.

You may want to control this array with standard interfaces and protocols, collect statistics, and perform additional tasks in each DSP block. This is a job for processor cores. But you can't just place the same number of MicroBlaze processor cores as embedded multipliers on the FPGA. For these purposes, simple processor cores with little register sets, short commands, and small command memories (even based on distributed RAM) may prove very useful.

Non-Symmetrical Register Architecture

A MicroBlaze embedded processor realizes a symmetrical three-address register architecture. This means that any registers may perform the same operations (with few exceptions for MicroBlaze processors), while general commands may choose operands and destinations separately.

A three-address architecture with many registers is good for compilers, because many variables can be loaded into the processor without needing to be stored back into memory. If you have fewer registers, some variables may be forced to unload when the number of active variables

```
-- register declaration omitted for saving article space
process(clk)
begin
  if rising_edge(clk) then
    case conv_integer(cmd) is
      when 0 => pc <= pc + 1; -- no operation, but we must remember to increment program counter
      when 0x100 to 0x1FF => RA <= cmd(7 downto 0); pc <= pc + 1; -- load imm value to RA
      when 0x200 to 0x2FF => if conv_integer(RA) = 0
                            then pc <= cmd(7 downto 0);
                             else pc <= pc + 1;
                            end if; -- jump if RA = 0
      -- we can add any 'simple' commands, which will affect any desired registers
      when 1 => RA <= RA + RB; pc <= pc + 1;
      --- etc, with format: when <value> => RA <op> RB; pc <= pc + 1;

      -- also, let's look on the parallel data processing and combined commands
      when 100 => RA <= RA + 1; RB <= RB - 1; pc <= pc + 1; -- different operations with two registers

      -- we may also add other complex commands, which have no direct analogs in common instruction sets

    end case;
  end if;
end process;
```

Figure 1 – Simple decoder and ALU for custom processor

```
-- in architecture section
signal st : bit; -- state of core: 0 - fetch, 1 - execute;
-- after 'begin' keyword
process(clk)
begin
  if rising_edge(clk) then
    case st is
      when '0' => st <= '1'; -- simple wait for BRAM, go to EXECUTE state
      when '1' => st <= '0'; -- return to FETCH state
      case conv_integer(cmd) is
        when 0 => pc <= pc + 1;
        ... -- all of the 'case' code from the previous example
      end case;
    end case; -- st
  end if;
end process;
```

Figure 2 – Simple processor engine for use with block RAM

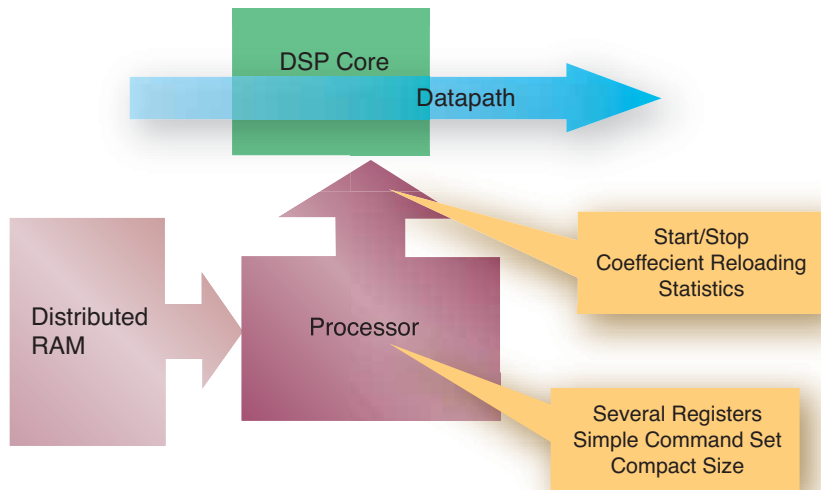


Figure 3 – A DSP coprocessor with general-purpose core

becomes greater than the number of registers. (This is not a problem for less complex algorithms, when calculations are simple enough to impose some limitations on processor architecture.)

For example, you can use a dedicated accumulator to act as a destination register for all ALU activity. This allows command width to shrink because you can exclude information about destination registers (it is always known). Looking further, it is possible to realize a zero-operand processor (with a stack architecture like Forth processors or Java machines) with the smallest possible command width. Code compactness is pretty useful in the FPGA world because of the limited amount of on-chip block RAM.

I/O-Oriented Project

Here's another uncommon approach to I/O module integration. In generic processor cores, it is necessary to assign some addresses to each I/O port and then write low-level drivers that will exchange with these I/O cores through in and out commands.

But if you implement any high-performance-critical I/O cores, you may want to boost and simplify data exchange with them. You can also turn to dedicated buses for modules with dedicated data exchange commands, too. This takes additional resources and adds special commands, of course, so the result will not be

universal (indeed, you can't add hundreds of I/O modules this way as you can for an on-chip peripheral bus). However, it may be an "ad hoc" solution.

Advantages of FPGA-Based Processors

The unlimited reprogrammability of FPGAs allows you to realize very uncommon instructions. This is important, because it is virtually impossible to add any non-standard, uncommon instruction to ASIC processors without ensuring its further usage. If you don't meet the market requirements for ASICs, you'll lose time and money, but an experimental instruction set costs almost nothing with FPGAs.

If an experimental architecture with additional commands is not required, you can simply reset the FPGA configuration and download the MicroBlaze processor as a proven, supported solution. Experimental architectures may be very effective for particular purposes, and we can always try them at the beginning of a new project. Uncommon algorithms, parallel calculations, or operations with ultra-wide numbers are quite appropriate reasons to use custom processor architectures.

Xilinx FPGAs have a complex set of features that make all processor implementations effective. For example:

- The balanced set of resources and register-rich architecture allow for imple-

mentation of deeply pipelined, register-rich processor cores.

- On-chip, dual-port block RAM stores program and data on the same chip to make FPGAs a system-on-chip. Two independent ports also extend memory bandwidth and provide easy downloading and debugging.
- Distributed RAM in SliceM allows you to create small memory sections, such as register files, buffers, stacks, and FIFOs. All of these are important parts of a processor system and can significantly improve performance and functionality.
- The newest Xilinx Spartan™-3AN FPGAs with internal flash memory allow the implementation of a single-chip system with as much as 11 Mb of flash memory and 72 kb of dual-port SRAM. This is enough to complete a serious embedded system-on-chip.
- High-performance Virtex™-5 FPGAs provide better results for processor soft cores, thanks to their six-input look-up tables (LUTs). In real projects (with ALU based on combinatorial logic), this makes it possible to implement the processor with fewer logic levels than with previous Virtex platform FPGA generations (and other FPGAs with four-input LUTs). In general, when compared to Virtex-4 devices, processor core designs become 30% faster.

Conclusion

Now that designers have access to real hardware prototyping devices, it is high time to take a fresh look at the embedded processor world. With powerful and easy-to-use development tools like ISE™ software, you can easily try processor designing. This is a promising field for research teams, qualified designers, those who need a specific computing platform, and IT enthusiasts.

CTC Inline Group in Russia, a training center, is now holding training courses on this topic; additionally, you will find a non-commercial community of those creating experimental processor implementations with FPGAs at www.ffmpeg.winglion.ru (with text in both Russian and English). 🌟