GMSK100 Bible

February 2020

GENERAL MANUAL GMSK100 P&D

February 2020

GMSK100 Project Description

Maarten Baert (maarten.baert@esat.kuleuven.be)

Thomas Bos (thomas.bos@esat.kuleuven.be)

Umut Celik (umut.celik@esat.kuleuven.be)

Carl D'Heer (carl.dheer@esat.kuleuven.be

Kaizhe Guo (kaizhe.guo@esat.kuleuven.be)

Pouya Housmand (pouya.houshmand@esat.kuleuven.be)

Yifan Lyu (yifan.lyu@esat.kuleuven.be)

Prof. Wim Dehaene (wim.dehaene@esat.kuleuven.be)

Prof. Michiel Steyaert (michiel.steyaert@esat.kuleuven.be)

Prof. Patrick Reynaert (patrick.reynaert@esat.kuleuven.be)

Prof. Filip Tavernier (filip.tavernier@esat.kuleuven.be)

GENERAL MANUAL GMSK100 P&D

GMSK PROJECT DESCRIPTION

INTRODUCTION

You are following the Master of Electrical Engineering - *Electronics and Chip Design*. For the P&D course H09L9A, we will design a mixed-mode chip that builds on the various aspects of this Master program: electronics, signal processing, analog and digital circuits and system design, technology … We decided to make a mixed-mode chip for receiving a 100bps GMSK signal. You will go through the entire design process, from specifications down to prototype!

This is a Master Course, so you are expected to be able to work independently, and find solutions based on information that will be provided to you by the teaching assistants (TAs). The TAs will not solve your problem. They will only give you some input that allows you to solve it yourself. After all, there is no right-wrong solution. There are only different considerations and this P&D is the excellent way for you to learn how to make the best decision, based on the technical considerations and trade-offs. As said before, the TA will not make that decision for you. It's up to you now!

Of course, we will help and support you in this process. Several documents exist that will guide you through the GMSK system, the (rough) architecture we propose, the analog design flow and the digital design flow. You will also receive some digital IP blocks that you can plug into your digital architecture.

Planning and *teamwork* are the key-words for this project. We will have intermediate presentations of 15 minutes where you present your achievements and how you will approach the problem. What is your methodology? What is your time-line? How will you divide the work in your team? …

75% of your points will be based on your final presentation and your report. 25% will be based on your daily work. The exact time and date of the intermediate and final presentations will be provided later, as well as the deadline to hand-in your report.

This project is your opportunity to design, simulate and test your own chip. We hope that you will enjoy it and take the opportunity to learn.

P. Reynaert – W. Dehaene – M. Steyaert

Contents

THE GMSK SYSTEM

APPLICATION OVERVIEW

Gaussian Minimum-Shift Keying (GMSK) is a spectrally efficient, low-energy, Frequency-Shift Keying (FSK) modulation scheme. This scheme is used in multiple low energy, low throughput schemes such as Bluetooth 1.0, DECT and GSM.

The GMSK scheme will be used in a typical Internet-of-Things (IoT) application, depicted in Figure 1. Such an IoT application typically consists out of multiple small, battery operated sensor nodes and a large, often grid powered base station. The sensor nodes collect environmental information about for example air pollution, traffic congestion, room air quality, … They reside in places without easy access to the electrical grid, hence they are battery powered. Further, the base station collects the sensor information by polling the sensor nodes on updates. In order to exchange information, a (half-) duplex communication link has to be developed.

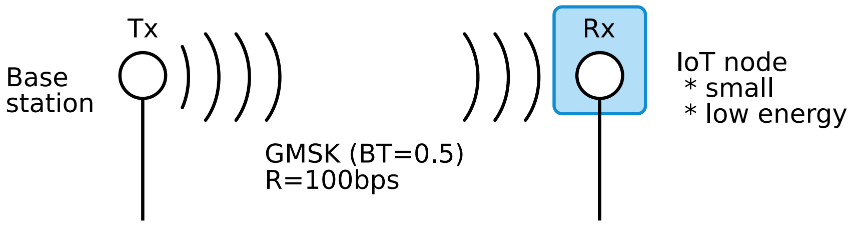


FIGURE 1: EXAMPLE GENERIC IOT SYSTEM

In this P&D we will, however, make abstraction of the specific type of IoT sensor. Your design task is to develop a performant, low-energy receiver to the GMSK signal further described in this document. You can think of an application such as a parking sensor, a pollution monitor, or … In general any IoT node application that relies on a low throughput, low energy, wireless communication link.

THE GMSK SIGNAL

Gaussian Minimum-Shift Keying (GMSK) is a variant of Minimum-Shift Keying (MSK), which itself is a type of Continuous-Phase Frequency-Shift Keying (CPFSK). CPFSK transmits digital bits by switching between two different transmission frequencies corresponding to a 0 or a 1. The frequency switching is done in such a way that the phase is continuous, which ensures that the signal has no discontinuities. MSK is equivalent to CPFSK with a modulation index of exactly 0.5, which means that the difference between the low (0) and high (1) frequency is equal to exactly half the bit rate. This means that relative to the nominal carrier frequency, the phase of the frequency-modulated signal will change by -90 degrees (0) or +90 degrees (1) in one bit period.

GMSK is very similar to MSK, the only difference is that the digital data signal is first smoothed with a Gaussian filter, such that the transmitted frequency changes smoothly rather than abruptly. The amount of filtering is described by the 'BT' value, which corresponds to the 3 dB bandwidth of the gaussian filter (B) multiplied by the bit period (T). The signal used in this project is modulated with a BT value of 0.5. More information about MSK and GMSK can be found online (see further under “useful references”).

The digital data is encoded as Varicode. Varicode is a variable-length encoding for ASCII characters which is relatively easy to decode because it is self-synchronizing. Characters are separated by two or more 0 bits. The code for each character starts and ends with a 1 bit, and never contains consecutive 0 bits. More information about Varicode can be found online (see further under “useful references”).

In order to reliably transmit data at low signal-to-noise ratios, forward error correction can be optionally used. The standard NASA convolutional code with rate ½ and constraint length 7 is used, which uses the generators 1111001 and 1011011. In this mode, the physical bit rate is 100bps while the actual data rate is 50bps (since the convolutional code doubles the number of bits). More information about convolutional codes can be found online (see further under "useful references").

A visualization on the communication chain in this project is given in Figure 2. A certain text message *m[k]* is encoded with Varicode into a 100 bps bit stream. This stream is further GMSK modulated and converted for passband transmission. In this project, we omit the RF passband (de)modulation and allow to start the design process on the intermediate frequency (IF) signal. From that signal, you need to design a receiver that retrieves the text message *m[k]*.

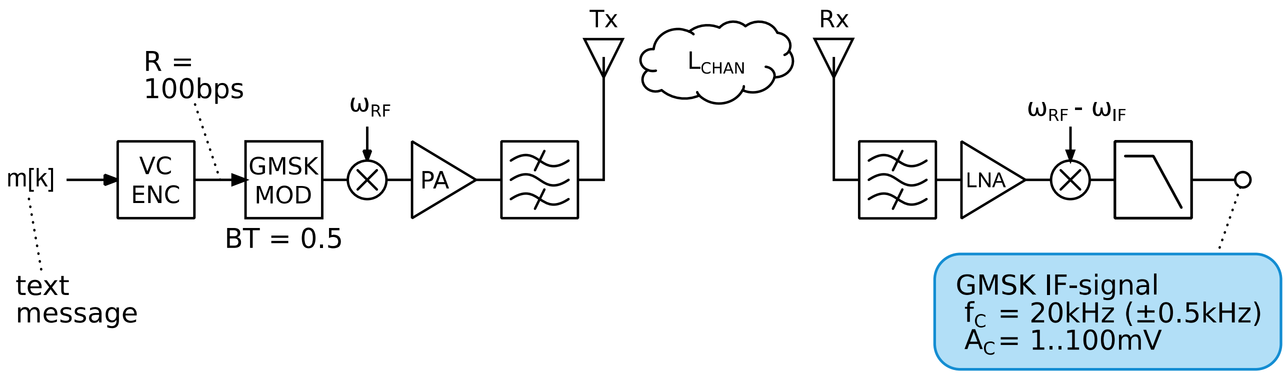


FIGURE 2: COMMUNICATION CHAIN OF GMSK SIGNAL

Next table provides a summary on the GMSK signal specifications:

Carrier frequency	20 kHz +/- 0.5 kHz
Carrier amplitude	1 mV … 100 mV
Modulation	Gaussian Minimum-Shift Keying (GMSK), BT=0.5
Bit rate	100 bit/s
Encoding	Varicode with optional convolutional code

Below the Varicode source encoding table is given:

Varicode	Ascii			Varicode	Ascii			Varicode	Ascii			Varicode	Ascii		
Binary	Dec	Hex	Char	Binary	Dec	Hex	Char	Binary	Dec	Hex	Char	Binary	Dec	Hex	Char
1010101011	0	0	NUL	1	32	20	SP	1010111101	64	40	@	1011011111	96	60	`
1011011011	1	1	SOH	111111111	33	21	!	1111101	65	41	A	1011	97	61	a
1011101101	2	2	STX	101011111	34	22	"	11101011	66	42	B	1011111	98	62	b
1101110111	3	3	ETX	111110101	35	23	#	10101101	67	43	C	101111	99	63	c
1011101011	4	4	EOT	111011011	36	24	$	10110101	68	44	D	101101	100	64	d
1101011111	5	5	ENQ	1011010101	37	25	%	1110111	69	45	E	11	101	65	e
1011101111	6	6	ACK	1010111011	38	26	&	11011011	70	46	F	111101	102	66	f
1011111101	7	7	BEL	101111111	39	27	'	11111101	71	47	G	1011011	103	67	g
1011111111	8	8	BS	11111011	40	28	(	101010101	72	48	H	101011	104	68	h
11101111	9	9	HT	11110111	41	29	)	1111111	73	49	I	1101	105	69	i
11101	10	0A	LF	101101111	42	2A	*	111111101	74	4A	J	111101011	106	6A	j
1101101111	11	0B	VT	111011111	43	2B	+	101111101	75	4B	K	10111111	107	6B	k
1011011101	12	0C	FF	1110101	44	2C	,	11010111	76	4C	L	11011	108	6C	l
11111	13	0D	CR	110101	45	2D	-	10111011	77	4D	M	111011	109	6D	m
1101110101	14	0E	SO	1010111	46	2E	.	11011101	78	4E	N	1111	110	6E	n
1110101011	15	0F	SI	110101111	47	2F	/	10101011	79	4F	O	111	111	6F	o
1011110111	16	10	DLE	10110111	48	30	0	11010101	80	50	P	111111	112	70	p
1011110101	17	11	DC1	10111101	49	31	1	111011101	81	51	Q	110111111	113	71	q
1110101101	18	12	DC2	11101101	50	32	2	10101111	82	52	R	10101	114	72	r
1110101111	19	13	DC3	11111111	51	33	3	1101111	83	53	S	10111	115	73	s
1101011011	20	14	DC4	101110111	52	34	4	1101101	84	54	T	101	116	74	t
1101101011	21	15	NAK	101011011	53	35	5	101010111	85	55	U	110111	117	75	u
1101101101	22	16	SYN	101101011	54	36	6	110110101	86	56	V	1111011	118	76	v
1101010111	23	17	ETB	110101101	55	37	7	101011101	87	57	W	1101011	119	77	w
1101111011	24	18	CAN	110101011	56	38	8	101110101	88	58	X	11011111	120	78	x
1101111101	25	19	EM	110110111	57	39	9	101111011	89	59	Y	1011101	121	79	y
1110110111	26	1A	SUB	11110101	58	3A	:	1010101101	90	5A	Z	111010101	122	7A	z
1101010101	27	1B	ESC	110111101	59	3B	;	111110111	91	5B	[	1010110111	123	7B	{
1101011101	28	1C	FS	111101101	60	3C	<	111101111	92	5C	\	110111011	124	7C	|
1110111011	29	1D	GS	1010101	61	3D	=	111111011	93	5D	]	1010110101	125	7D	}
1011111011	30	1E	RS	111010111	62	3E	>	1010111111	94	5E	^	1011010111	126	7E	~
1101111111	31	1F	US	1010101111	63	3F	?	101101101	95	5F	_	1110110101	127	7F	DEL

USEFUL REFERENCES

For more information on GMSK and MSK:

- https://en.wikipedia.org/wiki/Minimum-shift_keying
- https://www.researchgate.net/publication/2575678_GMSK_in_a_nutshell
- https://dsp.stackexchange.com/questions/30653/what-is-a-bt-bandwidth-time-product-with-reference-to-modulation
- https://en.wikipedia.org/wiki/Varicode
- https://en.wikipedia.org/wiki/Convolutional_code

THE GOAL OF THIS P&D

INTRODUCTION

Problem solving and design is an exercise course which aims at developing critical reasoning and analysis skill of an engineer. With this in mind, a P&D exercise framework is designed, allowing to explore this aspect of education. It involves a number of subtasks which together form a complex activity. There can be multiple approaches for the subtasks and you are expected to bring out a balanced choice. There can be constraints which have to be taken into consideration. Some trade-offs will have to be evaluated and justified. On top of that, there can also be some conditions and constraints imposed by the TAs in order to limit the design space within bounds. It can be considered as a framework within which you will have to explore. You have to understand these boundaries and respect them.

The intention of this project is multi-fold. It aims to bring together the various aspects of electrical engineering -from design, project management to deliverables- under a single umbrella. You will be working in a small group of three to four people. This gives you an opportunity to collaborate with your colleagues. In doing so, this exercise wants to show you the importance of working in groups and to appreciate it as you work towards a common outcome. This project will give you a taste of how to begin from a text description to a working implementation, which is a design flow quite common in industry. Students who have done the Eagle project in the 3rd bachelor year have a fair idea about it. After reading the documents provided to you, feel free to ask questions if some things are still unclear. It is important to know what you are expected to do.

The GMSK receiver you will be working on consists of an analog front-end and a digital back-end. Integration of these two fundamental blocks requires good communication among the students involved. Everyone needs to have a clear understanding of the complete system. It is necessary to already get the big picture early on.

GOAL OF THIS P&D

Towards the end of this project (near the 3rd week of May) you need to propose your IF-GMSK receiver implemented in a mixed-signal integrated circuit (IC). An example of the high-level implementation of such a receiver is shown in Figure 3:

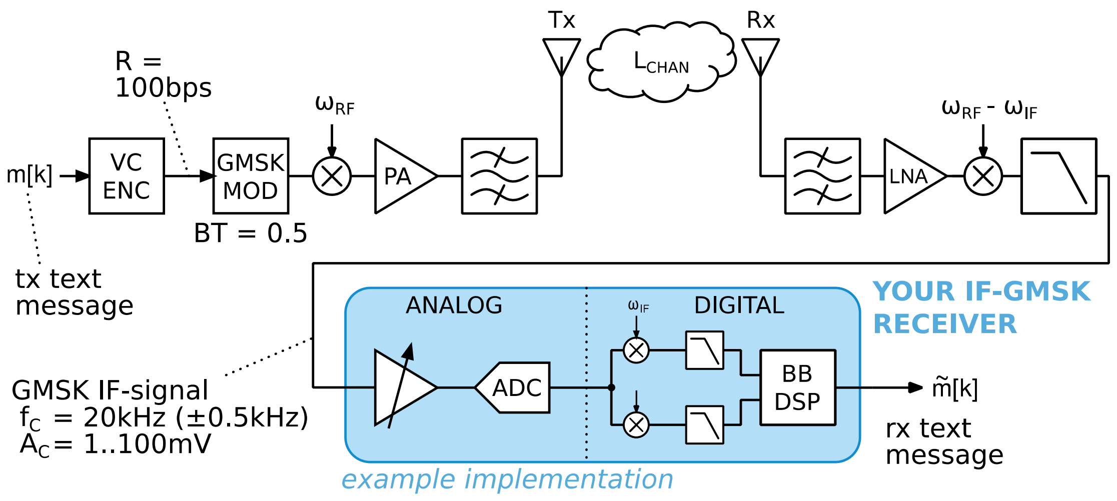


FIGURE 3: DESIGN GOAL OF YOUR IF-GMSK RECEIVER

Through both a written report and an oral presentation format, you are asked to clearly present your overall system design. In these evaluations you need to reason your design choices, elaborate them with simulations and show the final performance metrics.

To aid you in the design process and to make the project more hands-on, you will implement your design on a prototype board. This setup will be given by the TA's in the second phase of the project. Using this setup, you can (within certain limits) emulate your custom analog front-end. The digital back-end will be implemented on a Spartan 3 FPGA. More details on the capabilities of this prototype board can be found further in this document.

TOOLS AND FLOW

To help you gain familiarity with the design flow, a few example exercises are provided. They will help you to get comfortable with the intricacies without losing the overview of what is happening when, where and why. The analog and digital part of the project have their own design flow. It is recommended to follow them as this will help you to keep your design cycle manageable.

The analog flow will introduce you to the netlist entry, the spice simulator and the waveform viewer. The digital flow will introduce you to VHDL programming, FPGA implementation and debugging. Examples are provided for each step. Using the know-how you gained from these examples, you have to tackle the DCF77 project as a group.

Another important aspect of building complex designs is Intellectual Property (IP). This can be a readymade design which a company buys from another company for various reasons (lack of expertise, time, cost, legal reasons, etc.). If you receive such IPs to work with, it is important that you know the functionality and its interfaces well, so that you can confidently include it in the complete design.

PLANNING AND EVALUATION

You will have about 60 lab hours in total to complete the project. So come up with a plan early on and divide the work in a balanced manner! Participation of all the students of a group is necessary for a successful outcome. You will have a mid-project review in the form of a presentation. It will give you the opportunity to explain where you stand. It will also act as a feedback mechanism which will enable you to be critical regarding your own project and help you achieve the best results. The final evaluation will be a presentation somewhere in the month of May. The actual dates will be communicated to you. You will also have to write a report detailing your design approach, methodology, simulation results etc.

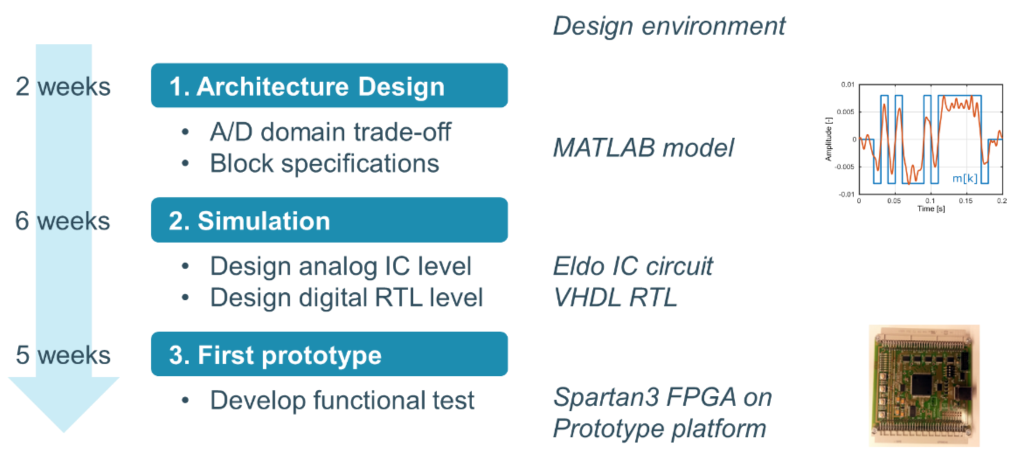


FIGURE 4: PROJECT PHASES

ANALOG FRONT-END

The analog front-end of the GMSK receiver has to be designed by you during this project. This analog front-end circuit should convert a small, noisy input signal into a digital output that can be demodulated by the digital part of the receiver. The most classical topology would be the combination of a variable gain amplifier (VGA) and analog-to-digital converter (ADC). How to implement these blocks is completely up to you.

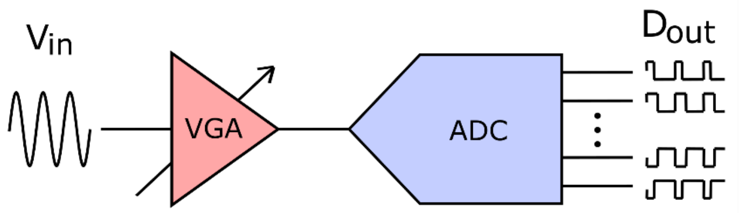


FIGURE 8: GENERIC TOPOLOGY OF AN ANALOG FRONT-END RECEIVER

The **variable gain amplifier** should be able to amplify the input signal and filter out high-frequency (out-of-band) noise. The effective gain of the amplifier should be variable as the amplitude of the input signal can vary and input range of the ADC will be limited. The gain should preferably be controlled through a number of digital inputs. A possible implementation is an OTA with a tuneable feedback network, such as a resistor bank with digitally controllable switches.

The **analog-to-digital converter** will convert the analog output of the VGA into a digital signal which will be processed by the digital part. The number of bits with which you represent your digital signal should be determined by you. More bits might make the digital demodulation easier, but complicates the analog design of the ADC. A lot of different ADC topologies are possible. The flash ADC is most straightforward, but you're free to explore different options like the integrating, SAR and even sigma-delta ADC.

As an example, a possible implementation is given below consisting of a 3-bit resistive feedback VGA and 2-bit flash ADC.

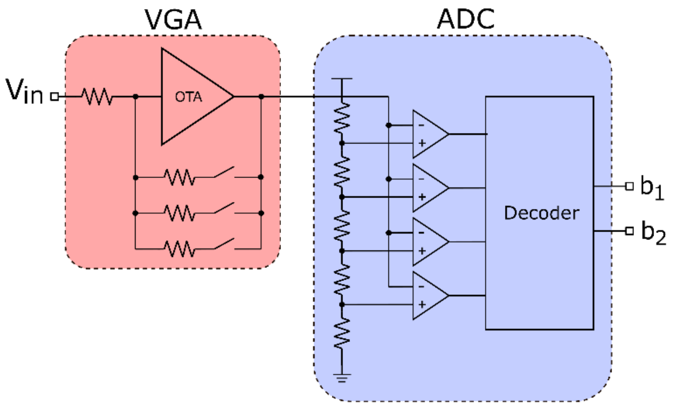


FIGURE 9: POSSIBLE IMPLEMENTATION OF ANALOG FRONT-END

The design of the analog front-end happens in two steps: a high-level analysis and transistor-level implementation and design.
During the **high-level analysis** you will determine the optimal topology taking into account the necessary specifications of the analog and digital part.
Once your topology is fixed, each block should be **implemented and designed at transistor-level.** You will need to determine the appropriate sizes for all transistors and all necessary bias voltages. More details on the design can be found in the analog flow manual.

DIGITAL DECODING

GMSK can be demodulated either with a coherent or an incoherent demodulator. A coherent demodulator can potentially perform better, but is significantly more complicated and more sensitive to synchronization issues and clock jitter. For this reason we strongly recommend starting with an incoherent demodulator.

A typical receiver contains the following blocks:

- **Automatic gain control:** This module checks the amplitude of the signal and changes the gain if necessary. The goal is to keep the amplitude within a certain acceptable range so that the decoding algorithm will work properly.
- **Quadrature mixer:** The input signal is mixed with a sine and cosine wave to produce a complex envelope (I/Q) signal. For an incoherent demodulator it is sufficient to use a frequency that is relatively close to the actual frequency, it does not need to match exactly. For a coherent demodulator the frequency and phase must match those of the transmitter exactly. The sine and cosine waves are generated by a numerically controlled oscillator (NCO). Note that it is not necessary to use an exact sine and cosine wave, an approximation will also work, however the results will be less accurate. The CORDIC algorithm can be used to produce high-quality sine and cosine waves if necessary.
- **Low-pass filter:** The I and Q signals are low-pass filtered to remove out-of-band noise.
- **Carrier recovery:** Since the carrier frequency is not known exactly in advance, it must be recovered first, for example by sweeping the mixer frequency across the entire frequency range and selecting the frequency with the strongest signal.
- **Phase extraction:** The phase can be extracted from the complex envelope signal as 'arctan2(Q, I)'. This function can be implemented in hardware using the CORDIC algorithm, or can be approximated in many other ways.
- **Derivator:** The derivative of the phase is calculated in order to derive the relative frequency. Special care should be taken to avoid spikes near the 180/-180 degree transition. The relative frequency will be zero when the input frequency is equal to the mixer frequency, positive when it is higher, and negative when it is lower.
- **Clock recovery:** A clock signal with a frequency equal to the bit rate must be generated in order to determine the correct sampling times. The phase of the sampling clock must be constantly adjusted to ensure that it tracks the actual bits. This can be done based on the zero crossings of the relative frequency signal.
- **Quantizer:** The sampling clock is used to sample the relative frequency. Ideally the mixer frequency should be (almost) equal to the carrier frequency, such that a positive relative frequency corresponds to a 1 bit and a negative relative frequency corresponds to a 0 bit.
- **Convolutional code decoder (optional):** In order to decode the convolutional code (if used), the Viterbi algorithm can be used. Note that this block is very complicated and implementing it should only be attempted after everything else already works perfectly. Convolutional codes are only used at low signal-to-noise ratios (levels 8-15 of the generator board), so the convolutional code decoder is not strictly required, but it will improve the performance of the system significantly.
- **Varicode decoder:** The bits are decoded from Varicode to ASCII text. The decoded text is sent to the computer using the UART interface. Since the same UART interface is used for debugging, we recommend using one of the switches to determine whether the UART interface should transmit decoded text or debug information.

A few extra blocks which are useful for testing and to interface with the outside world will be provided to you as part of the example project:

- **Display driver:** This module handles the process of displaying digits on the display.
- **Debouncer:** This module filters the signals from the switches and buttons.
- **UART transmitter:** This module allows you to send data to the PC to simplify debugging. The module is described in more detail in the digital flow manual.

HIGH-LEVEL MODEL

You will be provided with a high-level MATLAB model of a GMSK modulator and demodulator. The modulator is already fully implemented and doesn't need to be modified. The demodulator however is incomplete, you should add the missing components yourself and determine the best parameters for optimal performance. The high-level model can be found here:

```
$ /esat/micas-data/data/design/i3t50/gmsk100-2020/model/
```

EXAMPLE WAVEFORMS

In addition to the test signals that you can generate using the high-level model, you will also be given example signals that have been measured on the actual hardware. For practical reasons these waveforms were captured after mixing and downsampling, but are otherwise unmodified. The sample rate is 6.4 kHz (i.e. an oversampling ratio of 64). These waveforms can be found here:

```
$ /esat/micas-data/data/design/i3t50/gmsk100-2020/waveforms/
```

EXAMPLE ONLINE RESOURCES

These resources can be used to help you with the implementation of the digital building blocks described above:

- Numerically Controlled Oscillator (NCO):
  https://en.wikipedia.org/wiki/Numerically_controlled_oscillator
- CORDIC algorithm for calculating sin(), cos(), arctan2():
  https://en.wikipedia.org/wiki/CORDIC
- Cascaded integrator-comb filter, a very efficient digital low-pass/downsampling filter:
  https://en.wikipedia.org/wiki/Cascaded_integrator%E2%80%93comb_filter
- Matched filter theory (not 100% applicable to GMSK but still relevant):
  https://en.wikipedia.org/wiki/Matched_filter
- Extra information about math in VHDL (note, this doesn't replace the DCF77 VHDL tutorial):
  http://www.synthworks.com/papers/vhdl_math_tricks_mapld_2003.pdf

For some of the building blocks example implementations can be found online, but make sure that you understand how the underlying algorithms work!

DESIGN PROCEDURE

Since this is a rather complicated system, it is highly recommended to implement it step-by-step and thoroughly test each block before moving on to the next one. It is a good idea to rely on simulations for the initial implementation of each block, because this allows for much easier debugging, but you should not neglect testing on the actual hardware. Do not delay this until the very end of the project or you will run into unpleasant surprises. The fact that something works in simulation is no guarantee that it will work in reality!

The recommended design procedure is as follows:

- Create a high-level model of your system in MATLAB, Python or some other high level language
- Implement one block of the model in VHDL. Use the high-level model to generate test data for the block, and create a testbench that tests the block using this data. Verify that the output matches that of the high-level model. Repeat this for every block.
- Combine the blocks you have implemented and simulate them together. Compare the results to the high-level model.
- Add the implemented blocks to the hardware implementation one at a time. Use the UART debug interface to obtain waveforms and compare them to what you expect based on the high-level model. You can also feed the waveforms obtained this way into your high-level model to verify how well the model works with real-world data.
- Once you have a complete hardware implementation, test it at various difficulty levels and try to tweak it to maximize the performance. When data is occasionally not decoded correctly, try to find out exactly where the decoder went wrong.

Many of these steps can be done in parallel, so you should not try to do everything sequentially! It is a very good idea to start testing on the actual hardware once you have implemented a few blocks, because you will gain experience this way that will help you implement the next blocks more effectively.

Some real-world aspects are hard to test with high-level models. This includes things like synchronization, phase noise, clock frequency offset/drift, interactions between the analog and digital parts (such as automatic gain control), limited hardware resources etc. Try to start real-world testing as soon as possible!

You do not necessarily have to implement the entire system to pass this course. It is much better to implement half of the blocks properly and demonstrate them on the actual hardware, than to implement all blocks poorly or only in simulation. For example, a system that can receive and demodulate data but can't decode it is far more useful than a system that does everything but only exists in simulation! As mentioned earlier, the use of convolutional codes is optional, and should only be attempted if you are already able to decode simpler signals reasonably well (levels 0-7 of the generator board).

ANALOG DESIGN FOR THE GMSK RECEIVER

February 2020

Analog design flow

Maarten Baert (maarten.baert@esat.kuleuven.be)

Thomas Bos (thomas.bos@esat.kuleuven.be)

Umut Celik (umut.celik@esat.kuleuven.be)

Carl D'Heer (carl.dheer@esat.kuleuven.be

Kaizhe Guo (kaizhe.guo@esat.kuleuven.be)

Pouya Housmand (pouya.houshmand@esat.kuleuven.be)

Yifan Lyu (yifan.lyu@esat.kuleuven.be)

Prof. Wim Dehaene (wim.dehaene@esat.kuleuven.be)

Prof. Michiel Steyaert (michiel.steyaert@esat.kuleuven.be)

Prof. Patrick Reynaert (patrick.reynaert@esat.kuleuven.be)

Prof. Filip Tavernier (filip.tavernier@esat.kuleuven.be)

Analog DESIGN for the GMSK Receiver

ANALOG DESIGN BASICS AND FLOW DESCRIPTION

INTRODUCTION

The GMSK receiver will contain an analog front-end, followed by digital demodulation circuitry. The front-end will convert the analog input signal into a digital output which will be used by the digital circuit to detect the amplitude of the input GMSK signal and ultimately demodulate the data.

To make sure that the different signal levels can be received, a well-designed analog front-end needs to be designed. We will not go into details on this specific design in this document. However, we will give you a basic overview of a typical analog design flow, which you will translate to the GMSK case.

For illustrative reasons, an on-chip delay line will be considered throughout this text. This is a very simple analog circuit that should enable new designers to grasp the basics of analog design.

CONTENTS

VERSION HISTORY

V1.0: February 2020 - Please forward errata and suggestions.

NOTATIONS

Command line commands

Throughout this document, many command line actions are explained. In order to make sure that you enter these commands in the right terminal, conventions are defined.

System command line

The system shell is the default Linux bash shell. Commands for this shell are preceded by a "$". It is important to run each command in its right directory. Two rules are valid:

1. "source" and "cd" commands do not have to be executed in any particular directory
2. All other commands need to be run in the right folder of the considered tool.

```
$ some_shell_command
```

Tool command line

Many tools provide a command line to interface with this particular tool. In this flow, the SKILL and TCL scripting language is often chosen by the tool vendors. In-tool commands have a mark-up as shown next:

```
> some_in_tool_command
```

Important notifications and warnings

Important notifications and warnings are marked by a double bordered box. Read this information very carefully.

Important notifications go here.

ANALOG FLOW INTRODUCTION

This part will focus on the analog design and simulation of the circuit and which tools to use for which purpose. First, we need to set up all directory structures in the correct way.

We highly recommend you to follow all guidelines provided in this document in a very strict way. Failure to do so might result in the failure of your design. This recommendation counts for: directory structure, used tools, proposed scripts and even button clicks.
Special attention goes to the way you interact with the many "automation" scripts which are found in the provided directory structure. These scripts give you the pseudo-feeling of handing over your design to a fully automatic, error-free design procedure. However, every script line's output can report errors while still enabling the tool's nominal behaviour for further scripting. This means that errors, once popped-up, propagate throughout the analog tool flow and might never be noticed. DO NOT CONSIDER THESE SCRIPTS AS A FULL AUTOMATIC DESIGN STEP. Be an engineer, remain sceptic about tools.

SETUP

Directories

Everything starts with a good directory structure. In order to help you construct the here-proposed directory structure, an install script is provided. Open a terminal and insert the following command. Note: Linux terminal commands (bash) are preceded by a "$" in this text.

```
$ /esat/micas-data/data/design/i3t50/gmsk100-2020/scripts/setup-analog.sh
```

This setup can take a while. Do not interrupt it. Your home account now contains a directory "gmsk100-analog" containing the here-listed subdirectories (Table 1) + some content.

Subdirectories

The created directory contains some subdirectories.

TABLE 1: ANALOG FLOW CORE DIRECTORIES

Subdirectory name	Purpose	Tool
spice/	Directory containing the netlists	Kate + HSpice / Eldo
simulate/	Directory containing the simulated output files	Hspice / Eldo + EZwave
spice_lvs/	Contains the files to perform an LVS check	Cadence Virtuoso

DESIGN AND SIMULATION

In the analog flow, there will be a constant iteration between design and simulation. An initial sizing will be made for the analog circuit and this circuit will be simulated. After simulation, the performance of the analog circuit will be evaluated. If the objectives are not reached, another design and simulation step need to be done to reach the wanted objectives eventually.

SETUP TOOLS

In this flow, the tools for simulation of netlists and viewing results are Eldo and EZWave. In order to use them, always **source** the following script in the terminal you will use for simulations:

```
$ source /esat/micas-data/data/design/i3t50/gmsk100-2020/scripts/start-spice.sh
```

WRITING NETLISTS

A good design starts with a clear and concise SPICE-netlist. To be able to work in an efficient way, we will use Kate for entering netlists. In order to open Kate, open a terminal and go to the spice folder:

```
$ cd gmsk100-2020-analog/spice
$ kate &
```

Kate will remember some history. When you open the program again on another moment, always choose to open an existing session. In Kate, the terminal is not shown automatically. The students need to go to 'Settings > Configure Kate > Plugins' and enable 'Terminal Tool View' first.

To make your life easier while writing netlists, you can select SPICE highlighting: Tools – Highlighting – Hardware – Spice. To make it even easier, always save these netlists with a .sp extension, then Kate will select the correct highlighting automatically for you.

Open the file **delayLine.sp** within the Kate environment, which is an inverter chain that delays the input signal by a certain amount of time. Since the number of inverters is odd, the output will be inverted. In order to get a non-inverted output, add one more inverter at end of the chain. To avoid stupid errors during this task, it is recommended to first draw the circuit on a piece of paper.

If you have trouble with SPICE syntax, you can always have a look in the ELDO manual at

```
$ okular /esat/micas-data/software/ams_2016.4/docs/pdfdocs/eldo_ref.pdf
```

IMPORTANT: Never start entering a netlist without having made a circuit drawing on a piece of paper yourself. On this circuit blueprint, determine all node and component names beforehand. This will save you a lot of time debugging netlists with stupid errors because of carelessness.

SIMULATING NETLISTS USING ELDO

Once you have edited the delay line circuit, it is time to simulate this netlist. As seen in the **delayLine.sp** file, this block is a sub-circuit. The supply needs to be defined. To do this in a clear way, we will use a test-bench file, defining the supply and defining some simulation parameters.

This file can be opened in Kate as well: **delayLine_tb.sp**

Kate has a terminal included at the bottom. This is a convenient way to do a quick simulation of the delay line block. Go to this terminal and type the following command:

```
> eldo -64b delayLine_tb.sp –outpath ../simulate
```

This command calls the Eldo simulator, to simulate the netlist.

NOTE: the option "-outpath" will make sure that all simulation files are written into the simulate folder in your analog directory tree. This directory is a link to the /tmp directory on your local computer. It has a certain "erase-policy", so make sure you do not put crucial data into this directory! The reason using this /tmp directory is because of 1) the limited disk space in your home account (the ~/ directory) for simulation output data, 2) the faster local access (versus over the network) and 3) the seperation of the netlist files and simulation files to keep it organized.

INSPECTING SIMULATION RESULTS USING EZWAVE

To view the SPICE simulation results, change directory to the simulate directory (using the Kate terminal):

```
> cd ../simulate
> ls
```

You will see that some files have been generated. We are interested in two files:

delayLine_tb.chi : this file contains all DC-operating point information. This is simulated using the **.op statement** in the **delayLine_tb.sp** file.

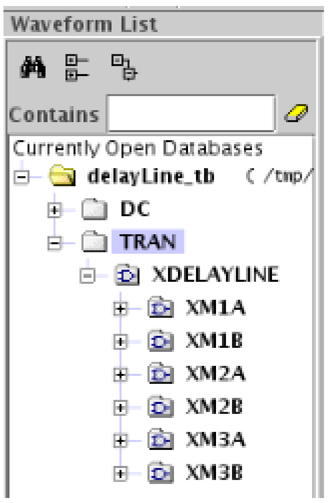
delayLine_tb.wdb : this file contains all simulated waveforms. This is the transient behavior of the circuit, simulated using the **.tran statement** in the **delayLine_tb.sp** file.

To inspect these files:

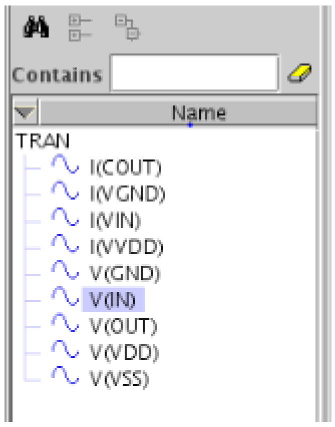
```
> gedit delayLine_tb.chi &
> ezwave delayLine_tb.wdb &
```

EZwave basics

At the left-hand top side of the window, you can choose which simulation results you want to see:



At the left-hand bottom side of the window, you can select which signals you want to see, by double clicking:



This will enable you to view the input and the output signal of the analog block.

To inspect these waveforms, add a cursor (Cursor - Add). You can draw this cursor to inspect the waveform more closely.

MONTE-CARLO SIMULATION

In the previous simulations, we were using the nominal parameters of the devices. However, in reality these parameters can change from chip to chip (process variation) and within the same die (mismatch). An analog designer should take into account all this variations, so that his chip will be working in all possible scenarios. In this project, it is sufficient to only consider the mismatch and not the process variation, since the former has a dominant impact on the analog front-end of the GMSK100 receiver.

To model the effect of mismatch on your circuit, we will use a Monte-Carlo analysis. This performs multiple simulation runs, each for a different set of model and device parameters of your technology. In the **delayLine_tb.sp** file, add *monte*=30 at the end of the line that begins with ".tran". Then, run the simulation as explained before and open your simulation results in ezwave. For each signal, you get an H and an L version, which are the maximum and minimum of all 30 signals, respectively.
What is the effect of mismatch on the output of the inverter chain?

Multiply the width of all transistors by a factor 4 and check again the effect on the mismatch.

NOTE: Make sure that the width and length of your transistors are a multiple of their corresponding minimum value, because otherwise the layout tool will round these values to fit the limited resolution of the fabrication process. This would give problems later on when making the layout of the circuit.

SUMMARY

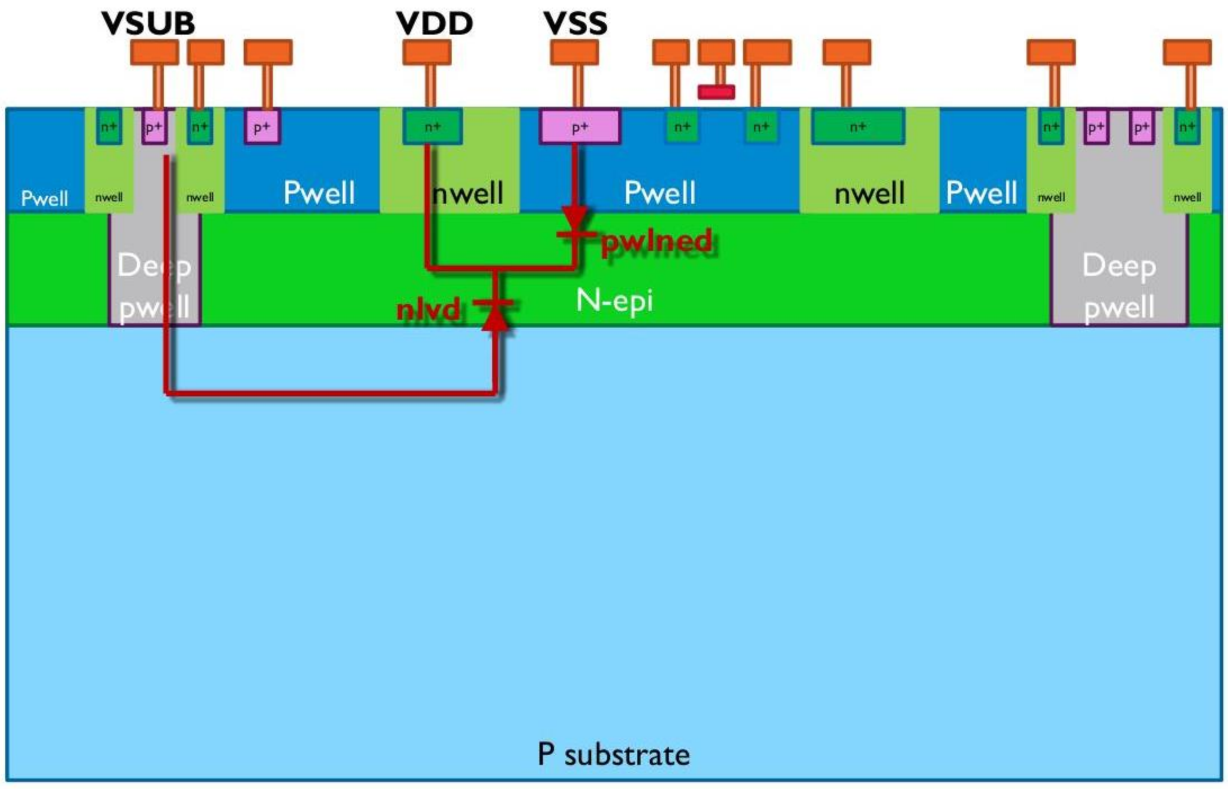
This is all you need to know to get started with the first step in analog design.

- Start by making a clear circuit **drawing**.
- Make a **netlist** for your analog block, using Kate as an easy highlighter.
- Make a **test-bench** to simulate your analog block. Don't forget that the header of the sub-circuit needs to match.
- **Simulate** your test-bench using ELDO.
- View the DC and TRANsient simulation **results** using "gedit" and "EZwave" respectively.

You will need to go through this flow iteratively when actually designing a circuit (in this case, the delay line was already correctly sized). You start with a calculated initial sizing, simulate the circuit and then evaluate the initial simulations results. Based on these results you make the necessary changes to the circuit to get closer to the desired performance. Also, it might be a good idea to first design each block separately before simulating everything together.

Remember, your job is to design a circuit which converts the GMSK analog input signal into a digital signal with adequate resolution while consuming the least amount of power.

Lastly, the figure below shows the technology that you will be designing your circuit in: i3t50.



DIGITAL DESIGN FOR THE GMSK RECEIVER

February 2020

VHDL hardware description
& Digital tool flow

Maarten Baert (maarten.baert@esat.kuleuven.be)

Thomas Bos (thomas.bos@esat.kuleuven.be)

Umut Celik (umut.celik@esat.kuleuven.be)

Carl D'Heer (carl.dheer@esat.kuleuven.be

Kaizhe Guo (kaizhe.guo@esat.kuleuven.be)

Pouya Housmand (pouya.houshmand@esat.kuleuven.be)

Yifan Lyu (yifan.lyu@esat.kuleuven.be)

Prof. Wim Dehaene (wim.dehaene@esat.kuleuven.be)

Prof. Michiel Steyaert (michiel.steyaert@esat.kuleuven.be)

Prof. Patrick Reynaert (patrick.reynaert@esat.kuleuven.be)

Prof. Filip Tavernier (filip.tavernier@esat.kuleuven.be)

Digital Design for The GMSK Receiver

VHDL HARDWARE DESCRIPTION BASICS & DIGITAL FLOW

INTRODUCTION

The GMSK100 receiver will require a significant amount of digital circuitry to convert the output of the analog front-end into a human-readable text message. In order to allow fast experimentation, this digital circuit will be implemented in a Field Programmable Gate Array (FPGA).

Due to the high complexity of modern digital systems, it is no longer practical to design these with discrete transistors or logic gates. For this reason, a high-level Hardware Description Language is used: VHDL. This document describes only the subset of VHDL that you will need to complete this project.

The conversion from VHDL code to a functioning FPGA happens in four steps:

1. *Synthesis*: The source code is converted to a circuit consisting of logic gates.
2. *Implementation*: The optimal physical location of the logic gates and the required wiring is calculated.
3. *Bitstream generation*: The physical description is encoded in a bitstream file.
4. *Configuration*: The bitstream file is copied to the configuration memory of the FPGA, which will then run the designed circuit.

These four steps are executed by tools provided by the FPGA vendor (Xilinx). The first three steps can be accessed through a program called 'ISE', and the fourth step through 'Impact'.

Contents

INSTALLATION

Before you start using the tools, you should run the following script:

```
$ /esat/micas-data/data/design/i3t50/gmsk100-2020/scripts/setup-digital.sh
```

This will create a directory called "gmsk100-digital" in your current working directory. You can run this script from your home folder, or any other folder if you want to place the files in a different location.

WORKING WITH XILINX TOOLS

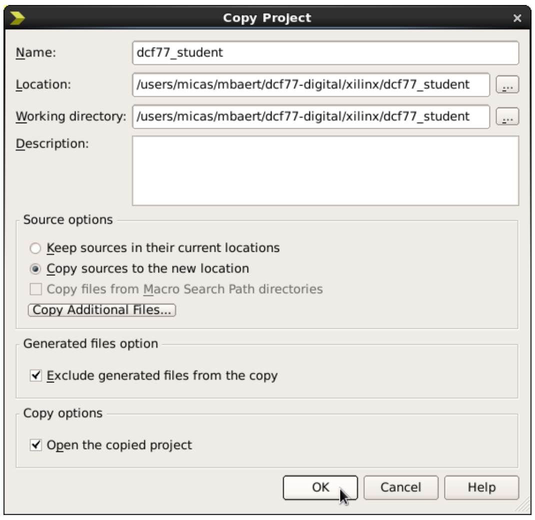
After you have run the setup script, you can start using the Xilinx tools. The main software package is called "ISE" and can be started with the following command:

```
$ /esat/micas-data/data/design/i3t50/ gmsk100-2020/scripts/start-ise.sh
```

Normally you would have to create a new project in ISE and specify the device that you want to use. In order to simplify this process, an example project is provided to you. Start ISE, then click "File" > "Open Project" and open the example project in the "gmsk100-digital" folder you created earlier. The file you need to open is "xilinx/gmsk100_example/gmsk100_example.xise".

A window will pop up, indicating the project is made using an earlier version of ISE. This should not pose any problems. Click "migrate only".

Do not edit the example project itself – you may still need it later. Instead, create a copy by clicking "File" > "Copy Project". Enter a name for your project and check "Open the copied project", then click "OK". You can now edit the source files, implement the design and run simulations.



Name: dcf77_student
Location: /users/micas/mbaert/dcf77-digital/xilinx/dcf77_student
Working directory: /users/micas/mbaert/dcf77-digital/xilinx/dcf77_student
Description:

Source options
- ○ Keep sources in their current locations
- ● Copy sources to the new location
- ☐ Copy files from Macro Search Path directories

Generated files option
- ☑ Exclude generated files from the copy

Copy options
- ☑ Open the copied project

Source files

A Xilinx project requires at least the following source files:

1. UCF file: This is also known as the 'constraints file'. This file defines the physical inputs and outputs of the circuit as well as various timing parameters, such as the clock speed. The 'dcf77.ucf' file provided in the example should be sufficient to complete this project – there is no need to modify it.
2. VHDL files: These files contain the VHDL code that defines the behaviour of the circuit. Each file describes one VHDL module. VHDL modules are hierarchical: each module can contain instances of other modules. One of these files is known as the top-level module. The top-level module

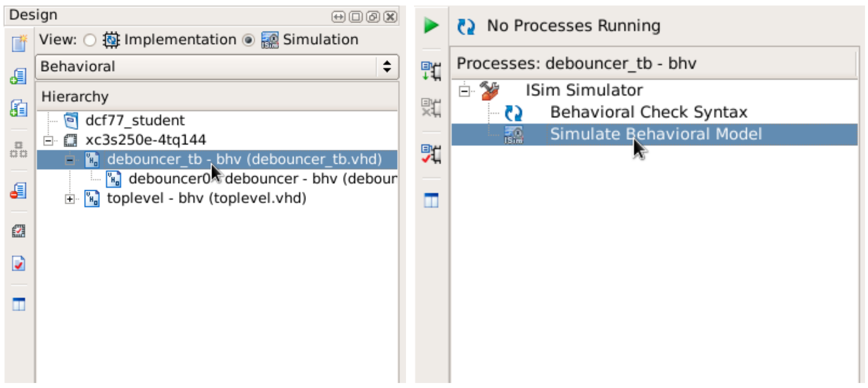
contains all the circuitry that needs to be included on the FPGA. The inputs and outputs of the top-level module must match the inputs and outputs that are defined in the UCF file.
It is possible to create VHDL files which are only used for simulation purposes and don't appear in the final circuit. These files are known as 'testbenches' and can be recognized by the "_tb" suffix.

New source files can be added by clicking "Project" > "New Source".

Simulating your design

It is a good idea to simulate your code first before you try to run it on the FPGA. This can be done by following these steps:

1. Make sure that the view is set to "Simulation", and select the testbench that you want to simulate.
2. Double-click "Simulate Behavioral Model".

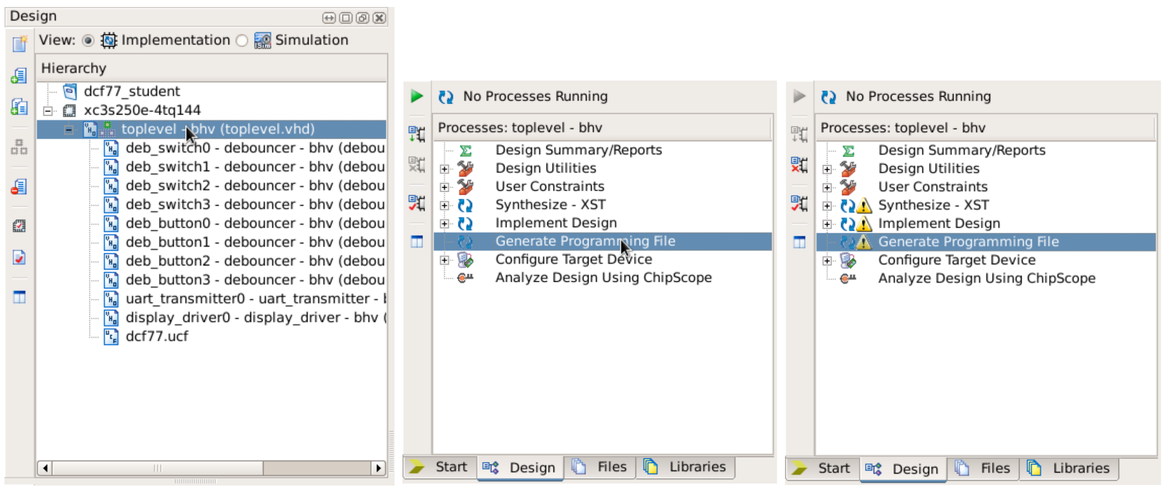

Design
View: Implementation
Simulation
Behavioral
Hierarchy
dcf77_student
xc3s250e-4tq144
debouncer_tb - bhv (debouncer_tb.vhd)
debouncer0 - debouncer - bhv (debour
toplevel - bhv (toplevel.vhd)
No Processes Running
Processes: debouncer_tb - bhv
ISim Simulator
Behavioral Check Syntax
Simulate Behavioral Model

3. The ISim window will open and the simulation will start.

Implementing your design

The following steps must be followed to get your VHDL source code running on the FPGA:

1. Make sure that the view is set to "Implementation", and select your top-level VHDL module.
2. Double-click "Generate Programming File".
3. Wait until the process completes. Make sure there are no errors (warnings are normal).

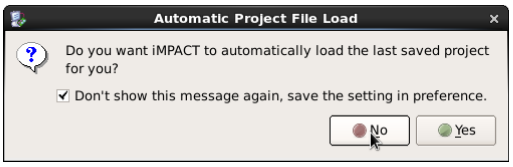

Design
View: Implementation
Simulation
Hierarchy
dcf77_student
xc3s250e-4tq144
toplevel - bhv (toplevel.vhd)
deb_switch0 - debouncer - bhv (debou
deb_switch1 - debouncer - bhv (debou
deb_switch2 - debouncer - bhv (debou
deb_switch3 - debouncer - bhv (debou
deb_button0 - debouncer - bhv (debou
deb_button1 - debouncer - bhv (debou
deb_button2 - debouncer - bhv (debou
deb_button3 - debouncer - bhv (debou
uart_transmitter0 - uart_transmitter - b
display_driver0 - display_driver - bhv (
dcf77.ucf
No Processes Running
Processes: toplevel - bhv
Design Summary/Reports
Design Utilities
User Constraints
Synthesize - XST
Implement Design
Generate Programming File
Configure Target Device
Analyze Design Using ChipScope
Start
Design
Files
Libraries
No Processes Running
Processes: toplevel - bhv
Design Summary/Reports
Design Utilities
User Constraints
Synthesize - XST
Implement Design
Generate Programming File
Configure Target Device
Analyze Design Using ChipScope
Start
Design
Files
Libraries

4. Start the Impact tool from a terminal:

```
$ /esat/micas-data/data/design/i3t50/gmsk100-2020/scripts/start-impact.sh
```

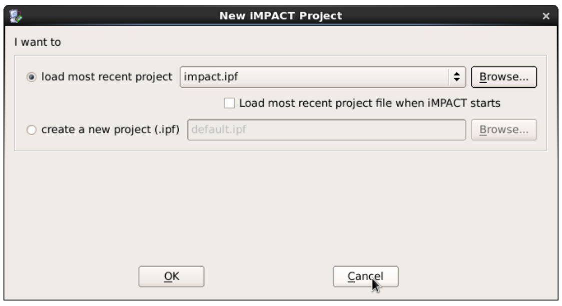
Do not start Impact from within ISE! Doing so will likely result in problems with the FPGA programmer connection.

5. If Impact asks you to load the last saved project, click “No”. If Impact asks you to load a different project or create a new project, click “Cancel”.

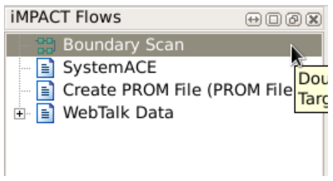


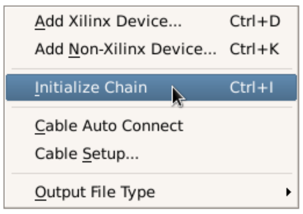
Do you want iMPACT to automatically load the last saved project for you?

☑ Don't show this message again, save the setting in preference.

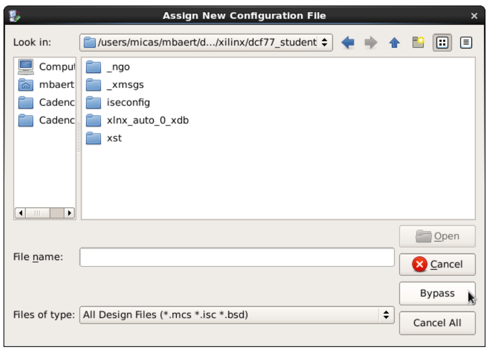


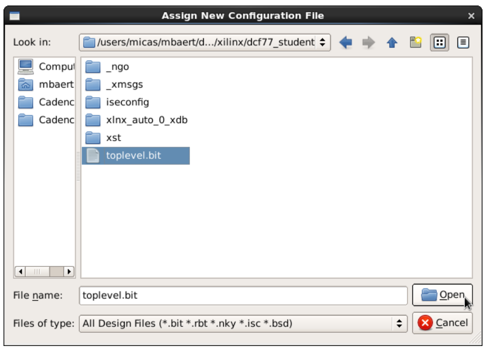
6. Make sure that the FPGA programmer is properly connected to the computer and the FPGA, and that the FPGA is powered. The status light on the programmer cable should be green.
7. Double-click “Boundary Scan”.
8. Right-click inside the large white area and click “Initialize Chain”. The chain will show two chips: the first one is the flash chip (xcf02s), the second one is the FPGA (xc3s250e).



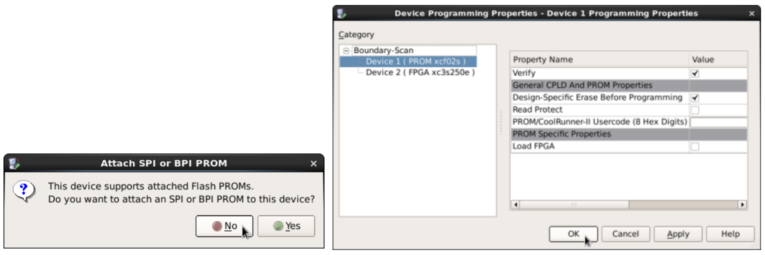


9. The “Auto assign configuration file(s)” will appear. Click “Yes” to proceed.
10. The **first** dialog asks for a PROM file which you don’t have at this point, so click “Bypass” to skip it. The **second** dialog asks for a bitstream file, so select the “toplevel.bit” file in your project directory.





11. The “Attach SPI or BPI PROM” dialog will appear. Click “No”.
12. The “Device Programming Properties” dialog will appear. Click “OK”.


Attach SPI or BPI PROM

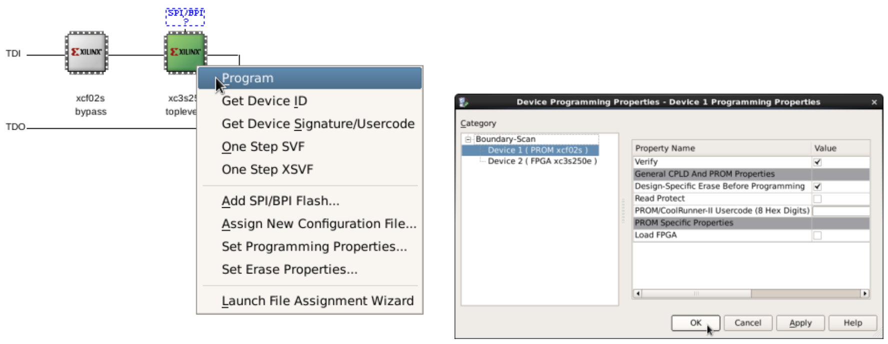
This device supports attached Flash PROMs.
Do you want to attach an SPI or BPI PROM to this device?

Device Programming Properties - Device 1 Programming Properties

Category
Boundary-Scan
Device 1 (PROM xcf02s)
Device 2 (FPGA xc3s250e)

Property Name	Value
Verify	✓
General CPLD And PROM Properties	
Design-Specific Erase Before Programming	✓
Read Protect	☐
PROM/CoolRunner-II Usercode (8 Hex Digits)	
PROM Specific Properties	
Load FPGA	☐

13. Right-click the second chip and click “Program”.
14. The “Device Programming Properties” dialog will appear again. Click “OK”.



Device Programming Properties - Device 1 Programming Properties

Category
Boundary-Scan
Device 1 (PROM xcf02s)
Device 2 (FPGA xc3s250e)

Property Name	Value
Verify	✓
General CPLD And PROM Properties	
Design-Specific Erase Before Programming	✓
Read Protect	☐
PROM/CoolRunner-II Usercode (8 Hex Digits)	
PROM Specific Properties	
Load FPGA	☐

15. You should see the “Program succeeded” message. Your FPGA is now running the new design.
16. If you keep the Impact window open, you can program an updated bitstream file simply by clicking “Program”. If you want to pick a different bitstream file, you can do so by double-clicking the second chip.
17. If you are feeling adventurous, you can even try saving your Impact project so you can skip most of these steps next time. Sometimes this works, sometimes not.

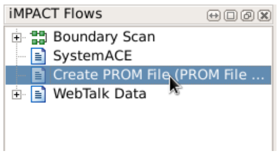
Impact may stop working if you unplug the programmer, or even if you just restart the program. These problems can usually be resolved by restarting Impact again, double-clicking “Boundary Scan”, and then clicking “Output” > “Cable Reset” followed by “Output” > “Cable Auto Connect”. If this still doesn’t solve your problem, then close Impact, plug the programmer into a different USB port, and go through the same procedure again.

During synthesis and implementation, the Xilinx tools will usually produce lots of warnings. This is considered normal. In some situations it can be useful to read the warnings to find bugs in the code, but you should not attempt to eliminate all warnings, since most warnings are in fact harmless. The same is not true for errors: a design with errors will simply not work.

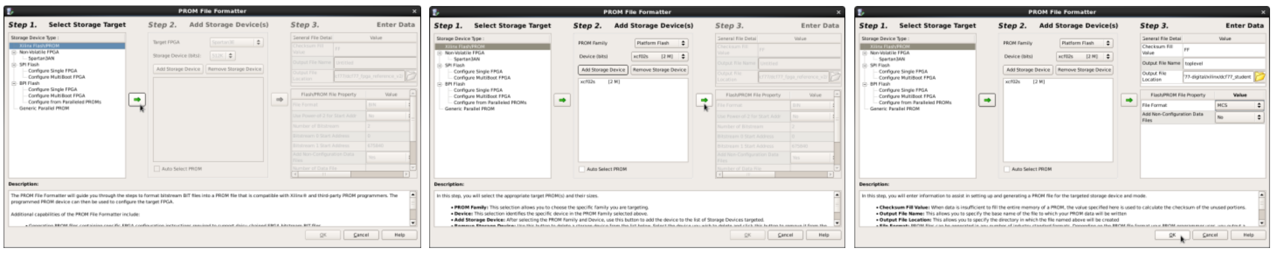
Writing your configuration to flash memory

The FPGA configuration memory is volatile: as soon as the FPGA is powered off, the configuration is lost. If you want to move the FPGA without keeping it powered (e.g. for the demo), you should write the configuration to the flash memory on the FPGA board. This can be done by following these steps:

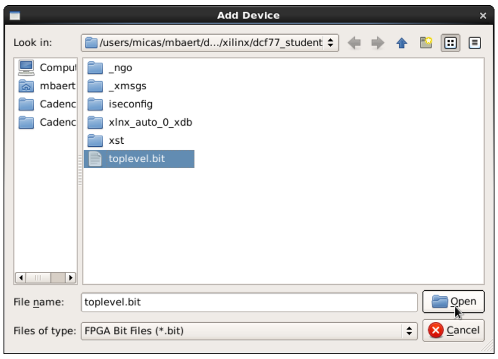
1. In Impact, double-click “Create PROM File”.

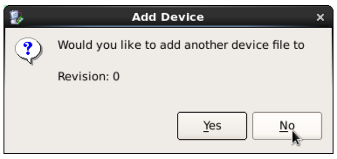


2. Select “Xilinx Flash/PROM” and click the arrow button.
3. Set the device to “xcf02s”, click “Add Storage Device”, and click the second arrow button.
4. Set the file name to “toplevel” and change the location to your project directory, then click “OK”.



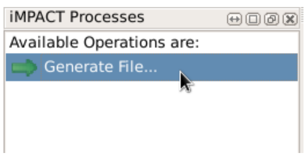
5. Impact will ask you to select the bitstream file. Select the “toplevel.bit” file in your project directory.
6. When Impact asks you whether you want to add another file, click “No”.
7. Double-click “Generate File”. You should see the “Generate Succeeded” message.



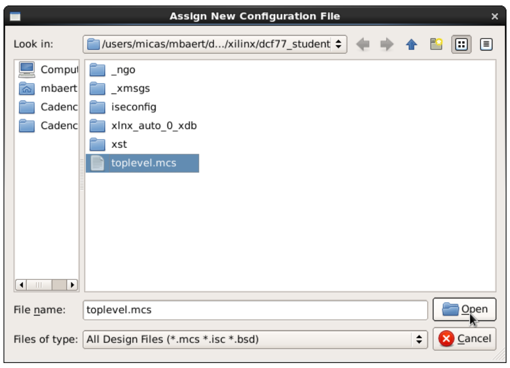


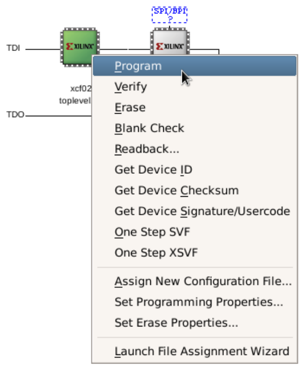
Would you like to add another device file to

Revision: 0



8. Click “Boundary Scan” again, and double-click the **first** chip.
9. Select the “toplevel.mcs” file.
10. Right-click the **first** chip and click “Program”. This step will take some time.





11. Eventually you should see the “Program Succeeded” message. The configuration is now stored on the flash chip and will be loaded into the FPGA the next time it is powered on.
12. If you change the bitstream file, you can update the PROM file by double-clicking “Generate File” again, and then programming the flash chip again.

VHDL PROGRAMMING

VHDL data types

A physical digital circuit knows only one data type: the 'bit'. However, VHDL is a high-level language which allows you to use more convenient high-level data types. The synthesis tool will convert all of these types to bits, so you don't need to worry about this. In order to use VHDL correctly, it is important that you understand the meaning of the different data types. The following sections describe the most commonly used data types and their function.

std_logic

This data type is the closest equivalent of a physical 'bit'. However, unlike a traditional bit, this data type supports more values than just 0 and 1. VHDL uses '9-valued logic':

'U'	uninitialized
'X'	strong drive, unknown logic value
'0'	strong drive, logic zero
'1'	strong drive, logic one
'Z'	high impedance
'W'	weak drive, unknown logic value
'L'	weak drive, logic zero
'H'	weak drive, logic one
'-'	don't care

In most cases, you will only use the values 0 and 1 explicitly, but some other values may appear in simulations: the value U indicates that a signal has not been initialized, the value X indicates that a signal is driven by two different outputs, and the value Z indicates that a signal is not driven at all. `std_logic` constants are written with single quotes, such as '0' or '1'. The following example shows the definition of a `std_logic` signal:

```
signal data_valid: std_logic := '0';
```

std_logic_vector

This type stores a vector of `std_logic` values. It is commonly used for data buses and other uses where more than one bit is needed, but the value should not be interpreted as a number. `std_logic_vector` constants are written with double quotes, such as "0010", which represents the `std_logic` values '0', '0', '1' and '0' (in that order).
When a signal is defined as a `std_logic_vector`, the number of bits must be specified using a range, either as 'A to B' or 'B downto A' (always with A < B). The type of range ('to'/'downto') is important, since it indicates in what order the bits will be addressed. It is common practice to use 'downto' for `std_logic_vector`. The following example shows the definition of an 8-bit data bus:

```
signal data_bus: std_logic_vector(7 downto 0) := "00000000";
```

signed and unsigned

These types are similar to `std_logic_vector`, except that the bits stored by these types represent a signed or unsigned integer, respectively. The main difference between these types and `std_logic_vector` is that it is possible to use these types in mathematical operations, such as addition or multiplication. As with `std_logic_vector`, the number of bits must be specified explicitly with a range. Again, it is common practice to use 'downto' when defining these types. The following example shows the definition of an 8-bit `unsigned` counter:

```
signal counter: unsigned(7 downto 0) := to_unsigned(0, 8);
```

integer

The `integer` type represents integers in a more abstract way than the `signed` and `unsigned` types. It is not possible to specify the number of bits that will be used for an integer – this depends on the synthesis tools (32 bits is a common value). For this reason you should try to use `signed` and `unsigned` instead whenever possible. There are a few cases where integers are useful though:

- Numeric constants in VHDL code (such as 1234) are integers.
- The index (address) of a register bank or memory is an integer.
- The temporary value used by a 'for' loop is an integer.

boolean

The `boolean` type has two possible values: true or false. It appears whenever two values are compared for equality or inequality. For example, the expression a = '1' will produce a `boolean` value. The condition of an 'if' statement must always be a `boolean`.

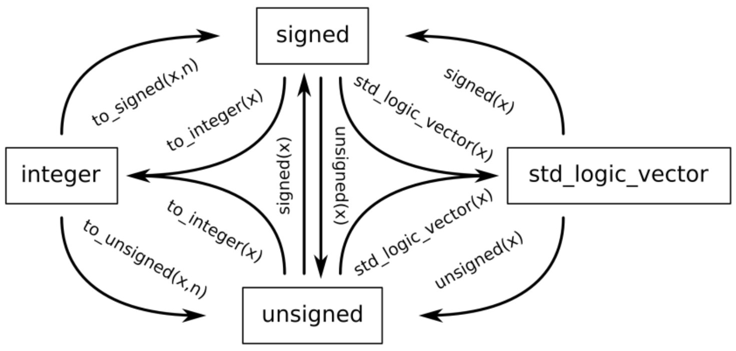
Libraries

VHDL provides a number of standard libraries which contain commonly used features. Most of the data types from the previous section are in fact not part of the VHDL core language, but defined in the IEEE library. In order to use them, the following lines must be added to the VHDL file:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

Type conversion

VHDL is a strongly typed language, which means that different data types are treated differently and conversions must be done explicitly. The following figure shows the most common conversions:



Conversions involving an `integer` require that you specify the desired number of bits. For all other conversions, the number of bits is kept the same.

Sometimes it is necessary to change the number of bits of `signed` and `unsigned` values (for example, to avoid integer overflow). This can be done with the `resize(x,n)` function.

VHDL modules

It is possible to create your entire digital circuit with just one VHDL file, but the result will likely be very complex and unreadable. In order to make the design more manageable, it is usually split into multiple modules. A module is a user-defined building block with a number of inputs and outputs. Multiple modules can be connected together to create the complete circuit.

In VHDL, a module definition is split into two parts: the ‘entity’, which defines the inputs and outputs of the module, and the ‘architecture’, which describes the actual behaviour of the module (i.e. what is inside).

An entity is defined like this:

```
entity entityname is
    port(
        inputname: in inputtype;
        inputname: in inputtype;
        inputname: in inputtype;
        outputname: out outputtype;
        outputname: out outputtype;
        outputname: out outputtype
    );
end entityname;
```

The architecture of the entity is defined like this:

```
architecture bhv of entityname is
    signal definitions
begin
    module instances
    connections and asynchronous logic
    process(clk)
        variable definitions
    begin
        if clk'event and clk = '1' then
            synchronous logic
        end if;
    end process;
end bhv;
```

Once this module has been defined, it can be instantiated in other modules. Some textbooks will instruct you to do this by first defining a component and then creating instances of it, like this:

```
component example is
    port(
        clk: in std_logic;
        reset: in std_logic;
        output: out unsigned(7 downto 0)
    );
end component;

example1: example port map(
    clk => global_clk,
    reset => global_reset,
    output => output1
);
example2: example port map(
    clk => global_clk,
    reset => global_reset,
    output => output2
);
```

This method works, but it is unnecessarily complex. It is much easier to instantiate the entity directly, like this:

```
example1: entity work.example port map(
    clk => global_clk,
    reset => global_reset,
    output => output1
);
example2: entity work.example port map(
    clk => global_clk,
    reset => global_reset,
    output => output2
);
```

The name 'work' is a VHDL keyword that means 'the current library'. The advantage of this method is that you don't need to update the component definition if you decide to add a new input or output to your module.

Signals and variables

VHDL makes a distinction between signals and variables. Signals are physical wires (and in some cases registers) whereas variables are just temporary values that are used to make the code more readable.

Signals are declared as part of the architecture:

```
architecture bhv of example is
    signal counter: unsigned(7 downto 0);
begin
```

Variables on the other hand are declared as part of a process:

```
process(clk, reset)
    variable counter: unsigned(7 downto 0);
begin
```

Another important difference is that signal assignments use the '<=' operator (which only takes effect at the end of the process) whereas variable assignments use the ':=' operator (which takes effect immediately).

Processes

The actual behaviour of a module is described as a process, which is part of the architecture. The statements within a process are executed sequentially, but the results will not be visible until the end of the process. The most common use of processes is to describe synchronous logic. Such a process looks like this:

```
process(clk)
    variable definitions
begin
    if clk'event and clk = '1' then
        synchronous logic
    end if;
end process;
```

The line `process(clk)` marks the start of the process and also defines the 'sensitivity list'. This is a list of all signals which can trigger a change in the outputs of the process. Since this process describes synchronous logic, the only signal that can trigger a change is the clock signal.

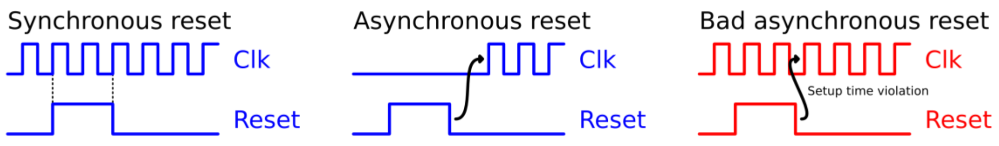
The line `if clk'event and clk = '1' then` indicates that the process is triggered by the rising edge of the clock. All statements between `then` and `end if;` are executed on the rising clock edge, and the results will be visible once this clock event has passed. In reality, the statements will be translated to digital logic on the FPGA which obviously will not respond instantly, but since the results will always be ready before the next rising edge of the clock, this won't change the behavior of the circuit.

The reset signal (or lack thereof)

Digital ASICs need a global reset signal to ensure that all flip-flops start up in a known state. Without a reset signal, each flip-flop would have a random, unpredictable value right after the power is applied to the chip. For this reason, many VHDL textbooks will tell you to add a reset input to every single VHDL module, without exception. However, this is unnecessary and often undesirable when working with FPGAs. When an FPGA is configured by loading a bitstream file into the configuration memory, every flip-flop, lookup table and block RAM inside the FPGA is initialized to the value encoded in the bitstream file. The state of all flip-flops in the FPGA is well-defined and known from the start, so there is no need for a reset signal. It is still possible to add a global reset signal, but this adds unnecessary complexity to the design and will use up valuable FPGA resources (the reset circuit uses logic cells that could be better used for other things). So in most cases, you are better off without a reset signal.

Now, if for whatever reason you *really* want to add a reset signal anyway, it is important that you do it properly. There are two types of resets: synchronous and asynchronous. A synchronous reset signal is synchronized with the clock, and is the easiest to implement in an FPGA. Since the external reset signal is generally not synchronized with the internal clock, the reset signal should first be passed through at least two flip-flops to eliminate metastability before it is actually used.

Many people seem to think that an asynchronous reset is simply a reset signal that does not have to be synchronized with the clock, but this is wrong! An asynchronous reset must be activated **before the clock is started**. If an asynchronous reset is activated while the clock is already running, this can lead to races and setup/hold time violations that lead to metastability. Even though such a flawed reset circuit will often appear to work correctly, it is unreliable and should not be used.



The following code examples show the same VHDL module implemented without reset, with synchronous reset, and with asynchronous reset.

No reset

```
entity example is
     port(
          clk: in std_logic;
          output: out unsigned(7 downto 0)
     );
end example;

architecture bhv of example is
     signal counter: unsigned(7 downto 0) := to_unsigned(0, 8);
begin
     output <= counter;
     process(clk)
     begin
```

```
        if clk'event and clk = '1' then
            counter <= counter + 1;
        end if;
    end process;
end bhv;
```

Synchronous reset

```
entity example is
    port(
        clk: in std_logic;
        reset: in std_logic;
        output: out unsigned(7 downto 0)
    );
end example;

architecture bhv of example is
    signal counter: unsigned(7 downto 0);
begin
    output <= counter;
    process(clk)
    begin
        if clk'event and clk = '1' then
            if reset = '1' then
                counter <= to_unsigned(0, 8);
            else
                counter <= counter + 1;
            end if;
        end if;
    end process;
end bhv;
```

Asynchronous reset

```
entity example is
    port(
        clk: in std_logic;
        reset: in std_logic;
        output: out unsigned(7 downto 0)
    );
end example;

-- Important: The clock *must* be disabled externally (and without creating glitches)
-- when reset is high. If you are not sure how this is done, don't attempt to use this.

architecture bhv of example is
    signal counter: unsigned(7 downto 0);
begin
    output <= counter;
    process(clk, reset)
    begin
        if reset = '1' then
            counter <= to_unsigned(0, 8);
        elsif clk'event and clk = '1' then
            counter <= counter + 1;
        end if;
    end process;
end bhv;
```

Synthesizable versus unsynthesizable code

The VHDL language contains certain features which can not be translated to physical hardware. The most common example is the `wait for` statement. These statements can be very useful for simulation purposes, but should only be used in testbenches. If you try to use such statements in the actual hardware implementation, the synthesis step will fail. This is why such code is called ‘unsynthesizable’. So far this document has focused only on synthesizable code, however some unsynthesizable features will be used in testbenches.

Testbenches

The purpose of a VHDL testbench is to test another module to make sure that it works correctly. A testbench uses VHDL code just like a normal module. The only real difference is that this module does not have any inputs or outputs, instead it generates all the necessary signals internally. This is done with one or more processes that run continuously. For example, the following process can be used to create a clock:

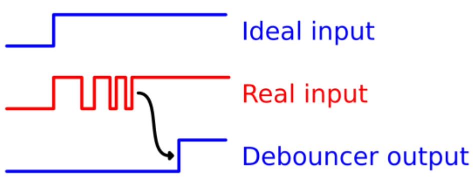
```
process
begin
    clk <= '0';
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
end process;
```

The `wait for` statement will pause the process for some time. This process will produce a clock which is 0 for 10 ns, then 1 for 10 ns, then 0 again for 10 ns, and so on. The process runs indefinitely – when it reaches the end, it restarts from the beginning.

It is possible to generate all test signals with a single process, but it is often easier to use multiple processes to generate different signals. For example, it is often convenient to use one process to generate the clock and another one to generate the data.

A simple VHDL module

We will implement a simple VHDL module as an example. The module is a simple circuit which is known as a ‘debouncer’. The purpose of a debouncer is to eliminate ‘contact bounce’ in mechanical buttons. Contact bounce is a physical phenomenon caused by oscillation of the metal contacts inside a button: when the button is pressed, the metal contacts will touch, but they will often bounce back due to their inertia. This repeats for a few cycles until the oscillation disappears. This phenomenon creates a digital signal with glitches instead of the clean zero-to-one transition that we want. A debouncer solves this problem by ignoring input changes until the input remains stable at a new value for a certain amount of time.



In this case, we will create a debouncer that waits for 2^20 cycles. Since the FPGA used in this project runs at 20 MHz, this corresponds to approximately 52 ms. The debouncer will add some latency to the output signal, but this is not an issue for human-controlled buttons (a delay of 52 ms is barely noticeable).

The complete VHDL file for the module looks like this:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity debouncer is
    port(
        clk: in std_logic;
        input: in std_logic;
        output: out std_logic
    );
end debouncer;

architecture bhv of debouncer is

    signal counter: unsigned(19 downto 0) := to_unsigned(0, 20);
    signal reg: std_logic := '0';

begin

    output <= reg;

    process(clk)
    begin
        if clk'event and clk = '1' then
            if input = reg then
                counter <= to_unsigned(0, 20);
            elsif counter = (19 downto 0 => '1') then
                counter <= to_unsigned(0, 20);
                reg <= input;
            else
                counter <= counter + 1;
            end if;
        end if;
    end process;

end bhv;
```

A simple VHDL testbench

Now that we have created a simple module, it's time to write a testbench to verify that the module actually does what we want. We will use one process to generate the clock, and another one to generate the input data. The input will contain a few glitches and a short interruption to make sure that the debouncer handles these cases correctly.

The complete VHDL file for the testbench looks like this:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity debouncer_tb is
end debouncer_tb;

architecture bhv of debouncer_tb is

    signal clk: std_logic := '0';
    signal input: std_logic := '0';
    signal output: std_logic;

    constant clk_period: time := 50 ns;

begin

    debouncer0: entity work.debouncer port map(
        clk => clk,
        input => input,
        output => output
    );

    -- clock process
    process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    -- stimulus process
    process
    begin

        -- start
        wait for 30 ms;

        -- low to high with contact bounce
        input <= '1';
        wait for 5 ms;
        input <= '0';
        wait for 1 ms;
        input <= '1';
        wait for 3 ms;
        input <= '0';
        wait for 1 ms;
        input <= '1';
        wait for 100 ms;

        -- short interruption
        input <= '0';
        wait for 10 ms;
        input <= '1';
        wait for 100 ms;

        -- high to low with contact bounce
        input <= '0';
        wait for 3 ms;
        input <= '1';
        wait for 2 ms;
        input <= '0';
        wait for 6 ms;
        input <= '1';
        wait for 2 ms;
        input <= '0';
        wait for 100 ms;
```

```
         -- end
         wait for 1000 ms;

    end process;

end;
```

THE UART DEBUG INTERFACE

Not all problems can be found by simulation, especially when interaction with analog circuits is involved. During this project, you will likely run into problems that only happen when your code runs on the actual FPGA, and not during simulation. In order to debug these types of problems more easily, we have implemented a simple UART debug interface. This interface allows you to send data from the FPGA to the PC, and plot the results. It is up to you to decide what data you want to send – the interface doesn't specify this.

The protocol

UART is a very simple serial protocol. It simply transfers a stream of bytes without any special formatting. It is up to the application to choose how the data is represented. For the debug interface, we chose a very simple text-based format. The byte stream looks like this:

```
1 22 3333
4 55 6666
7 88 9999
a bb cccc
d ee ffff
```

Each line in the byte stream represents one time point. Each line is split into multiple words, and each word corresponds to a different data channel. Each data value is encoded as a hexadecimal number. In this example, the first channel is transmitting the data `[1, 4, 7, 10, 13]`. The number of hexadecimal digits used to represent a number can be chosen freely.

Transmitting data

Transmitting data is done with the "uart_transmitter" VHDL module which is part of the example project. The example also shows you how to use the module.

The UART interface is configured to use a baud rate of 1Mbit/s. Since the UART protocol uses 10 bits to transfer one byte (start bit, 8 data bits, stop bit), this means you can transfer at most 100kB/s. Due to further imperfections, the actual data rate will be slightly lower. If you try to transfer more than 90kB/s, you risk losing data. It is your responsibility to ensure that you stay below this limit.

Receiving data

In order to receive and display the data, you can use the "uart_plotter.py" script in the "uart" directory. This is a Python script which uses a framework called 'pylab'. First go to the "uart" directory using the "cd" command, then start pylab with the following command:

```
$ /esat/micas-data/data/design/i3t50/gmsk100-2020/scripts/start-pylab.sh
```

In pylab, you can run the script by typing:

```
> run uart_plotter.py
```

The script will receive data from the FPGA and then show a plot of the collected data. The data is also saved to the file "uart_data.mat". This file can be loaded into Pylab or Matlab for further analysis, if this is necessary.

The “uart_plotter.py” script is configured to receive data from the example project. If you modify the example so it sends different data, you should modify the “uart_plotter.py” script accordingly. The following lines at the top of the file are used to configure how the data should be interpreted:

```
# data configuration
samplerate = 10
samples = samplerate * 5
names = ["Counter", "Switches/buttons", "Test value"]
```

- “samplerate” should match the rate at which the sampled data is transmitted inside the FPGA, otherwise the horizontal time scale will be incorrect.
- “samples” determines how much data the script will collect before saving and displaying it.
- “names” determines the names of the data channels. These are used to label the subplots.

EXERCISES

In order to familiarize yourself with the digital toolchain, complete the following exercises:

1. Try out the instructions given in the section ‘Working with Xilinx tools’ to make sure that all tools are working properly and the files have been installed correctly on your user account.
2. Run the example project on the FPGA and test the buttons and switches to verify that it is working correctly.
3. Run the UART plotter script as described in the section ‘The UART debug interface’ to verify that the debug interface is working. Try to press the buttons while the interface is running to see how this affects the output. Does the result match what you expect?
4. Change the size of ‘counter2’ from 4 bit to 16 bit. Also change the code that transmits the value of ‘counter2’ through the UART debug interface so that it will transmit all 16 bits.
5. Display the value of ‘counter2’ on the display, in hexadecimal format.
6. In the example, ‘counter2’ is incremented by one every 100 ms, or in other words, it is counting at a frequency of 10 Hz. Change this to 50 Hz.
7. Add a new 16 bit register ‘counter3’, increment it by one when button 0 is pressed, and decrement it by one when button 1 is pressed. Transmit this new counter through the UART interface, and update the UART plotter script to add the correct label.
8. When switch 0 is set to ‘1’, let the display show the value of ‘counter3’ instead of ‘counter2’.
9. Add code that prevents ‘counter3’ from overflowing or underflowing.
10. Change the display code so that when button 2 is pressed, the names of the team members are shown on the display, by making the names scroll through one character at a time. Once all names have been displayed, the display reverts to normal operation. Try to do this with simple code and minimal hardware usage. A huge finite state machine is most likely neither the simplest nor the most efficient implementation.

PROTOTYPE BOARD FOR THE GMSK RECEIVER

February 2020

GMSK PROTOTYPE BOARD

Maarten Baert (maarten.baert@esat.kuleuven.be)

Thomas Bos (thomas.bos@esat.kuleuven.be)

Umut Celik (umut.celik@esat.kuleuven.be)

Carl D'Heer (carl.dheer@esat.kuleuven.be

Kaizhe Guo (kaizhe.guo@esat.kuleuven.be)

Pouya Housmand (pouya.houshmand@esat.kuleuven.be)

Yifan Lyu (yifan.lyu@esat.kuleuven.be)

Prof. Wim Dehaene (wim.dehaene@esat.kuleuven.be)

Prof. Michiel Steyaert (michiel.steyaert@esat.kuleuven.be)

Prof. Patrick Reynaert (patrick.reynaert@esat.kuleuven.be)

Prof. Filip Tavernier (filip.tavernier@esat.kuleuven.be)

Prototype Board for the GMSK Receiver

PROVIDED PROTOTYPE BOARD DESCRIPTION

INTRODUCTION

You will effectively test your design, thereby using the prototype board provided by the TAs. This setup allows you to generate a GMSK signal of increasing difficulty level, mimic your analog front end (within the freedom provided by the boards), and implement your digital code in a Spartan3 FPGA. This way, your digital code performance and implementation efficiency is tested.

This chapter provides you with all details of the prototype board.

Contents

THE TEST SETUP

The aim of this project is to receive and demodulate the signal on the 20 kHz carrier. In order to simplify experimentation, the test setup is split into multiple boards, each of which has a specific function:

- Generator board: Uses a microcontroller to generate an artificial GMSK100 signal.
- AFE board: Contains a variable gain amplifier and 6 comparators.
- FPGA board: Contains the FPGA which will be used to implement the decoding algorithm.
- Display board: Contains a display which can be used to show up to 4 digits.

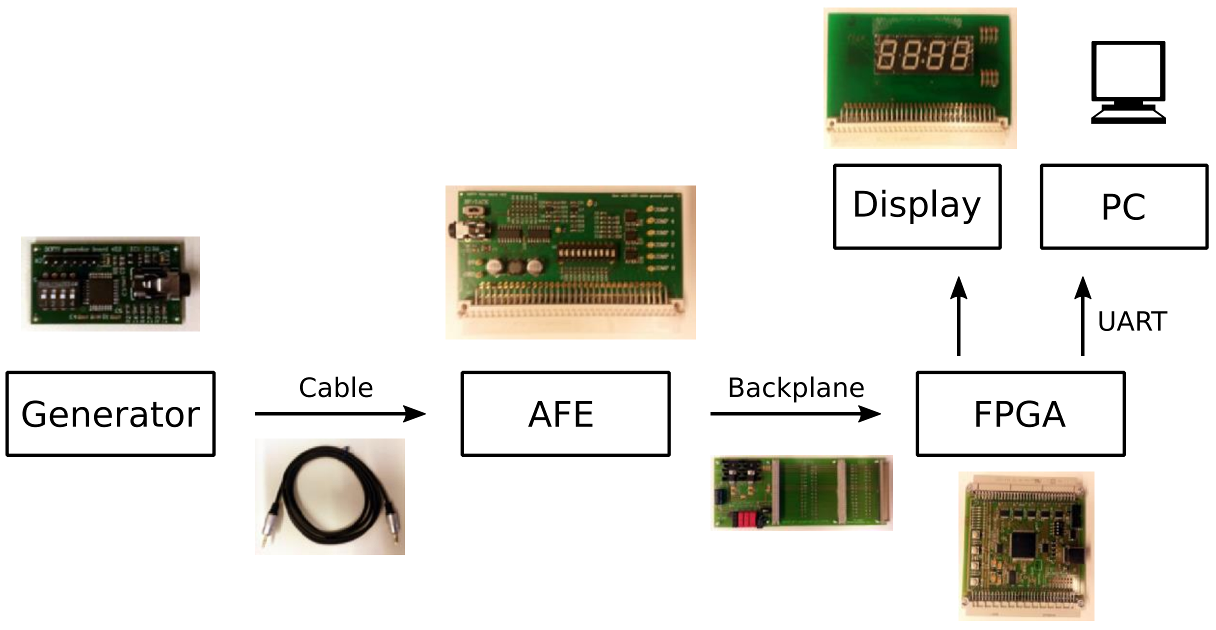


FIGURE 1: OVERVIEW OF THE DIFFERENT COMPONENTS OF THE TEST SETUP

Schematics for all boards can be found at the end of this manual.

Please realize that these boards are not indestructible. They must be handled with care to avoid damage. Therefore you should follow these guidelines whenever working with the boards:

- **Never hot-plug boards!** Always disconnect the power source before making changes.
- **Make sure there are no metal objects below the PCBs!** These could easily create short circuits and damage components. Keep the table you are working on clear of any loose wires.
- **Never connect the ground leads of scope probes to anything other than ground!** The ground connection of the oscilloscope is connected to mains earth, and the same is true for the ground connection of the USB ports of a computer, as well as any USB device plugged into it (such as the Tiepie scope, the FPGA programmer, or the USB connector on the FPGA board). If you connect the ground lead to anything other than ground, you will create a short circuit! A side effect of this is that the probes will continue to work even when the ground lead is not connected, but relying on this is not recommended because it will result in a lot of extra noise.
- **Do not drop the boards, especially those with crystals!** The crystals can break internally and won't perform properly anymore.

BACKPLANE

The backplane is used to connect multiple boards together, as well as to provide power to the boards. The backplane has multiple power sources, some analog and some digital. For practical reasons, the boards derive all power rails from the digital 5V rail, even for analog circuits (the boards contain supply filters where necessary). This means you only need to connect the 12V digital supply (barrel connector), the analog supplies (banana plugs) are unnecessary.

GENERATOR BOARD

The generator board uses a microcontroller with a built-in digital-to-analog converter to generate an analog waveform which generates a realistic IF GMSK signal under various conditions. The generator board contains a DIP switch which is used to change the difficulty level. The difficulty level is a 4-bit value. The leftmost switch (labelled '1') is the **least** significant bit. For example, the setting '1100' corresponds to level 3, and '0001' corresponds to level 8.

The lowest difficulty level (0) produces an ideal signal. Higher difficulty levels have a progressively lower signal to noise ratio, as well as different amplitudes and carrier frequencies. The highest difficulty level (15) contains so much noise that decoding is very difficult and sometimes not even possible. The SNR corresponding to each level is:

Level 0	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7
60.00 dB	50.26 dB	42.17 dB	35.46 dB	29.87 dB	25.19 dB	21.22 dB	17.80 dB

Level 8	Level 9	Level 10	Level 11	Level 12	Level 13	Level 14	Level 15
14.79 dB	12.10 dB	9.63 dB	7.34 dB	5.22 dB	3.25 dB	1.49 dB	0.00 dB

For levels 0 to 7, the digital data is transmitted without forward error correction. For levels 8 to 15, forward error correction is enabled.

AFE BOARD

The AFE board contains a variable gain amplifier and 6 comparators. The variable gain amplifier can be controlled digitally from the FPGA, and the outputs of the comparators are sent to the FPGA. The purpose of the variable gain amplifier is to keep the output amplitude consistent regardless of input amplitude. Ideally, your system should continue to work even when the input amplitude changes over time, provided that the change happens slowly. The digital gain value ranges from 0 to 63. The actual analog gain can be calculated with the following formula:

$$A = 100 \cdot \frac{gain + 1}{64}$$

The highest possible analog gain is 100. When the input amplitude is 10 mV, the output amplitude will be 1V.

This board can take its input from two sources: the backplane ('BP'), or the 3.5mm jack ('JACK'). You can use the switch right above the 3.5mm jack to select the input source. The backplane option is provided for backward compatibility and is never necessary when using the current boards.

The 3.5mm jack also provides power (at 5V) to the antenna or generator board. Note that even though this connector looks like a headphone jack, **it is a really bad idea to connect headphones to it**. Doing so will likely damage the headphones and/or the AFE board.

The AFE board contains a tri-state DIP switch that can be used for configuration. The first 6 switches (labelled '1' to '6') control the gain ('1' is the **least** significant bit). The remaining 4 switches control the comparator thresholds.

Gain: If the switch is set in the center position, that bit is floating (we call this 'Z') which means that the FPGA can set this bit. However, if the switch is set low or high, it will produce a '0' or '1' respectively, overriding the value from the FPGA. If the FPGA is not plugged in and a switch is in the center position ('Z'), the pin is left completely floating and the value is undefined. You should always avoid this.

Comparator thresholds: Depending on the setting of the last 4 switches, different thresholds can be chosen. The recommended setting is '1Z0Z'.

Switches	Comp 0	Comp 1	Comp 2	Comp 3	Comp 4	Comp 5
'1Z0Z'	-0.5 V	-0.3 V	-0.1 V	0.1 V	0.3 V	0.5 V
'Z1Z0'	0.0 V	0.2 V	0.4 V	0.6 V	0.8 V	1.0 V

FPGA BOARD

This board contains the FPGA as well as 8 LEDs, 4 buttons and 4 switches that can be used for testing purposes. One side of the FPGA board must be plugged into the backplane (in the slot closest to the power connectors), the other side connects to the display board. Detailed instructions on how to use the FPGA are given in the digital flow manual.

DISPLAY BOARD

This board must be plugged into the FPGA – **not the backplane**!