Demonstration of Multitasking using ThreadX RTOS on Microblaze and PowerPC

Awais M. Kamboh, Adithya H. Krishnamurthy and Jaya Krishna K. Vallabhaneni

Abstract— The objective of this project is to implement and demonstrate multiprocessing in a real-time environment using ThreadX RTOS on PowerPC and MicroBlaze processor cores. We compared the performance of ThreadX on both the processors using custom applications. One of the major challenges involved in the project was to make ThreadX work on both the above mentioned processors seamlessly. For the implementation of a multi-tasking model, a producer consumer application that utilizes a mutex, two semaphores and a timer was programmed. A multi-threaded Matrix multiplication program was used to determine the execution as well as context switching times on both the cores. The results were then compared and the performance of the hardware configurations was analyzed. Though both Microblaze and PowerPC performed well, the PowerPC core on a Virtex-II Pro Platform was better with faster execution and context switching times.

Index Terms—**Multitasking, ThreadX, Microblaze, PowerPC**

I. Motivation and Introduction

ASSEMBLY level programs were used in the past for Embedded applications. With the advances in memory, processor speed, and development tools, high level languages like C are now used to develop embedded applications. Real-Time OS takes care of Task scheduling, Memory management, File handling etc. ThreadX is one such RTOS developed by Express Logic Inc. ThreadX Can be customized to run on various processors. Our project demonstrates a multiprocessing application on Microblaze soft-core and PowerPC. The performance of ThreadX on both these cores was measured and a comparative study is presented here.

RTOS' seldom run the same way on different underlying processor architectures. Most RTOS' are built to take advantage of specific processor architectures. For example ThreadX saves only registers a1-a4 during a Context switch in ARM processor. This warrants a thorough performance analysis of commonly-used RTOS' on different platforms. A study of development time for different platforms can be indicative of time- to-market windows for commercial products.

II. Background

A. Multitasking

Multitasking is a technique to allocate processing time among various duties or jobs, which the overall software program must perform. This usually means that the software is divided into tasks, or smaller subsets of the total problem and at run-time, creating an environment that provides each task with its own virtual processor. A virtual processor typically includes a register set, a program counter, a stack memory area, and a stack pointer. A multitasking run-time environment controls overall task execution. When a higher-priority task needs to execute, the currently running task's registers are saved in memory and the higher-priority tasks registers are recovered from memory. The process of swapping the execution of tasks is commonly called context switching and context-switching time is a commonly quoted specification for operating systems targeting the real-time or embedded systems market.

State machines and **time slicing** are two popular multitasking methods. State machines have been used to design complex systems with high reliability requirements. State machines require that the task is split into states. The state machine stays in one state at a time, and switches to another state when the specified conditions are met. Actions are performed during the transitions. States represent a situation that is stable for some time interval. Time slicing means that the kernel interrupts each process after some milliseconds and gives control to another task. Thus, each task is given CPU processing at regular intervals.

B. Microblaze

The MicroBlaze core is a 32-bit Harvard RISC architecture with a rich instruction set optimized for embedded applications. The processor is a soft core, which is implemented using general logic primitives rather than a hard, dedicated block in the FPGA. The MicroBlaze soft processor is supported in the Xilinx Spartan and Virtex series of FPGAs.

The MicroBlaze solution is designed to be flexible, giving the user control of a number of features such as the cache sizes, interfaces, and execution units. The configurability allows the user to trade-off features for size, in order to achieve the necessary performance for the target application at the lowest possible cost point.

Soft Processor: Intellectual Property (IP) core implemented using the logic primitives of an FPGA. Key benefits: Configurability or trade off between price and performance,

faster time to market, easy integration with the FPGA fabric, no processor obsolescence. The Microblaze performance depends on the configuration of the processor and the target FPGA architecture and speed grade. The number of Microblaze processors on a single FPGA is only limited by the size of the FPGA. Microblaze configurable interfaces and peripherals include timers, UARTs, interrupt controllers, GPIOs, external flash and memory controllers.

C. PowerPC

The PowerPC (Performance Optimization With Enhanced RISC) Architecture, is a 64-bit specification with a 32-bit subset. Almost all PowerPCs with a few exceptions are 32-bit.

PowerPC processors have a wide range of implementations, from high-end server CPUs to the embedded CPU market. PowerPC processors have a strong embedded presence because of good performance, low power consumption, and low heat dissipation. PowerPC 405 is a 32-bit RISC hard IP core. Virtex II Pro platform FPGAs provide upto two PowerPC 405 cores on a single device. IBM PowerPC 405 has wide acceptance in performance oriented applications as well as comprehensive 3rd party tools support. It offers excellent performance vs power characteristics in a small die area. PowerPC core supports a system frequency of atleast 300MHz, corresponding to more than 420 Dhrystone MIPS. The processor frequency can be dynamically changed for reduced system power dissipation.

D. ThreadX

ThreadX is Express Logic's advanced Real-Time Operating System (RTOS) designed specifically for deeply embedded applications. ThreadX has many advanced features, including its picokernel architecture, preemption-threshold, and a rich set of system services. ThreadX is implemented as a C library. Only the features used by the application are brought into the final image. The minimal footprint of ThreadX is as small as 2.5KB on CISC processors.

Why ThreadX? It supports a wide spectrum of processors. The complete ANSI C code is available and there are no royalties! ThreadX has a very small footprint (as low as 4KB), Unlimited Threads, Queues, Event Flags, Timers, Semaphores, Mutexes, Block Pools, and Byte Pools. Execution is fast with almost 1.7µs context switch @ 40MHZ.

III. OUR IMPLEMENTATION

A good working knowledge of the following resources/tools is required. The following tools would be used:

- Xilinx Platform Studio v8.1i
- XUP Virtex-II Pro Board (PowerPC core)
- Digilent Inc., Spartan-3 Rev E Board (Microblaze)
- ThreadX RTOS from Express Logic
- Windows Hyperterminal

A. Xilinx Platform Studio v8.1i

The 8.1i version of the Xilinx® Platform Studio tool suite is used for embedded processing design. This latest release incorporates a new graphical user interface that improves platform-based design by making common, tasks easy. Platform Studio 8.1i supports PowerPC and MicroBlaze processor designs for the Xilinx Virtex™-4, Virtex-II Pro, Spartan™-3 and Spartan-3E Platform FPGAs.

The Platform Studio suite is conveniently bundled with a processing IP library, software drivers, documentation, reference designs and the MicroBlaze soft processor IP core in the new 8.1i release of the Xilinx Embedded Development Kit (EDK). The Embedded Development Kit (EDK) bundle is an integrated software solution for designing embedded processing systems. This pre-configured kit includes the Platform Studio tool suite as well as all the documentation and IP that is required for designing Xilinx Platform FPGAs with embedded PowerPC™ hard processor cores and/or MicroBlaze™ soft processor cores. The Embedded Development Kit includes the following tools and IP:

Xilinx Platform Studio (XPS)

• Graphical and command line tools for developing and debugging the hardware and software platforms for an embedded application.
• Hardware platform that includes graphical and textual definition tools and generation of simulation and implementation netlists for use with the ISE logic design tools.
• Software platform definition that includes graphical and textual tools for matching it to the hardware platform, editing source code, running the compiler tool chains and library generation.

Software Development Tools

• GNU C/C++ compiler for MicroBlaze™ and PowerPC™
• GNU Debugger for MicroBlaze and PowerPC
• Other GNU utilities
• XMD – Xilinx Microprocessor Debug engine for MicroBlaze and PowerPC. It provides host-based target control using command line tools that enable complex regression testing.
• Data2MEM – a stand alone application for loading and updating on-chip memory content directly within the FPGA bitstream.
• Base System Builder – Wizard to streamline configuring hardware elements, processor options, bus system, IP options, and automatically generate memory map and design files
• Platform Studio SDK (Software Development Kit) – SW focused development and debug environment.

Board Support Packages (BSPs)

• Stand Alone BSP – For non-RTOS systems (MicroBlaze and PowerPC)
• Wind River VxWorks – For PowerPC Platform FPGAs
• MontaVista Linux – For PowerPC Platform FPGAs
• Support for Xilinx MicroKernel (XMK) Systems

Processor IP

• PowerPC and MicroBlaze infrastructure and peripheral IP cores and Microblaze soft processor core.

Base System Builder

The Base System Builder (BSB) automates several basic hardware and software platform configuration tasks common to most processor designs. If the target is one of the supported embedded processor development boards available from Xilinx, BSB picks the peripherals available on that board, and automatically matches the FPGA pinout to the board, and create a completed platform and test application that is ready to download and run on the board. There is also the option of designing a custom board, by using BSB to select and interconnect one of the available processor cores (MicroBlaze™ or PowerPC™, depending on the selected target FPGA device) with a variety of compatible, commonly used peripheral cores from the library. This gives us a hardware platform to use as a starting point from which we can add more processors and peripherals if needed, including custom peripherals, using the tools provided in XPS.

In all cases, BSB customizes following attributes of the system:

- Processor type (MicroBlaze or PowerPC, depending on the selected target FPGA device)
- Processor and bus clock frequency (BSB automatically infers and configures a Digital Clock Manager (DCM) primitive when needed)
- Standard processor buses (all peripherals are automatically connected via appropriate buses)
- Debug interface
- Cache configuration
- Memory size and type (both on-chip BRAM and controllers for off-chip memory devices)
- Common peripherals (such as general purpose I/O, UART, and timer)
- Interrupt sources (from among the applicable selected peripherals)

When targeting one of the supported embedded processor development boards, BSB narrows the choices of peripherals that control off-chip devices to those features provided on the specific board. Any deselected peripherals are omitted from the processor system design to minimize FPGA utilization.

Upon exit of BSB, a Hardware Specification (MHS) file is created and loaded into the XPS project. We can then further enhance the design in XPS or continue to implement the design using the Xilinx implementation tools.

Optionally, BSB can also create one or more software projects. Each project contains a sample application and linker script that can be compiled and run on the hardware on the target development board. XPS supports multiple software projects for every hardware system, each of which contains its own set of source files and linker script.

B. Digilent Inc. Spartan III (Microblaze)

Spartan-3 development board is a low-cost solution for evaluating the Xilinx Spartan-3 XC3S200 FPGA. The Spartan-3 Starter Board provides a powerful, self-contained development platform for designs targeting the new Spartan-3 FPGA from Xilinx. It features a 200K gate Spartan-3, on-board I/O devices, and 1MB fast asyncronous SRAM, making it the perfect platform to experiment with any new design, from a simple logic circuit to an embedded processor core. The board also contains a Platform Flash JTAG-programmable ROM, so designs can easily be made non-volatile. The Spartan-3 Starter Board is fully compatible with all versions of the Xilinx ISE tools. The major features of the board are:

- Xilinx Spartan-3 FPGA w/ twelve 18-bit multipliers, 216Kbits of block RAM, and up to 500MHz internal clock speeds
- On-board 2Mbit Platform Flash (XCF02S)
- 8 slide switches, 4 pushbuttons, 9 LEDs, and 4-digit seven-segment display
- Serial port, VGA port, and PS/2 mouse/keyboard port
- Three high-current voltage regulators (3.3V, 2.5V, and 1.2V)
- Works with JTAG3 programming cable, and P4 & MultiPRO cables from Xilinx
- 1Mbyte on-board 10ns SRAM (256Kb x 32)

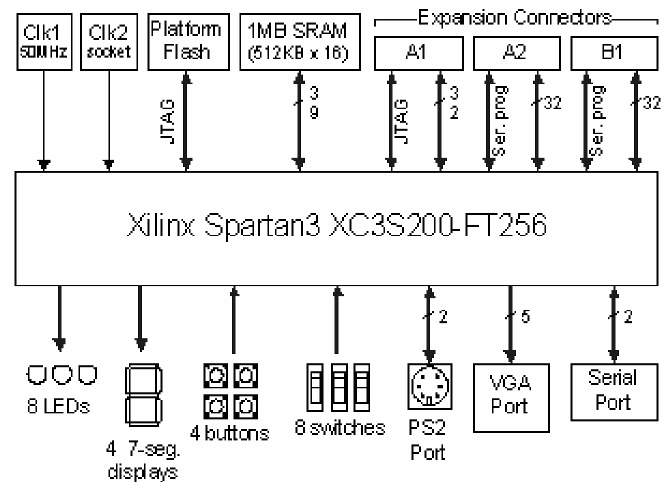

Fig. 1: Xilinx Spartan-3 platform architecture diagram

C. Digilent Inc.XUP Virtex II PRO Board (PowerPC)

The XUP Virtex-II Pro Development System provides an advanced hardware platform that consists of a high performance Virtex-II Pro Platform FPGA surrounded by a comprehensive collection of peripheral components that can be used to create a complex system to demonstrate the capability of the Virtex-II Pro Platform FPGA. Some features of the board:

- Virtex-2 Pro XC2VP30 FPGA with 30,816 Logic Cells, 136 18-bit multipliers, 2,448Kb of block RAM, and two PowerPC Processors
- DDR SDRAM DIMM that can accept up to 2Gbytes of RAM
- 10/100 Ethernet port
- USB2 port
- Compact Flash card slot
- XSGA Video port
- Audio Codec
- SATA, and PS/2, RS-232 ports

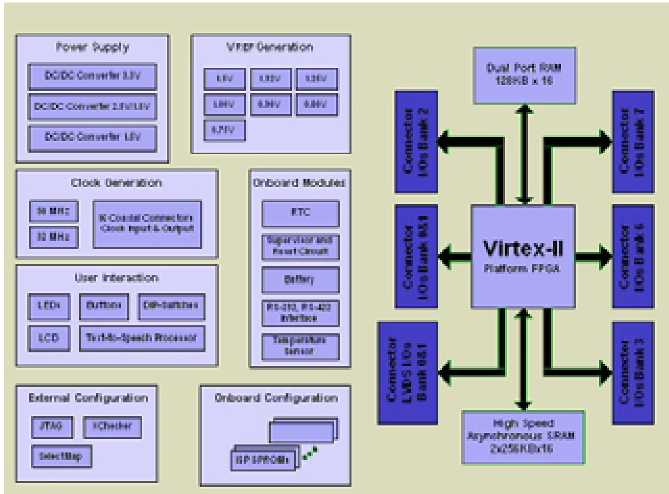

Fig. 2: Virtex II Pro board architecture diagram

D. *ExpressLogic ThreadX RTOS*

ThreadX is a high performance real time kernel designed specifically for embedded applications. It is highly scalable from small microcontroller applications to powerful RISC and DSP processors. ThreadX services are implemented as a C library, only those services actually used by the application are brought into the run-time image. Hence, the actual size of ThreadX is completely determined by the application. For most applications, the instruction image of ThreadX ranges between 2 KBytes and 15 KBytes in size.

Instead of layering kernel functions on top of each other like traditional *microkernel* architectures, ThreadX services plug directly into its core. This results in the fastest possible context switching and service call performance. This non-layering design is called *picokernel* architecture.

ThreadX is written primarily in ANSI C. A small amount of assembly language is needed to tailor the kernel to the underlying target processor. This design makes it possible to port ThreadX to a new processor family in a very short time.

Most distributions of ThreadX include the complete C source code as well as the processor-specific assembly language. This eliminates the “black-box” problems that occur with many commercial kernels. The source code also allows for application specific modifications. Although not recommended, it is certainly beneficial to have the ability to modify the kernel if it is absolutely required.

Because of its versatility, high-performance *picokernel* architecture, and great portability, ThreadX has the potential to become an industry standard for embedded applications.

Using ThreadX is easy. Basically, the application code must include ***tx_api.h*** during compilation and link with the ThreadX run-time library ***tx.lib***. There are four steps required to build a ThreadX application:

- Include the ***tx_api.h*** file in all application files that use ThreadX services or data structures.
- Create the standard C ***main*** function. This function must eventually call ***tx_kernel_enter*** to start ThreadX. Application-specific initialization that does not involve ThreadX may be added prior to entering the kernel.
- Create the ***tx_application_define*** function. This is where the initial system resources are created. Examples of system resources include threads, queues, memory pools, event flag groups, mutexes, and semaphores.
- Compile application source and link with the ThreadX run-time library ***tx.lib***. The resulting image can be downloaded to the target and executed.

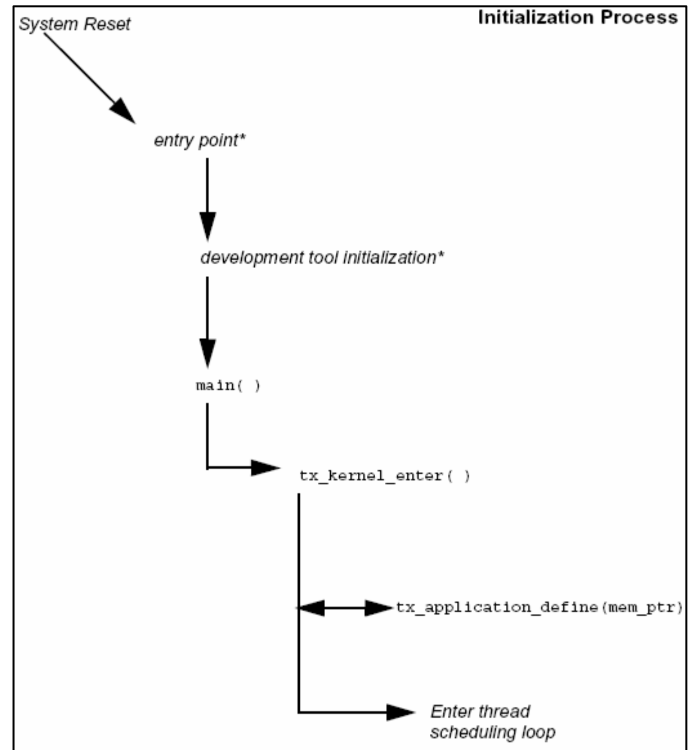

Fig. 3: Flow diagram for ThreadX initialization

ThreadX provides several API services such as Byte and Block Memory services, Event Flag, Interrupt, Message Queue, Semaphore and Mutex services, Thread Control and Timer Services.

Context Switch: When one thread is executing and an interrupt occurs, triggering the execution of a higher priority thread, the previously executing thread is interrupted, its context saved, and the processor is directed to start or resume execution of the instructions of the new thread. This context switch must be performed quickly, because real-time systems can require many context switches in a short period of time. On the front end of interrupt service routines, only the compiler’s scratch registers are saved initially. If it turns out that thread preemption is required, then the remaining registers in the set are also saved. ThreadX optimizes context switching on the ARM processor. Only those registers preserved across function calls are saved (registers v1-v4, fp, and lr). As a result, ThreadX performs context switch on the ARM processor in only about 100 cycles, which is one microsecond on a 100MHz processor.

Application Timers:

Application timers are maintained by *ThreadX* to provide "count-down" services to provide a variety of time-related duties. Timers can be set up to operate once (an one-shot timer), or for recurring operation (a periodic timer). When a timer expires, it generates an interrupt and a timer function (similar to an Interrupt Service Routine or "ISR") is executed. The timer function can affect the processing or even the scheduling of application threads, depending on which threads have been set up to execute when this timer expires.

E. Windows Hyperterminal

The RS232 Serial port was used for communication between the Development boards and the host PC.

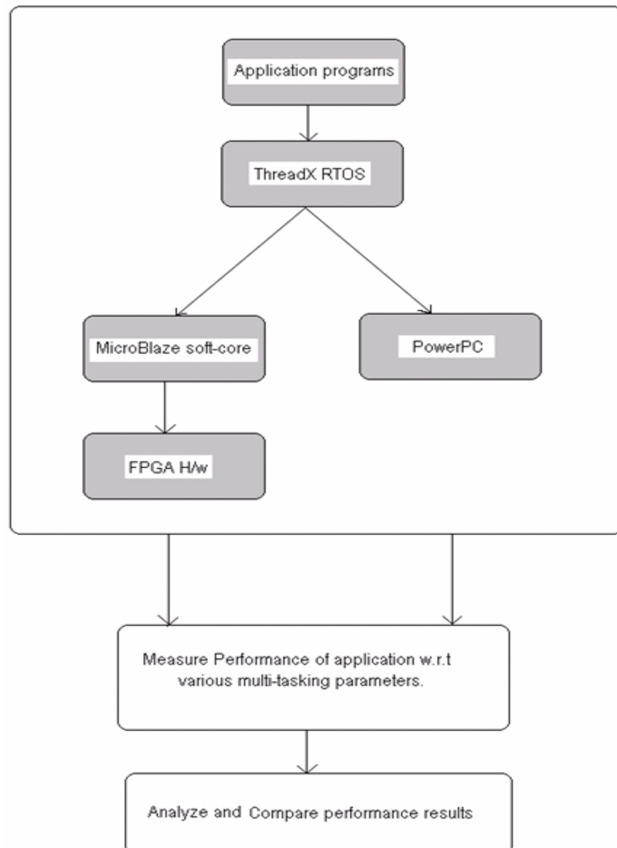

Fig. 4: Block diagram of Project plan

IV. PROCEDURE

Here we outline the procedure for creating and running the ThreadX multitasking application on the development boards.

The major steps involved for Spartan-3 board are: (this list is in support of our custom test application(s) and is not exhaustive. For a complete guide please refer to the Xilinx guides)

1. Connect the Parallel Cable between your host computer and the Spartan-3 Starter Board.
2. Connect the serial cable between your host computer and the Spartan-3 Starter Board.
3. Apply power to the Spartan-3 Starter Board.
4. Start a hyperterminal (or similar) session on your host computer with the following settings:

- Start a hyperterminal (or similar) session on your host computer with the following settings:
- Select the COM port corresponding to connected serial port on your host computer
- Baud Rate = 9600
- Data = 8 bits
- Parity = none
- Stop = 1 bit
- Flow control = none

5. Invoke Xilinx Platform Studio (XPS) and choose the Base System Builder (BSB) Wizard.

Creating the Hardware Platform with the BSB:

6. In BSB the following configuration is chosen for the Xilinx Spartan-3 board processor apart from the default settings: Under Local Memory, for Data and Instruction, select 16 KB.
7. In BSB Configure I/O Interfaces: Uncheck LEDs, 7 segment LEDs and Check RS232 and SRAM with default settings.
8. In BSB Add Internal Peripherals: Add an OPB Timer Peripheral and choose a Count Bit Width of 32 and a single timer mode with interrupts.
9. All the other options in BSB are chosen with default values.

Preparing the Software Application:

1. Source code for the Matrix Multiplication program is provided in Appendix A.
2. Make sure the tx.a, tx_api.h, and tx_port.h files are in project folder.
3. After adding the test project given in the appendix, set the compiler options as follows:
4. Select the default linker script option.
5. Give the "> mapfile.map -save-temps" option for compiler options to be appended.
6. Type -M <location/foldername>/tx.a in the Linker –WI option.

The major steps for XUP Virtex-II Pro board:

Base System Builder Settings:

1. In BSB Configure I/Os: Uncheck DDRSDRAM, EEPROM, LEDs and all remaining peripherals except RS 232.
2. In BSB Add Internal Peripherals: For PLB BRAM IF CNTLR, select 64KB Memory size. Also add OPB BRAM IF CNTRL and OPB Timer and choose one timer with interrupts.

All other options are default.

Prepare the Software Application in the same way as above. Source Code for the Producer-Consumer Application is provided in Appendix B.

V. TEST APPLICATIONS

Two multitasking applications were designed using ThreadX RTOS for execution on Microblaze and PowerPC cores.

1. Producer-Consumer Application

One Producer and Two Consumers were modeled with 3 threads. Two ThreadX Semaphores and one Mutex were used.

2. Matrix Multiplication
Two Threads were created and each of them was performing 100 runs of 6x6 matrix multiplication.

Both the programs were run successfully on each of the cores and development boards. The execution time for the matrix multiplication and the context switching time for the threads was noted.

VI. EXPERIMENTAL RESULTS

Performance Analysis Methods

Types of Performance analysis methods
- Intrusive
- Non-intrusive

Non-intrusive is 'ideal' but requires use of special H/w like logic analyzers.

Intrusive is easy to implement with a small tradeoff. Usually a small piece of code is placed in the application which measures the performance metrics such as execution speed, time etc. But this code itself will have some overhead associated with it.

TABLE 1: A COMPARISON OF MICROBLAZE AND POWERPC CORES

PowerPC

Clock Frequency	Dhrystone MIPS	DMIPS/MHZ
100 MHz	135	1.35
200 MHz	271	1.35
300 MHz	407	1.35
400 MHz	542	1.35

Microblaze

Clock Frequency	Dhrystone MIPS	DMIPS/MHZ
100 MHz	92	0.92
150 MHz	138	0.92
180 MHz	166	0.92

We employed an intrusive method of Performance analysis to determine the performance of the ThreadX RTOS on the cores. The context switch times were obtained by finding the difference in timer counts (ticks) at the end of a thread and the beginning of the next thread.

For Matrix Multiplication program:

✓ **Context Switch time**
PowerPC: 9 µs @ 100 MHz
Microblaze: 16 µs @ 50 MHz

✓ **Execution time for 100 runs**
PowerPC: 7.2 ms @ 100 MHz
Microblaze: 16.07 ms @ 50 MHz

The context switch times obtained above appear higher than the expected values or the times given in the data sheets. This might be because of the intrusive method used. The tx_time_get() function of ThreadX which returns the timer ticks elapsed, itself has some amount of overhead associated.

VII. CONCLUSIONS

- ThreadX allows the designer to handle multiple threads and inter-thread communication at a higher level.
- ThreadX resources used in our project are timers, threads, mutex and semaphores.
- Microblaze soft processor gives us the independence to choose the configuration and peripheral we like.
- PowerPC gives better performance when compared with Microblaze in terms of lower switching times and faster execution.
- Xilinx's XPS 8.1i is an easy to use tool to integrate hardware description with software applications.

VIII. FUTURE WORK

Multiple Microblaze softcores can be implemented on a single FPGA and Microblaze Debug Module allows debugging of 8 microblaze processors at a time. The XUP Virtex-II Pro Platform has two PowerPC cores embedded in the FPGA. The combination of multiple processor cores integrated with co-processing capability enables a wide range of performance optimizing options for parallel processing applications. Investigation of the performance of the RTOS with multiple processors could be carried out. Comparison of ThreadX with other commercially available RTOS' such as: VXWorks, Nucleus, uC/OS-II, uCLinux. Comparison of IDEs (Compilers and Debuggers) such as Nucleus Debugger (for both MB and PPC), E9524A Inverse Assembler (for MB) etc.

REFERENCES

[1] Jie Liu, Edward A Lee, 2003, Multitasking for Real time Embedded systems. IEEE Control Systems Magazine

[2] Sung I. Park, Vijay Ranganathan, Mani B. Srivastava Energy Efficiency and Fairness Tradeoffs in Multi-Resource, Multi-Tasking Embedded Systems, ISLPED'03, Korea

[3] Lamie, W., ThreadX® Performance Analysis, Express Logic, Inc., Whitepaper.

[4] Szewinski J., Kaleta, P., Fafara, P., Pucyk, P., Koprek, W., Pozniak, K., Romaniuk, R., Software for development and communication with FPGA based hardware. Institute of Electronic Systems, Warsaw University of Technology, Poland.

[5] Dr. Ed Lamie, Express Logic Real-Time Embedded Multithreading: Using ARM Cores and the ThreadX RTOS

[6] Xilinx University Program, EDK Base System Builder (BSB) Support for XUPV2P Board.

[7] ThreadX User Guide by Express Logic

[8] ThreadX Data Sheets for Microblaze and PowerPC 405

[9] ThreadX Demos for Spartan-3 and Virtex-II Pro by Express Logic.

[10] M. Young, The Techincal Writers Handbook. Mill Valley, CA: University Science, 1989.

[11] Spartan-3 MB and Virtex-II Pro Demos by Xilinx Inc.

[12] www.xilinx.com

[13] www.digilentinc.com/xupv2p

[14] www.rtos.com

IX. APPENDIX A

Source Code for Matrix Multiplication program using ThreadX on Microblaze:

```
// Located in: microblaze_0/include/xparameters.h
#include "xparameters.h"
#include "stdio.h"
#include "xutil.h"

#include "xgpio_l.h" /* general-purpose I/O peripheral control functions */
#include "xtmrctr_l.h" /* timer/counter peripheral control functions */
#include "xuartlite_l.h" /* uartlite peripheral control functions */
#include "xintc_l.h" /* interrupt controller peripheral control functions */

/* End of MicroBlaze Specific Includes.  */

#include    "tx_api.h"

#define    TX_DISABLE_ERROR_CHECKING
#define    DEMO_STACK_SIZE    1024
#define    MATRIX_SIZE        6
#define    NO_RUNS                    100

TX_THREAD              thread_main;
TX_THREAD              thread_main1;

void    thread_main_entry(ULONG thread_input);
void    thread_main1_entry(ULONG thread_input);
void    convert_to_ascill(ULONG value, CHAR *buffer_ptr);

/* Global variables */
unsigned int timer_count = 500;//16777216; /* initial timer period in OPB cycles ~= 0.3 sec */
ULONG time1, time2;
char buffer[10];

/* Timer interrupt service routine */
/* Note: This ISR was registered statically in the Software Platform Settings dialog */
void timer_int_handler(void * baseaddr_p) {
  unsigned int csr;
  /* Read timer 0 CSR to see if it requested the interrupt */
  csr =
XTmrCtr_mGetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 0);
  if (csr & XTC_CSR_INT_OCCURED_MASK) {

       _tx_timer_interrupt();
  }
  /* Clear the timer interrupt */
  XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 0, csr);
}

/* Interrupt test routine */
void InterruptTest(void)
{
  /* Start the interrupt controller */
  XIntc_mMasterEnable(XPAR_OPB_INTC_0_BASEADDR);

  /* Set the number of cycles the timer counts before interrupting */
  XTmrCtr_mSetLoadReg(XPAR_OPB_TIMER_1_BASEADDR, 0, timer_count);

  /* Reset the timers, and clear interrupts */
  XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 0,
    XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );

  /* Enable timer and UART interrupt requests in the interrupt controller */
  XIntc_mEnableIntr(XPAR_OPB_INTC_0_BASEADDR,
    XPAR_OPB_TIMER_1_INTERRUPT_MASK                    /*|
XPAR_RS232_INTERRUPT_MASK*/);

  /* Start the timers */
  XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 0,
    XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK
|
    XTC_CSR_AUTO_RELOAD_MASK                             |
XTC_CSR_DOWN_COUNT_MASK);
}

/* End user-supplied interrupt test routine */

int main (void)
{
  //print("-- Entering main() --\r\n");

  /* Enter the ThreadX kernel.  */

  tx_kernel_enter();

  //print("-- Exiting main() --\r\n");
  return 0;
}

void    tx_application_define(void *first_unused_memory)
{
   CHAR    *pointer;

   /* Put system definition stuff in here, e.g. thread creates and other assorted
      create information.  */

   /* Setup pointer.  */
   pointer =  (CHAR *) first_unused_memory;

   /* Create the main thread.  */
   tx_thread_create(&thread_main, "thread main", thread_main_entry, 0,
         pointer, DEMO_STACK_SIZE,
         2, 2, 0, TX_AUTO_START);
   pointer =  pointer + DEMO_STACK_SIZE;

     tx_thread_create(&thread_main1, "thread main1", thread_main1_entry, 0,
         pointer, DEMO_STACK_SIZE,
         2, 2, 0, TX_AUTO_START);
   pointer =  pointer + DEMO_STACK_SIZE;

    /* Initialize MicroBlaze Timer.  */
    InterruptTest();
}

/* Define the test threads. */
void thread_main_entry(ULONG thread_input)
{
  int                                    a[MATRIX_SIZE][MATRIX_SIZE],
b[MATRIX_SIZE][MATRIX_SIZE], c[MATRIX_SIZE][MATRIX_SIZE];
  int i, j, k, m;
    ULONG sttime, endtime;

  sttime = tx_time_get();

    for (i=0; i < MATRIX_SIZE; i++)
    {
     for (j=0; j < MATRIX_SIZE; j++)
     {
       a[i][j] = i;
         b[i][j] = i+j;
         c[i][j] = 0;
     }
    }

  for (m=0; m < NO_RUNS; m++)
    {
     for (i=0; i < MATRIX_SIZE; i++)
```

```
        {
          for (j=0; j < MATRIX_SIZE; j++)
          {
              for (k=0; k < MATRIX_SIZE; k++)
              {
                  c[i][j] += a[i][k]*b[k][j];
              }
          }
        }
      }

      //tx_thread_sleep(3);
      endtime = tx_time_get();
      endtime = endtime - sttime;
      convert_to_ascill(endtime,buffer);
      //print("Total ticks to run matmul : ");
      print(buffer);
      //time1 = tx_time_get();
}

void thread_main1_entry(ULONG thread_input)
{
   //time2 = tx_time_get();
      //time2 = time2-time1;
      //convert_to_ascill((time2-time1),buffer);
      //print(buffer);

      int a[MATRIX_SIZE][MATRIX_SIZE],
b[MATRIX_SIZE][MATRIX_SIZE], c[MATRIX_SIZE][MATRIX_SIZE];
   int i, j, k, m;
      char buffer[10];
      ULONG sttime, endtime;

   sttime = tx_time_get();

      for (i=0; i < MATRIX_SIZE; i++)
      {
        for (j=0; j < MATRIX_SIZE; j++)
        {
           a[i][j] = i;
             b[i][j] = i+j;
             c[i][j] = 0;
        }
      }

   for (m=0; m < NO_RUNS; m++)
      {
        for (i=0; i < MATRIX_SIZE; i++)
        {
          for (j=0; j < MATRIX_SIZE; j++)
          {
              for (k=0; k < MATRIX_SIZE; k++)
              {
                  c[i][j] += a[i][k]*b[k][j];
              }
          }
        }
      }

      //tx_thread_sleep(3);
      endtime = tx_time_get();
      endtime = endtime - sttime;
      convert_to_ascill(endtime,buffer);
      //print("Total ticks to run matmul : ");
      print(buffer);

}


void    convert_to_ascill(ULONG value, CHAR *buffer_ptr)
{
    ULONG   temp;
    UINT    i = 1;

    /* Fill with spaces.  */
    for (i = 0; i < 16; i++)
        buffer_ptr[i] = ' ';
    buffer_ptr[16] = 0; /* NULL */

    /* Calculate the number of places.  */
    i = 1;
    temp = value;
    while (temp)
    {

        temp = temp/10;
        if (temp)
            i++;
    }

    /* Load string with number.  */
    i--;
    temp = value;
    do
    {
        buffer_ptr[i] =  (CHAR) ((temp % 10) + 0x30);
        temp = temp / 10;
        if (i)
            i--;
    } while (temp);
}
```

X. APPENDIX B

Source Code for the Producer Consumer ThreadX Application on PowerPC:

Producer-Consumer

```
// Located in: ppc405_0/include/xparameters.h
#include "xparameters.h"
#include "stdio.h"
#include "xutil.h"

/* PPC Specific Includes.  */

#include    "xgpio_l.h"
#include    "xparameters.h"
#include   "tx_api.h"

#define TX_DISABLE_ERROR_CHECKING
#define     DEMO_STACK_SIZE         1024

/* Define the ThreadX object control blocks...  */

TX_THREAD              thread_producer;
TX_THREAD              thread_consumer1;
TX_THREAD              thread_consumer2;
TX_SEMAPHORE           sem_flag1;
TX_SEMAPHORE           sem_flag2;
TX_MUTEX               mutex;

void    thread_producer_entry(ULONG thread_input);
void    thread_consumer1_entry(ULONG thread_input);
void    thread_consumer2_entry(ULONG thread_input);
void    convert_to_ascill(ULONG value, CHAR *buffer_ptr);

int buffer, num;

/* Begin user-supplied interrupt test routine for PPC_ML310_Tutorial_8_1 */

/* This example demonstrates how to use an interrupt controller
 * that responds to interrupts from two peripherals (UART and OPB_timer)
 * in a PowerPC based system.
 * This interrupt test routine has been added to the test application
(TestApp_Memory)
 * generated by the Base System Builder.
 */

#include "xgpio_l.h" /* general-purpose I/O peripheral control functions */
#include "xtmrctr_l.h" /* timer/counter peripheral control functions */
#include "xuartlite_l.h" /* uartlite peripheral control functions */
#include "xintc_l.h" /* interrupt controller peripheral control functions */
#include "xexception_l.h" /* PPC exception handler control functions */

/* Global variables */
unsigned int timer_count = 33554432; /* initial timer period in OPB cycles ~=
0.3 sec */
volatile unsigned int exit_command = 0; /* flag from UART ISR to exit
InterruptTest routine */

/* Timer interrupt service routine */
/* Note: This ISR was registered statically in the Software Platform Settings
dialog */
void timer_int_handler(void * baseaddr_p) {
  unsigned int csr;
  /* Read timer 0 CSR to see if it requested the interrupt */
  csr                                                                     =
XTmrCtr_mGetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 0);
  if (csr & XTC_CSR_INT_OCCURED_MASK) {

XTmrCtr_mEnableIntr(XPAR_OPB_TIMER_1_BASEADDR, 0);
  _tx_timer_interrupt();
        /* Clear the timer interrupt */

  }
 }

/* Interrupt test routine */
void InterruptTest(void) {
  print("-- Entering InterruptTest() --\r\n");

  XIntc_mMasterEnable(XPAR_OPB_INTC_0_BASEADDR);
    /* Set the number of cycles the timer counts before interrupting */
  XTmrCtr_mSetLoadReg(XPAR_OPB_TIMER_1_BASEADDR,              0,
timer_count);
  /* Reset the timers, and clear interrupts */
  XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 0,
    XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );
  /* Enable timer and uart interrupt requests in the interrupt controller */
  XIntc_mEnableIntr(XPAR_OPB_INTC_0_BASEADDR,
    XPAR_OPB_TIMER_1_INTERRUPT_MASK                              /*|
XPAR_RS232_UART_1_INTERRUPT_MASK*/);
  /* Start the timers */
  XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 0,
    XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK
|
    XTC_CSR_AUTO_RELOAD_MASK                                       |
XTC_CSR_EXT_GENERATE_MASK/*XTC_CSR_DOWN_COUNT_MAS
K*/);
  /* Enable PPC non-critical interrupts */
  XExc_mEnableExceptions(XEXC_NON_CRITICAL);


  /* Disable PPC non-critical interrupts */
  XExc_mDisableExceptions(XEXC_NON_CRITICAL);
  print("-- Exiting InterruptTest() --\r\n");
}

/* End user-supplied interrupt test routine */

//====================================================

int main (void) {

  print("-- Entering main() --\r\n");
  /*
   * MemoryTest routine will not be run for the memory at
   * 0xffff0000 (plb_bram_if_cntlr_1)
   * because it is being used to hold a part of this application program
   */

  /* Testing BRAM Memory (opb_bram_if_cntlr_1)*/
  {
    XStatus status;

    print("Starting MemoryTest for opb_bram_if_cntlr_1:\r\n");
    print("  Running 32-bit test...");
    status                                                              =
XUtil_MemoryTest32((Xuint32*)XPAR_OPB_BRAM_IF_CNTLR_1_BASE
ADDR, 512, 0xAAAA5555, XUT_ALLMEMTESTS);
    if (status == XST_SUCCESS) {
      print("PASSED!\r\n");
    }
    else {
      print("FAILED!\r\n");
    }
    print("  Running 16-bit test...");
    status                                                              =
XUtil_MemoryTest16((Xuint16*)XPAR_OPB_BRAM_IF_CNTLR_1_BASE
ADDR, 1024, 0xAA55, XUT_ALLMEMTESTS);
    if (status == XST_SUCCESS) {
      print("PASSED!\r\n");
    }
    else {
      print("FAILED!\r\n");
```

```
    }
    print("  Running 8-bit test...");
    status =
XUtil_MemoryTest8((Xuint8*)XPAR_OPB_BRAM_IF_CNTLR_1_BASEADDR, 2048, 0xA5, XUT_ALLMEMTESTS);
    if (status == XST_SUCCESS) {
      print("PASSED!\r\n");
    }
    else {
      print("FAILED!\r\n");
    }
  }
  InterruptTest();
  /* Enter the ThreadX kernel.  */
   tx_kernel_enter();
  /* Run user-supplied interrupt test routine */


  print("-- Exiting main() --\r\n");
  return 0;
}

void    tx_application_define(void *first_unused_memory)
{
   CHAR    *pointer;

   /* Put system definition stuff in here, e.g. thread creates and other assorted
      create information.  */

   /* Setup pointer.  */
   pointer =  (CHAR *) first_unused_memory;

   /* Create the main thread.  */
   tx_thread_create(&thread_producer, "thread producer",
thread_producer_entry, 0,
         pointer, DEMO_STACK_SIZE,
         2, 2, 1, TX_AUTO_START);
   pointer =  pointer + DEMO_STACK_SIZE;

   tx_thread_create(&thread_consumer1, "thread consumer1",
thread_consumer1_entry, 1,
         pointer, DEMO_STACK_SIZE,
         2, 2, 5, TX_AUTO_START);
   pointer =  pointer + DEMO_STACK_SIZE;

   tx_thread_create(&thread_consumer2, "thread consumer2",
thread_consumer2_entry, 2,
         pointer, DEMO_STACK_SIZE,
         2, 2, 5, TX_AUTO_START);
   pointer =  pointer + DEMO_STACK_SIZE;

   /* Create a semaphore */
   tx_semaphore_create(&sem_flag1, "semaphore flag1", 1);

   /* Create a semaphore */
   tx_semaphore_create(&sem_flag2, "semaphore flag2", 1);

   /* Create the mutex used to access the critical region.  */
   tx_mutex_create(&mutex, "mutex 0", TX_NO_INHERIT);

     /* Initialize MicroBlaze Timer.  */
     InterruptTest();
}

/* Define the test threads.  */
void thread_producer_entry(ULONG thread_input)
{
  char tempstr[10];
  int num=0;
  UINT status;
  ULONG ptime;

  while(1)                    // loop forever
  {
        num++;

     status = tx_semaphore_get(&sem_flag2, TX_WAIT_FOREVER);

        /* Get the mutex with suspension.  */
     status =   tx_mutex_get(&mutex,  TX_WAIT_FOREVER);     //  enter
critical section
        convert_to_ascill(num, tempstr);
        print(tempstr);
        print(" In Producer thread...\r\n");
       buffer=num;       // create a new number to put in the buffer

      /* Release the semaphore flag */
     status =  tx_semaphore_put(&sem_flag1);
        print("** Producer releasing semaphore\r\n");

        /* Release the mutex.  */
     status =  tx_mutex_put(&mutex);  // leave critical section
        //ptime = tx_time_get();
        //convert_to_ascill(ptime, tempstr);
        //print(tempstr);

      tx_thread_sleep(10);
  }
}

void thread_consumer1_entry(ULONG thread_input)
{
  UINT    status;
  char tempstr[5];
  int i=0;

  while(1)                    // loop forever
  {
     status =  tx_semaphore_get(&sem_flag1, TX_WAIT_FOREVER);

     /* Get the mutex with suspension.  */
     status =   tx_mutex_get(&mutex,  TX_WAIT_FOREVER);     //  enter
critical section
        convert_to_ascill(buffer,tempstr);

        print(tempstr);

        if (buffer == num+1)
        {
          print("  In Consumer 1...\r\n");
          num = buffer;
        }

        /* Release the mutex.  */
     status =  tx_mutex_put(&mutex);  // leave critical section

        status = tx_semaphore_put(&sem_flag2);
        print("** Consumer 1 releasing semaphore\r\n");
        tx_thread_sleep(3);
  }
}

void thread_consumer2_entry(ULONG thread_input)
{
  UINT    status;
  char tempstr[5];
  int i=0;

  while(1)                    // loop forever
  {
     status =  tx_semaphore_get(&sem_flag1, TX_WAIT_FOREVER);

     /* Get the mutex with suspension.  */
     status =   tx_mutex_get(&mutex,  TX_WAIT_FOREVER);     //  enter
critical section
        convert_to_ascill(buffer,tempstr);
```

```
            print(tempstr);

            if (buffer == num+1)
            {
              print("  In Consumer 2...\r\n");
              num = buffer;
            }

            /* Release the mutex.  */
        status =  tx_mutex_put(&mutex);  // leave critical section
            status = tx_semaphore_put(&sem_flag2);
            print("** Consumer 2 releasing semaphore\r\n");

            tx_thread_sleep(4);
  }
}

void    convert_to_ascill(ULONG value, CHAR *buffer_ptr)
{
   ULONG   temp;
   UINT    i = 1;

   /* Fill with spaces.  */
   for (i = 0; i < 16; i++)
      buffer_ptr[i] = ' ';
   buffer_ptr[16] = 0; /* NULL */

   /* Calculate the number of places.  */
   i = 1;
   temp = value;
   while (temp)
   {

      temp = temp/10;
      if (temp)
         i++;
   }

   /* Load string with number.  */
   i--;
   temp = value;
   do
   {
      buffer_ptr[i] =  (CHAR) ((temp % 10) + 0x30);
      temp = temp / 10;
      if (i)
         i--;
   } while (temp);
}
```