POLITECNICO DI TORINO

Master degree course in Embedded systems

Master Degree Thesis

Functional test of peripheral devices

Testing the Pulpissimo SPI module

Supervisors

prof. Matteo Sonza Reorda

dott. Riccardo Cantoro

Candidates

Riccardo Coggiola

matricola: 255306

Academic year 2018-2019

Contents

List of Figures

Abstract

The present work covers a software based test of the SPI peripheral included in the Pulpissimo SOC and, in particular, of its internal controller. Functional test of peripheral controllers can be a very difficult task, since they are very deeply embedded in the peripheral and therefore very difficult to access and their responses are hard to observe.

In the case of the Pulpissimo the task was not straightforward, as the built-in functions present in the C library didn't test the SPI controller, and therefore an ad hoc driver had to be written, in order to send arbitrary commands and get the finite state machine of the SPI controller to reach all possible states, thus achieving an adequate coverage. At first an evolutionary approach was tried: the initial test program was divided into several small tasks that were recombined using MicroGP in order to achieve higher coverage values. Sadly this approach didn't bring substantial results, as the controller coverage did not benefit from an increase in program size and number of tests.

Moving back to a manual approach, writing and optimizing a test program that adequately stimulates the controller eventually led to a satisfying result, since the global stuck-at fault coverage achieved for the entire SPI peripheral was 84%, with a 74% coverage on the controller module, and the test duration was around 500ms, whereas a regular test program written with the built-in functions would run for several seconds, reaching coverage results around 70% for the entire SPI peripheral, with no more than 23% fault coverage on the controller module.

Part I

First part

Chapter 1

Introduction: SPI protocol and its usages

1.1 The SPI protocol

SPI (short for Serial Peripheral Interface) is a synchronous serial communication protocol, very widely used at board level.
In its main usage it supports a single master (usually the microprocessor) and several slaves. It has separate wires for clock, slave selection, outgoing data (also called MOSI, Master Out Slave in) and incoming data (also known as MISO). The signalling mechanism for this protocol is an easy to implement *Non Return to Zero* (NRZ) and all actors are assumed to have an internal shift register controlled by a master generated clock, as well as two registers for transmitted (TX) and received (RX) data.
The data transmission from the master to the slave is carried by the MOSI pin, which has to be connected to the input registers of the slave. To do this, the internal shift register of the master is used to change the state of the pin together with the one of the slave registers.
When the slave has to send the response to the master, the opposite of the

MOSI transmission happens on the MISO pin: this time it is the slave turn to change the pin state and trigger the master input shift register.
On top of that, some *Chip Input Select* pins are necessary to choose which slave the message is directed to, and which one to get the answer from.
The following pictures show a block diagram of the connection of an SPI master with two slaves, along with the correspondent communication waveforms.

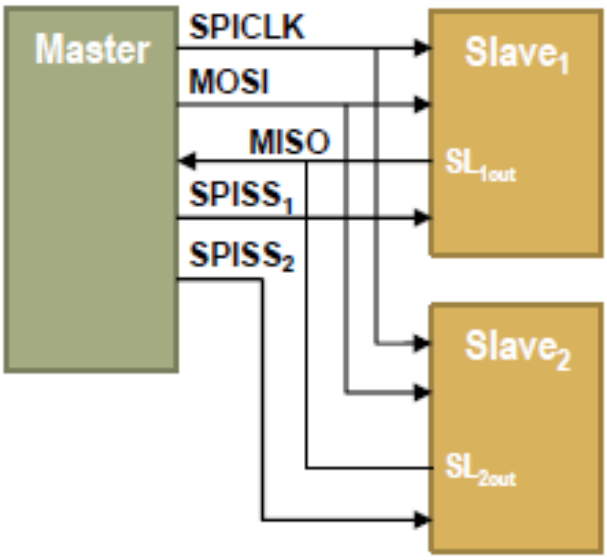

Figure 1.1. SPI block diagram - 1 master and two slaves [1]

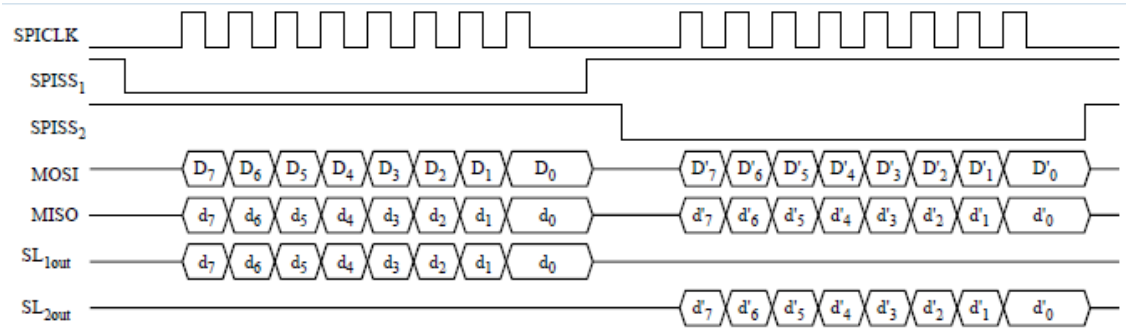

Figure 1.2. SPI communication waveform - 1 master and two slaves [1]

1.2 Communication example

Let us suppose that we have to send a n-bit data to the k-slave, where k is constrained in a range from 1 to i. First of all, we need to freeze the state of the other slaves so that we do not send the data to each of them: to do so, it is just required to pull down all the SPISS pins, except for the k-pin, which has to be in a logic *high* state.
Now that we have selected which slave the communication is directed to, the master has to fill the shift register with the data that is going to be sent out of the MOSI pin. For each bit that gets sent, one bit returns to the master through the MISO pin, like a continuous stream of bits, where the ones entering on the MOSI line push the others out the MISO one. In case the answer from the slave is not needed, the solution is for the slave to send back a *dummy word* which does not correspond to any action from the master, creating a one-direction communication.
One common use case where the MISO pin is awful is LCDs driving, where the slave has nothing to send back to the master.
An advantage over the I^2C protocol is that once the SPISS is set, it is possible to send multiple sequential blocks of data to the slave, like how it is done in memory write cycles, while using I^2C it is required to always send the register address first.

1.3 SPI modes and configurations

Since the SPI protocol depends on polarity and phase, it is possible to make it work in four different modes by just setting the two pins to high or low.
The polarity decides if the SCLK pin is needed to go high or low to start the communication, whilst the phase samples the rising or falling edge of this signal.

The result determines if the communication has to be started or ended. Regarding the possible configurations, there are three main topologies. The simplest is the *single-ended* one, which is composed by just the master and one slave. The *parallel* configuration is the most used one, which benefits from the presence of multiple SPISS pins, because SCLK, MOSI and MISO pins are shared among all the slaves. The less common one instead is the *daisy-chain* configuration, that is used when the less amount of connections possible is needed: in this case the MOSI signal travels through as many slaves as it is needed to get to the target one, then the bits to be received travel to the last MISO pin that finally gets back into the master. Transmission delay is one side effect which requires to have longer clock period to compensate for the extra travel time. [1]

1.4 Use cases

1.4.1 LCDs

LCDs driving is one of those applications where the I^2C protocol can be too slow if a high refresh rate is required, while the SPI protocol is a full duplex protocol type, granting at least double the transmission speed of the I^2C equivalent. Some real advantages in speed are experienced when instead of an LCD unit, a touch screen is connected: in this case being able to send information to the display, while retrieving data about the touch unit beneath can reduce delays and give a better user experience.

1.4.2 IO Expanders

As for the case of LCDs, multiple series connection of IO expanders can lead to some latency problems if it is also required to wait for a response using just one wire. Having two separated data lines allows for better expandability of

the system if we want to use just one master, without too many compromises.

1.4.3 Sequential read / write operations

Being able to read long data blocks quickly can be accomplished by choosing SPI over other protocols because the register selection for multiple sequential read or write cycle is required only for the first data block, then it can be omitted, or used only when there are other blocks in between that we do not want to write to or read from.

Chapter 2

Testing, fault model and software based self test

2.1 Testing: an increasing importance issue in electronics

Testing is a process aiming at identifying faulty products, without causing damage to the working ones.
Despite the production cost per transistor has always been decreasing since the earliest days, the testing cost per transistor has actually been increasing, and, to this date, the testing expenses are still increasing, driven by the increasing sophistication of modern electronic devices, made out of smaller and smaller components.
The progress of testing and production costs per transistor over the years are shown in the following graph.

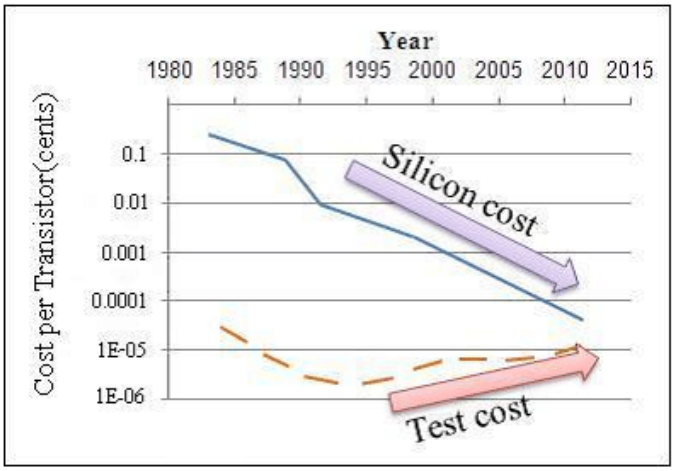

Figure 2.1. Test and production cost per transistor over time [3]

In the last years, mainly due to the increase in IoT devices and in the complexity of automotive electronics, testing peripheral has become a crucial point, in some cases even more important than the processor core itself.

2.2 The stuck-at fault model

Since the cost and difficulty of testing keeps increasing, it is almost impossible to simulate actual physical faults, and therefore *logic faults* are used, in order to easily simulate physical faults by means of a *fault model*.
As of now, the most commonly used fault model is still the *stuck-at* fault, basically consisting in an electric line being stuck to a logic value (0 or 1) and unable to change it, regardless of its drivers. In order to be useful, the stuck-at fault needs to be excited by creating a difference between the good circuit and the faulty one and then observed, which means propagating the consequences of the falut all the way to the primary outputs of the circuit, in order to cause a misbehavior, avoiding the fault to be masked.
This test model presents several advantages:

1. It is able to model several physical defects.

2. It is widely used and well documented in literature.

3. It is reasonably simple, thus having short simulation times (with respect to other fault models).

4. It is well supported by CAD tools.

5. It is often possible to collapse the fault list further reducing the test times, assuming that no timing information is considered.

The main problem with the stuck-at fault model is that it does not model all possible physical defects; for example, the interruption of the connection highlighted in the following figure cannot be modeled by means of a stuck-at fault. [4]

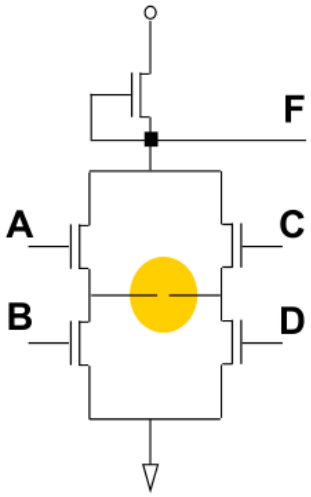

Figure 2.2. A line interruption cannot be modeled with a stuck-at fault [4]

2.3 Fault simulation and software based self test

The fault simulation is a process in which a software tool (called fault simulator) estimates the percentage of faults in a circuit that can be excited and observed by applying a certain input pattern to the primary input of the circuit.
Aside from a fault model, this process clearly needs an HDL description of the device under test (typically a gate level netlist, since this kind of code describes in detail the hardware structure of the device, whereas an RTL level description would not be as exhaustive) and a set of patterns to be applied for the simulation.
The patterns can be obtained in two ways:

1. The automatic test pattern generator (ATPG in short) is a software tool that, given the netlist of the DUT generates sets of pattern aiming to maximize its fault coverage. It takes a very long time for sequential circuits, since the insertion of memory elements implies that a fault may require several clock cycles before reaching the primary outputs. The pattern generated by an ATPG typically offer a very high coverage, but sometimes many faults identified by the ATPG are impossible to excite and/or observe during regular operation, and therefore functionally not testable.

2. in functional tests like the one covered in this work, it is possible to generate patterns in a pseudo-manual way, by making a processor execute a program and recording the inputs that are fed to the DUT along with the expected outputs. In this case the coverage is usually lower with respect to a set of patterns generated by an ATPG, but the tested faults tend to be the most likely to occur during the actual operation of the

device, and therefore they tend to be functionally testable.

This second approach is also called **Software based self test**, and even if it does not offer a very high coverage it is widespread since it doesn't need any hardware overhead and these kinds of self test can be performed autonomously by the device at startup or during idle times.

Chapter 3

The Pulpissimo project: hardware architecture and software support

3.1 Pulpissimo SoC

Pulpissimo is the most recent RISC-V distribution for PULP platform chips, developed by ETH Zurich and the University of Bologna.
Despite being, like its predecessor Pulpino, a single core processor it is still a rather innovative core, mainly for its set of peripherals, as it includes an SPI master, I2C, CPI (camera interface), I2S, UART and JTAG. All of these peripherals are connected to a DMA subsystem which, once programmed, handles the transfer operations autonomously, thus not stressing the RISC-V core itself.

The overall architecture of the SOC is reported in the following picture:

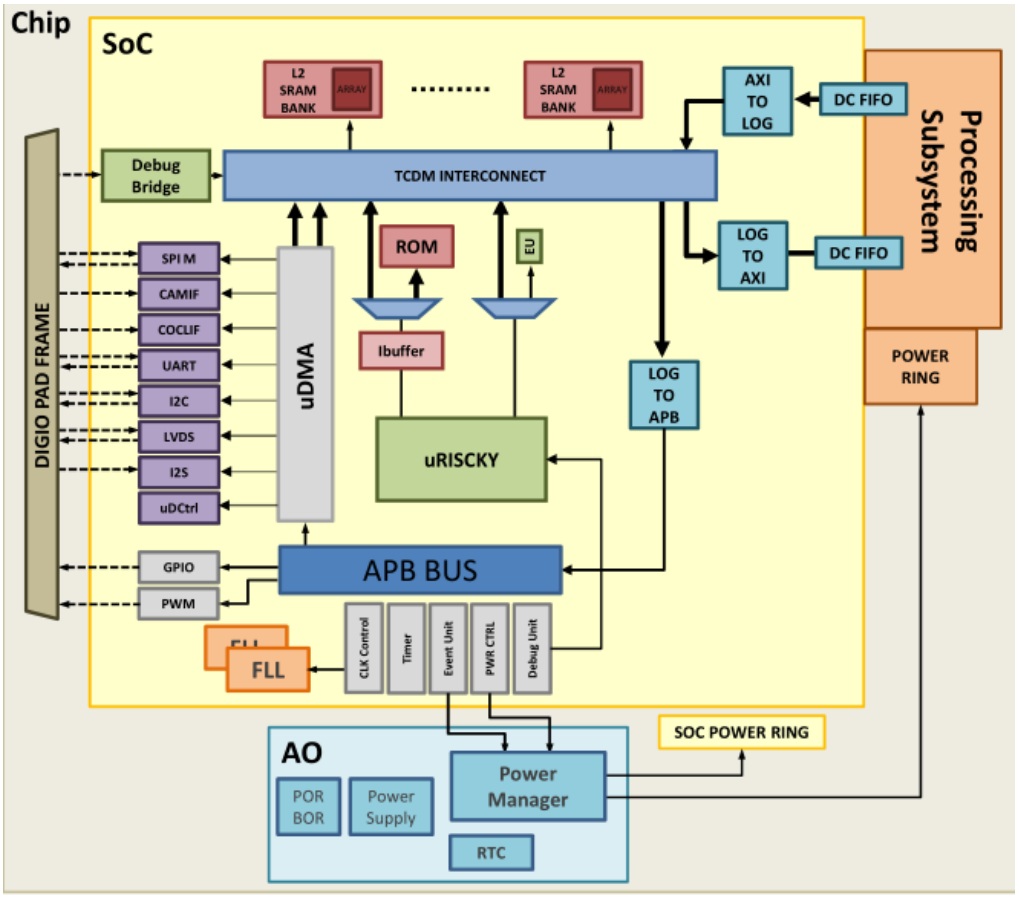

Figure 3.1. Architecture of the Pulpissimo SOC [2]

For the sake of power optimization, the system is divided into 3 different power domains: the always-on domain includes a power manager, a real-time clock and the wake-up logic, whereas the other two are switchable, the first contains a tiny CPU (called fabric controller), the peripheral subsystem, clock generators and main memory while the latter contains the processing subsystem. Both subsystems can be switched off and the clock generators can achieve very fast wakeup times.[2]

3.2 μDMA subsystem

A peripheral can have one or more data channels depending on its bandwidth requirements and bidirectional capabilities. Channels are mono-directional, hence a minimum of 2 channels is needed for a peripheral supporting both input and output. The architecture of a generic peripheral is shown in the picture below.

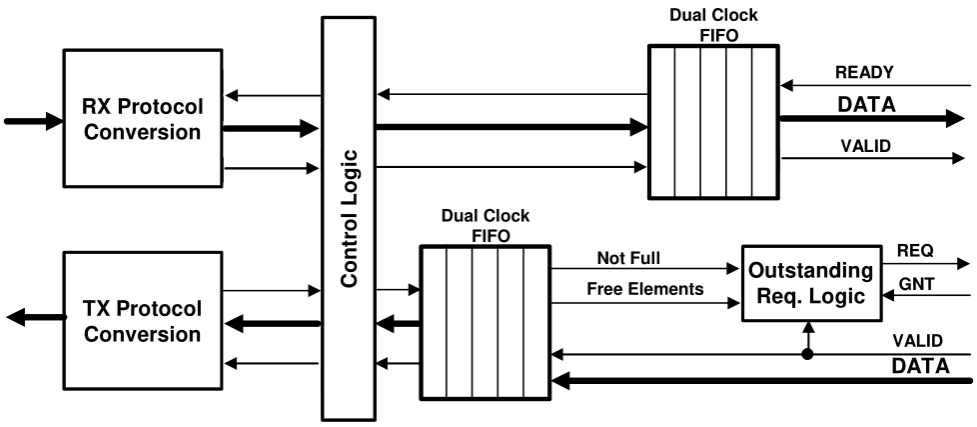

Figure 3.2. Architecture of generic peripheral [2]

The μDMA has 2 ports connecting the SOC interconnect directly to the interleaved memory, and it is therefore limited to access only the system memory, not allowing direct transfers to the processing subsystem nor to other peripherals connected on the APB bus. Ports towards memory are 32 bit wide, the supported bit widths are 8, 16, 32 which can be selected at programming time (for example UART and I2C use fixed 8 bit wide channels) or at runtime (SPI and I2S are configurable).
Starting a transaction requires only 3 accesses plus peripheral configuration, as the software needs to program the source or target pointer, the transfer length and send a start signal.[2]

3.3 RX channels

All RX channels share the same connections to the memory, therefore the μDMA subsystem needs to perform an arbitration between them: whenever a data is available to be transferred from the peripheral to the memory, peripherals rise the valid signal to notify the μDMA, which performs an arbitration and acknowledges the data transfer to the winning peripheral. The μDMA stores the ID of said channel, along with the data, and the data size of the channel; at the next cycle, the channel ID is used to select the channel resource (i.e. a set of information about the channel comprised of memory pointer, bytes remaining, status of pending transfers and channel enable) and the memory address for the transfer.
The μDMA logic is fully pipelined and capable of handling one transfer per clock cycle, provided that there are no contentions on memory banks.
The structure of the RX channel is shown, along with its handshake, in the following picture.

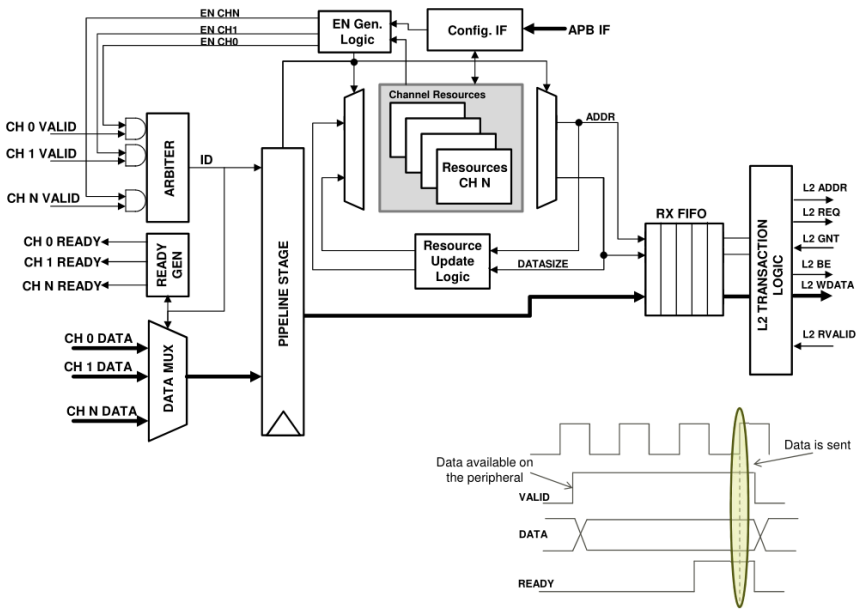

Figure 3.3. Achitecture of the RX channel and its handshake [2]

3.4 TX channels

Similarly to RX channels, all TX channels share the same memory access, however their handshake is more complicated with respect to the RX counterpart, as it has separate request and response paths. This split is necessary in order to support outstanding requests from high bandwidth peripherals, which do not need to wait for data in response while issuing subsequent requests.
Once the request from the peripheral has been granted by the μDMA, the ID of the winning channel is stored in a pipeline stage. In the next cycle the address and data size are fetched, while a new address and data size can be calculated and stored in the FIFO. At the FIFO output, the memory transaction logic pops an address + data size from the cycle and performs the memory access.
Figure 3.4 shows the TX channel architecture, along with its handshake waveforms.

In order to give an idea of the improvement achieved through this architecture it is worth mentioning that a single TX channel with outstanding request support can fully saturate the TX port, whereas a regular TX channel can only occupy a quarter of its bandwidth. This limit is introduced by the performance degradation due to the round-trip latency from request to response, which is completely avoided by allowing the peripheral to issue subsequent requests.
[2]

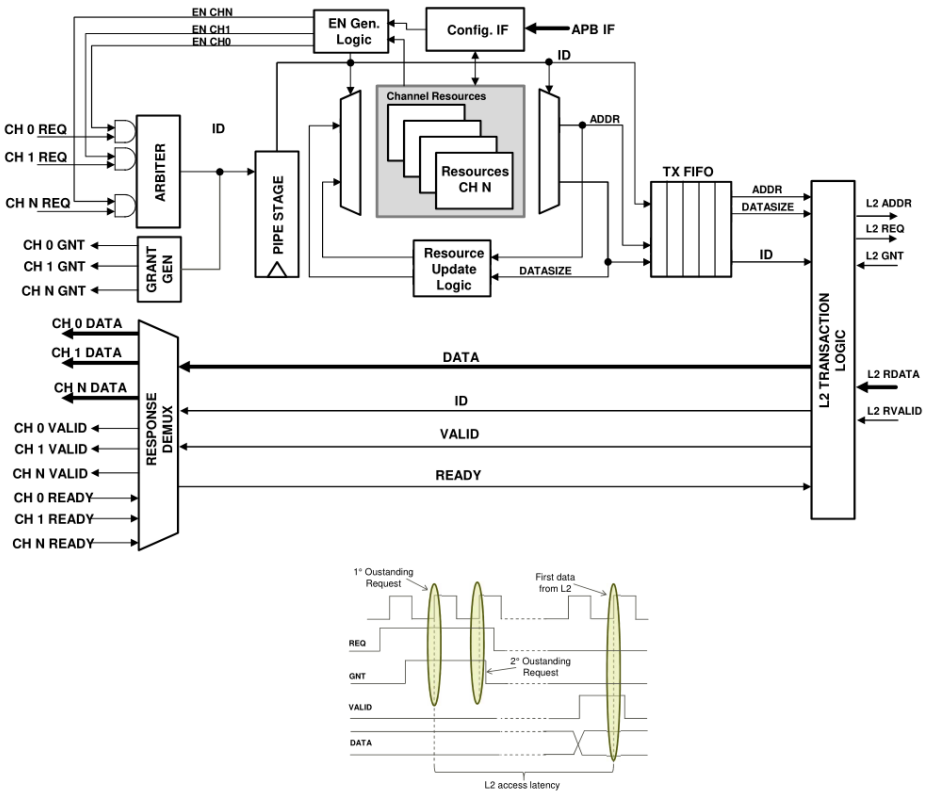

Figure 3.4. Architecture of the TX channel and its handshake [2]

3.5 Runtime software support

Communication is handled as in a classic microprocessor delegating tasks to the processing subsystem, containing services such as scheduling, memory allocations and driver.

3.5.1 Scheduler

The scheduler is a simple run-to-completion task scheduler with no preemption.

This allows using a single stack for all tasks, thus avoiding storing all saved

contexts in memory, which is crucial since the target of the application are ultra low power systems with consequently small memories.
The scheduler pops the first task from the stack, executes it until it returns and then turns to executing the following one; this operation is iterated until the stack is empty, in which case the scheduler enters a sleep mode. It is worth mentioning that a task can be enqueued while the previous task is executed, thus deferring some work.
Hardware events (the most common being the end of transfer event) are handled through interrupt service routines and can also enqueue task to handle events outside the interrupt service routine.

3.5.2 Driver interactions with scheduler

It is possible to attach a task to every asynchronous event, and said task will be enqueued or executed when the event occurs. This can be useful, for example, for re-enqueueing a transfer operation immediately after the previous one is completed.
The interrupt handler of the μDMA is called upon the end of a transfer, acknowledges the presence of a task attached to the transfer channel, enqueues it to the scheduler and leaves, later the scheduler will actually schedule the task (and enqueue another task at the end of it, if needed).
If the delay introduced by the scheduler constitutes an issue it can be eliminated by replacing the regular task with a handler, allowing the request to be enqueued directly by the interrupt handler.

3.5.3 Use case

The typical use case for this type of subsystem in a microcontroller unit is that in which data are sampled from a peripheral and sent to the system memory through some communication protocol (such as SPI) and sent to

the outside after some elaboration and processing from the microcontroller itself, once again using the μDMA.

Since the transfer happens asynchronously between peripheral and system memory, 2 buffers are allocated for each data transfer, one for transferring peripheral data to the level 2 memory, and the other for the processing subsystem.

Each time a transfer is finished on a channel, the interrupt handler handler enqueues a task to the cluster to allow the buffer processing to continue.

The following picture shows how the system resources are using during a data transfer and the processing the received data. [2]

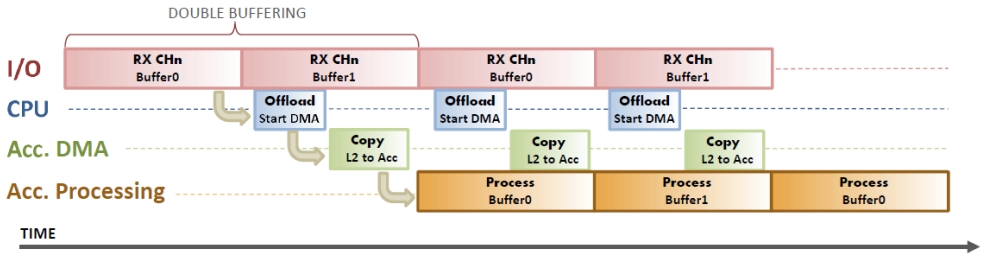

Figure 3.5. Resource allocation with double buffering mechanism [2]

Part II

Second part

Chapter 4

Testing of the Pulpissimo SPI module

4.1 Basic approach

The first approach adopted for testing the SPI module of the Pulpissimo SOC consisted in simply using the built-in functions from the library for sending and receiving data, hopefully stimulating (and therefore testing) the entire module.
The built-in functions performed the following operations:

- Sending data to a slave
- Receiving data from a slave
- Full duplex communication with a slave
- Switching from regular SPI to QSPI mode
- Configuring clock parameters such as phase, polarity and baudrate

This set of operations is sufficient to cover the regular operation of an SPI peripheral, guaranteeing a complete and rather configurable communication protocol, however, this method proved to be rather ineffective when it came to testing the controller module of the SPI, as it was impossible to obtain a fault coverage above 24% for said module.

4.1.1 Analysis of built-in functions

Taking a look at the provided SPI driver, we can see that these functions send a sequence of commands to the SPI controller, thus programming the SPI peripheral to perform the desired operation. The information concerning the kind of operation to be carried out is stored in the first 4 bits of the command, encoded according to the following table:

Binary	**Command**	**Explaination**
0000	CFG	Configure clock phase, polarity, baudrate and QSPI mode
0001	SOT	Signals the beginning of the data to be transferred
0010	SEND_CMD	Sends a command
0011	SEND_ADDR	Sends and address
0100	DUMMY	Sends a dummy (its value ranges from 0 to 31)
0101	WAIT	Waits for an event
0110	TX_DATA	Beginning of a TX operation
0111	RX_DATA	Beginning of a RX operation
1000	RPT	Beginning of the sequence to be repeated
1001	EOT	End of transmission
1010	RPT_END	End of the sequence of commands to be repeated
1011	RX_CHECK	Confronts the received data with a given vector
1100	FULL_DUPL	Beginning of a full duplex transfer operation
1101	WAIT_CYCLE	Sets a counter, starts it and waits until it has finished

The clock configuration operation only exerts a CFG command, whereas the other pre built functions issue commands the shown in the following subsections.

Send operation

1. CFG command for setting up the clock divider and the QSPI option
2. SOT command indicating the beginning of the actual transmission
3. TX_DATA command indicating the amount of data to be sent
4. A variable number of data bytes, indicated in the TX_DATA command
5. EOT command

Receive operation

1. CFG command for setting up the clock divider and the QSPI option
2. SOT command indicating the beginning of the actual transmission
3. RX_DATA command indicating the amount of data to be received
4. EOT command

Clearly in this case no data bytes are needed, since they are received from the slave.

Full duplex transfer operation

1. CFG command for setting up the clock divider and the QSPI option
2. SOT command indicating the beginning of the actual transmission
3. FULL_DUPL command indicating the amount of data to be transferred
4. A variable number of data bytes, indicated in the TX_DATA command
5. EOT command

It is clear that, even if these functions offer an almost complete set of possible operations and configurations, most commands are never issued, making it impossible to achieve an adequate coverage for the controller.

4.2 SPI controller architecture

The reason for this substantial inadequacy comes out even clearer by looking at the RTL level code of the `spi_ctrl` module: it is implemented as a finite state machine, whose main STG is shown in figure 1.

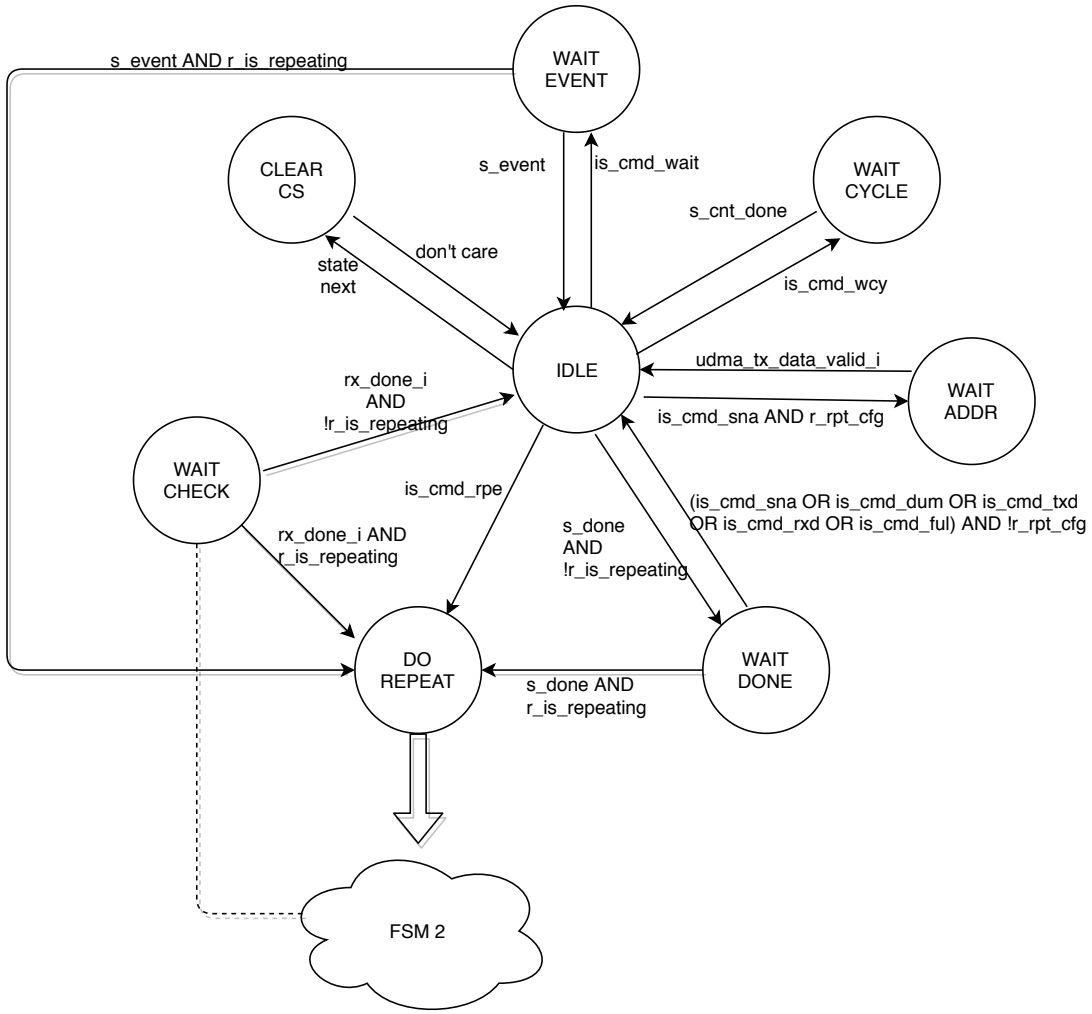

Figure 4.1. The STG of the main controller FSM

The part referred to as FSM2 in Figure is shown in the following picture:

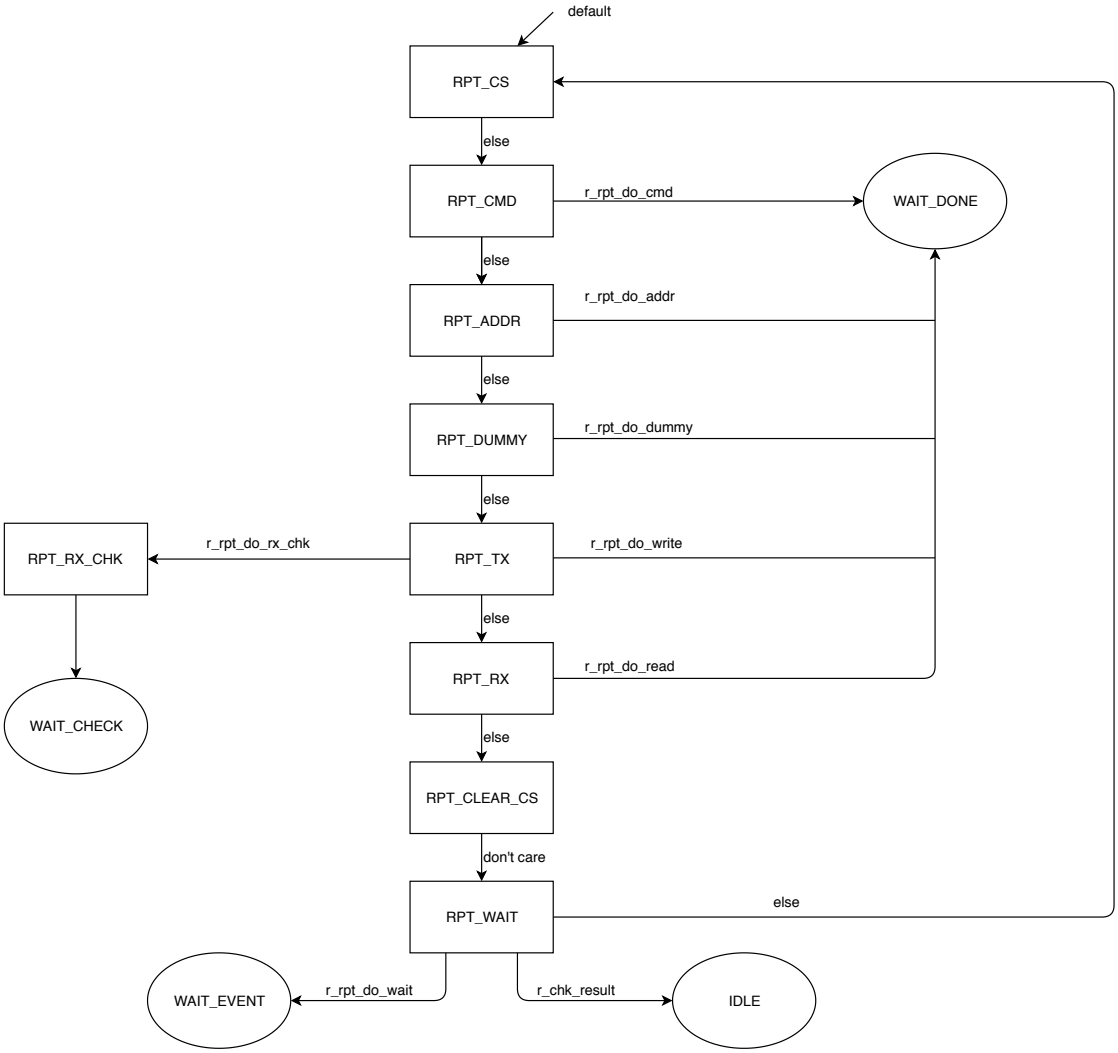

Figure 4.2. The STG of the second FSM in charge of dealing with repeated instructions

The fault list of the controller highlights that most possible faults concern the internal memory elements of the FSM, i.e. the registers storing the parameters used by the controller for its operation, such as:

- Register for RX check operation (16 bits).
- Register for repeated RX check operations (6 bits).
- Register for size of send operation (16 bits).
- Register for size of repeated read operation (18 bits).
- Register for size of receive operation (16 bits).
- Register for size of repeated write operation (18 bits).
- Register for repeated send address instructions (32 bits).
- Register for repeated send command instructions (32 bits).
- Clock divider register (8 bits).
- Register for number of repetitions (16 bits).
- Register for storing the type of operation being repeated, with one-hot encoding (8 bits).
- Counter registers (16 bits, 8 for counter state and 8 for counter target).

Moreover it was observed that while focusing on obtaining a higher coverage on said memory elements the other logic elements connecting them were satisfyingly tested.
Some registers were not exhaustively tested: for example the register containing the number of repetitions to be executed was a 16b register, which means that, in order to exhaustively test it, an operation had to be repeated 2^{16} times, which would have required a unreasonably long time.

4.3 Evolutionary approach

At first, an attempt was made to optimize a test program through an evolutionary approach using MicroGP. In order to do so, after a study of the controller FSM the instructions were executed one by one and the patterns produced on the primary inputs of the SPI controller (depending on several parameters, such as clock polarity, phase, size of transmission etc.) were manually recorded.
At this point a very crude assembly language was set up, in which the instructions could be programmed in order to be compiled into input patterns by an equally crude C++ parser; said patterns were saved into a `.txt` file and fed to the SPI controller by means of a system verilog testbench.

This kind of simulation served two purposes:

1. Allowed evolutionary optimization, by transforming the instructions into a language that could be parsed and replicated by the evolutionary tool.

2. Significantly shortened the simulation, since it was not necessary to simulate the entire pulp platform, but the controller could be simulated alone, still using only patterns that could be replicated with the controller being embedded in its peripheral.

Sadly, despite the efforts, this approach did not turn out to be very useful: in facts the very structure of the controller was very unfit for evolutionary optimization, as it does not have many memory elements or any logic that is benefited by large programs, naturally favored in the selection operated by the evolutionary tool.
Some of the programs generated with this procedure provided a good coverage, but at the cost of a very high test duration (some test programs lasted over 6 seconds) which obviously came with a very high evaluation time (the

fault simulation could last even several hours for each candidate). This led to dropping this method in favor of a hand written program, which is described in the following sections.

4.4 Structure of the test program

The structure of the new test program is rather simple, as its structure is similar to that of a full duplex operation: first a receive operation is issued to the base address of the RX channel, allowing it to be updated when data is received from the slave, and while the RX buffer is enabled a series of operations are performed, by sending all possible commands. Before each operation the clock baudrate is set to a different value, in order to test the clock divider.

4.4.1 Simple operations

In this first part of the program all functions were tested without repetitions. The typical commands sent for each operation were the following:

1. CFG command in order to set clock parameters
2. SOT command indicating the beginning of the operation
3. Actual commands for the operation to be tested, followed, when needed, by additional data for operations like send, full duplex transmission and send address
4. EOT command to conclude the communication and set the FSM back into its IDLE state

The performed operations were the following:

Send and receive operations

A fair amount of simple send and receive operations is performed, in order to test other parts of the SPI peripheral; the amount of data to be sent and received varies from an operation to another, in order to test the registers containing the data sizes, which are contained in the SPI controller.

Wait cycle operation

The WAIT_CYCLE command is sent, setting the wait counter to its maximum possible value and letting it count down, therefore testing it thoroughly.

Send command, send address and dummy instructions

These operations are issued in order to test the related registers and logic in the controller.

4.4.2 Repeated operations

Testing repeated operations turned out to be a longer, yet still manageable task. At first, the register containing the number of repetitions to be performed had to be tested, and in order to do so an instruction had to be repeated many times. The most suitable operation was a wait cycle with the down counter set to 0, since it was the fastest, thus shortening the testing time.

Order of repeated operations

Due to a flaw in the netlist (which was corrected in the updated version) when signals `r_rpt_do_cmd`, `r_rpt_do_addr`, `r_rpt_do_dummy`, `r_rpt_do_write`, `r_rpt_do_rx_chk` and `r_rpt_do_read` there is no way of setting them back to zero, aside from resetting the entire SPI module (which means resetting

the entire SOC and it is obviously not a viable option). Given the structure of the FSM in figure, this can clearly be a problem, since one of the signals being set to 1 prevents the FSM from reaching further states, therefore making the test impossible.

In this case, the only viable solution was to perform the operation in the correct order, which is:

1. Repeated read
2. Check on read
3. Repeated write
4. Repeated dummy send
5. Repeated address send
6. Repeated command send

It is important to mention that, after this test routine this part of the FSM cannot be used without a reset and therefore it is not actually usable due to this major flaw.

4.5 Final test program

In the end, the sequence of performed operation in the test program was the following:

- Receive operations: data is sent by the slave.
- Repeat for a high number of times a wait cycle with the counter set to 1 in order to test the register containing the number of time an instruction has to be repeated.

- Repeated receive operation.
- Dummy send: sends dummy data and toggles the dummy registers.
- Repeater RX check operation.
- Send operations.
- Full duplex transmission.
- A second dummy send to better cover the dummy registers.
- Send address and send command operations.
- Remaining repeater operations in the aforementioned order: send dummy, address, command, transfer, TX data.

This rather simple program allowed to reach all the states of the previously depicted finite state machines, except for those related to asynchronous events. In facts, as previously explained such events are handled in software at scheduler level, and therefore no event was seen at the controller level, even when asynchronous events were used.

Chapter 5

Conclusions: simulation results and possible further developments

5.1 Simulation and coverage results

The coverage results for the test program described in the previous section are shown in the following table:

Module	Total faults	Fault coverage
reg_if	2428	61.41%
clockgen	1018	87.91%
tx	8414	94.34%
fifo	1650	88.48%
rx	8398	96.93%
spictrl	7774	72.78%
txrx	7228	79.75%
TOP MODULE	**37630**	**84.46%**

And the total duration for this functional test routine was 127ms (including a 19ms initial JTAG sequence).

Despite not being stunning, these results compare very well with those obtained using only the standard libraries, being the following:

Module	Total faults	Fault coverage
reg_if	2428	53.29%
clockgen	1018	80.83%
tx	8414	95.50%
fifo	1650	86.79%
rx	8398	96.42%
spictrl	7774	23.54%
txrx	7228	73.70%
TOP MODULE	**37630**	**72.30%**

with a program duration of 996ms.

5.2 Final considerations

It must be noted that even programs that lasted way more than a second didn't provide any substantial improvement to this coverage value, but only some slight, hardly sensible increase in the coverages of the TX and RX modules, when the main issue was the controller.
It is also worth noticing that whereas these programs performed long sequences of send and receive operations in order to achieve higher coverages, they reach values that are not very different from those obtained mainly focusing on the controller. It is therefore reasonable to suppose that, when testing an embedded peripheral it is a good idea to start from its controller.

It is the most embedded part and therefore the most difficult to test, but it is quite likely that its test will cover many other parts of the peripheral.

5.3 Possible further developments

The test program presented in this work could be refined and extended in order to achieve a higher coverage on the various modules of the Pulpissimo SPI peripheral, but it would probably be better to extend the test to other embedded peripherals of the pulp platform by replicating (and therefore further validating) this method.
By starting from the peripheral controller and forcing it into all possible state while performing operation on reasonably large sets of data it is simple to achieve high values of coverage for the entire peripheral.
Sadly, this kind of task proved to be particularly unfit for evolutionary applications, which tend to perform better when large programs with many repetitions of the same instructions are needed, and therefore it would not make much sense to employ such tools while testing embedded peripherals with this method.

Bibliography

[1] Claudio Passerone, *Analog and digital electronics for embedded systems*, CLUT, Torino, 2015, http://hdl.handle.net/11583/2651591

[2] Antonio Pullini, Davide Rossi, Germain Haugou, Luca Benini *μDMA: An Autonomous I/O Subsystem For IoT End-Nodes* , Integrated Systems Laboratory, ETH Z¨urich, Gloriastr. 35, 8092 Zurich, Switzerland
https://ieeexplore.ieee.org/document/8106971
DEI, University of Bologna, Via Risorgimento 2, 40136 Bologna, Italy.

[3] Uemori, Satoshi Yamaguchi, Takahiro Ito, Satoshi Tan, Yohei Kobayashi, Haruo Takai, Nobukazu Niitsu, Kiichi Ishikawa, Nobuyoshi. (2011). ADC linearity test signal generation algorithm. 44 - 47. 10.1109/APCCAS.2010.5774755.
https://www.researchgate.net/publication/224238915_ADC_linearity_test_signal_generation_algorithm

[4] Sonza Reorda, Matteo. (2019). Slides from the "Testing and fault tolerance" course, Politecnico di Torino. http://www.polito.it