# Plasma Microprocessor

Adrian P. P. Jongenelen

October 2013

Abstract

The Plasma Microprocessor is a small 32-bit RISC microprocessor. It is a design written in synthesisable VHDL to enhance the capability of FPGA-based systems.

FPGAs are programmable logic devices especially suited to customised interfaces or highly parallelised computation tasks. However, in many cases they lack the simplistic configurability and PC interfacing of common microprocessors. This deficiency can be overcome by instantiating a soft microprocessor core such as Plasma within the FPGA device.

The advantage of the Plasma Microprocessor is that the core and associated tool chain are free and open source, in contrast to the various microprocessor cores provided by FPGA manufacturers.

Source code for this project and related material is available online from <https://github.com/adrianj/Plasma>. This document and any attached files refer to the v1\_1 branch of the project.

CONTENTS

[1. Introduction 2](#_Toc369694064)

[1.1. Disclaimer 2](#_Toc369694065)

[2. Plasma Core Details 4](#_Toc369694066)

[2.1. Core Behaviour 4](#_Toc369694067)

[2.2. Memory Mapped I/O 7](#_Toc369694068)

[2.3. DDR2 RAM and Cache 8](#_Toc369694069)

[3. Plasma Peripherals 13](#_Toc369694070)

[3.1. Universal Asynchronous Receiver/Transmitter 14](#_Toc369694071)

[3.2. Interrupts 16](#_Toc369694072)

[3.3. LEDs, Switches, Push Buttons and PMOD General I/O 19](#_Toc369694073)

[3.4. Pseudo Random Number Generator 19](#_Toc369694074)

[3.5. Counter 20](#_Toc369694075)

[3.6. Cache Statistics 20](#_Toc369694076)

[3.7. Flash RAM 21](#_Toc369694077)

[3.8. General Purpose FIFO 21](#_Toc369694078)

[4. C code tool setup and boot loader code 23](#_Toc369694079)

[4.1. Compilation Tools Setup 23](#_Toc369694080)

[4.2. Boot Loader 27](#_Toc369694081)

[5. C# boot loader app 28](#_Toc369694082)

[6. Testing 29](#_Toc369694083)

[7. Example designs 30](#_Toc369694084)

[7.1. PlasmaBlockRAM ISE Project 30](#_Toc369694085)

[7.2. PlasmaAtlysDDR ISE Project 30](#_Toc369694086)

[7.3. PlasmaDataAcquisition ISE Project 31](#_Toc369694087)

[7.4. Example C Project 33](#_Toc369694088)

[7.5. PlasmaDataAcquisition C Project 33](#_Toc369694089)

[REFERENCES 35](#_Toc369694090)

[Appendix A – Assembly Boot Code 1](#_Toc369694091)

[Appendix B – Plasma Header File 6](#_Toc369694092)

[DOCUMENT CONTROL SHEET 1](#_Toc369694093)

[Initial Distribution: 2](#_Toc369694094)

## 

### Introduction

Field Programmable Gate Array (FPGA) devices consist of an array of logic gates that can be programmed to directly implement an application specific integrated circuit. Compared to more traditional controller devices such as microprocessors they are well suited to digital signal processing applications due to their parallel architecture. They are also well suited to implementing non-standard or custom communications protocols due to the ability to provide I/O timing accurate to within a clock cycle with high clock speeds. For example, the Xilinx Spartan‑6 XC6SLX45 mounted on the Digilent ATLYS™ Spartan‑6 FPGA Development Board is capable of output switching times as low as 4.1 ns (-2 Speed Grade, 3.3 V CMOS) and internal clock speeds up to 375 MHz [1].

Microprocessors have useful features that an FPGA does not have by default, for example, built-in PC communications and the ability to perform sequential tasks in a manner easily defined through software code. However, that is not to say an FPGA cannot do this, only that it must be designed to do so by implementing a “soft” microprocessor core within it.

The Plasma CPU, developed by Steve Rhoads and made publically available through OpenCores.org (www.opencores.org) is “… a small synthesisable 32-bit RISC microprocessor. … The Plasma CPU executes all MIPS I™ user mode instructions except unaligned load and store operations.” [2].

This report documents extensions to the design available at OpenCores.org, targeting the Digilent ATLYS™ Spartan-6 FPGA Development Board [4]. The development tool chain for a Microsoft Windows 7 environment is described and boot loader and test programs are provided.

Occasionally this report will refer to the Plasma example design. This is the reference design for which the source code is provided and is intended to be used as a starting point for future designs. The appeal of generating a soft microprocessor core is that it can be application specific, i.e., as full-featured or lightweight as desired. It is expected that future designs will modify the example design as appropriate.

Source code for this project and related material is available online from <https://github.com/adrianj/Plasma>. This document and any attached files refer to the v1\_1 branch of the project.

#### Disclaimer

The following is the disclaimer and legal notice provided by the original author on the OpenCores.org website:

“MIPS ® is a registered trademark and MIPS I™ is a trademark of MIPS Technologies, Inc. [3] in the United States and other countries. MIPS Technologies, Inc. does not endorse and is not associated with this project. OpenCores and Steve Rhoads are not affiliated in any way with MIPS Technologies, Inc.

The Plasma CPU project has been placed into the public domain by its original author and is free for commercial and non-commercial use.

This software is provided "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed.”

Identical to the original disclaimer above, this project has been placed into the public domain “as is” and is free for commercial and non-commercial use.

### Plasma Core Details

The Plasma core is based on the Von Neumann architecture model, where instruction code and data reside in the same memory space. An individual instruction takes at least three clock cycles to execute. Instructions are pipelined so that the instruction rate is typically equal to the clock rate, though the core will pause if needed, for example when reading or writing memory. Figure 1 shows the relationship between the major elements of the Plasma core [2].

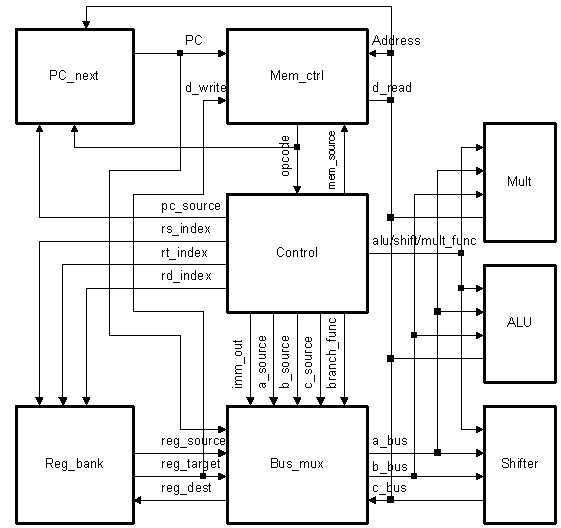


Figure . Diagram of Plasma Core (http://opencores.org/ project,plasma)

#### Core Behaviour

Explaining the behaviour of the core is perhaps best achieved through an example. The code listing below instructs the core to read a value from memory location 0x20000070, add 1 to it and then write it to memory location 0x20000060.

7b0: 3c022000 lui $v0,0x2000

7b4: 34420070 ori $v0,$v0,0x70

7b8: 8c500000 lw $s0,0($v0)

7bc: 3c022000 lui $v0,0x2000

7c0: 34420060 ori $v0,$v0,0x60

7c4: 26100001 addiu $s0,$s0,1

7c8: ac500000 sw $s0,0($v0)

Figure 2 shows a selection of signals related to the Plasma core implementing the code listing above. The “bitwise OR immediate” instruction (“ori”, opcode 001101) is highlighted in black to demonstrate the basic 3 stage pipeline.

* Stage 0: “Mem\_ctrl” block sets memory address to current program counter (0x7b4);
* Stage 1: Opcode is retrieved from memory (0x34420070).
* Stage 2:
  + The “Control” entity decodes the opcode and sets appropriate ALU function flags (0x5), register source codes (0, 1 and 1), register source and destination indices (both 0x02 for register $v) and extracts the immediate value (0x0070);
  + The “Reg\_bank” returns the contents of the source register at index 0x02;
  + The “Bus\_mux” puts the register source onto the A bus (code 0) and the immediate onto the B bus (code 1);
  + The “ALU” performs the bitwise OR operation;
  + The “Bus\_mux” puts the ALU output onto the C bus, aka register destination (code 1);
  + The “Reg\_bank” performs a write to destination index 0x02.

Highlighted in dark grey is the “Load Word” instruction (“lw”, opcode 100011). This instruction requires a minimum of 4 pipeline stages as the “Mem\_ctrl” block instructs the core to pause (note the logical high of “mem\_pause”) to perform the read instruction. In this case, it is a memory mapped peripheral being read which issues its own memory pause to account for latency in reading the peripheral. This pausing mechanism allows the Plasma core to directly read from external memory locations of varying latency.

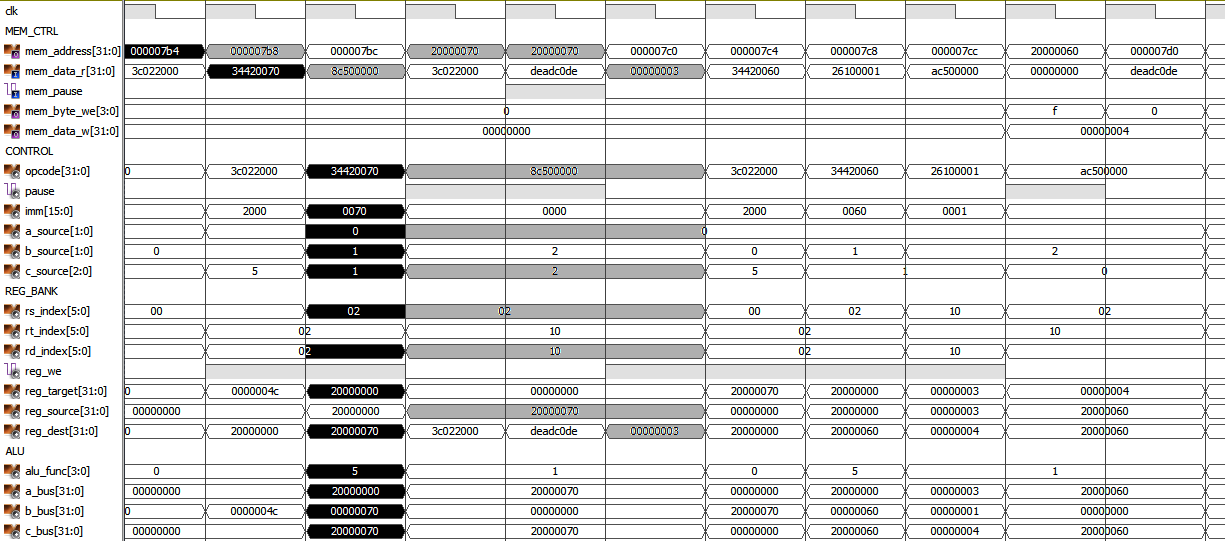


Figure . Plasma Core Signals – ORI operation highlighted in black, LW operation in dark grey.

#### Memory Mapped I/O

The Plasma core can directly read from any 32-bit word aligned memory location with an address width of 32 bits; however ignoring the bottom two bits for word alignment. This gives a theoretically addressable memory space of 230 unique 32-bit wide locations, or 4 GB. In practice, the core as implemented for this project has certain sections of memory reserved as per Table 1.

Table . Reserved memory space.

|  |  |  |
| --- | --- | --- |
| Address Offset | Length (bytes) | Description |
| 0x00000000 | 0x2000 | Default program code. |
| 0x00002000 | 0x6000 | Internal (in-chip) general purpose memory. |
| 0x00008000 | 0x1FFF8000 | Unused. Reads undefined. |
| 0x20000000 | 0x10000000 | Processor peripherals. Refer Chapter 0. |
| 0x30000000 | 0x10000000 | External Bus. |
| 0x40000000 | 0x20000000 | External general purpose memory. |
| 0x60000000 | 0xC0000000 | Unused. Reads will return 0. |

Unless unused or otherwise noted all memory locations can be both read and written, for example within various peripheral functions.

To read and write directly to/from memory in the C programming language the below macros may prove useful, and indeed are used in the examples elsewhere in this document.

#define MemoryRead(A) (\*(volatile unsigned int\*)(A))

#define MemoryWrite(A,V) \*(volatile unsigned int\*)(A)=(V)

These macros and others, as well as definitions for the various peripherals included in the example designs are listed in the header file: “plasma.h”. This header file is included in

##### Program Code

Within the design there is a dedicated block of RAM that is initialised through the FPGA binary file. At address offset zero it is the very first routine that begins as soon as the global reset is released. Most designs will make use of this space to host a boot loader program that performs initialisation functions and loads other programs.

Software developers should take note that this region is writable, so should take care to not overwrite the boot loader code unless they expressly intend to do so and are willing to accept the consequences. The region is writable and quick to access (no additional latency delay), a feature that can be taken advantage of by initialising the call stack here.

##### Internal General Purpose Memory

This section of memory is an uninitialised continuation of the Program Code block. It can be used for any purpose; one suggestion is to make the call stack larger.

##### Processor Peripherals

Various peripherals, for example UART communications, are mapped directly into memory from offset 0x20000000. Reading from this section of memory incurs an extra delay of at least one clock cycle, possibly more depending on the peripheral. Not all memory locations are occupied. For more information on the peripherals and their addresses refer to Chapter 0.

##### External Bus

The External Bus is a simple 32-bit wide bidirectional data bus that can be used for integrating the Plasma microprocessor into application specific designs. The interface consists of two 32-bit wide data signals: one input, one output; a 28-bit wide address; an active high write enable; and an active high read enable.

##### External General Purpose Memory

From offset 0x4000000 to 0x60000000 is enough space to reference 512 MB of general purpose memory separate from the on-chip memory within the Plasma core. For simple designs, or designs where very precise timing is needed this external (external with respect to the Plasma core) memory can be directed to more block RAM resources. For designs requiring a large amount of memory, this address space can be mapped to an appropriate controller for an external memory chip, for example, the 1 GBit DDR2 memory chip present on the ATLYS board.

#### DDR2 RAM and Cache

The ATLYS board features a Micron MT47H64M16-25E 1 GBit (128 MB) DDR2 memory chip. The Spartan-6 family of FPGAs has dedicated on-chip Memory Controller Blocks (MCBs) for interfacing to DDR RAM. The controller core for this project has been generated using the Xilinx Memory Interface Generator (MIG) tool [**Error! Reference source not found.**] with the following parameters:

* Memory Type: DDR2 SDRAM on Bank 3;
* Frequency 300 MHz (3333 ps period);
* MT47H64M16XX-25E part;
* Memory Options:
  + Fullstrength Drive Strength;
  + 50 Ohms termination;
  + DQS# Enabled;
  + High temperature self-refresh rate disabled.
* Three 32-bit bi-directional ports (Ports 0, 1 and 2);
* Custom Arbitration (see below);
* FPGA Options:
  + Class II SSTL Output Drive Strength;
  + Calibrated Input Termination (RZQ = L6, ZIO = C2);
  + Debug Signals disabled;
  + System Clock Single-Ended.

The memory controller is designed with 3 32-bit wide bi-directional ports, numerated as 0, 1 and 2. The custom arbitration is set up to service Port 0 twice as frequently as Ports 1 and 2, i.e., time slots cycle through 0108, 1028, 0208, 2018. In the final example design only Port 0 is used, but future designs may require RAM interfaces that are logically separate from the Plasma core.

The memory interface includes a PLL to generate the memory clock (300 MHz) from the input clock generated by the on-board oscillator (100 MHz). The user side of the memory interface is configured to operate at 50 MHz, and this becomes the global clock net for the Plasma core and most peripherals. To generate these clocks additional modification of the source file produced by the MIG tool is required. For a design named “atlys\_ddr2” a source file “atlys\_ddr2.vhd” is produced where a number of constants are defined within the architecture. These are:

* C3\_CLKFBOUT\_MULT = 6 (VCO frequency = 6 × 100 = 600 MHz);
* C3\_DIVCLK\_DIVIDE = 1;
* C3\_CLKOUT0\_DIVIDE = 1 (VCO frequency = memory clock × 2);
* C3\_CLKOUT1\_DIVIDE = 1;
* C3\_CLKOUT2\_DIVIDE = 12 (user clock = VCO frequency /12 = 50 MHz);
* C3\_INCLK\_PERIOD = 10000; (input clock period in ps).

Note that the memory interface on the ATLYS board is theoretically capable of running at a higher frequency than this; though this has not been attempted.

Using the DRAM memory interface is more complicated than on-chip block RAM or SRAM. Each port has three First-in-first-out (FIFO) buffers for command instructions, read data and write data. In the source files these buffers are accessed with signals prefixed by “cmd”, “rd” and “wr” respectively, although in some cases they can be connected to the same signal, for example the clocks. To simplify this interface an additional entity called “ddr2\_cache” has been designed that presents to the Plasma core a set of signals that behave more like internal block RAM.

Figure 3 shows the interaction between DRAM memory chip signals and the Plasma core via the DDR2 cache entity. To incorporate the DRAM controller in a design requires a modification to the top level entity regarding the global clock and reset signals. The global clock net is provided as an output from the DRAM controller and the asynchronous reset signal is active while the DRAM internal reset is high or in the calibration state.

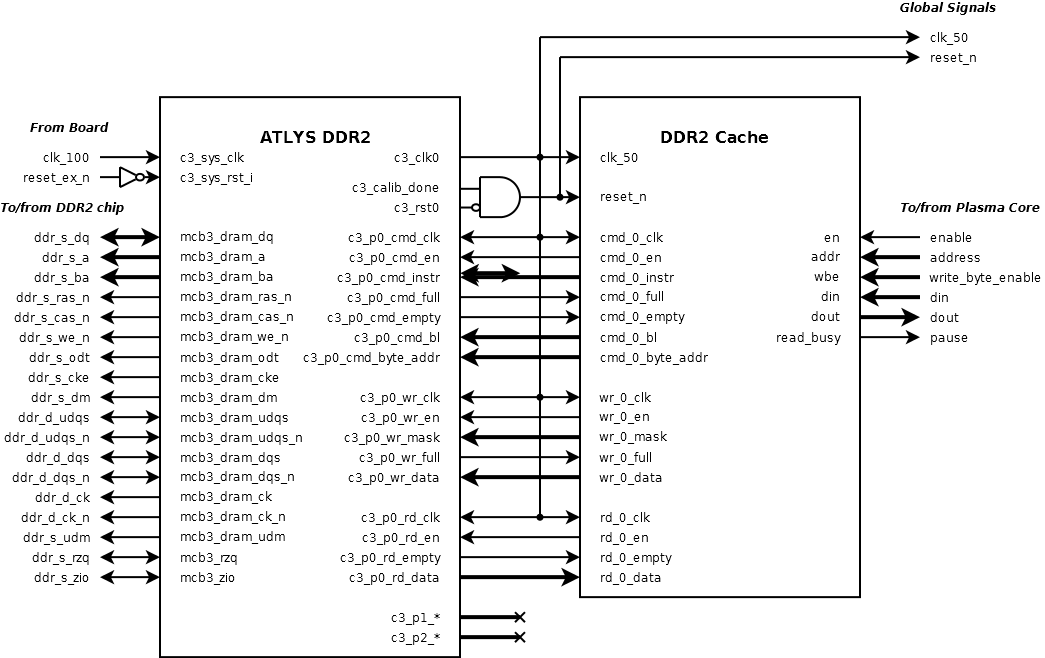


Figure . ATLYS DDR2 and cache connections.

##### DDR2 Constraints

The naming convention of signals to/from the ATLYS DDR2 entity would be familiar to a reader who has used the MIG tool. The higher level version of the signals that connect to the memory chip are slightly renamed with the prefix “ddr\_s” or “ddr\_d”; referring to single ended or differential signals respectively. The purpose of this is to simplify the pin I/O standard declaration within the Xilinx ISE user constraints file (UCF). The following lines are required to be added to the UCF file:

NET “ddr\_d\_\*” IOSTANDARD = DIFF\_SSTL18\_II;

NET “ddr\_s\_\*” IOSTANDARD = SSTL18\_II;

NET “ddr\_\*dq\*” IN\_TERM = NONE;

NET “\*/selfrefresh\_mcb\_mode” TIG;

NET “\*/c?\_pll\_lock” TIG;

INST “\*/mcb\_soft\_calibration\_inst/DONE\_SOFTANDHARD\_CAL\*” TIG;

NET “\*/mcb\_soft\_calibration\_inst/CKE\_Train” TIG;

The first set of constraints explicitly state what I/O standard the pins should be using and that the inputs should have no termination. The second set is for the benefit of the static timing analysis tool, instructing it to ignore timing constraints for a particular set of signals. Analysing the timing of these signals is ineffective as they typically cross clock domains.

Pin locations for the various signals can be found in the ATLYS Board Reference Manual [6].

##### DDR Cache

Truly random access reading from a high density DRAM memory chip such as DDR2 is slow (high latency) compared to on-chip block RAM, primarily because of the time taken to precharge and activate a row. To alleviate this deficit, memory chips are designed for high speed burst access where consecutive memory locations can be read more efficiently. The cache, also known as CPU cache or Level 1 cache, acts as an intermediate storage medium that reduces the number of burst reads from the memory chip and thereby reduces the average latency experienced by the Plasma core. For more information on CPU cache design see [7].

This project implements a cache of size 4 kB organised into 64 lines of 64 bytes. It is a write-though cache that is fully associative using a content addressable memory. A first-in-first-out (FIFO) replacement policy (also known as round robin) is used due to its low overhead and simplicity to implement.

### Plasma Peripherals

The Plasma example designs have a number of peripheral units incorporated within it. Some of these could be considered essential, for example, the UART that can be used to communicate with a PC. Others are included as an example of what is possible, for example, the pseudo random number generator.

Software interaction with peripherals is accomplished by reading and writing registers mapped to specific memory locations. Table 2 lists the peripherals mapped in the default example design. When referring to bit positions within the registers, the least significant bit, bit 0, is the right-most bit. Further details on each peripheral can be found throughout this chapter.

Table . Peripheral Register Map.

|  |  |  |
| --- | --- | --- |
| Address Offset | Label | Description |
| 0x20000000 | UART\_TX | UART transmit data |
| 0x20000004 | UART\_RX | UART receive data |
| 0x20000008 | UART\_STATUS | UART status |
| 0x2000000C | UART\_CONTROL | UART control |
| 0x20000010 | UART\_BAUD\_DIV | UART baud rate divider |
| 0x20000040 | IRQ\_STATUS | Interrupt status |
| 0x20000044 | IRQ\_STATUS\_CLR | Interrupt status clear |
| 0x2000004C | IRQ\_VECTOR | Interrupt vector |
| 0x20000050 | IRQ\_MASK | Interrupt enable mask |
| 0x20000054 | IRQ\_MASK\_SET | Interrupt enable mask set |
| 0x20000058 | IRQ\_MASK\_CLR | Interrupt enable mask clear |
| 0x20000060 | LEDS\_OUT | LEDs output |
| 0x20000070 | SWITCHES | Switches input |
| 0x20000074 | BUTTONS | Push buttons input |
| 0x20000078 | RAND\_GEN | Pseudo random number generator |
| 0x20000080 | PMOD\_OUT | PMOD general purpose output |
| 0x20000090 | PMOD\_IN | PMOD general purpose input |
| 0x20000094 | PMOD\_TRIS | PMOD general purpose tristate control |
| 0x200000A0 | COUNTER | Counter value |
| 0x200000A4 | COUNTER\_PS | Counter prescale |
| 0x200000A8 | COUNTER\_TS | Counter terminal count value |
| 0x200000B0 | CACHE\_HITCOUNT | DRAM cache hit count |
| 0x200000B4 | CACHE\_READCOUNT | DRAM cache read count |
| 0x200000C0 | FLASH\_CON | Flash RAM control |
| 0x200000C4 | FLASH\_DATA | Flash RAM data |
| 0x200000C8 | FLASH\_TRIS | Flash RAM tristate control |
| 0x200000D0 | FIFO\_DIN | General Purpose FIFO receive data |
| 0x200000D4 | FIFO\_DOUT | General Purpose FIFO transmit data |
| 0x200000D8 | FIFO\_CON | General Purpose FIFO status and control |

#### Universal Asynchronous Receiver/Transmitter

A Universal Asynchronous Receiver/Transmitter (UART) translates data between serial and parallel forms and is used in many communications standards, such as RS-232 and RS-485. The ATLYS board includes an EXAR USB-UART bridge chip that facilitates UART communications between the FPGA and the USB port of a PC. The implementation for this project has the protocol fixed at 8 data bits, 1 stop bit, no parity bit and no handshaking. Baud rate, *fs*, is user configurable through the integer value of the UART Baud Rate Divider Register, *D*, using the equation

|  |  |  |
| --- | --- | --- |
|  | , | () |

where *fclk* is the 50 MHz system clock. Maximum baud rate (*D* = 0) is 925.9 kHz, which closely matches the standard 921.6 kHz frequency within 0.5%.

Both received and transmitted data are buffered independently from the Plasma core and main memory using FIFO buffers capable of storing up to 2 kB for each direction. Status bits relating to these FIFOs act as indicators to software when the UART is ready to send (transmit FIFO is not full) or received data are available (receive FIFO is not empty). The bits of the UART Status Register are presented in Table 3. A mechanism exists to reset the UART and empty each of the FIFO buffers through the UART Control Register, presented in

Table 4. To initiate transmission of a byte simply write it to the UART Transmit Data Register, which will add it to the FIFO queue and automatically begin sending. To read available data from the receive FIFO simply read from the UART Receive Data Register which will automatically remove it from the FIFO.

Table . UART Status Register

|  |  |
| --- | --- |
| Bit | Description |
| 0 | Transmit FIFO is Full. |
| 1 | Transmit FIFO is Not Empty. |
| 2 | Receive FIFO is Full. |
| 3 | Receive FIFO is Not Empty, i.e., Receive Data Available. |
| 4 | Transmitter is Active. |
| 31:5 | Unused. Reads return 0s. |

Table . UART Control Register

|  |  |
| --- | --- |
| Bit | Description |
| 0 | Reset Transmit FIFO. |
| 1 | Reset Receive FIFO. |
| 31:2 | Unused. |

An example application to poll data incoming and echo back out the UART is shown in the C code listing below.

int main(void) {

char c;

MemoryWrite(UART\_BAUD\_DIV, 1); // Set baud to 460.8 k

MemoryWrite(UART\_CONTROL,0x3); // Reset FIFOs

while(1) {

if((MemoryRead(UART\_STATUS)&0x8)>0) {

c = MemoryRead(UART\_RX);

MemoryWrite(UART\_TX,c);

}

}

}

#### Interrupts

There is rudimentary support for interrupts within the Plasma core. If any bit of the Interrupt Status Register is a logical 1 the Plasma core will immediately jump to the hard coded address of 0x80. It is the responsibility of software to insert a basic interrupt service routine at this address, for example, in boot loader code. This service routine should push the current state of core registers onto the stack; jump to a user defined interrupt handler, with masked status as a parameter; finally, restore the register state from the stack. Assembly code to implement a basic interrupt service routine is shown below.

# void interrupt\_service\_routine()

.equiv IRQ\_STATUS, 0x40

.equiv IRQ\_STATUS\_CLR, 0x44

.equiv IRQ\_VECTOR, 0x4C

.equiv IRQ\_MASK, 0x50

.global interrupt\_service\_routine

.ent interrupt\_service\_routine

interrupt\_service\_routine:

.set noat

#Registers $26 and $27 are reserved for the OS

#Save all temporary registers

#Slots 0($29) through 12($29) reserved for a0-a3

addi $29, $29, -104 #adjust sp

sw $1, 16($29) #at

sw $2, 20($29) #v0

sw $3, 24($29) #v1

sw $4, 28($29) #a0

sw $5, 32($29) #a1

sw $6, 36($29) #a2

sw $7, 40($29) #a3

sw $8, 44($29) #t0

sw $9, 48($29) #t1

sw $10, 52($29) #t2

sw $11, 56($29) #t3

sw $12, 60($29) #t4

sw $13, 64($29) #t5

sw $14, 68($29) #t6

sw $15, 72($29) #t7

sw $24, 76($29) #t8

sw $25, 80($29) #t9

sw $31, 84($29) #lr

mfc0 $26, $14 #C0\_EPC=14 (Exception PC)

addi $26, $26, -4 #Backup one opcode

sw $26, 88($29) #pc

mfhi $27

sw $27, 92($29) #hi

mflo $27

sw $27, 96($29) #lo

lui $6, 0x2000

lw $4, IRQ\_STATUS($6)

lw $5, IRQ\_VECTOR($6)

lw $6, IRQ\_MASK($6)

and $4, $4, $6 # Handler parameter = Status

# Jump and link to IRQ\_VECTOR\_ADDR

jalr $5

addi $5, $29, 0

#Restore all temporary registers

lw $1, 16($29) #at

lw $2, 20($29) #v0

lw $3, 24($29) #v1

lw $4, 28($29) #a0

lw $5, 32($29) #a1

lw $6, 36($29) #a2

lw $7, 40($29) #a3

lw $8, 44($29) #t0

lw $9, 48($29) #t1

lw $10, 52($29) #t2

lw $11, 56($29) #t3

lw $12, 60($29) #t4

lw $13, 64($29) #t5

lw $14, 68($29) #t6

lw $15, 72($29) #t7

lw $24, 76($29) #t8

lw $25, 80($29) #t9

lw $31, 84($29) #lr

lw $26, 88($29) #pc

lw $27, 92($29) #hi

mthi $27

lw $27, 96($29) #lo

mtlo $27

addi $29, $29, 104 #adjust sp

isr\_return:

ori $27, $0, 0x1 #re-enable interrupts

jr $26

mtc0 $27, $12 #STATUS=1; enable interrupts

.end interrupt\_service\_routine

.set at

The address location of the user defined interrupt handler is configurable through software by assigning the function’s address to the Interrupt Vector Register. User interrupt handler routines are provided with information about the cause of the interrupt as an integer function parameter loaded with the contents of the Interrupt Status Register, and should make sure to clear the flag by writing a 0 to the corresponding bit position of the Interrupt Status Register.

Each of the 32 interrupt sources represented by the bits of the Interrupt Status Register have a corresponding bit in the Interrupt Mask Register. Interrupts for any given source are only enabled if the corresponding bit in the Interrupt Mask Register is a logical 1.

Additionally, for any interrupts to work interrupts must be enabled globally. This is achieved using the special assembler instructions to “move to coprocessor 0, mtc0” and “move from coprocessor 0, mfc0”. Bit 0 of coprocessor 0 register 12 is the global interrupt enable bit. Assembly code to enable interrupts and return the previous value is shown below.

# int OS\_AsmInterruptEnable(int enable)

.global OS\_AsmInterruptEnable

.ent OS\_AsmInterruptEnable

OS\_AsmInterruptEnable:

.set noreorder

mfc0 $2, $12

jr $31

mtc0 $4, $12

.set reorder

.end OS\_AsmInterruptEnable

The default Plasma design has very few interrupt sources configured: an external interrupt from the Up button and the free running counter, as per Table 5. An example application that counts presses of the Up button and displays the count on the LEDs is shown in the C code listing below.

int count = 0;

void InterruptHandler(int status) {

count++;

MemoryWrite(IRQ\_STATUS\_CLR,status); // clear flags

MemoryWrite(LEDS\_OUT, count);

}

int main(void) {

MemoryWrite(IRQ\_STATUS,0); // Clear all interrupts

// interrupt vector to handler

MemoryWrite(IRQ\_VECTOR, (unsigned int)InterruptHandler);

MemoryWrite(IRQ\_MASK,0xFF); // Enable interrupts.

OS\_AsmInterruptEnable(1); // Global enable.

while(1){

// Do nothing in main. Work happening in handler

}

}

Table . Bit positions of interrupts sources

|  |  |
| --- | --- |
| Bit | Interrupt source |
| 0 | Up button is pressed. |
| 1 | Counter reached terminal count. |
| 31:2 | Unconnected |

#### LEDs, Switches, Push Buttons and PMOD General I/O

The ATLYS board has a number of basic input/output elements including 8 green LEDs, 5 push buttons (excluding system reset), 8 slide switches and an 8-bit general purpose port connected directly to FPGA pins, i.e., capable of input, output and high impedance.

The bit positions of the LEDs output register and Switches input register correspond directly to the physical positions of the relevant elements on the board. The push buttons on the board are arranged in a cross pattern with an up, down, left, right and centre. Bit positions corresponding to each button are shown in Table 6.

Table . Push Buttons Input Register

|  |  |
| --- | --- |
| Bit | Description |
| 0 | Up button is pressed. |
| 1 | Left button is pressed. |
| 2 | Down button is pressed. |
| 3 | Right button is pressed. |
| 4 | Centre button is pressed. |
| 31:5 | Unused. |

The PMOD connector has three registers associated with it, with each bit position corresponding to the relevant pin number. PMOD General Purpose Output Register will force that signal to the specified logic state provided the corresponding bit of the PMOD General Purpose Tristate Control Register is 0. PMOD General Purpose Input Register can be used to read the actual logic level of the pin: if the tristate control bit is 0 then the input will be equal to the output; if the tristate control bit is 1 then the input will be determined by an external source.

Low frequency inputs such as the buttons and switches have HDL implemented low‑pass filters to filter out switching noise. These filters add an extra 10.5 ms of latency to the signals.

#### Pseudo Random Number Generator

A pseudo random number generator implemented by a 32-bit linear feedback shift register is a part of the example design. To advance the shift register and receive a new value, software should perform a read from the Pseudo Random Number Generator Register. To load the register with a new value perform a write to the register.

It should be noted that the design is very simple, using only a single tap at the 14th bit position giving a maximal sequence of 232 - 1 unique values. With this knowledge it would be trivial to predict the next value in the sequence; a feature that might prove useful for some applications, e.g., testing communications channels, but disastrous for other applications, e.g., cryptography.

#### Counter

This free running counter can be used to accurately measure time intervals or to generate regularly timed interrupts. The current value of the counter can be read or set by reading or writing the counter value register. The frequency of the counter, *fcount*, is determined by the Counter Prescale Register,*P*, by the equation

|  |  |  |
| --- | --- | --- |
|  | , | () |

where *fclk* is the 50 MHz system clock frequency. The counter will automatically restart from 0 after reaching a user defined terminal count, which can be adjusted by modifying the Counter Terminal Count Register. An interrupt is generated (if enabled, see Chapter 3.2) each time the counter reaches the terminal count, *TC*, giving an interrupt frequency of *fcount / (TC + 1)*. The code snipped below shows how to set up the counter with a frequency of 1 MHz and an interrupt period of 1 ms.

MemoryWrite(COUNTER\_PS, 49); // fCount = 1 MHz.

MemoryWrite(COUNTER\_TC, 999); // interrupt period = 1 ms.

MemoryWrite(IRQ\_MASK\_SET, 2); // enable counter interrupt

OS\_AsmInterruptEnable(1); // global interrupt enable

MemoryWrite(COUNTER, 0); // Restart counter

The counter in the example design is free running and cannot be stopped by software. This could be easily changed for a future design by adding an extra control register with a run/stop control bit.

#### Cache Statistics

Two read only registers, DRAM Cache Hit Count Register and DRAM Cache Read Count Register are provided to give statistical information about the DDR cache. Read count refers to the total number of read accesses initiated by the Plasma core. Hit count refers to the total number of read accesses that have accessed the cached copy of memory. Writes to memory are always at full speed and do not affect the statistics.

#### Flash RAM

On the ATLYS board is a 128 Mbit Numonyx N25Q12 Serial Flash memory device. The interface to the device is a 6 pin SPI bus: a serial clock line, a chip select line and up to 4 data lines depending on the selected SPI protocol.

There are three registers associated with the Flash RAM for directly controlling the 6 signal lines. The Flash RAM Data Register and Flash RAM Tristate Control Register are both 4 bits wide and have the same behaviour as the bidirectional PMOD port. The Flash RAM Control Register is used to drive the serial clock and chip select lines, with bit positions as per Table 7.

Table . Flash RAM Control Register

|  |  |
| --- | --- |
| Bit | Description |
| 0 | Flash RAM serial clock. |
| 1 | Flash RAM active low chip select |
| 31:2 | Unused. |

#### General Purpose FIFO

Interfacing to entities external to the Plasma core can be problematic when the data transfers between them are irregular. Common methods to work around this problem are to use interrupts or regular polling. Interrupt driven transfers have limited bandwidth due to the overhead of context switching in the absence of a processor core optimised for this. Polling has a limitation in that it reduces the amount of time the processor could be performing other useful work. A first-in-first-out (FIFO) buffer allows data to be temporarily stored before the user of the data acts on it at a more convenient time, independently of the rate at which the creator of the data inserts into the buffer.

The example design implements a general purpose FIFO buffer for transfers into and out of the Plasma core. The General Purpose FIFO Receive Data Register is loaded with received data as the buffer is filled. Performing a read from this register will automatically load the next word of data if any is available. The General Purpose FIFO Transmit Data Register performs the opposite function, where writes to the register will add data to the buffer. Note that both registers are operating on the same FIFO; in a typical application only one direction will be useful and the circuitry external to the Plasma core will need to be designed accordingly. The General Purpose FIFO Status and Control Register presents information to software regarding the current state of the buffers, e.g., if data are available or if the buffer is full, and enables some control of the buffer, e.g., flushing or emptying. The bit positions of these status and control bits are shown in Table 8.

Table . General Purpose FIFO Status and Control Register

|  |  |
| --- | --- |
| Bit | Description |
| 0 | FIFO is full. |
| 1 | FIFO is empty. |
| 2 | Active high FIFO reset pulse. |
| 31:3 | Unused. |

The FIFO is 8 bits wide and has a capacity of 2 kB.

### C code tool setup and boot loader code

Software targeting the MIPS™ compatible Plasma microprocessor has been written in C and compiled using a version of the GNU Compiler Collection (gcc) that can target the MIPS-1 architecture. Compilation tools are included with the source code for this project.

#### Compilation Tools Setup

The integrated development environment (IDE) used for this project is Microsoft Visual C++ 2010 Express version 10.0 running on the Microsoft Windows 7 operating system. This IDE is currently available for free from the Microsoft website [8]. The default installation of the IDE allows the developer to create a Makefile Project which allows for more customisation of the compilation steps.

What follows is a description of how to set up a Makefile Project in Microsoft Visual Studio C++ 2010 Express to target the Plasma microprocessor. We will use the example of a “Hello World” program with the following source code:

#define MemoryWrite(A,V) \*(volatile unsigned int\*)(A)=(V)

#define UART\_TX 0x20000000

void print(char \* buf) {

while(\*buf != 0)

MemoryWrite(UART\_TX,\*buf++);

}

int main(void) {

print(“Hello World\n”);

while(1) { }

}

There is only one external dependency for this source code: the assembler boot code listing shown in Appendix A.

To create the project, perform the following steps:

1. Gather the compilation tools and library files into one folder. Contents should include files as per Table 9. Note the location.
2. Within Microsoft Visual Studio C++ 2010 Express, create a new Makefile Project:
   1. File > New > Project;
   2. Select “Makefile Project”, give it a name and solution directory;
   3. Configuration Settings: ignore for now and click finish.
3. Right-click on Project > Properties > Configuration Properties > NMake:
   1. Build Command Line:

gmake target Target=$(TargetName) && move /Y test.bin $(TargetName).bin

* 1. Rebuild Command Line: Identical to Build Command Line.
  2. Clean Command Line:

del \*.o \*.map \*.lst \*.axf \*.bin code.txt

* 1. Include Search Path:

..\..\Tools\lib;c:\program files (x86)\microsoft visual studio 10.0\vc\include

* 1. Other fields left as default.

1. Add source file(s) to project, e.g., HelloWorld.c.
2. Copy the makefile as per the listing below to the project directory:

# “Makefile”

BOOT\_OFFSET = 0x40000000

BIN\_MIPS = ..\..\Tools

PLASMA\_LIB = $(BIN\_MIPS)\lib

export CPATH = $(PLASMA\_LIB)

PATH = %PATH%;$(BIN\_MIPS)

SOURCE = $(Target).c

CFLAGS = -O2 -Wall -c -s -fno-pic -mips1 -mno-abicalls

target:

as.exe $(PLASMA\_LIB)\boot\_os.s -o boot\_os.o

gcc.exe $(CFLAGS) $(SOURCE)

ld.exe -Ttext 0x00000000 -e entry -Map $(Target)\_boot.map \

-s –N boot\_os.o $(Target).o -o test.axf

objdump.bat test.axf $(Target)\_boot.lst

convert\_bin.exe

ram\_image.exe $(PLASMA\_LIB)\ram\_xilinx.vhd \

code.txt ram\_$(Target).vhd

ld.exe -Ttext $(BOOT\_OFFSET) -e entry -Map $(Target).map \

-s -N boot\_os.o $(Target).o -o $(Target).axf

objdump.bat $(Target).axf $(Target).lst

1. Build.

By default, the Visual Studio definition $(TargetName) used in step 3.a is the name of the project. This definition is passed in to the makefile as $(Target), which then assumes the top level source file to be compiled is called $(Target).c. For example, a project named “HelloWorld” would have a source file called “HelloWorld.c”. The unintended consequence is that the project should have the top level source file the same name as the project, or a modification should be made to the makefile to set the “SOURCE” variable correctly.

Regarding the “Include Search Path” (step 3.d), the locations pointed to may be different depending on your installation. The “..\..\Tools\lib” path is pointing to the Plasma compilation tools folder relative to the project file. This should be modified as appropriate for your setup, in addition to the “BIN\_MIPS” path of the makefile.

Within the makefile the build process is repeated twice to generate two output files: a “.axf” file which is the final MIPS executable in “Executable-and-Linkable Format” (ELF), and a “.vhd” file which describes a hard coded Block RAM entity pre-loaded with the instruction and read-only segments of the “.axf” file. Other major differences between the two are that the “.vhd” file includes the “boot\_os.s” boot initialisation code and the program addresses are all linked from memory address 0; the “.axf” file uses the stripped down boot code “boot.s” and addresses are offset by the value of “BOOT\_OFFSET”. The alternative boot offset makes it possible to copy the program to a non-zero offset in memory and run it from there; a feature that is useful for applications transferred and started by a boot loader.

Table . C Compilation Tools and Library Files

|  |  |
| --- | --- |
| File name | Description |
| ./gcc.exe | Main GNU Compiler Collection executable. |
| ./cpp.exe | C language pre-processor, invoked by gcc. |
| ./cc1.exe | C language compiler, invoked by gcc. |
| ./as.exe | Assembly language compiler. |
| ./ld.exe | Linker executable. |
| ./objdump.exe | Disassembly tool. |
| ./objdump.bat | Disassembly tool helper script. |
| ./cygwin1.dll | Executable library for the Cygwin UNIX-like windows environment. |
| ./convert\_bin.exe | Tool for extracting loadable binary content from an Executable and Linkable Format (ELF) file. |
| ./ram\_image.exe | Tool for inserting binary file into VHDL Block RAM source code template. |
| ./lib/boot\_os.s | Boot assembly code including initialisation routines. |
| ./lib/boot.s | Stripped down boot assembly code. |
| ./lib/ram\_xilinx.vhd | VHDL Block RAM source code template. |
| ./lib/plasma.h | Plasma core specific definitions, including peripherals addresses. |

##### Stack Initialisation

The boot assembly code initialises the stack using the code below.

.ifndef StackLoc

.set StackLoc, 0x8000

.endif

la $sp, StackLoc

By default, the stack pointer is initialised to address 0x8000. This address is chosen because in the example design it points to the end of internal Block RAM memory. By convention the stack grows downwards.

To initialise the stack at a different address, modify the Makefile to include a command line option for the as.exe line: --defsym StackLoc=0xNNNN.

There is no logic provided to detect a stack overflow condition.

##### Heap Initialisation

To facilitate dynamic memory allocation it is necessary to have a portion of memory assigned for the heap. This functionality is not provided in the example designs; however, the boot assembly code does include a function to return a pointer to unallocated memory space, listed below.

#define location of heap.

.ifndef HeapLoc

.set HeapLoc, \_end

.endif

# void\* OS\_GetHeap()

.global OS\_GetHeap

.ent OS\_GetHeap

OS\_GetHeap:

la $2, HeapLoc

jr $31

nop

.end OS\_GetHeap

By default the address of the heap is defined as the end of the “.bss” section. To initialise the heap at a different address, modify the Makefile to include a command line option for the as.exe line: --defsym HeapLoc=0xNNNN.

#### Boot Loader

A boot loader program has been developed that is intended to be the first program to run on system reset and can load and branch to other programs. The alternative to loading a program through the boot loader is to include a newly written program in the hard-coded VHDL Block RAM entity and recompile the full HDL source. This process is typically very time consuming, especially for large HDL designs, and becomes frustrating when making only small changes to the program.

The boot loader program is designed to read an Executable and Linkable Format (ELF) file either transmitted to the device via UART or stored on the non-volatile Flash RAM chip. To ready the program to receive over UART the user must press reset while holding down the ‘BTND’ button; otherwise the file is loaded from offset 0x180000 of the Flash RAM. After loading the boot loader program will immediately branch to the entry point of the newly loaded program.

Only ELF files targeting the MIPS-I architecture are suitable for loading. If loading is unsuccessful the boot loader program will begin to flash an error code on the LEDs. Table 10 lists the specific parameters that the ELF must satisfy in order to be loaded, along with the corresponding error code if it is not satisfied.

Table . ELF File Parameters

|  |  |  |
| --- | --- | --- |
| Parameter | Required Setting | Error Code |
| Identification, aka, “Magic Number” | { 0x7F, ‘E’, ‘L’, ‘F’ } | 1 |
| Bit Format | 1, 32-bit | 2 |
| Endianness | 2, big endian | 3 |
| Target Architecture | 8, MIPS-I | 4 |
| Program Header Table Entry Size | 32 bytes | 5 |
| Number of Program Header Entires | 32 | 6 |
| Section Header Table Entry Size | 40 bytes | 7 |
| Minimum Segment Virtual Address | 0x2000 | 8 |

To upload an ELF file to the Flash RAM of the ATLYS board for subsequent automatic boot loading use the Digilent Adept program. Under the ‘Flash’ tab use the “Write data file to Flash” feature, specifying the start address as 180000 hex.

### C# boot loader app

A Microsoft Windows application has been written to simplify the transmission of ELF files from a PC to a Plasma design running the boot loader code. The Boot Loader App is a command line application written in C# and using the Microsoft .NET 4.0 framework.

Below is the text of the application help output.

Plasma/CS/PlasmaLoaderApp

Windows application for transmitting a file to the Plasma Microprocessor. After transmission completes a terminal is opened to communicate with the device.

Usage: PlasmaBootLoader [options...] [file]

Options:  
 -b <baudRate>, default = 460800  
 -c <comPort>, default = COM10  
 -l <logFile>  
 -s = silent mode  
 -? = print this information  
 -Help = same as -?

Omitting the [file] parameter will open a console without sending anything.

Example: PlasmaBootLoader -b 115200 -c COM9 -l log.txt test.axf

### Testing

ISim, the HDL simulator included with the free “Webpack” Xilinx ISE tools can be used to simulate the Plasma core. For this project version 13.3 is used. The HDL design can be compiled with a number of options (VHDL generics) to aid behavioural simulation, typically to save time by either reducing the complexity of the design or bypass time consuming tasks such as UART transfers and boot code.

The first option is called ‘simulateRam’ which is ‘0’ by default. When set to ‘1’ the design will simplify the DRAM memory controller in the design (if present) and phase-locked-loop entities for clock generation are replaced with unsythesisable testbench code.

The second option is called ‘simulateProgram’ which is also ‘0’ by default. In default operation the boot loader code (see Section 4.2) is run, and the testbench is responsible for loading additional programs in a way that is expected by the boot loader, for example, simulating the transmission of a file via the UART. By setting the ‘simulateProgram’ generic to ‘1’ the design will initialise the program RAM with the anticipated program directly.

Waiting for UART transmissions within a simulation is not useful when the content of the message is the purpose of the test rather than the method or robustness of the transmission. The example HDL design includes ports to read/write data directly from/to the UART FIFO buffers, increasing simulated transmission baud rates to over 250 MHz. The receiver interface is a simple 8-bit wide input with a single control line that is toggled to signal a write transaction into the RX FIFO. The transmitter interface is the inverse; an 8-bit wide output with a control line that toggles to indicate the data is valid.

Hardware inputs such as switches and buttons have HDL implemented low-pass filters that add latency of 10 ms. Simulated designs, where the transitions of the inputs are ideal, use modified low-pass filters that reduce the latency to only 180 ns.

### Example designs

Three Xilinx ISE projects and two Visual Studio C++ 2010 Express projects have been created as examples that future designs can extend.

#### PlasmaBlockRAM ISE Project

This is the simplest example design to demonstrate the Plasma core. The section of memory referred to as External General Purpose Memory (See Section 2.2) is mapped to 32 kB of on-chip Block RAM. All peripherals listed in Section 3 are included except for the registers related to DRAM cache statistics which are meaningless in this context. The top-level design includes ports for the General Purpose FIFOs and the External Bus, however, these are not mapped to any pins of the hardware so are unusable.

The PlasmaBlockRAM project includes two VHDL simulation testbenches; one which runs the ‘Example’ program directly and another to run the ‘BootLoader’ which then loads and runs the ‘Example’ program.

Xilinx Spartan 6 XC6SLX45 device utilisation for the design is modest, leaving plenty of resources available to extend the design for specific applications. Device utilisation is summarised in Table 11.

Table . PlasmaBlockRAM Device Utilisation

|  |  |  |
| --- | --- | --- |
| Resource Type | Number Used / Available | Percentage Used |
| Occupied Slices | 1164 / 6822 | 17% |
| 18 kB RAM Blocks | 39 / 116 | 33% |
| DSP48A1s | 0 / 58 | 0% |
| User I/O Pins | 39 / 218 | 17% |

#### PlasmaAtlysDDR ISE Project

To demonstrate the MCB interface to the ATLYS 1 GBit DDR2 DRAM chip, the PlasmaBlockRAM design has been extended to replace Block RAM referenced as the External General Purpose Memory. This design includes the DRAM controller, cache unit and cache statistics as well as relevant pin mappings and constraints.

Device utilisation is summarised in Table 12. At a cost of 333 occupied slices compared to the PlasmaBlockRAM design the size of the External General Purpose Memory section is increased from 32 kB to 128 MB.

Table . PlasmaAtlysDDR Device Utilisation

|  |  |  |
| --- | --- | --- |
| Resource Type | Number Used / Available | Percentage Used |
| Occupied Slices | 1497 / 6822 | 21% |
| 18 kB RAM Blocks | 23 / 116 | 19% |
| DSP48A1s | 0 / 58 | 0% |
| User I/O Pins | 86 / 218 | 39% |

#### PlasmaDataAcquisition ISE Project

A more plausible real world application for the Plasma core is to act as a communication controller between an application specific interface and a PC. Figure 4 illustrates the basic functionality of the design:

* A Data Source generates data – it could be anything; in the example project it is simply the sampled values of the buttons, switches, PMOD port, and a 2 MHz free running 32-bit counter.
* Using the External Bus, two custom registers are created: Sample Divider to set the rate at which the Data Source is producing data; Selector to select which source(s) are inserted into the FIFO.
* The General Purpose FIFO transfers data into the Plasma core.
* The Plasma core runs the PlasmaDataAcquisition program (see Section 7.5)
* A connected PC runs the PlasmaLoaderApp to listen to the UART and write the incoming data to a log file (see Section 0).

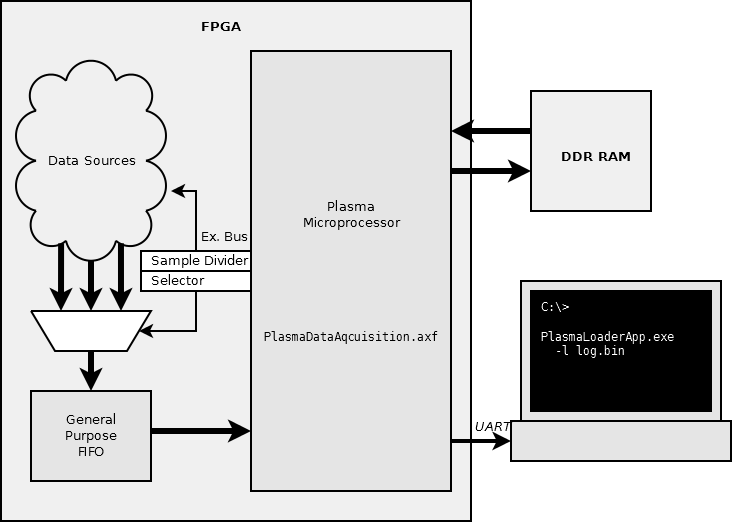


Figure . Plasma Data Acquisition Design

The Selector Register selects which source(s) are to be inserted into the FIFO as per Table 13, where a ‘1’ in a bit position will insert that data. When multiple data sources are selected, the data stream logged by the PC will have the sources interleaved. The sample rate, *fS* is determined by

|  |  |  |
| --- | --- | --- |
|  | , | () |

where *fclk* is the 50 MHz system clock and *D* is the value of the Sample Divider Register. The data rate into the FIFO is equal to *fS* multiplied by the number of ‘1’s in the Selector Register.

Table . Plasma Data Acquisition Selector Register

|  |  |
| --- | --- |
| Bit | Description |
| 0 | PMOD Input |
| 1 | Switches Input |
| 2 | Buttons Input |
| 3 | Nonsense (combination of counter and fixed value) |
| 4  5 | 2 MHz Counter(7:0)  2 MHz Counter(15:8) |
| 6 | 2 MHz Counter(23:16) |
| 7  31:8 | 2 MHz Counter(31:24)  Unused |

#### Example C Project

This project builds a simple C program for demonstrating some basic features of the Plasma core. Firstly it transmits the string “Hello World” out of the UART as well as the value of the switches incremented by one. It then enters an infinite loop of reading characters coming in on the UART and echoing them back out again.

Meanwhile, the push buttons interrupt is enabled (listening to presses of the Up button), and the interrupt handler is incrementing a counter and displaying it on the LEDs.

#### PlasmaDataAcquisition C Project

For use with the PlasmaDataAcquisition ISE project, the PlasmaDataAcquisition C program handles the transfer of data from the FIFO to the PC. The most accurate way to describe the program is to show the code.

#include <plasma.h>

#define BUFFER\_LEN (RAM\_SDRAM\_SIZE/2)

char \* buffer;

int main(void)

{

int value, writeIndex, readIndex;

int maxLen = BUFFER\_LEN;

buffer = (char\*)OS\_GetHeap();

MemoryWrite(0x30000004,0x40); // Set the selector

MemoryWrite(0x30000000, 100); // Set the sample divider

while(1)

{

MemoryWrite(FIFO\_CON, FIFO\_CLEAR\_MASK);

writeIndex = 0;

readIndex = 0;

while(readIndex < maxLen)

{

while(writeIndex < BUFFER\_LEN && FIFO\_DOUT\_RDY)

{

value = MemoryRead(FIFO\_DOUT);

buffer[writeIndex] = value;

writeIndex ++;

}

if(readIndex < writeIndex && UART\_TX\_RDY)

{

value = buffer[readIndex];

MemoryWrite(UART\_TX, value);

MemoryWrite(LEDS\_OUT,readIndex>>10);

readIndex++;

}

// Break early if CENTRE button pressed.

if(BUTTON\_CENTRE\_DOWN)

break;

}

MemoryWrite(LEDS\_OUT,0xCC);

// Wait for CENTRE button press

while(!BUTTON\_CENTRE\_DOWN);

while(BUTTON\_CENTRE\_DOWN);

}

return 0;

}

In summary: the program declares a large buffer of uninitialised memory; selects the channels to record and the sample rate (see Section 7.3); then enters an infinite loop with another loop nested inside. The outer loop initially clears the FIFO; resets read and write indices to zero; enters the inner loop; then waits for the user to press the centre button.

The inner loop is polling the FIFO data ready flag, and if data are available it transfers the bytes to the buffer. If the UART transmission service is ready and the buffer contains data not yet transmitted, the data is read back from the buffer and inserted into the UART transmit FIFO. The inner loop will end when an entire buffer’s length of data is sent, or the user presses the centre button.

If the data rate into the FIFO is too high (e.g., due to many sources selected or sample divider too low) such that the processor cannot service the FIFO, the FIFO will eventually become full and drop data. The maximum data rate into the FIFO is 2 MB/s.

## REFERENCES

| **No** | **Reference** |
| --- | --- |
|  | Xilinx Inc. Spartan 6 DC and Switching Characteristics. |
|  | Plasma Project Overview, http://opencores.org/project,plasma accessed 4/9/13. |
|  | MIPS Technologies, Inc. http://www.mips.com |
|  | Digilent Inc., ATLYS™ Spartan-6 FPGA Development Board. http://www.digilentinc.com/Products/Detail.cfm?Prod=ATLYS |
|  | Xilinx Inc. Spartan-6 FPGA Memory Interface Solutions – UG416. June 2011. |
|  | Digilent Inc., ATLYS™ Board Reference Manual. February 2011. |
|  | Smith, Alan Jay. "Cache memories." ACM Computing Surveys (CSUR) 14.3 (1982): 473-530. |
|  | Microsoft Corporation, Visual Studio 2010 Express Products. http://www.microsoft.com/visualstudio/eng/products/visual-studio-2010-expressr |
|  |  |
|  |  |
|  |  |

APPENDIX A

## Appendix A – Assembly Boot Code

The following MIPS-I assembly code initialises the stack, provides default implementations of the interrupt service routine and provides some exception handling routines.

.text

#define location of stack.

.ifndef StackLoc

.set StackLoc, 0x8000

.endif

#define location of heap.

.ifndef HeapLoc

.set HeapLoc, \_end

.endif

# Make sure these registers match the hardware!

.equiv IRQ\_STATUS, 0x40

.equiv IRQ\_STATUS\_CLR, 0x44

.equiv IRQ\_VECTOR, 0x4C

.equiv IRQ\_MASK, 0x50

#.align 2

.global entry

.ent entry

entry:

.set noreorder

#These four instructions should be the first instructions.

# NB: la = two instructions: lui + addiu

la $gp, \_gp #initialize global pointer

la $5, \_\_bss\_start #$5 = .sbss\_start

la $4, \_end #$2 = .bss\_end

la $sp, StackLoc

$BSS\_CLEAR:

sw $0, 0($5)

slt $3, $5, $4

bnez $3, $BSS\_CLEAR

addiu $5, $5, 4

# Set ISR to OS\_DefaultISR

la $2, OS\_DefaultISR

lui $3, 0x2000

sw $2, IRQ\_VECTOR($3)

jal main

nop

$L1:

j $L1

.end entry

###################################################

# int OS\_GetHeap()

# Gets a pointer to 'Heap' space, typically an area

# of memory beyond the bss section.

###################################################

.global OS\_GetHeap

.ent OS\_GetHeap

OS\_GetHeap:

la $2, HeapLoc

jr $31

nop

.end OS\_GetHeap

###################################################

# A simple interrupt handler that does nothing

# except clear the relevant flag.

###################################################

.global OS\_DefaultISR

.ent OS\_DefaultISR

OS\_DefaultISR:

.set noreorder

lui $2,0x2000

jr $31

sw $4,IRQ\_STATUS\_CLR($2)

.set reorder

.end OS\_DefaultISR

###################################################

# int OS\_AsmInterruptEnable(int enable)

# Global interrupt enable (enable == 1)

# Returns previous enable state.

###################################################

.global OS\_AsmInterruptEnable

.ent OS\_AsmInterruptEnable

OS\_AsmInterruptEnable:

mfc0 $2, $12

jr $31

mtc0 $4, $12 #STATUS=1; enable interrupts

.end OS\_AsmInterruptEnable

###################################################

# Interrupt service routine:

# Saves current context,

# Jumps to address of user defined IRQ\_VECTOR\_ADDR,

# [function of type void handler(int status) ]

# Restores context.

###################################################

.balign 0x80

.global interrupt\_service\_routine

.ent interrupt\_service\_routine

interrupt\_service\_routine:

.set noreorder

.set noat

#Registers $26 and $27 are reserved for the OS

#Save all temporary registers

#Slots 0($29) through 12($29) reserved for saving a0-a3

addi $29, $29, -104 #adjust sp

sw $1, 16($29) #at

sw $2, 20($29) #v0

sw $3, 24($29) #v1

sw $4, 28($29) #a0

sw $5, 32($29) #a1

sw $6, 36($29) #a2

sw $7, 40($29) #a3

sw $8, 44($29) #t0

sw $9, 48($29) #t1

sw $10, 52($29) #t2

sw $11, 56($29) #t3

sw $12, 60($29) #t4

sw $13, 64($29) #t5

sw $14, 68($29) #t6

sw $15, 72($29) #t7

sw $24, 76($29) #t8

sw $25, 80($29) #t9

sw $31, 84($29) #lr

mfc0 $26, $14 #C0\_EPC=14 (Exception PC)

addi $26, $26, -4 #Backup one opcode

sw $26, 88($29) #pc

mfhi $27

sw $27, 92($29) #hi

mflo $27

sw $27, 96($29) #lo

lui $6, 0x2000

lw $4, IRQ\_STATUS($6)

lw $5, IRQ\_VECTOR($6)

lw $6, IRQ\_MASK($6)

and $4, $4, $6

# Jump and link to IRQ\_VECTOR\_ADDR

jalr $5

addi $5, $29, 0

#Restore all temporary registers

lw $1, 16($29) #at

lw $2, 20($29) #v0

lw $3, 24($29) #v1

lw $4, 28($29) #a0

lw $5, 32($29) #a1

lw $6, 36($29) #a2

lw $7, 40($29) #a3

lw $8, 44($29) #t0

lw $9, 48($29) #t1

lw $10, 52($29) #t2

lw $11, 56($29) #t3

lw $12, 60($29) #t4

lw $13, 64($29) #t5

lw $14, 68($29) #t6

lw $15, 72($29) #t7

lw $24, 76($29) #t8

lw $25, 80($29) #t9

lw $31, 84($29) #lr

lw $26, 88($29) #pc

lw $27, 92($29) #hi

mthi $27

lw $27, 96($29) #lo

mtlo $27

addi $29, $29, 104 #adjust sp

isr\_return:

ori $27, $0, 0x1 #re-enable interrupts

jr $26

mtc0 $27, $12 #STATUS=1; enable interrupts

.end interrupt\_service\_routine

.set at

###################################################

.global setjmp

.ent setjmp

setjmp:

.set noreorder

sw $16, 0($4) #s0

sw $17, 4($4) #s1

sw $18, 8($4) #s2

sw $19, 12($4) #s3

sw $20, 16($4) #s4

sw $21, 20($4) #s5

sw $22, 24($4) #s6

sw $23, 28($4) #s7

sw $30, 32($4) #s8

sw $28, 36($4) #gp

sw $29, 40($4) #sp

sw $31, 44($4) #lr

jr $31

ori $2, $0, 0

.set reorder

.end setjmp

###################################################

.global longjmp

.ent longjmp

longjmp:

.set noreorder

lw $16, 0($4) #s0

lw $17, 4($4) #s1

lw $18, 8($4) #s2

lw $19, 12($4) #s3

lw $20, 16($4) #s4

lw $21, 20($4) #s5

lw $22, 24($4) #s6

lw $23, 28($4) #s7

lw $30, 32($4) #s8

lw $28, 36($4) #gp

lw $29, 40($4) #sp

lw $31, 44($4) #lr

jr $31

ori $2, $5, 0

.set reorder

.end longjmp

###################################################

.global OS\_AsmMult

.ent OS\_AsmMult

OS\_AsmMult:

.set noreorder

multu $4, $5

mflo $2

mfhi $4

jr $31

sw $4, 0($6)

.set reorder

.end OS\_AsmMult

###################################################

.global OS\_Syscall

.ent OS\_Syscall

OS\_Syscall:

.set noreorder

syscall 0

jr $31

nop

.set reorder

.end OS\_Syscall

APPENDIX B

## Appendix B – Plasma Header File

The following is the contents of the header file “plasma.h”.

#ifndef \_\_PLASMA\_H\_\_

#define \_\_PLASMA\_H\_\_

#define MemoryRead(A) (\*(volatile unsigned int\*)(A))

#define MemoryWrite(A,V) \*(volatile unsigned int\*)(A)=(V)

extern void\* OS\_GetHeap();

extern int OS\_AsmInterruptEnable(int enable);

#define RAM\_INTERNAL\_BASE 0x00000000

#define RAM\_INTERNAL\_SIZE 0x00008000 // 32 kB

#define RAM\_EXTERNAL\_BASE 0x40000000

#define RAM\_BRAM\_SIZE 0x00008000 // 32 kB

#define RAM\_SDRAM\_SIZE 0x08000000 // 128 MB

#define PERIPH\_BASE 0x20000000

#define UART\_OFFSET (PERIPH\_BASE+ 0x00)

#define IRQ\_STATUS (PERIPH\_BASE+ 0x40)

#define IRQ\_STATUS\_CLR (PERIPH\_BASE+ 0x44)

#define IRQ\_VECTOR (PERIPH\_BASE+ 0x4C)

#define IRQ\_MASK (PERIPH\_BASE+ 0x50)

#define IRQ\_MASK\_SET (PERIPH\_BASE+ 0x54)

#define IRQ\_MASK\_CLR (PERIPH\_BASE+ 0x58)

#define LEDS\_OFFSET (PERIPH\_BASE+ 0x60)

#define PMOD\_OFFSET (PERIPH\_BASE+ 0x80)

#define SWITCHES (PERIPH\_BASE+ 0x70)

// FYI: BUTTONS(4 downto 0) = BTNC & BTNR & BTND & BTNL & BTNU

#define BUTTONS (PERIPH\_BASE+ 0x74)

#define BUTTON\_UP\_MASK 0x01

#define BUTTON\_LEFT\_MASK 0x02

#define BUTTON\_DOWN\_MASK 0x04

#define BUTTON\_RIGHT\_MASK 0x08

#define BUTTON\_CENTRE\_MASK 0x10

#define BUTTON\_CENTRE\_DOWN

((MemoryRead(BUTTONS)&BUTTON\_CENTRE\_MASK) != 0)

#define RAND\_GEN (PERIPH\_BASE+ 0x78)

#define LEDS\_OUT (LEDS\_OFFSET+0x0)

#define LEDS\_SET (LEDS\_OFFSET+0x4)

#define LEDS\_CLR (LEDS\_OFFSET+0x8)

#define LEDS\_TGL (LEDS\_OFFSET+0xC)

#define PMOD\_OUT (PMOD\_OFFSET+0x0)

#define PMOD\_SET (PMOD\_OFFSET+0x4)

#define PMOD\_CLR (PMOD\_OFFSET+0x8)

#define PMOD\_TGL (PMOD\_OFFSET+0xC)

#define PMOD\_IN (PMOD\_OFFSET+0x10)

#define PMOD\_TRIS (PMOD\_OFFSET+0x14)

#define UART\_TX (UART\_OFFSET+0x0)

#define UART\_RX (UART\_OFFSET+0x4)

#define UART\_STATUS (UART\_OFFSET+0x8)

#define UART\_CONTROL (UART\_OFFSET+0xC)

#define UART\_BAUD\_DIV (UART\_OFFSET+0x10)

#define UART\_TX\_RDY\_MASK 0x01

#define UART\_TX\_RDY

((MemoryRead(UART\_STATUS)&UART\_TX\_RDY\_MASK) == 0)

#define COUNTER1 (PERIPH\_BASE+ 0xA0)

#define COUNTER1\_PS (PERIPH\_BASE+ 0xA4)

#define COUNTER1\_TC (PERIPH\_BASE+ 0xA8)

#define CACHE\_HITCOUNT (PERIPH\_BASE+ 0xB0)

#define CACHE\_READCOUNT (PERIPH\_BASE+ 0xB4)

#define FLASH\_CON (PERIPH\_BASE+ 0xC0)

#define FLASH\_DATA (PERIPH\_BASE+ 0xC4)

#define FLASH\_TRIS (PERIPH\_BASE+ 0xC8)

#define FIFO\_DIN (PERIPH\_BASE+ 0xD0)

#define FIFO\_DOUT (PERIPH\_BASE+ 0xD4)

#define FIFO\_CON (PERIPH\_BASE+ 0xD8)

#define FIFO\_CLEAR\_MASK 0x04

#define FIFO\_DOUT\_RDY ((MemoryRead(FIFO\_CON) & 0x02) == 0x02)

#endif //\_\_PLASMA\_H\_