The Simple MicroBlaze Microcontroller (SMM) is a small form-factor 32-bit microcontroller based on the MicroBlaze processor that can be instantiated into an FPGA design quickly and easily.

Using a small microcontroller programmed in C or C++ can be more efficient than doing the same function in HDL. This application note provides a new way to easily add a simple MicroBlaze microcontroller without having to learn new tools. The controller is instantiated directly into the HDL and can be used immediately in a standard FPGA design flow without special scripts or complicated steps. Only three files are necessary to get started. The microcontroller can be used for a wide range of applications; like user interface control, general purpose processing, monitoring, encryption/decryption engine, or as a state machine replacement to name a few.

The hardware and software requirements for this reference system are:

- Xilinx® ML605 board, Xilinx SP605 board, or Xilinx Spartan®-3A Starter Kit
- RS232 serial cable and serial communication utility (HyperTerminal)
- Xilinx Software Development Kit (SDK) 12.1
- Xilinx Integrated Software Environment (ISE®) 12.1
- Xilinx Embedded Development Kit (EDK) 12.1 is optional

The microcontroller comes initially implemented for Spartan®-6, Spartan-3A/AN, Virtex®-6, and Virtex-5 architectures and can be modified to support other architectures.

Starting with ISE® Design Suite 11.1, MicroBlaze software development is fully supported by the standalone SDK, allowing C and C++ applications to be created and debugged without the need for EDK.

The microcontroller comes pre-configured with two options: a UART option and a debug option. Once the code is debugged, the user can switch netlists to reduce the size of the design.

With a total of 3 files (2 for hardware implementation, 1 for software), a complete 32-bit MicroBlaze microcontroller can be added to any FPGA design.
Simple MicroBlaze Microcontroller Features

- 32-bit MicroBlaze RISC processor
- 8KB of internal RAM/ROM
- Software application is stored in BRAMs at configuration time
- 32-bit wide Input and Output ports
- 64KB of addressable user space
- Interrupt input
- Optional UART
- Full C/C++ support
- Optional debug support
- Simple hardware interface

SMM is available in 4 pre-configured netlists for the Spartan-6, Spartan-3 (A, AN, ADSP), Virtex-6 and Virtex-5 architectures.

- SMM with Debug support and UART option
- SMM with Debug support and no UART option
- SMM with UART option and no Debug support
- SMM with no UART option and no Debug support

The user could start with Debug support during the development phase then switch to the no Debug support netlist for production. The estimated size for each implementation using default mapping options for ISE 12.1 is shown in Table 1. Different mapping options might yield slightly different results.

Table 1: SMM Size

<table>
<thead>
<tr>
<th>FPGA Family</th>
<th>Spartan-6</th>
<th>Spartan-3</th>
<th>Virtex-6</th>
<th>Virtex-5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LUTs</td>
<td>FFs</td>
<td>Slices</td>
<td>LUTs</td>
</tr>
<tr>
<td>Configuration</td>
<td>LUTs</td>
<td>FFs</td>
<td>Slices</td>
<td>LUTs</td>
</tr>
<tr>
<td>SMM + UART + Debug</td>
<td>1340</td>
<td>1200</td>
<td>510</td>
<td>1770</td>
</tr>
<tr>
<td></td>
<td>1060</td>
<td>860</td>
<td>320</td>
<td>1510</td>
</tr>
<tr>
<td>SMM + UART</td>
<td>970</td>
<td>750</td>
<td>370</td>
<td>1390</td>
</tr>
<tr>
<td>SMM + Debug</td>
<td>690</td>
<td>440</td>
<td>250</td>
<td>1130</td>
</tr>
<tr>
<td>SMM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Microcontroller Interfaces

The block diagram of the Simple MicroBlaze Controller is shown in Figure 1.
## Signal Descriptions

The Simple MicroBlaze Controller signal names and descriptions are shown in Table 2.

**Table 2: Simple MicroBlaze Controller Input and Output Signals**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIN [0:31]</td>
<td>Input</td>
<td>Data input port: input data for read operations. The data is sampled on the rising edge of CLK.</td>
</tr>
<tr>
<td>SIN</td>
<td>Input</td>
<td>Serial input: UART input signal (optional)</td>
</tr>
<tr>
<td>INTERRUPT</td>
<td>Input</td>
<td>Interrupt Input: generate an interrupt to the microcontroller. Interrupts have to be enabled in the application code</td>
</tr>
<tr>
<td>RESET</td>
<td>Input</td>
<td>Reset Input: Active high reset input</td>
</tr>
<tr>
<td>CLK</td>
<td>Input</td>
<td>Clock Input: the clock is used by the entire system and interfaces and can be from DC to the max operating frequency for the particular FPGA</td>
</tr>
<tr>
<td>DOUT [0:31]</td>
<td>Output</td>
<td>Data Output port: output data is valid during CS for two clock cycles</td>
</tr>
<tr>
<td>BE [0:3]</td>
<td>Output</td>
<td>Byte Enables: shows the active byte lanes. Active high</td>
</tr>
<tr>
<td>ADDR [0:15]</td>
<td>Output</td>
<td>User Address: the address is valid for 2 clock cycles during CS</td>
</tr>
<tr>
<td>CS</td>
<td>Output</td>
<td>Chip Select: active high select for read and write operations. Asserted for 2 clock cycles</td>
</tr>
<tr>
<td>RNW</td>
<td>Output</td>
<td>Read Not Write: active high for reads and active low for write. Valid during CS</td>
</tr>
<tr>
<td>SOUT</td>
<td>Output</td>
<td>Serial Output: UART output signal (optional)</td>
</tr>
<tr>
<td>INTERRUPT_ACK</td>
<td>Output</td>
<td>Interrupt Acknowledge output: asserted high for one clock when the interrupt is being serviced</td>
</tr>
</tbody>
</table>
The user interface provides 64KB of input and output access through a simple controller. The access time for reads and writes is always 2 clock cycles based on the assertion of CS. The read and write timing diagram is shown in Figure 2.

The address comes from the microcontroller user interface peripheral with a fixed address map from 0x1000_0000 to 0x1000_FFFF.

The data follows a big endian addressing scheme. The relationship between the address and data for words, half-word, and byte transfers is shown in Figure 3. Byte enables can be ignored when doing only word (32-bit) transfers. Unaligned transfers are not recommended.
Interrupt Timing

The system supports one level sensitive, active high external interrupt source. The microcontroller will respond to interrupts only if they have been enabled in software. If more interrupts are required, they can be multiplexed in the FPGA logic. A user addressable register could be created in the FPGA logic to keep track of the multiple interrupts. The interrupt acknowledge output signal can be used to clear the incoming interrupt signal. It is asserted for one clock cycle when the interrupt event is serviced.

Clock and Reset

The clock input can be as low as DC and as high as the ISE tools will allow. Assuming that a clock constraint is present on the clock used, no other constraints need to be specified.

The Reset input is active high and can be synchronous or asynchronous to the clock input. The reset input will be synchronized to the clock internally.

Serial Input and Output

An option for a serial 16450 UART is provided as a pre-configured version. The Baud Rate is programmed in software to keep the UART independent from the clock input. Features for the UART include:

- 5, 6, 7 or 8 bits per character
- Odd, even or no parity detection and generation
- 1, 1.5 or 2 stop bit detection and generation
- False start bit detection and recover
- Line break detection and generation
- Internal loop back diagnostic functionality

Debug Interface

The microcontroller uses the BSCAN resource of the target FPGA to communicate through JTAG. The interface is internal to the FPGA and does not require pins. It uses the dedicated FPGA JTAG pins. For architectures with a single BSCAN primitive, the microcontroller system would need to be customized or used in the no-debug configuration to be able to simultaneously use ChipScope™ Pro logic analyzer.
Using the Microcontroller

The microcontroller system comes in four pre-configured implementations for Spartan-6, Spartan-3, Virtex-6, and Virtex-5 FPGA architectures. For other architectures, see Customizing the Simple MicroBlaze Microcontroller, page 23.

For the FPGA design, ISE is required. For the software flow, SDK is required. EDK is only needed if there is a need to customize the microcontroller system beyond what can be accomplished with the user interface.

As shown in Figure 5, only three files are necessary to complete a microcontroller FPGA project: two files to be used with ISE, and one file to be used with SDK.

**Design Directory Contents**

The directories and files shown are provided to enable a Simple MicroBlaze Microcontroller design.

Each pre-configured directory contains an `hw` and `sw` directory. The `hw` directory contains the files needed for ISE. The `sw` directory contains the file needed for SDK. The `SMM_Src` directory will be covered in the customization chapter. The `SMM_Ref` directory is described in Reference Design Overview, page 26.
SMM FPGA Design Flow

The FPGA design flow follows the standard ISE design flow (Figure 6). The microcontroller system can be instantiated into VHDL and Verilog designs as a black box. The BRAM Memory Map file (*smm.bmm*) must be added to the ISE project. The netlist file (*smm.ngc*) is picked up during the translate phase.

![ISE Design Flow Diagram](image)

**Figure 6: ISE Design Flow**

After BitGen, a new BRAM Memory Map file (*smm_bd.bmm*) will contain the locations of the BRAMs inside the FPGA. That information will be used to store code into the microcontroller main memory.
SMM HDL Instantiation Templates

The microcontroller can be instantiated in VHDL and Verilog design at any level of hierarchy. Use the module declarations and instantiation templates shown in Figure 7, Figure 8, Figure 9, page 9, and Figure 10, page 9.

--- With UART

```vhdl
component smm is
    port (
        SIN : in std_logic;
        SOUT : out std_logic;
        CLK : in std_logic;
        RESET : in std_logic;
        DIN : in std_logic_vector(0 to 31);
        DOUT : out std_logic_vector(0 to 31);
        BE : out std_logic;
        RNW : out std_logic;
        ADDR : out std_logic_vector(0 to 15);
        CS : out std_logic;
        INTERRUPT : in std_logic;
        INTERRUPT_ACK : out std_logic
    );
end component;
```

attribute BOX_TYPE : STRING;
attribute BOX_TYPE of smm : component is "black_box";

--- Without UART

```vhdl
component smm is
    port (
        CLK : in std_logic;
        RESET : in std_logic;
        DIN : in std_logic_vector(0 to 31);
        DOUT : out std_logic_vector(0 to 31);
        BE : out std_logic;
        RNW : out std_logic;
        ADDR : out std_logic_vector(0 to 15);
        CS : out std_logic;
        INTERRUPT : in std_logic;
        INTERRUPT_ACK : out std_logic
    );
end component;
```

attribute BOX_TYPE : STRING;
attribute BOX_TYPE of smm : component is "black_box";

--- With UART

```vhdl
smm_i : smm
    port map (
        SIN => SIN,
        SOUT => SOUT,
        CLK => CLK,
        RESET => RESET,
        DIN => DIN,
        DOUT => DOUT,
        BE => BE,
        RNW => RNW,
        ADDR => ADDR,
        CS => CS,
        INTERRUPT => INTERRUPT,
        INTERRUPT_ACK => INTERRUPT_ACK
    );
```

--- Without UART

```vhdl
smm_i : smm
    port map (
        CLK => CLK,
        RESET => RESET,
        DIN => DIN,
        DOUT => DOUT,
        BE => BE,
        RNW => RNW,
        ADDR => ADDR,
        CS => CS,
        INTERRUPT => INTERRUPT,
        INTERRUPT_ACK => INTERRUPT_ACK
    );
```

Figure 7: VHDL Component Declaration

Figure 8: VHDL Component Instantiation
// With UART
module smm
(SIN, SOUT,
  CLK, RESET,
  DIN, DOUT, BE, RNW, ADDR, CS,
  INTERRUPT, INTERRUPT_ACK);

input SIN;
output SOUT;
input CLK;
input RESET;
input [0:31] DIN;
output [0:31] DOUT;
output [0:3] BE;
output RNW;
output [0:15] ADDR;
output CS;
input INTERRUPT;
output INTERRUPT_ACK;
endmodule

// Without UART
module smm
(CLK, RESET,
  DIN, DOUT, BE, RNW, ADDR, CS,
  INTERRUPT, INTERRUPT_ACK);

input CLK;
in
put RESET;
input [0:31] DIN;
output [0:31] DOUT;
output [0:3] BE;
output RNW;
output [0:15] ADDR;
output CS;
input INTERRUPT;
output INTERRUPT_ACK;
endmodule

Figure 9: Verilog Module Declaration

// With UART
(*BOX_TYPE="black_box")
smm_smm_i (
  .SIN (SIN),
  .SOUT (SOUT),
  .CLK (CLK),
  .RESET (RESET),
  .DIN (DIN),
  .DOUT (DOUT),
  .BE (BE),
  .RNW (RNW),
  .ADDR (ADDR),
  .CS (CS),
  .INTERRUPT (INTERRUPT),
  .INTERRUPT_ACK (INTERRUPT_ACK)
);

// Without UART
(*BOX_TYPE="black_box")
smm_smm_i (
  .CLK (CLK),
  .RESET (RESET),
  .DIN (DIN),
  .DOUT (DOUT),
  .BE (BE),
  .RNW (RNW),
  .ADDR (ADDR),
  .CS (CS),
  .INTERRUPT (INTERRUPT),
  .INTERRUPT_ACK (INTERRUPT_ACK)
);

Figure 10: Verilog Module Instantiation
BRAM Memory Map (BMM) File

The smm.bmm file provides the memory map and BRAMs allocation for the BRAMs used by the processor as main memory.

The BMM file is the same for all pre-configured versions of the microcontroller for a given architecture. The contents of the file are listed in Figure 11.

The hierarchy of the BMM file must match the hierarchy of the FPGA design. The BMM file assumes that the microcontroller is instantiated at the top level with the name smm_i.

If the microcontroller is instantiated in a sub module, the BMM file needs to be modified to reflect each hierarchy level.

For example if the microcontroller was instantiated inside a sub-module named micro below the top level, the BMM file BUS_BLOCK hierarchy would have to be modified to:

```plaintext
top/micro/smm_i/lmb_bram/lmb_bram/ramb...
```

**Using an S3ADSP part:** The RAMB16 primitive for the S3ADSP part is slightly different from the one used in the S3A part. An additional modification needs to be made to the BMM file BUS_LOCK hierarchy as shown below to account for the automatic primitive replacement.

```plaintext
smm_i/lmb_bram/lmb_bram/ramb16bwe_0/RAM16BWER [31:24];
smm_i/lmb_bram/lmb_bram/ramb16bwe_1/RAM16BWER [23:16];
smm_i/lmb_bram/lmb_bram/ramb16bwe_2/RAM16BWER [15:8];
smm_i/lmb_bram/lmb_bram/ramb16bwe_3/RAM16BWER [7:0];
```
Adding the Software Application to the Bitstream

The software application can be loaded into the BRAMs as part of the configuration data. This enables the application to run as soon as the FPGA is configured and the microcontroller Reset signal has been de-asserted.

There are two ways to generate the final bitstream file:

1. Use SDK as shown in [Step-by-Step SMM Design Example](#).
2. Use the utility `data2mem` in command line mode. The files needed are the ISE `{design}.bit` file, the `{smm_bd}.bmm` file, and the `{application}.elf` executable file generated by SDK. In a DOS prompt you would type:

   ```
data2mem -bm smm_bd.bmm -bt design.bit -bd application.elf tag microblaze_0 -o b final.bit
   ```

   The `final.bit` file would contain the application code pre-loaded into the BRAMs.
The SMM Software Flow

A single file (smm.xml) contains all the information necessary to start a software project. Using SDK does not require ISE or EDK to be installed. A standalone SDK installation would be sufficient.

The bit file and bmm file generated by ISE will be needed to program the FPGA and debug the application. They are not required to develop the application.

Drivers and libraries, constituting the software platform, are provided to design a C or C++ application. SDK includes the GNU Compiler Collection as well as common utilities to support C and C++ applications.
SMM Address Map

Table 3 shows the address map for the microcontroller. The user interface address on ADDR [0:15] correlates to the 64KB User Interface space allocated.

<table>
<thead>
<tr>
<th>SMM RAM/ROM</th>
<th>User Interface</th>
<th>UART 16450 Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 to 0x00001FFF</td>
<td>0x10000000 to 0x1000FFFF</td>
<td>0x80000000 to 0x8000FFFF</td>
</tr>
</tbody>
</table>

Keeping a Small Software Footprint

The EDK libraries contain standard C functions for I/O, such as printf and scanf. These functions are large and are not well suited for a simple microcontroller application. The Xilinx library provides smaller equivalent stdio functions that should be used instead.

To print a string of characters to the UART use the command `void print (char *)`.

To output an integer to the UART use the command `void putnum (int)`.

Instead of using printf use the command `void xil_printf (const *char ctrl1,...)`.

This function is similar to printf but much smaller in size (only 1 kB). It does not have support for floating point numbers. xil_printf also does not support printing of long (such as 64-bit numbers).

Some other suggestions:

- When creating the linker script for the application, make sure that the stack and heap sizes are appropriate for the application.
- After the code is debugged in SDK, switch to the active configuration for the application from Debug to Release.
- When using Xilinx peripherals (like the UART), use the low level drivers for the peripheral. The low level drivers have a "_l.h" file name. For example: `xuartns550_l.h`
- Be careful of unnecessary variable initialization. Initialized variables are part of the data section which will increase the size of the executable.
- Use the smallest variable type needed for the application.
The following steps create a simple microcontroller design to illustrate the flow. The example design covers a Spartan-6 FPGA board as well as a Virtex-6 FPGA board.

The design makes use of the board switches, LEDs, and RS232 UART to create a simple microcontroller example design. The processor reads the switches values using its input port and writes the values to the LEDs using its output port. The UART is used to display simple messages to a terminal.

**Hardware Steps**

1. Start ISE 12.1 Project Navigator and create a new project. *File > New Project...*
2. Select a project location (for example: `C:\SMM_proj\` and a project name (for example: `SimpleMicro`).
3. Click Next.
4. Select the device properties.
   a. For the Spartan-6 SP605 board:
      - Family: Spartan-6
      - Device: XC6SLX45T
      - Package: FGG484
      - Speed: -3
   b. For the Virtex-6 ML605 board:
      - Family: Virtex-6
      - Device: XC6VLX240T
      - Package: FF1156
      - Speed: -1
5. Select the project properties:
   - Synthesis Tool: XST
   - Preferred Language: VHDL (Verilog can also be used)
6. Leave the default settings for the other properties.
7. Click Next, then Finish.
8. Go to *Project > New Source...*
   a. Select **VHDL Module**
   b. Type the file name `simplemicro`.
   c. Click Next.
9. Add the ports for the clock, reset, UART, pushbuttons, and LEDs (Figure 13).

![Figure 13: Simplemicro Module Ports](X1141_13_060809)

10. Click **Next**, then **Finish**.

11. Copy the `smm.ngc` and `smm.bmm` files from the SMM\SMM_S6\SMM_Full\hw directory for the **SP605 project**, or SMM\SMM_V6\SMM_Full\hw directory for the **ML605 project** to the ISE project directory (for example: `C:\SMM_proj\SimpleMicro`)

12. In ISE, open the `simplemicro.vhd` file.

13. Add the SMM component declaration between the architecture and the begin statements, as shown below.

```vhdl
architecture Behavioral of simplemicro is
    component smm is
        port (  
            SIN : in std_logic;  
            SOUT : out std_logic;  
            CLK : in std_logic;  
            RESET : in std_logic;  
            DIN : in std_logic_vector(0 to 31);  
            DOUT : out std_logic_vector(0 to 31);  
            BE : out std_logic_vector(0 to 3);  
            RNW : out std_logic;  
            ADDR : out std_logic_vector(0 to 15);  
            CS : out std_logic;  
            INTERRUPT : in std_logic;  
            INTERRUPT_ACK : out std_logic 
        );
        end component;

        attribute BOX_TYPE : STRING;
        attribute BOX_TYPE of smm : component is "black_box";
    begin
```
14. Add the signal declarations below before the begin statement for the internal signals.

```vhdl
signal CS : std_logic;
signal RNW : std_logic;
signal DOUT : std_logic_vector (0 to 31);
signal DIN : std_logic_vector (0 to 31);
signal leds_int : std_logic_vector (3 downto 0);
begin
```

15. Add the SMM instantiation after the begin statement, as shown below.

```vhdl
begin

smm_i : smm
port map (  
  SIN => UART_sin,
  SOUT => UART_sout,
  CLK => Clock,
  RESET => Reset,
  DIN => DIN,
  DOUT => DOUT,
  BE => open,
  RNW => RNW,
  ADDR => open,
  CS => CS,
  INTERRUPT => '0',
  INTERRUPT_ACK => open
);
```

Because we will be using only one read address and one write address, there is no need to decode the address coming from the microcontroller. We will respond to all addresses in the 64KB range. The DIN port will be used to sample the pushbuttons and the DOUT port will drive the LEDs. We will capture the DOUT port during CS to hold the LED values.

16. Add the HDL code before the end Behavioral; statement for the pushbuttons and LEDs, as shown below.

```vhdl
process( Clock ) is  begin
  if ( Clock'event and Clock = '1' ) then
    if ( CS = '1' and RNW = '0') then
      leds_int <= DOUT (0 to 3);
    else
      leds_int <= leds_int;
    end if;
  end if;
end process;

LEDs <= leds_int;
DIN(0 to 3) <= Buttons;
end Behavioral;
```

17. Save and close the HDL file.

18. Add the `smm.bmm` and `smm.ngc` files to the ISE project. Project > Add Source... Select `smm.bmm` and `smm.ngc`. 
19. Click OK.
   The BMM file does not need to be modified since the microcontroller is instantiated at the top level.

20. In ISE, double-click on Synthesize - XST to synthesize the design.

21. Now add the timing and pin constraints.
   - For the ML605 use the position LEDs and pushbuttons.
   - For the SP605 use the GPIO LEDs and the position pushbuttons.

22. In the ISE processes window, expand the User Constraints selection.


24. Click Yes to add the UCF file to the project.

25. Double-click below TIMESPEC Name to add a new constraint.

26. Select the clock period for the board, then click OK.
   The board single-ended clock was chosen to keep the design simple.
   - SP605: 27 MHz
   - ML605: 66 MHz

27. Close the Timing Constraint window. Click Yes to save the constraint file.

29. Add the pin assignment constraints shown below for the ports.

For the SP605 Board:

NET "Reset" LOC = "H8" | IOSTANDARD = LVCMOS15;
NET "Clock" LOC = "AB13" | IOSTANDARD = LVCMOS25;
NET "UART_sin" LOC = "H17" | IOSTANDARD = LVCMOS25;
NET "UART_sout" LOC = "B21" | IOSTANDARD = LVCMOS25;
NET "LEDs<0>" LOC = "D17" | IOSTANDARD = LVCMOS25;
NET "LEDs<1>" LOC = "AB4" | IOSTANDARD = LVCMOS25;
NET "LEDs<2>" LOC = "D21" | IOSTANDARD = LVCMOS25;
NET "LEDs<3>" LOC = "W15" | IOSTANDARD = LVCMOS25;
NET "Buttons<0>" LOC = "F3" | IOSTANDARD = LVCMOS15;
NET "Buttons<1>" LOC = "G6" | IOSTANDARD = LVCMOS15;
NET "Buttons<2>" LOC = "F5" | IOSTANDARD = LVCMOS15;
NET "Buttons<3>" LOC = "C1" | IOSTANDARD = LVCMOS15;

For the ML605 Board:

NET "Reset" LOC = "H10" | IOSTANDARD = SSTL15 | PULLUP;
NET "Clock" LOC = "U23" | IOSTANDARD = LVCMOS25;
NET "UART_sin" LOC = "J24" | IOSTANDARD = LVCMOS25;
NET "UART_sout" LOC = "J25" | IOSTANDARD = LVCMOS25;
NET "LEDs<0>" LOC = "AD21" | IOSTANDARD = LVCMOS25;
NET "LEDs<1>" LOC = "AE21" | IOSTANDARD = LVCMOS25;
NET "LEDs<2>" LOC = "AH27" | IOSTANDARD = LVCMOS25;
NET "LEDs<3>" LOC = "AH28" | IOSTANDARD = LVCMOS25;
NET "Buttons<0>" LOC = "H17" | IOSTANDARD = LVCMOS15;
NET "Buttons<1>" LOC = "G17" | IOSTANDARD = LVCMOS15;
NET "Buttons<2>" LOC = "A19" | IOSTANDARD = LVCMOS15;
NET "Buttons<3>" LOC = "A18" | IOSTANDARD = LVCMOS15;

30. Save and close the file.
31. Select simplemicro.vhd in the ISE Sources window.
   The design is ready to be implemented.
32. Double-click on Generate Programming File in the ISE Processes window.
   The hardware steps are complete. Verify that the ISE project directory contains a simplemicro.bit file and a smm_bd.bmm file.
Software Steps

1. Inside the C:\SMM_proj directory, create a SimpleMicro_SDK directory.
   - Copy the smm.xml file from the SMM\SMM_S6\SMM_Full\sw directory for the SP605 project, or
   - SMM\SMM_V6\SMM_Full\sw directory for the ML605 project
   20, to the SimpleMicro_SDK directory.
2. Start Xilinx SDK 12.2.
3. Browse to the C:\SMM_proj\SimpleMicro_SDK directory for the workspace selection.
4. Click OK.
   The Welcome page contains information and help topics for the software development kit.
   The first step is to select the hardware platform for the project.
5. Go to File > New > Xilinx Hardware Platform Specification.
7. Click Finish.

Creating the Software Application

SDK will create a Board Support Package based on the hardware platform. The application will be using the libraries and drivers created for the platform.

1. Go to File > New > Xilinx C Project
2. Select simple for the project name, select Empty Application, then click Next.

Adding the Application

1. Add a new source file for the project by selecting File > New > Source File
2. Name the Source File simple.c.
3. Click Finish.
4. Add the application as shown below to the simple.c file you just created.

```c
#include <stdio.h>
#include "xuartns550_l.h"
#include "xio.h"
#include "xbasic_types.h"

#define STDOUT_BASEADDR 0x80000000

int main()
{
    Xuint8 Buttons;

    /* Setup the UART Baud Rate to 9600*/
    XUartNs550_SetBaud(STDOUT_BASEADDR, 27000000, 9600);
    /* Use this for ML605
    XUartNs550_SetBaud(STDOUT_BASEADDR, 66000000, 9600);
    */
    XUartNs550_SetLineControlReg(STDOUT_BASEADDR,
                                  XUN_LCR_8_DATA_BITS);
```
print("Simple Microcontroller Application\n\r");

while (1) {
   /* Read the Push Button Values */
   Buttons = XIo_In8(0x10000000);
   /* Output to the LEDs */

   XIo_Out8(0x10000000, Buttons);
}

The second argument in the XUartNx550_SetBaud function call indicates the clock frequency of the microcontroller. The third argument indicates the desired UART baud rate. The application reads the pushbutton values and outputs the result to the LEDs.

5. Save the file. The project builds automatically.
   The included files are provided by the standalone platform and can be found under standalone/microblaze_0/include in the C/C++ Projects perspective window. The include files for the UART 16450 drivers are located in that directory.

6. Create a new Linker Script for the application by selecting Xilinx Tools > Generate Linker Script…

7. Click Generate, then Yes.
   The software application is now ready to be used.
Programming the FPGA

1. Connect the two Type-A to Mini-B USB cables.
2. Power-on the board.
3. Open Hyperterminal and connect at 9600, no parity.
4. Go to Xilinx Tools > Program FPGA...
5. Browse to the ISE project C:\SMM_proj\SimpleMicro and use simplemicro.bit for the bit file.
6. Browse to the ISE project C:\SMM_proj\SimpleMicro and use smm_bd.bmm for the Bmm File.
7. Select simple.elf to be initialized inside the BRAM memory (Figure 15).

![Program FPGA](image)

**Figure 15:** FPGA Programming

8. Click Program. The software application will be added to the BRAMs inside the bitstream. A new bit file, download.bit, is created.

9. Check Hyperterminal for the print statement and press the pushbuttons to turn on the LEDs.
Debugging the Application

Because we selected a microcontroller version with Debug Support, we can debug the application.

1. Go to Run > Debug Configurations
2. Click on New launch configuration

A new debug configuration is created for the simple application.

3. Click Debug.

The Debug perspective provides all the necessary tools and information to debug a software application.

The editor window shows the file being debugged. Placing the cursor over variables shows the current value for the variable.

The Debug window provides information on the status of the session. The application is currently stopped, waiting for input. The green arrow resumes the application; the red square terminates the session. Multiple stepping options are available, including stepping in assembly mode.

4. Click the Registers tab in the top right window to view the MicroBlaze registers. Any changes from the last context are shown in red.

5. To view the source disassembly select Window > Show View > Disassembly.

6. Click the Memory tab on the bottom left.

7. Click on the “+” sign to add a memory selection. Type 0x0, which is the start of the main memory, in the address field of the memory window. Click OK.

8. Double-click in the column on the left of the print statement to set a breakpoint.

9. Click the Resume icon on the toolbar to go to the breakpoint.

The line will be highlighted in green.

10. Click on the Step Over icon on the toolbar to print the line.

Registers and memory values can be modified from the debugger.

11. Step to the XIo_Out8 line. Click the Variables tab in the top right window.

The value of the variable Buttons reflects the pushbutton values.

12. Right-click the variable Buttons and select Change Value... Set the value to 0xFF.

13. Step to execute the XIo_Out8 function. All the LEDs should turn on.

Figure 16: Debug Configuration
14. Click the **Terminate** icon to stop the debug session.

15. Click the C/C++ perspective above the Debug window to go back to the projects view.

The bitstream file created by SDK with the application loaded into BRAMs is located at

C:\SMM_proj\SimpleMicro_SDK\SMM_Full_hw_platform\download.bit.

The updated bit file can be used to create a PROM file or can be used directly by iMPACT to configure the FPGA.

---

Customizing the Simple MicroBlaze Microcontroller

As the microcontroller is built using MicroBlaze, more standard peripherals and options are available to customize the embedded system. A user might want to use a different FPGA architecture or add more main memory or a Floating Point Unit or a standard SPI or I2C peripheral.

To customize the microcontroller, EDK 12.1 is required. This section is not intended as a Xilinx Platform Studio (XPS) tutorial. Some prior knowledge of XPS is recommended. To understand the flow to customize the microcontroller we increase the main memory for the V5 from 8 KB to 16 KB. **Figure 17** shows the XPS project diagram for the microcontroller.

**Figure 17**: XPS Project Diagram
SMM Source Directory Contents

The source files to generate the microcontroller are provided for each configuration (Figure 18).

Each architecture directory contains a complete XPS project for the four pre-configured versions of the Simple MicroBlaze Microcontroller. Our example modifies the SMM_Src\V5\SMM_Full project.

Figure 18: Source Directory Contents
SMM Customization

1. Copy the SMM_Src\V5\SMM_Full directory to a working directory. For example C:\SMM_proj\custom\SMM_Full.
2. Start Xilinx Platform Studio (XPS) 12.1 and open the smm.xmp project.
3. Select the Addresses tab in the System Assembly View.
4. Expand microblaze_0's Address Map.
5. For the dlimb_cntlr and ilimb_ctlr, select 16K.
6. Go to Hardware > Generate Netlist to create the new netlist and BMM file.
7. When complete, go to Project > Export Hardware Design to SDK… to create the new xml file. Click on Export Only.
8. Copy the three required SMM files to the working ISE and SDK projects. The files locations are:
   - smm.ngc: C:\SMM_proj\custom\SMM_Full\implementation\smm.ngc
   - smm.bmm: C:\SMM_proj\custom\SMM_Full\implementation\smm_stub.bmm
   - smm.xml: C:\SMM_proj\custom\SMM_Full\SDK\SDK_Export\hw\smm.xml

When replacing an existing smm.ngc file in the ISE project, a full re-implementation of the FPGA is necessary.

When replacing an existing smm.xml file in the SDK project, SDK will automatically detect a new xml file and will rebuild the software platform to match the new configuration.

SMM Reference Designs

Reference designs are provided in the SMM_Ref directory. They are two identical designs, one for the S3A Starter Kit and the other for the ML605 board.

The reference design uses the SMM to print information to the character LCD screen. The ISE and SDK projects are provided using the smallest pre-configured version of SMM (without Debug and UART). The directory structure for the reference directory is shown in Figure 19.
Reference Directory Contents

The src directory contains the VHDL source files for the ISE project. The ise directory contains the ISE project, including the constraints file. The sdk directory contains the SDK project, including the C source file.

Reference Design Overview

The reference design uses a combination of FPGA logic and C code running on the microcontroller to output messages to the LCD screen.

The microcontroller handles the initialization of the LCD and the printing of string messages to the LCD.

Because the LCD timing is very slow, with some instructions requiring up to 1.6us of wait time, a 32-bit timer located in the FPGA logic handles the wait times. The timer generates an interrupt to the SMM to indicate that the timer has reached a loaded value.

The LCD hardware timing is created in the FPGA logic. The microcontroller sends commands and data through a single 16-bit register. Bit 0 is used to differentiate between LCD data and commands.
VHDL Source Files Overview

- **lcd_ref.vhd**: Top level VHDL file. Contains the input and output ports for the design, the SMM instantiation, the 32-bit timer instantiation, and the LCD Controller instantiation. For the S3A Kit, the LCD is used in 8-bit mode. For the ML605, the LCD is used in 4-bit mode. The South pushbutton is used as a user input. The pushbutton is sampled into a register using the clock input. The register is cleared when read by the SMM.

- **timer.vhd**: Sub-module VHDL file. Contains a 32-bit counter. The timer is loaded with a software accessible register. The counter starts after the load. An interrupt is generated when the counter reaches the loaded value. The INTERRUPT_ACK signal from the SMM is used to clear the interrupt.

- **lcd_ctlr.vhd**: Sub-module VHDL file. Contains the LCD screen logic. A 16-bit software accessible register is used to output data and commands to the LCD_DB pins. Bit 0 of the register is used to generate the LCD_RS bit. The least significant bits of the register are used for LCD_DB. Once a value is written to the register, the LCD_E signal is asserted. The logic handles the hold and width times of the LCD_E signal using a small counter.

ISE Files Overview

- **lcd_ref.xise**: ISE project file
- **lcd_ref.ucf**: ISE project constraints file
- **smm.ngc**: SMM netlist without Debug Support and without UART
- **smm.bmm**: SMM BRAM Memory Map

SDK Files Overview

The sdk directory contains files needed for an SDK project. There is a single C source file for the processor.

- **lcd_ref.c**: C source file. Contains the LCD reference application source code. The application enable interrupts, initializes the LCD, writes a couple lines to the LCD, and waits for the pushbutton input to continue. Some local functions are created to make the code more readable.
  - void Timer_Wait (Xuint32 delay): Sends the delay value to the FPGA logic timer and waits for the interrupt to be serviced.
  - void timer_int_handler(void * arg): Interrupt service routine.
  - void lcd_init (void): Handles the LCD screen initialization.
  - void lcd_print (Xuint32 Line, Xuint8 *Str): Prints characters to the LCD. The Line argument indicates which of the 2 lines to print to.
  - void lcd_clear (void): Clears the LCD Screen. It includes the 1.56ms delay required after a clear.
  - lcd_4_write (Xuint16 value): For the ML605, the LCD is used in 4-bit mode. This function takes an 8-bit value and sends two 4-bit values to the LCD. It only applies for the ML605 design.
Running the Design

Follow these instructions to run the design on the S3A Kit or ML605.

1. Open the ise directory and double-click on the lcd_ref.xise file.
2. In Project Navigator double-click on Generate Programming File.
3. Close Project Navigator when finished.
5. Browse to the sdk directory for the workspace selection and Click OK.
8. Select standalone for the project name then click Finish, then OK.
9. Go to File > Import… Inside the General folder, select Existing Projects into Workspace. Click Next.
10. Browse to the sdk directory. Make sure that only the lcd_ref project is checked.
11. Click Finish.
   Wait for the project to build.
12. Connect the USB JTAG cable.
13. Power-on the board.
14. Go to Xilinx Tools > Program FPGA...
15. Browse to the ISE project and use lcd_ref.bit for the Bit File.
16. Browse to the ISE project and use smm_bd.bmm for the Bmm File.
17. Select lcd_ref.elf to be initialized inside the BRAM memory.
18. Click Program.
19. Check the LCD screen. Press the South pushbutton to clear the screen and output a new line.

References

1. UG081 MicroBlaze Processor Reference Guide
2. UG334 Spartan-3A/3AN FPGA Starter Kit Board User Guide
3. UG534 ML605 Hardware User Guide
4. UG526 SP605 Hardware User Guide
Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Description of Revisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>07/08/09</td>
<td>1.0</td>
<td>Initial Xilinx release.</td>
</tr>
<tr>
<td>02/08/10</td>
<td>2.0</td>
<td>Added support for Spartan-6 and Virtex-6 families. Replaced the ML505 reference design</td>
</tr>
<tr>
<td></td>
<td></td>
<td>with the ML605 reference design.</td>
</tr>
<tr>
<td>11/09/10</td>
<td>3.0</td>
<td>Updated tools versions, content in Table 1, and figures. Updated steps in Step-by-Step</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SMM Design Example and SMM Reference Designs.</td>
</tr>
</tbody>
</table>

Notice of Disclaimer

Xilinx is disclosing this Application Note to you “AS-IS” with no warranty of any kind. This Application Note is one possible implementation of this feature, application, or standard, and is subject to change without further notice from Xilinx. You are responsible for obtaining any rights you may require in connection with your use or implementation of this Application Note. XILINX MAKES NO REPRESENTATIONS OR WARRANTIES, WHETHER EXPRESS OR IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, IMPLIED WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL XILINX BE LIABLE FOR ANY LOSS OF DATA, LOST PROFITS, OR FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR INDIRECT DAMAGES ARISING FROM YOUR USE OF THIS APPLICATION NOTE.