Arduissimo Datasheet

Document Version 0.1-draft

Editor: Tobias Strauch
Munich, Bavaria
tobias at cloudx dot cc

December 20, 2019

Contributors to all versions of the spec in alphabetical order (please contact editors to suggest corrections): Tobias Strauch.

Preface

System Hyper Pipelining: With this project I want to demonstrate a certain digital design technique called system hyper pipelining (SHP), which creates high performance digital designs. The project is created to have a working example, which can be used for comparison. Please find more information on system hyper pipelining (SHP) technology online at `http://www.cloudx.cc/shp.html`.

ASIC vs. FPGA: The technique can be used for ASICs and FPGA alike. I'm just using an FPGA for this project. The hardware is optimized towards the famous low-cost ARTY board from Xilinx. There is no reason not to use SHP for designs targeting other FPGA devices or silicon chips.

RISC-V: I use the RISC-V for this project due to its raising popularity. Previous versions are based on a Cortex-M3 design, for instance. SHP is not limited to processors or particular processor families.

Dynamic Multi-threading and Virtual Peripherals: One benefit of SHP is its dynamic multi-threading capability. This project is optimized towards using an SHP-ed 32-bit RISC-V quad core MP-SoC to support virtual peripherals such as RS232. I2C, SPI, 1-write, CAN, PWM, ... Nevertheless, I believe that SHP also has a future in HPC, etc.. It is not limited to the capabilities shown by this particular project.

Arduissimo: For me the classical term Arduino stands for simple and easy microcontroller programming. Still, each microcontroller (Atmel, Mirochip, TI, ...) has its own set of peripherals and a microcontroller specific driver code is needed to use them. In this project, a reasonable set of virtual peripherals should be fairly easily re-usable. This aspect gave the project its name, the simplification and flexibility of peripheral programming on top of simple microcontroller programming, Arduissimo. Free the world from a fixed set of peripherals implemented in a fxied hardware!

IDE: The Arduissimo IDE still needs some work and is not released yet. Alternatively eclipse based Freedo Studio IDE from SiFive can be used. I hope that the community helps me in finding the right way to handle the hardware beast I created more efficiently. The problem is, that the SHP programming requires slightly more detailed programming than the processing method used in the Arduino world offers. There are still stack handling issues for such a simplified but multithreaded environment.

Future: I would love to see the specific ideas of this very projects somehow realized on an ASIC. I think the resulting virtual peripheral speeds are fast enough to replace the fixed set of peripherals on microcontroller families. The programmable realtime unit subsystem (PRU) on some of TI's

microcontroller is already an exampe for such a concept.

Contents

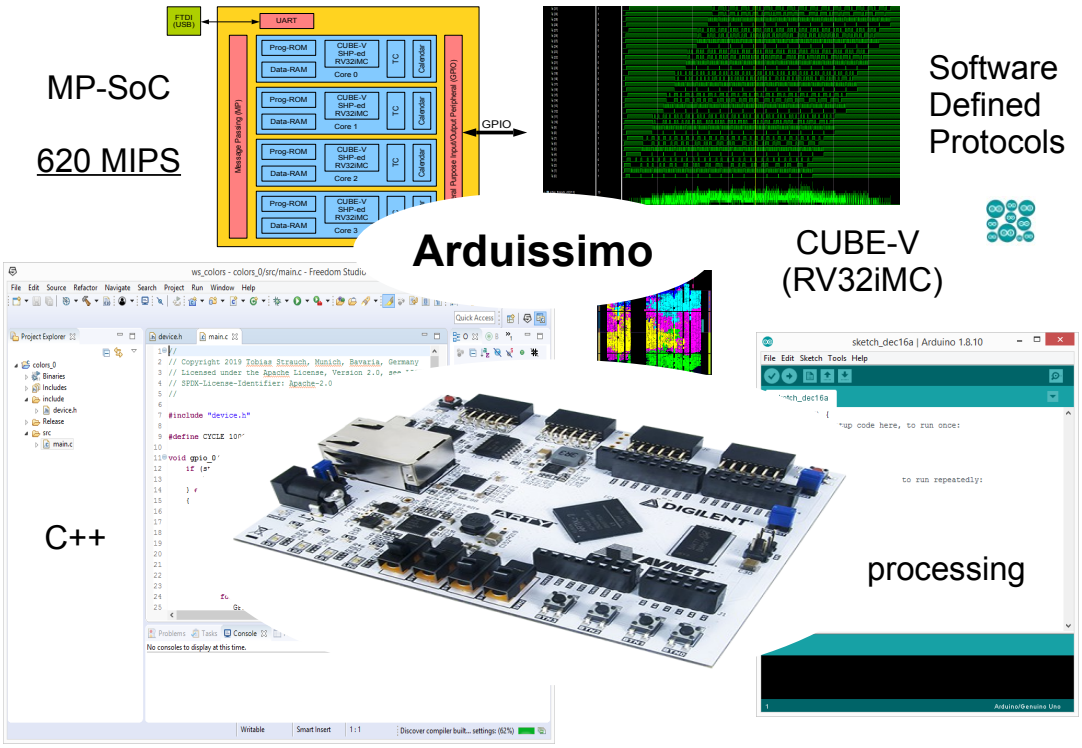

Figure 1.1: The big picture.

Chapter 1

System concept and relevant aspects at a glance

For those of you, who don't want to read the complete doc, so basically all of you, here are the key aspects of this project:

As already mentioned, the projects is created to demonstrate the benefits of system hyper pipelining

(SHP). We have 4 CUBE-V cores which run RISC-V 32-bit compatible code. Each one has 3 pipeline stages (P3). The complete design runs at 180MHz. This clock generates what we call a micro-cycle. We applied C-Slow-Retiming `http://www.cloudx.cc/csr.html` generating 4 copies of each CUBE-V core (C4). It therefore takes 4 micro-cycles to finish one functional cycle, or in other words one macro-cycle is equal to 4 micro-cycles. We then apply the final SHP-step, using a memory depth of 16 lines (D16) to be able run up to 16 threads at the same time on each core in a time sliced fashion.

The number of active threads can be changed dynamically. When less or equal 4 threads are executed, then each thread runs at a macro-cycle speed of (180MHz / 4 =) 45 MHz. When more than 4 threads are active, let's say n, then each thread runs at a macro-cycle speed of 180MHz / n. So for n = 10, to make is easy for you, each thread runs at 18 MHz.

A thread can be started by another thread or by a peripheral. There are no interrupts in the sense that a running program is interrupted, instead a new thread is started without affecting active threads (except timing-wise when n >= 4, as mentioned before). Only a thread can kill itself.

You can program at which program address a thread is started. So a thread can say, my dear thread controller, please start a new thread at 0xC0DE. You can also program any peripheral to start a new thread at a start address of your choice. You can also provide the new thread some extra information (like a task identification number for instance), which is automatically written into the link register a0 of the RISC-V register file. For example, in the a0 register the GPIO pin number is written, when an edge is detected at that given pin number.

Each core has a calendar, which can be seen as a complex timer. So you can program a list of future events into the calendar, which automatically sorts these events in a timely order and starts a thread at a programmable address, once the time for that event has come.

The CUBE-V core is based on the RV32IMC ISA, but the FENCE, ECALL and EBREAK instructions are not implemented. Also no control and system registers (CSR) are implemented.

The CUBE-V core achieves 0,86 IPC (instructions per cycle, here macro-cycle) based on the CH-Stone testcases. This relative high number comes from the fact, that register values are not only written into the register file (RF), but also applies already at the output of the RF when needed in the next cycle, so a RF-writethrough is implemented if you want. With an IPC of 0,86 one core reaches 155 MIPS at 180 MHz. Therefore 620 MIPS can be reached on the quad core setting that fits on the selected FPGA board.

The project is served on a bed of Windows project files with some Verilog strings attached.

Chapter 2

CFP and open source under the Apache License Version 2.0

This is the v0.1 version of this project and certainly far from being finished or even bullet proven.

Call for participation: I invite everybody who is interested to share ideas and thoughts about anything project related on the System Hyper Pipelining Google user forum `https://groups.google.com/forum/#!forum/shp_`. In fact, I reach out to you guys to solve some open issues in the most elegant way (e.g. stack handling in multithreaded environment, etc.).

The project is licensed under the Apache License, Version 2.0

Chapter 3

The shortest path to happiness

Two steps need to be executed to run a demo program after connecting the ARTY board to the PC:

First, the "bit" file needs to be downloaded into the FPGA. You can use the stand-alone LAB tool from Xilinx, or just open the fpga_top project by double click on the $project/vivado/fpga_top/fpga_top.xpr file. Select the $project/vivado/bit/fpga_top.bit file and download it onto the FPGA.

Second, one or (multiple) .hex files need to be downloaded, which takes in case of the color demo program approximately 20 seconds. Go to $project/hw/arty_ftdi/work/. When using cygwin just use "make colors" and "make downloadHex". The results will be reported once the download has finished. Alternatively use the colors.bat command when executed in the Command Prompt. Here you will see the download progress directly.

The 4 color LEDS on the ARTY board will blink as if there is no tomorrow. Each core blinks one LED.

Chapter 4

Release directory

The release on github is structured as follows:

constraints: Pinning constraints for the FPGA on ARTY board.

demo: Demo Freedom Studio software projects. With the project "colors", the design can be checked stand alone without additional hardware on the ARTY board. The other projects are set up to simulate virtual peripherals.

doc: This PDF document and Latex source for this document.

dut: Verilog files for the complete FPGA design.
dut/mem_sim: Memory simulation files. Used for simulation speed-up.
dut/mem_syn: Memory synthesis files. Used for synthesis.

ftdi/arty_ftdi: Supporting files to download "hex" files to the ARTY board via the FTDI chip.

sw: C include files for device, drivers, etc.. The file "device.h" lists the system registers.

testbench: Verilog testbench files.
testbench/testbench_P3C4D16.v: Regression on CUBE-V design, incl. CHStone tests.
testbench/testbench_quad.v: Runs regression on quad core implementation, testing the core's multithreading capabilities and interaction with peripherals.
testbench/testbench_top.v: Simulates the complete design, including the UART but without DCM.

tests: Freedom Studio software projects and C files for simulation and verification.
tests/basicC: Just some very basic C based tests.
tests/chstone: Self-testing CHStone tests to estimate MIPS.
tests/codeGenHex: C file to generate tests on assembler level.
tests/drivers: Freedom Studio software projects to test I2C, PWM, RS232 and SPI drivers.
tests/system: Freedom Studio software projects to test system level features.

vivado: Vivado project directory.
vivado/bit: Precompiled "bit" file of the design.
vivado/fpga_top: Syntheses and simulation project for top level.

vivado/simuate_quad: Simulation project on quad-core level.
vivado/simulate_cubev: Simulation project on CUBE-V level.

work: Makefile to compile "elf" file to "hex" files and to generate assembler test.

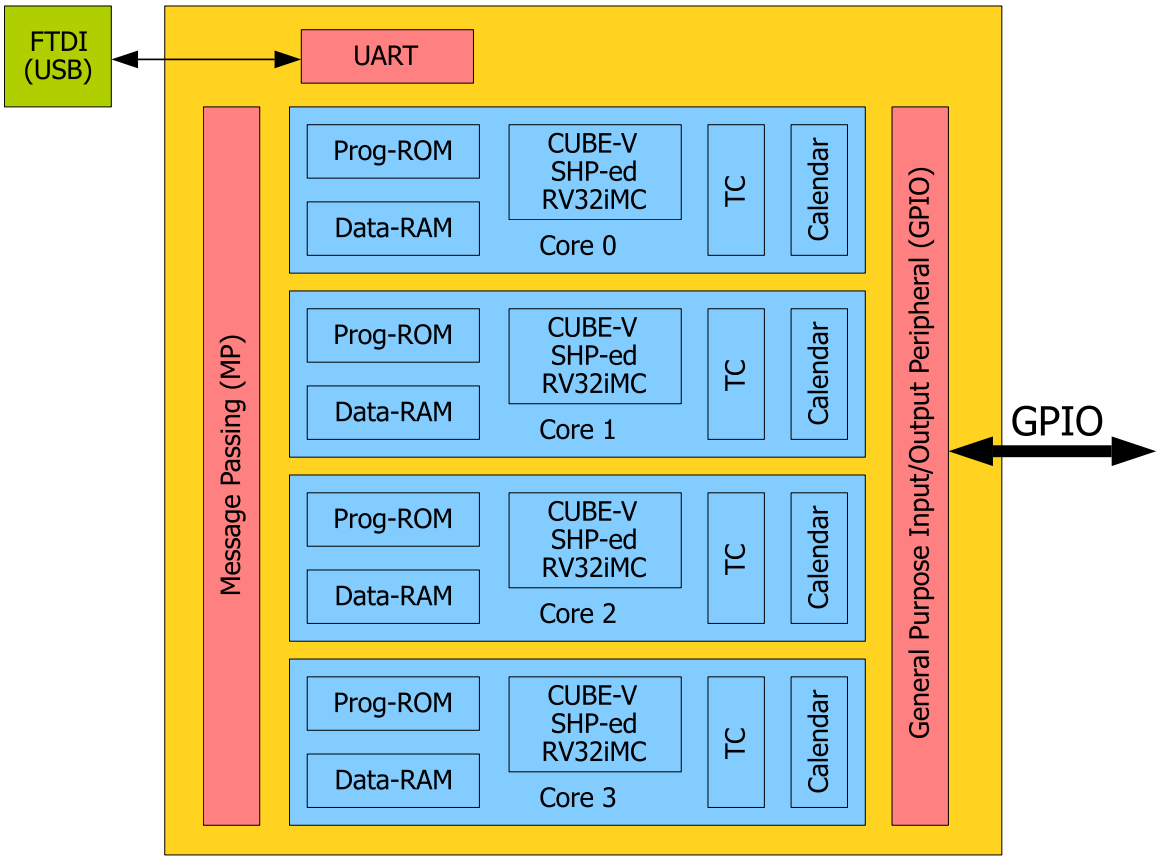

Figure 5.1: Overview of the FPGA design.

Chapter 5

The hardware

The hardware consists of 4 cores and some supporting peripherals (see Figure 5.1).

Each core has one CUBE-V processor, program and data memories as well one thread controller (TC) and one calendar (CA).

The peripherals are one UART (which is connected to the FTDI chip on the ARTY board), one

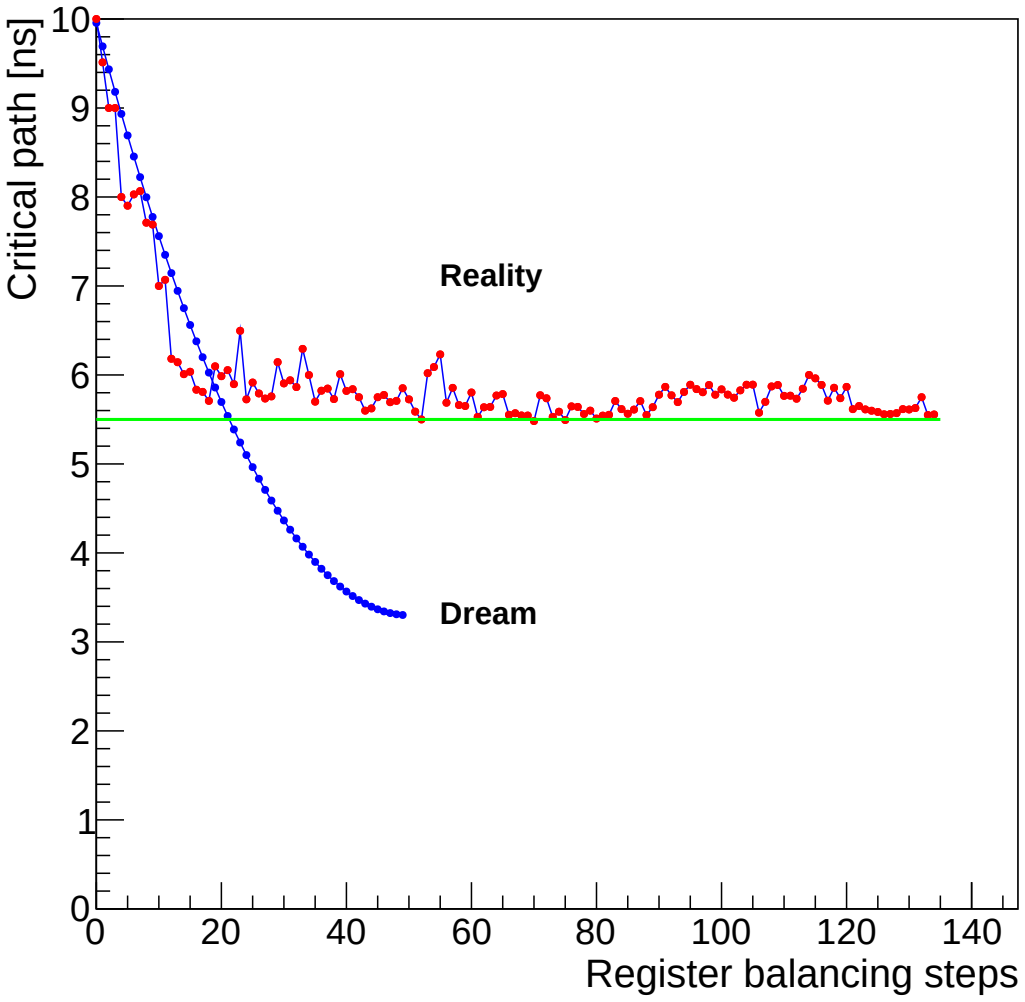

Figure 5.2: Register balancing steps of SHP-ed RV32iMC on Artix-7.

message passing (MP) block and a GPIO block. The cores can communicate with each other using message passing. They have equal rights to access the GPIO. Only core 0 can communicate with the UART. The UART (USB interface) is also used to download program (or data) and to readback the data-RAM of all cores.

5.1 Clocking and performance

The system runs at 180 MHz. A micro-cycle takes therefore 5.55 nsec, a macro-cycle a minimum of 22.22 nsec. A thread runs at 45 MHz when less or equal than 4 threads are active at a time. When all 16 threads of a core are active, all threads run at 11.25 MHz.

For performance evaluation is based on the famous CHStone testcases `http:/www.ertl.jp/chstone/`. They are used in the processor and the high level synthesis (HLS) domain in many papers. I explicitly use the GSM, ASM, Motion, ADCPM, SHA and Blowfish testcase. When running these testcases stand-alone an IPC (instructions per cycle) of 0,86 is reached, which results in 155 MIPS (million instructions per second) per core. When projected on the complete system 620 MIPS are reached.

Figure 5.2 shows the performance increase during register balancing. Register balancing is the process to move the additional registers (which are added to the design during the C-Slow Retiming step) through the design to achieve the highest possible performance. The goal based on the early estimations was to reach 3.3 nsec, which would have resulted in a 300MHz micro-cycle performance, 1.2 GHz clock performance overall and 1.05 GIPS (Giga instructions per cycle). Unfortunately, as Figure 5.2 shows, I run into a virtual place and route barrier at 5.55 nsec. I was way more successful with this method when doing it on a Thumb-2 based design with a "longer" critical starting path.

5.2 Reset

After power up or when the external reset button on the ARTY board was active, the complete design is in reset mode. Each core has its own reset flag. These reset flags can only be deactivated by the UART/USB interface, by writing the right value to the right address. The same interface can also activate the individual flags again. Please refer to Section "Download bit and hex files to the ARTY board".

Once the reset of a core is disabled then the initial thread of that core becomes active. This active thread after reset starts at address 0x0000.

5.3 RISC-V, RV32iMC, CUBE-V-RV32iMC-P3C4D16

The project uses a RV32iMC implementation of the RISC-V. It is called "CUBE-V". It does not support the FENCE, ECALL and EBREAK instructions and it is therefore not fully RV32I compatible, which I indicate by using the lower case "i" in RV32iMC.

CUBE-V also does not support the control and status registers "CSR".

A CUBE-V has 3 pipeline stages ("P3").

The complete design runs at 180MHz. This clock generates what we call a micro-cycle.

We applied C-Slow-Retiming generating 4 copies of each CUBE-V core ("C4"). It therefore takes 4 micro-cycles to finish one functional cycle, or in other words one macro-cycle is equal to 4 micro-cycles.

We then apply the final SHP-step, using a memory depth of 16 lines ("D16") to be able to run up to 16 threads at the same time on each core. We define, that a program (also called a thread) is executed at macro-cycle speed.

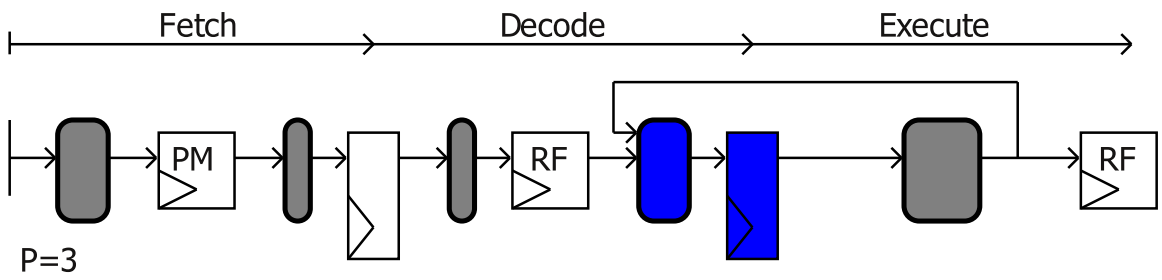

Figure 5.3: Pipeline stages and register file write-through short cut.

Unlike other high speed RV32 implementations, almost all instructions are executed in one macro-cycle. The exceptions are:

DIV[U] and REM[U]: 33 macro-cycles (might be optimized)
MULH[[S]U] 64-bit: 2 macro-cycles
Data memory read: 2 macro-cycles
GPIO register read: 8 micro-cycles

A register file (RF) write-through policy is implemented. This means, when a register dependency is detected, then the register value is not only written into the RF, but also at the output of the RF, so that the data can be used already in the next cycle (Figure 5.3). There are only a few exceptions to this rule.

The number of active threads can be changed dynamically. When less or equal 4 threads are executed, then each thread runs at a macro-cycle speed of (180MHz / 4 =) 45 MHz. When more than 4 threads are active, let's say n, then each thread runs at a macro-cycle speed of 180MHz / n.

5.4 On-chip memory

Each RV32iMC core has it own 6144x32 bits (24 KB) of program memory and 1024x32 bits (4 KB) of data memory. This can certainly be a limitation. In a next version, the number of cores can be reduced, which leaves more memory to individual cores. The external memory DDR3L on the ATRY board might be connected in one of the next versions at the cost of at least one CUBE-V.

It should be mentioned, that each thread on a core shares the same memory. So a peripheral driver code is shared among all virtual peripherals for instance. It is important to know that each thread also shares the same stack, which can be problematic, unless the stack pointer is taken care of.

5.5 Thread controller (TC)

Each core is supported by an individual thread controller (TC). A thread can be initialized by one of the sources listed in Table 5.1, which also lists the overall execution priority.

Source	Priority
Write to TC_START, TC_SAK	6
Calendar	5
GPIO	4
MP	3
UART RX	2
UART TX	1
Thread FIFO	0

Table 5.1: Thread initialization sources and thread priorities.

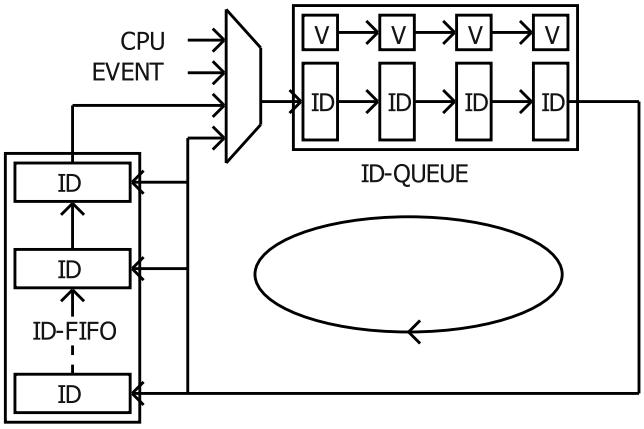

Figure 5.4: Abstract view of the thread controller mechanism.

A thread is started by writing its program start address to the **TC_START** or **TC_SAK** (TC start and kill) register. An individual thread ID is automatically applied to a thread when started. So far the thread ID wasn't relevant for programming and up to now it cannot be read. There is no exception called when more than 16 threads are initialized. Instead incoming events are stalled, until they can be served. Nevertheless, still a deadlock situation can occur when the system is poorly programmed.

Figure 5.4 shows that once a thread (with a certain thread identification number "ID" and valid flag "V") leaves the hyper-pipelined execution mechanism, it is reinserted into the execution mechanism again by default. When another thread needs to be initialized at that very cycle (by the CPU or a peripheral event) or the so-called ID-FIFO contains thread(s) waiting to be executed, then the thread itself is "parked" in the ID-FIFO instead. In the case when multiple threads a waiting to be executed, the thread which has the highest priority listed in Table 5.1 is executed. The priority scheme is fixed and cannot be modified (as of now). Events are stalled when threads with higher priorities are served instead.

A thread kills itself by writing to its **TC_SAK** register or by writing (any value) to its **TC_KILL** register (Table 5.2).

Threads can be initialized by writing into the **TC_START** or **TC_SAK** register the new thread's start address (Table 5.2). When a new thread is initialize by hardware this start address must be programmed ahead of time into the relevant peripheral's register. An example procedure is

			31 14	13 0
TC_START	0x80000000	w	don't care	thread start address [14:1]
TC_KILL	0x80000004	w	don't care	don't care
TC_SAK	0x80000008	w	don't care	thread start address [14:1]
			18	14

Table 5.2: Registers: **TC_START**, **TC_KILL**, **TC_SAK**.

described in more detail in the "Event handling and virtual peripherals" Chapter.

5.6 Calendar (CA)

Each SHP-ed RV32iMC core is supported by an individual calendar (CA). The CA is equivalent to a basic timer but can hold up to 240 entries. It can be written just like any other peripheral and stores a timely ordered list of future events. The CA can access GPIO registers or it can initiate a new thread once the time counter reaches the timestamp of the first event in the list.

A CA entry is a combination of a calendar command **CA_COM** and an event time **CA_ET**. The command register **CA_COM** must be programmed first. It is thread specific, which means that each thread has its own **CA_COM** register. It cannot be overwritten by another thread.

				31 29	28 12	11 8	7 0
CA_COM	0x80001000	w	command	code	not used	port	pin
				3	17	4	8
			clear bank	000		port number	pin[7:0]
			set bank	001		port number	pin[7:0]
			clear output	010		port number	pin[7:0]
			set output	011		port number	pin[7:0]

Table 5.3: Register: CA Commands (**CA_COM**).

				31 30	29 14	13 0
CA_ET	0x80001004	w	command	code	a0_value [15:0]	thread start address [14:1]
				2	16	14
			no a0 load	10		start address
			with a0 load	11	a0_value	start address

Table 5.4: Register: CA Event Time (**CA_ET**).

Writing to the **CA_ET** starts a mechanism, which picks up the thread specific **CA_COM** value, combines it with the **CA_ET** value and inserts this new entry in a timely ordered linked list. If a second **CA_ET** write follows, the last **CA_COM** entry of that thread is used again. Writing to

the **CA_ET** takes one cycle for the writing thread, but stalls any other thread which tries to write to the CA after that, until the event is inserted in the timely ordered linked list.

The user can read the value of the freely running 23-bit wide current timer register **CA_CT** at any given time (see Table 5.5). A cycle takes (1 / 180 MHz =) 5.55nsec. The timer starts at 0 again after an overflow. In order to handle the overflow problematic, the following constraint is defined. The user may only program an event which is $2^{22\text{-}1}$ (= 1f.ffffh = 2.097.151d) cycles ahead of the current time. At the same time, the user does not need to take care of the overflow and can write a low 23-bit wide value, which will be considered as an event after the next overflow. This period is equivalent to 11.6 msec when running at a frequency of 180 MHz. Longer periods must be handled by software counters and multiple events.

The associated command is issued, once the **CA_CT** timer matches any of the programmed event time. This command execution takes 5 cycles and is executed immediately. When multiple events should occur at the same time, they are handled sequentially with 8 cycle delay time. It is therefore recommended to assign an offset to individual event groups, in order to reduce this late command execution, in case a very precise timing is required.

			31 24	22 0
CA_CT	0x80001008	r	0	current time
			9	23

Table 5.5: Register: CA Current Time (**CA_CT**).

Another delay of an event execution can occur, when an entry is inserted into the linked list and the algorithm just checks the very first entry of this list. The command execution is disabled during these 7 cycles. For the remaining time of the entry insertion period, one match can be handled.

The CA looks back $2^{12\text{-}1}$ (= fffh = 4095d) cycles to catch up with missed matches.

A command can access GPIO registers or can initiate a thread.

A GPIO command can clear and set individual GPIO port directions and output values (see Table 5.3).

A CA can also be programmed to directly start a thread at a programmable thread start address. It can be defined, whether a data value of 16 bits is copied over into the register file at address 10d (register name a0) or not (see Table 5.4).

5.7 Message Passing (MP)

The message passing feature is implemented as a simple 32 bit wide data transfer from a transmitting core tx to a receiving core rx. All permutations with tx = 0, 1, 2, 3 and rx = 0, 1, 2, 3 are valid except for data transfer within a single core. The relevant registers are listed in Table 5.6 and 5.7.

The MP peripheral can transfer data in two modes. Either the receiving data is read when valid,

or the tranmitting core initiates a thread in the receiving core.

For a data transfer from tx to rx, the transmitting core tx writes the 32-bit wide data to its message passing out-going register **MP_OUT_[rx]**. This data is then transferred to the message passing in-coming register of the receiving core **MP_IN_[tx]**. The transmitting core tx stalls until the message is read from the message receiving core.

The MP peripheral can be programmed to initiate a thread at the receiving core, once a new message is valid. To enable this mechanism, bit 14 must be set in the **MP_COM_[rx]** register, and the thread start address must be programmed as well (see Table 5.7). Still, the transmitting core stalls until the receiving register **MP_IN_[rx]** is read.

			31 0
MP_OUT_0	0x80040000	w	send data
MP_OUT_1	0x80040004	w	send data
MP_OUT_2	0x80040008	w	send data
MP_OUT_3	0x8004000c	w	send data
MP_IN_0	0x80040010	r	receive data
MP_IN_1	0x80040014	r	receive data
MP_IN_2	0x80040018	r	receive data
MP_IN_3	0x8004001c	r	receive data

Table 5.6: Register: Message passing out and in (**MP_OUT_N, MP_IN_N**).

			14 14	13 0
MP_COM_0	0x80040020	w	thread enable bit	thread start address [14:1]
MP_COM_1	0x80040024	w	thread enable bit	thread start address [14:1]
MP_COM_2	0x80040028	w	thread enable bit	thread start address [14:1]
MP_COM_3	0x8004002c	w	thread enable bit	thread start address [14:1]

Table 5.7: Register: Message passing communication control (**MP_COM_N**).

5.8 GPIO

The GPIO peripheral is accessible by all cores. It supports 14 banks with 8 pins each and therefore 112 pins. Its register set is optimized towards the connector bundles (8-bits) of the ARTY board. The direction can be set (output) or cleared (input) for each pin using the **GPIO_N_DIR_SET** or the **GPIO_N_DIR_CLR** register respectively (see Table 5.8). N stands for the bank number. The setting of the bit in the **GPIO_N_OUT_SET** (**GPIO_N_OUT_CLR**) register makes an output pin switch to 3.3V (ground).

The value of each GPIO pin can be read by accessing the **GPIO_N_IN** register. The input signal passes always through a filter first. It uses a 3 bit-wide shift register running at 180MHz. A majority decoder extracts the actual (filtered) signal value.

Each GPIO pin of the first 9 banks can be programmed to be level (and therefore edge) sensitive.

	offset		31 8	7 0
GPIO_N_DIR_CLR	0x0000	w	don't care	pin[7:0]
GPIO_N_DIR_SET	0x0004	w	don't care	pin[7:0]
GPIO_N_OUT_CLR	0x0010	w	don't care	pin[7:0]
GPIO_N_OUT_SET	0x0014	w	don't care	pin[7:0]
GPIO_N_IN	0x0020	r	0	pin[7:0]
GPIO_N_LVL0	0x0030	w	don't care	pin[7:0]
GPIO_N_LVL1	0x0034	w	don't care	pin[7:0]
GPIO_N_CAP	0x0040	w	don't care	pin[7:0]
			24	8

Table 5.8: Registers: **GPIO_N_DIR_CLR, GPIO_N_DIR_SET, GPIO_N_OUT_CLR, GPIO_N_OUT_SET, GPIO_N_IN, GPIO_N_LVL0, GPIO_N_LVL1, GPIO_N_CAP**

Setting a bit in the **GPIO_N_LVL0** (**GPIO_N_LVL1**) registers programs the input logic of the relevant bit to be level sensitive to low (high). The level sensitive logic uses the filtered input signal only.

Once the relevant signal matches the programmed input level (zero or one) the GPIO starts an internal event handling mechanism. When the level matches the programmed input level during programming already, then this event handling mechanism is started immediately. If not, it starts one cycle after the filtered input signal reaches the programmed level. It can be argued, that this mechanism is therefore (also) edge sensitive.

One task of the event handling mechanism is to capture the filtered input signal of the neighboring pin with the next higher index of the same bank. This value is then stored in the readable **GPIO_N_CAP** register at the bit location of the neighboring pin with the next higher index. If the level sensitive pin is at bit 7 of the bank, then the neighboring pin is bit 0 of the same bank.

Another task of the event handling mechanism is to start a thread at a predefined program address. A round robin arbiter mechanism takes care, that all events are propagated with the same priority and that no event is missed due to multiple consecutive events of some other pin(s).

8 cycles after the relevant filtered signal equals the defined level, the GPIO peripheral requests from the TC to initiate a thread at a programmable start address. (In other terms, 15 cycles (83 nsec) from an input edge to the first program fetch of the start address, which is very fast, considering the fact, that a filter and an arbiter is involved). Each core can program its core specific thread start address into the core specific **GPIO_EVENT_ADD** register (see Ttable 5.9). The event is propagated to only one single core. The core is selected, which was the last one to program the pin specific **GPIO_N_LVL0** or **GPIO_N_LVL1** registers. Also, all events for one core end up at the same thread starting address.

			31 14	13 0
GPIO_EVENT_ADD	0x80031000	w	don't care	thread start address [14:1]
			18	14

Table 5.9: Register: **GPIO_EVENT_ADD**

The program needs to know, which input pin started the particular event. For that, the global pin

index is propagated into the link register 10d (a0) of the core register file. The global pin index is the result of the bank number multiplier by 8, plus the pin location within the bank. In other words, it is a unique pin number.

Once an event has been handled and a thread is initialized, the pin specific level sensing mechanism needs to be reprogrammed. The **GPIO_EVENT_ADD** does not need to be reprogrammed.

Due to area limitations of the FPGA, the GPIOs supporting incoming event propagation is scaled down to 72 pins (first 9 banks). The upper banks are connected to LEDs and switches anyway.

The base address of the GPIO block is 0x80031000.

The address step to the next bank addressing is 0x100.

For more information please see the "Register map" Chapter.

5.9 UART

The implemented UART is used for communication with the external FTDI USB chip at a fixed baudrate of 5 MBaud. The system can therefore communicate with a PC (for example) via USB. The UART peripheral communicates internally only with core 0. Nevertheless, program and data memories of all cores can be written and data memories of all cores can be read via this PC-link as well. Please see Chapter "Download bit and hex files to the ARTY board" for more information.

The UART uses 3 registers for controlling the sending mechanism (**UART_SEND, UART_SEND_STAT, UART_TX_COM**) and 3 registers for the receiving part (**UART_REC, UART_REC_STAT, UART_RX_COM**). They are listed in Table 5.10, 5.11 and 5.12.

Both directions can be executed in normal or advanced mode.

When a byte should be send via the UART to the FTDI chip in normal mode, then the data must be written into the **UART_SEND** register. When the UART is still busy sending the last byte, then the thread trying to write to the **UART_SEND** register is stalled until the sending of the last byte has finished. This allows a simple programming of a thread which serves as DMA engine. In case the stalling should be reduced, the send_busy flag in the **UART_SEND_STAT** register can be polled to check whether the sending block is still active.

The UART can be programmed to initiate a thread (advanced mode), once the sending of the data has finished. To enable this mechanism, the thread enable bit in the **UART_TX_COM** register must be set and the thread start address must be programmed.

The UART can be configured using the **UART_RX_COM** register to handle receiving bytes in normal mode or advanced mode. In normal mode (Bit 14 in **UART_RX_COM** is 0) the received value can be read using the **UART_REC** register. If the **UART_REC** is not valid or has been read already, then the reading thread is stalled until the value is valid (again). This allows a simple programming of a thread which serves as DMA engine. In case the stalling should be reduced, the receive_valid flag in the **UART_REC_STAT** register can be polled to check whether a received byte is valid.

Once the UART received a serial byte from the FTDI chip and the UART receiving part is set into advanced mode (Bit 14 in **UART_RX_COM** is 1), then a thread is initialized at core 0. The least significant 14 bits of the **UART_RX_COM** determine the start address. The received byte is valid in register a0 (register 10d) of the register file. The received value can also be read via the **UART_REC** register in advanced mode as well.

The receive value must be read via the **UART_REC** register or the relevant thread must be executed (received data is saved in a0), otherwise the data will be overwritten.

			31 8	7 0
UART_SEND	0x80020000	w	don't care	send byte
UART_REC	0x80020020	r	0	received byte
			24	8

Table 5.10: Register: **UART_SEND**, **UART_REC**

			31 1	0 0
UART_SEND_STAT	0x80020004	w	don't care	send_busy
UART_REC_STAT	0x80020024	r	0	receive_valid
			30	1

Table 5.11: Register: **UART_SEND_STAT**, **UART_REC_STAT**

			31 15	14 14	13 0
UART_TX_COM	0x80020010	w	don't care	thread enable bit	thread start address [14:1]
UART_RX_COM	0x80020030	w	don't care	thread enable bit	thread start address [14:1]
			17	1	14

Table 5.12: Register: **UART_TX_COM**, **UART_RX_COM**

Chapter 6

Implementation and simulation

The release comes with an initial Vivado 2019.2 project under Windows. When opening the $project/vivado/fpga_top/fpga_top.xpr project, it can be seen, that the project is not compiled yet. The project can be compiled by clicking on the "Generate Bitstream" link. It will roughly take 30 minutes or so. Alternatively, the precompiled "bit" file $project/bit/fpga_top.bit can be used for downloading.

The project has been developed using version 2017.2. After testing the release using various versions (2018.2, 2019.2) it turned out, that there are too many variations. 2018.2 gives me an overmapped (103% LUT) error for unknown reasons and 2019.2 didn't make the timing at that time. So the project is released using the latest version 2019.2 and the timing is clean now.

The design hierarchy can be seen in Fig. 6.1. The parameters are stored in MPSOC_parameter. The rest is pretty much self-explaining. The "_syn" memory files are used for synthesis. The memory files also have a bypass data register implemented to improve the RAMs write-through capabilities.

The "clk_100M" module converts the incoming 100 MHz clock into an internal 180 MHz clock.

This Vivado project also has a top level simulation setup "tb_top". It simulates the complete FPGA but without the clk_100M module. It tests the UART connectivity, data and program memory write and read as well as reset functionality. It also downloads a program which writes and reads some UART data.

There are two additional Vivado simulation projects. One project "$project/vivado/simulation_CUBEV" simulates the CUBE-V core, by executing all instructions and by running some CHStone tests, the other project "$project/vivado/simulation_quad" tests 4 CUBE-V instantiations at the same time, their individual system features and how they interact with each other or with the GPIO peripheral.

All simulation so far are self-testing. There are additional demo-tests in the "simulation_quad" project, which demonstrate the usage of multithreaded drivers.

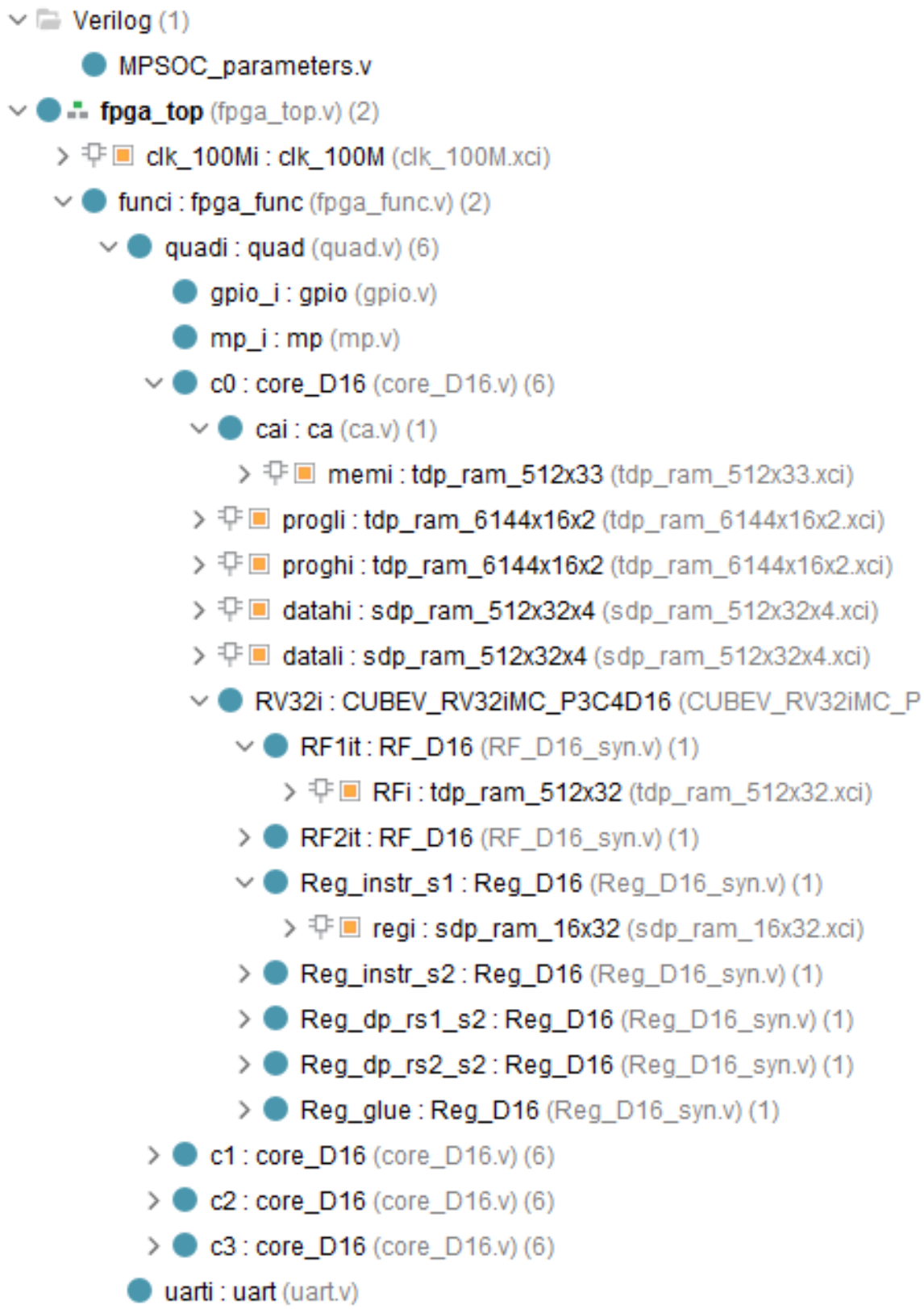

Figure 6.1: Overview of the design hierarchy.

Chapter 7

Event handling and virtual peripherals

In a standard microcontroller system, an interrupt interrupts the processing of the CPU, saves these register file values on the stack, which might be overwritten by the interrupt routine and continues with the interrupt routine. After that, the stack values are copied back into the relevant register file locations.

In this project's environment, an event starts a new thread and does not interrupt any of the running threads. In this context we denote that as event handling, instead of interrupt handling.

The question is, how do we identify the relevant start address after linking. This is the current proposal, based on the GPIO peripheral:

```
void gpio_event(unsigned tag, int start_time) {
  if (start_time >= 0) {
    GPIO_EVENT_ADD = (((unsigned)&&gpio_event_label >> 1) & 0x3fff);
  } else
  {
    gpio_event_label:
      TC_SAK = gpio_event_hash[tag];
  }
}
```

When calling the gpio_event function, the address of the gpio_even_label is stored in the **GPIO_EVENT_ADD** register of the GPIO peripheral. When the GPIO detects a relevant edge, it starts a thread at **GPIO_EVENT_ADD** register, which happens then to be the gpio_event_label.

A second trick is used here. The GPIO pin number is written in the a0 link register, which is identical to the "tag" entry. To cope with it, we have to build a gpio event hash ahead of time. The line:

TC_SAK = gpio_event_hash[tag];

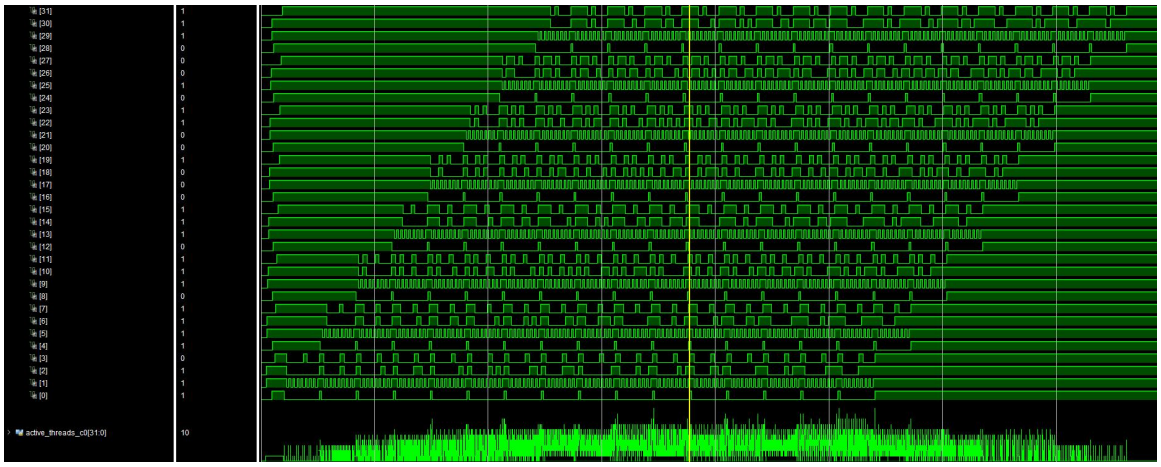

Figure 7.1: Running 8 SPI master and 8 SPI slave virtual peripherals from one core.

then looks up the start address of the particular pin handler and starts a thread there, while killing the current thread at the very same cycle.

This is the currently implemented alternative for interrupt handling. If you have any ideas or suggestions, please let me know.

With this scheme in mind, we can now program virtual peripherals. A protocol is partitioned into individual timepoints, at which a protocol must be handled by a thread. For the time period, at which nothing needs to be done, the threads kills itself after it has programed the calendar to wake up a thread at a certain program location, which then continues with handling the program. Figure 7.1 shows a demo simulation when running 8 SPI master and 8 SPI slave virtual peripherals on a single CUBE-V core. The bottom line shows the number of active threads, which is 10 at its peek. This is just one possibility to code virtual peripherals.

The current release contains initial versions of RS232 (115kBaud), I2C (50kHz) and SPI (25kHz) drivers. This is more of a prove of concept and is optimized to run 16 (15 in case of RS232) individual virtual peripherals at the same time on one core. The following improvements can be made in the future:

1) The provided drivers can be code-size and timing optimized.

2) When equal or less than 4 threads (VPs) are running on a core, or when an exact number of threads (VPs) are running on a core, then the runtime of a thread is predictive, and the VPs could continuously execute the code (without being interrupted). This leads to higher baudrates.

3) When the design needs to support the master side of the protocol, multiple VPs can be executed in a parallel or sequential fashion. Parallel could mean for instance to extend the data width of an SPI to 5 bits to support 5 individual 1-bit SPIs. Sequential execution could mean, that in case the protocol does not need to be continuously driven, the serving of individual (and different) interfaces could alter accordingly.

Certainly one of the great benefits of such a software driven protocol is, that many custom "or strange" protocols variations are possible.

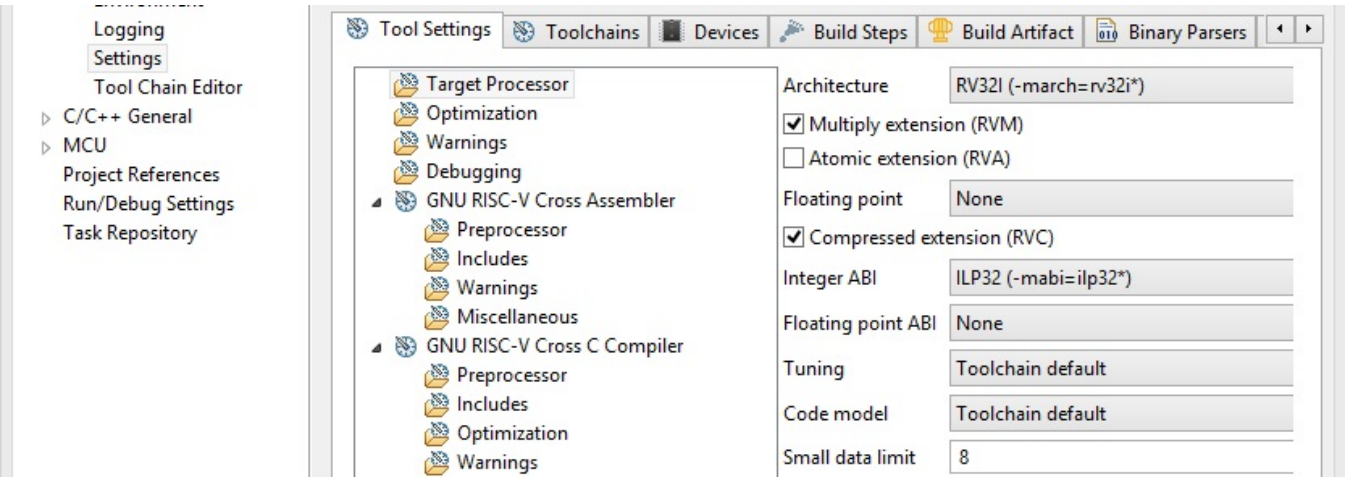

Figure 8.1: Compiler settings.

Chapter 8

Software

As the project's name might indicate, the programming of the system can be done using processing in the Arduino-style. The Arduissimo IDE is not part of this release but it is planned to have this included in version v0.2.

As of now the system can be programmed using the Freedom Studio framework from SiFive. Figure 8.1 shows the recommended compiler settings.

Each thread on a core shares the same program memory and the same data memory. This is why the provided examples are compiled as a single program. It is important to know that all threads potentionally use the same stack, which can be problematic, unless the stack pointer is taken care of.

In the Freedom Studio projects that come with this release, the project's name postfix ("_0", ..., "_3") indicate, on which core the code is executed (see Figure 8.2). Compiling the projects always results in the error message, that "elf" execution fails. The "elf" to "hex" conversion is then done seperately by executing the relevant Makefile command. The right command in the Makefile copies

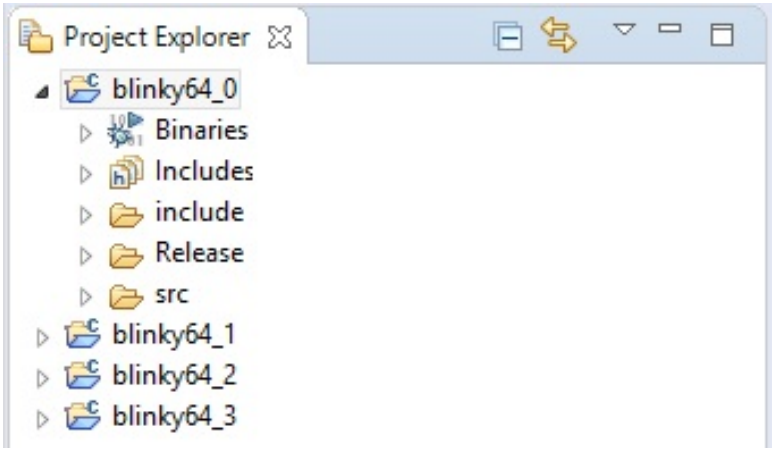

Figure 8.2: Project's postfix indicate individual core.

the relevant "elf" file into a "work" directory, and from there it is converted into "hex". This example is self explaining:

```
gsm_32 :
   cd ../tests/chstone/gsm/work ;
   cp ../gsm_ws_rv32imc/gsm/Release/gsm.elf . ;
   riscv64-unknown-elf-objdump.exe -S gsm.elf > main.disasm ;
   riscv64-unknown-elf-objcopy.exe -j.text -O verilog gsm.elf main.hex ;
   riscv64-unknown-elf-objcopy.exe -j.data -O verilog gsm.elf data.hex
```

In the next chapters, the downloading of a "hex" file to the ARTY board is discussed.

Chapter 9

Download bit and hex files to the ARTY board

The release comes with a precompiled "bit" file ($project/bit/fpga_top.bit), which can be used for downloading via the standalone LAB tool version from Xilinx, or by using the Hardware Manager in the Vivado GUI.

In order to understand the proposed "hex" file download process, it is best to look at the two examples in $project/ftdi/arty_ftdi/work/. When using cygwin, "make colors" and "make downloadHex" can be used to download the colors example "hex" files. The results will be reported once the download has finished. Alternatively the colors.bat command can be executed in a Windows command prompt. Here the downloading progress will be prompted directly.

In this initial version of the Arduissimo project, programming via the UART basically means:

1) to loopback a byte in order to test the USB - FTDI - FPGA(UART) link,
2) to control the reset status of the system,
3) to write to the individual program memories,
4) to write to and to read from the data memories of the individual cores and
5) to communicate with core 0.

As of now, no debugging capabilities are implemented (other than the data memory read feature).

In order to access the system via the UART, a byte or a sequence of bytes has to be written via the FTDI chip first. The first byte defines the access type:

Byte 0	Access type
0x1X	set/clear reset
0x20	loopback
0x30	memory write follows
0x40	memory read follows
0x50	user communication in write direction follows

Table 9.1: Access type resulting from byte 0.

A reset command sets or clears the internal system reset. The 4 LSB of that byte define the reset flag state of indivial core.

In loopback mode, the next byte is sent back via uart_tx_out.

When the memory write or memory read option is chosen, the following bytes define the data stream:

Byte	Meaning
1	bit [17:16] of memory start address
2	bit [15:8] of memory start address
3	bit [7:0] of memory start address
4	high byte of access length
5	low byte of access length

Table 9.2: Meaning of bytes 1 through 5.

The program and data memory offsets are as follows (from the UART perspective):

Core/Memory	Offset
core 0	0x00000000
core 1	0x00020000
core 2	0x00040000
core 3	0x00060000
program	0x00000000
data	0x00010000

Table 9.3: Meaning of bytes 1 through 5.

When the memory write option is chosen, the user must write the memory content according to the programmed access length. The received data is automatically replied. The user can use this feature to verify the transferred data.

In case of the memory readback option, the FPGA will send the requested memory content according to the programmed access length directly after the configuration stream has finished.

For setting or clearing reset flags and memory write or read access, the C program $project/ftdi/arty_ftdi/source/downloadHex.c provides the relevant routines to be used for interfacing. The arguments are :

-srb: Set reset at beginning.
-srb: Set reset at end.
-lb: Loopback test.
-dc: Download to core [0..3] the following ”hex” file.

When no argument is given, the following arguments will be used as default:

downloadHex -srb f -sre 0 -lb -dc 0 main_0.hex -dc 1 main_1.hex -dc 2 main_2.hex -dc 3 main_3.hex

When the user communication in write direction (PC -> FTDI -> FPGA) option is chosen, then the following bytes qualify the data stream. The user must write the user communication content

according to the programmed access length.

byte	meaning
1	high byte of transfer length
2	low byte of transfer length
[3...n]	write stream content

Table 9.4: Meaning of bytes 1 through n.

The user communication in read direction (FPGA -> FTDI -> PC) is initiated and defined by the program executed on core 0. The USB driver running on the PC must be ready and capable to handle the upcoming data stream. As of now, no programming example is provided.

Appendix A

Register map

This section gives an overview of the register map. The same information is provided in the $project/sw/include/device.h file.

These core specific register maps apply to each core 0,..,3:

Register Name	Address
TC_START	0x80000000
TC_KILL	0x80000004
TC_SAK	0x80000008
CA_COM	0x80001000
CA_ET	0x80001004
CA_CT	0x80001008
GPIO_EVENT_ADD	0x80031000

Table A.1: Core specific registers.

Core 0 can access the UART:

Register Name	Address
UART_SEND	0x80020000
UART_TX_COM	0x80020010
UART_REC	0x80020020
UART_RX_COM	0x80020030

Table A.2: UART registers accessable by core 0 only.

The following registers can be accessed by any core related to its identifier:

Register Name	Address
MP_OUT_0	0x80040000
MP_OUT_1	0x80040004
MP_OUT_2	0x80040008
MP_OUT_3	0x8004000c
MP_IN_0	0x80040010
MP_IN_1	0x80040014
MP_IN_2	0x80040018
MP_IN_3	0x8004001c
MP_COM_0	0x80040020
MP_COM_1	0x80040024
MP_COM_2	0x80040028
MP_COM_3	0x8004002c

Table A.3: Message passing registers.

The following GPIO registers can be accessed by any core:

Register Name	Address
GPIO_0_DIR_CLR	0x80030000
GPIO_0_DIR_SET	0x80030004
GPIO_0_OUT_CLR	0x80030010
GPIO_0_OUT_SET	0x80030014
GPIO_0_IN	0x80030020
GPIO_0_LVL0	0x80030030
GPIO_0_LVL1	0x80030034
GPIO_0_CAP	0x80030040
GPIO_1_DIR_CLR	0x80030100
GPIO_1_DIR_SET	0x80030104
GPIO_1_OUT_CLR	0x80030110
GPIO_1_OUT_SET	0x80030114
GPIO_1_IN	0x80030120
GPIO_1_LVL0	0x80030130
GPIO_1_LVL1	0x80030134
GPIO_1_CAP	0x80030140
GPIO_2_DIR_CLR	0x80030200
GPIO_2_DIR_SET	0x80030204
GPIO_2_OUT_CLR	0x80030210
GPIO_2_OUT_SET	0x80030214
GPIO_2_IN	0x80030220
GPIO_2_LVL0	0x80030230
GPIO_2_LVL1	0x80030234
GPIO_3_DIR_CLR	0x80030300
GPIO_3_DIR_SET	0x80030304
GPIO_3_OUT_CLR	0x80030310
GPIO_3_OUT_SET	0x80030314
GPIO_3_IN	0x80030320
GPIO_3_LVL0	0x80030330
GPIO_3_LVL1	0x80030334
GPIO_4_DIR_CLR	0x80030400
GPIO_4_DIR_SET	0x80030404
GPIO_4_OUT_CLR	0x80030410
GPIO_4_OUT_SET	0x80030414
GPIO_4_IN	0x80030420
GPIO_4_LVL0	0x80030430
GPIO_4_LVL1	0x80030434

Table A.4: GPIO registers part I.

Register Name	Address
GPIO_5_DIR_CLR	0x80030500
GPIO_5_DIR_SET	0x80030504
GPIO_5_OUT_CLR	0x80030510
GPIO_5_OUT_SET	0x80030514
GPIO_5_IN	0x80030520
GPIO_5_LVL0	0x80030530
GPIO_5_LVL1	0x80030534
GPIO_6_DIR_CLR	0x80030600
GPIO_6_DIR_SET	0x80030604
GPIO_6_OUT_CLR	0x80030610
GPIO_6_OUT_SET	0x80030614
GPIO_6_IN	0x80030620
GPIO_6_LVL0	0x80030630
GPIO_6_LVL1	0x80030634
GPIO_7_DIR_CLR	0x80030700
GPIO_7_DIR_SET	0x80030704
GPIO_7_OUT_CLR	0x80030710
GPIO_7_OUT_SET	0x80030714
GPIO_7_IN	0x80030720
GPIO_7_LVL0	0x80030730
GPIO_7_LVL1	0x80030734
GPIO_8_DIR_CLR	0x80030800
GPIO_8_DIR_SET	0x80030804
GPIO_8_OUT_CLR	0x80030810
GPIO_8_OUT_SET	0x80030814
GPIO_8_IN	0x80030820
GPIO_8_LVL0	0x80030830
GPIO_8_LVL1	0x80030834
GPIO_9_DIR_CLR	0x80030900
GPIO_9_DIR_SET	0x80030904
GPIO_9_OUT_CLR	0x80030910
GPIO_9_OUT_SET	0x80030914
GPIO_9_IN	0x80030920
GPIO_10_DIR_CLR	0x80030A00
GPIO_10_DIR_SET	0x80030A04
GPIO_10_OUT_CLR	0x80030A10
GPIO_10_OUT_SET	0x80030A14
GPIO_10_IN	0x80030A20
GPIO_11_DIR_CLR	0x80030B00
GPIO_11_DIR_SET	0x80030B04
GPIO_11_OUT_CLR	0x80030B10
GPIO_11_OUT_SET	0x80030B14
GPIO_11_IN	0x80030B20
GPIO_12_DIR_CLR	0x80030C00
GPIO_12_DIR_SET	0x80030C04
GPIO_12_OUT_CLR	0x80030C10
GPIO_12_OUT_SET	0x80030C14
GPIO_12_IN	0x80030C20
GPIO_13_DIR_CLR	0x80030D00
GPIO_13_DIR_SET	0x80030D04
GPIO_13_OUT_CLR	0x80030D10
GPIO_13_OUT_SET	0x80030D14
GPIO_13_IN	0x80030D20

Table A.5: GPIO registers part II.

Appendix B

Pinning

Happy reading!

Pin Number	Bank Number	Register Bit	Board Naming	Schematic Naming	FPGA Pin
0	0	0	JA.1	JA1	G13
1	0	1	JA.2	JA2	B11
2	0	2	JA.3	JA3	A11
3	0	3	JA.4	JA4	D12
4	0	4	JA.7	JA7	D13
5	0	5	JA.8	JA8	B18
6	0	6	JA.9	JA9	A18
7	0	7	JA.10	JA10	K16
8	1	0	JB.1	JB1_P	E15
9	1	1	JB.2	JB1_N	E16
10	1	2	JB.3	JB2_P	D15
11	1	3	JB.4	JB2_N	C15
12	1	4	JB.5	JB3_P	J17
13	1	5	JB.6	JB3_N	J18
14	1	6	JB.7	JB4_P	K15
15	1	7	JB.8	JB4_N	J15
16	2	0	JC.1	JC1_P	U12
17	2	1	JC.2	JC1_N	V12
18	2	2	JC.3	JC2_P	V10
19	2	3	JC.4	JC2_N	V11
20	2	4	JC.5	JC3_P	U14
21	2	5	JC.6	JC3_N	V14
22	2	6	JC.7	JC4_P	T13
23	2	7	JC.8	JC4_N	U13

Table B.1: Pinning list part I.

Pin Number	Bank Number	Register Bit	Board Naming	Schematic Naming	FPGA Pin
24	3	0	JD.1	JD1	D4
25	3	1	JD.2	JD2	D3
26	3	2	JD.3	JD3	F4
27	3	3	JD.4	JD4	F3
28	3	4	JD.7	JD7	E2
29	3	5	JD.8	JD8	D2
30	3	6	JD.9	JD9	H2
31	3	7	JD.10	JD10	G2
32	4	0	IOL.1	CK_IO0	V15
33	4	1	IOL.2	CK_IO26	U11
34	4	2	IOL.3	CK_IO1	U16
35	4	3	IOL.4	CK_IO27	V16
36	4	4	IOL.5	CK_IO2	P14
37	4	5	IOL.6	CK_IO28	M13
38	4	6	IOL.7	CK_IO3	T11
39	4	7	IOL.8	CK_IO29	R10
40	5	0	IOL.9	CK_IO4	R12
41	5	1	IOL.10	CK_IO30	R11
42	5	2	IOL.11	CK_IO5	T14
43	5	3	IOL.12	CK_IO31	R13
44	5	4	IOL.13	CK_IO6	T15
45	5	5	IOL.14	CK_IO32	R15
46	5	6	IOL.15	CK_IO7	T16
47	5	7	IOL.16	CK_IO33	P15
48	6	0	IOH.1	CK_IO8	N15
49	6	1	IOH.2	CK_IO34	R16
50	6	2	IOH.3	CK_IO9	M16
51	6	3	IOH.4	CK_IO35	N16
52	6	4	IOH.5	CK_IO10	V17
53	6	5	IOH.6	CK_IO36	N14
54	6	6	IOH.7	CK_IO11	U18
55	6	7	IOH.8	CK_IO37	U17
56	7	0	IOH.9	CK_IO12	R17
57	7	1	IOH.10	CK_IO38	T18
58	7	2	IOH.11	CK_IO13	P17
59	7	3	IOH.12	CK_IO39	R18
60	7	4	DDR3_CKE0	CK_IOB	N5
61	7	5	IOH.14	CK_IO40	P18
62	7	6	IOH.15	CK_IOA	M17
63	7	7	IOH.16	CK_IO41	N17

Table B.2: Pinning list part II.

Pin Number	Bank Number	Register Bit	Board Naming	Schematic Naming	FPGA Pin
64	8	0	A1	CK_A0	F5
65	8	1	A2	CK_A6	B7
66	8	2	A3	CK_A1	D8
67	8	3	A4	CK_A7	B6
68	8	4	A5	CK_A2	C7
69	8	5	A6	CK_A8	E6
70	8	6	A7	CK_A3	E7
71	8	7	A8	CK_A9	E5
72	9	0	A9	CK_A4	D7
73	9	1	A10	CK_A10	A4
74	9	2	A11	CK_A5	D5
75	9	3	A12	CK_A11	A3
76	9	4	SCL	CK_SCL	L18
77	9	5	SCA	CK_SDA	M18

Table B.3: Pinning list part III.

Pin Number	Bank Number	Register Bit	Board Naming	Schematic Naming	FPGA Pin
78	9	6		SW0	A8
79	9	7		SW1	C11
80	10	0		SW2	C10
81	10	1		SW3	A10
82	10	2		BTN0	D9
83	10	3		BTN1	C9
84	10	4		BTN2	B9
85	10	5		BTN3	B8
86	10	6		LED0_B	E1
87	10	7		LED0_R	G6
88	11	0		LED0_G	F6
89	11	1		LED1_B	G4
90	11	2		LED1_R	G3
91	11	3		LED1_G	J4
92	11	4		LED2_B	H4
93	11	5		LED2_R	J3
94	11	6		LED2_G	J2
95	11	7		LED3_B	K2
96	12	0		LED3_R	K1
97	12	1		LED3_G	H6
98	12	2		LED4	H5
99	12	3		LED5	J5
100	12	4		LED6	T9
101	12	5		LED7	T10
102	12	6		CK_MISO	G1
103	12	7		CK_SCK	F1
104	13	0		CK_SS	C1
105	13	1		CK_MOSI	H1
106	13	2		QSPI_DQ0	K17
107	13	3		QSPI_DQ1	K18
108	13	4		QSPI_DQ2	L14
109	13	5		QSPI_DQ3	M14
110	13	6		QSPI_CS	L13
111	13	7		QSPI_SCK	L16

Table B.4: Pinning list part IV.