CMOD S6 SOC SPECIFICATION

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

May 6, 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
0.1	4/22/2016	Gisselquist	First Draft

Contents

Page

1 Introduction 1

2 Architecture 2

3 Operation 4

4 Registers 5
- 4.1 Debugging Scope 5
- 4.2 Internal Configuration Access Port 5
- 4.3 Real–Time Clock 6
- 4.4 I/O Peripherals 6

5 Clocks 9

6 IO Ports 10

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

4.1. Spartan–6 ICAPE Usage 6
4.2. SPIO Control Register 7
4.3. GPIO Control Register 7

Tables

Table Page

4.1.	Address Regions	5
4.2.	I/O Peripheral Registers	6
6.1.	List of IO ports	10
6.2.	Physical Locations of Device I/O Ports	11

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 it's 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 is simple: up and until the software works, the S6 will power up into the alternate architecture of Fig. 2.2. While in this state, the flash may be examined and programmed. Once complete, a UART command to the ICAPE port will tell the S6 to load the (primary) FPGA configuration from an alternate flash location. This alternate location will contain a configuration image containing the CPU. The CPU will then begin following the instructions given to it from the flash.

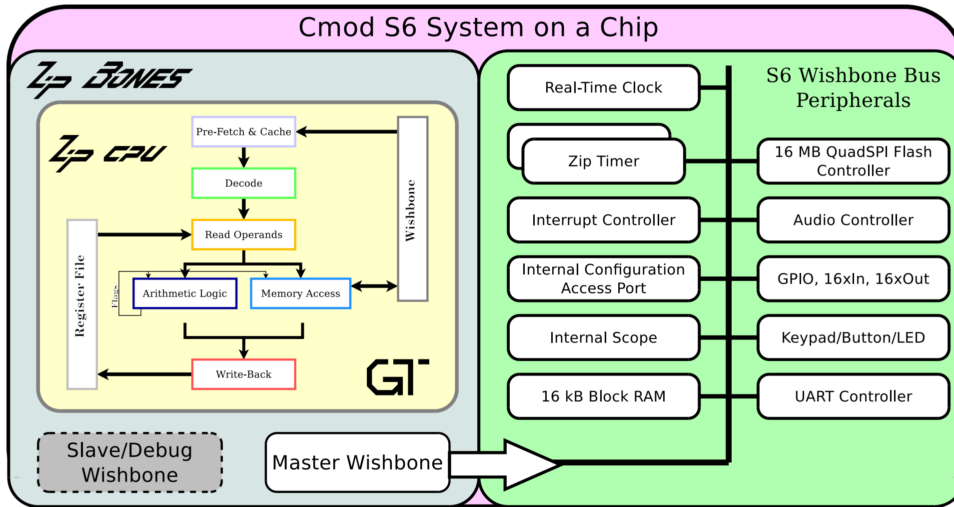

Figure 2.1: CMod S6 SoC Architecture: ZipCPU and Peripherals

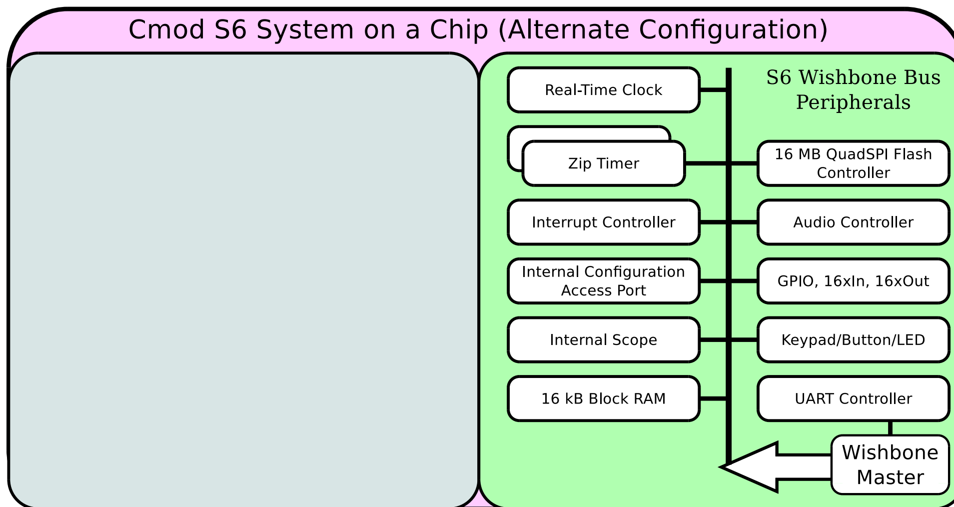

Figure 2.2: Alternate CMod S6 SoC Architecture: Peripherals, with no CPU

3. Operation

4. Registers

There are several address regions on the S6 SoC, as shown in Tbl. 4.1. 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 4.1: 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.

4.1 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.

4.2 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. 4.1 should reconfigure the FPGA from any given Quad SPI `address`.[1]

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

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

```
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 4.1: Spartan–6 ICAPE Usage

4.3 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.

4.4 I/O Peripherals

Tbl. 4.2 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
KYPAD	`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 4.2: I/O Peripheral Registers

The interrupt controller is identical to the one found with the ZipSystem. Please read the ZipSystem documentation for how to control this.

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.

The two ZipTimer's are ZipSystem timer's, placed onto this peripheral bus. They are available for the CPU to use. Common uses might include I2C or SPI speed control, or multi–tasking task-swap control. For further details, please see the ZipSystem documentation.

Audio Controller

	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 4.2: SPIO Control Register

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

Figure 4.3: GPIO Control Register

Register `KYPAD`, as shown in Fig. 4.2, 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 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.

The controller will generate a keypad interrupt whenever any row input is zero, and a button interrupt whenever any button value is a one.

The General Purpose Input and Output (GPIO) control register, shown in Fig. 4.3, 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 pins 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.

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, and GPIO line one is an I2C clock line. 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.

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. There is a single byte data buffer, so reading from the port has a real–time requirement associated with it.

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 a similar fashion, writes to this port will send the lower 8–bits of the write out the serial port. If the port is already busy, a single byte will be buffered.

5. 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.

6.

IO Ports

See Table. 6.1.

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 6.1: List of IO ports

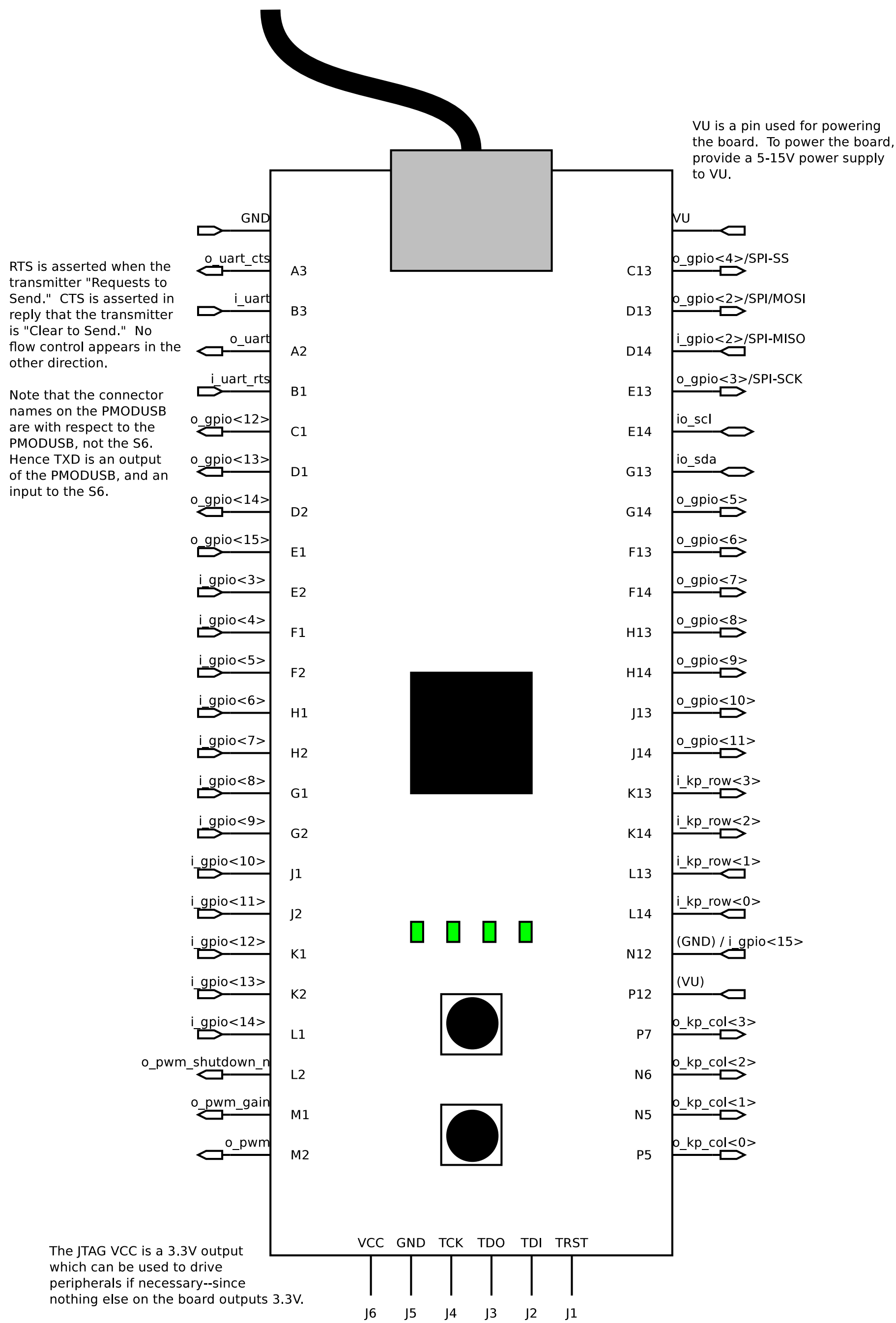

Table 6.2: Physical Locations of Device I/O Ports