Albahit Journal of Applied Sciences
"open-access, peer-reviewed biannual journal"
Volume 5, Issue 1, 2025
Page No: 82-93
***Website**: https://albahitjas.com.ly/index.php/albahit/en/index*

Design and implementation of a High efficiency UART model using VHDL and PicoBlaze on Xilinx FPGA platforms

Hamed F. Alsalhin Saleh [1]*, Fathalla I. Solman [2], Zead Hamad Abdulkarim [3]*
[1,2,3] Higher Technical Educations, Department of Electrical Engineering,College of Engineering Technologies – Alqubah-Libya

**Corresponding author:* hlibya60@gmail.com

Received: March 16, 2025	Accepted: June 12, 2025	Published: July 18, 2025

Abstract:
Digital design using the Hardware Description Language (VHDL) is a fundamental approach in developing embedded systems, as it enables the construction of highly efficient circuits that can be implemented on FPGA platforms. The importance of this research lies in exploring the integration between the PicoBlaze processor and Xilinx software for designing a high-efficiency UART unit, thereby enhancing data transmission performance while minimizing resource utilization in modern digital systems.
The research problem stems from the limited applied studies that address the effective integration of the PicoBlaze architecture with VHDL-based UART units, particularly in terms of achieving efficient data processing and transmission with low resource consumption. This gap highlights the need for a practical model that demonstrates the feasibility of such integration.
In this study, the methodology involved designing a UART unit using VHDL and integrating it with the PicoBlaze processor through the Xilinx ISE environment. The main objective was to perform an addition operation on the digits of the student's identification number (SID) and transmit the result via UART in hexadecimal format, thereby validating the efficiency of hardware–software co-design.
The results confirmed the successful implementation, as the seven digits were correctly summed and the output transmitted through UART. Simulation outcomes further demonstrated functional accuracy and efficient resource utilization. The study concludes that combining PicoBlaze with VHDL provides a high-efficiency framework for signal processing and embedded system control, with potential for future extensions such as multi-channel support and flow control enhancements.

Keywords: VHDL, PicoBlaze, UART, Embedded Systems, Xilinx, FPGA.

تصميم وتنفيذ وحدة UART عالية الكفاءة باستخدام VHDL ومعالج PicoBlaze في بيئة Xilinx

حامد فضل الله الصالحين صالح [1]*، فتح الله ابراهيم سليمان [2]، زياد حمد عبد الكريم [3]*
[1،2،3] التعليم العالي التقني، قسم الهندسة الكهربائية، كلية التقنيات الهندسية، القبة، ليبيا

الملخص
يمثل التصميم الرقمي باستخدام لغة وصف العتاد (VHDL) أحد الأساليب الجوهرية في تطوير الأنظمة المدمجة، حيث يتيح بناء دوائر عالية الكفاءة قابلة للتنفيذ على منصات FPGA. وتكمن أهمية هذا البحث في استعراض إمكانات الدمج بين معالج PicoBlaze وبرمجيات Xilinx لتصميم وحدة UART عالية الكفاءة، بما يساهم في تحسين كفاءة نقل البيانات وتقليل استهلاك الموارد داخل الأنظمة الرقمية الحديثة.
تكمن المشكلة البحثية في محدودية الدراسات التطبيقية التي تستعرض تكاملاً فعالًا بين معمارية PicoBlaze ووحدات UARTالمصممة بلغةVHDL ، خاصة فيما يتعلق بمعالجة الإشارات ونقل البيانات بكفاءة عالية مع الحفاظ على استهلاك منخفض للموارد. ومن هنا تأتي الحاجة إلى تقديم نموذج تطبيقي يبرهن على جدوى هذا الدمج.
في هذا البحث، تم اعتماد منهجية تقوم على تصميم وحدة UART باستخدام VHDL ودمجها مع معالج PicoBlaze من خلال بيئة Xilinx ISE. شملت الخطوات الرئيسة كتابة الكود، المحاكاة، ثم التوليف والتنفيذ على شريحة FPGA الهدف.

الرئيس كان إجراء عملية جمع للأرقام المكوّنة لرقم هوية الطالب (SID) وإرسال الناتج عبر UART بصيغة سداسية عشرية، بما يختبر كفاءة التكامل بين البرمجيات والأجهزة.
أظهرت النتائج نجاح عملية التصميم والتنفيذ، حيث تم جمع الأرقام السبعة بشكل صحيح وإرسال الناتج عبر UART، كما أثبتت المحاكاة صحة الأداء وكفاءة استهلاك الموارد. خلص البحث إلى أن الدمج بين PicoBlaze وVHDL يوفر بنية عالية الكفاءة لمعالجة الإشارات والتحكم في الأنظمة المدمجة، مع فتح آفاق لتطوير وظائف متقدمة كالدعم المتعدد القنوات والتحكم في التدفق.

الكلمات المفتاحية: VHDL، PicoBlaze، UART، الأنظمة المدمجة، Xilinx، FPGA

Introduction

VHDL has a rich and interesting history[1]. But since knowing this history is probably not going to help you write better VHDL code, it will only be briefly mentioned here. Consulting other, lengthier texts or search engines will provide more information for those who are interested. Regarding the VHDL acronym, the V is short for yet another acronym: VHSIC or Very High-Speed Integrated Circuit. The HDL stands for Hardware Description Language. Clearly, the state of technical airs these days has done away with the need for nested acronyms. VHDL is a true computer language with the accompanying set of syntax and usage rules.[2] But, as opposed to higher-level computer languages, VHDL is primarily used to describe hardware. The tendency for most people familiar with a higher-level computer language such as C or Java is to view VHDL as just another computer language. This is not altogether bad approach if such a view facilitates the understanding and memorization of the language syntax and a structure. The common mistake made by someone with this approach is to attempt to program in VHDL as they would program a higher-level computer language. Higher- level computer languages are sequential in nature; VHDL is not [3].

VHDL was invented to describe hardware and in fact VHDL is a concurrent language. What this means is that, normally, VHDL instructions are all executed at the same time (concurrently), regardless of the size of your implementation. Another way of looking at this is that higher-level computer languages are used to describe algorithms (sequential execution) and VHDL is used to describe hardware (parallel execution). This inherent difference should necessarily encourage you to re-think how you write your VHDL code. Attempts to write VHDL code with a high-level language style generally result in VHDL code that no one understands. Moreover, the tools used to synthesize2 this type of code have a tendency to generate circuits that generally do not work correctly and have bugs that are nearly impossible to trace. And if the circuit does actually work, it will most likely be inefficient due to the fact that the resulting hardware was unnecessarily large and overly complex. This problem is compounded as the size and complexity of your circuits becomes greater.[4]

There are two primary purposes for hardware description languages such as VHDL. First, VHDL can be used to model digital circuits and systems. Although the word \model" is one of those overly used words in engineering, in this context it simply refers to a description of something that presents a certain level of detail. The nice thing about VHDL is that the level of detail is unambiguous due to the rich syntax rules associated with it. In other words, VHDL provides everything that is necessary in order to describe any digital circuit. Likewise, a digital circuit/system is any circuit that processes or stores digital information. Second, having some type of circuit model allows for the subsequent simulation and/or testing of the circuit. The VHDL model can also be translated into a form that can be used to generate actual working circuits.[5]

The VHDL model is magically3 interpreted by software tools in such a way as to create actual digital circuits in a process known as synthesis. There are other logic languages available to model the behavior of digital circuit designs that are easy to use because they provide a graphical method to model circuits. For them, the tendency is to prefer the graphical approach because it has such a comfortable learning curve. But, as you can easily imagine, your growing knowledge of digital concepts is accompanied by the ever-increasing complexity of digital circuits you are dealing with. The act of graphically connecting a bunch of lines on the computer screen quickly becomes tedious. The more intelligent approach to digital circuit design is to start with a system that is able to describe exactly how your digital circuit works (in other words, modeling it) without having to worry about the details of connecting massive quantities of signal lines. Having a working knowledge of VHDL will provide you with the tools to model digital circuits in a much more intelligent manner[6].

Finally, you will be able to use your VHDL code to create actual functioning circuits. This allows you to implement relatively complex circuits in a relatively short period of time. The design methodology you will be using allows you to dedicate more time to designing your circuits and less time \constructing" them. The days of placing, wiring and troubleshooting multiple integrated circuits on a proto-board are gone. VHDL is a very exciting language that can allow the design and implementation of functions capable of processing an enormous amount of data by employing a relatively low-cost and low-power hardware. Moreover, what is really impressive is that, via simple VHDL modules, you can have direct access to basic ns-level logic events as well as communicate using a USB port or drive a VGA monitor to visualize graphics of modest complexity. Modeling digital circuits with VHDL is a form of modern digital design distinct.[7].

Tools Needed for VHDL Development

VHDL is a programming language used to implement hardware which will run other software (for example C). A Field Programmable Gate Array (FPGA) is probably the most common device that you can use for your VHDL implementations. If you want to do VHDL coding for FPGAs you will have to play within the rules that current major FPGA manufacturers have drawn up to help you (rules which also ensure their continued existence in the market). The successful implementation of a VHDL-based system roughly calls for the following steps: VHDL code writing, compiling, simulation and synthesis. All major FPGA manufacturers have a set of software and hardware tools that you can use to per-form the mentioned steps. Most of these software tools are free of charge but are not open-source. Nevertheless, the same tools follow a license scheme, whereby paying a certain amount of money allows you to take advantage of sophisticated software features or get your hands on proprietary libraries with lots of components (e.g. a 32-bit processor) that you can easily include in your own project.

Background

KCPSM3 is a very simple 8-bit microcontroller primarily for the Spartan -3 device but also suitable for use in Virttex –II and Virttex-IIPRO devices. Although it could be used for processing of data, it is most likely to be employed in applications requiring a complex ,but non-time critical state machine .Hence it has the name of '(K) constant coded Programmable state machine'.

This revised version of popular KCPSM macro has still been developed with one dominant factor being held above all others –Size ! The result is a microcontroller which occupies just 96 spartant -3 slices which is just 5% of XC3S200 device and less than 0.3 % of the XC3S5000 device .Together with small amount of logic ,a single block RAM is used to form a ROM store for a program of up to 1024 instructions .Even with such size constructions ,the performance is respectable at approximately 43 to 66 MIPS depending on device type and speed grade.[9].

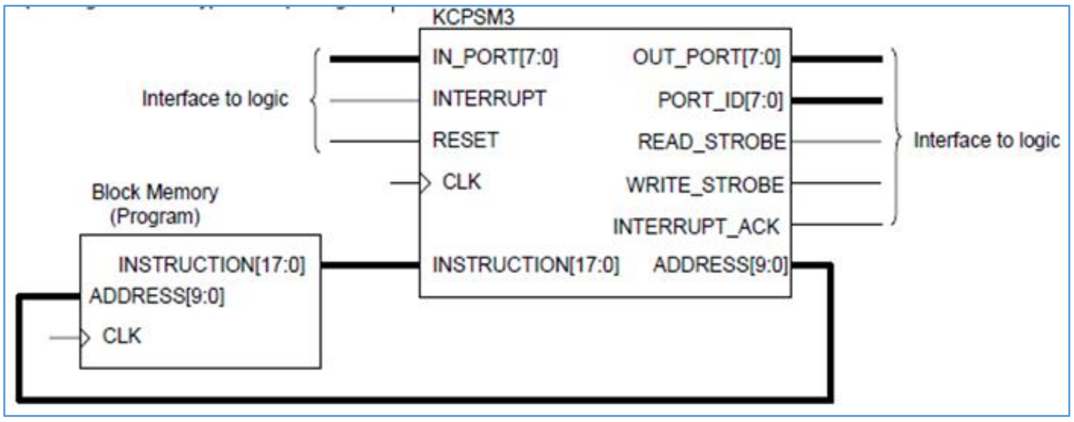


Figure 1. KCPSM3 8-bit microcontroller.

One of the most exciting feature of the KCPSM3 is that it is totally embedded into the device and require no external support. The very fact that any logic can be connected to the module inside the Spartan-3 or Virttex –II dvice means that any additional features can be added to provide ultimate flexibility .It is not so much which inside the KCPSM3 module that makes it useful but the environment in which it live .[8].

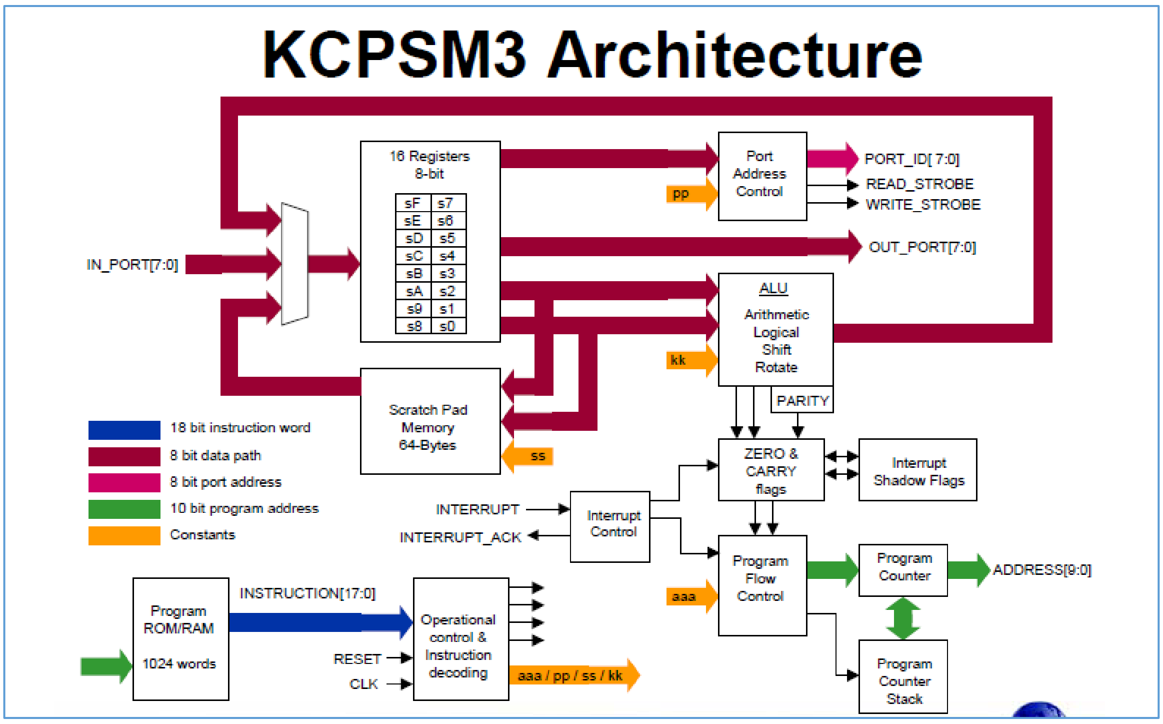


Figure 2: KCPSM3 Architecture.

Using KCPSM3 (VHDL)

The principal method by which KCPSM3 will be used in a VHDL design flow .the KCPSM3 macro is provided as a source VHDL (KCSPM3.vhdl) which has been written to provide an optimum and implementation in a spartan-3 or Virttex –II (PRO) device .the code is suitable for implementation and simulation of thed macro .It has been developed and tested using XST for implementation and ModelSim for simulation .The code should not be modified in any way [9].

KCPSM3 Assembler

The KCPSM3 Assembler is provided as a simple DOS executable file together with three template files .Copy all files KCPSM3.EXE,ROM_form.v and ROM_form coe into your working directory .

Programs are best written with either the standard Notepad or Word pad tools .The file is saved with a psm file extention (8 character name limit).

Open a DOS box and navigate to the working directory .Then run the assembler 'Kcpsm3<filename>[psm]' to assemble your program.It all happens very fast.

Simulation of KCPSM3

KCPSM3 is supplied as VHDL macro together with an assembler .No tools are currently supplied for the direct simulation of code. However, this immediate lack of simulation tools does not appear to have deterred many thousands of Engineers from using PicoBlaze macros over the past few years . Common reasons for this acceptance of this situation are :-

Interaction with hardware

It is very common for PicoBlaze to be highly interactive with the hardware in which it is embedded. With virtually continues interaction between the processor and the input and output ports. It would be difficult to simulate these interactions in a purely software isolated environment. In a similar way, the simulation of the hardware design require-s the stimulus from the processor. So in many cases, the simulation of the processor will become part of the hardware simulation using a tool such as ModelSim. The following pages illustrates how the KCPSM3 macro can be used directly in a VHDL simulation and describes some features withen the coding of the macro witch enhence the simulation of the PSM softwre execution as well as the I/O ports.[10] .

Objectives

1. Integrate the **PicoBlaze embedded processor** with a **VHDL-based UART**.
2. Develop a **PicoBlaze assembly program** to send a fixed text message via the UART.
3. Simulate the complete system in **Xilinx ISE** to verify correct message transmission..

Literature review

1. Introduction and scope

A Universal Asynchronous Receiver/Transmitter (UART) provides simple, byte-oriented serial communication and is widely used for control, debug and data links in FPGA-based systems. Embedding a UART inside an FPGA commonly uses a hardware core written in an HDL (VHDL/Verilog) and—for flexible control—can be paired with a soft microcontroller such as Xilinx's PicoBlaze. The literature on this subject covers pure HDL UART cores, UARTs with FIFO buffering, PicoBlaze-based solutions using prewritten UART macros, and optimizations to boost throughput, reliability and resource-efficiency on Xilinx devices. [17].

2. Technical background

- **UART basics.** A UART consists of a baud-rate generator, transmitter (TX) and receiver (RX) logic, often with start/stop/parity handling, and commonly integrates FIFOs to decouple host and serial timings. Implementations vary from simple bit-serial state machines to multi-channel, buffered designs supporting higher throughput and robustness. [17]
- **PicoBlaze softcore.** PicoBlaze (KCPSM family) is an 8-bit, resource-light Xilinx soft microcontroller commonly supplied with VHDL sources and a set of reference macros (including UART macros and FIFO helpers). Many FPGA SoC designs use PicoBlaze to handle protocol parsing, control and less timing-critical parts of serial I/O while the fast bit-serial timing is in hardware. The PicoBlaze documentation and UART macro user-guides provide common integration approaches and reference designs[17].

3. Representative designs and reference implementations

- **Simple HDL UARTs.** Several academic and conference papers present classic VHDL UARTs (baud generator + RX/TX state machines). These are used for teaching and low-throughput links; they are straightforward to synthesize onto Spartan/Artix devices. (e.g., many publications and online tutorials). [18]
- **UART + FIFO / multi-channel designs.** To avoid data loss and to allow burst transfers, many implementations add asynchronous FIFOs (TX and RX). Papers and project reports show UART+FIFO implemented in VHDL with successful deployment on Spartan/Artix families; such designs permit clock-domain crossing, buffering for host processing, and variable baud rates. [18].
- **PicoBlaze + UART macros / reference designs.** Xilinx-supplied or community-supplied PicoBlaze projects commonly include UART macros (with small FIFOs) and application notes showing how the PicoBlaze can drive/consume bytes and handle higher-level protocol logic (command parsing, debug consoles). The KCPSM6 and earlier KCPSM3 documentation include UART macro examples and integration notes. [18].

4. Efficiency techniques reported in literature

For "high efficiency" (interpreted as high throughput, low-latency, and/or low resource use), the literature repeatedly emphasizes these approaches:

1. **Hardware FIFO buffering** (TX/RX FIFOs) to decouple serial timing from processing and avoid dropped bytes under bursty traffic. FIFO depth tuned to worst-case processing latency yields reliability at modest area cost. [18].
2. **Baud-rate generation and oversampling** — accurate baud generation and receiver oversampling (e.g., 4x–16x) reduce bit errors and allow digital filtering; some designs sample at system clock and use accumulators or running-sum filters for noisy channels. Filtering/oversampling choices trade area & clock constraints for robustness. [18].
3. **Parallelization / multi-channel cores** — implementing several UART channels in parallel or time-multiplexed with shared resources supports multiple serial links (paper examples with 4 channels). This increases throughput per FPGA but raises area/complexity.[18].
4. **Offloading time-critical parts to pure hardware** while keeping higher-level parsing/flow control in PicoBlaze. Hardware handles bit timing and framing; PicoBlaze handles command-level logic. This hybrid approach leverages the strengths of both HDL and softcore. [18].
5. **Resource-aware coding (VHDL optimization)** — designers minimize logic by using counters rather than fully parallel logic for baud generation, leveraging Xilinx primitives (SRL16, distributed RAM for FIFOs where appropriate), and careful floorplanning when necessary to meet timing on faster baud rates. Community projects and Xilinx notes often highlight these optimizations. [18].

5. Noise tolerance & reliability strategies

Some literature addresses robust UART operation over noisy channels: e.g., using digital filters like recursive running-sum filters to eliminate spurious transitions, programmable sample windows to adapt to bitrate and noise, and CRC or packet framing at higher layers to recover from residual errors. These improvements increase gate count but are necessary in industrial applications. [19].

6. Tools, target devices and development flows

- Xilinx tools (ISE for older Spartan devices, Vivado for 7-series and later) are the common synthesis/implementation environments. Many published projects targeted Spartan-3E, Spartan-6, Artix-7 and other Xilinx families. PicoBlaze source and macros are provided in VHDL and reference examples are widely available. Github repositories provide working testbenches and board-level examples (Basys3, Nexys, Zybo) for rapid prototyping. [19].

7. Gaps and opportunities identified in the literature

- **Benchmarking & standardization.** There is limited standardized benchmarking of "efficiency" across designs (area vs throughput vs latency). Many papers provide functionality and device usage for a given FPGA family but comparing different optimizations is hard without common benchmarks.
- **Integration with modern Xilinx toolchains & IP.** Some community designs still target older toolchains (ISE) or older PicoBlaze variants; updates for Vivado flows and 7-series/UltraScale devices can be improved and documented.
- **Hardware offload variants.** There is space for more systematic studies of how much functionality to move into hardware vs PicoBlaze (e.g., flow control, DMA-like burst transfers, multi-drop addressing) for different application classes.
- **Security & advanced features.** Little work in the surveyed literature addresses encrypted or authenticated serial links within lightweight FPGA UARTs; low-cost crypto offload could be a niche. [20].

Results and discussion

PART A

VHDL TOP LEVEL CODE

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library STD;
use STD.textio.all;
use IEEE.std_logic_textio.all;
entity hamed_SID is
```

This defines the name of the design

```
   port(
      clk, reset: in std_logic;
      tx: out  std_logic
   );
end hamed_SID;
```

This defines the end of the entity

```
architecture arch of ha                                          ure
   -- KCPSM3/ROM signals
   signal address: std_logic_vector(9 downto 0);
   signal instruction: std_logic_vector(17 downto 0);
   signal port_id: std_logic_vector(7 downto 0);
   signal in_port, out_port: std_logic_vector(7 downto 0);
   signal wstrobe, rstrobe: std_logic;
   signal interrupt, interrupt_ack: std_logic;
   signal kcpsm_reset: std_logic;
```

The signal interface is declared

```
   -- I/O port signals
   -- output enable
   --signal en_d: std_logic_vector(6 downto 0);
   -- uart
   signal w_data: std_logic_vector(7 downto 0);
   signal rd_uart, rx_not_empty, rx_empty: std_log
   signal wr_uart, tx_full: std_logic;
```

The UART signal is declared

```
   -- multiplier
begin
   -- ==============================================
   --  File Writing
   -- ==============================================
         output_file: process (wstrobe)

         file datafile : text open write_mode is "saleh_output.txt";
```

Text code to output the input file

```
                  variable serialout :  line ;
         begin
           if rising_edge (wstrobe) then
                         write(serialout,"5370160 Output UART_HEX --  ");
                            hwrite(serialout,out_port);
              writeline(datafile,serialout);
         end if;
      end process output_file;

   -- ===============================================
   --  KCPSM and ROM instantiation
   -- ===============================================
   proc_unit: entity work.kcpsm3
      port map(
         clk=>clk, reset =>reset,
         address=>address, instruction=>instruction,
         port_id=>port_id, write_strobe=>wstr
         out_port=>out_port, read_strobe=>rs
         in_port=>in_port, interrupt=>interr
         interrupt_ack=>interrupt_ack);

   saleh_rom: entity work.saleh
      port map(clk => clk, address=>address,
               instruction=>instruction);
   -- =========================================================
   -- Port Mapping and component instantiation for transmitter

   -- =========================================================
   uart_xmitter: entity work.xmitter(arch)
   port map( clk     => clk    , reset   => reset,
                         wr_uart => wstrobe, w_data  => out_port,
                         tx_full => tx_full, tx      => tx     );

  in_port(7) <= tx_full;
   in_port( 6 downto 0) <= "0000000";

end arch;
```

This instantiates the kcpsm3 picoblaze processor

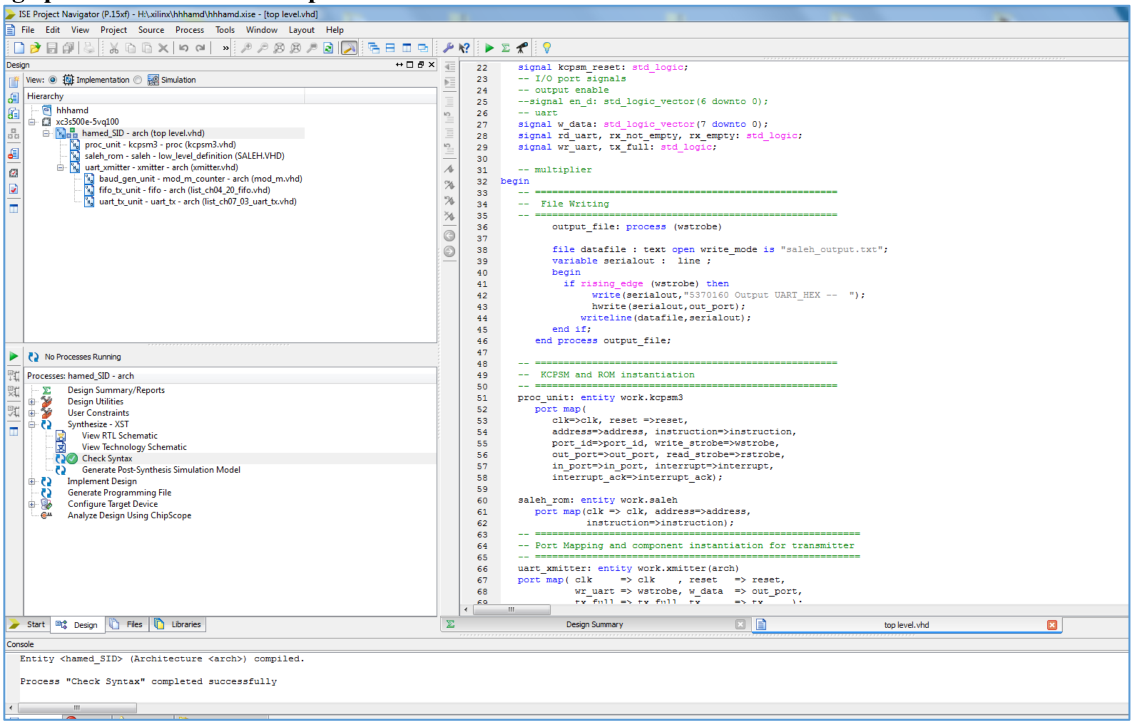
Instantiates the ROM file

Evidence for successful compilation of result

```
Started : "Check Syntax for hamed_SID".
Running xst...
Command Line: xst -intstyle ise -ifn H:/.xilinx/hhhamd/hamed_SID.xst -ofn
hamed_SID.stx

=========================================================================
*                          HDL Compilation                              *
=========================================================================
Compiling vhdl file "C:/Users/salehh2/Desktop/new cw/mod_m.vhd" in Library
work.
Entity <mod_m_counter> compiled.
Entity <mod_m_counter> (Architecture <arch>) compiled.
Compiling vhdl file "C:/Users/salehh2/Desktop/new cw/list_ch04_20_fifo.vhd"
in Library work.
Entity <fifo> compiled.
Entity <fifo> (Architecture <arch>) compiled.
Compiling         vhdl          file          "C:/Users/salehh2/Desktop/new
cw/list_ch07_03_uart_tx.vhd" in Library work.
Entity <uart_tx> compiled.
Entity <uart_tx> (Architecture <arch>) compiled.
Compiling vhdl file "C:/Users/salehh2/Desktop/new cw/kcpsm3.vhd" in Library
work.
```

```
Entity <kcpsm3> compiled.
Entity <kcpsm3> (Architecture <proc>) compiled.
Compiling vhdl file "C:/Users/salehh2/Desktop/demo/Assembler/SALEH.VHD" in
Library work.
Entity <saleh> compiled.
Entity <saleh> (Architecture <low_level_definition>) compiled.
Compiling vhdl file "C:/Users/salehh2/Desktop/new cw/xmitter.vhd" in Library
work.
Entity <xmitter> compiled.
Entity <xmitter> (Architecture <arch>) compiled.
Compiling vhdl file "C:/Users/salehh2/Desktop/new cw/top level.vhd" in
Library work.
Entity <hamed_SID> compiled.
Entity <hamed_SID> (Architecture <arch>) compiled.

Process "Check Syntax" completed successfully
```

Image proof of successful compilation


ISE Project Navigator (P.15xf) - H:\.xilinx\hhhamd\hhhamd.xise - [top level.vhd]
File Edit View Project Source Process Tools Window Layout Help
Design
View: Implementation Simulation
Hierarchy
No Processes Running
Start Design Files Libraries
Design Summary
top level.vhd
Console

```
hhhamd
xc3s500e-5vq100
  hamed_SID - arch (top level.vhd)
    proc_unit - kcpsm3 - proc (kcpsm3.vhd)
    saleh_rom - saleh - low_level_definition (SALEH.VHD)
    uart_xmitter - xmitter - arch (xmitter.vhd)
      baud_gen_unit - mod_m_counter - arch (mod_m.vhd)
      fifo_tx_unit - fifo - arch (list_ch04_20_fifo.vhd)
      uart_tx_unit - uart_tx - arch (list_ch07_03_uart_tx.vhd)
```

```
Processes: hamed_SID - arch
  Design Summary/Reports
  Design Utilities
  User Constraints
  Synthesize - XST
    View RTL Schematic
    View Technology Schematic
    Check Syntax
    Generate Post-Synthesis Simulation Model
  Implement Design
  Generate Programming File
  Configure Target Device
  Analyze Design Using ChipScope
```

```vhdl
   signal kcpsm_reset: std_logic;
   -- I/O port signals
   -- output enable
   --signal en_d: std_logic_vector(6 downto 0);
   -- uart
   signal w_data: std_logic_vector(7 downto 0);
   signal rd_uart, rx_not_empty, rx_empty: std_logic;
   signal wr_uart, tx_full: std_logic;

   -- multiplier
begin
   -- =====================================================
   --  File Writing
   -- =====================================================
         output_file: process (wstrobe)

         file datafile : text open write_mode is "saleh_output.txt";
         variable serialout :  line ;
         begin
           if rising_edge (wstrobe) then
                write(serialout,"5370160 Output UART_HEX --  ");
                hwrite(serialout,out_port);
              writeline(datafile,serialout);
         end if;
      end process output_file;

   -- =====================================================
   --  KCPSM and ROM instantiation
   -- =====================================================
   proc_unit: entity work.kcpsm3
      port map(
         clk=>clk, reset =>reset,
         address=>address, instruction=>instruction,
         port_id=>port_id, write_strobe=>wstrobe,
         out_port=>out_port, read_strobe=>rstrobe,
         in_port=>in_port, interrupt=>interrupt,
         interrupt_ack=>interrupt_ack);

   saleh_rom: entity work.saleh
      port map(clk => clk, address=>address,
               instruction=>instruction);
   -- =========================================================
   -- Port Mapping and component instantiation for transmitter
   -- =========================================================
   uart_xmitter: entity work.xmitter(arch)
   port map( clk     => clk    , reset    => reset,
             wr_uart => wstrobe, w_data   => out_port,
```

```
Entity <hamed_SID> (Architecture <arch>) compiled.

Process "Check Syntax" completed successfully
```

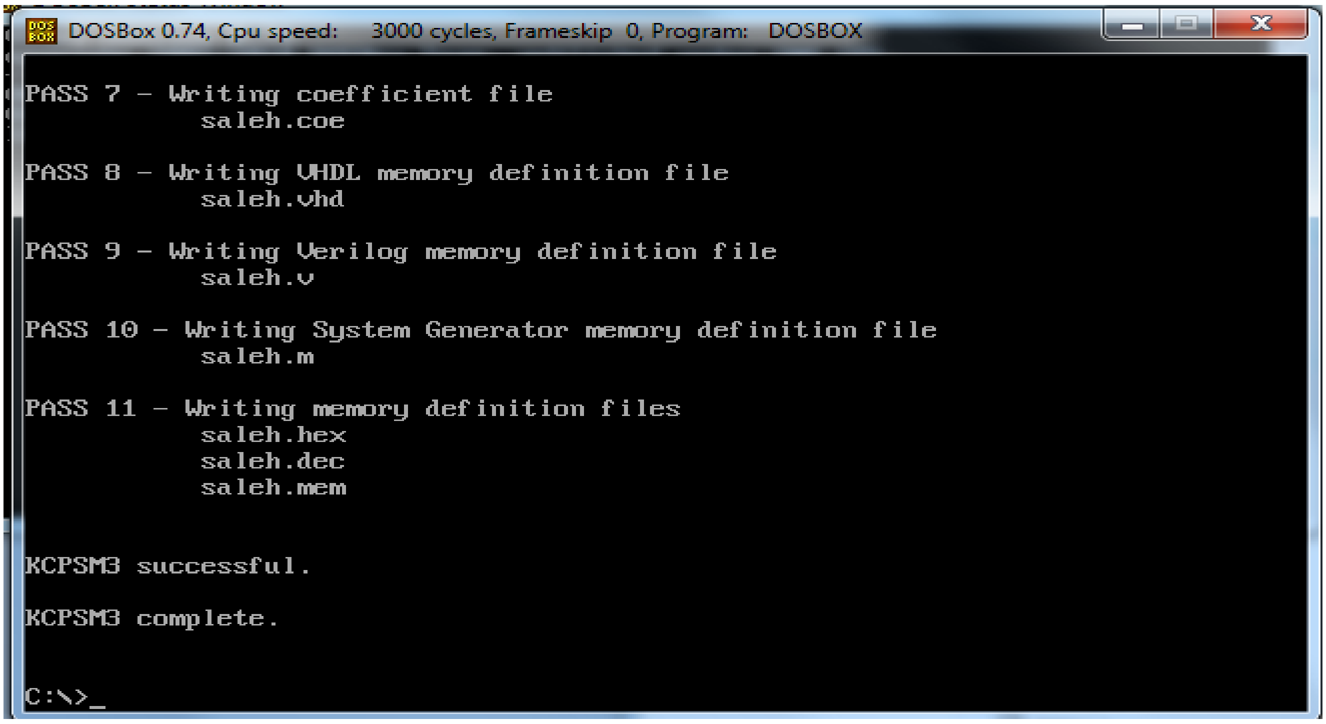
Figure 3: Image proof of successful compilation PART B

```
;5370160 = 5+3+7+0+1+6+0
begin:      LOAD   s1,05          ;
            LOAD   s2,s1          ;
            CALL   tx             ;
            OUTPUT s1,80          ;
            LOAD   s1,03          ;
            ADD    s2,s1          ;
            CALL   tx             ;
            OUTPUT s1,80          ;
            LOAD   s1,07          ;
            ADD    s2,s1          ;
            CALL   tx             ;
            OUTPUT s1,80          ;
            LOAD   s1,00          ;
            ADD    s2,s1          ;
            CALL   tx                    ;
            OUTPUT s1,80                 ;
            LOAD   s1,01                 ;
            ADD    s2,s1                 ;
            CALL   tx                    ;
            OUTPUT s1,80                 ;
            LOAD   s1,06                 ;
            ADD    s2,s1                 ;
            CALL   tx                    ;
            OUTPUT s1,80                 ;
            LOAD   s1,00                 ;
            ADD    s2,s1                 ;
            CALL   tx                    ;
            OUTPUT s1,80                 ;
            LOAD   s1,s2                 ;
            CALL   tx                    ;
            OUTPUT s1,80                 ;
            JUMP   begin                 ;
;----------------------------------
;Status of tx wait for tx
;----------------------------------
tx :        INPUT  s0,00                 ;
            AND    s0,80                 ;
            JUMP   NZ,tx                 ;
            RETURN                       ;
```

The input file is read here.

All the digits are added and loaded before transmitting to UART character.

Image to show that the picoblaze assembler compiled successful

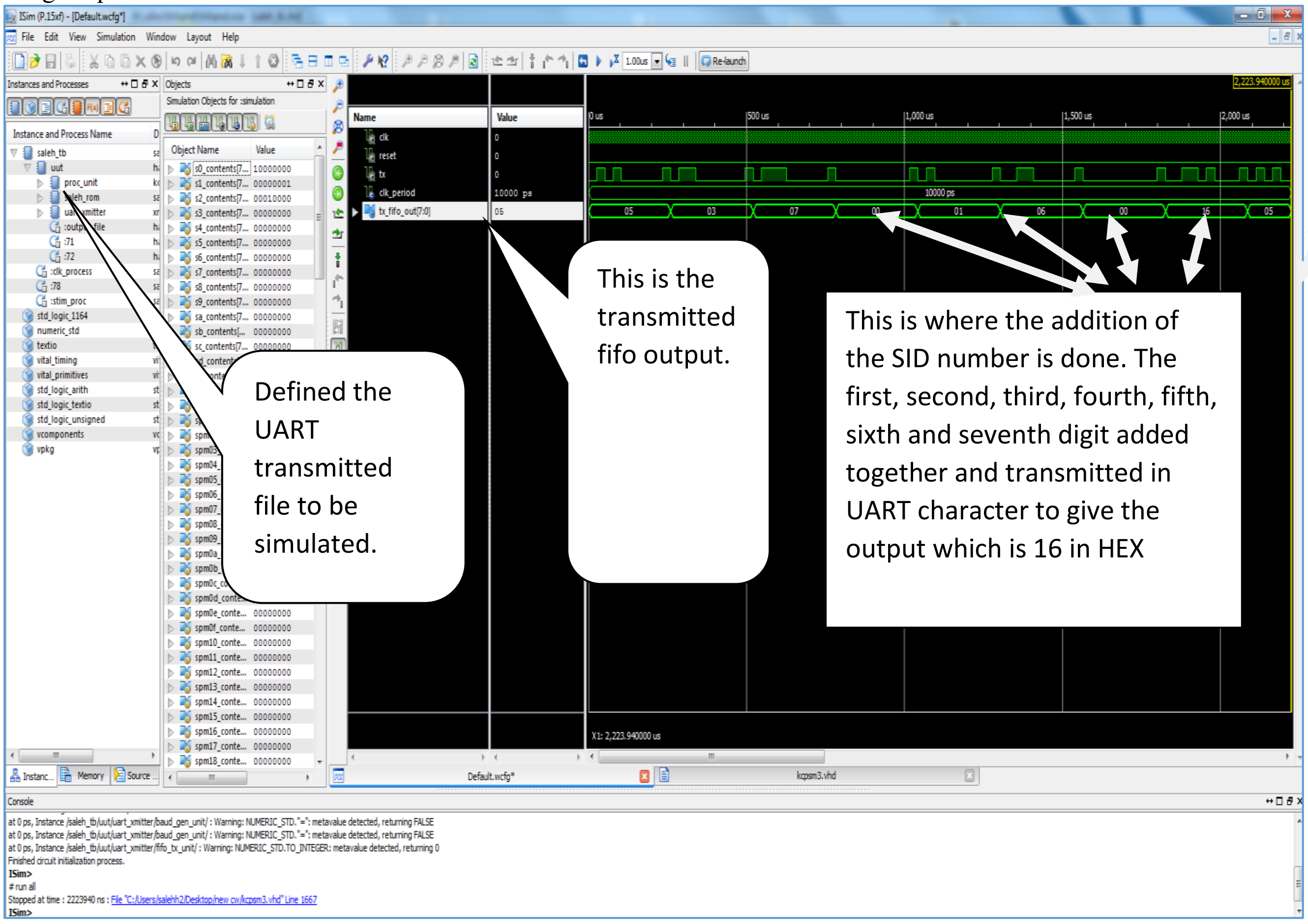


DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

```
PASS 7 - Writing coefficient file
           saleh.coe

PASS 8 - Writing VHDL memory definition file
           saleh.vhd

PASS 9 - Writing Verilog memory definition file
           saleh.v

PASS 10 - Writing System Generator memory definition file
           saleh.m

PASS 11 - Writing memory definition files
           saleh.hex
           saleh.dec
           saleh.mem


KCPSM3 successful.

KCPSM3 complete.


C:\>_
```

Figure 4.Image to show that the picoblaze assembler compiled successful.

PART C
Image capture of Simulation behaviour


This is the transmitted fifo output.
Defined the UART transmitted file to be simulated.
This is where the addition of the SID number is done. The first, second, third, fourth, fifth, sixth and seventh digit added together and transmitted in UART character to give the output which is 16 in HEX
clk
reset
tx
clk_period
tx_fifo_out[7:0]
05
03
07
00
01
06
00
16
05

```
at 0 ps, Instance /saleh_tb/uut/uart_xmitter/baud_gen_unit/ : Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
at 0 ps, Instance /saleh_tb/uut/uart_xmitter/baud_gen_unit/ : Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
at 0 ps, Instance /saleh_tb/uut/uart_xmitter/fifo_tx_unit/ : Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
Finished circuit initialization process.
ISim>
# run all
Stopped at time : 2223940 ns : File "C:/Users/salehh2/Desktop/new cw/kcpsm3.vhd" Line 1667
ISim>
```

Figure 5: Image capture of Simulation behaviour.

Output text file result

```
5370160 Output UART_HEX --  05
5370160 Output UART_HEX --  03
5370160 Output UART_HEX --  07
5370160 Output UART_HEX --  00
5370160 Output UART_HEX --  01
5370160 Output UART_HEX --  06
5370160 Output UART_HEX --  00
5370160 Output UART_HEX --  16
5370160 Output UART_HEX --  05
5370160 Output UART_HEX --  03
5370160 Output UART_HEX --  07
5370160 Output UART_HEX --  00
```

The output results from Xilinx.

The loop is repeated

The VHDL code that instantiated the text file result

```
-- =====================================================
   --  File Writing
   -- =====================================================
         output_file: process (wstrobe)

         file datafile : text open write_mode is "saleh_output.txt";
                  variable serialout :  line ;
         begin
           if rising_edge (wstrobe) then
```

```
                    write(serialout,"5370160 Output UART_HEX --  ");
                       hwrite(serialout,out_port);
              writeline(datafile,serialout);
         end if;
      end process output_file;
```

Conclusion

1. The task was successfully implemented, and the generated code in **Xilinx** was adequately documented with proper comments.
2. The inclusion of the **SID number** was correctly achieved within the design.
3. The first through seventh digits of the number were added together, and the result was transmitted via **UART**, producing an output value of **16 (HEX)**.
4. A comprehensive understanding of digital design using **VHDL** was accomplished through the practical application.
5. Evidence from academic studies, application notes, and open-source projects supports a **hybrid approach** that combines:
    a. Implementing time-critical UART functions in **VHDL** with optional **FIFO** buffering and noise filtering.
    b. Employing the lightweight **PicoBlaze** softcore for higher-level protocol processing and control logic.
6. Efficiency improvements are achieved through:
    a. Careful sizing of **FIFO** buffers according to application requirements.
    b. Leveraging block RAM or distributed memory resources to minimize logic utilization.
    c. Offloading repetitive byte-handling tasks to dedicated hardware components.
    d. Writing optimized **VHDL** code that maps effectively onto **Xilinx FPGA** architectures.
7. The literature further highlights potential research directions, including:
    a. Establishing standardized benchmarks for evaluating UART performance and efficiency.
    b. Updating reference designs to align with modern **Vivado** toolchains and next-generation FPGA families.
    c. Exploring advanced features such as multi-channel support, DMA-like burst transfers, and enhanced communication security.

Recommended architecture for a “high-efficiency” design

1. **Hardware UART core (VHDL)** that implements precise baud generation, oversampled receiver (configurable oversample rate), start/stop/parity support, and optional digital filtering (programmable running-sum or majority voting).[20].
2. **Asynchronous TX and RX FIFOs** (depth sized for worst-case PicoBlaze latency and expected bursts). Use block RAM or distributed RAM depending on size and device family to save LUTs.
3. **Flow control and DMA-style bursting**: Add simple hardware signals to notify PicoBlaze of FIFO fill levels and allow block transfers (PicoBlaze reads/writes bursts from the FIFO via port-mapped IO).
4. **PicoBlaze firmware** for higher-level parsing, protocol handling, error recovery, and optional command interface (console). Keep timing-critical tasks in hardware to avoid dropped bytes. Use PicoBlaze UART macros if suitable. [20].
5. **Verification**: Build testbench with variable jitter/noise models, exercise multiple baud rates and channel error injection. Target both functional and timing closure in the chosen Xilinx toolchain (Vivado for modern FPGAs). [20].

References

[1] Albahit Journal of Applied Sciences. (2025). Design and implementation of a high efficiency UART model using VHDL and PicoBlaze on Xilinx FPGA platforms. Albahit Journal of Applied Sciences, 5(1), 1–11.
[2] Ashenden, P. J. (2002). The designer’s guide to VHDL (2nd ed.). Morgan Kaufmann Publishers.
[3] Brown, S., & Vranesic, Z. (2004). Fundamentals of digital logic with VHDL design (2nd ed.). McGraw-Hill.
[4] Clemente, J. A. (2014). Introduction to VHDL programming (M. Sánchez-Élez, Trans.). Retrieved November 13, 2014.
[5] GCC. (n.d.). GNU Compiler Collection (GCC). http://gcc.gnu.org
[6] GHDL. (n.d.). Open-source VHDL simulator GHDL. http://ghdl.free.fr
[7] Mano, M. M., & Kime, C. (2000). Logic and computer design fundamentals. Prentice Hall.
[8] Perry, D. (1998). VHDL (3rd ed.). McGraw-Hill.
[9] PicoBlaze UART & KCPSM6 User Guide/Macros. (n.d.). UART macros and PicoBlaze integration notes. Retrieved from viterbi-web.usc.edu, eng.auburn.edu

[10] Qualis Design Corporation. (n.d.). VHDL reference cards. http://www.vhdl.org/rassp/vhdl/guidelines/vhdlqrc.pdf
[11] Qualis Design Corporation. (n.d.). 1164 VHDL reference guidelines. http://www.vhdl.org/rassp/vhdl/guidelines/1164qrc.pdf
[12] ResearchGate. (n.d.). Academic articles on efficient UART architectures. Retrieved from https://www.researchgate.net
[13] Roth, C. H. (1997). Digital systems design using VHDL. ITP Nelson.
[14] Skahill, K. (1996). VHDL for programmable logic. Addison Wesley.
[15] Wikipedia. (n.d.). VHDL. In Wikipedia. http://en.wikipedia.org/wiki/VHDL
[16] Xilinx. (n.d.). ISE Design Suite. http://www.xilinx.com/tools/designtools.htm
Yalamanchili, S. (2001). Introductory VHDL: From simulation to synthesis. Prentice Hall.