CMOD S6 SOC SPECIFICATION

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

May 14, 2016

Copyright (C) 2016, Gisselquist Technology, LLC

This project is free software (firmware): you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTIBILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see `http://www.gnu.org/licenses/` for a copy.

Revision History

Rev.	Date	Author	Description
0.2	5/14/2016	Gisselquist	Updated Draft, still not complete
0.1	4/22/2016	Gisselquist	First Draft

Contents

Page

1 Introduction . . . 1

2 Architecture . . . 2

3 Software . . . 4
3.1 Directory Structure . . . 4
3.2 ZipCPU Tool Chain . . . 5
3.3 Bench Test Software . . . 5
3.4 Host Software . . . 5
3.5 ZipCPU Programs . . . 6
3.6 ZipOS . . . 6
3.6.1 Traps . . . 7
3.6.2 Scheduler . . . 8

4 Operation . . . 9

5 Registers . . . 10
5.1 Peripheral I/O Control . . . 10
5.1.1 Interrupt Controller . . . 11
5.1.2 Last Bus Error Address . . . 11
5.1.3 ZipTimer . . . 11
5.1.4 PWM Audio Controller . . . 12
5.1.5 Special Purpose I/O . . . 12
5.1.6 General Purpose I/O . . . 13
5.1.7 UART Data Register . . . 14
5.2 Debugging Scope . . . 14
5.3 Internal Configuration Access Port . . . 14
5.4 Real–Time Clock . . . 15
5.5 On-Chip Block RAM . . . 15
5.6 Flash Memory . . . 15

6 Clocks . . . 16

7 IO Ports . . . 17

Figures

Figure Page

2.1. CMod S6 SoC Architecture: ZipCPU and Peripherals 3
2.2. Alternate CMod S6 SoC Architecture: Peripherals, with no CPU 3

5.1. Programmable Interrupt Control (PIC) Register 11
5.2. PWM Audio Controller Bitfields 12
5.3. SPIO Control Register 13
5.4. GPIO Control Register 13
5.5. Spartan–6 ICAPE Usage 14

Tables

Table Page

5.1. Address Regions 10
5.2. I/O Peripheral Registers 10
5.3. Flash Address Regions 15

7.1. List of IO ports 17
7.2. Physical Locations of Device I/O Ports 18

Preface

The Zip CPU was built with the express purpose of being an area optimized, 32–bit FPGA soft processor.

The S6 SoC is designed to prove that the ZipCPU has met this goal.

Dan Gisselquist, Ph.D.

1. Introduction

This project is ongoing. Any and all files, to include this one, are subject to change without notice.

This project comes from my desire to demonstrate the Zip CPU's utility in a challenging environment. The CMod S6 board fits this role nicely.

1. The Spartan–6 LX4 FPGA is very limited in its resources: It only has 2,400 look–up tables (LUTs), and can only support a 4,096 Word RAM memory (16 kB).
2. With only 4kW RAM, the majority of any program will need to be placed into and run from flash. (The chip will actually support more, just not 8k RAM.)
3. While the chip has enough area for the CPU, it doesn't have enough area to include the CPU and . . . write access to the flash, debug access, wishbone command access from the UART, pipelined CPU operations, and more. Other solutions will need to be found.

Of course, if someone just wants the functionality of a small, cheap, CPU, this project does not fit that role very well. While the S6 is not very expensive, it is still an order of magnitude greater than it's CPU competitors in price. This includes such CPU's as the Raspberry Pi Zero, or even the TeensyLC.

If, on the other hand, what you want is a small, cheap, CPU that can be embedded within an FPGA without using too much of the FPGA's resources, this project will demonstrate that utility and possibility. Alternatively, if you wish to study how to get a CPU to work in a small, constrained environment, this project may be what you are looking for.

2. Architecture

Fig. 2.1 shows the basic internal architecture of the S6 SoC. In summary, it consists of a CPU coupled with a variety of peripherals for the purpose of controlling the external peripherals of the S6: flash, LEDs, buttons, and GPIO. External devices may also be added on, such as an audio device, an external serial port, an external keypad, and an external display. All of these devices are then available for the CPU to interact with.

If you are familiar with the Zip CPU, you'll notice this architecture provides no access to the Zip CPU debug port. There simply wasn't enough room on the device. Debugging the ZipCPU will instead need to take place via other means, such as dumping all registers and/or memory to the serial port on any reboot.

Further, the ZipCPU has no ability to write to flash memory. For this reason, there exists an alternate CMod S6 SoC architecture, as shown in Fig. 2.2. Using this alternate architecture, it should be possible to test the peripherals and program the flash memory. Both architectures may be loaded into the flash, together with the programming code for the Zip CPU.

The basic approach to loading the board is actually quite simple. Using the Digilent ADEPT JTAG configuration program, `djtgcfg`, the alternate configuration may be written directly to the device. Once this alternate configuration has been loaded, the flash may be examined and programmed. This includes programming a primary and alternate configuration into the configuration section of the flash. Once complete, the system may then be reloaded with the primary configuration file which will contain an image of the CPU. The CPU will then begin following the instructions found in flash memory.

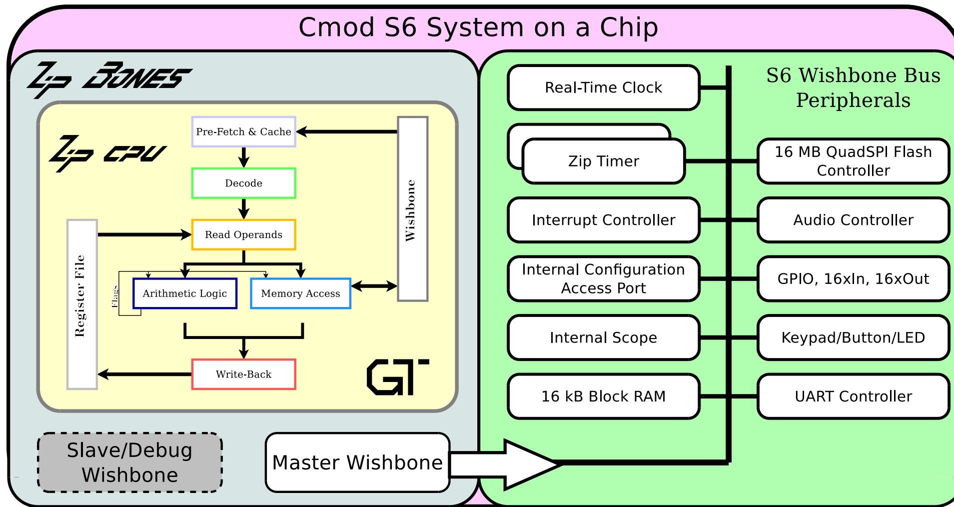

Figure 2.1: CMod S6 SoC Architecture: ZipCPU and Peripherals

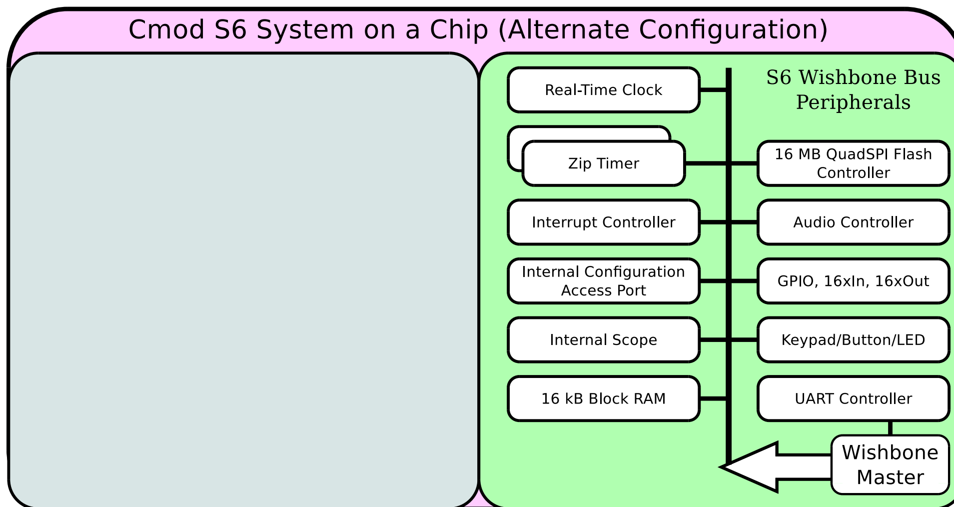

Figure 2.2: Alternate CMod S6 SoC Architecture: Peripherals, with no CPU

3. Software

3.1 Directory Structure

trunk/bench Contains software for emulating the S6 without the S6 present.

trunk/bench/cpp All of the bench testing software is written in C++, so it is found in this directory. Primary among these programs is the `zip_sim` program which will simulate the ZipCPU within the S6–Soc. Specifically, it simulates everything at or below the `busmaster.v` level.

Some, although not all, of the peripherals have been simulated and made a part of this simulation. These include the Quad–SPI flash, the UART, and the LED's.

trunk/doc All of the documentation for the S6SoC project may be found in this documentation directory. Specifically, I would commend your attention to anything with a `.pdf` extension, as these are the completed documents. Among these you should find a copy of the GPL copyright under which this software is released, as well as a pre–built copy of this document.

trunk/doc/gfx Here is where the graphics are located in support of this specification document.

trunk/doc/src And here is where the LaTeXfiles are kept that were used in building both this document as well as the GPL copyright.

trunk/rtl Verilog files

trunk/rtl/cpu Verilog files containing the ZipCPU core and peripherals. The toplevel file here is the `zipbones.v` file, although some of the peripherals, such as the `ziptimer.v` are referenced independently.

trunk/sw The main software directory, primarily a repository for software subdirectories beneath it.

trunk/sw/dev This directory holds a variety of simple programs for the ZipCPU, such as `helloworld`, `doorbell` and `doorbell2`, as well as software drivers for various peripherals, such as the real–time clock simulator, and the keypad and display device drivers.

trunk/sw/host This directory holds support software which can be built on and run on the host machine. Building this software will involve adjusting the Makefile so that it knows where your local ADEPT installation directory is.[1] Once built, you will find a variety of very useful programs within here.

[1] Many of the programs also depend upon the serial number of my CMod S6 device. This will need to be adjusted in any new install.

`trunk/sw/zipos` This directory contains the source code for a rudimentary, very basic, operating system that I call the ZipOS.

3.2 ZipCPU Tool Chain

To build programs for the ZipCPU, you will need the ZipCPU toolchain. You can find this as part of the ZipCPU project, available at OpenCores. Building the ZipCPU project should result in a set of binaries in the `trunk/sw/install/cross-tools/bin` directory. Make this directory a part of your path, and you should be able to build the CMod S6 ZipCPU software. Specifically, you will need to use `zip-gcc`, `zip-as` `zip-ld`, and `zip-cpp`. Other tools, such as `zip-objdump` and `zip-readelf`, may also prove to be very useful when trying to figure out what is going on within the SoC.

3.3 Bench Test Software

Bench testing software currently consists of the `zip_sim` program found within `trunk/bench/cpp`. This program requires Verilator to run, and simulates in a cycle accurate fashion, the entire S6SoC from `busmaster.v` on down. Further, the external Quad–SPI flash memory, UART, and LED's are also simulated, although the 2–line display, audio, and keypad are not.

3.4 Host Software

These include:

- `dumpuart`: My current approach to debugging involves dumping the state of the registers and memory to the UART upon reboot. The dumpuart command found here is designed to make certain that the UART is first set up correctly at 9600 Baud, and that second everything read from the UART is directly sent to both a file and to the screen. In this fashion, it is similar to the UNIX `tee` program, save for its serial port attachment.
- `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`
- `zipload`: This is the primary program you will need to get your software loaded on the CMod. It takes three arguments. The first is the name of the primary Xilinx configuration file, the second is the name of the alternate Xilinx configuration file, and the third is the name of the ZipCPU program you wish to write to Flash memory.

  Each of these arguments is optional. For example, if only one configuration file is given, the loader will load the primary configuration. If only one ZipCPU program is given, the program will be loaded into the program memory area and the configuration file areas will be left untouched.

3.5 ZipCPU Programs

- `helloworld`: The first program any programmer should build, "Hello, world!" This program sends the string, "Hello, world!" over the UART connection once per second. It is a very valuable program because, if you can get this program running, you know you have a lot of things working and working correctly. For example, running this program means you can run the `zip-gcc` compiler, load the auxiliar configuration, load the program info flash memory, load the primary configuration, and read from the UART port. It also means that you must have the UART port properly configured and wired to your CMod board.
- `doorbell`: This annoying program verifies the functionality of the audio device by playing a doorbell sound to the audio port. It will then wait ten seconds, and play the doorbell sound again (and again, and again). (It gets old after a while ...)
- `doorbell2`: This adds to the functionality of the `doorbell` program a wait for keypress, and a display of the current time on the 2–line display. While almost our fully functional program, this does not include any menus to configure the device or set time, since it doesn't include any keypad functionality.
- `kptest`: A test of whether or not they keypad driver works. When run, anytime a key is pressed, the key's value (in hex) will be sent to the UART. Further, pressing an 'F' on the keypad will also send a newline over the UART, in case you wish to keep your lines from getting too long.

3.6 ZipOS

This operating system is pre–emptive and multitasking, although with many limitations. Those familiar with the internals of the Linux kernel may laugh that I call this an Operating System at all: it has no memory management unit, no paging, no virtual memory, no file I/O access, no network stack, no ability to dynamically add or remove tasks, indeed it hardly has any of the things most often associated with an Operating System. It does, however, handle interrupts, support multiple pre–emptive tasks in a multitasking, timesharing fashion, and it supports some very basic and rudimentary system calls. In a similar fashion, it does contain just about all of the functionality necessary for a multi–tasking microcontroller built around a do–forever loop. For its size, I consider it an impressive achievement. You are welcome to disagree with me, however.

This version of the ZipOS starts in the resetdump.s code, so that upon any startup the ZipOS will dump register contents, the BusError register, and any scope contents to the UART. This can take some time, so you may wish to configure what you really wish to send–if anything. If desired, this will also dump the entire memory as well. All of this is quite useful in case the ZipCPU encounters a bus error or other sort of error that causes it to hang, stall, or reboot, as these registers are very carefully not touched prior to being sent to the UART output port.

`resetdump.s` also calls a rudimentary bootloader, to load the parts of the ZipOS that need to run faster into Block RAM. The choice of what parts to load into Block RAM is made on a file by file basis, and found within the linker script, `cmodram.ld`.

Upon completion, `resetdump.s` calls the entry routine for the O/S, `kernel_entry()` found in `kernel.c`. This is the main task loop for the entire O/S, and worth studying if you are interested in understanding how the O/S works.

The user tasks are found (mostly) within `doorbell.c`, also found in the ZipOS directory. This file contains two kernel entry points, `kntasks()`, which returns the number of tasks the kernel needs to know about, and `kinit()`, which builds each of the tasks and connects their file descriptors to the various devices they will be referencing.

3.6.1 Traps

The ZipCPU supports a variety of traps, listed here:

- WAIT: Halts the execution of a process until an event takes place, or a timeout has been reached. The events that can take place are a bitmask of the various interrupts the CPU supports, together with a bitmask of the values found in `swint.h`.

  The timeout value can either be zero, to return immediately with the list of events that have taken place, negative, to wait indefinitely, or a positive number of milliseconds in order to wait at least that time for the event of interest to take place.

  This also allows a process to sleep for any number of milliseconds.

  When wait returns, any events returned by the wait have been cleared.

  The other thing to be aware of is that events may accumulate before the wait system call. They will only be returned and cleared, though, if the wait call indicates an interest in those events.

- CLEAR: This system call works closely with the wait system call. Indeed, it is very similar to a wait system call with a zero timeout. It clears any of the requested events which may be pending. If given a timeout (in milliseconds), it will start a timer and generate a `SWINT_TIMEOUT` event to be sent to this process when the timer completes.

- POST: Certain devices, such as the real–time clock and the doorbell reader, need the ability of being able to post events to any listener within the O/S. The POST system call allows them to POST events in this manner.

- YIELD: This is simply a way of being nice to other processes. This system call takes no arguments and simply asks the scheduler to schedule the next process. It does not take this process off of the ready to run list, so the next process may be this one. However, since the scheduler is a round–robin scheduler, it will only return to this process if nothing else is available to run.

- READ: This is roughly the same system call as the POSIX read() system call. It reads some number of memory addresses (words, not octets), to the given file descriptor. If the memory requested is unavailable, the read will wait until it is available, possibly indefinitely.

- WRITE: This is roughly the same system call as the POSIX write() system call. It writes some number of memory addresses (words, not octets), to the given file descriptor. If nothing is reading from the device, the write stall the task forever, otherwise it will only stall the task until the data is either written to the receiving task, or copied into a memory buffer.

- TIME: Returns the number of seconds since startup. Eventually, this will return the number of seconds since January 1, 1970, and be identical to the UNIX system time() command, but that may not happen on this project.

- MALLOC: Allocates memory from the system/kernel heap. This is a very low overhead memory allocator that, while it does allocate memory, cannot free it later. It is nearly 100% efficient since only one memory address, the top of the heap, is used to determine what memory has been allocated.
- FREE: Not currently implemented.

3.6.2 Scheduler

The ZipCPU currently supports only a round–robin scheduler. Tasks are executed in the order they were created, as long as they are available to be executed. If no tasks are available to be run, the Scheduler will run the idle task which puts the CPU to sleep while waiting for an interrupt.

4. Operation

5. Registers

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

Start	End		Purpose
0x000100	0x000107	R/W	Peripheral I/O Control
0x000200	0x000201	R/(W)	Debugging scope
0x000400	0x00043f	R/W	Internal Configuration Access Port
0x000800	0x000803	R/W	RTC Clock (if present)
0x002000	0x002fff	R/W	16kB On-Chip Block RAM
0x400000	0x7fffff	R	16 MB SPI Flash memory

Table 5.1: Address Regions

that are made up of RAM or flash act like memory. The RAM can be read and written, and the flash acts like read only memory.

This isn't quite so true with the other address regions. Accessing the I/O region, while it may be read/write, may have side-effects. For example, reading from the debugging scope device's data port will read a word from the scope's buffer and advance the buffer pointer.

5.1 Peripheral I/O Control

Tbl. 5.2 shows the addresses of various I/O peripherals included as part of the SoC.

Name	Address	Width	Access	Description
PIC	0x0100	32	R/W	Interrupt Controller
BUSERR	0x0101	32	R	Last Bus Error Address
TIMA	0x0102	32	R/W	ZipTimer A
TIMB	0x0103	32	R/W	ZipTimer B
PWM	0x0104	32	R/W	PWM Audio Controller
SPIO	0x0105	32	R/W	Special Purpose I/O, Keypad, LED Controller
GPIO	0x0106	32	R/W	GPIO Controller
UART	0x0107	32	R/W	UART data

Table 5.2: I/O Peripheral Registers

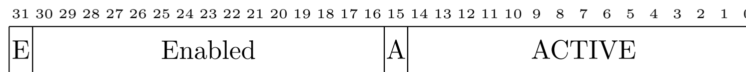

Figure 5.1: Programmable Interrupt Control (PIC) Register

5.1.1 Interrupt Controller

The interrupt controller is identical to the one found with the ZipSystem. The layout of the PIC bits is shown in Fig. 5.1. This controller supports up to fifteen interrupts, however only twelve are defined within the SoC. If any interrupt line is active, the PIC controller will have that bit set among it's active set. Once set, the bit and hence the interrupt can only be cleared by writing to the controller. Interrupts can also be enabled as well. The enabled bit mask controls which interrupt lines are permitted to interrupt the CPU. Hence, just because an interrupt is active doesn't mean it will interrupt the CPU–the enabled line must be set as well. Finally, then `A` or `ANY` bit will be high if any interrupts are both enabled and active, whereas the `E` or global interrupt enable bit can be set to allow the PIC to interrupt the CPU or cleared to disable all interrupts.

To keep operations on this register atomic, most of the bits of this register have special meanings upon write. The one exception to this is the global interrupt enable bit. On any write, interrupts will be globally enabled or disabled based upon the value of this bit. Further, the `ANY` bit is a read only bit, so writes to it have no effect.

Enabling specific interrupts, via writes to the enable lines, are different. To enable a specific interrupt, enable all interrupts and set the wire high associated with the specific interrupt you wish to enable. Hence writing a `0x80010000` will enable interrupt line zero, while also enabling all previously enabled interrupts. To disable a specific interrupt, disable all interrupts and write a one to the enable line of the interrupt you wish to disable. In this fashion, writing a `0x00010000` will disable all interrupts and leave interrupt line zero disabled when the interrupts are re–enabled later, whereas `0x07fff0000` will disable all specific interrupts.

Interrupts are acknowledged in a fashion similar to enabling interrupts. By writing a '1' to the active bit mask, the interrupt will be acknowledged and reset, whereas writing a '0' leaves the interrupt untouched. In this fashion, as individual interrupts are handled, a '1' may be written to this bottom mask to clear the interrupt. Be aware, however, that any interrupt acknowledgement may also globally enable or disable interrupts.

5.1.2 Last Bus Error Address

The Bus Error peripheral simply records the address of the last bus error. This can be useful when debugging. While the peripheral may only be read, setting it is really as easy as creating a bus error and trapping the result. Another use for this is upon any reboot, it is possible to read the address of the last bus error and perhaps learn something of what caused the CPU to restart.

5.1.3 ZipTimer

The S6 Soc contains two ZipTimers, available for the CPU to use. These are countdown timers. Writing any non–zero value to them will cause them to immediately start counting down from that

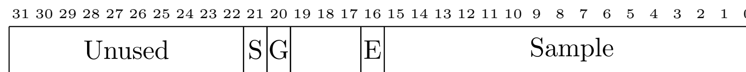

Figure 5.2: PWM Audio Controller Bitfields

value towards zero, and to interrupt the CPU when they reach zero. Writing a new value while the timer is running will cause that new value to automatically load into the CPU and start counting from there. Writing a zero to the timer disables the timer, and causes it to stop.

ZipTimer A can be set to auto reload. When set, the timer will automatically load it's last set value upon reaching zero and interrupting the CPU. This effectively turns it into an interrupt timer if desired. To set this feature, write to the timer the number of clock ticks before an interrupt, but also set the high order bit. In this fashion, writing a `0x80013880` will interrupt the CPU every millisecond, starting one millisecond after the write takes place (assuming an 80 MHz system clock).

ZipTimer B has been wired for a different purpose. ZipTimer B does not support auto reload, nor will it interrupt the CPU. Instead, ZipTimer B has been wired as a watchdog timer. When this timer reaches zero, the CPU will be rebooted. One way to use this timer would be in conjunction with the ZipTimer A, and to write a number to it upon any entry to the interrupt service routine. If given enough time, this would cause the CPU to reboot if for any reason it locked up.

5.1.4 PWM Audio Controller

The bit fields of the PWM Audio controller are shown in Fig. 5.2. This controller has been designed for easy writing. To send a sample to the PWM audio controller, simply write the sample to the controller and clear the PWM audio interrupt. When the audio interrupts the CPU again, it is ready for the next sample.

The audio sample rate has been fixed at 8 kHz. While changing this rate is easy to do within `busmaster.v`, the rate itself takes some work to keep up with, so I wouldn't recommend going much (any) faster.

The audio controller supports two additional functionalities, however. The first is that the `E` bit will be set upon any read when or if the audio controller is ready for another sample. Equivalently, the audio interrupt will be asserted.

The second functionality has to do with the two auxiliary control bits present in the PModAMP2 audio device. These are the gain and shutdown bits. To set these bits, write a sample to the controller while also setting the `E` bit. When the `E` bit is set upon any write, the shutdown and gain bits will also be set. (Be aware, the shutdown bit is negative logic.) Hence, one may start this interface by writing a `0x0310000` to the device, and later shut it back off by writing a `0x010000`.

5.1.5 Special Purpose I/O

Register `SPIO`, as shown in Fig. 5.3, is a Special Purpose Input/Output (SPIO) register. It is designed to control the on-board LED's, buttons, and keypad. Upon any read, the register reads the current state of the keypad column output, the keypad row input, the buttons and the LED's. Writing is more difficult, in order to make certain that parts of these registers can be modified atomically. Specifically, to change an LED, write the new value as well as a '1' to the corresponding LED change

	31–16	15–12	11–8	7–6	5–4	3–0
Read	Zeros	Kpad Col Out	Kpad Row In	00	Btn	LED
Write	Ignored	Col Out	Col Enable	LED Enable		LED

Figure 5.3: SPIO Control Register

31–16	15–0
Current Input Vals (x16)	Current Output Values (16-outs)
Output Change Enable	

Figure 5.4: GPIO Control Register

enable bit. The same goes for the keypad column output, a '1' needs to be written to the change enable bit in order for a new value to be accepted.

As examples, writing a `0x0ff` to the `SPIO` register will turn all LED's on, `0x0f0` will turn all LED's off, and `0x011` and `0x010` will turn LED0 on and then off again respectively.

The controller will generate a keypad interrupt whenever any row input is zero, and a button interrupt whenever any button value is a one. This also means that, once generated, the interrupt must be disabled until the key or button is released.

5.1.6 General Purpose I/O

The General Purpose Input and Output (GPIO) control register, shown in Fig. 5.4, is quite simple to use: when read, the top 16–bits indicate the value of the 16–input GPIO pins, whereas the bottom 16–bits indicate the value being placed on the 16–output GPIO pins. To change a GPIO pin, write the new pin's value to this register, together with setting the corresponding pin in the upper 16–bits. For example, to set output pin 0, write a `0x010001` to the GPIO device. To clear output pin 0, write a `0x010000`. This makes it possible to adjust some output pins independent of the others.

The GPIO controller, like the keypad or SPIO controller, will also generate an interrupt. The GPIO interrupt is generated whenever a GPIO input line changes. The interrupt is not selective: if any line changes, a GPIO interrupt will be generated. There are no do not care lines.

Of the 16 GPIO inputs and the 16 GPIO outputs, two lines have been taken for I2C support. GPIO line zero, for both input and output, is an I2C data line, `io_sda`, and GPIO line one is an I2C clock line, `io_scl`. If the output of either of these lines is set to zero, the GPIO controller will drive the line. Otherwise, the line is pulled up with a weak resistor so that other devices may pull it low. If either line is low, when the output control bit is high, it is an indicator that another device is sending data across these wires.

```
warmboot(uint32 address) {
    uint32_t *icape6 = (volatile uint32_t *)0x<ICAPE port address>;
    icape6[13] = (address<<2)&0x0ffff;
    icape6[14] = ((address>>14)&0x0ff)|((0x03)<<8);
    icape6[4] = 14;
    // The CMod S6 is now reconfiguring itself from the new address.
    // If all goes well, this routine will never return.
}
```

Figure 5.5: Spartan–6 ICAPE Usage

5.1.7 UART Data Register

Moving on to the UART ... although the UART module within the S6 SoC is highly configurable, as built the UART can only handle 9600 Baud, 8–data bits, no parity, and one stop bit. Changing this involves changing the constant `uart_setup` within `busmaster.v`. Further, the UART has only a single byte data buffer, so reading from the port has a real–time requirement associated with it–the data buffer must be emptied before the next value is read. Attempts to read from this port will either return an 8–bit data value from the port, or if no values are available it will return an `0x0100` indicating that fact. In general, reading from the UART port involves first waiting for the interrupt to be ready, second reading from the port itself, and then third immediately clearing the interrupt. (The interrupt cannot be cleared while data is waiting.) Writing to the UART port is done in a similar fashion. First, wait until the UART transmit interrupt is asserted, second write to the UART port, and then third clear the interrupt. As with the read interrupt, clearing the interrupt prior to writing to the port will have no effect.

5.2 Debugging Scope

The debugging scope consists of two registers, a control register and a data register. It needs to be internally wired to 32–wires, internal to the S6 SoC, that will be of interest later. For further details on how to configure and use this scope, please see the `WBSCOPE` project on OpenCores.

5.3 Internal Configuration Access Port

The Internal Configuration Access Port (ICAP) provides access to the internal configuration details of the FPGA. This access was designed so as to provide the CPU with the capability to command a different FPGA load. In particular, the code in Fig. 5.5 should reconfigure the FPGA from any given Quad SPI `address`.[1]

One subtle problem with this port is that it will not work if the CMod is plugged in to the USB JTAG port. It will only work if the CMod has been provided with an independent power supply, leaving the USB JTAG unplugged.

[1] According to Xilinx's technical support, this will only work if the JTAG port is not busy.

For further details, please see either the `WBICAPETWO` project on OpenCores as well as Xilinx's "Spartan-6 FPGA Configuration User Guide".

5.4 Real–Time Clock

The Real Time Clock will be included if there is enough area to support it. The four registers correspond to a clock, a timer, a stopwatch, and an alarm. If space is tight, the timer and stopwatch, or indeed the entire clock, may be removed from the design. For further details regarding how to set and use this clock, please see the `RTCCLOCK` project on OpenCores.

There is currently not enough area on the chip to support the Real–Time Clock together with all of the other peripherals listed here. You can adjust whether the clock is included or not by adjusting the `‘define` lines at the top of `busmaster.v`. For example, it may be possible to get the RTC back by disabling the ICAPE2 interface.

5.5 On-Chip Block RAM

The block RAM is the fastest memory available to the processor. It is also the *only* writeable memory available to the processor. Hence all non-constant program data *must* be placed into block RAM. The ZipCPU can also run instructions from the block RAM if extra speed is desired. When runnning from block RAM, the ZipCPU will nominally take 8 clocks per instruction, for an effective rate of 8 MIPS. Loads or stores to block RAM will take one instruction longer.

5.6 Flash Memory

The flash memory has been arbitrarily sectioned into three sections, one for a primary configuration, a second section for an alternate configuration file, and the third section for any program and data. These regions are shown in Tbl. 5.3. The host program `zipload` can be used to load a

Start	End		Purpose
0x400000	0x43ffff	R	Primary configuration space
0x440000	0x47ffff	R	Alternate configuration space
0x480000	0x7fffff	R	ZipCPU program memory

Table 5.3: Flash Address Regions

ZipCPU program and configuration files into this address space. To use it, first load the alternate configuration into the FPGA. Then pass it, as arguments, the primary, and alternate if desired, configuration files followed by the ZipCPU program file. Then, when the primary configuration is loaded again, perhaps upon power up, the ZipCPU will automatically start running from it's `RESET_ADDRESS`, `0x480000`.

When running from Flash memory, the ZipCPU will nominally take 52 clocks per instruction, for an effective speed of about 1.5 MIPS.

6. Clocks

The S6 SoC is designed to run off of one master clock. This clock is derived from the 8 MHz input clock on the board, by multiplying it up to 80 MHz.

7. IO Ports

See Table. 7.1.

Port	Width	Direction	Description
i_clk_8mhz	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	2	Input	Inputs from the two on-board push-buttons
o_led	4	Output	Outputs controlling the four on-board LED's
o_pwm	1	Output	Audio output, via pulse width modulator
o_pwm_shutdown_n, 1		Output	Audio output shutdown control
o_pwm_gain	1	Output	Audio output 20 dB gain enable
i_uart	1	Input	UART receive input
o_uart	1	Output	UART transmit output
i_uart_cts	1	Input	
o_uart_rts	1	Output	
i_kp_row	4	Output	Four wires to activate the four rows of the keypad
o_kp_col	4	Output	Return four wires, from the keypads columns
i_gpio	14	Output	General purpose logic input lines
o_gpio	14	Output	General purpose logic output lines
io_scl	1	Input/Output	I2C clock port
io_sda	1	Input/Output	I2C data port

Table 7.1: List of IO ports

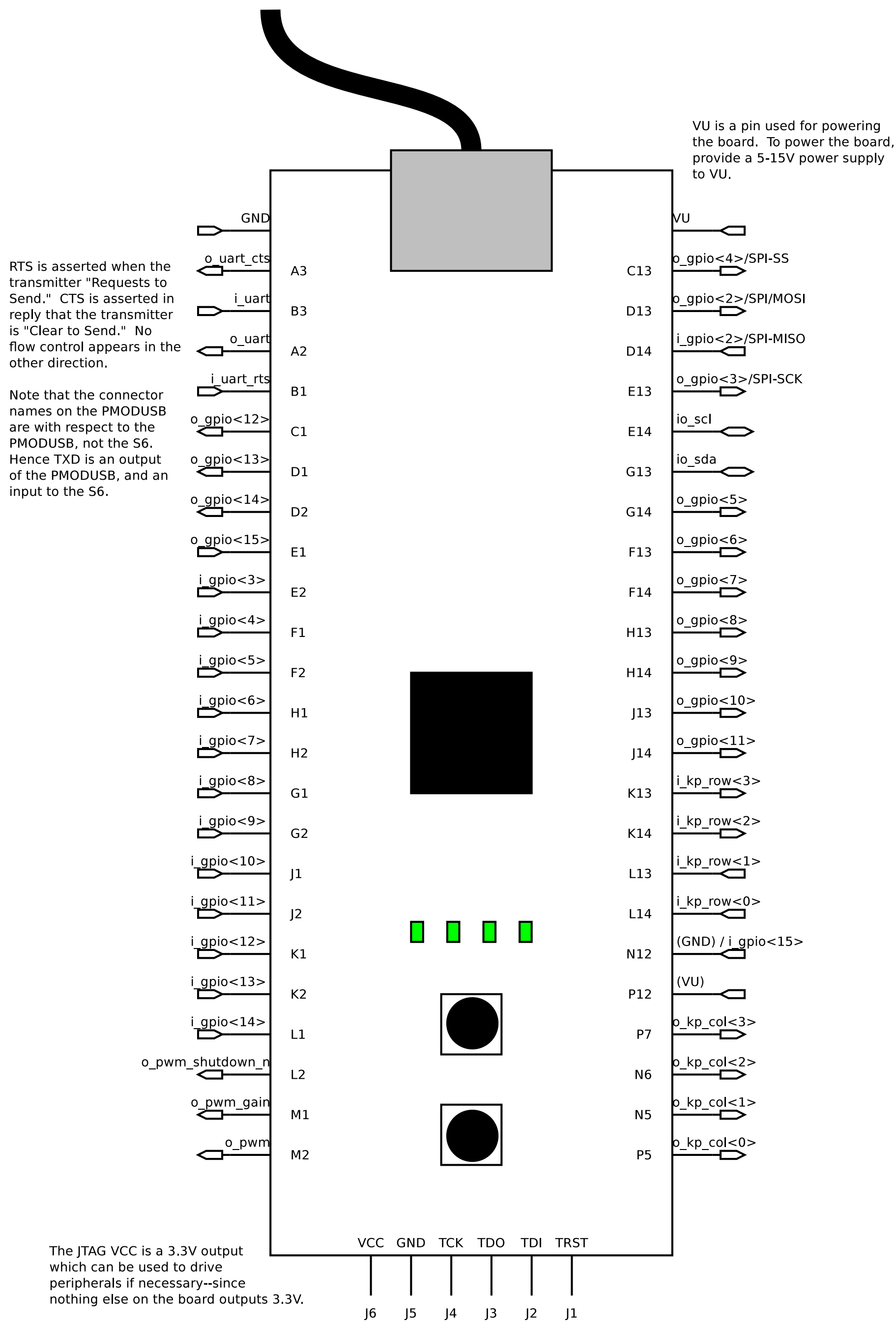

Table 7.2: Physical Locations of Device I/O Ports