THE CASE FOR SOFT-CPUS IN ACCELERATOR CONTROL SYSTEMS

W. W. Terpstra, GSI Helmholtz Centre for Heavy Ion Research GmbH, Darmstadt, Germany

Abstract

The steady improvements in Field Programmable Gate Array (FPGA) performance, size, and cost have driven their ever increasing use in industry, science, and IT. As FPGA sizes continue to increase, more and more devices and logic move from dedicated chips to FPGAs. For simple hardware components, the savings in board area and chip count are compelling. For logic involving dynamic memory and complicated control flow, the trade-off is not as clear.

Traditionally, this has been the domain of CPUs and software programming languages. In hardware designs already including an FPGA, it is tempting to remove the CPU and implement all logic in the FPGA. However, if that logic is then be implemented in the more constraining hardware description languages, it cannot be as easily debugged or traced, and typically requires significant FPGA area. For performance-critical tasks, this trade-off can make sense. However, for the myriad slower and dynamic tasks, software programming languages remain the better choice.

One great benefit of a CPU is that it can perform many tasks. Thus, by including a small "Soft-CPU" or softcore inside the FPGA, many low performance tasks can be aggregated into a single component. These tasks may then reuse existing software libraries and debugging techniques, while retaining ready access to the FPGA's internals.

This paper discusses requirements for using Soft-CPUs in this niche, especially for the FAIR project[1]. Several open-source alternatives will be compared and recommendations made for the best way to leverage a hybrid design.

BACKGROUND

FPGA chips allow developers to implement custom circuitry. A developer designs his desired circuit in a Hardware Description Language (HDL), and a compiler translates this into the underlying gates and wires required by the design. The target FPGA chip can then be programmed with the resulting bitstream. A given FPGA has a limited number of wires and gates and when a design is loaded, it consumes some of the *area* on the chip.

Developers organize their HDL project into components. A component corresponds roughly to a chip design, with internal logic and input/output pins. For example, a component might implement a memory chip with address and data pins. Each component can be instantiated multiple times, like placing several chips with the same specification. These instances can in turn be wired together to produce a larger component, akin to wiring several chips together on a card. The resulting amalgam component will have it's own input/output pins. This hierarchical composition results in a tree of nested component instances.

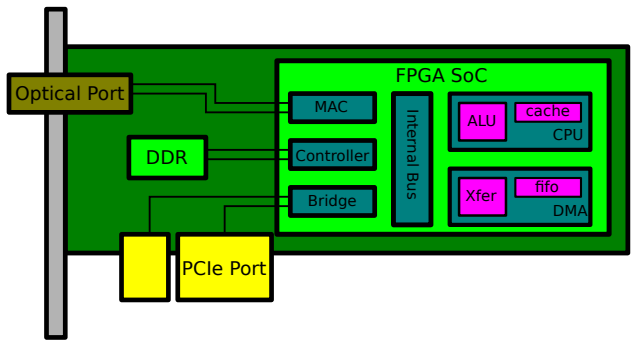

Figure 1: A SoC including a Soft-CPU.

Typically, the top-most component or root has its input/output pins mapped to the physical pins on the FPGA. The FPGA's pins are then physically wired to external chips mounted on the board and the connectors required by the type of device manufactured. For example, in the FAIR project, our control devices have FPGA pins connected to a DDR memory chip, an optical network transceiver, and a PCI express connector.

Within the FPGA, to ease interconnection of several components, developers often use a bus protocol. Just like the PCI express bus of a PC, this allows developers to plug together multiple components without needing to modify their designs. The resulting FPGA is called a System on Chip (SoC), because it is effectively an entire computer in a single chip. The key advantage of a bus is that it allows multiple logical connections. Where a simple wire between components only connects those two components, a bus allows n master components to control m slave components, resulting in nm logical connections. Figures 1 and 2 illustrate two SoC designs.

Within a SoC system, one can include a CPU. This CPU is typically a bus master which controls the slave devices on the SoC bus. As the CPU is not a physically distinct chip, and is implemented in HDL within the design, it is called a Soft-CPU or SoftCore.

This paper will cover: when it makes sense to include a Soft-CPU, what they require from the memory subsystem, an analysis of the best available Soft-CPUs, and our recommendations.

[1] A new international accelerator facility for antiproton and ion beams.

WHEN TO USE A SOFT-CPU

Soft-CPUs are often, but not always, a good thing to include in a SoC design. On a modern FPGA the area they consume is relatively minor; an LM32 Soft-CPU uses 2-4% of a EP2AGX125, depending on selected features. However, they typically impose a hefty memory cost, discussed later. Furthermore, they introduce another master running on the bus, which may not be desirable due to arbitration.

Comparison to Custom HDL

Any functionality one can implement in software, one can also implement directly in HDL; it's just more work. For example, `memcpy(dst, src, len)` run on a Soft-CPU can move data from one slave device to another. In HDL one could implement a DMA controller to perform the transfer. Broadly speaking, the advantage of HDL is raw speed and software is ease of development. More specifically, the strengths of a Soft-CPU are:

- ✓ Soft-CPUs can run traditional C/Fortran/Java code. They can re-use existing libraries and applications.
- ✓ A single Soft-CPU can implement many features with one component; one processor can run multiple programs. In HDL each feature requires more circuitry.
- ✓ Execution order is easier in software. In HDL a state machine tracks the current operation. In software this is implicit in the program's instruction pointer.
- ✓ Dynamic resource management is easier in software. Software systems have a stack and a heap. Hardware only has static (in the C/C++ sense) variables.
- ✓ Software debuggers can single-step programs and inspect/modify every variable. Hardware signal tapping captures execution after matching crude triggers and can only inspect new variables with a recompile.
- ✗ Software is *much* slower. Hardware is always parallel, executing many things at once. However, Soft-CPUs can leverage this parallelism with custom instructions.
- ✗ Soft-CPUs require a memory subsystem.
- ✗ Soft-CPUs require an additional developer toolchain.

It is important to understand that Soft-CPUs and custom HDL are not mutually exclusive; the above pro/con list presents a false dichotomy. One of the key strengths of a Soft-CPU compared to an external CPU is that it is directly connected to customizable hardware. The question is not whether or not to use custom HDL. When using an FPGA that is a given. The question is if any of the required functionality could benefit from a Soft-CPU. If a Soft-CPU is useful, then a hybrid design should be used.

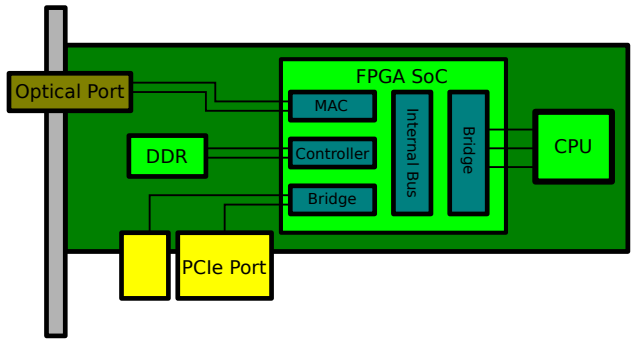

Figure 2: A SoC connected to an external CPU.

Comparison to an External CPU

If a CPU is required, instead of putting it inside the FPGA, one can attach it as an external chip. See, for example, Figure 2. All of the same trade-offs for custom HDL that apply to a Soft-CPU apply to an external CPU. However, Soft- and external CPUs have their own trade-offs.

The main advantage of an external CPU is performance. Soft-CPUs are as simple as possible to minimize the FPGA area required. Generally, their designs mimic RISC architectures from the mid 1980s. Modern superscalar CPUs are significantly more complex and much faster clock for clock. Furthermore, the fairly decent EP2AGX125 FPGA run an LM32 Soft-Core at 200MHz compared to the typical 3GHz clock rates of a modern CPU. Thus, an external CPU will dispatch over 30 times as many instructions as a Soft-CPU. Nevertheless, for a particular problem, custom hardware can be faster still and a Soft-CPU will be directly connected to that hardware.

Since the main advantage of an external CPU is raw performance, it will be run at a faster clock rate than the FPGA. Otherwise, one might as well use a Soft-CPU and save money. In this configuration, the trade-offs are:

- ✓ The Soft-CPU solution costs one less chip.
- ✗ An external CPU issues instructions $> 30\times$ faster.
- ✓ The Soft-CPU can use custom instructions.
- ✗ The external CPU runs a standard OS and toolchain.
- ✓ A Soft-CPU operates synchronously with the FPGA.
  - It is directly connected to the internal bus. There is no bridge, no variable latency.
  - It is in the same clock domain as the devices.

When money is no object, the main advantage of a Soft-CPU is deterministic execution speed; the Soft-CPU executes instructions at a known rate. In a hybrid design, it is possible for a custom HDL component to dispatch work to the Soft-CPU and be guaranteed to receive the result after a fixed number of cycles. Similarly, when the Soft-CPU pushes data to a device known to use 5 cycles, it doesn't

need to synchronize or check for completion. It just burns 4 cycles before reading the result. This tight integration makes it possible for a Soft-CPU to fill the not-easily-done-in-HDL gaps of a mostly custom HDL solution.

MEMORY ARCHITECTURE

The real cost of adding a Soft-CPU to a design is not FPGA area, but memory. The more complex the logic that the Soft-CPU executes, the larger the executable it must run. While it is true that a single Soft-CPU can implement many features in one component, the hidden cost is this increasing memory foot print.

Unfortunately, a larger memory footprint comes with decreased determinism. As the memory requirements grow, the memory must move further from the CPU. As the memory moves further from the CPU, the memory hierarchy deepens. Caches make an otherwise straight-forward program take an unpredictable amount of time. This weakens the greatest strength of a Soft-CPU. Unfortunately, if the problem requires more memory, this is unavoidable.

Certainly, one could imagine running a Soft-CPU with an SRAM chip for memory and no cache. An SRAM chip will take at least 4 cycles to provide each instruction to the Soft-CPU. FPGA internal cache memory could provide it in 1 cycle. If one runs the Soft-CPU with no cache or prefetch, it is exactly 4 times slower. With cache, it is at worst 4 times slower and at best full speed. Omitting the cache doesn't improve the deadline behaviour of the Soft-CPU; it just makes it (much) slower on average. Thus, when the Soft-CPU's memory footprint exceeds the memory available at one level in the memory hierarchy, that level becomes cache for the next, larger level.

The important message is that one size does not fit all problems. A good hybrid design uses the smallest amount of memory possible, and thus the least levels of cache.

Another factor which can impact determinism is the bus itself. If the Soft-CPU must share access to the memory with other devices, it may need to wait for those devices to finish their access. We strongly recommend that a hybrid design dedicate a memory port exclusively to the Soft-CPU's instruction bus. The data bus is much less frequently used (only on load/store instructions) and can share access with other devices as long as access is carefully scheduled.

MMU: Yes or No

A common feature found on modern CPUs is the Memory Management Unit (MMU). This introduces a level of indirect addressing which is needed by modern operating systems to implement inter-process memory protection. Some Soft-CPUs always include an MMU, some don't, and for some it is configurable. An MMU implementation is typically tightly integrated with the CPU cache system. Usually a Translation Lookaside Buffer (TLB) maps virtual addresses to physical addresses. When the TLB cannot find an entry needed, it must be refilled, introducing an additional source of non-deterministic delay. The trade-offs:

✓ Standard OS supported, meaning more compatability.

✓ Supports efficiently garbage collected languages.

✓ Fault isolation and memory error detection.

✗ TLB misses add non-deterministic execution delay.

✗ Memory required for page table management.

The main advantages of an MMU only apply to large software systems. Both a standard OS and a garbage collected runtime entail significant memory consumption. Furthermore, both of these scenarios make steep concessions in pause times. If one uses a Soft-CPU with an MMU, one has almost given up the main advantage of a Soft-CPU compared to an external CPU. Both the MMU and attendant cache system introduce execution pauses, as does a standard OS and/or garbage collected runtime. Aside from cost, little remains to recommend a Soft-CPU + MMU.

SOFT-CPU SHOWDOWN

To choose a Soft-CPU for use in the FAIR project, we made a list of requirements that candidate Soft-CPUs should fulfill. In order of decreasing importance:

- Open source HDL: port to many FPGA manufacturers, add required features, trace and debug problems.
- Toolchain support. No compiler, no point.
- Configurable memory subsystem.
- Mature and well documented.
- Size and Speed.
- Debugger support.

We only considered CPUs which met at least the first two requirements. The surviving candidates we investigated are summarized in Table 1.

RECOMMENDATIONS

Inspecting Table 1 makes pretty clear that the only reasonable options are the LEON3, LM32, and OpenRISC processors. All three processors are RISC and have similar area and performance. The other open source offerings are either incomplete, poorly documented, or too large for an FPGA (S1). Of the runner ups, the ZPU is probably the most interesting due to its smaller foot-print. However, it is significantly slower clock-for-clock than the three main candidates which issue most instructions in one cycle.

For a hybrid design, the LM32 is the clear winner. It has very clearly specified instruction timings and is designed to work without an MMU. It's memory subsystem is very configurable; the instruction bus can use either internal FPGA memory, an 1/2-way instruction cache, or direct bus access. This makes it easily accommodate designs

Table 1: Feature Breakdown of Available Open Source CPUs as Synthesized on an Arria2

CPU	Memory Subsystem	Documentation	Area	Speed	Debugger
LM32	optional(I+D cache)	Very Good [4, 5]	2000	200MHz	yes
LEON3	optional(I+D cache + MMU)	Excellent [1, 2, 3]	3000	150MHz	yes
OpenRISC	I+D cache + MMU	Adequate [6, 5]	3300	150MHz	yes
ZPU	none	Poor	1000	200MHz	no
ZET	none	Poor	3000	60MHz	no
S1	I+D cache + MMU	Sparc [3]	45000	-	-
CPU86	none	Non-existant	3800	100MHz	no
Plasma	none	Non-existant	-	-	no
Navre	none	Non-existant	1000	200MHz	no
pavr	none	Poor	2400	120MHz	no
aemb	none	Non-existant	1100	140MHz	no
openfire	none	Non-existant	1300	160MHz	no
pacoblaze	none	Poor	650	200MHz	no
yasep	none	Poor	-	-	no
dspuva16	none	Non-existant	340	250MHz	no

with different memory requirements. Finally, it is architecturally much simpler than a sparc. A hand-crafted operating system can easily fit in under 1kB of memory, or it can just be used directly as a micro-controller.

Both the LEON3 and the OpenRISC are well suited to running a full Linux operating system. However, the only thing which recommends them over using a much faster external CPU is cost. When running linux, the operating system, MMU, and cache all conspire to eliminate the benefit of a direct connection to the FPGA bus. An SoC bridged over PCIe to an external CPU would likely meet tighter real-time deadlines.

In conclusion, given the low area cost of a Soft-CPU and the many potential benefits, it seems wise to plan on including a Soft-CPU in most SoC designs. As long as the executable code can fit in FPGA memory, the design impact is low. At the GSI, we have now used the LM32 in several designs and found it very useful not only in offloading low-priority tasks from custom HDL, but also in debugging attached hardware devices.

APPENDIX: LM32 FEATURES

A quick overview of our recommended Soft-CPU:

- 6-stage pipelined RISC architecture
- 32-bit data path and instructions
- 32 general-purpose registers and interrupts
- Optional instruction and data cache
- Wishbone [5] memory interfaces (instruction+data)
- Debug unit with breakpoints+watchpoints
- Can both be debugged over JTAG or internally

REFERENCES

[1] Aeroflex Gaisler, “GRLIB IP Library Users Manual”, 2010. www.gaisler.com/products/grlib/grlib.pdf

[2] Aeroflex Gaisler, “GRLIB IP Core Users Manual”, 2010. www.gaisler.com/products/grlib/grip.pdf

[3] SPARC Int’l, “The SPARC Architecture Manual Version 8”, Prentice-Hall, Inc., Upper Saddle River, NJ, 1992, www.sparc.com/standards/V8.pdf

[4] Lattice Semiconductor Corporation, “LatticeMico32 Processor Reference Manual”, August 2007, www.latticesemi.com/documents/doc20890x45.pdf

[5] R. Herveille, “WISHBONE System-on-Chip (SoC) Interconnection Architecturefor Portable IP Cores”, 2010, cdn.opencores.org/downloads/wbspec_b4.pdf

[6] D. Lampret et al., “OpenRISC 1000 Architecture Manual”, April 2006, opencores.org/svnget,or1k?file=/trunk/docs/openrisc_arch.pdf