Achieve Performance Increases and Product Differentiation with OPB Mastering

A simple OPB bus mastering IP core, in addition to the MicroBlaze processor, can allow for boosts in performance.

by Steven M. Spano
President
Finger Lakes Engineering
steve@flconsult.com

The Xilinx® MicroBlaze™ processor, part of the Embedded Development Kit (EDK) software suite, allows you to instantiate a powerful 32-bit processor derived from the IBM PowerPC™ architecture. Supporting core clock speeds as fast as 200 MHz, the MicroBlaze embedded processor has the flexibility to be as simple as a 1-bit output port that flashes an LED or as complex as µClinux-based routers with touch-screen LCDs and multiple high-speed ADC/DAC ports.

Customizing the MicroBlaze processor is typically achieved by adding IP cores into the architecture through the Xilinx EDK tool. Xilinx provides a series of IP cores targeted for the MicroBlaze embedded processor within the EDK package; some of these cores include UARTs, I2C, Ethernet MACs, and GPIO cores.

The EDK-provided IP cores offer you a starting point with which to create a configurable FPGA-based processor capable of interfacing with common external peripherals, much the same as any number of silicon-based 32-bit processors.

The key advantage in using the MicroBlaze core as an FPGA-based processor lies in the ability to add user-specific IP cores into the architecture. The addition of user-specific IP can provide performance gains and product differentiation by adding a feature that the competition may not have in their silicon processors. Through a proper understanding of the software executing on the MicroBlaze processor and the capabilities of its hardware architecture, it is possible to create simple IP cores that can offer tremendous boosts in overall performance.

In this article, I'll present an overview of the MicroBlaze system architecture, the on-chip peripheral bus (OPB), and its associated master and slave core functions. I'll also review a case study detailing how Finger Lakes Engineering utilized key MicroBlaze architecture elements to accelerate a raster-based thermal receipt printer.

Overview of the MicroBlaze Architecture

The MicroBlaze embedded processor is a high-performance 32-bit core included in the Xilinx EDK development suite (Figure 1). The core is synthesized from Xilinx-provided HDL source code and placed within a target FPGA device. The MicroBlaze core, by itself, is simply an execution unit that reads, processes, and writes data based on firmware instructions.

A typical MicroBlaze configuration might comprise the MicroBlaze core unit, a block RAM element to store program code for execution, a UART, an external memory controller (for large SRAM/SDRAM), and a GPIO port (Figure 2). This architecture could easily be synthesized and placed within a Spartan™-3E XC3S500E-PQ208C FPGA and may only occupy approximately 40% of the FPGA.

The key connection interface in the MicroBlaze embedded processor for adding IP cores is the OPB. Supporting a 2-GB address space, the OPB is essentially an internal address, data, read/write/chip-select interconnection that allows the MicroBlaze execution unit to communicate with on-chip peripherals.

The example MicroBlaze processor implements a very basic I/O processor; you

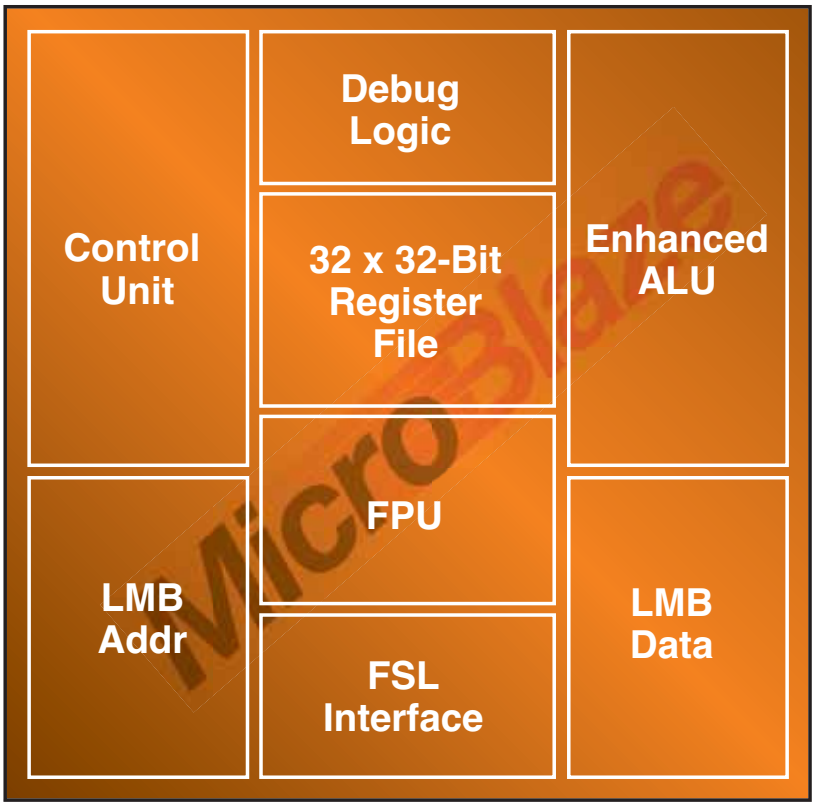

Figure 1 – MicroBlaze block diagram

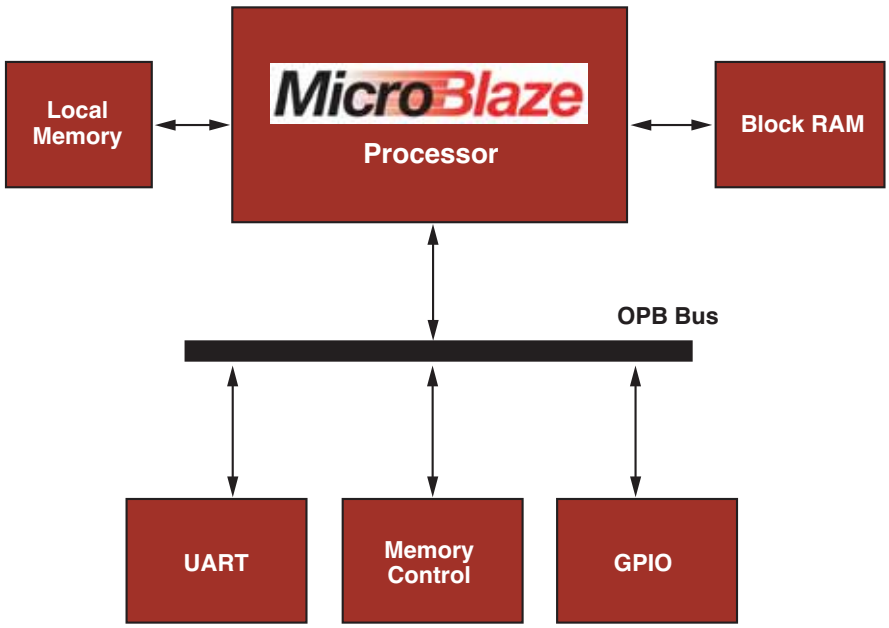

Figure 2 – Typical MicroBlaze configuration

could also write a small amount of C code to implement a system that receives commands from the UART, decodes the commands, and changes the state of the GPIO ports based on those commands. In fact, this is almost exactly how an impact/thermal printer – found in every grocery store or shopping mall – functions.

Slave and Master OPB Cores

The MicroBlaze OPB has a very important characteristic – it supports slave and master devices. Furthermore, the OPB provides support for dual master/slave peripherals.

The vast majority of available IP cores for the MicroBlaze processor can be referred to as slave peripherals. A slave device only performs an action when the MicroBlaze instruction unit, under firmware control, issues a read/write operation to the slave core.

The GPIO core is an OPB slave peripheral. The GPIO core provides an address window that allows the MicroBlaze execution unit to read/write the pin states associated with the core. You could then develop a simple firmware routine to flash LEDs or load data into an impact or thermal print head in a receipt printer.

It is also possible to develop a unique IP core and add it into the MicroBlaze architecture as a hybrid master/slave OPB core. You could also develop a master/slave OPB core such that it presents an address window to the MicroBlaze execution unit to receive instructions. Once the master/slave OPB core has received the instructions, the core could then assume control of the OPB bus and gain access to the entire address space available to the MicroBlaze execution unit. This type of core could easily access internal memory, external memory, or any other OPB-connected peripheral in the FPGA design.

The ability to add a custom IP core into the MicroBlaze architecture that acts as a bus master in conjunction with the MicroBlaze execution unit is one of the most powerful features of the MicroBlaze architecture. Properly applied, a simple master and slave OPB core could easily provide significant performance increases in overall functionality.

Applying the Master/Slave OPB

Four peripherals – a 32-bit MicroBlaze execution unit, a UART, an external memory controller, and a GPIO port – form the basic processing architecture for thermal receipt printers, commonly found in grocery stores, malls, and consumer electronic outlets across the country.

Basic Firmware of a Font-Based Printer

The processing architecture in a thermal receipt printer is conceptually simple. The printer receives an ASCII command from a host (usually a cash register), looks up a font to determine the picture of the ASCII command, assembles a string of fonts in an array in memory, and transfers that array of font characters to a print head in a raster fashion.

In a raster-based thermal printer, assembling a line of text from a font is the dominant process and requires the most amount of CPU power. The process for building a line of text from a font can be visualized by the following pseudo-source-code:

```
PlaceFontCharacter(char ASCII,
int column, int row)

{

  font_location=FindFontFromASCIICode(ASCII);
  (Line 1)

  for (i=0;i<height_of_font;i++)   (Line 2)

  {

  font_raster=*(font_location+i)  (Line 3)

  buffer_raster=*( MoveTo(column,row)+i) (Line 4)

  buffer_raster= ShiftLeftOrRight(font_raster) |
  buffer_raster          (Line 5)

}

}
```

Let's examine the basic routine fundamental to all raster-based printers. First, from the "idle" code of a printer, a function call is made to the PlaceFontCharacter routine. This routine receives the ASCII code for the letter to be printed and the row and column on the receipt where the letter should be placed.

Next, the location of the picture of the ASCII code from within the font (stored in external memory) must be determined (line 1).

A simple for-loop is then executed in which the process is to first read the raster of the character from the font (lines 2 and 3).

Next, a read is performed in which the current contents of the buffer (or the location to which the character should be written) are read into a temporary value (line 4).

Finally, the font raster must be shifted left or right, logically "or"ed with the current contents of the buffer, and this new value must be written into the buffer (line 5).

Further simplified, the loop looks like this:

1. Read font
2. Read buffer
3. Shift font
4. Logically or with buffer contents
5. Write to buffer
6. Increment loop index to get next font raster
7. Go back to step #1 if not complete

It is also important to note that the shift operations are required to create almost every ASCII text font known. For example, a standard font is 9 bits wide and 11 bits tall and has a character-to-character space of 2 bits, resulting in an 11 x 11 font. Because microprocessors operate on 8-bit boundaries, a series of shift operations must be performed to create an 11-bit alignment, properly framing a line of ASCII text within a buffer.

The pseudo-code and simplified loop is common to nearly every font-based printer produced.

Implementing this code can easily require more than 30 assembly instructions to manage the pointers, indexes, and temporary variables. And because the fonts and buffers are almost always located in external SRAM/SDRAM, at least three bus accesses (read-read-write) are required.

Totaling up the single access instructions (moves, adds, subtracts) to manage the loop variables, the clock cycles required for bus accesses, and the multi-clock-cycle instructions such as shift-left and shift-right operations, the basic process of constructing a font requires 50-70 CPU clock cycles per raster. The clock-cycles-per-raster count ultimately determines the maximum print speed.

OPB Mastering for Maximum Print Speed

Through the development of a simple master/slave OPB core, the process of building a character from a font can be changed from requiring 50-70 CPU clock cycles to requiring only the three read-read-write bus accesses actually required to build the font.

The firmware routine that places the font character could be changed to look like this:

```
PlaceFontCharacter(char ASCII, int column, int
row)

{

  font_location=FindFontFromASCIICode(ASCII);

  *(FontProcessorBaseAddress+FontLocation
  Register)=font_location;

  *(FontProcessorBaseAddress+TargetColumn
  Register)=column;

  *(FontProcessorBaseAddress+TargetRow
  Register)=row;

  *(FontProcessorBaseAddress+FontHeight
  Register)=global_font_height;

  *(FontProcessorBaseAddress+Enable
  Register)=1;

  do { NOP; } while
(*FontProcessorBaseAddress+StatusRegister)
==BUSY

}
```

You can recode the PlaceFontCharacter routine to write the required data to construct the character from the font into a series of register writes to a font processor.

You can develop the font processor IP core using the MicroBlaze OPB and Xilinx EDK tool to perform the hardware implementation of the firmware loop that places the font.

Once you have developed the font processor IP core, the MicroBlaze execution unit will configure the core, enable the core, and wait for the font processor to complete its operation.

You can also design the font processor IP core as a simple OPB slave/master peripheral. In slave mode, the core would receive the setup information from the MicroBlaze embedded processor.

Once enabled, the font processor would take control of the OPB, as the bushmaster, and then directly access the required memory locations to get the font rasters, the buffer rasters, perform the shift operations, and write the new data back into the buffer.

You can implement the font processor IP core as a simple synchronous state machine using the Xilinx EDK-provided OPB IPIF (IP interface functions) provided with the EDK environment.

The data pipeline for the core is shown in Table 1. This pipeline flow shows that only 15 clock cycles are required (once the font processor's pipeline is full) to build each raster of the font. Compared to a 50-70 clock-cycle count for a firmware-only implementation, the font processor realizes a performance boost of 3.3x-4.5x.

Conclusion

The MicroBlaze 32-bit processor provides a high-performance execution unit capable of operating at speeds as fast as 200 MHz in Xilinx FPGAs. The MicroBlaze core utilizes the OPB to connect its execution unit to all other IP cores within the FPGA.

The OPB has the unique capability of supporting IP cores that can act as master/slave peripherals.

Through the bus mastering feature of the OPB cores, it is possible to implement intelligent peripherals capable of performing high-speed data access and manipulation functions directly within FPGA logic.

You can invoke a bus mastering OPB core directly from the MicroBlaze firmware using simple, standard C functions. You can also add one or more OPB master cores to function in parallel with the execution unit of the MicroBlaze embedded processor.

Utilizing the OPB mastering functions in custom IP cores provides a unique method for accelerating the functions of a product by converting the functions of C code into direct hardware implementations within the FPGA architecture.

The source code and clock-cycle calculations presented in this article can be found under the "FreeFromFLE" category at *www.fl-eng.com.*

Data-Flow Stages	Stage #1	Stage #2	Stage #3	Stage #4	Stage #5
Memory Operations	ReadFont (Five Clocks)	ReadBuffer (Five Clocks)	ReadNextFont (Five Clocks)	WriteBuffer (Five Clocks)	Exit and Return to MicroBlaze Core
Processing Operations	Set Busy and Become OPB Master		Shift Left (One Clock) Shift Right (One Clock) Logic Or (One Clock) SaveResult (One Clock)	Return to Stage #2 if Not Complete	

Table 1 – MicroBlaze core data pipeline