University Bremerhaven

Course Documentation

System-On-Chip Design
[SY–SOC]

- Part 1: SoC Concept
- Part 2: Integration of 8 Bit Microprocessors
- Part 3: Intergation of 32 Bit Microprocessors
- Teil 4: Integrating Peripheral Functions
- Teil 5: Practical Lab Experiments

Revision: V1.1d (minor)

Release: March 2020

Prof. Dr.-Ing. Kai Mueller
University of Applied Sciences Bremerhaven
Institute for Automation and Electrical Engineering
An der Karlstadt 8

D–27568 Bremerhaven / Germany

Phone: +49 471 48 23 – 415
FAX: +49 471 48 23 – 555

Email: kmueller@hs–bremerhaven.de

I Introduction

I.I Course Documentation

See < http://www1.hs–bremerhaven.de/kmueller/> for updates.

I.II System-On-Chip Design

Beginning in the mid of the seventies microprocessor build the core of complex digital systems. The logic could easily be changed by reprogramming. Later the DSPs (DIgital Signal Processor) came up to cope with increasing demand on signal processing.

With increasing availability, low cost, reliablity and flexibility microprocessors are used in combination with programmable logic (first GALs, CPLDs, later FPGAs). Nowadays the FPGAs offer enough logic to include microprocessors or they already contain microprocessor cores (PowerPC, Sparc, ARM). The logic in programmable logic allows to implement all peripherals needed for an application inside of one single chip. The result is an (almost) unbeatable performance and lower overall cost.

This course teaches to design system-on-chip using VHDL, assembler, “C” language and modern support tools.

Bremerhaven, May 2011

Kai Müller
<kmueller@hs–bremerhaven.de>
Tel: (0471) 4823 – 415

II Contents

1 Motivation for System-on-Chip (SoC)

SoCs offer unbeatable price, performance, and reliability. At the same time the required power is minimized which is very important for portable device (e.g. medical equipment) or applications in aircrafts and space (e.g. satellites). Instead of using an microcontroller and several additional chips a single chip solutions wins in almost all aspects.

SoCs became also in interesting alternative to PCs for controlling systems, since several operating systems (Linux is a popular example) are executable on SoCs.

1.1 Brief Development History of SoCs

The early predecessor of a SoC was the SBC (Single Board Computer). All required logic was integrated on a single board.

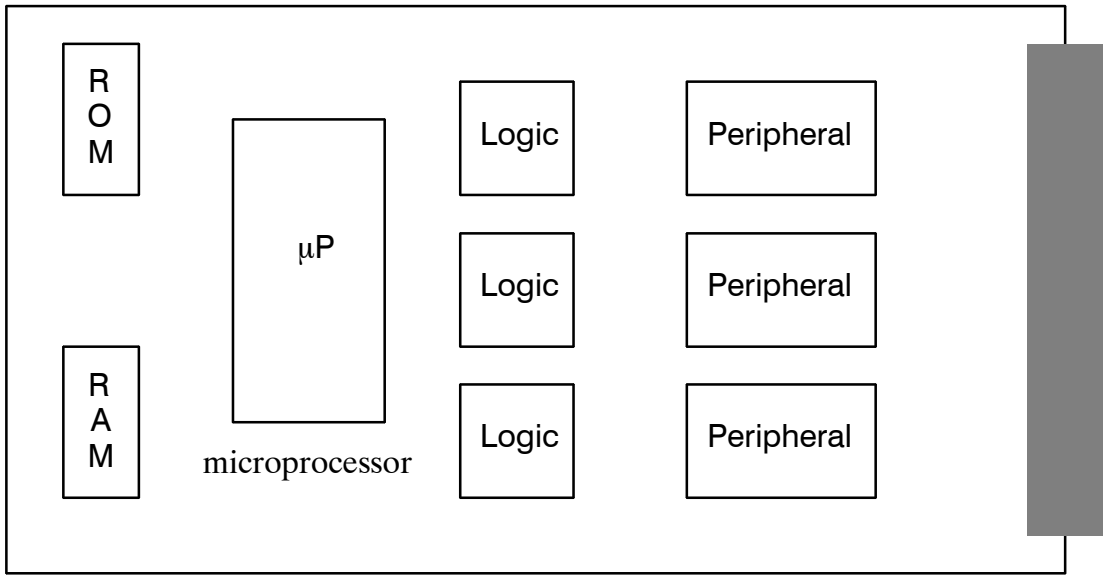


Figure 1.1: Single Board Computer

When it became possible to integrate more logic into ICs Memory and some peripherals where integrated into the microprocessor chip. The result is called “microcontroller”. A single board computer with microcontrollers contains less chips and becomes cheaper. However, still additional logic and peripherals are necessary, since a microcontroller does not contain all required peripherals for most applications.

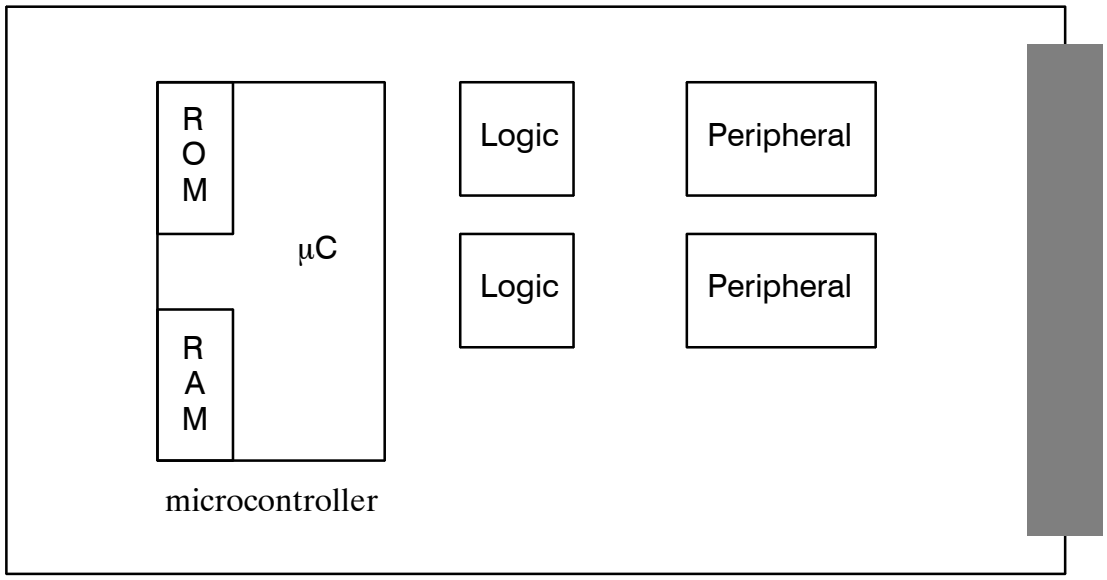


Figure 1.2: Single Board Computer with microcontroller

With the availability of programmable logic the discrete logic ICs (costly and require board space and several extra-wires) could disappear. Fig. 1.3 illustrates this step.

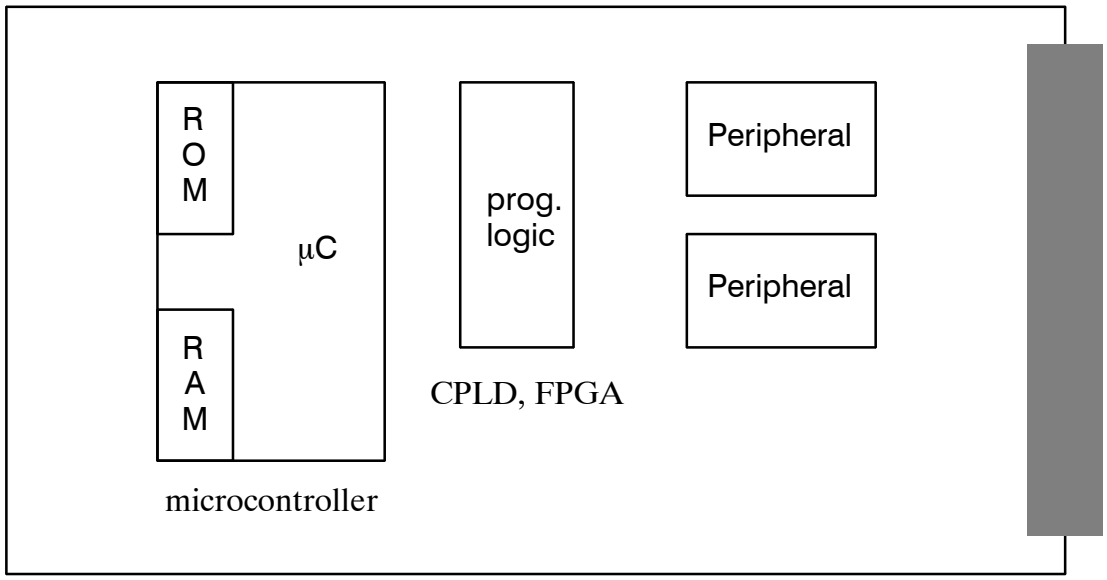


Figure 1.3: Single Board Computer with microcontroller and programmable logic

The FPGAs of today include microprocessor core, memories, and enough logic to include all kind of peripherals.

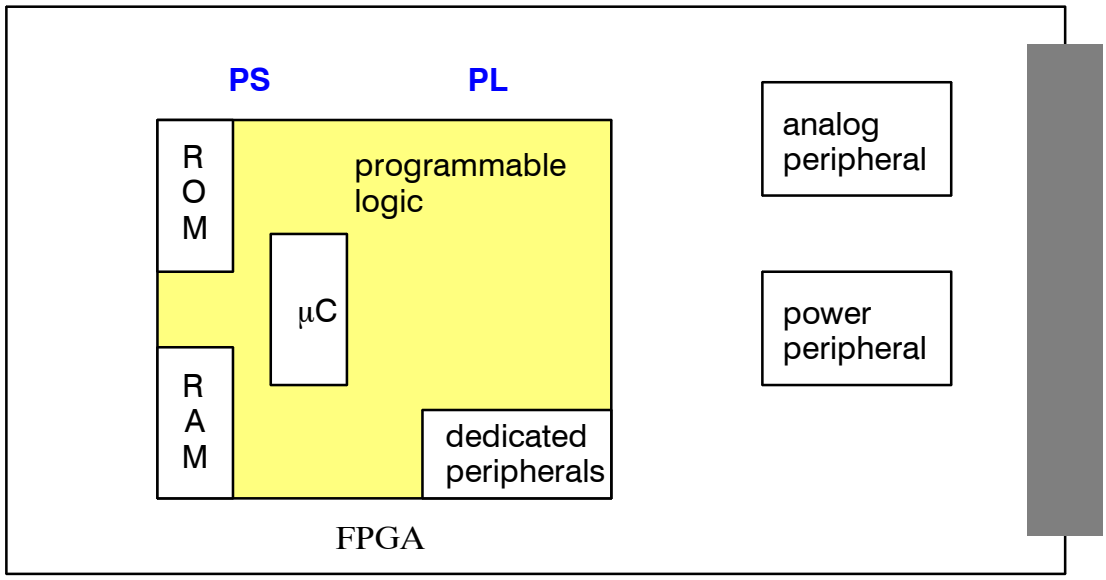


Figure 1.4: SoC structure

Although this is denoted as SoC, it is not possible to include all peripherals into an FPGA. Analog circuits consist of an entirely different technology. It is therefore not possible to integrate analog and digital blocks in the same chip. Some devices exist with analog and digital circuits but this results in poor speed for the digital part and limited precision in the analog part. New digital devices allow integration of up to 12 bit ADCs.

An example of a suitable SoC platform is Zynq™ from Xilinx Inc. (fig. 1.5). This device contains

- Dual ARM Cortex™-A9 MPCore (up to 800MHz)
  with single and double precision floating point unit and
  32kB Instruction / 32kB Data L1 Cache, 512kB L2 Cache, 256kB on-chip Memory
- DDR3, DDR2 and LPDDR2 Dynamic Memory Controller
- 2 x QSPI, NAND Flash and NOR Flash Memory Controller
- 2 x USB2.0 (OTG), 2x GbE, 2x CAN2,0B 2x SD/SDIO, 2x UART, 2x SPI, 2x I2C, 4x 32b GPIO
- AES & SHA 256b encryption engine for secure boot and secure configuration
- Dual 12bit 1Msps Analog-to-Digital converter
- Up to 17 Differential Inputs
- Advanced Low Power 28nm Programmable Logic:
  - 28k to 235k Logic Cells (approximately 430k to 3.5M of equivalent
  - ASIC Gates)
  - 240kB to 1.86MB of Extensible Block RAM
  - 80 to 760 18x25 DSP Slices (58 to 912 GMACS peak DSP performance)
- PCI Express® Gen2x8 (in largest devices)

- 154 to 404 User IOs (Multiplexed + SelectIO ™)
- 4 to 12 10.3Gbps Transceivers (in largest devices)

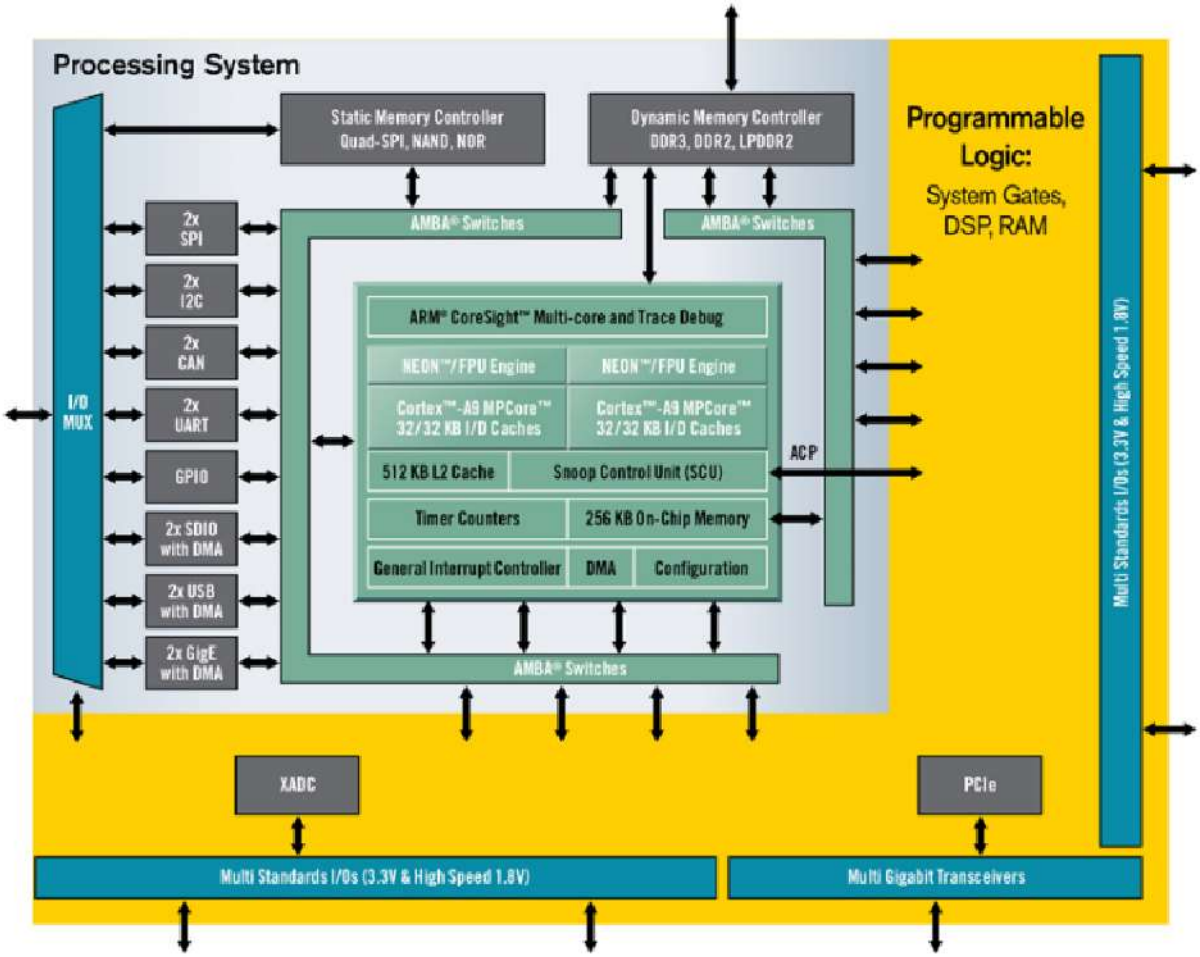


Figure 1.5: Xilinx Zynq ™ platform

Any other FPGA with sufficient logic resources can implement efficient SoCs. It it not required that a microcontroller in already integrated in the FPGA. A number of microcontrollers can be implemented in the programmable logic. it all cases internal “block RAMs” should be used for program and data memory.

Due to the complexity of many peripherals and interfaces the engineer takes advantage of existing blocks called IPs (Intellectual Properties). The main task is to integrate these components for a particular application. Of course, application specific logic is always required.

The designer can now decide if an algorithm is coded in hardware or if a software solution is possible (or both). This is referred to hardware-software codesign.

2 Integrating an 8-Bit Microcontroller

The PicoBlaze ™ cpu will be served as an example to integrate an 8 bit microcontroller into an application. PicoBlaze is a highly optimized design. It is property of Xilinx Inc. but it is open source and “zero cost”, which means it can be included freely into any system. Since it takes advantage of LUT primitives (look-up tables) and BRAM (block RAM) it should be targeted to the FPGA families it was designed for. The required resources are very low, several microcontrollers can be included even in small devices.

PicoBlaze is a modern RISC type cpu with a symmetric instruction set. Symmetric means that every instruction can be applied to all registers. All operations arithmetic and logic operations can be carried out within 2 x 16 registers (register bank A and register bank B). The remaining instructions involve flow control and simple LOAD/STORE to and from memory and IO ports.

PicoBlaze has been developed in a number of versions. The latest version is KCPMS6 which stands for

(K)Constant Coded Programmable State Machine version 6.

This indicates that PicoBlaze is a finite state machine. The developer of the architecture, coding, and software tools is Ken Chapman, so the name can also be interpreted as

Ken Chapman’s Programmable State Machine version 6.

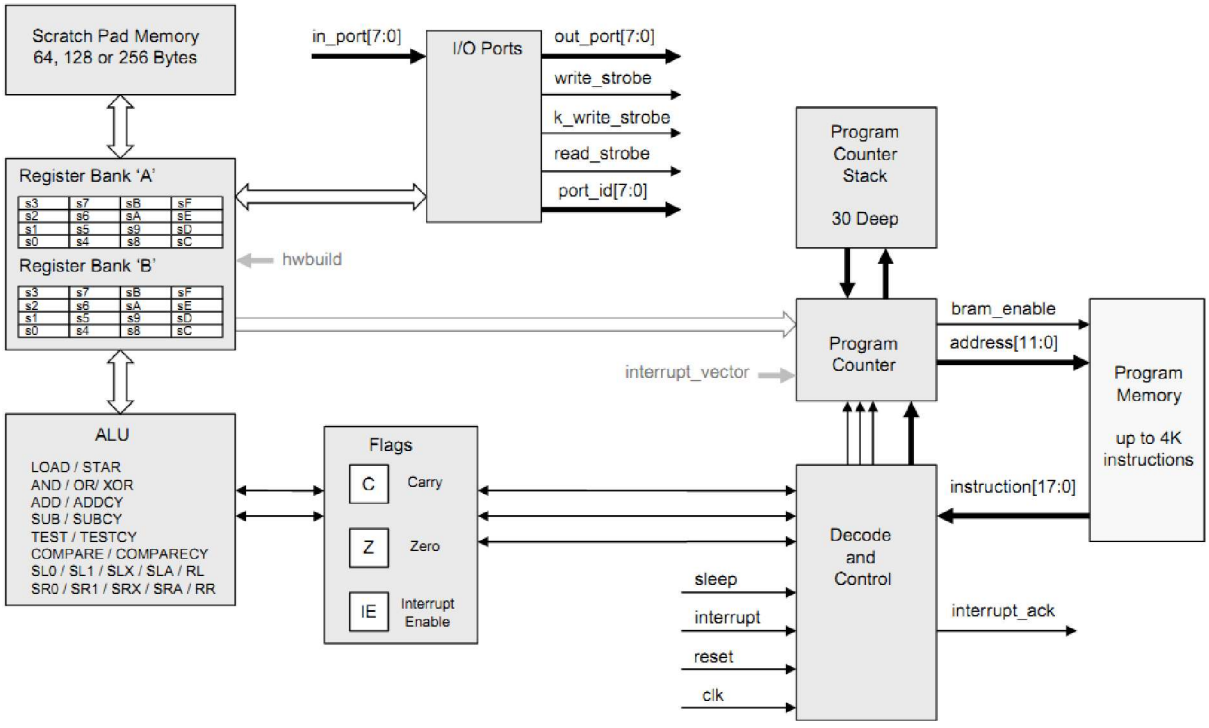


Figure 1.6: PicoBlaze KCPMS6 architecture and IO (© Xilinx)

The features can be explained by the blocks in fig. 1.6. New features in version 6 are marked in **bold** letters.

- The central component is the "Decode and Control" block (state machine). It can operate at high clock rates (**100 MHz and above**). The PicoBlaze execution time for all instructions is 2 clock cycles. With a 100 MHz *clk* signal it executes 50 million instructions per second.
  The signals of the "Decode and Control" block:

reset	clears registers and flags and starts execution at the address 0x000.
interrupt/ *interrupt_ack*	interrupts execution and starts at a predefined address which can be specified during instantiation of the processor component. The output interrupt_ack indicates that the interrupt has been serviced. External logic should reset interrupt if interrupt_ack is set.
sleep	The sleep signal stops program execution. The duration is unlimited.
instruction	18 bit instruction code from the block RAM (program memory). Every instruction is 18 bits wide.

- Program Counter
  The program counter is 10, **11, or 12 bits** wide, thus allowing to interface 1K, **2K, or 4K** instruction (18 b its wide). Not all FPGA families are suited for all sizes. On Spartan-6 the number of instructions should be limited to 2K (11 bits).

- Program memory
  Program memory can be up to **4K instructions**. Memory blocks are in most devices organized as 2KByte memory. Since the BRAM supports a parity bit and it can be organized as 1024 x 18 (instead of 2048 x 8 or 2048 x 9) the 18 bit instruction word fits perfectly into a BRAM. More than 1K instruction require more BRAM blocks. Small size FPGA have approximately 20 BRAM blocks.

- Program counter stack
  The stack captures 30 return addresses. This means that the number of calls within calls must not exceed 30. Please note that an interrupt requires also one stack word. Interrupts should never allowed within interrupts. This is one of the best methods to crash a microcontroller program!

- Register banks
  PicoBlaze contains now **two independent register banks** of 16 registers each. Transfer of registers between banks is possible with the **new STAR (store alternate register) instruction**.

- Scratch pad memory
  This is the RAM area for PicoBlaze. The size is now configurable from 64 **up to 256 bytes**.

- ALU
  The arithmetic logic unit carries out all logic and numeric operations. **Some new instructions have been added to improve usability (e.g. COMPARECY).**
- FLAGS
  Flags are used for program flow control (conditional jumps, calls and returns). **The new HWBUILD instructions allows to set the C (Carry) flag (not the only function of HWBUILD).**
- IO ports
  Up to 256 ports (input and output) allow interfacing to external logic or pins without extra logic (multiplexer or demultiplexer). Strobe signals are provided to indicate that data is written or read (important for some external interfaces). **The new k_write_strobe signal indicates a write cycle of a constant.**

***

Lab #01:

2.1 PicoBlaze6™ Microcontroller Core [Release 6 –NEW!]

The 8 bit microcontroller should be programmed into the Spartan6 FPGA. Since a microcontroller always requires software, an initial assembly program is required to verify proper operation of the device.

- Create a new project in Vivado and name it *picosp6s*
- Use the supplied top level file *sp6picor.vhd* . In order to satisfy all dependencies copy and import the following VHDL files:

kcpsm6.vhd	processor core
uart_tx6.vhd	UART transmitter (6 input LUT)
uart_rx6.vhd	UART receiver (6 input LUT
kcuart_rx.vhd	UART transmitter implementation
pfirmware.vhd	inital program ROM contents (block ram)

- The processor requires an initial program contained in
  pfirmware.vhd
  This is not a supplied file. instead it must be creates by the assembler *kcpsm6.exe* in the *ASSEMBLER* subdirectory. Therefore, an initial assembly file *pfirmware.psm* has to be coded (see template in fig. 1.8).

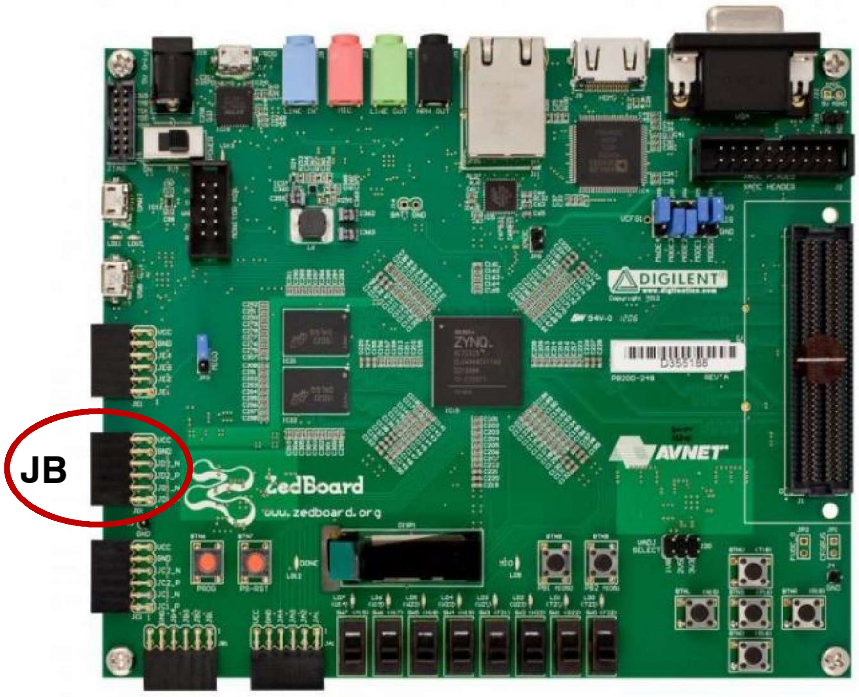


Figure 1.7: Zynq-7000 FPGA board (ZedBoard)

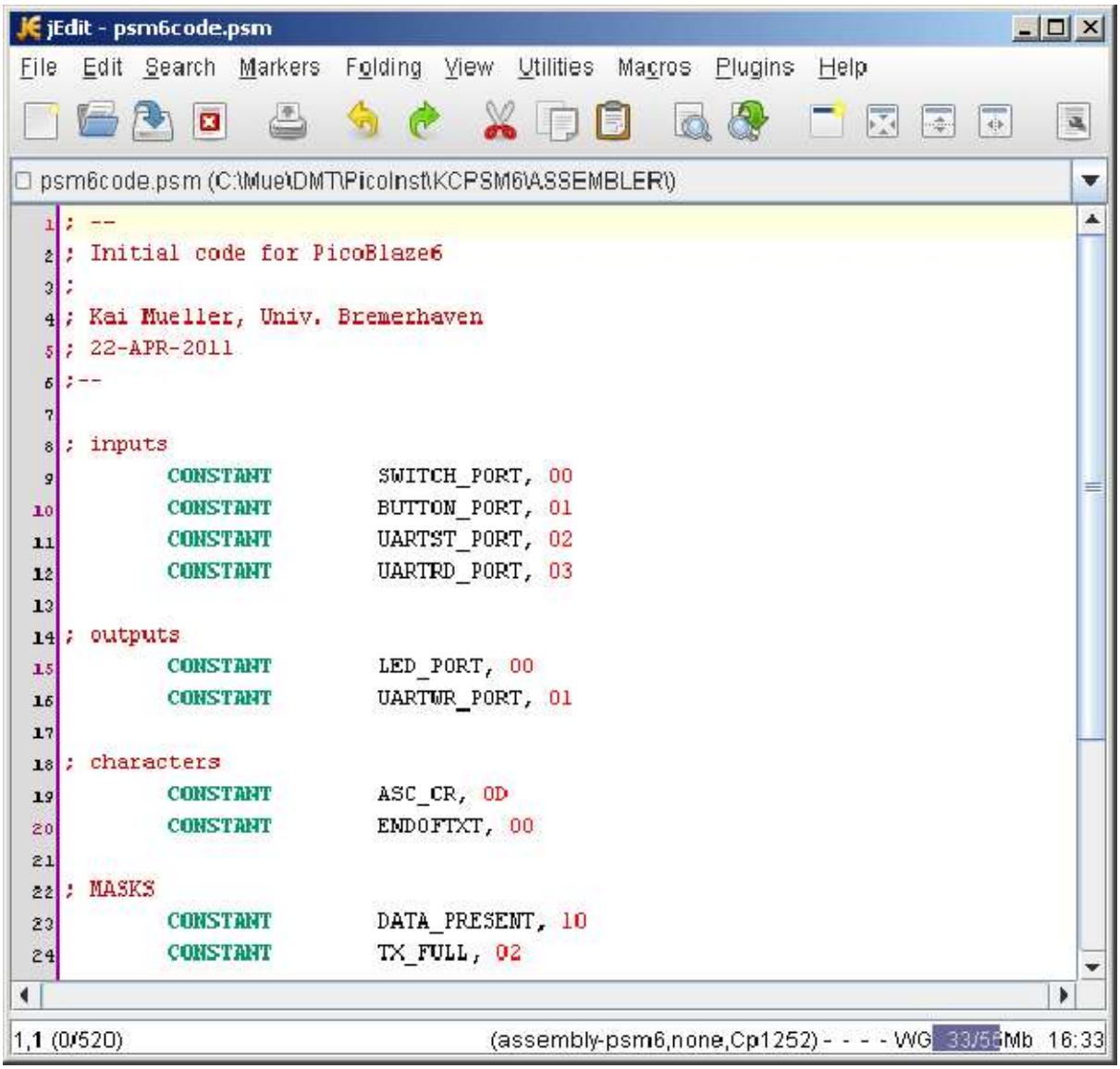


Figure 1.8: Editing an assembly file (.psm) with Jedit (ignore PORT definitions!)

According to CONSTANT definitions the following ports have been hard-coded in VHDL (see *sp6picor.vhd* for details):

Adresse [hex]	INPUT	OUTPUT	OUTPUT (const.) OUTPUTK
00	UART_STAT_port	– –	– –
01	UART_RX6_port	UART_TX6_port	RESET_UART_port
02	SWITCH_port	LED_port	– –
03	PUSHBUT_port	– –	– –
04	DIN4_port	DOUT4_port	– –

Port bit assignments:

UART_STAT_port (RD, ADDRESS= **00**):

7	6	5	4	3	2	1	0
0	0	RX full	RX half full	RX data present	TX full	TX half full	TX data present

UART_RX6_port (RD, ADDRESS= **01**):

7	6	5	4	3	2	1	0
RX7	RX6	RX5	RX4	RX3	RX2	RX1	RX0

SWITCH_port (RD, ADDRESS= **02**):

7	6	5	4	3	2	1	0
SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0

PUSHBUT_port (RD, ADDRESS= **03**): (updated, **PBU is PicoBlaze reset**)

7	6	5	4	3	2	1	0
0	0	0	0	PBL	PNC	PBR	PBD

DIN4_port (RD, ADDRESS= **04**, siehe Bild 1.9):

7	6	5	4	3	2	1	0
0	0	0	0	JB8	JB7	JB2	JB1

UART_TX6_port (WR, ADDRESS= **01**):

7	6	5	4	3	2	1	0
TX7	TX6	TX5	TX4	TX3	TX2	TX1	TX0

LED_port (WR, ADDRESS= **02**):

7	6	5	4	3	2	1	0
LED7	LED6	LED5	LED4	LED3	LED2	LED1	LED0

DOUT4_port routed through JB3, JB4, JB9, JB10 (WR, ADDRESS= **04**, see fig. 1.9):

7	6	5	4	3	2	1	0
X	X	X	X	X	b2h	b1h	ena

RESET_UART_port (WR CONSTANT, ADDRESS= **01**):

7	6	5	4	3	2	1	0
X	X	X	X	X	X	RX reset	TX reset

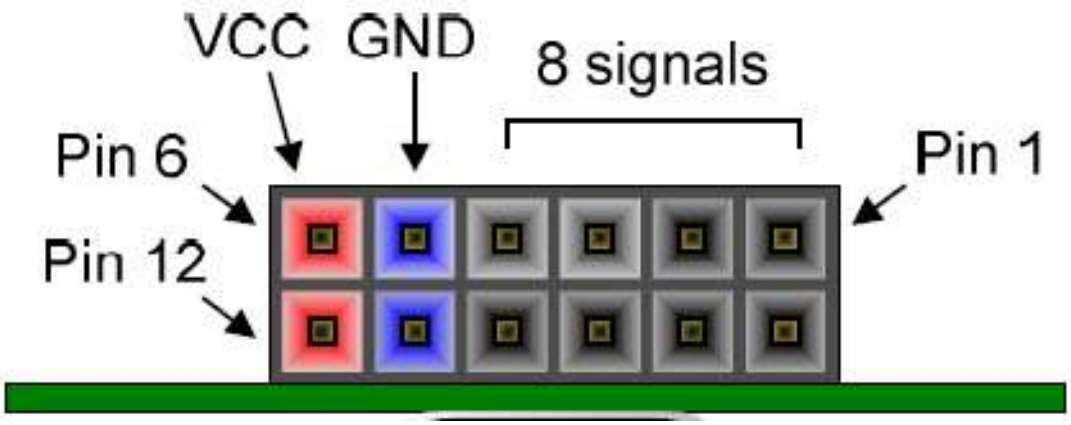


Figure 1.9: PMOD connector pin assignments on all FPGA boards

The pushbutton PBU (pushbutton "UP") is wired as a hardware "RESET" for PicoBlaze. This is the reason why it is not included in PUSHBUT_port.

Assemble your file with the assembler *kcpsm6.exe* . Copy the created VHDL file *pfirmware.vhd* into your ISE build directory.

- Build the bit file and inspect your design with the FPGA Editor in detail. Record where the program rom (block memory) is located!
- Configure the FPGA with *IMPACT* . Since the interface to the Atlys board is not included in the list of programming cables it needs top be specified as "third party device". The plugin name is "digilent_plugin". If the cable is not recognized the installation on your PC is incomplete and the plugin requires additional files.

3 Number Formats

Floating point numbers (single precision – 32 bits, double precision – 64 bits) are good for simulation and scientific computations. Real-time applications are dominated by fixed-point arithmetics. Fixed-point has better performance and better precision with a given word length. A 32 bit integer has 32 data bits while single precision has only 23 mantissa bits, which hold the precision information.

3.1 Signed and Unsigned Numbers

Integer numbers can be signed or unsigned. These data types differ in the interpretation of the MSB (most significant bit).

Figure 1.10: Bits of a n bit fixed point number

Unsigned number: the weight of the MSB is 2^{n-1} .
The possible Range is $0 \ldots 2^n - 1$.

Signed number: the weight of the MSB is -2^{n-1} (also known as 2's complement numbers).
The possible Range is $-2^{n-1} \ldots +2^{n-1} - 1$.

3.2 Fractional Numbers

Fixed point arithmetic is not limited to whole numbers (pure integers). To present fractions a decimal point may exist anywhere in the word.

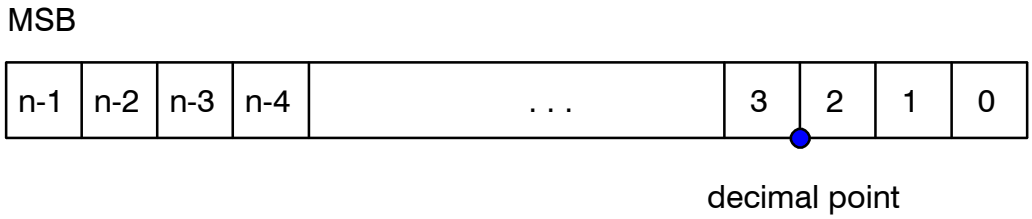


Figure 1.11: Integer with fractional bits (3 bits)

A fractional integer is denoted as **int_n.k** . The digit n is the number of total bits for the integer and k denotes the number of fractional bits. The parameter k can have any value $0 \le k \le n$. For instance the number format *int_8.3* has number range from

$$-16.000 \le i \le +15.875 .$$

The fractional number thus is scaled by the factor $2^{-k} = 2^{-3} = 0.125$.

An unsigned number *uint_8.3* has range

$$0.000 \le i \le +31.875 .$$

Fractional unsigned number: the weight of the MSB is 2^{n-k-1} .
The possible Range is $0 \ldots (2^n - 1)\, 2^{-k} = 0 \ldots 2^{n-k} - 2^{-k}$.

Fractional signed number: the weight of the MSB is -2^{n-k-1} .
The possible Range is $-2^{n-k-1} \ldots +2^{n-k-1} - 2^{-k}$.

3.2.1 Values Less Than One

Digital signal processing becomes often efficient, if all numbers are made less or equal one. This easily avoids overflows by multiplications, since the result of two of such numbers is less or equal one again.

From the notation of section 3.2 follows that these numbers are of the type *int_n.n*-1 or *uint_n.n*, respectively.

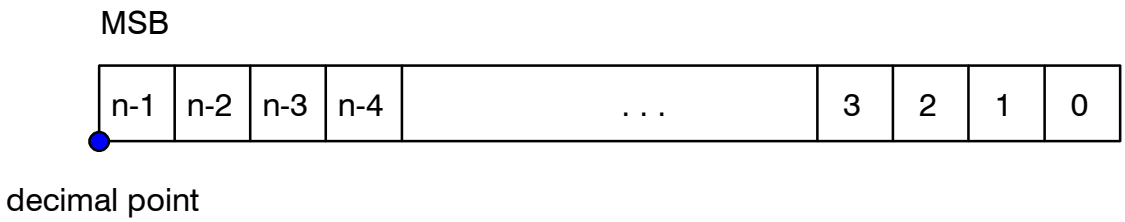


Figure 1.12: Unsigned integer less or equal one

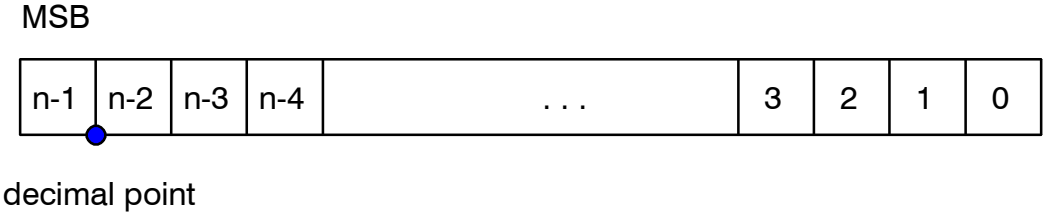


Figure 1.13: Signed integer less or equal one

The number format *uint_6.6* has number range from
$0.000 \le i \le +0.984375$.
This fractional number thus is scaled by the factor $2^{-k} = 2^{-6} = 0.015625$.

A signed number *int_6.5* has range
$-1.000 \le i \le +0.96875$.
The scaling factor is determined by k: $2^{-k} = 2^{-5} = 0.03125$.

> The scaling factor is also known as "precision".

3.3 Fixed Point Arithmetic

Floating point arithmetic causes a loss of precision because the result occupies always the same number of bits. Fixed point is lossless if the appropriate word length for the operation is provided. In order to prevent overflow the addition (or subtraction) of to n bit numbers require a $n+1$ bit result.

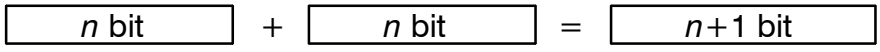


Figure 1.14: Safe word length for addition (subtract)

For multiplication the result must hold twice the word size to safely keep the product.

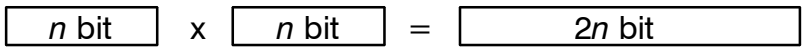


Figure 1.15: Safe word length for multiplication

However, if the data range can be restricted to a defined magnitude, the result can be stored with loss of precision in a smaller word.

In many applications the word size for the result is limited. Some fractional bits are then simply truncated ("thrown away") or rounded (better but more computational effort) to the required fractional length.

3.3.1 Fixed Point Sum or Difference Using Fractional Bits

For all additions and subtractions the number of fractional bits must be equal. If this is not true, integer fractional number need to be scaled until the number of fractional bits becomes identical. This is exactly what happens with every floating point add or subtract. The number with more fractional digits determines the scaling.

To prevent a possible overflow, the number of required digits is increased by one. All examples refer to signed numbers. Unsigned integers differ only in the weight of the MSB.

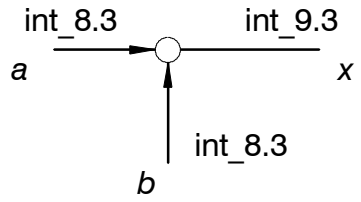


Figure 1.16: Addition of 8 bit numbers with 3 fractional bits

Since the addition (or subtraction) of two 8 bit integers may result in a 9 bit sum the lossless computation requires a different format (as shown in example in fig. 1.16).

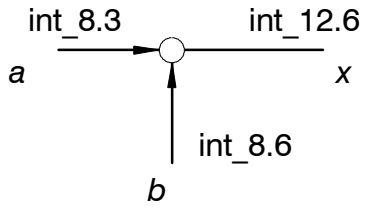


Figure 1.17: Addition of 8 bit numbers with different fractional bits

First the number *a* needs to be scaled to the format of *b*. The number a becomes the format int_11.6. Since the sum of *a* and *b* can cause a possible overflow the result should be stored in int_12.6 format.

The minimum number of required binary digits can be exploited in FPGAs. On a microcontroller the word length is restricted to multiples of 8 bits. The same operation on a microcontroller with fixed word lengths would look like fig. 1.18.

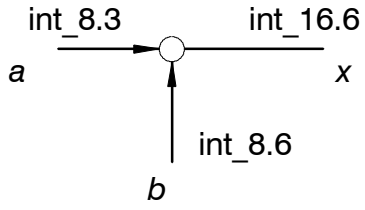


Figure 1.18: Addition of 8 bit numbers with different fractional bits on a microcontroller

Of course, it is safe to “overestimate” the required word length.

3.3.2 Fixed Point Multiplication Using Fractional Bits

As outlined at the beginning section 3.3 the number of required digits to hold a product is the sum of the bits of multiplier and multiplicand. To provide lossless computation the number of fractional bits in the product is the sum of fractional bits of multiplicand and multiplier.

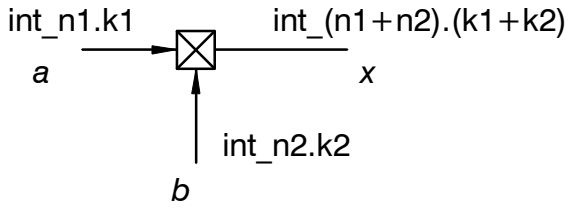


Figure 1.19: Lossless multiplication of *unsigned* fractional integers

For *signed* integers due to the fact that there are two sign bits, the number of digits for the product is reduced by one.

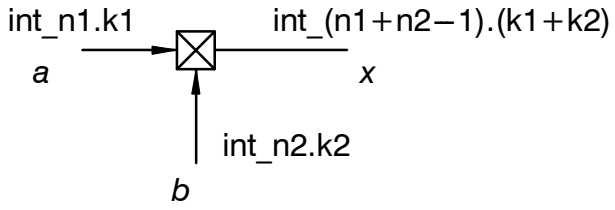


Figure 1.20: Lossless multiplication of *signed* fractional integers

It it obvious that lossless computation increases the number of required digits.

> In any practical application the number of digits for digital signal processing in limited. In order to keep a maximum word length, some bits must be "thrown away". This causes some loss of precision. It depends on the application which precision is required for the final result.

3.3.3 Example: PI Controller

The computation of the PI controller is lossless with respect to the provided precision. At the end a down conversion provides the same precision for the output y as for the input u.

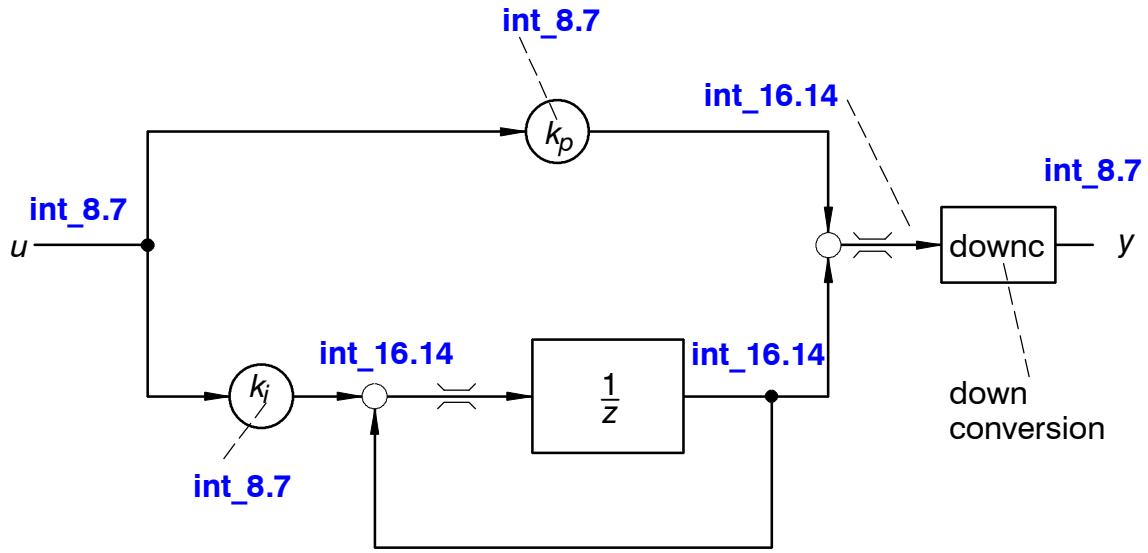


Figure 1.21: Number format example for a PI controller on an 8 bit microcontroller

The down conversion is illustrated in fig. 1.22. The bits 14, 6..0 are removed from the final result.

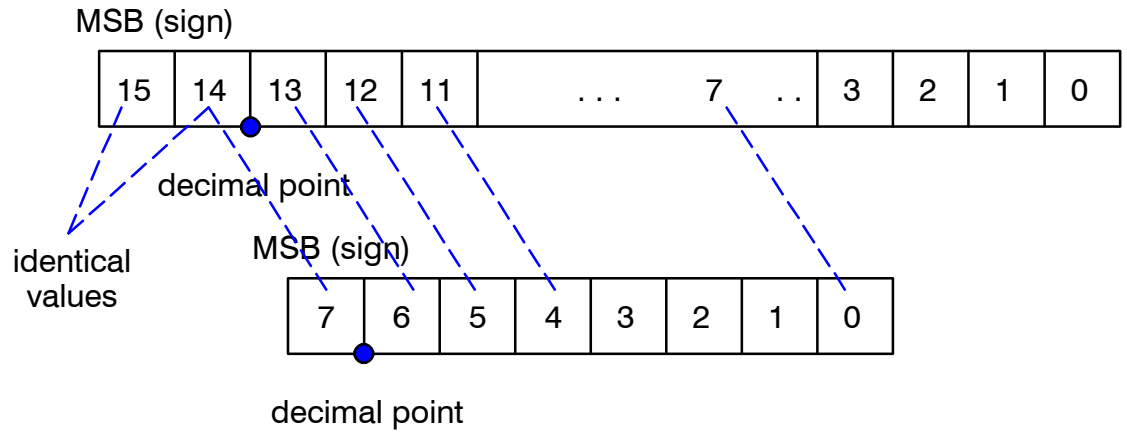


Figure 1.22: Down conversion from int_16.14 to int_8.7

Most software development systems do not recognize the proper display of values in fractional integer notation (except Matlab/Simulink with the Fixed-point-blockset). Therefore, it is helpful to use the formulas to convert integer to fractional notation with respect to the given format.

Conversion from fractional notation **int_n.k** to integer (with a possible loss of precision)

$$x_{\text{int}} = \text{round}\left(x_{\text{fract}}\, 2^{k}\right). \tag{1.1}$$

Conversion from integer to fractional value **int_n.k** (exact)

$$x_{\text{fract}} = x_{\text{int}}\, 2^{-k}. \tag{1.2}$$

For example, in int_8.5 format then number $\pi = 3.141592654$ is expressed as the integer value [according to (1.1)]

$$x_{\text{int}} = \text{round}\left(\pi\, 2^5\right) = \text{round}\,(100.531) = 101\,. \tag{1.3}$$

On the other hand, the integer value 101 corresponds to the fractional value [according to (1.2)]

$$x_{\text{fract}} = 101\ 2^{-5} = 3.1563\,, \tag{1.4}$$

causing an error of $+0.4666\ \%$.

Using 16 bits with the format int_16.13 we obtain $x_{int} = 25{,}736$ and thus xfract = 3.1416016. The error drops to $x_{err} = +0.00028\%$ which seems to be sufficient for most embedded applications.

With IEEE single precision float with a mantissa of 23 bits the error for π is in the same range $x_{err} = +0.0002338\%$.

Lab #02:

3.4 Discrete PI Controller Programming

The PicoBlaze microcontroller should compute a discrete PI controller with 8 and 16 bit precision. The block diagram within Matlab/Simulink™ shows the structure.

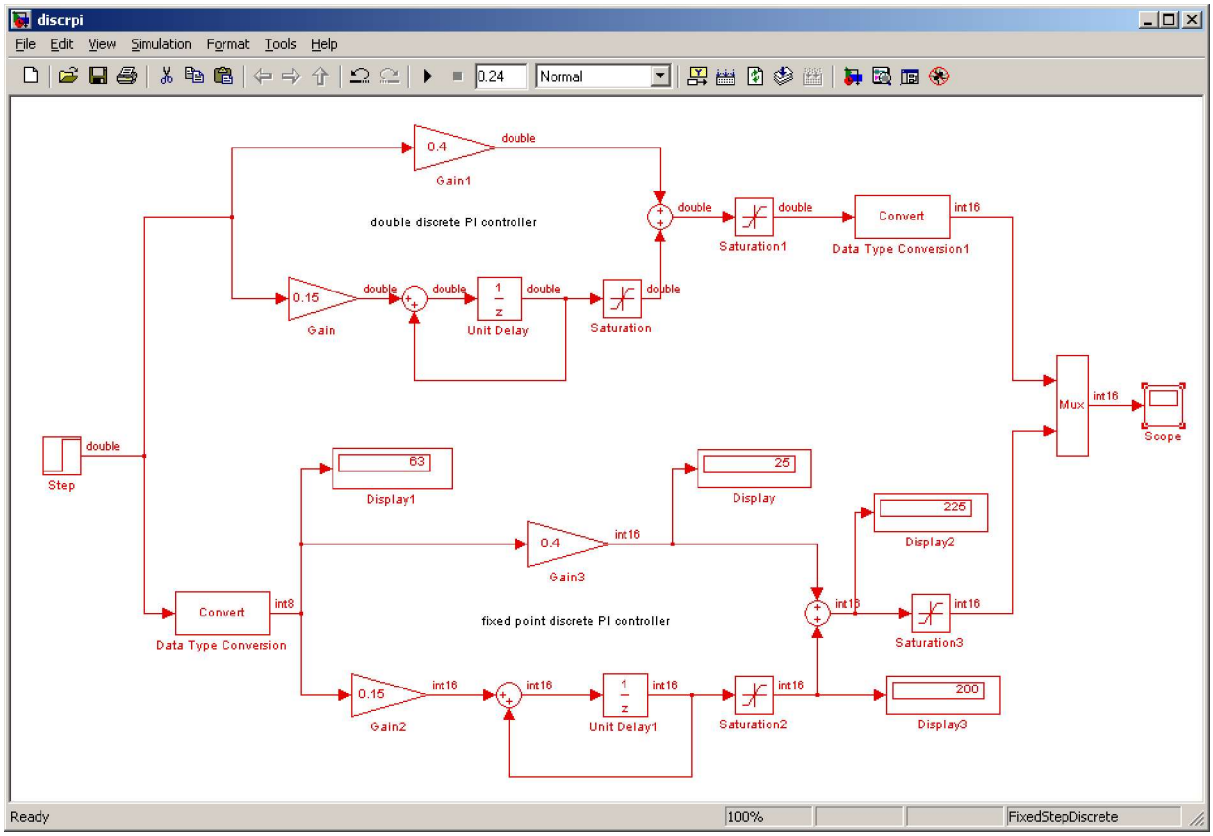


Figure 1.23: Structure of a discrete PI controller

The red color indicates the discrete components within Simulink. The lower part is computed in fixed point arithmetic.

Full fractional arithmetic requires a “fixed point blockset” license which is not available here.

- Simulate the system and explain the differences in the step response for the double floating point and the fixed point calculations.
- The PI controller requires a multiplication which is not part of the PicoBlaze instruction set. Therefore, a multiplication subroutine becomes necessary. Write and test the routine with the provided test program (verify results by debugger):

```
; --
; Signed 8x8 multiplication for PicoBlaze
; reference design
;--

cstart:   LOAD      s8, 3F              ; multiplicant =   63
          LOAD      s9, 7B              ; multiplier   =  123
          CALL      mult8x8s            ; ==> result = 1E45 (+7749)
          LOAD      s8, C1              ; multiplicant =  -63
          LOAD      s9, 7B              ; multiplier   =  123
          CALL      mult8x8s            ; ==> result =  E1BB (-7749)
          LOAD      s8, 3F              ; multiplicant =   63
          LOAD      s9, 85              ; multiplier   = -123
          CALL      mult8x8s            ; ==> result =  E1BB (-7749)
          LOAD      s8, C1              ; multiplicant =  -63
          LOAD      s9, 85              ; multiplier   = -123
          CALL      mult8x8s            ; ==> result =  1E45 (+7749)
forever:  JUMP      forever

; ------------------------------------------------------------------
; mult8x8s: signed 8 x 8 bit multiplication
;    s8 = multiplicant (not destroyed)
;    s9 = multiplicator (not destroyed)
;    [sB, sA] = result (16 bits)
;    [sD, sC] = temp variable (holds shifted multiplicant)
;    sE = mask
; ------------------------------------------------------------------
mult8x8s: .
          .
          .
          RETURN
```

The signed multiplication algorithm requires shifts and conditional add and one subtract. This is due to the fact that the MSB has a negative weight. The figure below shows the weights for every bit in an 8 bit signed integer.

MSB (neagtive weight!)

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Here is an example for a signed 4x4 bit multiplication of (–3) x (–5) give an 8 bit result:

```
 1 1 0 1 x 1 0 1 1      (-3) x (-5)
---------------------
  0 0 0 0  0 0 0 0      accumulated result
+ 1 1 1 1  1 1 0 1      multiplicant sign extended
-----------------
  1 1 1 1  1 1 0 1      result = -3
+ 1 1 1 1  1 0 1        shift multiplicant by 1
-----------------
  1 1 1 1  0 1 1 1      result = -9
+ 0 0 0 1  1            shift multiplicant by 3 (negated due to sign)
-----------------------
(1) 0 0 0 0  1 1 1 1    8 bit result = 15,  Carry  is ignored
```

- Write a PicoBlaze assembler program that computes the PI algorithm in fixed point. Results can be sent to the serial port (to be displayed on the PC). The sampling period is achieved by a 10ms interrupt. The timer is already connected in hardware. Just enable the interrupts with the **ENABLE INTERRUPT** instruction after providing an interrupt service routine.

- In order to display numbers (in ASCII notation for the terminal output) a division routine is useful. The division algorithm is shown below.

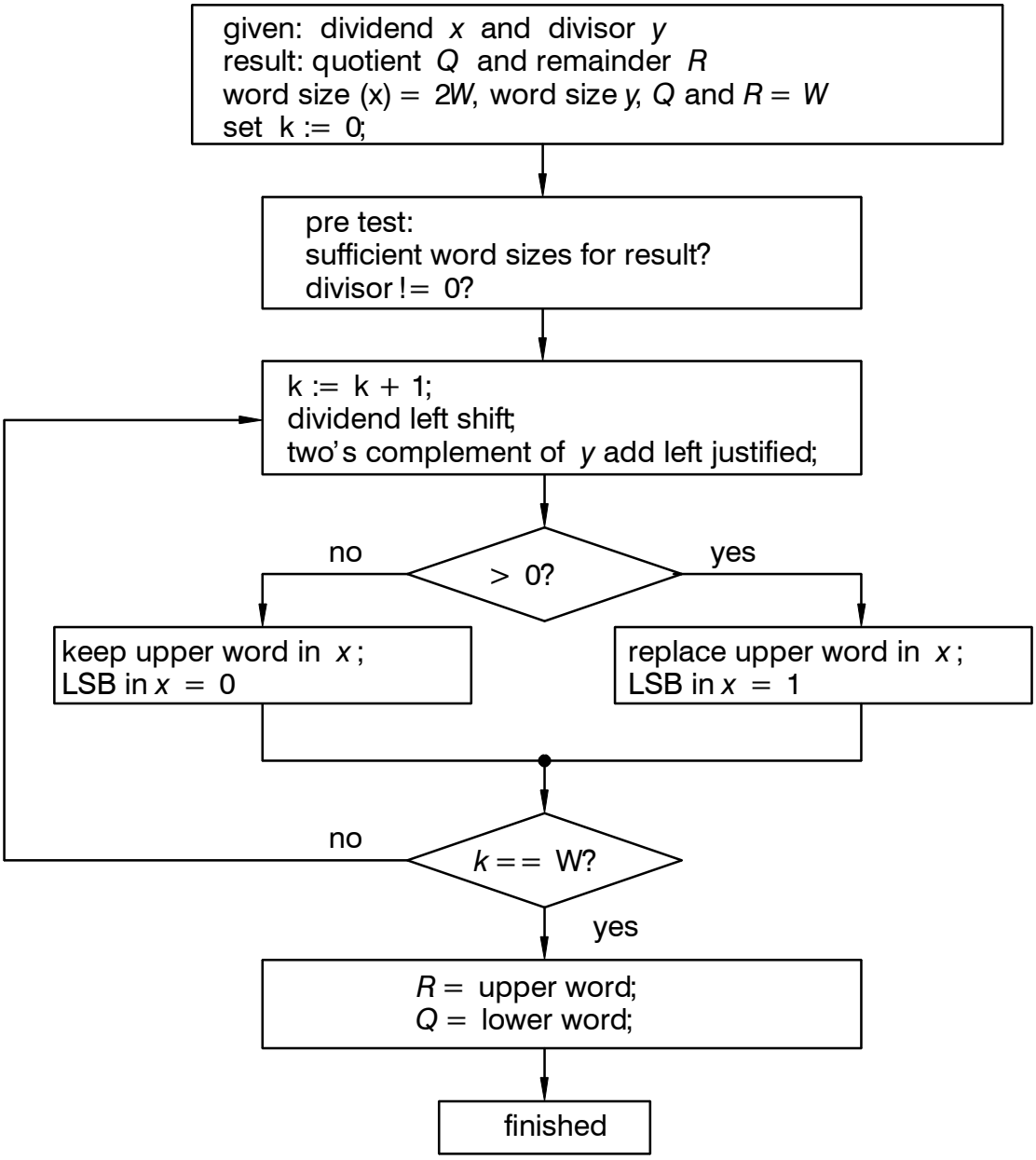


Figure 1.24: Division algorithm for unsigned numbers (shift-restore algorithm)

The following table shows a sample division (44 divided by 7)

```
     0 0 1 0  1 1 0 0    (44)
--------------------
     0 1 0 1  1 0 0 x    1. shift left of dividend
+    1 0 0 1             add of two´s complement of 7
 -------------------
     1 1 1 0             result negative => LSB = 0
--------------------
     0 1 0 1  1 0 0 0    => restore, LSB = 0
     1 0 1 1  0 0 0 x    2. shift left
+    1 0 0 1
--------------------
```

```
     0 1 0 0            result positive => LSB = 1
--------------------
     0 1 0 0  0 0 0 1   => replace, LSB = 1
     1 0 0 0  0 0 1 x   3. shift left
+    1 0 0 1
--------------------
     0 0 0 1            result positive  => LSB = 1
--------------------
     0 0 0 1  0 0 1 1   => replace, LSB = 1
     0 0 1 0  0 1 1 x   4. shift left
     1 0 0 1
--------------------
     1 0 1 1            result negative  => LSB = 0
--------------------
     0 0 1 0  0 1 1 0   final result => R = 2, Q = 6
====================
```

3.4.1 Assembly and Program Upload

Assembly and upload require setting the location of the source code folder (top left) and the name of the source file (top right) without the mandatory extension ".psm". In the example below the file `picotst.psm` must be present in the folder `C:\PicoDev`.

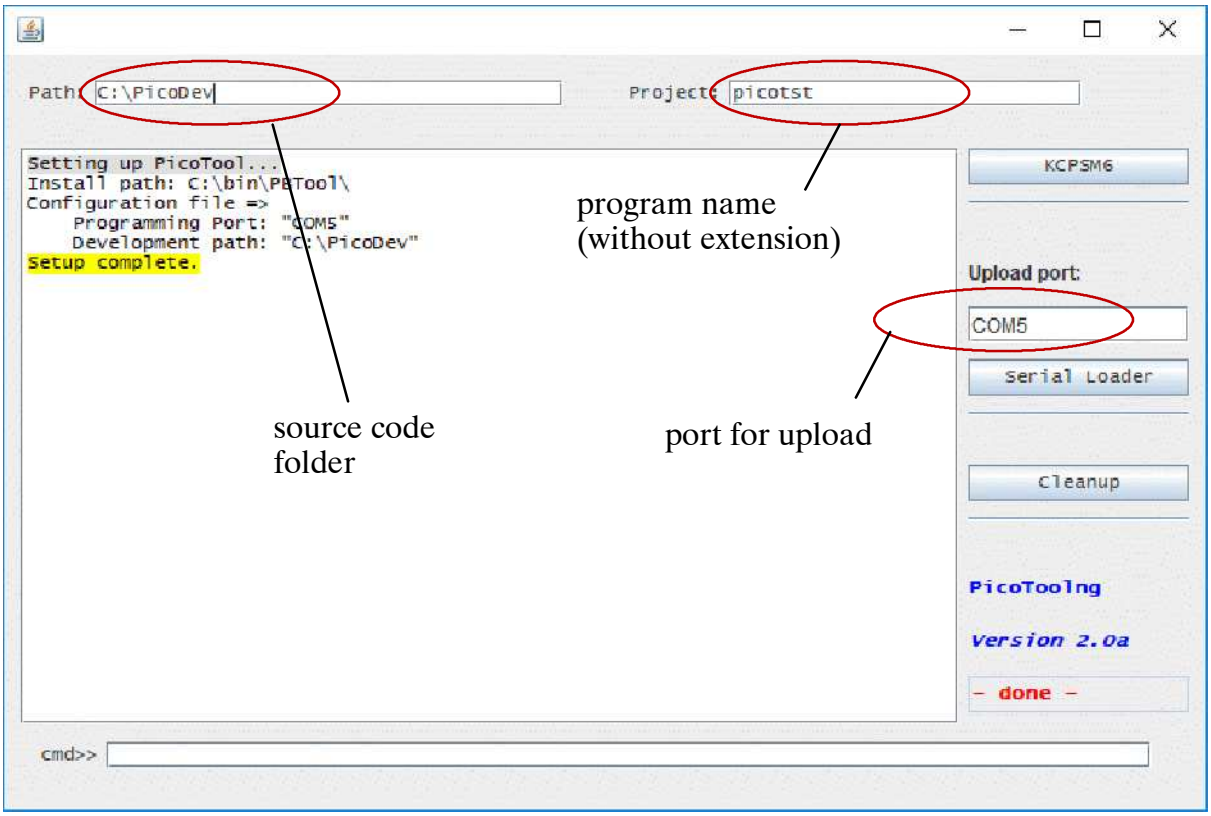


Figure 1.25: PicoToolng Java Application

Clicking the `KCPSM6` button will invoke the assembler. The assembler window will close automatically when there are no errors. The main window will contain the assembler listing.

In case of errors the assembler message window remains open. After correcting errors in source code the assembler window must be closed manually with the letter `q`.

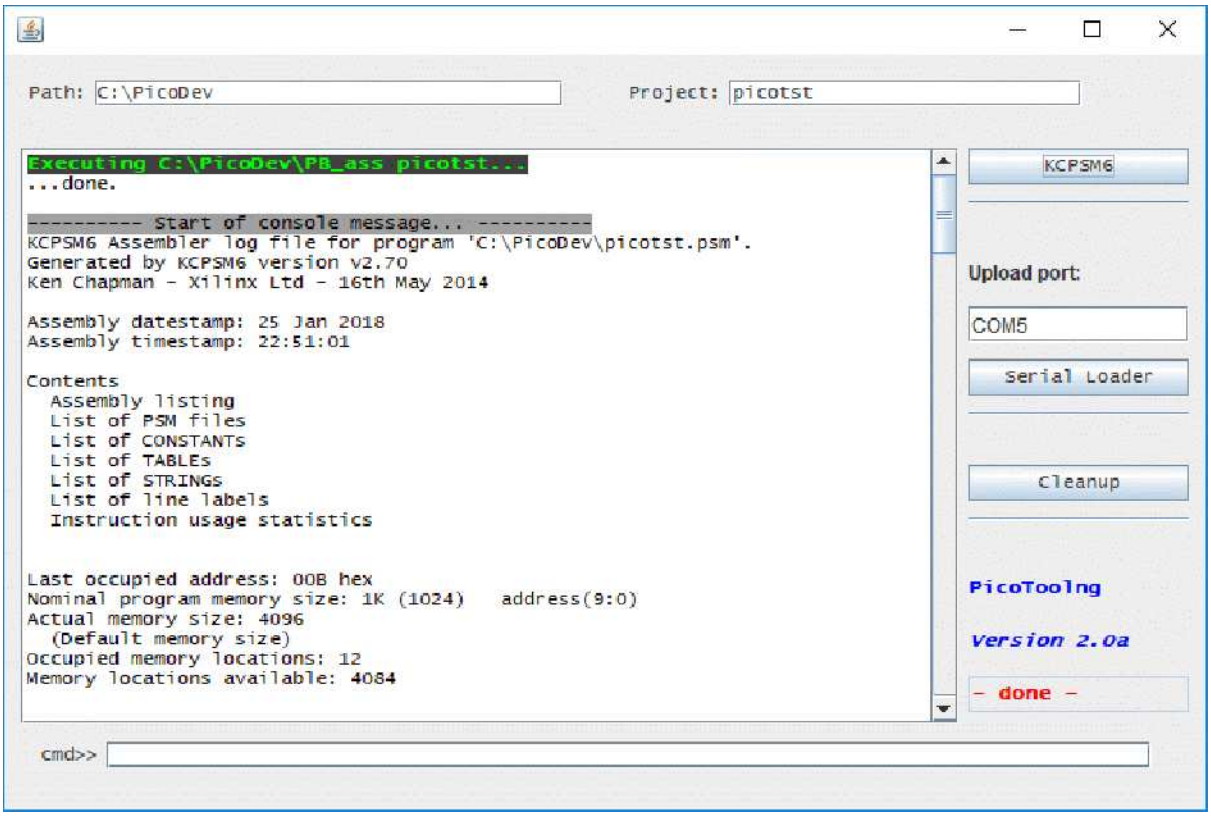


Figure 1.26: Successful assembly

For program loading the serial port for upload needs to be specified (can be seen from the "Windows Device Manager" under Ports).

A click on the `Serial Loader` button will put the PicoBlaze into reset, it overwrites the program memory, and releases reset (see fig. 1.27).

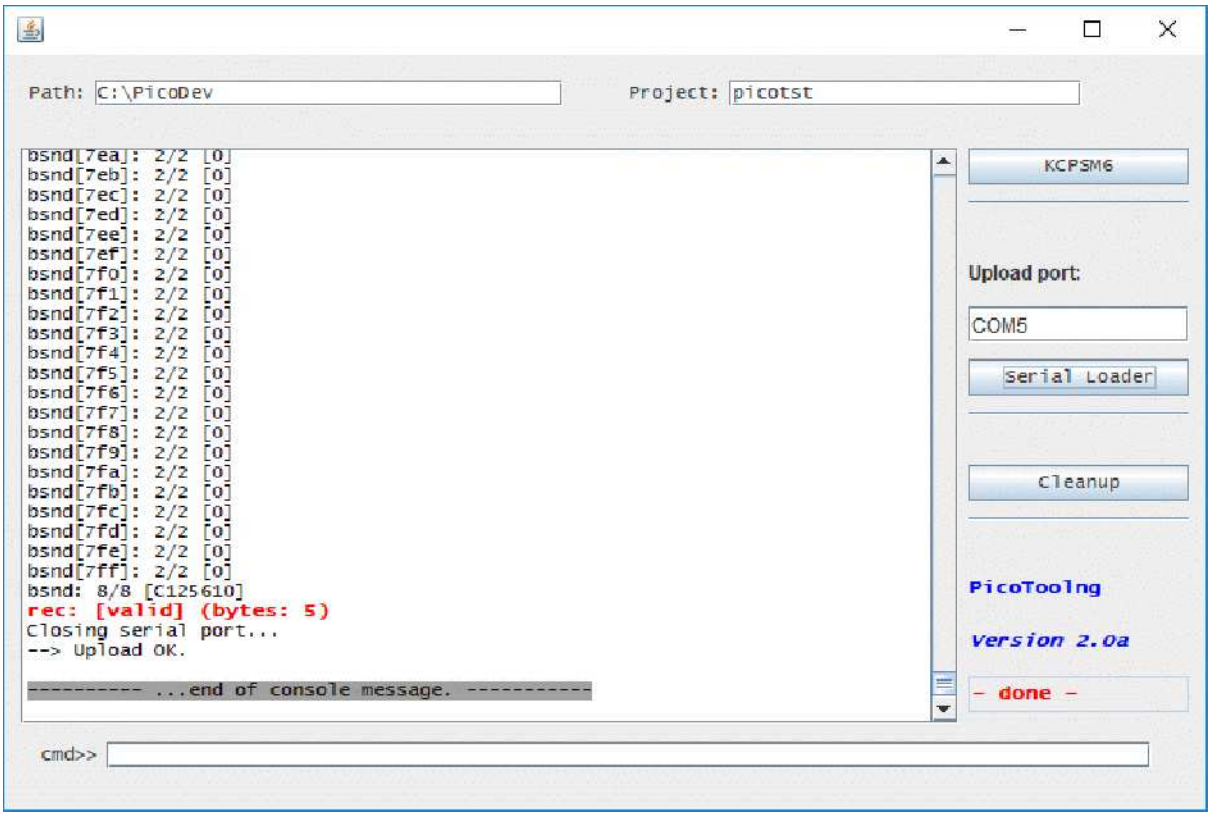


Figure 1.27: Successful upload to PicoBlaze

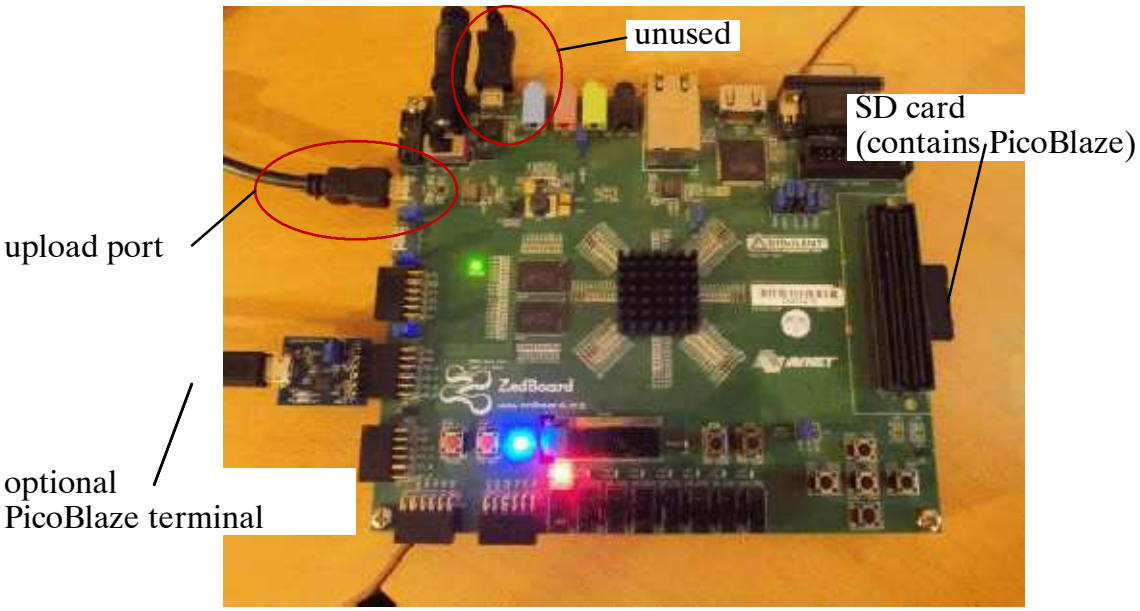


Figure 1.28: ZedBoard with running PicoBlaze

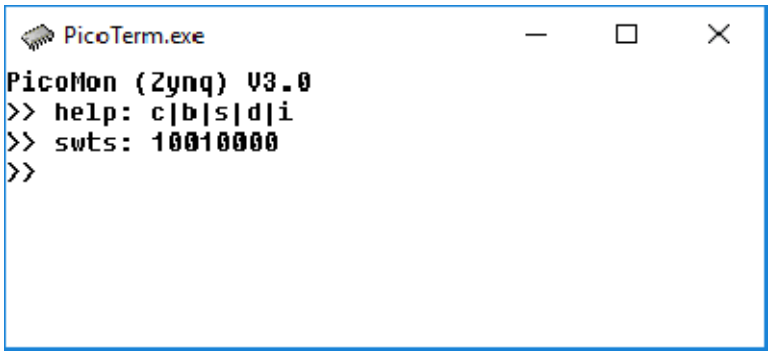


Figure 1.29: PicoBlaze says “Hello” on PicoTerm (initial Program on boot)

***

4 Embedded System With 32 Bit Microcontroller MicroBlaze™

Embedded System are designed to perform a single function. Almost all industrial, scientific, communication and medical devices are nowadays embedded systems. These systems are characterized by following properties:

- Strong real-time capabilities. The embedded controller must react to events and signals in a well defined maximum response time. The time requirements are typically in the nanoseconds or microseconds range.
- Hardware and software **coexistence**. Time-critical components are implemented in (programmable) hardware, slow or complex algorithms are carried out in software. The standard programing language for embedded systems is “C” or “C++”. New challenging is the use of JAVA for embedded programing.
- Small probably SoC solutions are inexpensive, efficient and flexible.
- Other important constraints are
  – power consumption
  – cost restriction
  – safety and security requirements
- Network capability
- Tailored integrated user interface

The SoC solution is the most flexible, cost efficient, reliable and offers highest performance. SoCs can be created using FPGA or ASIC technology. ASICs are economical only for high volume applications. Alternatives are implementations on microcontrollers (the majority of solutions today) or DSPs.

Using FPGAs is in almost all cases optimal, however, their use requires qualified experts. The expected development for embedded systems is outlined below.

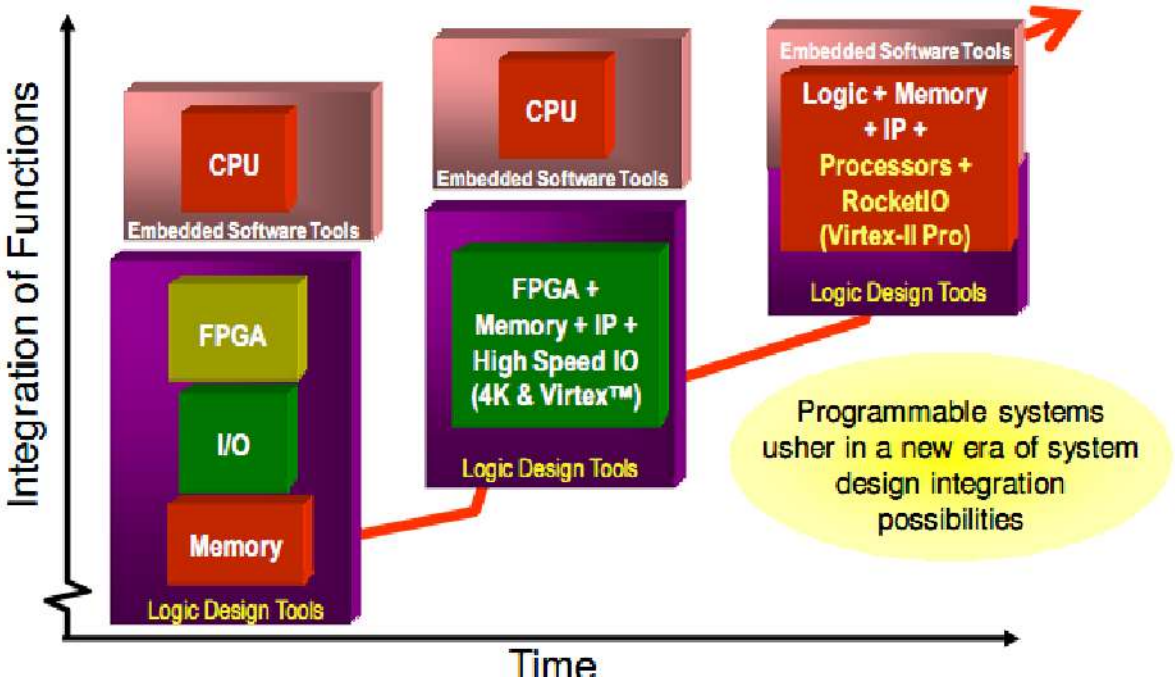


Figure 1.30: Embedded Systems development future © 2010 by Xilinx Inc.

4.1 Development Tasks

The development comprises the following tasks:

- Design of hardware logic
  – processor selection (programmable microcontroller, IP controller, included hardware
    microcontroller (Sparc, ARM, PowerPC or others)
- Implement user logic (interfaces, network, etc.) using VHDL or IP blocks
- Optionally install a real-time operating system (i.e. RTLinux, VxWorks, or others) or write your own scheduler for real-time tasks and interrupts
- Write drivers for systems devices
- Code user software functions

The development tasks are not straight-ahead, e.g. FPGA solutions have the flexibility to change all components without exchanging the target hardware.

4.2 MicroBlaze 32 Bit Microcontroller Integration

The MicroBlaze™ microcontroller is a high performance RISC type CPU for programmable logic. It is not available as source code, but it can be configured to meet the

specification for a given application. the configuration and the attachment of user logic is carried out in Xilinx EDK (Embedded Design Kit).

The user logic is usually attached to a bus system which is called PLB (Processor Local Bus, copyrighted by IBM). This bus is used also for PowerPC™ CPUs. Due to a shift to ARM type microcontroller new designs and newer microcontrollers are connected using the AXI4 bus (Advanced eXtensible Interface, developed by ARM Ltd.).

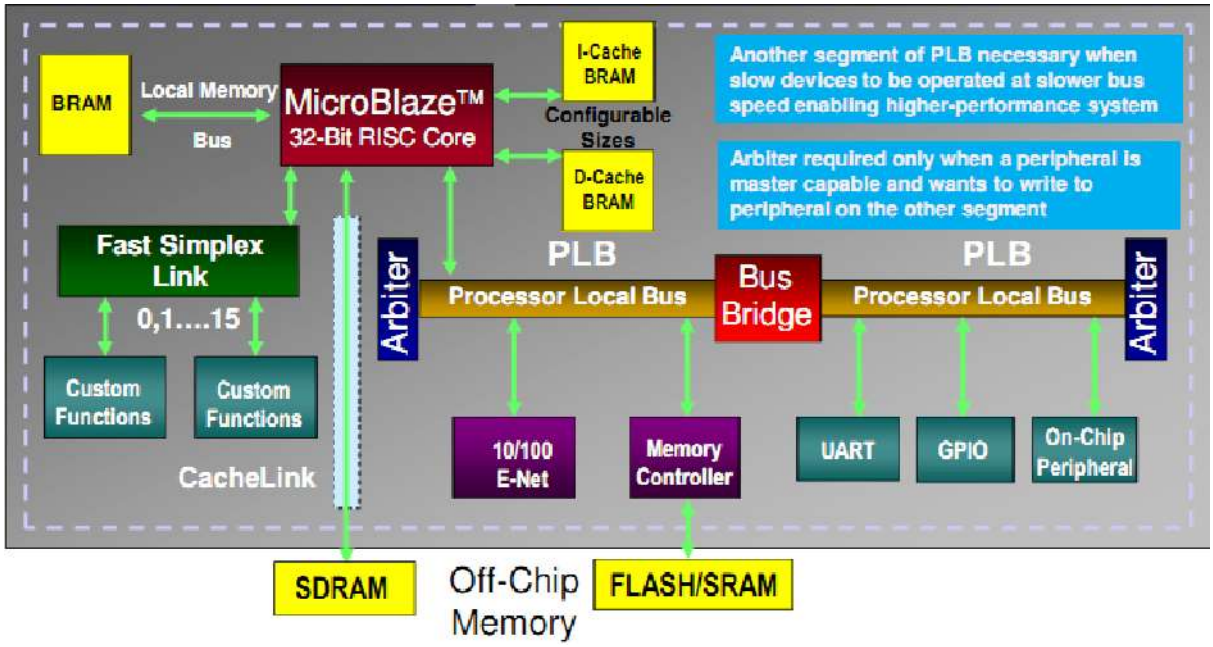


Figure 1.31: MicroBlaze CPU with PLB

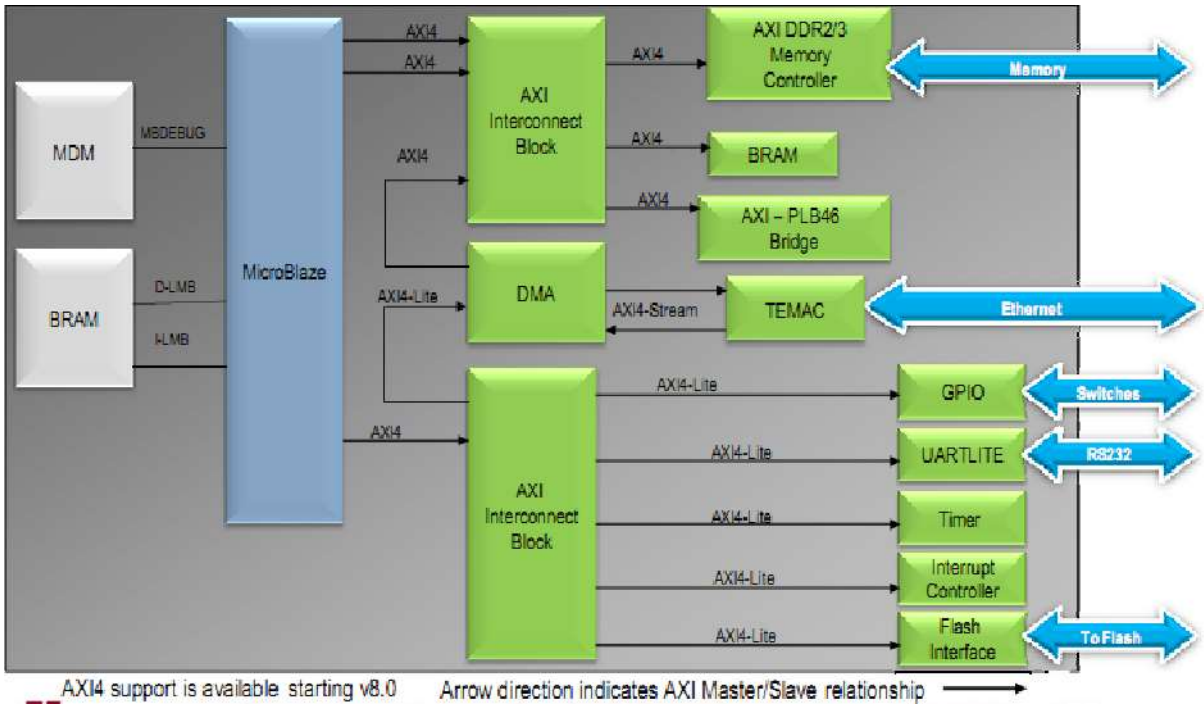


Figure 1.32: MicroBlaze CPU with AXI bus

5 Embedded System with 32 Bit Microcontroller ARM Cortex A9 (ZYNQ™)

Many embedded application require a processing unit. In theses cases an integrated processor inside of an FPGA offers several advantages over a programmable (softcore) CPU. This SoC solution allow to create almost any digital (control) system for industrial, scientific or medical devices including network, audio and image processing.

he tremendous success of these devices led to a low price; newer devices with several CPUs (quad-core ARM Cortex) are available now.

For ARM a highly optimized compiler (GCC) and all corresponding utilities are available for cross development. This includes several operating systems. The dominating OS is Linux (including similar operating systems like Android™). The market share of Linux is approximately 55%, the rest is shared by a number of OS'es. Non of them has a significant market share.

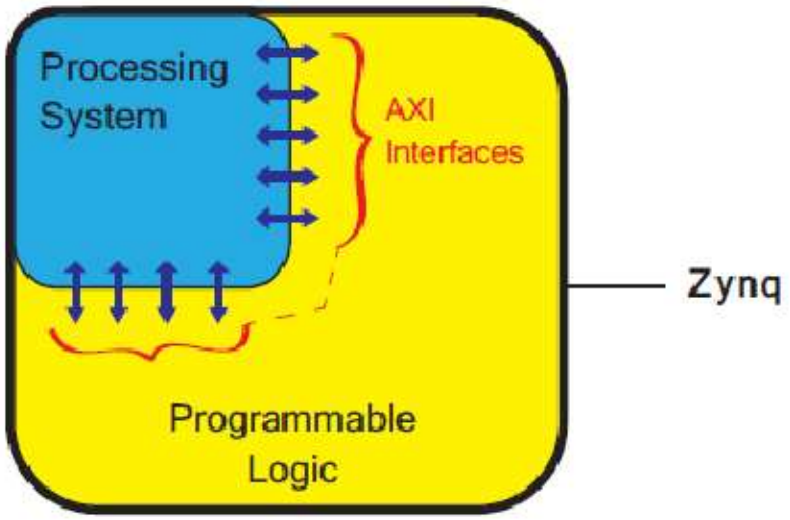


Figure 1.33: Zynq Architecture [3]

The ARM processor has an industrial standard AXI interface (Advanced eXtensible Interface) to connect with programmable logic blocks (IPs = intellectual property blocks) as well as many signal to communicate directly with the programmable logic (PL).

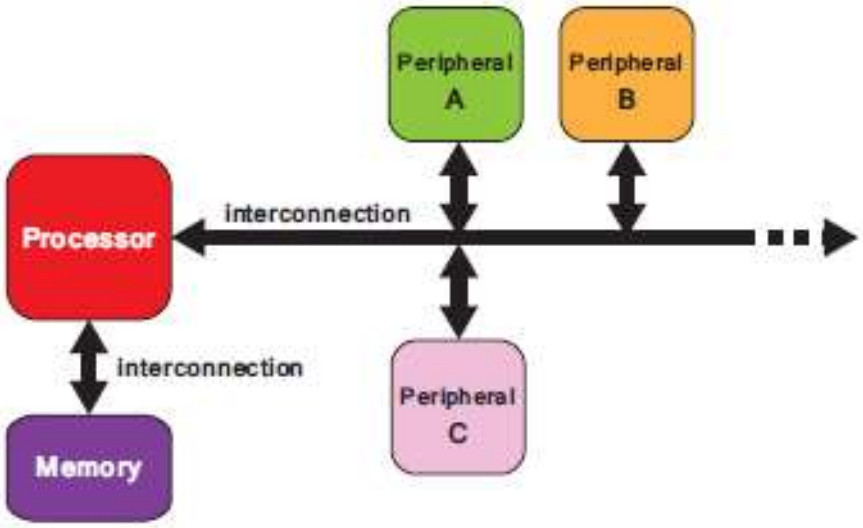


Figure 1.34: Adding hardware to the processing system [3]

Hardware (denoted as peripherals) can be added as AXI devices to the processor. AXI is similar to a bus system but allows also simultaneous transfers between two members (similar to a cross bar switch).

Depending on the requirements of the peripherals a suitable AXI bus type can be used (AXI4, AXI4-Stream or AXI4-Lite). AXI4-Lite is for low data volume device which do not require burst transfers or DMA.

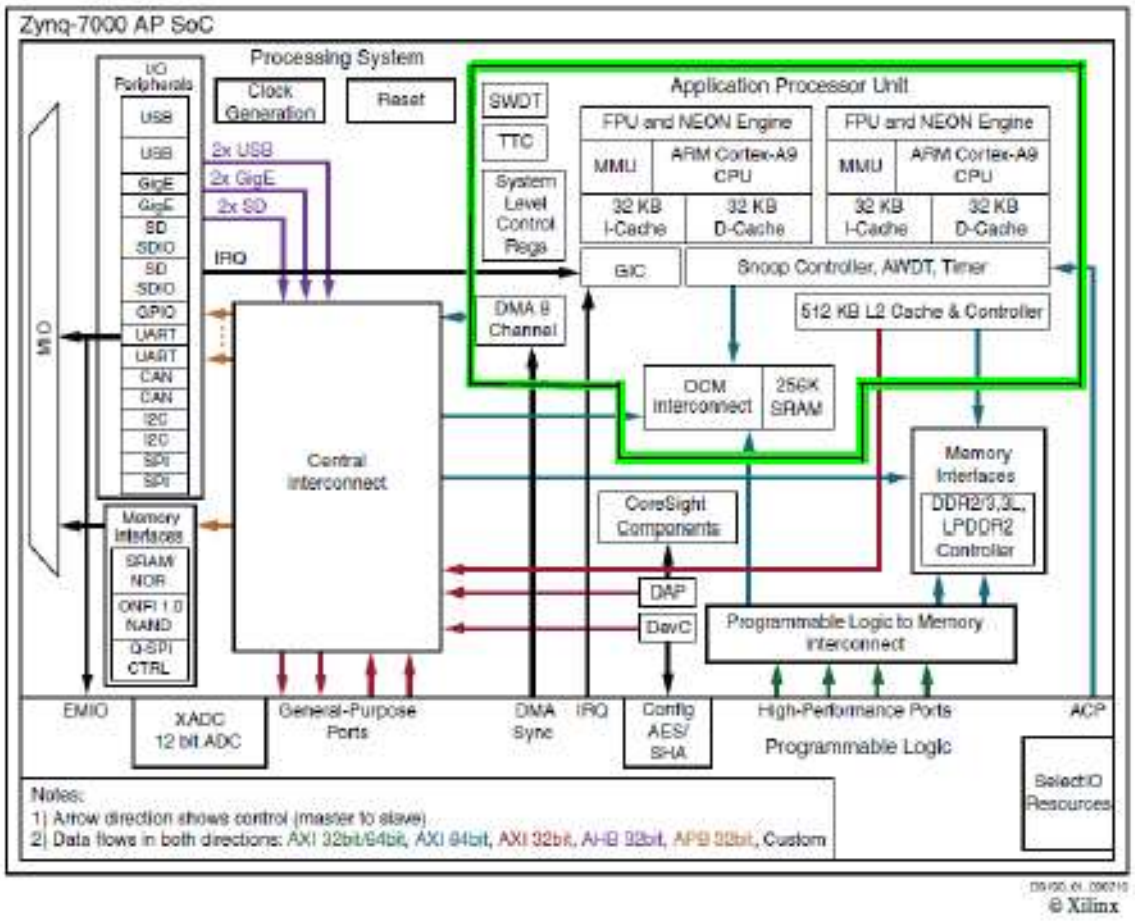


Figure 1.35: ARM core configuration

Since the ARM processor is fixed, i.e. it is not programmable, it can only be configured. The corresponding block diagram is shown in fig. 1.35. The CPU (without integrated peripherals) is highlighted in green. It is denoted as APU (application processing unit).

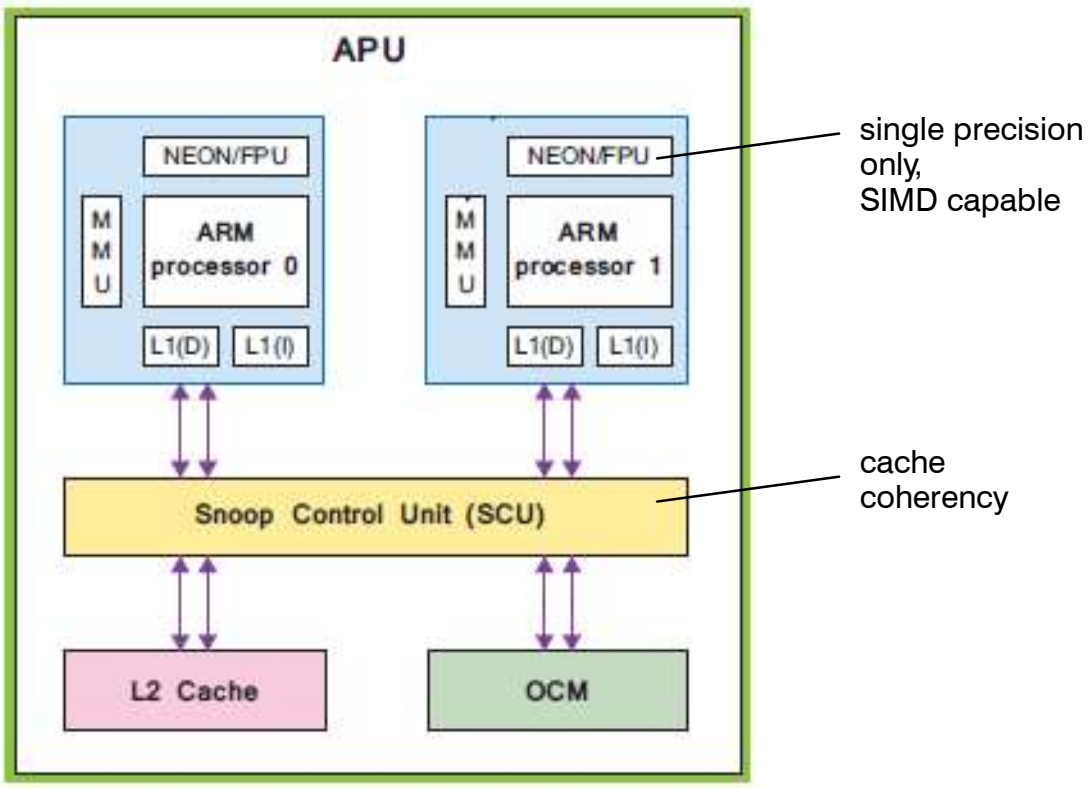


Figure 1.36: Detailed view of the application processing unit [3]

The dual cores share the level-2 cache (level-1 cache is private for each core) and the on-chip memory (OCM, 256 KByte).

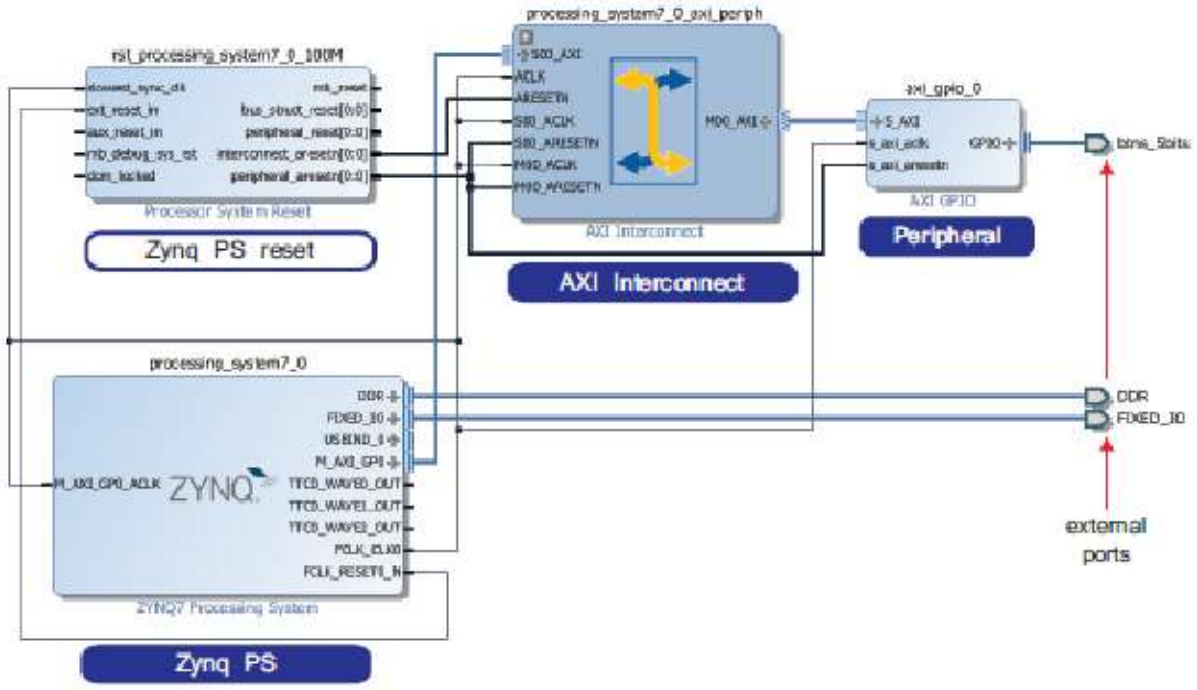


Figure 1.37: Configuring a SoC (Vivado) [3]

ARM core and IPs integration is carried out by making connections is the Vivado block diagram. The central element is the Zynq block.

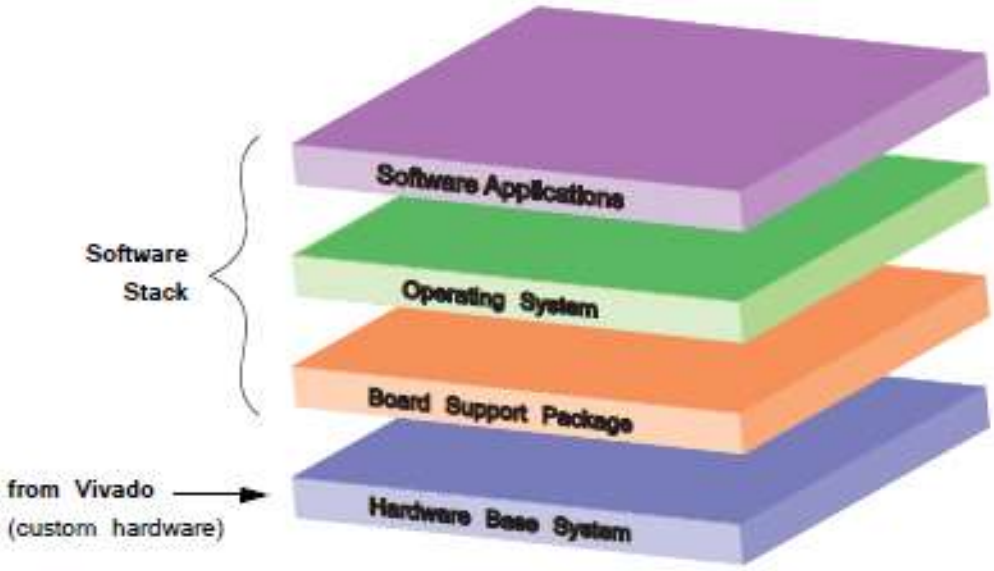


Figure 1.38: Hardware and software stack [3]

Software development is strictly separated into the XSDK (software development kit). SDK is Eclipse based. The GCC tool suite (as cross development environment) performs compilation, linking and debugging.

Start Address	Size	Description
0x0000_0000	1024 MB	DDR DRAM and OCM
0x4000_0000	1024 MB	PL AXI slave port 0
0x8000_0000	1024 MB	PL AXI slave port 1
0xB000_0000	256 MB	IOP devices
0xF000_0000	128 MB	Reserved
0xF800_0000	32 MB	Programmable registers access via AMBA APB
0xFA00_0000	32 MB	Reserved
0xFC00_0000	64 MB — 256 KB	Quad-SPI linear address base (except top 256 KB which is OCM), 64 MB reserved, only 32 MB is currently supported
0xFFFC_0000	256 KB	OCM when mapped to high address space

Figure 1.39: Memory Map for Zynq-7000 [3]

All hardware is memory mapped, i.e. no port address space exist for accessing configuration registers or AXI4 devices. The DDR3 memory in ZedBoard is 512MB. Therefore the range of external DDR memory ranges from 0x0000_0000 to **0x2000_0000**.

The total address space is 4GByte (2^ 32 = 0x1_0000_0000). The on-chip memory block is mapped to the high memory address otherwise it would conflict with DDR memory.

5.1 Interrupt System

The interrupt systems has dedicated IRQ IDs for all possible interrupt sources. It exist private interrupts for each core as well as interrupts which can be routed to one or both cores. the structure is shown below.

The core itself has only one IRQ (interrupt request input) and one FIQ (fast interrupt request input). Any FIQ immediately blocks all IRQs while the reverse in not true. The FIQ therefore can be compared to an NMI is other systems, although FIQs are maskerable. FIQs driver often require assembly programming (not possible to write the driver in “C”).

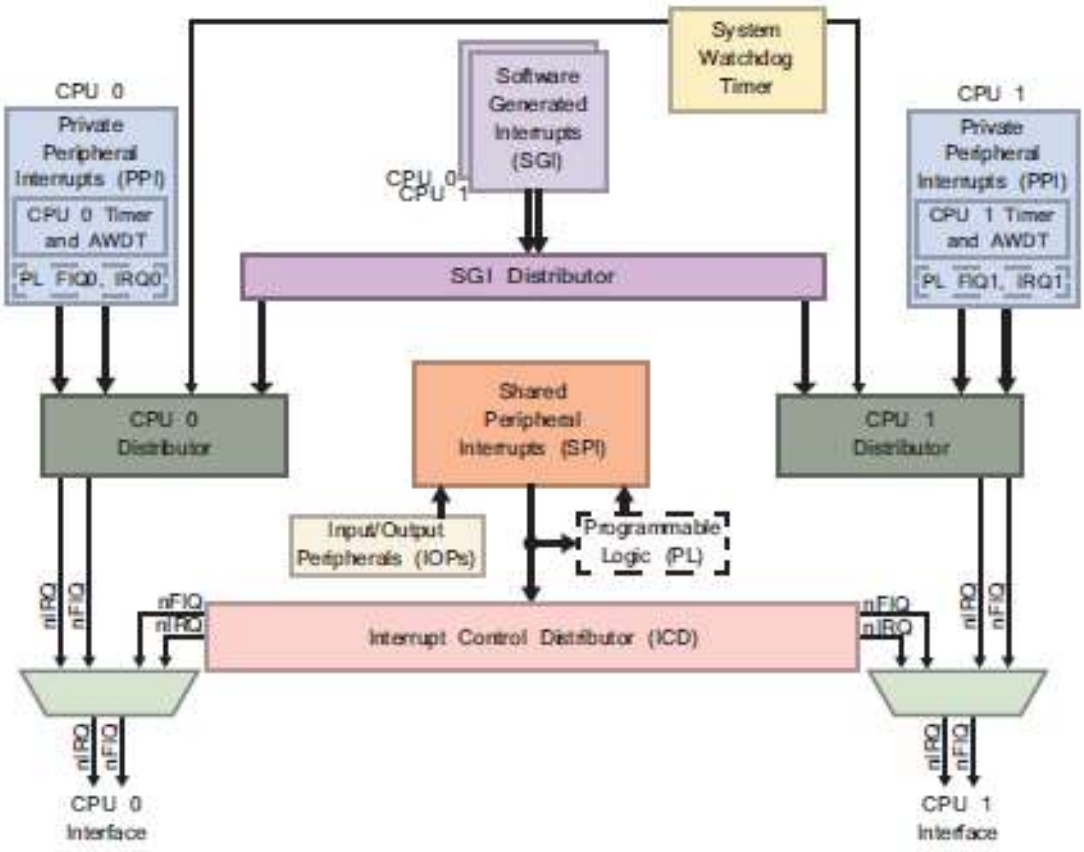


Figure 1.40: Zynq-7000 interrupt structure [3]

The private peripheral interrupt (PPIs) sources and numbers are listed below [3].

IRQ ID #	Name	PPI #	Type	Description
26:16	Reserved	-----	----	Reserved
27	Global Timer	0	Rising Edge	Global timer
28	nFIQ	1	Active Low level in PL (Active High at PS-PL interface)	Fast interrupt signal from the PL CPU0: IRQF2P[18] CPU1: IRQF2P[19]
29	CPU Private Timer	2	Rising Edge	Interrupt from private CPU timer
30	AWDT{0, 1}	3	Rising edge	Private watchdog timer for each CPU
31	nIRQ	4	Active Low level in PL (Active High at PS-PL interface)	Interrupt signal from the PL CPU0: IRQF2P[16] CPU1: IRQF2P[17]

The interrupt sources for PS and PL side are listed below [3]. It is possible to route 16 interrupts from the PL side to any or both ARM cores.

Source	Interrupt Name	IRQ ID#	Status Bits	Required Type	PS-PL Signal Name	I/O
APU	CPU 1, 0 (L2, TLB, BTAC)	33:32	spi_status_0[1:0]	Rising Edge	----	----
	L2 Cache	34	spi_status_0[2]	High Level	----	----
	OCM	35	spi_status_0[3]	High Level	----	----
Reserved	----	36	spi_status_0[4]	----	----	----
PMU	PMU [1,0]	38, 37	spi_status_0[6:5]	High Level	----	----
XADC	XADC		spi_status_0[7]	High Level	----	----
DVI	DVI	40	spi_status_0[8]	High Level	----	----
SWDT	SWDT	41	spi_status_0[9]	Rising Edge	----	----
Timer	TTC 0	43:42	spi_status_0[11:10]	High Level	----	----
Reserved	----	44	spi_status_0[12]	----	----	----
DMAC	DMAC Abort	45	spi_status_0[13]	High Level	IRQP2F[28]	O
	DMAC [3:0]	49:46	spi_status_0[17:14]	High Level	IRQP2F[23:20]	O
Memory	SMC	50	spi_status_0[18]	High Level	IRQP2F[19]	O
	Quad SPI	51	spi_status_0[19]	High Level	IRQP2F[18]	O
Debug	CTI	----	----	High Level	IRQP2F[17]	O
IOP	GPIO	52	spi_status_0[20]	High Level	IRQP2F[16]	O

Source	Interrupt Name	IRQ ID#	Status Bits	Required Type	PS-PL Signal Name	I/O
IOP	USB 0	53	spi_status_0[21]	High Level	IRQP2F[15]	O
	Ethernet 0	54	spi_status_0[22]	Rising Edge	IRQP2F[14]	O
	Ethernet 0 Wakeup	55	spi_status_0[23]	Rising Edge	IRQP2F[13]	O
	SDIO 0	56	spi_status_0[24]	High Level	IRQP2F[12]	O
	I2C 0	57	spi_status_0[25]	High Level	IRQP2F[11]	O
	SPI 0	58	spi_status_0[26]	High Level	IRQP2F[10]	O
	UART 0	59	spi_status_0[27]	High Level	IRQP2F[9]	O
	CAN 0	60	spi_status_0[28]	High Level	IRQP2F[8]	O
PL	FPGA [2:0]	63:61	spi_status_0[31:29]	High Level	IRQF2P[2:0]	I
	FPGA [7:3]	68:64	spi_status_1[4:0]	High Level	IRQF2P[7:3]	I
Timer	TTC 1	71:69	spi_status_1[7:5]	High Level	----	----
DMAC	DMAC [7:4]	75:72	spi_status_1[11:8]	High Level	IRQP2F[27:24]	O
IOP	USB 1	76	spi_status_1[12]	Rising Edge	IRQP2F[7]	O
	Ethernet 1	77	spi_status_1[13]	Rising Edge	IRQP2F[6]	O
	Ethernet 1 Wakeup	78	spi_status_1[14]	High Level	IRQP2F[5]	O
	SDIO 1	79	spi_status_1[15]	High Level	IRQP2F[4]	O
	I2C 1	80	spi_status_1[16]	High Level	IRQP2F[3]	O
	SPI 1	81	spi_status_1[17]	High Level	IRQP2F[2]	O
	UART 1	82	spi_status_1[18]	High Level	IRQP2F[1]	O
	CAN 1	83	spi_status_1[19]	High Level	IRQP2F[0]	O
PL	FPGA [15:8]	91:84	spi_status_1[27:20]	High Level	IRQF2P[15:8]	I
SCU	Parity	92	spi_status_1[28]	Rising Edge	----	----
Reserved	----	95:93	spi_status_1[31:29]	----	----	----

5.2 Software Development

Software is developed by the JCC compiler suite including the “binutils” which are responsible for linking, archiving, debugging and analyzing software.

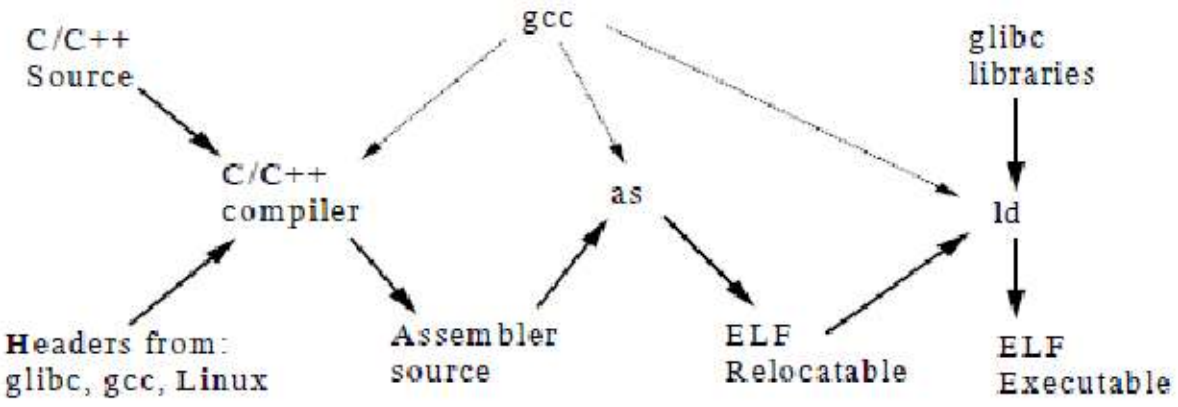


Figure 1.41: C/C++ compile flow

The dependencies on the GCC tools are shown in the figure above. Only a few modules need to be written in assembly language. The C compiler will generate assembler files (.s).

The cross development process is shown with a simple example with two C modules:

(1) amain.c:

```
#include "xil_printf.h"

int main()
{
    int fak10;

    print("-- cross C test ---\n");
    fak10 = fak(10);
    xil_printf("factorial of 10 is %d\n", fak10);
    print("Thank you.\n");
    return 0;
}
```

(2) fak.c:

```
int fak(int f)
{
    int  k, prod;

    prod = 1;
    if (f > 0) {
          for (k = 1; k <= f; k++) {
              prod *= k;
```

```
            }
        }
        return prod;
    }
```

The build process is carried out using a “makefile” which controls the build process:

```
################################################################################
# makefile
################################################################################

RM := rm -rf

# All of the sources participating in the build are defined here

OBJS=amain.o fak.o
SRCS=amain.c fak.c
LIBS=./lib/libxil.a

# Add inputs and outputs from these tool invocations to the build variables
ELFSIZE += \
aapp.elf.size \

# All Target
all: aapp.elf secondary-outputs

# Tool invocations
aapp.elf: $(OBJS) ./lscript.ld $(USER_OBJS)
          @echo ´Building target: $@´
          @echo ´Invoking: ARM gcc linker´
          arm-xilinx-eabi-gcc -Wl,-T -Wl,./lscript.ld -L./lib -o aapp.elf
$(OBJS) $(USER_OBJS) $(LIBS)
          @echo ´Finished building target: $@´
          @echo ´ ´

amain.o: amain.c
          arm-xilinx-eabi-gcc -g3 -c -I./include amain.c -o amain.o

fak.o: fak.c
          arm-xilinx-eabi-gcc -g3 -c -I./include fak.c -o fak.o

aapp.elf.size: aapp.elf
          @echo ´Invoking: ARM Print Size´
          arm-xilinx-eabi-size aapp.elf  |tee "aapp.elf.size"
          @echo ´Finished building: $@´
          @echo ´ ´

# Other Targets
clean:
          -$(RM) $(EXECUTABLES)$(OBJS)$(S_UPPER_DEPS)$(C_DEPS)$(ELFSIZE)
$(OBJS) aapp.elf
```

```
	-@echo ' '

secondary-outputs: $(ELFSIZE)

.PHONY: all clean dependents
```

The output of the build process is as follows:

```
mueller@alvxlx:~/GCCTST$ make
arm-xilinx-eabi-gcc -g3 -c -I./include amain.c -o amain.o
arm-xilinx-eabi-gcc -g3 -c -I./include fak.c -o fak.o
Building target: aapp.elf
Invoking: ARM gcc linker
arm-xilinx-eabi-gcc -Wl,-T -Wl,./lscript.ld -L./lib -o aapp.elf amain.o fak.o
           ./lib/libxil.a
Finished building target: aapp.elf

Invoking: ARM Print Size
arm-xilinx-eabi-size aapp.elf  |tee "aapp.elf.size"
   text      data       bss       dec       hex   filename
   3720        20     22564     26304      66c0   aapp.elf
Finished building: aapp.elf.size
```

The compiler is invoked with the "–g3" (debug) parameter to include symbols in the output. This allows to analyze the code using "objdump". In this example we see the generated assembly code for the "fak" function including source code:

```
arm-xilinx-linux-gnueabi-objdump -S aapp.elf | less


int fak(int f)
{
  100158:	e52db004 	push	{fp}		; (str fp, [sp,
#-4]!)
  10015c:	e28db000 	add	fp, sp, #0
  100160:	e24dd014 	sub	sp, sp, #20
  100164:	e50b0010 	str	r0, [fp, #-16]
	int  k, prod;
	prod = 1;
  100168:	e3a03001 	mov	r3, #1
  10016c:	e50b300c 	str	r3, [fp, #-12]
	if (f > 0) {
  100170:	e51b3010 	ldr	r3, [fp, #-16]
  100174:	e3530000 	cmp	r3, #0
  100178:	da00000d 	ble	1001b4 <fak+0x5c>
		for (k = 1; k <= f; k++) {
  10017c:	e3a03001 	mov	r3, #1
  100180:	e50b3008 	str	r3, [fp, #-8]
  100184:	ea000006 	b	1001a4 <fak+0x4c>
			prod *= k;
  100188:	e51b300c 	ldr	r3, [fp, #-12]
```

```
  10018c:       e51b2008        ldr     r2, [fp, #-8]
  100190:       e0030392        mul     r3, r2, r3
  100194:       e50b300c        str     r3, [fp, #-12]
  100198:       e51b3008        ldr     r3, [fp, #-8]
  10019c:       e2833001        add     r3, r3, #1
  1001a0:       e50b3008        str     r3, [fp, #-8]
  1001a4:       e51b2008        ldr     r2, [fp, #-8]
  1001a8:       e51b3010        ldr     r3, [fp, #-16]
  1001ac:       e1520003        cmp     r2, r3
  1001b0:       dafffff4        ble     100188 <fak+0x30>
                }
        }
        return prod;
  1001b4:       e51b300c        ldr     r3, [fp, #-12]
}
  1001b8:       e1a00003        mov     r0, r3
  1001bc:       e24bd000        sub     sp, fp, #0
  1001c0:       e49db004        pop     {fp}            ; (ldr fp, [sp], #4)
  1001c4:       e12fff1e        bx      lr
```

The size of the code can be analyzed by printing the sections headers (only first 4 sections):

```
mueller@alvxlx:~/GCCTST$ arm-xilinx-linux-gnueabi-objdump -h aapp.elf

aapp.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000d00  00100000  00100000  00008000  2**6
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .init         00000018  00100d00  00100d00  00008d00  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .fini         00000018  00100d18  00100d18  00008d18  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata       00000154  00100d30  00100d30  00008d30  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         0000000c  00100e84  00100e84  00008e84  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

Lab #03:

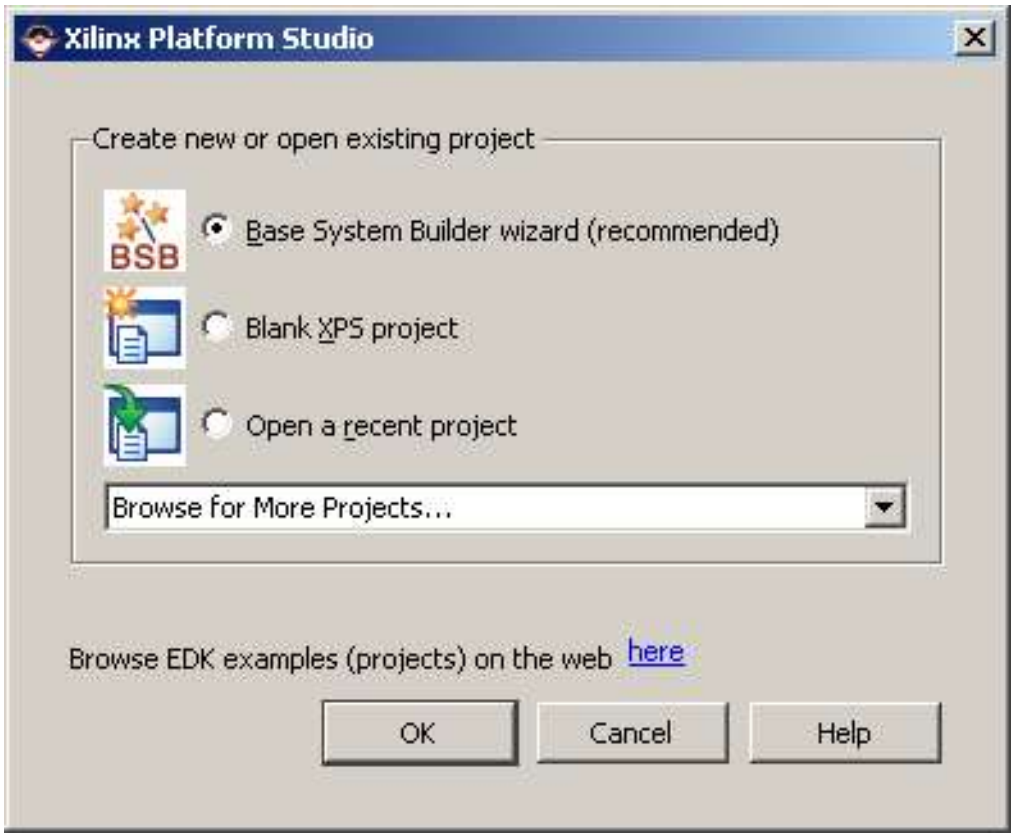
5.3 Minimal Realization of MicroBlaze™ on Atlys™ Board

A modern and powerful 32 bit microcontroller with a software development system (Eclipse IDE) is created by using IP (intellectual property). The use of an appropriate design software makes it easy to create a microcontroller system and to integrate user logic into the design flow. The following steps illustrate the ease of integrating the MicroBlaze microcontroller into a FPGA based system.

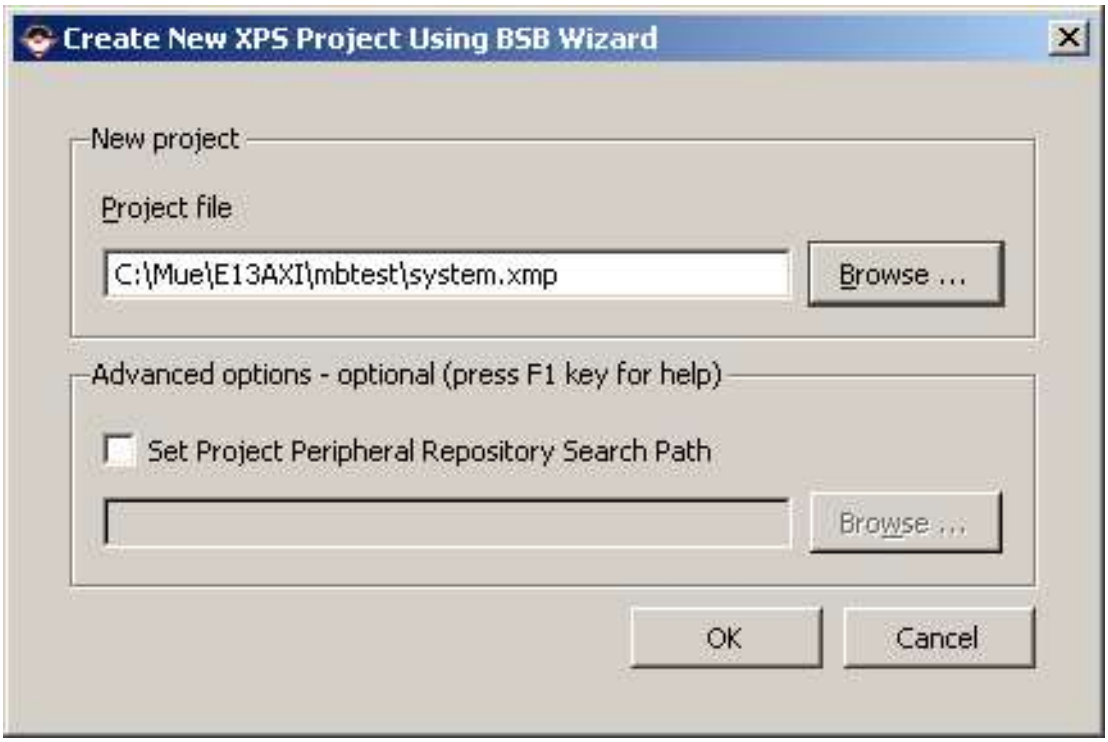
You might not understand each step in detail, the design flow will be explained later.

5.3.1 Hardware Design

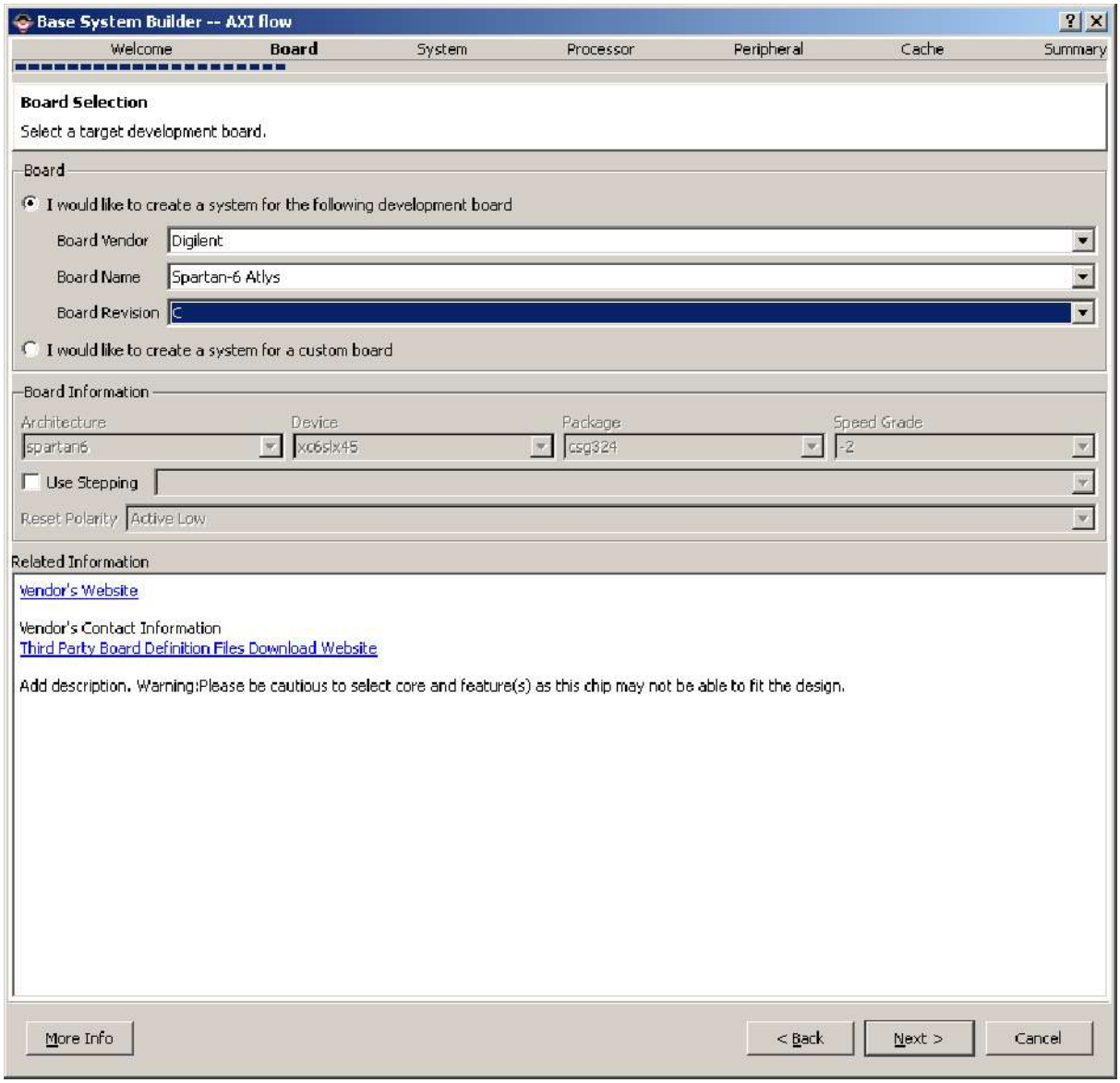
- Start XPS (Xilinx Platform Studio) to design the hardware. The board data (pins, FPGA device, peripherals must be made known to XPS by a board support package.
- Accept the "base system builder" and select as project file "C:\MUE\SOC\mbtest\system.xmp".



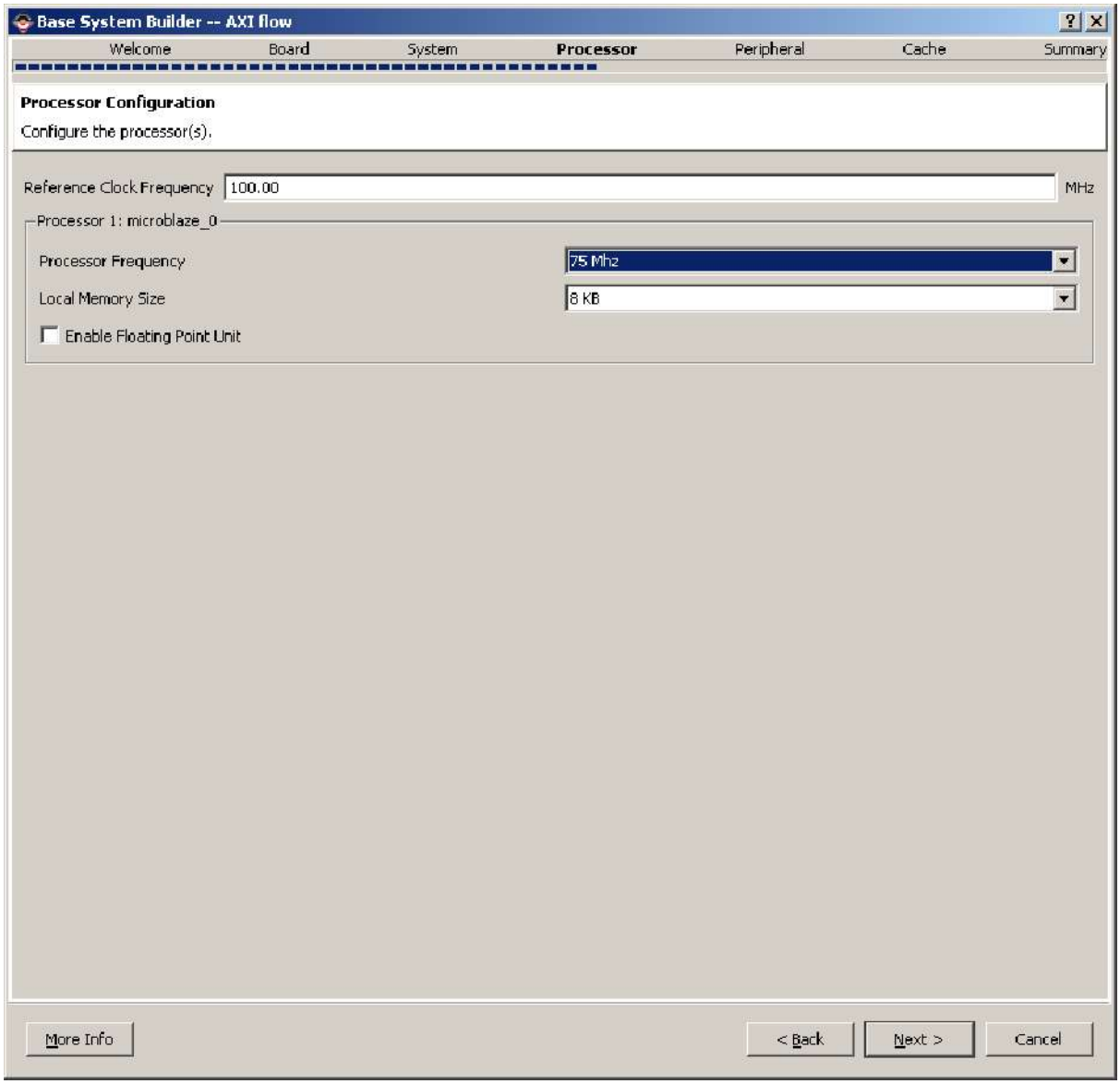
- Do not use the path from figure below!
  Use **C:\mueller\SOC\mbtest\system.cmp** instead.
  Press OK and wait a few seconds.



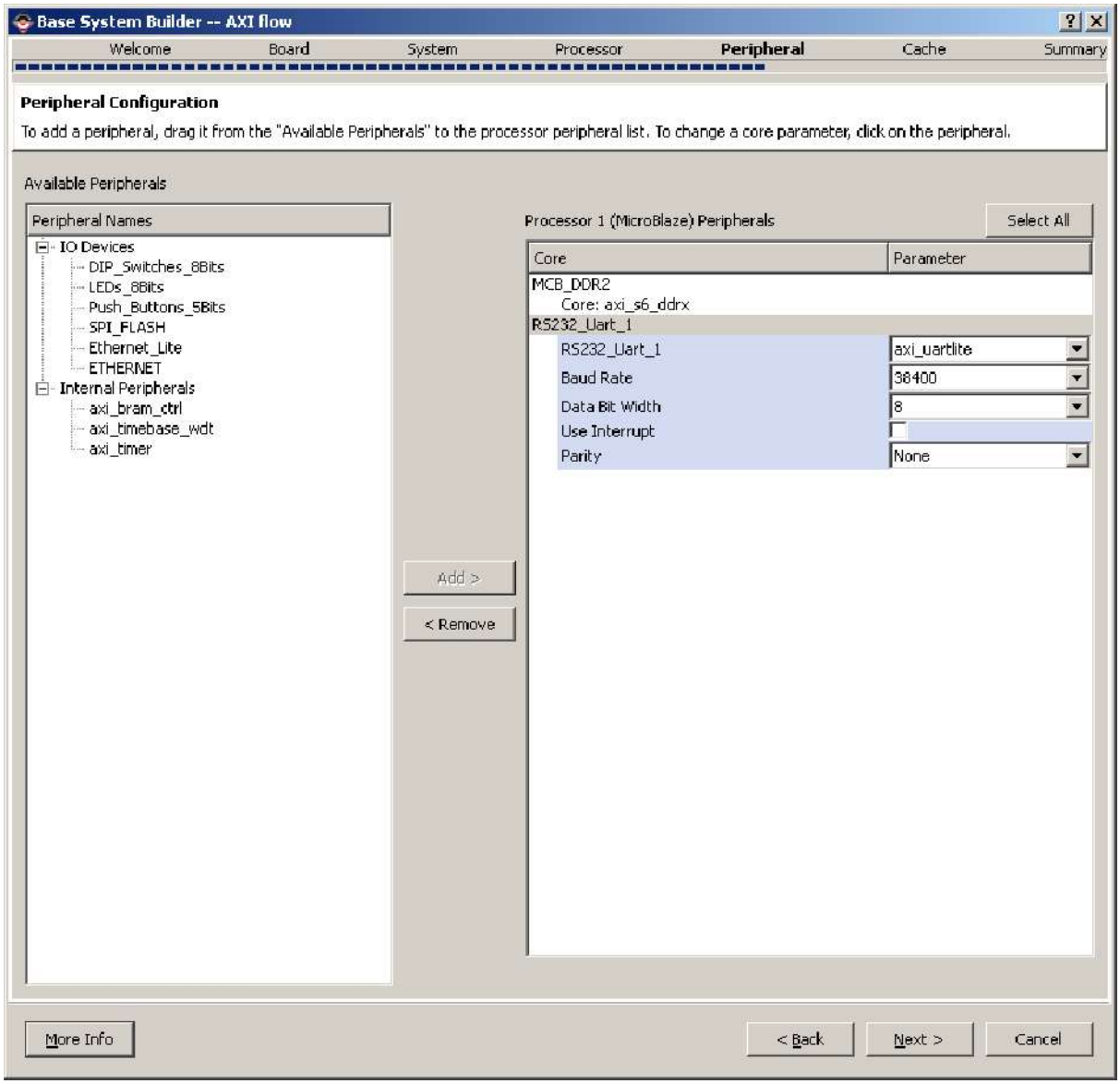
- Accept AXI system (new ARM compatible bus system) – press OK.
- Select on the next screen “ o I would like to create a new design” (default) and press NEXT.
- Fill in (if it is not offered by default) the appropriate board data and press NEXT.



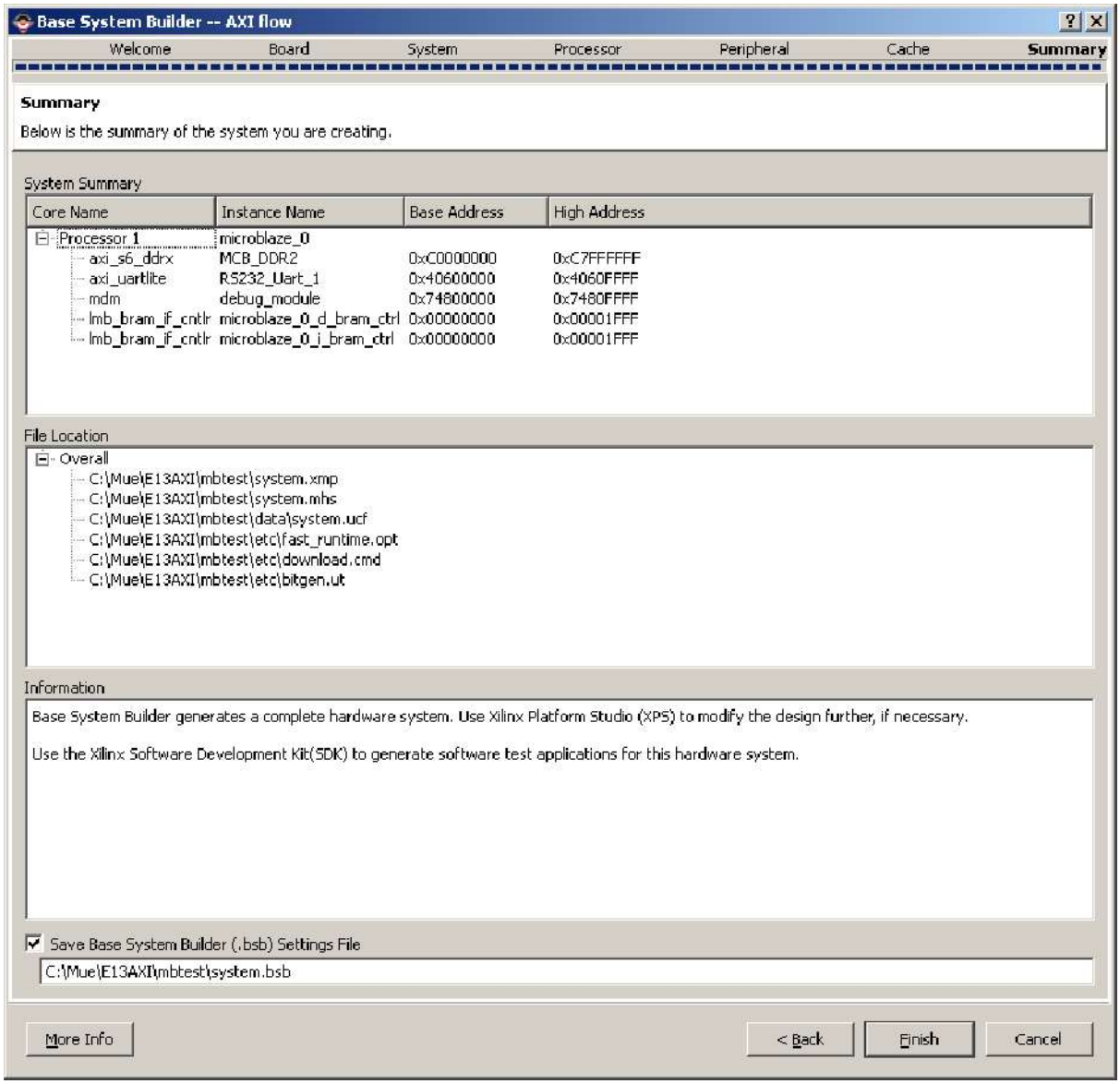
- Accept “AXI system with Single MicroBlaze Processor” and press NEXT.
- Change on Processor tab the “Processor Frequency” to 75 MHz, accept other defaults an press NEXT



- In the Peripheral section delete all peripherals except MCB_DDR2 (required memory) and RS232_Uart_1.
  In this peripheral change "Baud Rate" to 38400.
  Press NEXT.



- Accept the default for Cache and PRESS NEXT.
- Be patient for a minute until the summary screen appears. Don't push any button until you see this:
- Verify carefully the summary screen and press FINISH.



- Inspect the "System Assembly View" and the "Block Diagram" view to verify that all connections, ports and addresses are set up properly.

- You are ready now to create the FPGA configuration bitstream. This makes the hardware design complete.
  Select Hardware–> Generate Bitstream
  Be careful, this takes a long time to build!
  You can safely ignore the warnings – they can be explained.
  It take 7 minutes on an Intel i5™ 2.53MHz quad core CPU and
  10:14 minutes on an AMD Athlon II X4 605e CPU (with low power drives)

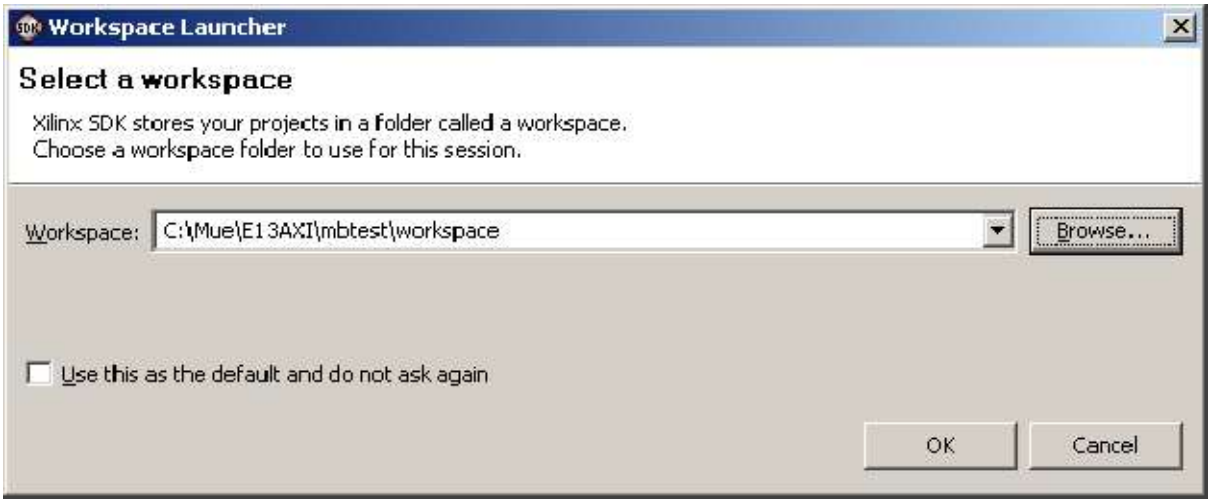
- Inspect under summary tab the resources consumed by the design.

5.3.2 Software Design

After the hardware has finished by creation of the configuration bitstream the software can be developed. This is done in a special version of Eclipse called SDK (Software Development Kit).

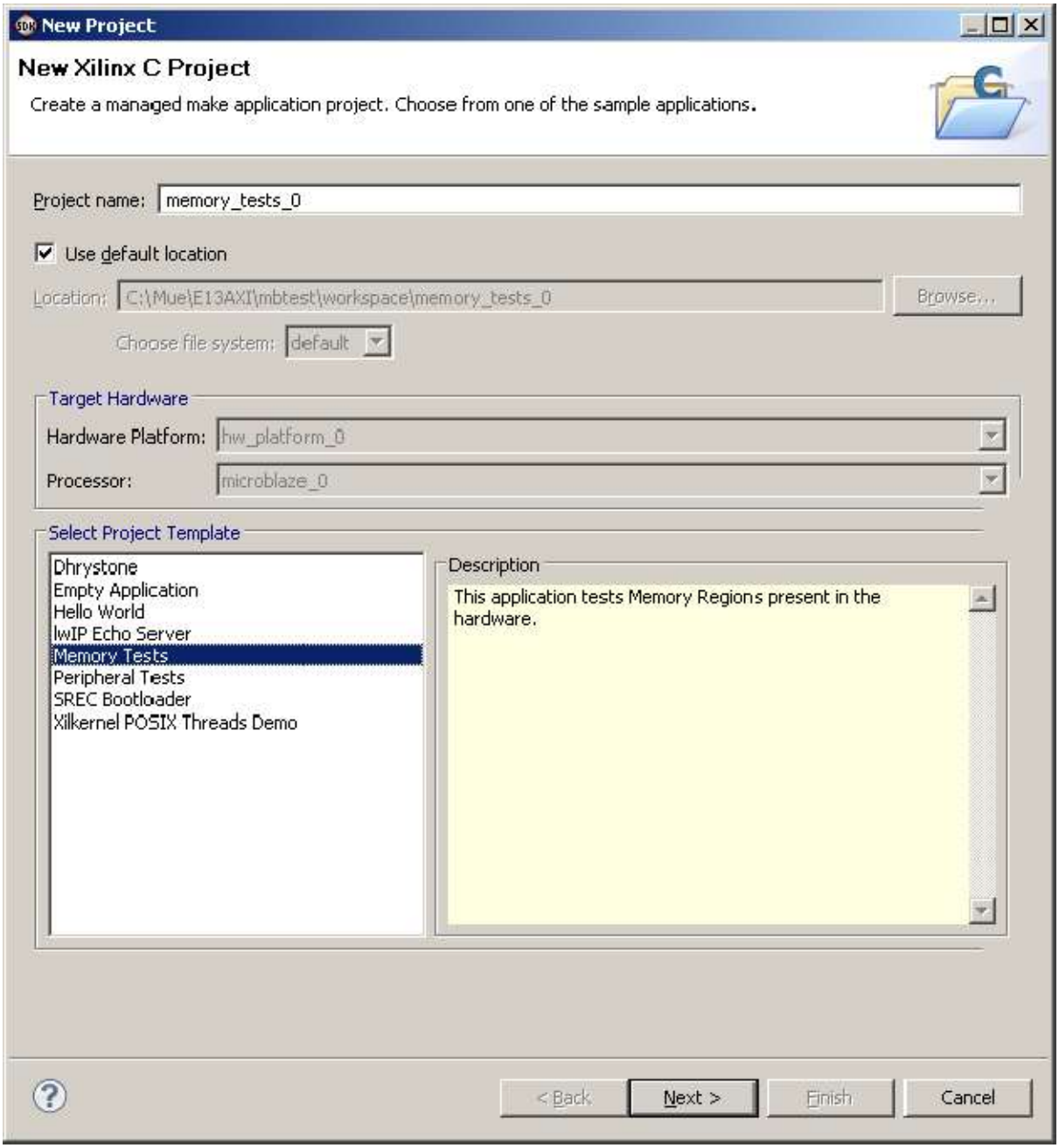
- From the main menu select Project–> Export Hardware Design to SDK
  Press Export and Launch SDK from the following pop-up window.

- In the Eclipse start-up screen select a workspace for your programs in your design directory.
  Press OK.
  SDK comes up with the imported hardware design files.



- Delete the welcome tab (it consumes window space and can be recovered again by selecting Help–> Welcome)

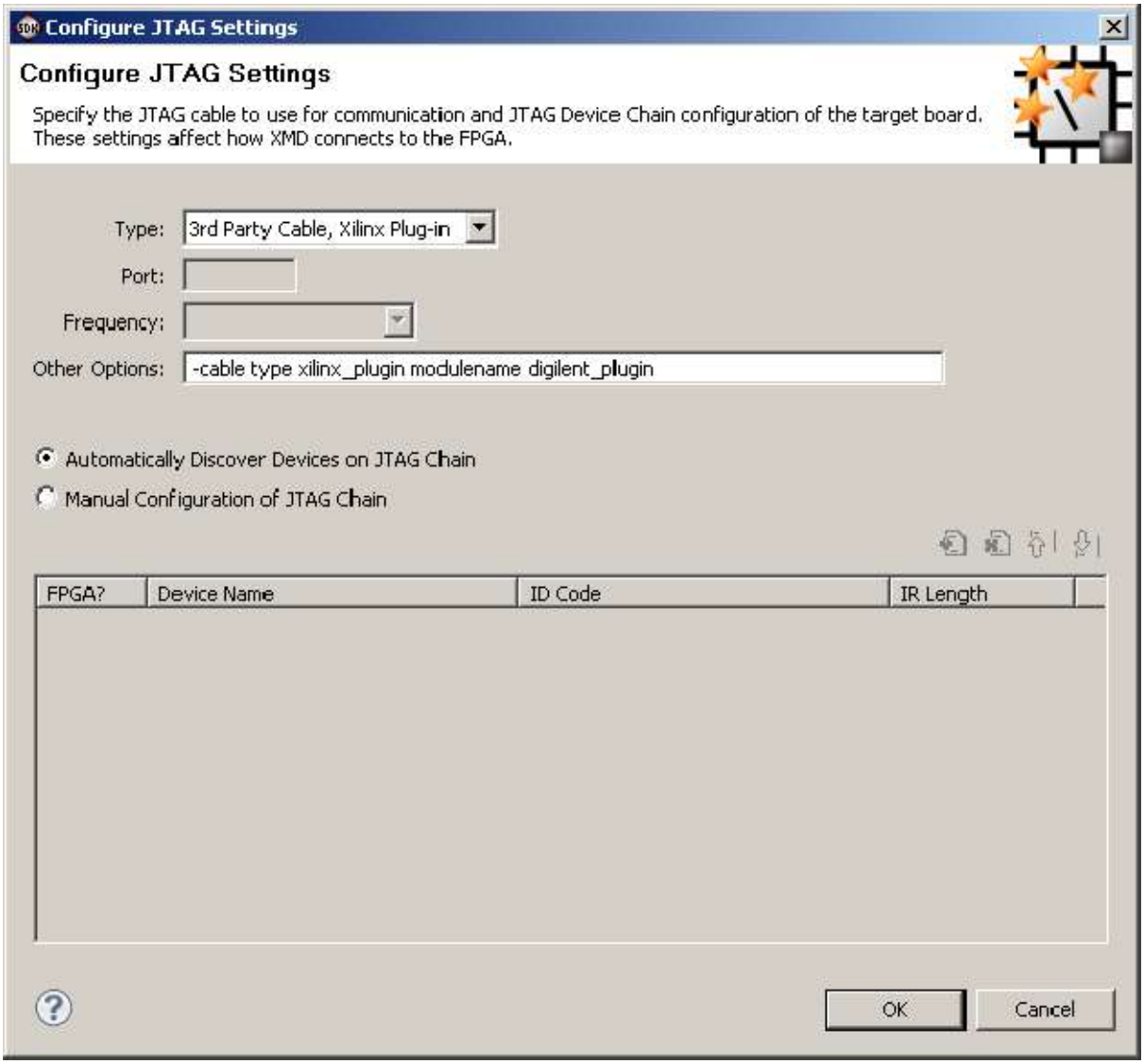
- We are ready now to create a sample project:
  Select Memory Tests Template and press NEXT.



- Accept the defaults on the next screen and press FINISH.
  The project should build automatically since automatic rebuild is the default setting. You should end with an ".elf" file called "memory_tests_0.elf" (Executable and Linkable Format). With the supplied .bmm file (Block ram Memory Map file) the .elf file can be loaded into processor memory.

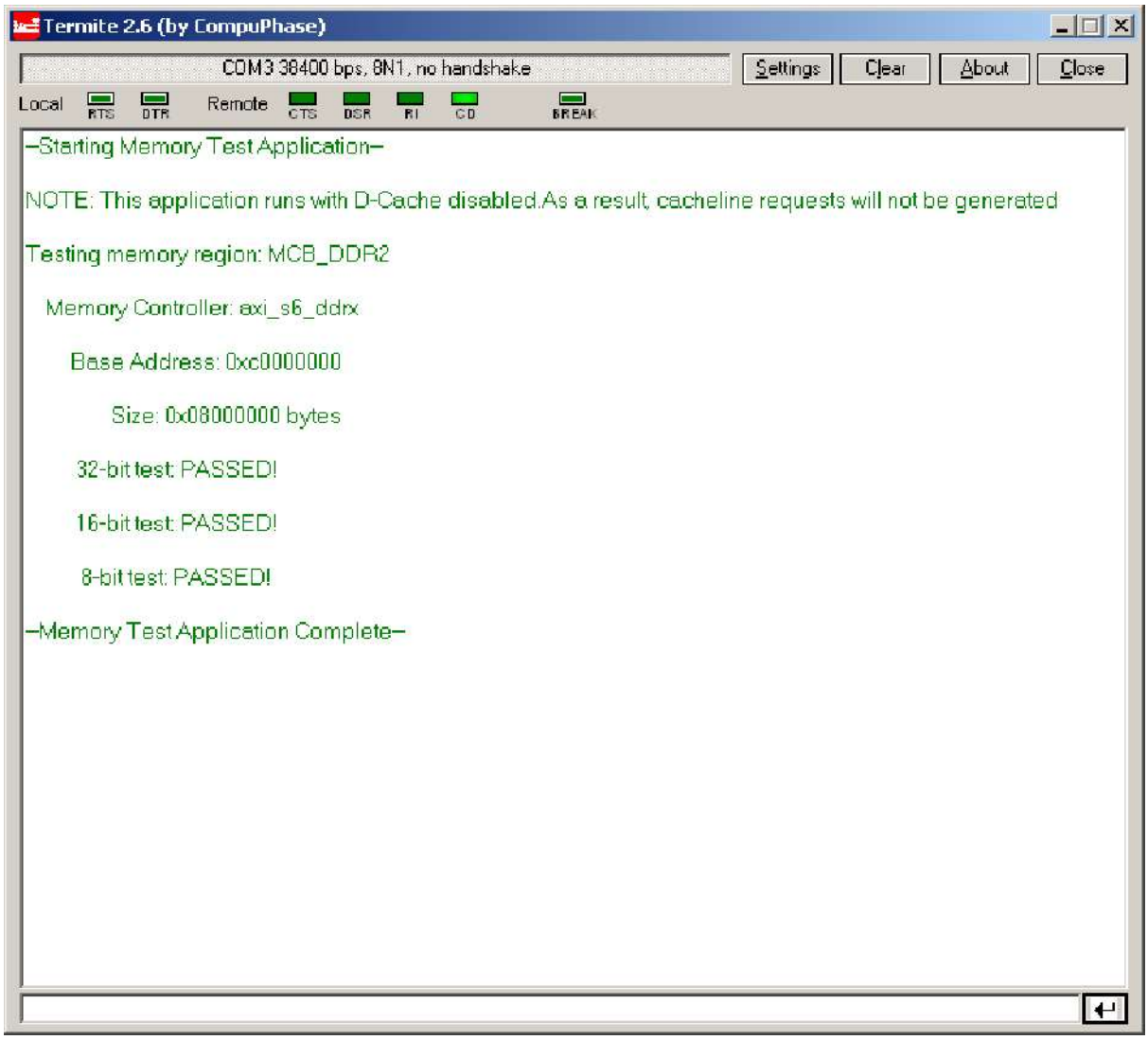
- Connect the Atlys board programmer to the *right* USB jack on the PC and the serial USB port to the *left* USB jack on the PC.
  Start the **termite** terminal program.
  Power on the Atlys board.

- Before FPGA configuration and program download the Digilent plugin must be made known to SDK:
  Select Xilinx Tools–> Configure JTAG Settings.



Make the appropriate changes and press OK

- You are now ready to run your first program.
  Select Xilinx Tools–> Program FPGA
  In the pop-up window select your .elf file: memory_tests_0.elf.
  Press Program. The result should look like this:



- Pressing on the Atlys board the red button (=RESET) will restart the program.
- Change the program and reprogram the FPGA again to see it's effect.

5.3.3 Embedded CPU Design Questions

The following question are required to be answered. Please fill into fields according to your EDK / SDK solutions.

- By which interface are the following peripherals connected?
  (From the EDK bus Interfaces diagram)

  debug_module: _____

MCB_DDR2: _____

RS232_UART_1: _____

- Which nets are connected to the following ports?
(From the EDK Ports tab folder)

RS232_Uart_1 – RX: _____

RS232_Uart_1 – TX: _____

Which physical pins are connetcted to these nets?

_____

_____

- List the Base address and High address of the following AXI devices (memory mapped addresses from the EDK Addresses tab):

RS232_Uart_1 – Base Address: _____

RS232_Uart_1 – High Address: _____

d_bram_ctrl – Base Address: _____

d_bram_ctrl – High Address: _____

i_bram_ctrl – Base Address: _____

i_bram_ctrl – High Address: _____

Does the size of the RS232_Uart_1 device matches the actual size?

If not, why? _____

5.3.4 Analyzing Software (Processing System)

The following question are required to be answered. Please fill into fields according to your SDK solutions. The software sections are located by the GNU linker (ld). The section can be assigned to any address by the linker script file. The address section must match the addresses assigned in the hardware design phase.

- According to the linker script file, where are the software sections located?

  memory region: _____

  physical (base) address: _____

Analyze the assigned software regions and the size of each code segment with the `mb-objdump` utility (part of the GNU binutils). Therefore open the `ISE Design Suite Command Prompt` from Xilinx Design Tools –> ISE Design Suite XX.yy –> Accessories. For a "C" program the following sections exist:

.text:	program code
.data:	program static initialized data (i.e. strings)
.sdata:	small data
.rodata:	read only data
.bss:	uninitialyzed global or static data
.sbss	small uninitialyzed data
.init:	code before `main()` is executed
.fini:	code after termination of `main()`
.got:	table of addresses (global offset table)
.heap	memory space for dynamic memory allocations
.stack	stack size (depend on number of calls, subroutine parameters, interrupts etc.

Heap and stack size is not easy to estimate since it depends on the program structure and the libraries used in an application. If network protocols are used, the required amount of heap and stacks increases significantly.

The compiler may create other sections for instance constructor or destructor sections for C++ code which will be explained here.

Analyze the output of mb-objdump –h xxx.elf and fill out the following list:

- .text:

  size: _____

  base address: _____

  memory region: _____

- .data:

  size: _____

  base address: _____

memory region: _____

- .sdata:

  size: _____

  base address: _____

  memory region: _____

- .rodata:

  size: _____

  base address: _____

  memory region: _____

- .bss:

  size: _____

  base address: _____

  memory region: _____

- .sbss:

  size: _____

  base address: _____

  memory region: _____

- .heap:

  size: _____

  base address: _____

  memory region: _____

- .stack:

  size: _____

  base address: _____

  memory region: _____

- Explain how program execution can be made faster by assigning different regions to program sections.

Lab #04:

5.4 Adding User Logic and IP to MicroBlaze

5.4.1 Hardware

In this section we will add existing IP (intellectual property) elements to the MicroBlaze microcontroller. This allows to configure an FPGA to meet the requirements of a specific embedded system.

In order to add several components to the microcontroller a standard bus system like AXI (Advanced eXtensible Interface by ARM™). Adding and changing components becomes then a matter of minutes and require no extra chips, board space, or extra cards.

Some years ago build such a systems involves a module rack and several modules. The realization was very expensive, time consuming and not really flexible.

In our example we will build a complete system with timer, interrupt controller and user logic to access special hardware which is not covered by a suitable IP. The specification of the system is shown in fig. 1.42. All components will reside on the same chip.

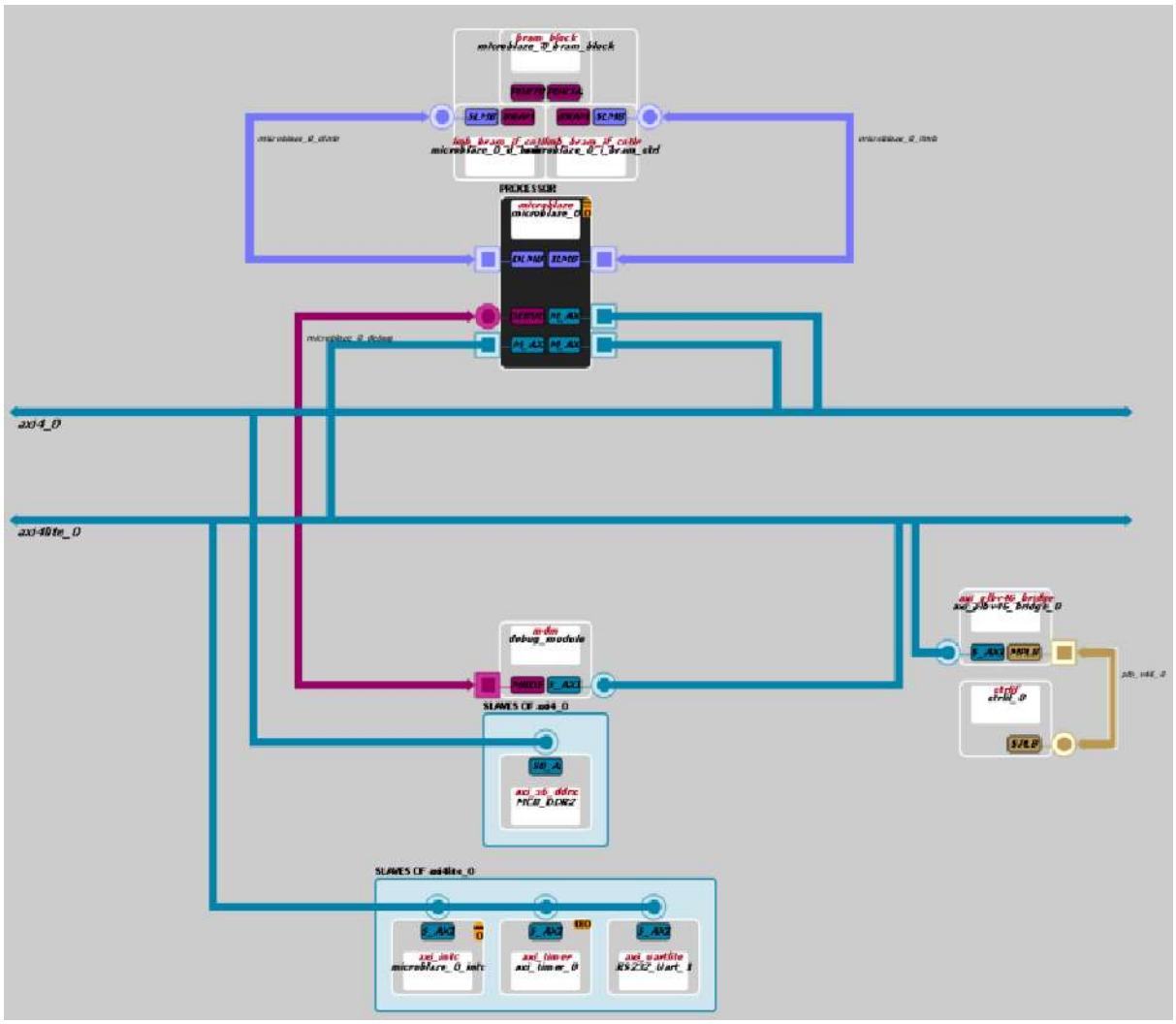


Figure 1.42: Complete microcontroller system

System specifications

- MicroBlaze microcontroller with 32 bit fixed point ALU. The CPU can operate with 75 MHz or 100 MHz. Note that the system clock is 100 MHz.
- 8K internal block ram for program (ilmb). This is suitable for very small programs or boot loader for programs from flash memory
- 8K internal block ram for data (dlmb). This is sufficient for data, stack and heap with very small programs. For large programs, stack and heap can reside in the very fast component.
- DDR2 controller for external memory (128 MByte) on AXI bus.
- 8K byte D-Cache for data in external memory.
- 8K byte I-Cache for program in external memory.
- MDM debug module for program loader and debugger.
- UART (lite) for communication between MicroBlaze and PC

- AXI Interrupt controller for processing timer interrupt.
- AXI 32 bit timer.
- “ctrlif” user logic module for accessing hardware on board (switches, pushbuttons and leds) for PLB bus. The PLB (Processor Local Bus) is the simplest way to interface user logic with low volume data (ADCs, DACs, Sensors etc.).
- AXI-to-PLB bridge for connecting the user logic. No speed penalty is involved here since the bus bridge is hardware.

The following figure outline the interconnections is detail. The block diagram will be created automatically by XPS (Xilinx Platform Studio, the hardware design software).

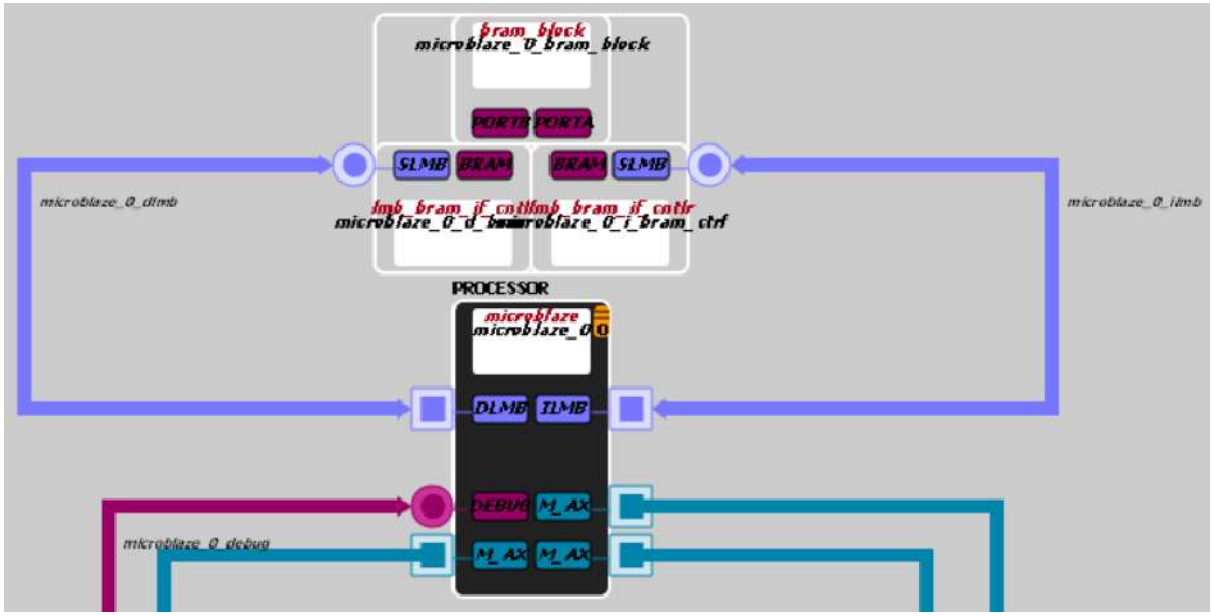


Figure 1.43: MicroBlaze local memories and busses

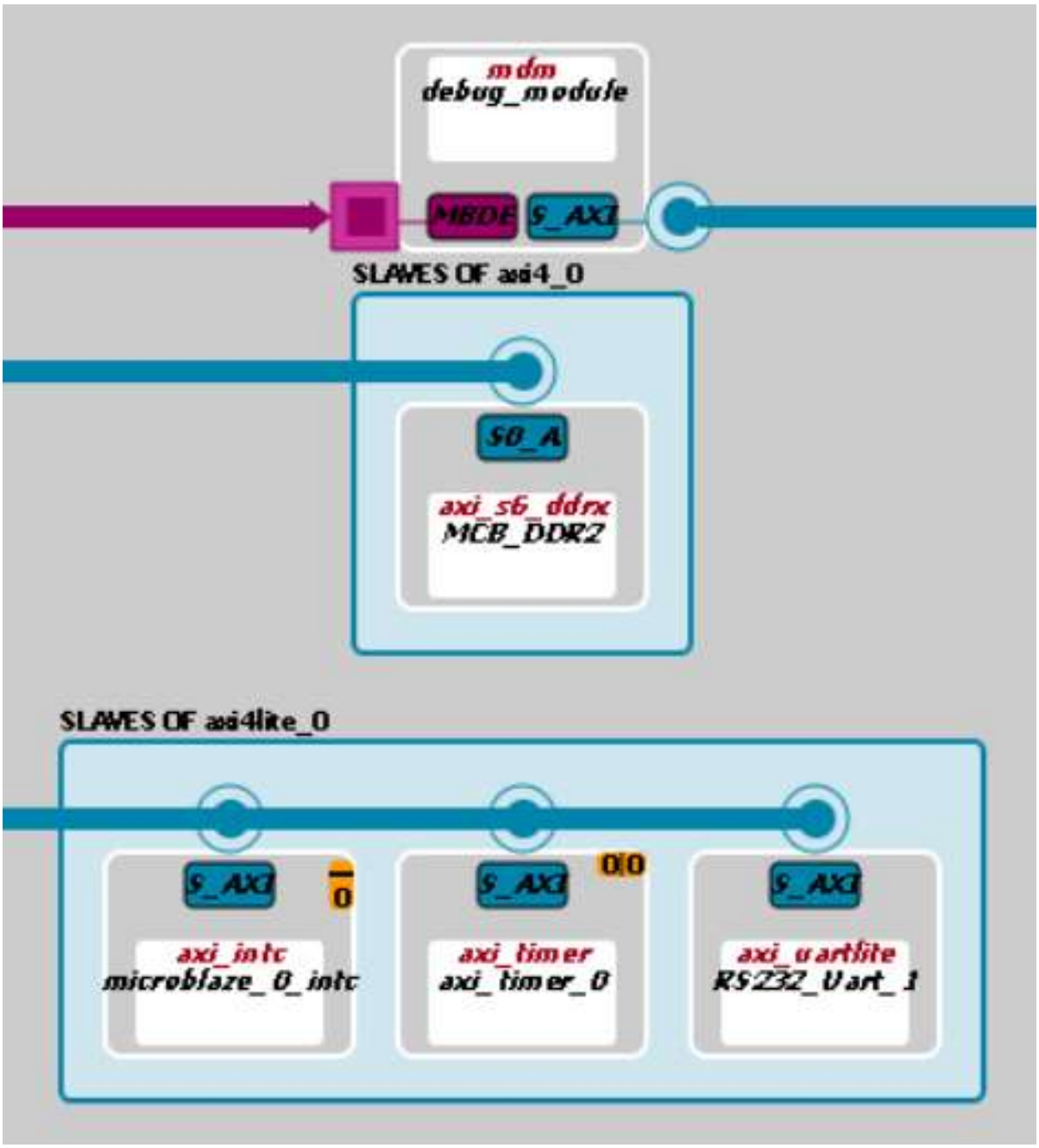


Figure 1.44: AXI devices (IP)

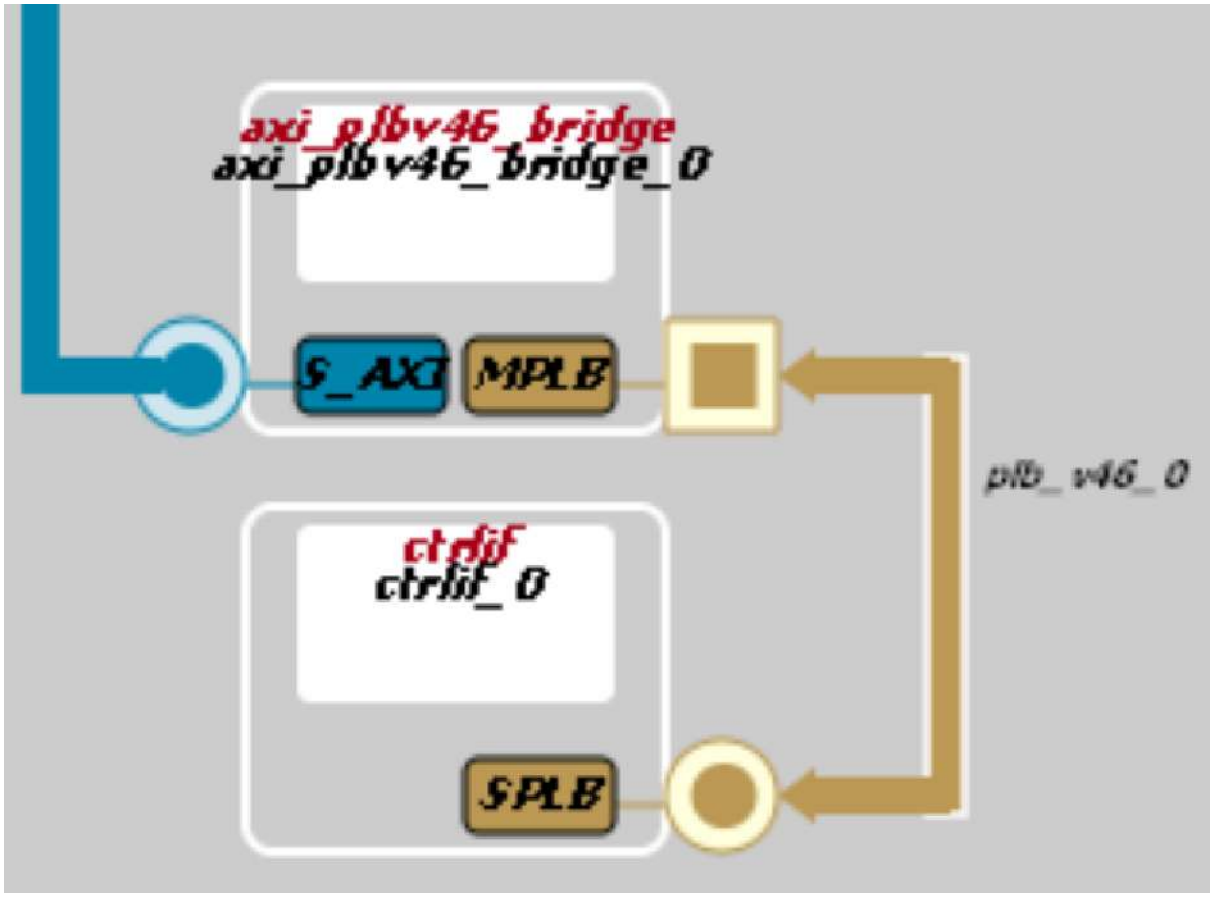


Figure 1.45: User logic attached to PLB (Processor Local Bus and AXI bridge)

The hardware design is done in several steps. First we create the microcontroller with all IP blocks and the user logic as template. If the design is operational then the user logic will be programmed in VHDL.

- Create a new project "mpsp6s"

- With the "Base System Builder" create an initial system.
  Specify appropriate sizes for dlmb and ilamb RAM size.
  Select 8 KByte cache for instructions and data memory.
  Include DRAM controller, UART, Timer and Interrupt Controller.

- Select from main menu
  Hardware–> Create or Import Peripheral and add an local pcore (peripheral core) for the PLB. Select "ctrlif" (control interface) as name. Choose 4 registers for your user logic.

- In the "System Assembly View" tab and in "IP Catalog" view add an "axi_plbv46_bridge".
  Connect it to the "axi4_lite" bus.
  In the ports tab be sure to connect "plb_v46" according to the following figure.

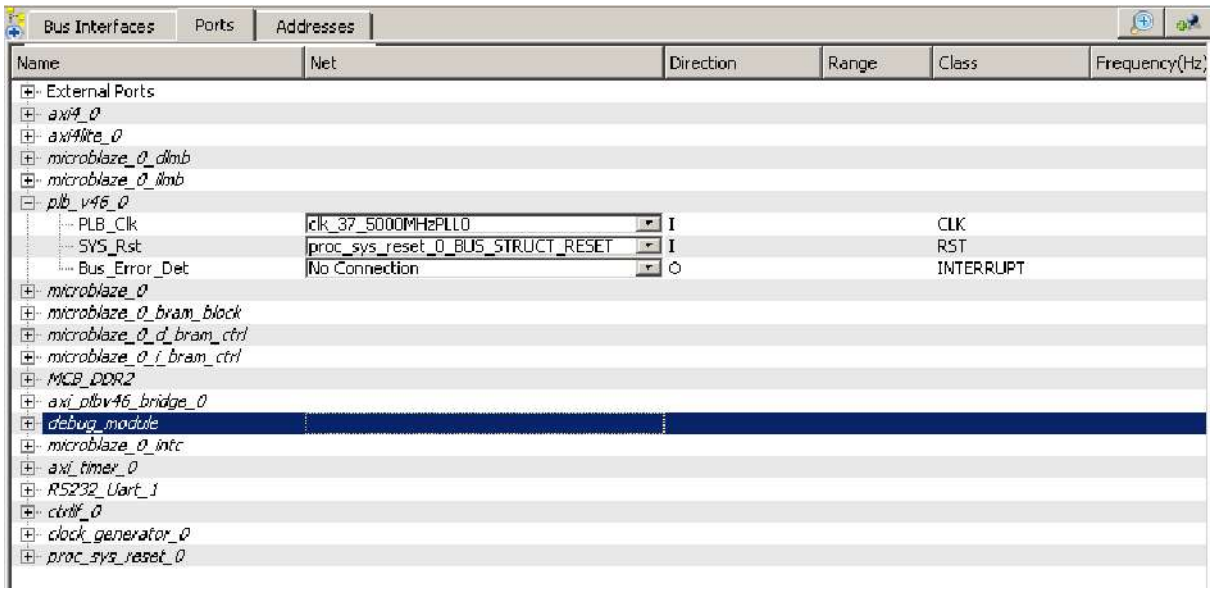


Figure 1.46: Required pbl_v46 port connections

- You can now connect your newly created user logic pcore to the PLB.
- In the "Addresses" tab press the "Generate Adresses" button and carefully watch the created address ranges. Do the address ranges make sense? Why are address ranges larger than required?
- Inspect your design in all tabs. If everything seems satisfying select Hardware–> Generate Bitstream from the main menu (and have a break of 10 minutes...).
- If the bitstream generation was successful enter Project–> Export Hardware to SDK to start software development.

5.4.2 Software

All Software design (board support library, device driver, program) is carried out in SDK (Xilinx Software Development Kit, the eclipse based IDE).

- From the main menu select
  File–> New–> Xilinx C Project
  and create a "memory tests" application.
- Execute the program in Atlys board to verify proper operation (on termite terminal).

***

Lab #05:

5.5 Ethernet Communication

Ethernet allows high speed communication up to 10Gbyte/s. Depending on the required real-time performance several protocol stacks are possible (Powerlink, Ethercat or others). Ethernet outperforms all existing field bus systems and most likely supersede currently existing field bus systems.

Transmission is possible over various media (copper, optical fiber, wireless, etc.). Every Ethernet connection has a similar as shown in fig 1.47.

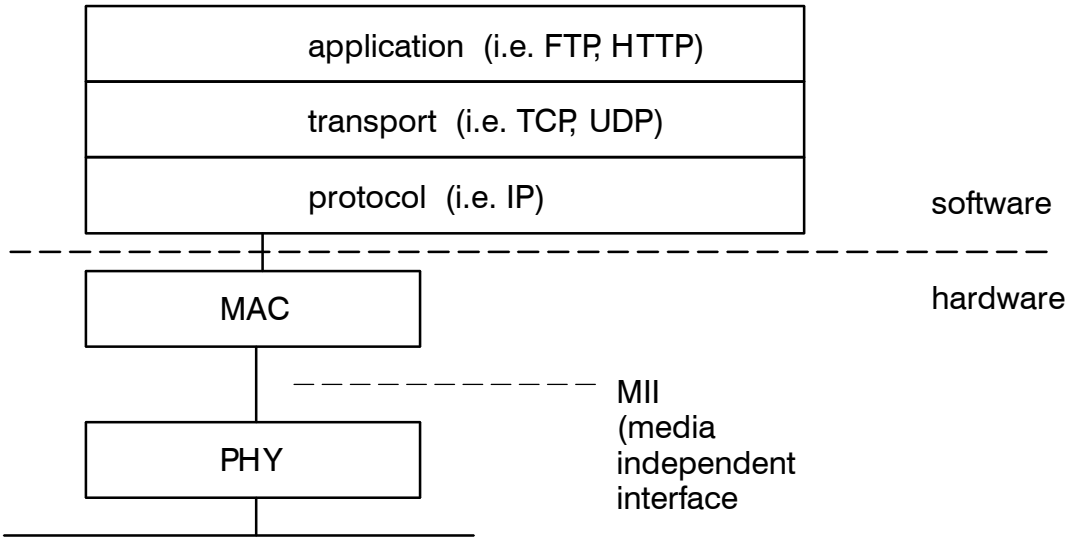


Figure 1.47: Ethernet structure

All Ethernet traffic is organized into frames.

802.3 Ethernet frame structure

Preamble	Start of frame delimiter	MAC destination	MAC source	802.1Q tag (optional)	Ethertype (Ethernet II) or length (IEEE 802.3)	Payload	Frame check sequence (32-bit CRC)	Interframe gap
7 octets of 10101010	1 octet of 10101011	6 octets	6 octets	(4 octets)	2 octets	42[note 2]–1500 octets	4 octets	12 octets
		64–1522 octets						
72–1530 octets								
84–1542 octets								

Figure 1.48: Ethernet frame (standard frame)

5.5.1 TCP Server Implementation Details

TCP/IP is a routable protocol, i.e. the route to a destination can be done by one or many gateways. In this example we will not use the gateway which is typical for an embedded application.

Every node requires a (unique) IP address. Usually it is assigned manually for embedded networks. This example uses the DHCP (dynamic host configuration protocol) to avoid IP address conflicts in the university network. Using DHCP requires lwIP library that is configured for DHCP.

It is mandatory that each member of an ethernet network has a unique ethernet hardware id. This ID is programmable (not for standard ethernet adapters, they have a factory assigned unique ID). Every group should assign a different ID to their Zynq nodes.

Resources:
The TCP server requires an Ethernet MAC, interrupts and timers. Since these component are part of the Zynq ARM core, no programmable logic is required. In order to use TCP/IP Ethernet all resources need to be initialized and configured.

Initialization of timer for Interrupts:

```
void platform_setup_timer(void)
{
    int Status = XST_SUCCESS;
    XScuTimer_Config *ConfigPtr;
    int TimerLoadValue = 0;

    ConfigPtr = XScuTimer_LookupConfig(TIMER_DEVICE_ID);
    Status = XScuTimer_CfgInitialize(&TimerInstance, ConfigPtr,
                                     ConfigPtr->BaseAddr);
    if (Status != XST_SUCCESS) {
          xil_printf("In %s: Scutimer Cfg initialization failed...\r\n",
                     __func__);
              return;
    }
    Status = XScuTimer_SelfTest(&TimerInstance);
    if (Status != XST_SUCCESS) {
          xil_printf("In %s: Scutimer Self test failed...\r\n",
                     __func__);
          return;
    }
    XScuTimer_EnableAutoReload(&TimerInstance);
    // Set for 125/250 milli seconds timeout. */
    TimerLoadValue = XPAR_CPU_CORTEXA9_0_CPU_CLK_FREQ_HZ / 8;
    XScuTimer_LoadTimer(&TimerInstance, TimerLoadValue);
    return;
}
```

Initialization of Interrupt:

```
void platform_setup_interrupts(void)
{
    Xil_ExceptionInit();
    XScuGic_DeviceInitialize(INTC_DEVICE_ID);
    // Connect the interrupt controller interrupt handler to the hardware
    // interrupt handling logic in the processor.
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,
                (Xil_ExceptionHandler)XScuGic_DeviceInterruptHandler,
                (void *)INTC_DEVICE_ID);
    // Connect the device driver handler
    XScuGic_RegisterHandler(INTC_BASE_ADDR, TIMER_IRPT_INTR,
                            (Xil_ExceptionHandler)timer_callback,
                            (void *)&TimerInstance);
    // Enable the interrupt for scu timer.
    XScuGic_EnableIntr(INTC_DIST_BASE_ADDR, TIMER_IRPT_INTR);
    return;
}
```

This is done in lwIP in a (huge) structure "struct netif". The components of this structure depend on the lwIP configuration. It is carried out with a pointer to this structure:

```
echo_netif = &server_netif;
```

All functions of lwIP Ethernet are initialized (must be preceded any lwIP operation) by

```
lwip_init();
```

The Ethernet MAC is added to the list of interfaces:

```
xemac_add(echo_netif, &ipaddr, &netmask,
          &gw, mac_ethernet_address,
          PLATFORM_EMAC_BASEADDR)
```

and set as default interface (there is only one interface in this system):

```
netif_set_default(echo_netif);
```

Enable Interrupts:

```
void platform_enable_interrupts()
{
    // Enable non-critical exceptions.
    Xil_ExceptionEnableMask(XIL_EXCEPTION_IRQ);
    XScuTimer_EnableInterrupt(&TimerInstance);
    XScuTimer_Start(&TimerInstance);
    return;
}
```

Interrupt Handler:

The handler is installed during the call to `platform_setup_interrupts(void)`.

```
void timer_callback(XScuTimer * TimerInstance)
{
    static int odd = 1;
    static int dhcp_timer = 0;
    odd = !odd;
    ResetRxCntr++;
    tcp_fasttmr();
    if (odd) {
          dhcp_timer++;
          dhcp_timoutcntr--;
          tcp_slowtmr();
          dhcp_fine_tmr();
          if (dhcp_timer >= 120) {
              dhcp_coarse_tmr();
              dhcp_timer = 0;
          }
    }
    if (ResetRxCntr >= RESET_RX_CNTR_LIMIT) {
          xemacpsif_resetrx_on_no_rxdata(echo_netif);
          ResetRxCntr = 0;
    }
    XScuTimer_ClearInterruptStatus(TimerInstance);
}
```

The Ethernet MAC can now be enabled:

```
netif_set_up(echo_netif);
```

Server Startup:

```
int start_application()
{
    struct tcp_pcb *pcb;
    err_t err;
    unsigned port = 7;

    // create new TCP PCB structure
    pcb = tcp_new();
    if (!pcb) {
          xil_printf("Error creating PCB. Out of Memory\n\r");
          return -1;
    }
    // bind to specified @port
    err = tcp_bind(pcb, IP_ADDR_ANY, port);
    if (err != ERR_OK) {
          xil_printf("Unable to bind to port %d: err = %d\n\r", port, err);
          return -2;
    }
    // we do not need any arguments to callback functions
    tcp_arg(pcb, NULL);
```

```
    // listen for connections
    pcb = tcp_listen(pcb);
    if (!pcb) {
          xil_printf("Out of memory while tcp_listen\n\r");
          return -3;
    }
    // specify callback to use for incoming connections
    tcp_accept(pcb, accept_callback);
    xil_printf("TCP echo server started @ port %d\n\r", port);
    // no errors!
    return 0;
}
```

accept_callback (see bottom of function `start_application()`) is a function that is called whenever a new connection was made:

```
err_t accept_callback(void *arg, struct tcp_pcb *newpcb, err_t err)
{
    static int connection = 1;
    // set the receive callback for this connection
    tcp_recv(newpcb, recv_callback);
    // just use an integer number indicating the connection id as the
    //    callback argument
    tcp_arg(newpcb, (void*)connection);
    // increment for subsequent accepted connections
    connection++;
    return ERR_OK;
}
```

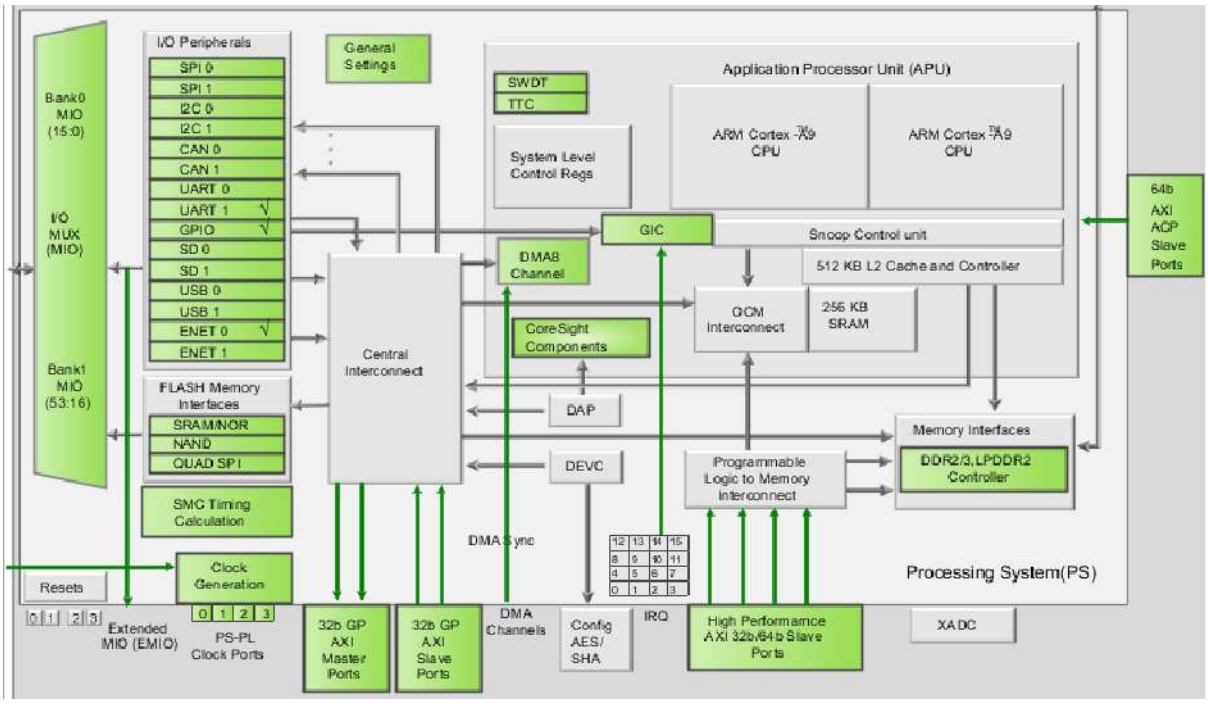
Any receive package will result in a call to `accept_callback(...)`:

```
err_t recv_callback(void *arg, struct tcp_pcb *tpcb,
                                struct pbuf *p, err_t err)
{
    char buffer[64];
    int k, slen;
    unsigned int xval, mask;

    // do not read the packet if we are not in ESTABLISHED state
    if (!p) {
          tcp_close(tpcb);
          tcp_recv(tpcb, NULL);
          return ERR_OK;
    }

    // indicate that the packet has been received
    tcp_recved(tpcb, p->len);
    slen = p->len;
    buffer[0] = '[';
    buffer[1] = 'R';
    buffer[2] = ']';
```

```
        buffer[3] = ':';
        buffer[4] = ' ';
        strncpy(&buffer[5], (char *)p->payload, slen);
        slen += 5;
        buffer[slen++] = '\r';
        buffer[slen++] = '\n';
        buffer[slen] = '\0';
        if (buffer[5] != '\r') {
              // this is not 0x0d/0x0a
              print(buffer);
              if (!strncmp(&buffer[5], "exit", 4)) {
                  Terminate_TCPServer = 1;
              } else if (!strncmp(&buffer[5], "led ", 4)) {
                  if (sscanf(&buffer[9], "%x", &xval) == 1) {
                        zbSetLed(xval);
                  } else {
                        buffer[slen++] = '#';
                        buffer[slen++] = '\r';
                        buffer[slen++] = '\n';
                        buffer[slen] = '\0';
                  }
              } else if (!strncmp(&buffer[5], "but", 3)) {
                  xval = zbGetButSwitch();
                  .
                  .
                  .
              } else if (!strncmp(&buffer[5], "switch", 6)) {
                  xval = zbGetButSwitch();
                  .
                  .
                  .
              }
              // send the buffer
              // in this case, we assume that the payload is < TCP_SND_BUF
              if (tcp_sndbuf(tpcb) > slen) {
                  err = tcp_write(tpcb, (void *)buffer, slen, 1);
              } else {
                  xil_printf("no space in tcp_sndbuf\n\r");
              }
        }
        // free the received pbuf
        pbuf_free(p);
        return ERR_OK;
}
```

5.6 Lab Tasks

- Create a new project "armetw"
- Configure an ARM Core system as block design, enable
  - UART 1 (for standard io),
  - ENET 0 (Gigabit Ethernet MAC),
  - GPIO (for PHY reset)

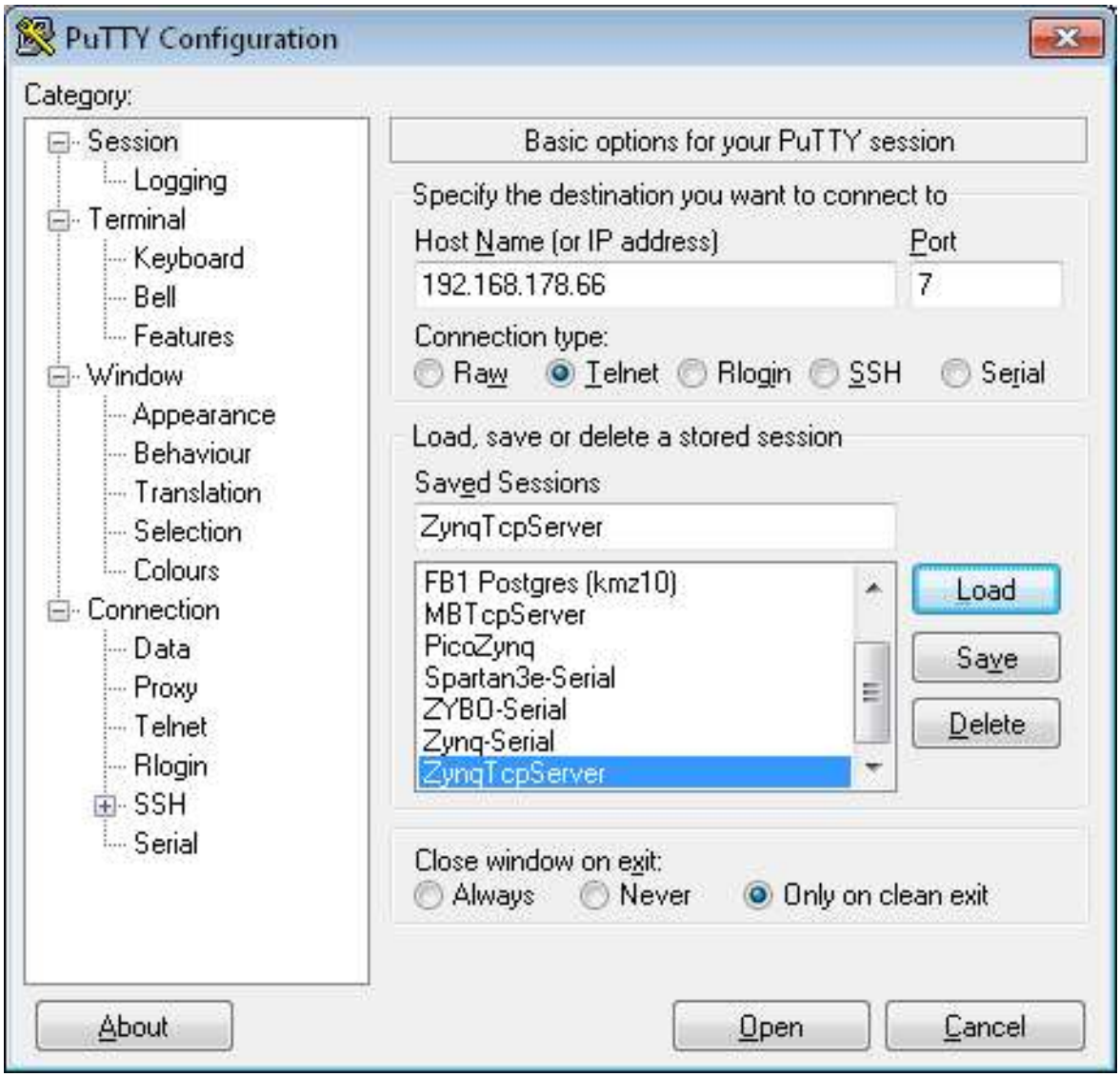


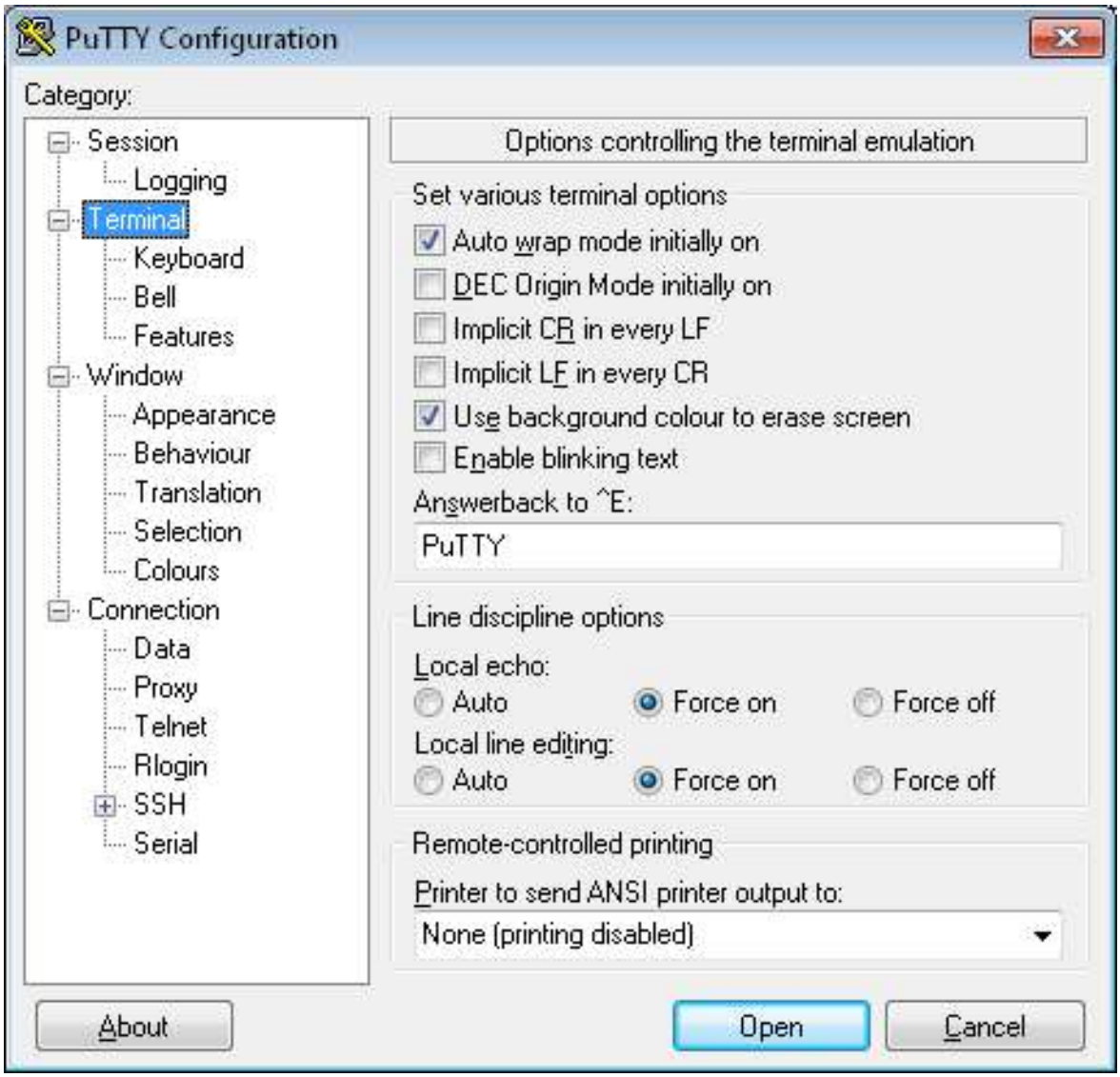
- Create a user logic peripheral to access LEDs, switches and buttons.
  This can be used from previous lab; copy the "ip_repo" into the local project folder.
- Add "armnetw.xdc" as constraints file (provided).
- Create configuration bitstream and transfer the design to SKD (Eclipse)
- Create a "hello_world" project to verify proper operation.
- Build a new board support package "armnetw_bsp_0".
  Add the lwIP library and modify the DHCP options
  dhcp_does_arp_check = true
  lwip_dhcp = true
  in the Board Support Package Settings.
- Create an empty "tcpserver" project and use the supplied files to build the project:
  main.c
  echo.c

lscript.ld
platform.h
platform_config.h
platform_zynq.c

- Increase HEAP and STACK to 0A000 (Linker Script File, if values are lower)
- Create a new header file
  zbperiph.h

  containing the prototypes (set LEDs and read pushbuttons and switches)
  ```
  void zbSetLed(unsigned int ledval);
  unsigned int zbGetButSwitch();
  ```
- Create the corresponding "zbperiph.c" to implement the "drivers" to the onboard hardware.
- Build the TCP server. Use a unique ethernet hardware, specified in "main.c" (select < n> in range 1 to 8.
  ```
  /* the mac address of the board. this should be unique per board */
  unsigned char mac_ethernet_address[] =
  { 0x00, 0x0a, 0x35, 0x00, 0x01, 0x5< n> };
  ```
- Verify proper operation the ethernet communication by using the "putty" telnet session. Putty configuration should be similar like this:


PuTTY Configuration
Category:
Session
Logging
Terminal
Keyboard
Bell
Features
Window
Appearance
Behaviour
Translation
Selection
Colours
Connection
Data
Proxy
Telnet
Rlogin
SSH
Serial
Basic options for your PuTTY session
Specify the destination you want to connect to
Host Name (or IP address)
Port
192.168.178.66
7
Connection type:
Raw
Telnet
Rlogin
SSH
Serial
Load, save or delete a stored session
Saved Sessions
ZynqTcpServer
FB1 Postgres (kmz10)
MBTcpServer
PicoZynq
Spartan3e-Serial
ZYBO-Serial
Zynq-Serial
ZynqTcpServer
Load
Save
Delete
Close window on exit:
Always
Never
Only on clean exit
About
Open
Cancel


PuTTY Configuration
Category:
Session
Logging
Terminal
Keyboard
Bell
Features
Window
Appearance
Behaviour
Translation
Selection
Colours
Connection
Data
Proxy
Telnet
Rlogin
SSH
Serial
Options controlling the terminal emulation
Set various terminal options
Auto wrap mode initially on
DEC Origin Mode initially on
Implicit CR in every LF
Implicit LF in every CR
Use background colour to erase screen
Enable blinking text
Answerback to ^E:
PuTTY
Line discipline options
Local echo:
Auto
Force on
Force off
Local line editing:
Auto
Force on
Force off
Remote-controlled printing
Printer to send ANSI printer output to:
None (printing disabled)
About
Open
Cancel

- Good Luck!

***

Lab #06:

5.7 Design of an Industrial Drive Control System

5.7.1 Embedded System Requirements

A typical system for control purposes must include interfaces for sensors and actuators as well as communication elements. The structure is shown in fig. 1.49.

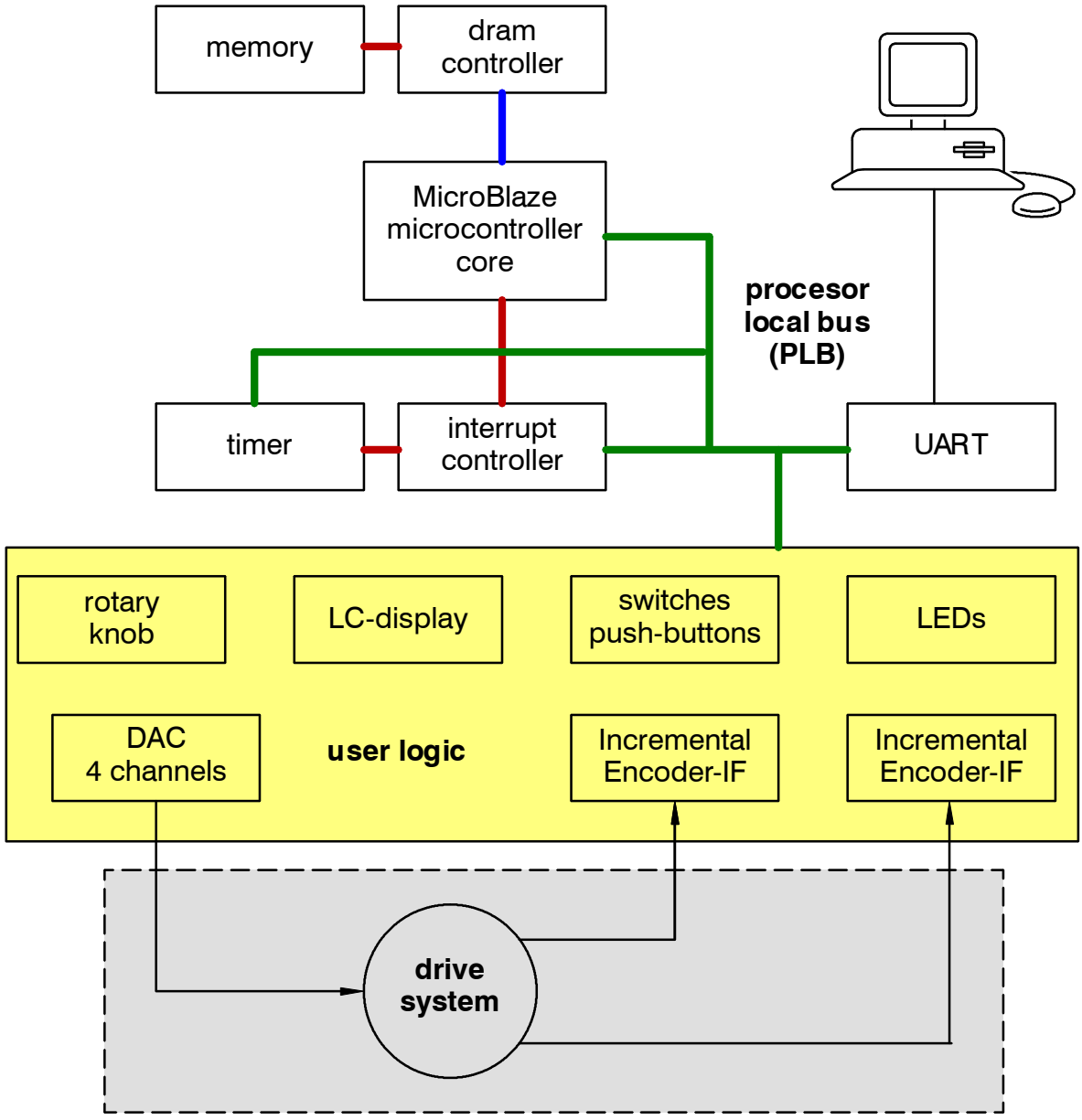


Figure 1.49: Embedded system for industrial drive control

All components reside on the same Chip. This ensures high performance, low price and straight-forward software development.

The analog output of the DAC supplies a torque signal to the drive while positions are capture by optical incremental encoders.

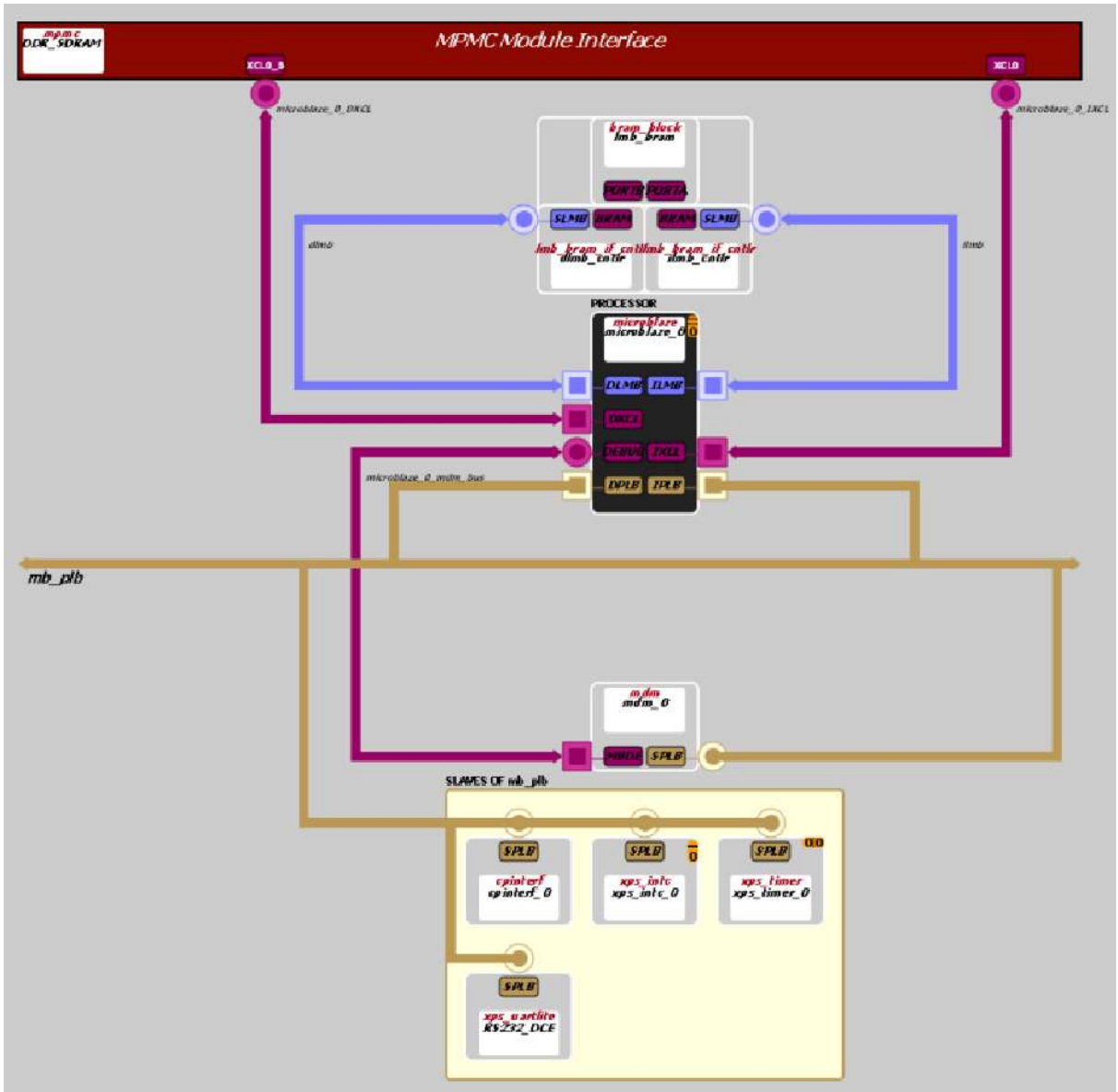


Figure 1.50: On chip block diagram

All components are for itself quite simple and fairly standard. These components need to connected carefully and require software drivers to access their functions for the main application software.

For the user logic, the following reference design could be used.

5.7.2 DAC Communication over SPI

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;

entity spis24 is
    Port ( sysclk : in  STD_LOGIC;
                              sreset : in STD_LOGIC;
           spistart : in  STD_LOGIC;
           spdin : in  STD_LOGIC_VECTOR (23 downto 0);
                              sck : out STD_LOGIC;
                              sdo : out STD_LOGIC;
           spidone : out  STD_LOGIC);
end spis24;


architecture Behavioral of spis24 is
CONSTANT TXBITS : NATURAL := 24;
TYPE state_type IS (idle, txs0, txs1, leadout);
SIGNAL  state, next_state : state_type;
SIGNAL  tx_count : NATURAL RANGE 0 TO TXBITS-1;
SIGNAL  sck_i, sdo_i : STD_LOGIC;

begin

          -- outputs
          sdo <= sdo_i;
          sck <= sck_i;

          -- sequential block of transmitter
          txproc : PROCESS(sysclk, sreset, state)
          BEGIN
                    IF rising_edge(sysclk) THEN
                              sck_i <= '0';
                              IF sreset='1' THEN
                                        state <= idle;
                              ELSE
                                        state <= next_state;
                                        IF state=idle THEN
                                                  tx_count <= 23;
                                                  sdo_i <= '0';
                                        ELSIF state=txs0 THEN
                                                  sdo_i <= spdin(tx_count);
                                        ELSIF state=txs1 THEN
                                                  sck_i <= '1';
                                                  tx_count <= tx_count - 1;
                                        END IF;
                              END IF;
                    END IF;
          END PROCESS txproc;
```

```
        -- combinational logic block of tansmitter
        clproc : PROCESS (state, spistart, tx_count)
        BEGIN
                next_state <= state;
                spidone <= '0';
                case state IS
                        WHEN idle =>
                                spidone <= '1';
                                IF spistart='1' THEN
                                        next_state <= txs0;
                                END IF;
                        WHEN txs0 =>
                                next_state <= txs1;
                        WHEN txs1 =>
                                IF tx_count=0 THEN
                                        next_state <= leadout;
                                ELSE
                                        next_state <= txs0;
                                END IF;
                        WHEN leadout =>
                                next_state <= idle;
                END CASE;
        END PROCESS clproc;

end Behavioral;
```

5.7.3 Rotary Knob Encoder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity rotenc_8 is
    Port ( rot_a : in  STD_LOGIC;
           rot_b : in  STD_LOGIC;
           rot_pos : out  STD_LOGIC_VECTOR (7 downto 0);
           sysclk : in  STD_LOGIC);
end rotenc_8;

architecture Behavioral of rotenc_8 is

signal rotary_in : std_logic_vector(1 downto 0);
signal rot_q1, rot_q2, rot_q1ff : std_logic;
signal cnt8 : std_logic_vector(7 downto 0) := "10000000";

begin

        rotary_filter: process(sysclk, rot_b, rot_a)
```

```
        begin
                if rising_edge(sysclk) then
                        rotary_in <= rot_b & rot_a;
                        case rotary_in is
                                when "00" =>
                                        rot_q1 <= '0';
                                        rot_q2 <= rot_q2;
                                when "01" =>
                                        rot_q1 <= rot_q1;
                                        rot_q2 <= '0';
                                when "10" =>
                                        rot_q1 <= rot_q1;
                                        rot_q2 <= '1';
                                when "11" =>
                                        rot_q1 <= '1';
                                        rot_q2 <= rot_q2;
                                when others =>
                                        rot_q1 <= rot_q1;
                                        rot_q2 <= rot_q2;
                        end case;
                end if;
        end process rotary_filter;

        -- flip-flop for count impulse
        impulse_ff: process(sysclk, rot_q1)
        begin
                if rising_edge(sysclk) then
                        rot_q1ff <= rot_q1;
                end if;
        end process impulse_ff;


        ud_counter: process (sysclk, rot_q1, rot_q2, rot_q1ff)
        begin
                if rising_edge(sysclk) then
                        if rot_q1='1' AND rot_q1ff='0' AND rot_q2='0'
then
                                if cnt8 /= x"FF" then
                                        cnt8 <= cnt8 + 1;
                                end if;
                        elsif rot_q1='1' AND rot_q1ff='0' AND
rot_q2='1' then
                                if cnt8 /= x"00" then
                                        cnt8 <= cnt8 - 1;
                                end if;
                        end if;
                end if;
        end process ud_counter;

        rot_pos <= cnt8;
```

```
end Behavioral;
```

5.7.4 Incremental Encoder (Generic Counter Size)

The incremental encoder interface has been programmed with low level primitives to obtain an efficient and small realization. Please note that the target system is Spartan-3, so 6-input LUTs cannot be used. They are not very efficient here anyway.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity increnc_gen is
          generic(IECTR_size : integer := 16);
          Port ( inc_a : in  STD_LOGIC;
                 inc_b : in  STD_LOGIC;
              inc_latch : in  STD_LOGIC;
                clr_pos : in  STD_LOGIC;
                inc_pos : out  STD_LOGIC_VECTOR (IECTR_size-1 downto 0);
                inc_clk : in  STD_LOGIC);
end increnc_gen;

architecture Mixed of increnc_gen is

signal  a0_0, a0_1, b0_0, b0_1 : std_logic;
signal  up_pulse, dn_pulse : std_logic;
signal  ictr, read_buf : std_logic_vector(IECTR_size-1 downto 0);

begin
          a0_ff: FD
          -- synthesis translate_off
          generic map (INIT => '0')
          -- synthesis translate_on
          port map (Q => a0_0,
          C => inc_clk,
          D => inc_a );

          a1_ff: FD
          -- synthesis translate_off
          generic map (INIT => '0')
          -- synthesis translate_on
          port map (Q => a0_1,
          C => inc_clk,
          D => a0_0 );
```

```
b0_ff: FD
-- synthesis translate_off
generic map (INIT => '0')
-- synthesis translate_on
port map (Q => b0_0,
C => inc_clk,
D => inc_b );

b1_ff: FD
-- synthesis translate_off
generic map (INIT => '0')
-- synthesis translate_on
port map (Q => b0_1,
C => inc_clk,
D => b0_0 );

updet_lut: LUT4
generic map (INIT => X"2814")
port map( I0 => a0_1,
    I1 => a0_0,
 I2 => b0_1,
          I3 => b0_0,
   O => up_pulse );

dwndet_lut: LUT4
generic map (INIT => X"4182")
port map( I0 => a0_1,
    I1 => a0_0,
 I2 => b0_1,
   I3 => b0_0,
 O => dn_pulse );

-- position measurement counter
process (inc_clk, clr_pos, inc_latch, up_pulse, dn_pulse)
begin
if rising_edge(inc_clk) then
                    ictr <= ictr;
                    read_buf <= read_buf;
                    if inc_latch='1' then
                              if clr_pos='1' then
                                        ictr <= (others => '0');
                              else
                                        read_buf <= ictr;
                              end if;
                    else
                              if up_pulse='1' then
                                        ictr <= ictr + 1;
                              end if;
                              if dn_pulse='1' then
                                        ictr <= ictr - 1;
                              end if;
```

```
                                end if;
            end if;
            end process;

   inc_pos <= read_buf;

end Mixed;
```

5.7.5 Tasks

The following tasks are required to obtain a running system.

- For the **Spartan-3E** create a new XPS project "mbctrl" with standard MicroBlaze and all available peripherals (see section 5.7.1).
- Add a user IP called "cpinterf" (Controller INTERFace) with 8 registers for PLB bus.
- Build the system (bitstream)
- Transfer the design to SDK and try the "Memory Test" application.
- Delete this application and create your own (empty) software project. You can use the same board support package as for "Memory Test". A new "Linker Script" is required to direct the linker to execute the program in external memory (DRAM).
- Modify your IP "cpinterf_0" with ISE and include peripheral support for
  - LEDs
  - Switches and Pushbuttons (leave out "Pushbutton South" since it is used for Reset)
  - rotary knob
  - LC display
  - incremental encoder interface
  - SPI interface for DAC communication
- Define the external signals in cpinterface_vx.x.mpd (peripheral description file)
- Import the IP back into XSP, make connections and address configurations
- Create external signals and specify them in the system UCF
  Your UCF might look as follows (only external signals shown). Of course the signal names may be different upon your selection:

```
###### User logic IO for cif
Net cpinterf_0_cif_Leds_pin<0> LOC=F12 | IOSTANDARD=LVTTL | SLEW=SLOW |
DRIVE=8;
Net cpinterf_0_cif_Leds_pin<1> LOC=E12 | IOSTANDARD=LVTTL | SLEW=SLOW |
DRIVE=8;
Net cpinterf_0_cif_Leds_pin<2> LOC=E11 | IOSTANDARD=LVTTL | SLEW=SLOW |
```

```
DRIVE=8;
Net cpinterf_0_cif_Leds_pin<3> LOC=F11 | IOSTANDARD=LVTTL | SLEW=SLOW |
DRIVE=8;
Net cpinterf_0_cif_Leds_pin<4> LOC=C11 | IOSTANDARD=LVTTL | SLEW=SLOW |
DRIVE=8;
Net cpinterf_0_cif_Leds_pin<5> LOC=D11 | IOSTANDARD=LVTTL | SLEW=SLOW |
DRIVE=8;
Net cpinterf_0_cif_Leds_pin<6> LOC=E9  | IOSTANDARD=LVTTL | SLEW=SLOW |
DRIVE=8;
Net cpinterf_0_cif_Leds_pin<7> LOC=F9  | IOSTANDARD=LVTTL | SLEW=SLOW |
DRIVE=8;
Net cpinterf_0_cif_Lcd_pin<0> LOC=M18 | IOSTANDARD=LVCMOS33 | DRIVE=4 |
SLEW=SLOW;
Net cpinterf_0_cif_Lcd_pin<1> LOC=L18 | IOSTANDARD=LVCMOS33 | DRIVE=4 |
SLEW=SLOW;
Net cpinterf_0_cif_Lcd_pin<2> LOC=L17 | IOSTANDARD=LVCMOS33 | DRIVE=4 |
SLEW=SLOW;
Net cpinterf_0_cif_Lcd_pin<3> LOC=M15 | IOSTANDARD=LVCMOS33 | DRIVE=4 |
SLEW=SLOW;
Net cpinterf_0_cif_Lcd_pin<4> LOC=P17 | IOSTANDARD=LVCMOS33 | DRIVE=4 |
SLEW=SLOW;
Net cpinterf_0_cif_Lcd_pin<5> LOC=R16 | IOSTANDARD=LVCMOS33 | DRIVE=4 |
SLEW=SLOW;
Net cpinterf_0_cif_Lcd_pin<6> LOC=R15 | IOSTANDARD=LVCMOS33 | DRIVE=4 |
SLEW=SLOW;
Net cpinterf_0_cif_Pushb_pin<0> LOC=D18 | IOSTANDARD=LVTTL | PULLDOWN;
Net cpinterf_0_cif_Pushb_pin<1> LOC=V4 | IOSTANDARD=LVTTL | PULLDOWN;
Net cpinterf_0_cif_Pushb_pin<2> LOC=H13 | IOSTANDARD=LVTTL | PULLDOWN;
Net cpinterf_0_cif_Switch_pin<0> LOC=L13 | IOSTANDARD=LVTTL | PULLUP;
Net cpinterf_0_cif_Switch_pin<1> LOC=L14 | IOSTANDARD=LVTTL | PULLUP;
Net cpinterf_0_cif_Switch_pin<2> LOC=H18 | IOSTANDARD=LVTTL | PULLUP;
Net cpinterf_0_cif_Switch_pin<3> LOC=N17 | IOSTANDARD=LVTTL | PULLUP;
Net cpinterf_0_cif_Rotbtn_pin LOC=V16 | IOSTANDARD=LVTTL | PULLDOWN;
Net cpinterf_0_cif_Rotbk_pin<0> LOC=K18 | IOSTANDARD=LVTTL | PULLUP;
Net cpinterf_0_cif_Rotbk_pin<1> LOC=G18 | IOSTANDARD=LVTTL | PULLUP;
Net cpinterf_0_cif_SF_CE0_pin LOC=D16 | IOSTANDARD=LVCMOS33 | DRIVE=4 |
SLEW=SLOW;
Net cpinterf_0_cif_SPI_MOSI_pin LOC=T4 | IOSTANDARD=LVCMOS33 | DRIVE=8 |
SLEW=SLOW;
Net cpinterf_0_cif_SPI_SCLK_pin LOC=U16 | IOSTANDARD=LVCMOS33 | DRIVE=8 |
SLEW=SLOW;
Net cpinterf_0_cif_DAC_CS_pin LOC=N8 | IOSTANDARD=LVCMOS33 | DRIVE=8 |
SLEW=SLOW;
Net cpinterf_0_cif_DAC_CLR_pin LOC=P8 | IOSTANDARD=LVCMOS33 | DRIVE=8 |
SLEW=SLOW;
Net cpinterf_0_cif_SPI_SS_B_pin LOC=U3 | IOSTANDARD=LVCMOS33 | DRIVE=6 |
SLEW=SLOW;
Net cpinterf_0_cif_AMP_CS_pin LOC=N7 | IOSTANDARD=LVCMOS33 | DRIVE=6 |
SLEW=SLOW;
Net cpinterf_0_cif_AD_CONV_pin LOC=P11 | IOSTANDARD=LVCMOS33 | DRIVE=6 |
SLEW=SLOW;
```

```
Net cpinterf_0_cif_FPGA_INIT_B_pin LOC=T3 | IOSTANDARD=LVCMOS33 | DRIVE=4 |
SLEW=SLOW;
```

- Rebuild the system and transfer the hardware information to SDK
- Write software drivers to access the functions of your IP
- Write test programs to verify proper operation.
- Congratulation: You have just created your own embedded control computer. A similar system from vendors like NI (National Instruments) or DSpace costs more than 15.000€!

A running system might look like this:

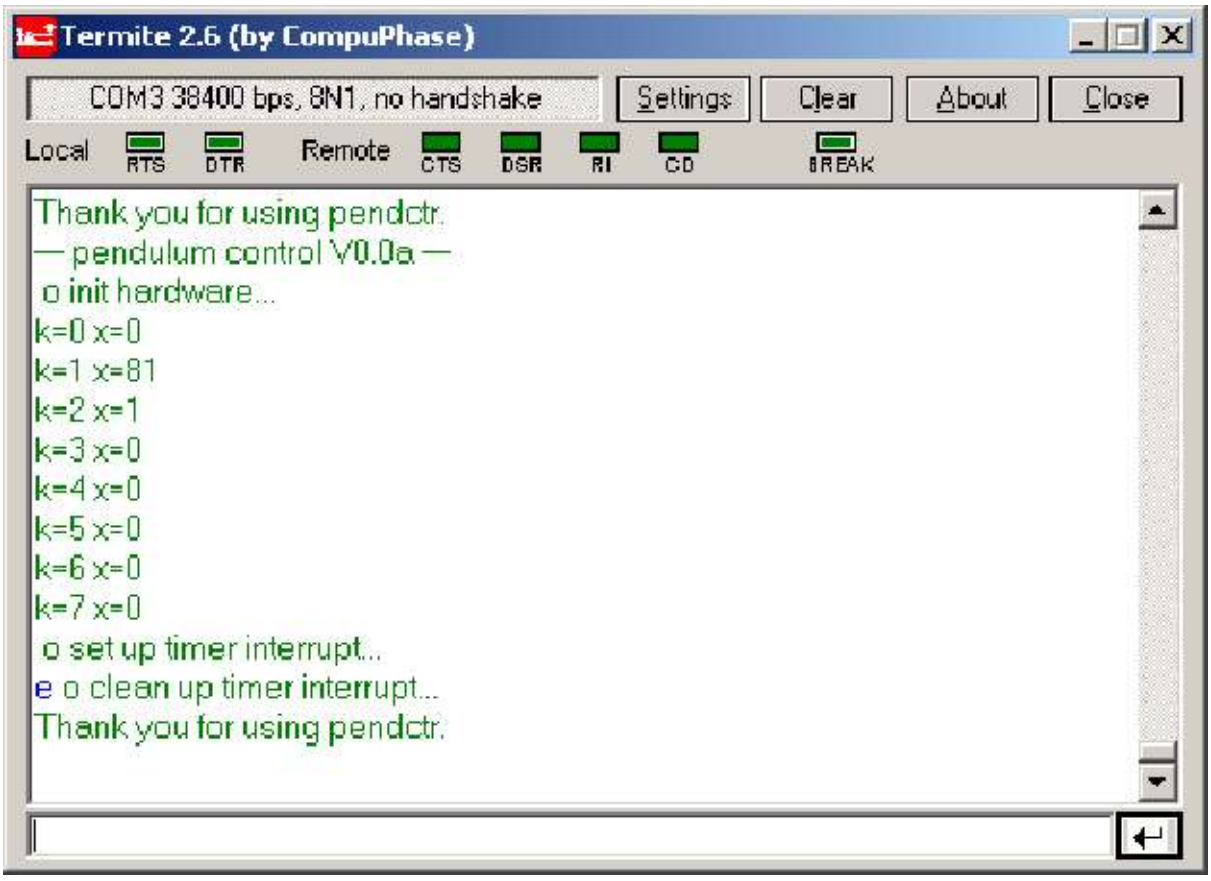


Figure 1.51: Embedded controller is running

***

Lab #07:

5.8 Hardware Accelerated DSP (Ultra DSP)

Digital signal Processing (DSP) for high sampling rates can be executed on programmable logic under software control from the microprocessor.

An 8. order bandpass filter calculation will be carried out in an AXI IP. Verification will be done by a C program in the ARM core.

For hardware computation the Transposed Direct Form I (TDF-I) is a suitable form as this filter type has short combinational paths (only multiply/add with is executable within one clock cycle). Moreover integrated MACs (Multiplier/Accumulator) are an ideal element for computation. The TDF-I general structure is shown below.

The filter's discrete transfer function is given by

$$H_{BP}(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_7 z^{-7} + b_8 z^{-8}}{1 + a_1 z^{-1} + a_2 z^{-2} \ldots a_7 z^{-7} + a_8 z^{-8}} \,. \tag{1.5}$$

The coefficient a_0 can always be made $a_0 = 1$.

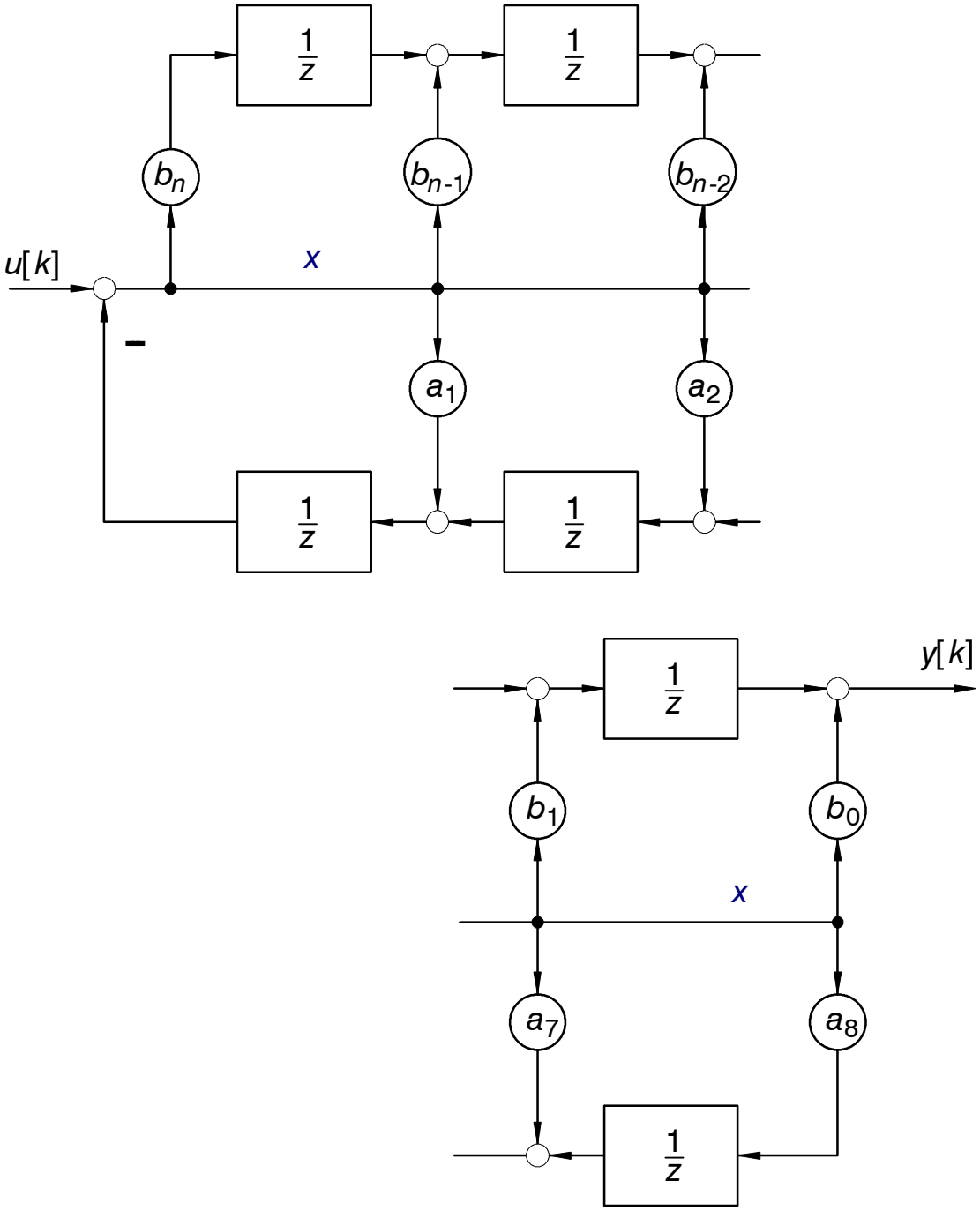


Figure 1.52: Transposed Direct Form I (TDF-I)

The bandpass filter has a bandwidth from 0.1 to 0.5 (relative to Nyquist frequency) and can be computed in Matlab with

```
[B, A] = butter(4,[0.1 0.5]);
```

The values of B and A are:

B = [0.0466 0 -0.1863 0 0.2795 0 -0.1863 0 0.0466] ,

A = [1 -3.4744 5.5467 -5.7183 4.3889 -2.4641 0.9263 -0.2203 0.0301] .

It can be seen that $a_0 = 1$ and every second coefficient in B is 0. Multiplication with zeros can be eliminated so a final diagram ready for VHDL programming is show in fig. 1.53.

You will need the following operations in **VHDL** (from `ieee.numeric_std.all`):

Conversion from std_logic_vector to integer:

```
to_integer(signed(sdtlv_value))
```

Conversion from integer to std_logic_vector:

```
std_logic_vector(to_signed(int_value, dest'length))
```

Right shift of integers (here 14 bits):

```
int_val / 2**14
```

For verification in **C**:

16 bit integer: `short int`
32 bit integer: `int`

Right shift of integer (here 14 bits): `int_val <<= 14;`

All computation in hardware and C must be carried out in fixed point arithmetic.

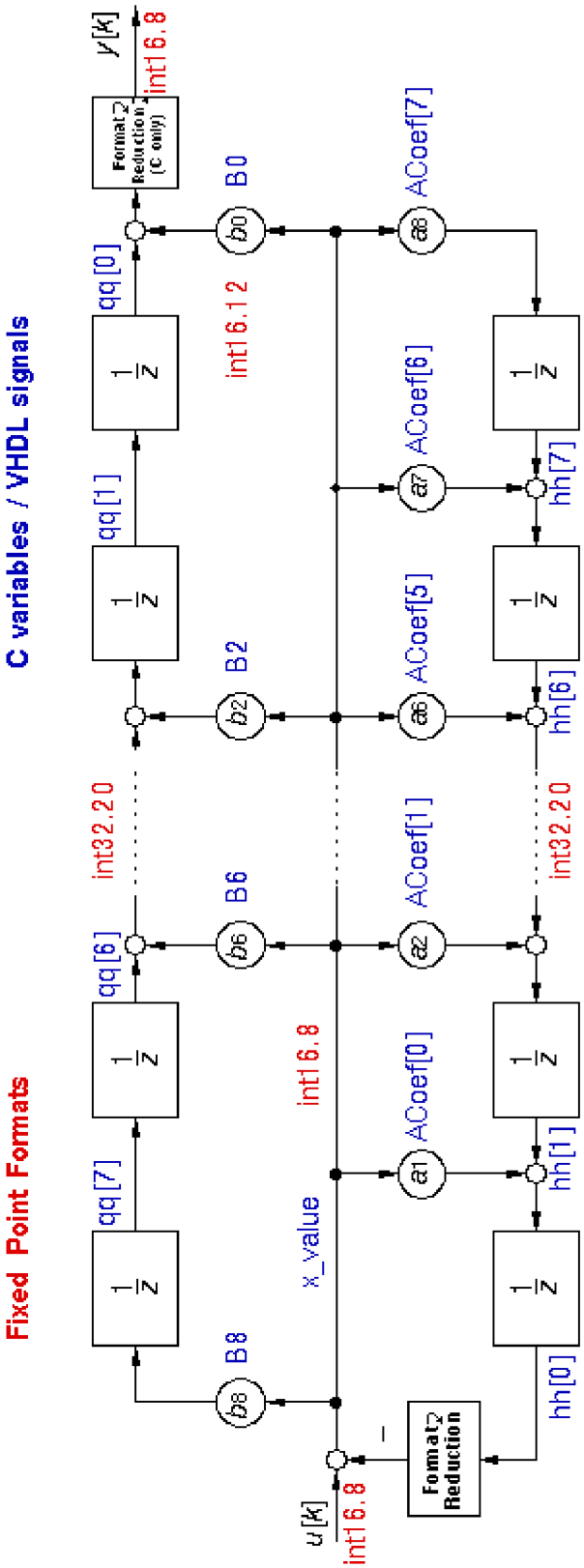


Figure 1.53: VHDL structure of 8. order Butterworth bandpass filter

***

Lab #08:

5.9 Software Development

The use of hardware components require drivers to access the devices in the application program. Since all hardware is *memory-mapped* in the microcontroller address space the driver can directly access the interface registers via it's decoded memory address.

The base address of each device is accessible by the automatically generated header file "xparameters.h". Therefore, it becomes necessary to specify

```
#include "xparameters.h"
```

in the top section of each C module. The compiler has the appropriate include directory in the search path. The complete list of required header files is as follows.

```
#include <stdio.h>                  (for print, xil_printf ...)
#include "platform.h"               (enable and disable chache)
#include "xbasic_types.h"           (type definitions like u32...)
#include "xparameters.h"            (see notes)

#include "mb_interface.h"           (enable and disable interrupts ...)
#include "xtmrctr_l.h"              (timer)
#include "xintc_l.h"                (interrupt controller)
#include "xuartlite_l.h"            (UART low level functions)
```

5.9.1 Installing the Timer Interrupt Handler

Using an interrupt for the included timer involves several steps to program the timer in repeat (reload) mode and to program the interrupt controller to acknowledge the timer interrupt signal.

```
/*
** initialize timer, interrupt controller and register handler
*/
static void CiSetupInterrupt() {
          // disable everything in timer
[1]       XTmrCtr_SetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0, 0);
          // set the load register
[2]       XTmrCtr_SetLoadReg(XPAR_XPS_TIMER_0_BASEADDR, 0, CI_COUNTER_10MS);
          // load counter
[3]       XTmrCtr_SetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0,
                              XTC_CSR_LOAD_MASK);
          // setup AXI interrupt
```

```
[4]       XIntc_RegisterHandler(XPAR_INTC_0_BASEADDR,
                                XPAR_INTC_0_TMRCTR_0_VEC_ID,
                                CITimerInterrupt,
                                (void *)XPAR_XPS_TIMER_0_BASEADDR);
[5]       XIntc_MasterEnable(XPAR_INTC_0_BASEADDR);
[6]       XIntc_EnableIntr(XPAR_INTC_0_BASEADDR,
                                XPAR_XPS_TIMER_0_INTERRUPT_MASK);
          // start counter
[7]       XTmrCtr_SetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0,
                                XTC_CSR_ENABLE_TMR_MASK
                            | XTC_CSR_AUTO_RELOAD_MASK
                            | XTC_CSR_DOWN_COUNT_MASK
                            | XTC_CSR_ENABLE_INT_MASK);
[8]       microblaze_enable_interrupts();
}
```

[1] Disable the timer completely to prevent any signals from timer.

[2] Provide a timer count value for the timer load register. Since the timer will count down to zero the time is < load value> / < counter frequency> , where counter frequency is 50 MHz.

[3] The count register is actually loaded with the supplied value from [2].

[4] Register the handler (the interrupt service function that should be called, when an interrupt occurs.

[5] Enable the interrupt controller (pass interrupt to MicroBlaze)

[6] Enable the timer interrupt.

[7] Start the timer with the required parameters (self-explanatory names).

[8] This enables the interrupts on MicroBlaze.

5.9.2 Disabling the Timer Interrupt

The reverse operations are required to stop interrupts. This function should be called before the MicroBlaze terminates the program.

```
/*
** cleanup timer interrupt stuff
*/
static void CiCleanupInterrupt() {
[1]       microblaze_disable_interrupts();
[2]       XIntc_DisableIntr(XPAR_INTC_0_BASEADDR,
                                XPAR_XPS_TIMER_0_INTERRUPT_MASK);
[3]       XIntc_MasterDisable(XPAR_INTC_0_BASEADDR);
```

```
            // disable everything in timer
[4]         XTmrCtr_SetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0, 0);
}
```

[1] Disable interrupts on MicroBlaze.

[2] Disable the interrupts for the timer on interrupt controller

[3] Disable all interrupts on interrupt controller

[4] Stop timer and disable interrupt signal generation

After calling *CiCleanupInterrupt();* it is safe to terminate a program. Use this function before *cleanup_platform();* .

All code fragments are found in the *pbrl_files* directory under *HARDWARE* or *SOFTWARE*, respectively.

***

6 Bibliography

[1] Ashenden, Peter J.: The Designer's Guide to VHDL, 3rd. Ed.
Morgan Kaufmann, 2008

[2] Bond, David: More than you ever wanted to know about GCC, GAS and ELF.
Tachion Software LLC, Nashville, TN, 2002

[3] Crocket, Louise H., Elliot Ross A., Enderwitz, Amrtin A., Stewart, Robert W.: The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq–7000 All Programmable SoC, First Edition, Strathclyde Academic Media, 2014

[4] Bashir M. Al-Hashimi: System-on-Chip: Next Generation Electronics.
Institution of Electrical Engineers, 2006

[5] John F. Wakerly: Digital Design, Principles & Practices.
Prentice Hall, 2001

[6] Volnei A. Predroni: Circuit Design and Simulation with VHDL, 2nd. Ed.
MIT Press, 2010

[7] Ricardo Reis, Marcelo Lubaszewski and Jochen Jess: Design of Systems on a Chip: Design and Test.
Springer, 2010

[8] Roger Lipsett, Carls Schaefer, Cary Ussery: VHDL Hardware Description and Design.
Kluwer Academic 1990

[9] J. Reichardt, B. Schwarz: VHDL-Synthese.
Oldenbourg, 2001

[10] Xilinx Embedded Systems Design Documentation
Xilinx, 2012

***