Gisselquist
Technology, LLC

OPENARTY SPECIFICATION

Dan Gisselquist, Ph.D.
dgisselq (at) opencores.org

October 28, 2016

Revision History

Rev.	Date	Author	Description
0.0	6/20/2016	Gisselquist	First Draft
0.0	10/21/2016	Gisselquist	More Comments Added

Contents

Page

1 Introduction . . . 1

2 Architecture . . . 2
2.0.1 Bus Structure . . . 2
2.0.2 DDR3 SDRAM . . . 3
2.0.3 Flash . . . 3
2.0.4 Block RAM . . . 3
2.0.5 Ethernet . . . 3
2.0.6 SD Card . . . 4
2.0.7 GPS Tracking . . . 4
2.0.8 Configuration port . . . 4
2.0.9 OLED . . . 5
2.0.10 Real Time Clock . . . 5
2.0.11 LEDs . . . 5
2.0.12 Buttons . . . 5
2.0.13 Switches . . . 5
2.0.14 Startup counter . . . 5
2.0.15 GPS UART . . . 6
2.0.16 Auxilliary UART . . . 6
2.0.17 GPIO . . . 6
2.0.18 Linker Script . . . 7

3 Software . . . 8
3.1 Directory Structure . . . 8
3.2 Zip CPU Tool Chain . . . 8
3.3 Bench Test Software . . . 8
3.4 Host Software . . . 8
3.5 Zip CPU Programs . . . 8
3.6 ZipOS . . . 8
3.6.1 System Calls . . . 8
3.6.2 Scheduler . . . 9

4 Operation . . . 10

5 Registers . . . 11
5.1 Peripheral I/O Control . . . 13
5.1.1 Interrupt Controller . . . 13
5.1.2 Last Bus Error Address . . . 15
5.1.3 General Purpose I/O . . . 15
5.1.4 UART Data Register . . . 15
5.2 Debugging Scopes . . . 15
5.3 Internal Configuration Access Port . . . 15
5.4 Real–Time Clock . . . 15
5.5 On-Chip Block RAM . . . 15
5.6 Flash Memory . . . 15

6 Wishbone Datasheet . . . 17

7 Clocks 18
8 I/O Ports 20

Figures

Figure Page

Tables

Table		Page
5.1.	Address Regions	11
5.2.	ZipSystem Addresses	12
5.3.	I/O Peripheral Registers	13
5.4.	Primary System Interrupts	14
5.5.	Auxilliary System Interrupts	14
5.6.	Bus Interrupts	15
5.7.	Flash control registers	16
7.1.	OpenArty clocks	19
8.1.	List of IO ports	21

Preface

Dan Gisselquist, Ph.D.

1.

Introduction

The goals of this project include:

1. Use entirely open interfaces

   This means not using the Memory Interface Generator (MIG), the Xilinx CoreGen IP, etc. Further, I wish to use all of Arty's on–board hardware: Flash, DDR3-SDRAM, Ethernet, and everything else at their full and fastest speed(s). For example, the flash will need to be clocked at 82 MHz, not the 50 MHz I've clocked it at before. The memory should also be able to support pipelined 32–bit interactions over the Wishbone bus at a 162 MHz clock. Finally, the Ethernet controller should be supported by a DMA capable interface that can drive the ethernet at its full 100Mbps rate.

2. Run using a 162.5 MHz clock, if for no other reason than to gain the experience of building logic that can run that fast.[1]

3. Modify the ZipCPU to support an MMU and a data cache, and perhaps even a floating point unit.

4. The default configuration will also include three Pmods: a USBUART, an SDCard, and the GPS Pmod.

I intend to demonstrate this project with a couple programs:

1. NTP Server

2. A ZipOS that can actually load and run programs from the SD Card

   This will require a functioning memory management unit (MMU), which will be a new addition to the ZipCPU created to support this project. For those not familiar with MMU's, an MMU translates memory addresses from a virtual address space to a physical address space. This allows every program running on the ZipCPU to believe that they own the entire memory address space, while allowing the operating system to allocate actual physical memory addresses as necessary to support whatever program needs more (or less) memory.

[1] The original goal was to run at 200 MHz. However, the memory controller cannot run faster than 83 MHz. If we run it at 81.25 MHz and double that clock to get our logic clock, that now places us at 162.5 MHz. 200 MHz is . . . too fast for DDR3 transfers using the Artix–7 chip on the Arty.

2.

Architecture

My philosophy in peripherals is to keep them simple. If there is a default mode on the peripheral, setting that mode should not require turning any bits on. If a peripheral encounters an error condition, a bit may be turned on to indicate this fact, otherwise status bits will be left in the off position.

2.0.1 Bus Structure

The OpenArty project contains four bus masters, three of them within the CPU. These masters are the instruction fetch unit, the data read/write unit, and the direct memory access peripheral within the ZipCPU, as well as an external debug port which can be commanded from over the main UART port connecting the Arty to its host.

There is also a second minor peripheral bus located within the ZipCPU ZipSystem. This bus provides access to a number of peripherals within the ZipSystem, such as timers, counters, and the direct memory access controller. This bus will also be used to configure the memory management unit once integrated. This bus is only visible to the CPU, and located starting at address `0xc0000000`.

The ZipCPU debug port is also available on the bus. This port, however, is only visible to the external debug port. It can be found at address `0x08000000` for the control register, and `0x08000001` for the data register.

Once the MMU has been integrated, it will be placed between the instruction fetch unit, data read/write unit, and the rest of the peripheral bus.

The actual bus chosen for this design is the Wishbone Bus, based upon the pipeline mode defined in the B4 specification. All optional wires required by this bus structure have been removed, such as the tag lines, the cycle type identifier, the burst type, and so forth. This was done to simplify the logic within the core.

However, because of the complicated bus structure–particularly because of the number of masters and slaves on the bus and the speed for which the bus is defined, there are a number of delays and arbiters placed on the bus. As a result, the stall wire which is supposed to be depend upon combinational logic only, has been registered at a number of locations. What this means is that there are a variety of delays as commands propagate through the bus structure. Most of these are variable, in that they can be turned on or off at build time, or even that the stall line may (or may not) be registered as configured.

All interactions between bus masters and any peripherals passes through the interconnect, located in `busmaster.v`. This interconnect divides the slaves into separate groups. The first group of slaves are those for which the bus is supposed to provide fast access to. These are the DDR3 SDRAM, the flash, the block RAM, and the network. The next group of slaves will have their acknowledgements delayed by an additional clock. The final group of slaves are those single register slaves whose results

may be known ahead of any read, and who only require one clock to access. These are grouped together and controlled from within `fastio.v`.

Further information about the Wishbone bus structure found within this core can be found either on the Wishbone datasheet (Ch. 6), or in the memory map table in the Registers chapter (Ch. 5).

2.0.2 DDR3 SDRAM

It is the intention of this project to use a completely open source DDR3 SDRAM controller. While the controller has been written, it has yet to be successfully connected to the physical pins of the port. Until that time, the design is running using a Wishbone to AXI bus bridge. Memory may still be read or written, after an initial pipeline delay of roughly 27 clocks per access, at one access per clock.

The open source SDRAM controller should be able to achieve a delay closer to 9 clocks per access–once I figure out how to connect it to the PHY.

2.0.3 Flash

2.0.4 Block RAM

The block RAM on this board has been arranged into one 32kW section. Programs that use block RAM will run fastest using the block RAM, both for instructions as well as for memory.

2.0.5 Ethernet

The ether net controller has been split into three parts. The first part is an area of packet memory. This part is simple: it acts like memory. The receive memory is read only, whereas the transmit memory is both read and write. Packets received by the controller will be found in the receive memory, packets transmitted must be in the transmit area of memory. The octets may be found in memory with the first octet in the most significant byte. This is the easy part.

The format of the packets within this memory is a touch more interesting. With no options turned on, the first 6 bytes are the destination MAC address, the next 6 bytes will be the source MAC address, and the *next 4 bytes* will be the EtherType repeated twice. This was done to align the packet, and particularly the IP header, onto word boundaries. If the hardware CRC has been turned off, the packet must contain its own CRC as well as ensuring that it has a minimum packet length (64 octets) when including that CRC.

With all options turned on, however, things are a touch simpler. The first two words of the packet contain the destination MAC (for a transmit packet) or the source MAC (for a received packet), followed by the two–octet EtherType. At this point the packet is word–aligned prior to the IP header. Since broadcast packets are sent to a special destination MAC other than our own, a flag in the command register will indicate this fact.

The second part of the controller is the MDIO interface. This follows from the specification, and can be used to toggle the LED's on the ethernet, to force the ethernet into a particular mode, either 10M or 100M, to control auto–negotiation of the speed, and more. Reads or writes to MDIO memory addresses will command reads or writes via the MDIO port from the FPGA to the ethernet PHY. As the PHY can only handle 16–bit words, only 16 bits will ever be transferred as a result of any read/write command, the top 16 bits are automatically set to zero. Further details of this capability may be found within the specification for the chip.

The MDIO interface may be ignored. If ignored, the defaults within the interface will naturally set up the network connection in full duplex mode (if your hardware supports it), at the highest speed the network will support. However, if you ignore this interface you may not know what problems you are suffering from this interface, if any. The `netsetup` program has been provided, among the host software, to help diagnose how the various MDIO registers have been set, and what the status is that is being reported from the PHY.

The third part of the controller is the packet command interface. This consists of two command registers, one for reading and one for writing. Before doing anything with the network, it must first be taken out of reset. According to the specification for the network chip, this must happen a minimum of one second after power up. This may be done by simply writing to the transmit command register with the reset bit turned off.

To send a packet, simply write the number of octets in the packet to the transmit control register and set the GO bit (`0x04000`). Other bits in this control register can be used to turn off the hardware MAC generation (and removal upon receive), the hardware CRC checking, and/or the hardware IP header checksum validation (but not generation). The GO bit will remain high while the packet is being sent, and only transition to low once the packet is away. While the packet is being sent, a zero may be written to the command register to cancel the packet–although this is not recommended.

Packets are automatically received without intervention. Once a packet has been received, the available bit will be set in the receive command register and a receive packet interrupt will be generated. The ethernet port will then halt/stall until a user has reset the receive interface so that it may receive the next packet. Without clearing this interface, the receive port will not accept further packets. Other status bits in this interface are used to indicate whether packets have been missed (because the interface was busy), or thrown out due to some error such as a CRC error or a more general error.[1]

2.0.6 SD Card

2.0.7 GPS Tracking

2.0.8 Configuration port

The registers associated with the ICAPE2 port have been made accessible to the core via the `wbicapetwo` core. More information about the meaning of these registers can be found in Xilinx's "7–Series FPGAs Configuration User's Guide".

Testing with the OpenArty board has tended to focus on the warmboot capability. Using this capability, a user is able to command the FPGA to reload its configuration. In support of this, two configuration areas have been defined within memory. The first is the default configuration, found at the beginning of the flash. This configuration is sometimes called the "golden configuration" within Xilinx's documentation because it is the configuration that the Xilinx device will always start up from after a power on reset. On the OpenArty, a second configuration may immediately follow the first in flash. Commanding the FPGA to reload it's configuration is as simple as setting the WBSTAR (warm boot start address) register to the location of the new configuration within the flash, and then writing a 15 (a.k.a. IPROG) to the FPGA command register (offset 4 from the beginning of the ICAPE2 addresses). Examples of doing this are found in the `sw/host/zprog.sh`

[1] It should be possible to extend this interface so that further packets may be read as long as the memory isn't yet full. This is left as an exercise to others.

and `sw/host/program.sh` scripts. The former programs the default configuration and then switches to it,

This configuration capability makes it possible for a user to 1) reprogram the flash with an experimental configuration in the second configuration location, and 2) test the configuration without actually touching the board. If the configuration doesn't work well enough to be communicated with, the board may simply be powered down and it will come back up with the initial or golden configuration. If the golden configuration ever gets corrupted, or loaded with a configuration that will not work, then the user will need to reload the FPGA from the JTAG port.

2.0.9 OLED

2.0.10 Real Time Clock

The Arty board contains a real time clock core together with a companion real time date/calendar core. The clock core itself contains not only current time, but also a stopwatch, seconds timer, and alarm. The real time date core can be used to maintain the current date. The real–time clock core uses the GPS PPS output, as schooled by the GPS tracking circuit, in order to synchronize their subsecond timing to the GPS itself. Further, the real–time clock core then creates a synchronization wire for the real–time date core.

Neither of these cores exports its subsecond precision to the rest of the design. This must be done using either the internal GPS tracking wires, or by reading the time information from the tracking test bench.

2.0.11 LEDs

The Arty board contains two sets of LEDs: a plain set of LEDs, and a colored set of LEDs.

The plain set of LEDs is controlled simply from the LED register. This register can be used to turn these LEDs on and off, either individually or as a whole. It has been designed for atomic access, so only one write to this register is necessary to set any particular LED.

The color LEDs are slightly different. Each color LED is supported by its own register, which controls three pulse width modulation controllers. Three groups of eight bits within the color LED register control the PWM thresholds, first for red, then green, and then in the lowest bits for blue. These are used to turn on and off the various color components of the LEDs. Using this method, there are 2^{24} different colors each of these LEDs may be set to.

2.0.12 Buttons

2.0.13 Switches

2.0.14 Startup counter

A startup counter has been placed into the basic peripheral I/O area. This counter simply counts the clocks since startup. Upon rollover, the high order bit remains set. This can be used to sequence the start up of components within the design if so desired.

2.0.15 GPS UART

The GPS UART, debug control UART, as well as the auxilliary UART, are all based upon the same underlying UART IP core, sometimes known as the WBUART32 core. The setup register is defined within the documentation for that core, and provides for a large baud rate selection, 5-8 data bits, 1-2 stop bits, and several parity choices. Within OpenArty, the GPS core is initialized to 9.6 kBaud, 8 data bits, no parity, and one stop bit.

When a value is ready to be read from the GPS uart, the GPS interrupt line will go high. Once read, and only when read, will this interrupt line reset. If the read is successful, only bits within the bottom eight will be set. If a read is attempted when there is no data, when the UART is in a reset condition, or when there has been a framing or parity error (were parity to be turned on), the upper bits of the UART port will be set.

In a like manner, the GPS device can be written to. Certain strings, if sent to the UART, can be used to change the UARTs baud rate, its serial port settings, or even its reporting interval. As with the read port, the transmit port will interrupt the CPU when it is idle. Writing a character to this port will reset the interrupt. Setting bits other than the bottom eight may result in a break condition being set on this port as well.

Interacting with a controller can therefore be somewhat tricky. The interrupt controller will trigger whenever the port is ready to be read from, and will re–trigger every clock until the port has been read from. At this point, the interrupt controller may be reset. If this is an auxilliary interrupt controller, such as the bus interrupt controller or the ZipSystem's auxiliary controller, the auxiliary controller will then need to be reset, and the bit in the primary controller associated with the auxiliary controller as well. It is for this reason that the UARTs have been placed on the primary controller only.

It should also be possible to use the DMA to read from (or write to) either UART port.

2.0.16 Auxilliary UART

The Auxilliary UART has roughly the same structure as the GPS UART, save that it's default configuration is for a 115,200 Baud configuration with 8 data bits, no stop bits, and no parity. Reads, writes, and interrupts are treated in the same fashion.

2.0.17 GPIO

A General Purpose I/O controller has been placed within the design as well. This controller can handle 16–generic input wires, and set 16–generic output wires. A single register is used to read both input and output wire values, as well as to set output values when written to.

However, to use this controller, you will need to manually configure it (i.e. change the Verilog source) within the core, in order to wire the various GPIO values up to a device of interest. This was done for the simple reason that wiring anything new up to the controller will require Verilog changes anyway. For this reason, the controller has no way of setting wires to high impedence, or pulling them up or down. Such control may be done within the top level design if necessary.

This controller will set an interrupt if ever any of the input wires within it are changed. The interrupt may be cleared in the interrupt controller.

2.0.18 Linker Script

A linker script has been created to capture the memory structure needed by a program. This script may be found in `sw/board/arty.ld`. It is a sample script, using it is not required.

The script defines three types of memory to the linker: flash, block RAM, and SDRAM. Programs using this script will naturally start in flash (acting as a ROM memory). A bootloader must then be used to copy, from flash, those sections of the program that are to be placed in block RAM or SDRAM into their particular memory locations.

The block RAM locations are reserved for the user kernel, and specifically for any part of the code in the `.kernel` section. C attributes, or assembly `.section` commands, must be used to place items within this section. A final symbol within this section, _top_of_stack, is used so that the initial boot loader knows what to set the initial kernel stack to.

The rest of the initial program's memory is placed into SDRAM.[2] At the end, a `_top_of_heap` symbol is set to reference the final location in the setup. This symbol can then be used as a starting point for a memory allocator.

An example bootloader is provided in `sw/board` that can be linked with any (bare metal, supervisor) program in order to properly load it into memory.

[2] Hopefully, I'll get a data cache running on the ZipCPU to speed this up.

3.

Software

3.1 Directory Structure

3.2 Zip CPU Tool Chain

3.3 Bench Test Software

3.4 Host Software

- `readflash`: As I am loathe to remove anything from a device that came factory installed, the `readflash` program reads the original installed configuration from the flash and dumps it to a file.
- `wbregs`: This program offers a capability very similar to the PEEK and POKE capability Apple user's may remember from before the days of Macintosh. `wbregs <address>` will read from the Wishbone bus the value at the given address. Likewise `wbregs <address> <value>` will write the given value into the given address. While both address and value have the semantics of numbers acceptable to `strtoul()`, the address can also be a named address. Supported names can be found in `regdefs.cpp`, and their register mapping in `regdefs.h`.
- `ziprun`:
- `zipload`:

3.5 Zip CPU Programs

- `ntpserver`:
- `goldenstart`:

3.6 ZipOS

3.6.1 System Calls

- `int wait(unsigned event_mask, int timeout)`

- `int clear(unsigned event_mask, int timeout)`
- `void post(unsigned event_mask)`
- `void yield(void)`
- `int read(int fid, void *buf, int len)`
- `int write(int fid, void *buf, int len)`
- `unsigned time(void)`
- `void *malloc(void)`
- `void free(void *buf)`

3.6.2 Scheduler

4.

Operation

5.

Registers

There are several address regions on the S6 SoC, as shown in Tbl. 5.1.

Binary Address	Base	Size(W)	Purpose
0000 0000 0000 0000 0001 000x xxxx	0x00000100	32	Peripheral I/O Control
0000 0000 0000 0000 0001 0010 0yyx	0x00000120	8	Debug scope control
0000 0000 0000 0000 0001 0010 10xx	0x00000128	4	RTC control
0000 0000 0000 0000 0001 0010 11xx	0x0000012c	4	SDCard controller
0000 0000 0000 0000 0001 0011 00xx	0x00000130	4	GPS Clock loop control
0000 0000 0000 0000 0001 0011 01xx	0x00000134	4	OLEDrgb control
0000 0000 0000 0000 0001 0011 1xxx	0x00000138	8	Network packet interface
0000 0000 0000 0000 0001 0100 0xxx	0x00000140	8	GPS Testbench
0000 0000 0000 0000 0001 0100 1xxx	0x00000148	8	*Unused*
0000 0000 0000 0000 0001 0101 xxxx	0x00000150	16	*Unused*
0000 0000 0000 0000 0001 011x xxxx	0x00000160	32	*Unused*
0000 0000 0000 0000 0001 100x xxxx	0x00000180	32	*Unused*
0000 0000 0000 0000 0001 101x xxxx	0x000001a0	32	Ethernet configuration registers
0000 0000 0000 0000 0001 110x xxxx	0x000001c0	32	Extended Flash Control Port
0000 0000 0000 0000 0001 111x xxxx	0x000001e0	32	ICAPE2 Configuration Port
0000 0000 0000 0000 10xx xxxx xxxx	0x00000800	1k	Ethernet RX Buffer
0000 0000 0000 0000 11xx xxxx xxxx	0x00000c00	1k	Ethernet TX Buffer
0000 0000 0000 1xxx xxxx xxxx xxxx	0x00008000	32k	On-chip Block RAM
0000 01xx xxxx xxxx xxxx xxxx xxxx	0x00400000	4M	QuadSPI Flash
0000 0100 0000 0000 0000 0000 0000	0x00400000		Configuration Start
0000 0100 0111 0000 0000 0000 0000	0x00470000		Alternate Configuration
0000 0100 1110 0000 0000 0000 0000	0x004e0000		CPU Reset Address
01xx xxxx xxxx xxxx xxxx xxxx xxxx	0x04000000	64M	DDR3 SDRAM
1000 0000 0000 0000 0000 0000 000x	0x08000000	2	ZipCPU debug control port—only visible to debug WB master

Table 5.1: Address Regions

Base	Size(W)	Purpose
0x0c0000000	1	Primary Zip PIC
0x0c0000001	1	Watchdog Timer
0x0c0000002	1	Bus Watchdog Timer
0x0c0000003	1	Alternate Zip PIC
0x0c0000004	1	ZipTimer-A
0x0c0000005	1	ZipTimer-B
0x0c0000006	1	ZipTimer-C
0x0c0000007	1	ZipJiffies
0x0c0000008	1	Master task counter
0x0c0000009	1	Master prefetch stall counter
0x0c000000a	1	Master memory stall counter
0x0c000000b	1	Master instruction counter
0x0c000000c	1	User task counter
0x0c000000d	1	User prefetch stall counter
0x0c000000e	1	User memory stall counter
0x0c000000f	1	User instruction counter
0x0c0000010	1	DMA command register
0x0c0000011	1	DMA length
0x0c0000012	1	DMA source address
0x0c0000013	1	DMA destination address
0x0c0000040	1	*Reserved for MMU context register*
0x0c0000080	32	*Reserved for MMU TLB*

Table 5.2: ZipSystem Addresses

5.1 Peripheral I/O Control

Tbl. 5.3 shows the addresses of various I/O peripherals included as part of the SoC. We'll walk

Name	Address	Width	Access	Description
VERSION	0x0100	32	R	Build date
PIC	0x0101	32	R/W	Bus Interrupt Controller
BUSERR	0x0102	32	R	Last Bus Error Address
PWRCOUNT	0x0103	32	R	Ticks since startup
BTNSW	0x0104	32	R/W	Button/Switch controller
LEDCTRL	0x0105	32	R/W	LED Controller
AUXSETUP	0x0106	29	R/W	Auxilliary UART config
GPSSETUP	0x0107	29	R/W	GPS UART config
CLR-LEDx	0x0108-b	32	R/W	Color LED controller
RTCDATE	0x010c	32	R/W	BCD Calendar Date
GPIO	0x010d	32	R/W	*Reserved for* GPIO controller
UARTRX	0x010e	32	R/W	Aux UART receive byte
UARTTX	0x010f	32	R/W	Aux UART transmit byte
GPSRX	0x0110	32	R/W	GPS UART receive byte
GPSTX	0x0111	32	R/W	GPS UART transmit byte
GPSSECS	0x0110	32	R/W	*Reserved for a one-up seconds counter*
GPSSUB	0x0110	32	R/W	GPS PPS tracking subsecond info
GPSSTEP	0x0111	32	R/W	Current GPS step size, units TBD

Table 5.3: I/O Peripheral Registers

through each of these peripherals in turn, describing how they work.

5.1.1 Interrupt Controller

The OpenArty design maintains three interrupt controllers. Two of them are found within the ZipSystem, and the third is located on the bus itself. Of these, the primary interrupt controller is located in the ZipSystem. This interrupt controller accepts, as interrupt inputs, the outputs of both the auxilliary interrupt controller as well as the bus interrupt controller. Hence, even though the CPU only supports a single interrupt line, by using these three interrupt controllers many more interrupts can be supported.

The primary interrupt controller handles interrupts from the sources listed in Tbl. 5.4. These interrupts are listed together with the mask that would need to be used when referencing them to the interrupt controller. In a similar fashion, the auxilliary interrupt controller accepts inputs from the sources listed in Tbl. 5.5. Finally, the bus interrupt controller handles the interrupts from the sources listed in Tbl. 5.6.

Name	Bit Mask	DMAC ID	Description
SYS_DMAC	0x0001		The DMA controller is idle.
SYS_JIF	0x0002	1	A Jiffies timer has expired.
SYS_TMC	0x0004	2	Timer C has timed out.
SYS_TMB	0x0008	3	Timer C has timed out.
SYS_TMA	0x0010	4	Timer C has timed out.
SYS_AUX	0x0020	5	The auxilliary interrupt controller sends an interrupt
SYS_PPS	0x0040	6	An interrupt marking the top of the second
SYS_NETRX	0x0080	7	A packet has been received via the network
SYS_NETTX	0x0100	8	The network controller is idle, having sent its last packet
SYS_UARTRX	0x200	9	A character has been received via the UART
SYS_UARTTX	0x400	10	The transmit UART is idle, and ready for its next character.
SYS_GPSRX	0x0800	11	A character has been received via GPS
SYS_GPSTX	0x1000	12	The GPS serial port transmit is idle
SYS_SDCARD	0x2000	13	The SD-Card controller has become idle
SYS_OLED	0x4000	14	The OLED port is idle

Table 5.4: Primary System Interrupts

Name	Bit Mask	DMAC ID	Description
AUX_UIC	0x0001	16	The user instruction counter has overflowed.
AUX_UPC	0x0002	17	The user prefetch stall counter has overflowed.
AUX_UOC	0x0004	18	The user ops stall counter has overflowed.
AUX_UTC	0x0008	19	The user clock tick counter has overflowed.
AUX_MIC	0x0010	20	The supervisor instruction counter has overflowed.
AUX_MPC	0x0020	21	The supervisor prefetch stall counter has overflowed.
AUX_MOC	0x0040	22	The supervisor ops stall counter has overflowed.
AUX_MTC	0x0080	23	The supervisor clock tick counter has overflowed.
AUX_RTC	0x0100	24	An alarm or timer has taken place (assuming the RTC is installed, and includes both alarm or timer)
AUX_BTN	0x0200	25	A button has been pressed
AUX_SWITCH	0x0400	26	A switch has changed state
AUX_FLASH	0x0800	27	The flash controller has completed a write/erase cycle
AUX_SCOPE	0x1000	28	The Scope has completed its collection
AUX_GPIO	0x2000	29	The GPIO input lines have changed values.

Table 5.5: Auxilliary System Interrupts

Name	Bit Mask	Description
BUS_BUTTON	0x0001	A Button has been pressed.
BUS_SWITCH	0x0002	The Scope has completed its collection
BUS_PPS	0x0004	Top of the second
BUS_RTC	0x0008	An alarm or timer has taken place (assuming the RTC is installed, and includes both alarm or timer)
BUS_NETRX	0x0010	A packet has been received via the network
BUS_NETTX	0x0020	The network controller is idle, having sent its last packet
BUS_UARTRX	0x0040	A character has been received via the UART
BUS_UARTTX	0x0080	The transmit UART is idle, and ready for its next character.
BUS_GPIO	0x0100	The GPIO input lines have changed values.
BUS_FLASH	0x0200	The flash device has finished either its erase or write cycle, and is ready for its next command. (Alternate config only.)
BUS_SCOPE	0x0400	A scope has completed collecting.
BUS_GPSRX	0x0800	A character has been received via GPS
BUS_SDCARD	0x1000	The SD-Card controller has become idle
BUS_OLED	0x2000	The OLED interface has become idle
BUS_ZIP	0x4000	True if the ZipCPU has come to a halt

Table 5.6: Bus Interrupts

5.1.2 Last Bus Error Address

5.1.3 General Purpose I/O

5.1.4 UART Data Register

5.2 Debugging Scopes

5.3 Internal Configuration Access Port

5.4 Real–Time Clock

5.5 On-Chip Block RAM

5.6 Flash Memory

Name	Address	Width	Access	Description
ewreg	0x0180	32	R	Erase/write control and status
status	0x0181	8	R/W	Bus Interrupt Controller
nvconf	0x0182	16	R	Last Bus Error Address
vconf	0x0183	8	R	Ticks since startup
evonc	0x0184	8	R/W	Button/Switch controller
lock	0x0185	8	R/W	LED Controller
flagstatus	0x0186	8	R/W	Auxilliary UART config
clear	0x0187	8	R/W	Clear status on write
Device ID	0x0188--0x018c	5x32	R	Device ID
asyncOTP	0x18e	32	W	Asynch Read OTP. Write starts the ASynch read, 0xff returned until complete
OTP	0x0190--0x19f	16x32	R/W	OTP Memory

Table 5.7: Flash control registers

6.

Wishbone Datasheet

The master and slave interfaces have been simplified with the following requirement: the `STB` line is not allowed to be high unless the `CYC` line is high. In this fashion, a slave may often be able to ignore `CYC` and only act on the presence of `STB`, knowing that `CYC` must be active at the same time.

7.

Clocks

Name	Source	Rates (MHz)		Description
		Max	Min	
i_clk_100mhz	Ext	100		100 MHz Crystal Oscillator
Future s_clk	PLL	152	166	Internal Logic, Wishbone Clock
s_clk	PLL	83.33	75.76	DDR3 SDRAM Controller Clock
mem_clk_200mhz		200 MHz		MIG Reference clock for PHASERs
ddr3_ck_*x*	DDR	166.67	303	DDR3 Command Clock
o_qspi_sck	DDR	95		QSPI Flash clock
o_sd_clk	Logic	50	0.100	SD–Card clock
o_oled_sck	Logic	166		OLED SPI clock
o_eth_mdclk	Logic	25	2.5	Ethernet MDIO controller clock

Table 7.1: OpenArty clocks

8.

I/O Ports

Table. 8.1 lists the various I/O ports associated with OpenArty.

Port	Width	Direction	Description
i_clk_100mhz	1	Input	Clock
o_qspi_cs_n	1	Output	Quad SPI Flash chip select
o_qspi_sck	1	Output	Quad SPI Flash clock
io_qspi_dat	4	Input/Output	Four-wire SPI flash data bus
i_btn	4	Input	Inputs from the two on-board push-buttons
i_sw	4	Input	Inputs from the two on-board push-buttons
o_led	4	Output	Outputs controlling the four on-board LED's
o_clr_led0	3	Output	
o_clr_led1	3	Output	
o_clr_led2	3	Output	
o_clr_led3	3	Output	
i_uart_rx	1	Input	UART receive input
o_uart_tx	1	Output	UART transmit output
i_aux_rx	1	Input	Auxiliary/Pmod UART receive input
o_aux_tx	1	Output	Auxiliary/Pmod UART transmit output
i_aux_rts	1	Input	Auxiliary/Pmod UART receive input
o_aux_cts	1	Output	Auxiliary/Pmod UART transmit output
i_gps_rx	1	Input	GPS/Pmod UART receive input
o_gps_tx	1	Output	GPS/Pmod UART transmit output
i_gps_pps	1	Input	GPS Part-per-second (PPS) signal
i_gps_3df	1	Input	GPS
o_oled_cs_n	1	Output	
o_oled_sck	1	Output	
o_oled_mosi	1	Output	
i_oled_miso	1	Input	
o_oled_reset	1	Output	
o_oled_dc	1	Output	
o_oled_en	1	Output	
o_oled_pmen	1	Output	
o_sd_sck	1	Output	SD Clock
i_sd_cd	1	Input	Card Detect
i_sd_wp	1	Input	Write Protect
io_cmd	1	In/Output	SD Bi-directional command wire
io_sd	4	In/Output	SD Bi-directional data lines
o_cls_cs_n	1	Output	CLS Display chip select
o_cls_sck	1	Output	CLS Display clock
o_cls_mosi	1	Output	CLS Display MOSI
i_cls_miso	1	Input	CLS Display MISO

Table 8.1: List of IO ports