FPGA Embedded Processors

Revealing True System Performance

Bryan H. Fletcher
Technical Program Manager
Memec
San Diego, California
www.memec.com
bryan_fletcher@mei.memec.com

Embedded Training Program
Embedded Systems Conference San Francisco 2005
ETP-367

1. Abstract

Embedding a processor inside an FPGA has many advantages. Specific peripherals can be chosen based on the application, with unique user-designed peripherals being easily attached. A variety of memory controllers enhance the FPGA embedded processor system's interface capabilities.

FPGA embedded processors use general-purpose FPGA logic to construct internal memory, processor busses, internal peripherals, and external peripheral controllers (including external memory controllers). Soft processors are built from general-purpose FPGA logic as well.

As more pieces (busses, memory, memory controllers, peripherals, and peripheral controllers) are added to the embedded processor system, the system becomes increasingly more powerful and useful. However, these additions reduce performance and increase the embedded system cost, consuming FPGA resources.

Likewise, large banks of external memory can be connected to the FPGA and accessed by the embedded processor system using included memory controllers. Unfortunately, the latency to access this external memory can have a significant, negative impact on performance.

FPGA manufacturers often publish embedded processor performance benchmarks. Like all other companies running benchmarks, these FPGA manufacturers purposely construct and use an FPGA embedded processor system that performs the best for each specific benchmark. This means that the FPGA processor system constructed to run the benchmark has very few peripherals and runs exclusively using internal memory. The embedded designer must understand how a real-world design's specific peripheral set and memory architecture will affect the system performance. However, no easy formula or chart exists showing how to compare the performance and cost for different memory strategies and peripheral sets.

This paper presents several case studies that examine the effects of various embedded processor memory strategies and peripheral sets. Comparing the benchmark system to a real-world system, the study examines techniques for optimizing the performance and cost in an FPGA embedded processor system.

2. FPGA Embedded Processors

The Field Programmable Gate Array (FPGA) is a general-purpose device filled with digital logic building blocks. The two market leaders in the FPGA industry, Altera and Xilinx, are the focus of this study. Many other programmable logic companies exist, although their products are not discussed in this paper.

The most primitive FPGA building block is called either a Logic Cell (LC) by Xilinx or a Logic Element (LE) by Altera. In either case, this building block consists of a look-up table (LUT) for logical functions and a flip-flop for storage. In addition to the LC/LE block, FPGAs also contain memory, clock management, input/output (I/O), and multiplication blocks. For the purposes of this study, LC/LE consumption is used in determining system cost.

2.1 Soft vs. hard processor

Both Xilinx and Altera produce FPGA families that embed a physical processor core into the FPGA silicon. A processor built from dedicated silicon is referred to as a "hard" processor. Such is the case for the ARM922T™ inside the Altera Excalibur family and the PowerPC™ 405 inside the Xilinx Virtex-II Pro and Virtex-4 families.

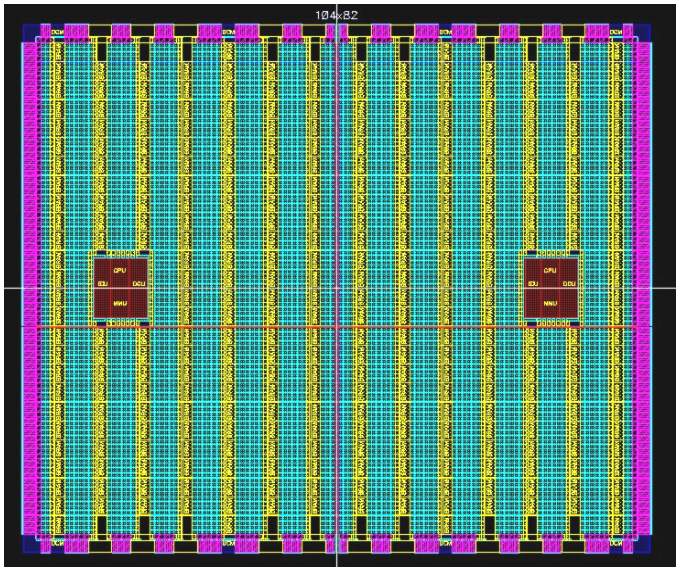

Figure 1 – Xilinx Virtex-II Pro Die Showing PowerPC Hard Processors

A "soft" processor is built using the FPGA's general-purpose logic. The soft processor is typically described in a Hardware Description Language (HDL) or netlist. Unlike the hard processor, a soft processor must be synthesized and fit into the FPGA fabric.

In both soft and hard processor systems, the local memory, processor busses, internal peripherals, peripheral controllers, and memory controllers must be built from the FPGA's general-purpose logic.

2.2 Advantages of an FPGA embedded processor

An FPGA embedded processor system offers many exceptional advantages compared to typical microprocessors including:

1) customization
2) obsolescence mitigation
3) component and cost reduction
4) hardware acceleration

2.2.1 Customization

The designer of an FPGA embedded processor system has complete flexibility to select any combination of peripherals and controllers. In fact, the designer can invent new, unique peripherals that can be connected directly to the processor's bus. If a designer has a non-standard requirement for a peripheral set, this can be met easily with an FPGA embedded processor system. For example, a designer would not easily find an off-the-shelf processor with ten UARTs. However, in an FPGA, this configuration is very easily accomplished.

2.2.2 Obsolescence mitigation

Some companies, in particular those supporting military contracts, have a design requirement to ensure a product lifespan that is much longer than the lifespan of a standard electronics product. Component obsolescence mitigation is a difficult issue. FPGA soft-processors are an excellent solution in this case since the source HDL for the soft-processor can be purchased. Ownership of the processor's HDL code may fulfill the requirement for product lifespan guarantee.

2.2.3 Component and cost reduction

With the versatility of the FPGA, previous systems that required multiple components can be replaced with a single FPGA. Certainly this is the case when an auxiliary I/O chip or a co-processor is required next to an off-the-shelf processor. By reducing the component count in a design, a company can reduce board size and inventory management, both of which will save design time and cost.

2.2.4 Hardware acceleration

Perhaps the most compelling reason to choose an FPGA embedded processor is the ability to make tradeoffs between hardware and software to maximize efficiency and performance. If an algorithm is identified as a software bottleneck, a custom co-processing engine can be designed in the FPGA specifically for that algorithm. This co-processor can be attached to the FPGA embedded processor through special, low-latency channels, and custom instructions can be defined to exercise the co-processor. With modern FPGA hardware design tools, transitioning software bottlenecks from software to hardware is much easier since the software C code can be readily adapted into hardware with only minor changes to the C code.[1]

2.3 Disadvantages

The FPGA embedded processor system is not without disadvantages. Unlike an off-the-shelf processor, the hardware platform for the FPGA embedded processor must be designed. The embedded designer becomes the hardware processor system designer when an FPGA solution is selected.

Because of the integration of the hardware and software platform design, the design tools are more complex. The increased tool complexity and design methodology requires more attention from the embedded designer.

Since FPGA embedded processor software design is relatively new compared to software design for standard processors, the software design tools are likewise relatively immature, although workable. Significant progress in this area has been made by both Altera and Xilinx. Within the next year, this disadvantage should be further diminished, if not eliminated.

Device cost is another aspect to consider. If a standard, off-the-shelf processor can do the job, that processor will be less expensive in a head-to-head comparison with the FPGA capable of an equivalent processor design. However, if a large FPGA is already in the system, consuming unused gates or a hard processor in the FPGA essentially makes the embedded processor system cost inconsequential.

2.4 Peripherals and memory controllers

To facilitate FPGA embedded processor design, both Xilinx and Altera offer extensive libraries of intellectual property (IP) in the form of peripherals and memory controllers. This IP is included in the embedded processor toolsets provided by these manufacturers. To emphasize the versatility and flexibility afforded the embedded designer using an FPGA, a partial list of IP included with the embedded processor design tools from Altera and Xilinx is listed below.

2.4.1 Peripherals & peripheral controllers[2]

- General purpose I/O
- UART
- Timer
- Debug
- SPI
- DMA Controller
- Ethernet (interface to external MAC/PHY chip)

2.4.2 Memory controllers[2]

- SRAM
- Flash
- SDRAM
- DDR SDRAM (Xilinx only)
- CompactFlash

2.5 Manufacturers' benchmarks

The industry standard benchmark for FPGA embedded processors is Dhrystone MIPs (DMIPs). Both Altera and Xilinx quote DMIPs for most, if not all, of the available embedded processors.

The achieved DMIPs reported by the manufacturers are based on several things that maximize the benchmark results. Some of these factors include the following:

- Optimal compiler optimization level
- Fastest available device family (unless otherwise noted)
- Fastest speed grade in that device family
- Executing from fastest, lowest latency memory, typically on-chip
- Optimization of processor's parameterizable features

The available embedded processors with the manufacturers' quoted maximum frequency and DMIPs are summarized here.

2.5.1 Altera[3]

Table 1 – Altera Embedded Processors and Performance

Processor	Processor Type	Device Family Used	Speed (MHz) Achieved	DMIPs Achieved
ARM922T™	hard	Excalibur	200	210
NIOS®	soft	Stratix-II	180	Not Reported
Nios® II	soft	Stratix-II	Not Reported	200
Nios® II	soft	Cyclone-II	Not Reported	100

2.5.2 Xilinx[4]

Table 2 – Xilinx Embedded processors and Performance

Processor	Processor Type	Device Family Used	Speed (MHz) Achieved	DMIPs Achieved
PowerPC™ 405	hard	Virtex-4	450	680
MicroBlaze	soft	Virtex-II Pro	150	123
MicroBlaze	soft	Spartan-3	85	65

3. Performance Enhancing Techniques

Some embedded designers get frustrated when the performance of their selected FPGA embedded processor is half or less of what was expected. In some cases, a designer is unable to duplicate the manufacturers' published results for a benchmark.

The reason that achieved performance is lacking may be caused by the designer not enacting all of the performance enhancing techniques available to FPGA embedded processors. The manufacturers obviously know what must be done to get the most out of their chips, and they take full advantage of every possible

enhancement when benchmarking. Embedded designers, who are familiar with standard microprocessor performance optimization, need to learn which software optimization techniques apply to FPGA embedded processors. Designers must also learn performance-enhancing techniques that apply specifically to FPGAs.

The design landscape is certainly more complicated with an FPGA embedded processor. The incredible advantages gained with this type of design are not without tradeoffs. Specifically, the increased design complexity is overwhelming to many, including experienced embedded or FPGA designers. The manufacturers and their partners put significant effort into training and providing support to designers experimenting with this technology. Taking advantage of a local field applications engineer is essential to FPGA embedded processor design success.

As an introduction to this type of design, a few performance-enhancing techniques are highlighted. Specific references below are based on research of Xilinx FPGAs and tools, although Altera is likely to have similar features.

3.1 Optimization techniques that are not FPGA specific

An embedded designer is familiar with many of the techniques discussed in this section. For that reason, significant detail is not given here. The main objective of this section is to emphasize that many standard microprocessor design optimization techniques apply to FPGA embedded processor design and can have excellent benefits.

3.1.1 Code manipulation

Many optimizations are available to affect the application code. Some techniques apply to how the code is written. Other techniques affect how the compiler handles the code.

3.1.1.1 Optimization level

Compiler optimizations are available in Xilinx Platform Studio (XPS) based on GCC. The current version of the MicroBlaze and PowerPC GCC-based compilers in EDK 6.3 is 2.95.3-4. These compilers have several levels of optimization, including: Levels 0, 1, 2, and 3 and also a size reduction optimization. An explanation of the strategy for the different optimization levels briefly follows[5]:

Level 0: No optimization

Level 1: First level optimization. Performs jump and pop optimizations.

Level 2: Second level optimization. This level activates nearly all optimizations that do not involve a speed-space tradeoff, so the executable should not increase in size. The compiler does not perform loop unrolling, function in-lining or strict aliasing optimizations. This is the standard optimization level used for program deployment.

Level 3: Highest optimization level. This level adds more expensive options, including those that **increase** code size. In some cases this optimization level actually produces code that is less efficient than the O2 level, and as such should be used with caution.

Size: Size optimization. The objective is to produce the smallest possible code size.

3.1.1.2 Use of manufacturer's optimized instructions

Xilinx provides several customized instructions that have been streamlined for Xilinx embedded processors. One example of this is **`xil_printf`**. The function is nearly identical to **`printf`** with the following exceptions: support for type real numbers is removed; not reentrant; and no longlong (64-bit). For these differences, the **`xil_printf`** function is 2953 bytes, making it much smaller than **`printf`**, which is 51788 bytes large.[6]

3.1.1.3 Assembly

Assembly, including in-line assembly, is supported by GCC. As with any microprocessor, assembly becomes very useful in fully optimizing time critical functions. Be aware, however, that some compilers will not optimize the remaining C code in a file if in-line assembly is also used in that file. Also, assembly code does not enjoy the code portability advantages of C.

3.1.1.4 Miscellaneous

Many other code-related optimizations can and should be considered when optimizing an FPGA embedded processor, including:

- locality of reference
- code profiling
- careful definition of variables (Xilinx provides a Basic Types definition)
- strategic use of small data sections, with accesses that can be twice as fast as large data sections
- judicious use of function calls to minimize pushing/popping of stack frames
- loop length (especially where cache is involved)

3.1.2 Memory Usage

Many processors provide access to fast, local memory, as well as an interface to slower, secondary memory. The same is true with FPGA embedded processors. The way this memory is used has a significant affect on performance. Like other processors, the memory usage in an FPGA embedded processor can be manipulated with a linker script.

3.1.2.1 Local memory only

The fastest possible memory option is to put everything in local memory. Xilinx local memory is made up of large FPGA memory blocks called BlockRAM (BRAM). Embedded processor accesses to BRAM happen in a single bus cycle. Since the processor and bus run at the same frequency in MicroBlaze, instructions stored in BRAM are executed at the full MicroBlaze processor frequency. In a MicroBlaze system, BRAM is essentially equivalent in performance to a Level 1 (L1) cache. The PowerPC can run at frequencies greater than the bus and has true, built-in L1 cache. Therefore, BRAM in a PowerPC system is equivalent in performance to a Level 2 (L2) cache.

Xilinx FPGA BRAM quantities differ by device. For example, the 1.5 million gate Spartan-3 device (XC3S1500) has a total capacity of 64KB, whereas the 400,000 gate Spartan-3 device (XC3S400) has half as much at 32KB. An embedded designer using FPGAs should refer to the device family datasheet to review a specific chip's BRAM capacity.

If the designer's program fits entirely within local memory, then the designer achieves optimal memory performance. However, many embedded programs exceed this capacity.

3.1.2.2 External memory only

Xilinx provides several memory controllers that interface with a variety of external memory devices. These memory controllers are connected to the processor's peripheral bus. The three types of volatile memory supported by Xilinx are SRAM, single-data-rate SDRAM, and double-data-rate (DDR) SDRAM. The SRAM controller is the smallest and simplest inside the FPGA, but SRAM is the most expensive of the three memory types. The DDR controller is the largest and most complex inside the FPGA, but fewer FPGA pins are required, and DDR is the least expensive per megabyte.

In addition to the memory access time, the peripheral bus also incurs some latency. In MicroBlaze, the memory controllers are attached to the On-chip Peripheral Bus (OPB). For example, the OPB SDRAM controller requires a four to six cycle latency for a write and eight to ten cycle latency for a read (depending

on bus clock frequency)[7]. The worst possible program performance is achieved by having the entire program reside in external memory. Since optimizing execution speed is a typical goal, an entire program should rarely, if ever, be targeted solely at external memory.

3.1.2.3 Cache external memory

The PowerPC in Xilinx FPGAs has instruction and data cache built into the silicon of the hard processor. Enabling the cache is almost always a performance advantage for the PowerPC.

The MicroBlaze cache architecture is different than the PowerPC because the cache memory is not dedicated silicon. The instruction and data cache controllers are selectable parameters in the MicroBlaze configuration. When these controllers are included, the cache memory is built from BRAM. Therefore, enabling the cache consumes BRAM that could have otherwise been used for local memory. Compared to local memory, cache consumes more BRAM than local memory for the same storage size because the cache architecture requires address line tag storage. Additionally, enabling the cache consumes general-purpose logic to build the cache controllers.

For example, an experiment in Spartan-3 enables 8 KB of data cache and designates 32 MB of external memory to be cached. This cache requires 12 address tag bits. This configuration consumes 124 logic cells and 6 BRAMs. Only 4 BlockRAMs are required in Spartan-3 to achieve 8 KB of local memory. In this case, cache is 50% more expensive in terms of BRAM usage than local memory. The 2 extra BRAMs are used to store address tag bits.

If 1 MB of external memory is cached with an 8 KB cache, then the address tag bits can be reduced to 7. This configuration then only requires 5 BRAMs rather than 6 (4 BRAMs for the cache, and 1 BRAM for the tags). This is still 25% greater than if the BRAMs are used as local memory.

Additionally, the achievable system frequency may be reduced when the cache is enabled. In one example, the system without any cache is capable of running at 75 MHz; the system with cache is only capable of running at 60 MHz. Enabling the cache controller adds logic and complexity to the design, decreasing the achieved system frequency during FPGA place and route. Therefore, in addition to consuming FPGA BRAM resources that may have otherwise been used to increase local memory, the cache implementation may also cause the overall system frequency to decrease.

Considering these cautions, enabling the MicroBlaze cache, especially the instruction cache, may improve performance, even when the system must run at a lower frequency. As will be shown later in the DMIPs section (see Section 4.1), a 60 MHz system with instruction cache enabled has a 150% advantage over a 75 MHz system without instruction cache (both systems store entire program in external memory). When both instruction and data caches are enabled, the 60 MHz outperforms the 75 MHz system by 308%.

This example is not the most practical since the entire DMIPs program will fit in the cache. A more realistic experiment is to use an application that is larger than the cache. Another precaution is regarding applications that frequently jump beyond the size of the cache. Multiple cache misses degrade the performance, sometimes making a cached external memory worse than the external memory without cache.

Given these warnings, enabling the cache is always worth an experiment to determine if it improves the performance for an application.

3.1.2.4 Combination: code partitioning in internal, external, and cached memory

The memory architecture that provides the best performance is one that only has local memory. However, this architecture is not always practical since many useful programs will exceed the available capacity of the local memory. On the other hand, running from external memory exclusively may have more than an eight times performance disadvantage due to the peripheral bus latency. Caching the external memory is an

excellent choice for PowerPC. Caching the external memory in MicroBlaze definitely improves results, but an alternative method is presented that may provide optimal results.

For MicroBlaze, perhaps the optimal memory configuration is to wisely partition the program code, maximizing the system frequency and local memory size. Critical data, instructions, and stack are placed in local memory. Data cache is not used, allowing for a larger local memory bank. If the local memory is not large enough to contain all instructions, the designer should consider enabling the instruction cache for the address range in external memory used for instructions.

By not consuming BRAM in data cache, the local memory can be increased to contain more space. An instruction cache for the instructions assigned to external memory can be very effective. Experimentation or profiling shows which code items are most heavily accessed; assigning these items to local memory provides a greater performance improvement than caching.

Express Logic's Thread-Metric™ test suite is an example of how partitioning a small piece of code in local memory can have a significant performance improvement (see Section 4.2). One function in the Thread-Metric Basic Processing test is identified as time critical. The function's data section (consisting of 19% of the total code size) is allocated to local memory, and the instruction cache is enabled. The 60 MHz, cached and partitioned-program system achieves performance that is 560% better than running a non-cached, non-partitioned 75 MHz system using only external memory.

However, the 75 MHz system shows even more improvement with code partitioning. If the time critical function's data and text sections (22% of the total code size) are assigned to local memory on the 75 MHz system, a 710% improvement is realized, even with no instruction cache for the remainder of the code assigned to external memory.

In this one case, the optimal memory configuration is one that maximizes local memory and system frequency without cache. In other systems where the critical code is not so easily pinpointed, a cached system may perform better. Designers should experiment with both methods to determine what is optimal for their design.

3.2 FPGA specific optimization techniques

Since the designer is actually building and creating the embedded processor system hardware in an FPGA, much can be done to improve the performance of the hardware. Additionally, with an FPGA embedded processor residing next to additional FPGA hardware resources, a designer can consider custom co-processor designs specifically targeted at a design's core algorithm.

3.2.1 Increase FPGA's operating frequency

Employing FPGA design techniques to increase the operating frequency of the FPGA embedded processor system increases performance. Several methods are considered.

3.2.1.1 Logic optimization and reduction

Only connect those peripherals and busses that will be used. A few examples are presented below.

If a design does not store and run any instructions using external memory, do not connect the instruction-side of the peripheral bus. Connecting both the instruction and data side of the processor to a single bus creates a multi-master system, which requires an arbiter. Optimal bus performance is achieved when a single master resides on the bus.

Debug logic requires resources in the FPGA and may be the hardware bottleneck. When a design is completely debugged, the debug logic can be removed from the production system, potentially increasing the system's performance. For example, removing a MicroBlaze Debug Module (MDM) with an FSL

acceleration channel saves 950 LCs. In MicroBlaze systems with the cache enabled, the debug logic will typically be the critical path that slows down the entire design.

The Xilinx OPB External Memory Controller (EMC) used to connect SRAM and Flash memories creates a 32-bit address bus even if 32 bits are not required to address the memory. Xilinx also provides a bus-trimming peripheral, which removes the unused address bits. When using this memory controller, the bus-trimmer should always be used to eliminate the unused addresses. This frees up routing and pins that would have otherwise been used. The Xilinx Base System Builder (BSB) now does this automatically.

Xilinx provides several GPIO peripherals. The latest GPIO peripheral version (v3.01.a) has excellent capabilities, including dual-channel support, bi-directionality, and interrupt capability. However, these features also require more resources, which affect timing. If a simple GPIO is all that the design requires, the designer should use a more primitive version of the GPIO, or at least ensure that the unused features in the enhanced GPIO are turned off. In the optimized examples in this study, GPIO v1.00.a is used, which is much less sophisticated, much faster, and approximately half the size (304 LCs for 7 GPIO v1.00.a peripherals as compared to 602 LCs for v3.01.a).

3.2.1.2 Area and timing constraints

Xilinx FPGA place and route tools perform much better when given guidelines as to what is most important to the designer. In the Xilinx tools, a designer can specify the desired clock frequency, pin location, and logic element location. By providing these details, the tools are able to make smarter trade-offs during the hardware design implementation.

Some peripherals require additional constraints to ensure proper operation. For example, both the DDR SDRAM controller and the 10/100 Ethernet MAC require additional constraints to guarantee that the tools create correct and optimized logic. The designer must read the datasheet for each peripheral and follow the recommended design guidelines.

3.2.2 Hardware acceleration

Dedicated hardware outperforms software. The embedded designer who is serious about increasing performance must consider the FPGA's ability to accelerate the processor performance with dedicated hardware. Although this technique consumes FPGA resources, the performance improvements can be extraordinary.

3.2.2.1 Turn on the hardware divider and barrel-shifter

MicroBlaze can be customized to use a hardware divider and a hardware barrel-shifter rather than performing these functions in software. Enabling these processor capabilities consumes more logic but improves performance. In one example, enabling the hardware divider and barrel-shifter adds 414 LCs, but the performance is improved by 18.1% (DMIPs benchmark).

3.2.2.2 Software bottlenecks converted to co-processing hardware

Custom hardware logic can be designed to offload an FPGA embedded processor. When a software bottleneck is identified, a designer can choose to convert the bottleneck algorithm into custom hardware. Custom software instructions can then be defined to operate the hardware co-processor.

Both MicroBlaze and Virtex-4 PowerPC include very low-latency access points into the processor, which are ideal for connecting custom co-processing hardware. Virtex-4 introduces the Auxiliary Processing Unit (APU) for the PowerPC[8]. The APU provides a direct connection from the PowerPC to co-processing hardware. In MicroBlaze, the low-latency interface is called the Fast Simplex Link (FSL) bus. The FSL bus contains multiple channels of dedicated, uni-directional, 32-bit interfaces. Because the FSL channels are dedicated, no arbitration or bus mastering is required. This allows an extremely fast interface to the processor[9].

Converting a software bottleneck into hardware may seem like a very difficult task. Traditionally, a software designer identifies the bottleneck, after which the algorithm is transitioned to an FPGA designer who writes VHDL or Verilog code to create the hardware co-processor. Fortunately, this process has been greatly simplified by tools that are capable of generating FPGA hardware from C code. One such tool is CoDeveloper™ from Impulse Accelerated Technologies. This tool allows one designer who is familiar with C to port a software bottleneck into a custom piece of co-processing FPGA hardware using CoDeveloper's Impulse C libraries.

Some examples of algorithms that could be targeted for hardware-based co-processors include:

- Inverse Discrete Cosine Transformation, used in JPEG 2000
- Fast Fourier Transform
- MP3 decode
- Triple-DES and AES encryption
- Matrix-manipulation

Any operation that is algorithmic, mathematical, or parallel is a good candidate for a hardware co-processor[10]. FPGA logic consumption is traded for performance. The advantages can be enormous, improving performance by tens or hundreds of times.

4. MicroBlaze True System Performance Revealed

To illustrate the difference these performance-enhancing techniques can have on real designs, several case studies are now presented. Each of these examples are built and run using Xilinx ISE and EDK software, version 6.3. Where possible, the examples are executed on real hardware, using Memec's Spartan-3 MB Development Board, with a 1.5 million gate Spartan-3 FPGA (XC3S1500).

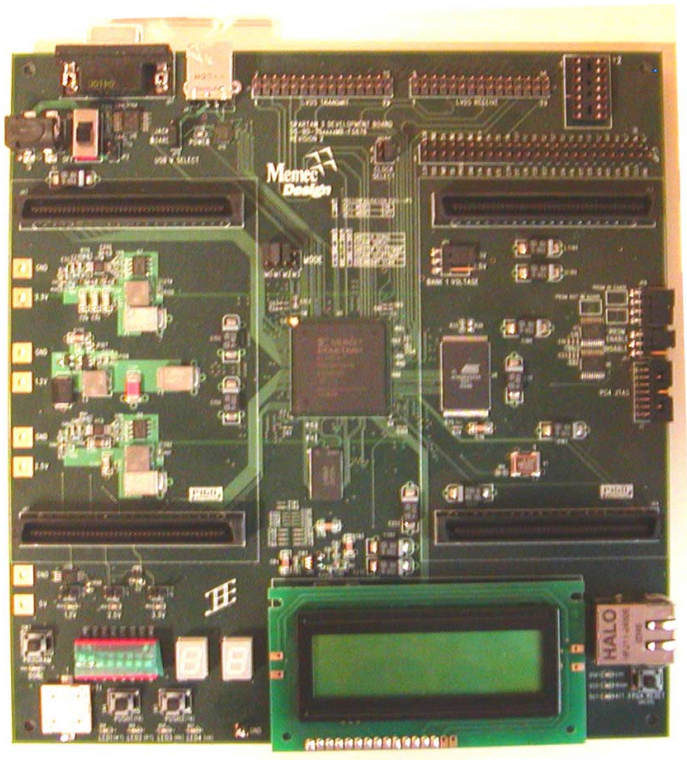

Figure 2 – Memec Spartan-3 MB Development Board

The value of any benchmark can be questioned. Therefore, to provide a more objective perspective, several applications are examined: Dhrystone MIPs, Express Logic's Thread-Metric RTOS benchmark, and a real-world application (Triple-DES encryption).

4.1 Dhrystone MIPs

As previously specified, Xilinx publishes a Spartan-3 MicroBlaze benchmark of 65 DMIPs running at 85 MHz. This case study investigates the system design for which this benchmark was achieved and compares it to other, more real-world processor systems.

4.1.1 Best performing, but minimal system

The best performing system is achieved under the following conditions:

- minimal peripheral set, including only those peripherals required to run the benchmark or report results
- highly optimized hardware, including several FPGA design optimization methods as well as the fastest speed grade for this family (-5)
- local memory only
- optimal compiler optimization

The IP included in this embedded processor design are listed below:

- MicroBlaze
  - hardware divider and barrel shifter included
  - no data or instruction cache
- Data-side peripheral bus (no instruction-side)
- 8KB instruction local memory
- 16KB data local memory
- No debug hardware
- RS232 UART
- Timer

A block diagram of the system is shown in Figure 3.

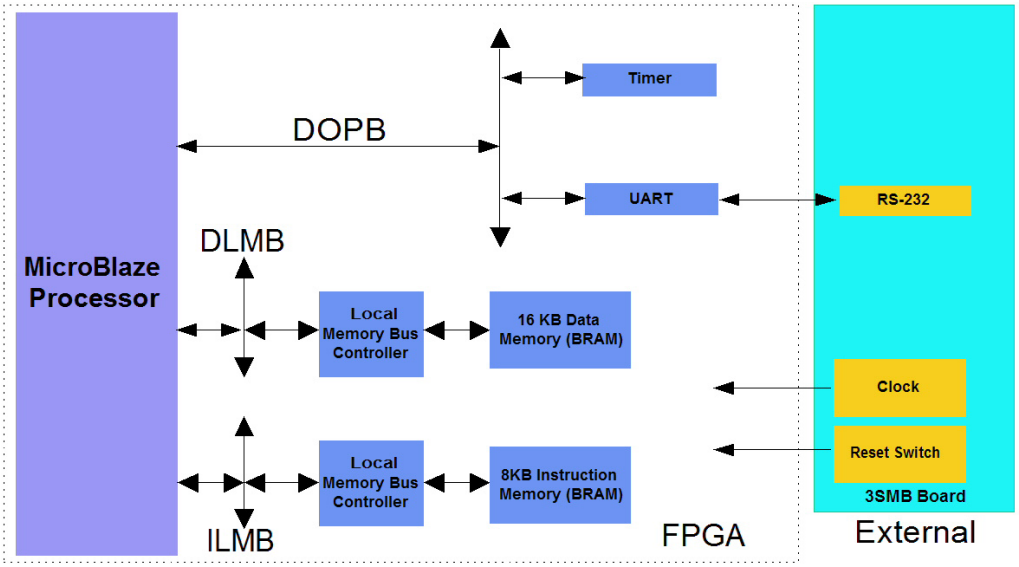

Figure 3 – Spartan-3 DMIPs System Block Diagram

Based on this processor system running in the slower, -4 speed grade on the Memec Spartan-3 MB board, the achieved results are 63.3 DMIPs. The MicroBlaze system runs at a maximum frequency of 81.8 MHz. Dividing the DMIPs by the operating frequency gives 0.773 DMIPs/MHz, which is useful when extrapolating performance numbers for a similar processor system running at a different frequency. This system consumes 2322 logic cells. Based on Xilinx press release numbers of $20 for an XC3S1500[11], this system cost can be estimated at $1.74.

When this same system is built using a -5 speed grade Spartan-3 device, the achieved frequency is 90.8 MHz. Since a development board with a -5 device is not available, the DMIPs performance is extrapolated based on the numbers achieved during the -4 experiment. Based on 0.773 DMIPs/MHz, this -5 system is capable of achieving 70.3 DMIPs. This result is better than the published Xilinx benchmark of 65 DMIPs!

4.1.2 Real-world, but unoptimized system

A more realistic system is now used to run the same benchmark. Although the peripherals aren't needed to run the benchmark, many useful peripherals are added to the system. Xilinx Platform Studio's Base System Builder (BSB) is used to generate the hardware platform. No hardware optimization is attempted. The entire program, including code and data, are stored in and run from external memory. The default optimization level (Level 2) is used. By default, the hardware divider and barrel-shifter are not enabled.

The system consists of the following:

- MicroBlaze
  - hardware divider and barrel shifter not included
  - no data or instruction cache
- Data- and instruction-side peripheral bus with arbiter
- 32KB shared instruction and data local memory
- Debug hardware with FSL acceleration channel
- RS232 UART
- USB-to-serial bridge UART
- GPIO
  - 4 LEDs
  - 2 seven-segment displays
  - 2 push buttons
  - 8 dip switches
  - Flash chip reset
  - Flash chip RDY/BUSYn
- Flash controller
- SRAM controller
- 10/100 Ethernet MAC
- Timer
- Interrupt controller

Figure 4 shows the block diagram for this system.

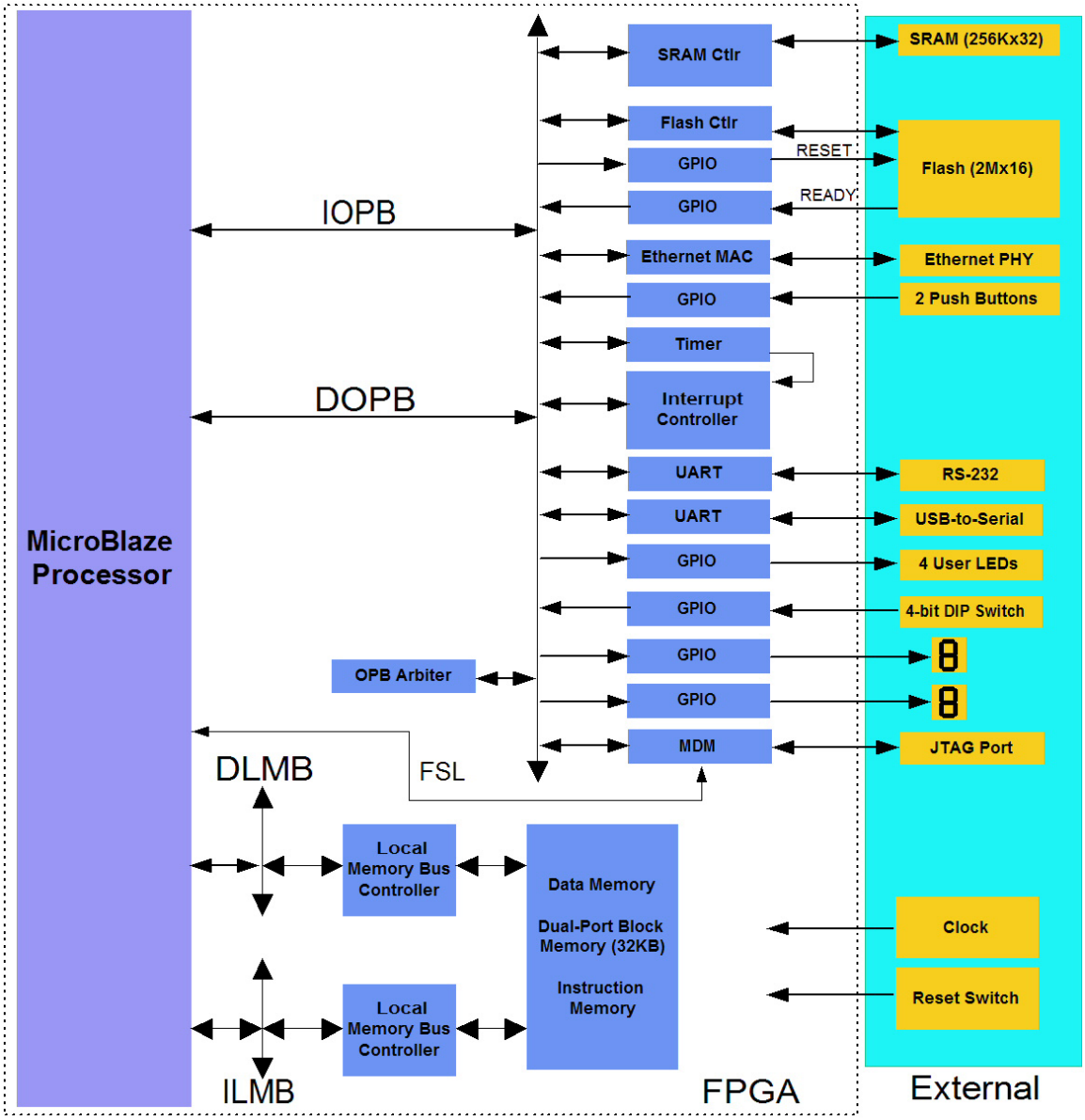

Figure 4 – Spartan-3 Real-world System Block Diagram

For those who are inexperienced with FPGA embedded processors and Xilinx MicroBlaze specifically, the default BSB system may very well be the typical system that a designer creates. The design will build and run as expected, but the performance degradation is severe. The operating frequency achieved is only 62.5 MHz. The DMIPs result is 4.505 DMIPs, which is more than 14 times less than the Xilinx published DMIPs number. This design consumes 7732 logic cells, which, based on the previous chip price quoted, equates to a system cost of $5.81.

4.1.3 Real-world, optimized system

Next, the "typical" system from the previous experiment is optimized in both hardware and software. The hardware divider and barrel-shifter are enabled. The GPIO units are replaced with a more primitive, but smaller and faster version. Using other FPGA design techniques, the hardware is optimized and rebuilt,

with results achieving 75 MHz for the non-cached MicroBlaze system and 60 MHz for the cached MicroBlaze system.

Although the code is small enough to entirely fit inside local memory, several alternative memory structures are investigated. The following parameters are adjusted to determine affects on performance:

- Location of instructions
- Location of small data
- Location of large data
- Location of stack
- Data cache
- Instruction cache
- Compiler optimization level

Optimizing the hardware system to run at 75 MHz with a hardware divider and barrel shifter increases LC usage by 986 at a cost of $0.74. This increases the performance to 6.846 DMIPs, which is a 52% improvement compared to the unoptimized system. Changing the Compiler Optimization Level to Level 3 further increases the performance to 7.139, an additional increase of 4.3%.

Enabling the instruction and data caches adds an additional $0.27 of LC cost but increases the performance to 29.130 DMIPs.

Eliminating the data cache saves $0.20 and allows the local memory to be increased by 16 KB. With the instruction cache enabled and assigning the stack and small data sections to local memory, 33.178 DMIPs are achieved. If all data is assigned to local memory, this performance increases to 47.811 DMIPs.

If both instruction and data cache are eliminated, and local memory is used for stack, small data sections, and instructions, the results are 41.483 DMIPs. When local memory is used for the entire program, the maximum performance for this hardware system is achieved at 59.785 DMIPs.

For this real-world system, the best results are within 8% of the manufacturer's published benchmark. Even when running cached external instruction memory with local memory data storage, the system is capable of running at 74% of the manufacturer's benchmark.

These results are summarized in Table 3.

Table 3 – DMIPs Experiment Results

	Frequency	DMIPs	LCs
Initial optimized system, running from SRAM	75 MHz	6.846	8718
Increase compiler optimization level to Level 3	75 MHz	7.139	8718
Added instruction and data caches	60 MHz	29.130	9076
Removed data cache, moved stack and small data sections to local memory	60 MHz	33.178	8812
Adding remaining data sections to local memory	60 MHz	47.811	8812
Removed instruction cache, moved large data sections back to SRAM, moved instructions to local memory	75 MHz	41.483	8718
Moved entire program to local memory	75 MHz	59.785	8718

4.2 Thread-Metric Real-Time Operating System (RTOS) benchmark

The performance differences between different optimized and unoptimized MicroBlaze systems are examined using an RTOS benchmark. The Thread-Metric test suite consists of eight distinct RTOS tests that are designed to highlight commonly used aspects of an RTOS. The tests measure the total number of RTOS events that can be processed during a specific timer interval[12]. For the purpose of examining the performance differences between systems, only the Basic Processing test is reported in these results. This test consists of a single thread. A 30 second time interval is used for these experiments. Express Logic's ThreadX® RTOS is used for this comparison.

4.2.1 Real-world, unoptimized

The unoptimized system used in Section 4.1.2 is reused for this benchmark. With the program running from SRAM, the results are 19713 events

4.2.2 Real-world, optimized

The hardware and software optimized system used in Section 4.1.3 is now used with Thread-Metric. Through experimentation, source file tm_basic_processing_test.c is identified as the critical function.

With the program running from SRAM, the results are 23823 events, an improvement of 21%, attributable solely to the increase in operating frequency from 62.5 MHz to 75 MHz. Next, the stack and all data sections are moved to local memory. The achieved results are 29808 events, an additional performance improvement of 25%.

With the entire program in SRAM and instruction cache enabled (system running at 60 MHz), the results are 64006 events. This is a significant improvement of 225% over the original, unoptimized results. When the critical function's data section is moved to local memory with instruction cache enabled, the performance is 157512 events.

By removing the instruction cache, the system speed is boosted to 75 MHz. The critical function's data and text sections are roughly one-fifth the total program size. When the critical function's data and text are assigned to local memory, the achieved results are 192867 events. If the entire program runs from local memory, the performance realized is maximized at 198787 events.

For this one example, only 3% performance is lost in the program-partitioned experiment running at 75 MHz. When a critical function can be identified and assigned to permanently reside in local memory, caching is not necessary, and the achieved results are nearly as good as if the entire program was running from local memory. These results are summarized in Table 4.

Table 4 – Thread-Metric Basic Test Results

	Frequency	Thread-Metric Basic Events
Initial optimized system, running from SRAM	75 MHz	23823
Stack and all data moved to local memory	75 MHz	29808
Running from SRAM, I-cache enabled	60 MHz	64006
Critical function's data moved to local memory	60 MHz	157512
Removed instruction cache, critical function's instructions also moved to local memory	75 MHz	192867
Moved entire program to local memory	75 MHz	198787

4.3 User application

A Triple-DES encryption algorithm is used to show the power of using a hardware co-processor. First, the algorithm runs in a MicroBlaze system using software only. Next, a piece of the algorithm is converted using Impulse C to a hardware co-processor connected through the MicroBlaze's FSL bus. The hardware used is a 1 million gate Virtex-II FPGA on the Memec Virtex-II MB Development Board.

4.3.1 Software-only implementation

As reported in *FPGA Journal*[13], 1000 text blocks were encrypted with Triple-DES. In the software only approach, this encryption took 252 ms running on a 100 MHz MicroBlaze system.

4.3.2 Hardware co-processor

Impulse C was used to create a hardware co-processor to assist the MicroBlaze processor in performing the Triple-DES encryption. The code was slightly modified to support hardware compilation using Impulse C. The Impulse C compiler then created a hardware representation in the Virtex-II FPGA for the Triple-DES algorithm. This newly built co-processor must be able to communicate with the MicroBlaze. Two FSL channels are used to transfer data between the MicroBlaze and the Triple-DES co-processor.

A custom instruction is defined to exercise the FSL channels. The MicroBlaze application is modified to use this newly defined instruction, which takes advantage of the co-processor. The hardware is now rebuilt. The complexity of the co-processor results in a lower system frequency of 24 MHz. However, the overall performance gain of the system is remarkable with a single encryption cycle taking only 6.9 ms. This is a 36 times increase in performance!

Upon further review, more sophisticated changes are made to the Impulse C version of the Triple-DES algorithm. Taking advantage of C programming techniques such as array splitting, loop unrolling, and pipelining, further optimization can be accomplished. Implementing the maximum optimization in the Impulse C code, the final system performance is 0.59 ms, an incredible boost of 425 times over the original software-only implementation![14]

Although more hardware is consumed in the FPGA, the performance enhancements are significant. The ability to create custom co-processing units specific to a designer's application makes the FPGA embedded processor solution unmatched in performance compared to any off-the-shelf processor!

5. Conclusion

Based on the experiments performed in this research, several conclusions are explained.

5.1 Reasonable expectations

All manufacturers want the best possible benchmark to publish. FPGA manufacturers take full advantage of system flexibility and FPGA design techniques to achieve high benchmarks. The embedded designer must understand how these benchmarks are achieved and realize that an actual FPGA embedded processor system will not be able to achieve such high marks. The benchmarks are still useful as a means of comparing one manufacturer with another.

Regarding the design process, remember that the hardware platform is part of the FPGA embedded processor design. Unlike off-the-shelf processors where the hardware is pre-defined and fixed, FPGAs have the flexibility and added complexity to create a multitude of different systems. A member of the design team must be capable of hardware development and optimization of the FPGA embedded processor.

If an application does not require high performance or any of the other advantages that an FPGA can provide, an off-the-shelf processor is most likely a less complicated, less expensive, and better solution.

5.2 Optimization through experimentation yields the best results

Much can be done to optimize an FPGA embedded processor system. In addition to standard software optimization, the embedded designer can perform many hardware optimizations. Careful use of memory has a large impact on the performance. Experimentation with different memory strategies is necessary to achieve the best performance.

Understand that the addition or removal of each peripheral, peripheral controller, or bus alters the design size, cost, and speed. Use only what is necessary and no more!

5.3 Take advantage of superior flexibility in FPGAs

An FPGA embedded processor has the power and ability to provide previously unachievable flexibility and performance. With an FPGA, a designer can specify exactly the peripherals required in a system. With an FPGA soft processor, a designer can purchase and own the source code for the processor. With an FPGA, a designer can adapt C code from a software bottleneck to create a custom hardware co-processing unit. These are excellent advantages that can only be realized in programmable hardware.

If the designer chooses not to take advantage of these capabilities, the realized FPGA embedded processor system performance may be a huge disappointment. However, for those who take full advantage of the FPGA embedded processor, specific application performance greatly exceeding typical microprocessor expectations is possible!

6. Acknowledgements

Shalin Sheth, Xilinx
Ron Wright, Memec

7. References

[1] www.impulsec.com/trueansic.htm

[2] For a complete list of available IP, visit the following websites:
www.altera.com/products/ip/processors/nios2/features/ni2-peripherals.html
www.xilinx.com/ise/embedded/edk_ip.htm

[3] The website URLs where these numbers were found, as of November 10, 2004, are listed below:
www.altera.com/products/devices/excalibur/exc-index.html
www.altera.com/products/ip/processors/nios/overview/nio-overview.html
www.altera.com/products/ip/processors/nios2/overview/ni2-overview.html

[4] The website URLs where these numbers were found, as of November 10, 2004, are listed below:
www.xilinx.com/products/virtex4/capabilities.htm
www.xilinx.com/ipcenter/processor_central/embedded/performance.htm
www.xilinx.com/ipcenter/processor_central/microblaze/performance.htm

[5] *An Introduction to GCC for the GNU Compilers gcc and g++*, Brian Gough, Network Theory Ltd., March 31, 2004

[6] See Xilinx Answer Record 19592 at www.support.xilinx.com

[7] *OPB Synchronous DRAM (SDRAM) Controller*, DS426, Xilinx, August 11, 2004

[8] PowerPC™ 405 Processor Block Reference Guide, UG018, Xilinx, August 20, 2004

[9] *Fast Simplex Link (FSL) Bus (v2.00a)*, DS449, Xilinx, August 17, 2004

[10] www.impulsec.com/applications.htm

[11] Volume pricing for 250k units, end of 2004 (www.xilinx.com/prs_rls/silicon_spart/0498s3.htm)

[12] *Thread-Metric RTOS Test Suite*, Express Logic, INC., April 21, 2004

[13] "FPGAs Provide Acceleration for Software Algorithms," *FPGA Journal*, David Pellerin and Milan Saini, 2004

[14] *Optimizing Impulse C Code for Performance*, Impulse Accelerated Technologies, Inc., Scott Thibault and David Pellerin, September 30, 2004