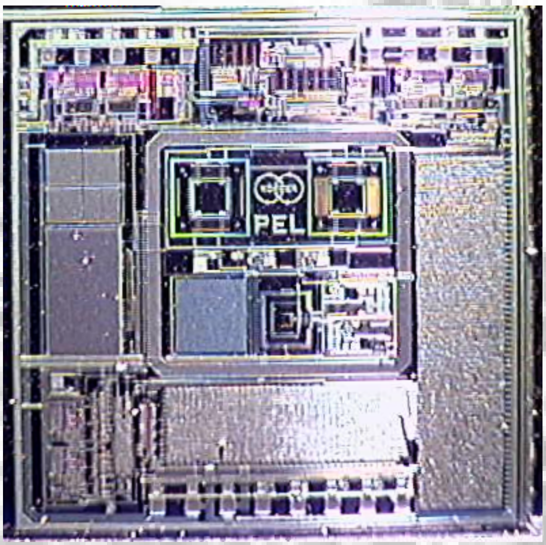
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A single chip micro-nose



Diploma Thesis
Christian Herzog
March 2001

ETH Zurich / Physical Electronics Lab

Head: Prof. Dr. Henry Baltes
Supervisor: Christoph Hagleitner

1 INTRODUCTION

1.1 OVERVIEW

In the project *A single chip micro nose*, an integrated gas sensing system realized in CMOS technology is being developed. Using appropriate polymers, this sensor can determine the concentration of a specific or some similar gases in an atmosphere. In order to get information about the composition of a gas mixture, several sensor chips using different polymers are linked together, and their output is processed by a micro controller.

1.2 THIS THESIS

The scope of this diploma thesis is the following: to give a description of the different parts (sensors, analog and digital part) of the micronose chip, to establish the communication sensor-µC and µC-PC, and, most importantly, to develop a test setup which enables us to discriminate working and defunct chips on the wafer level. Both the analog and digital side of the circuit should be tested. Furthermore, test the chips for overall functionality in a gas measurement setup.

1.3 THE PROJECT

The *single chip micro nose* is a joint project of the *Physical Electronics Lab* at the *Institute of Quantum Electronics*, *ETH Zurich*, Switzerland (*PEL*), the *Department of Electronics* (*DEIS*), *University of Bologna*, Italy, and the *Institut fuer physikalische Chemie (IPC)*, *University of Tuebingen*, Germany. The project is funded by the *Koerber Foundation* and should provide a working prototype in March 2001. The final goal is to have a battery powered handheld device that gives the concentrations of different gases in the atmosphere. The device contains six sensor chips or Standard Micronose Chips (SMCs) and a micro controller for data processing and the user interface. Each SMC is coated with a chemically sensitive polymer that exhibits changes in its chemical or physical properties depending on the concentration of one or several analytes. These changes are then measured exploiting three different sensor principles on each chip. A temperature sensor has also been included for thermal compensation. The output of these 24 sensors is preprocessed on-chip and is then sent via a serial interface to a µC which does the final processing and displays the results. The six SMCs are mounted on a ceramic substrate. Two different package types were produced. The first version (Fig. 1.1) uses flip-chip bonding to connect the six chips and fix

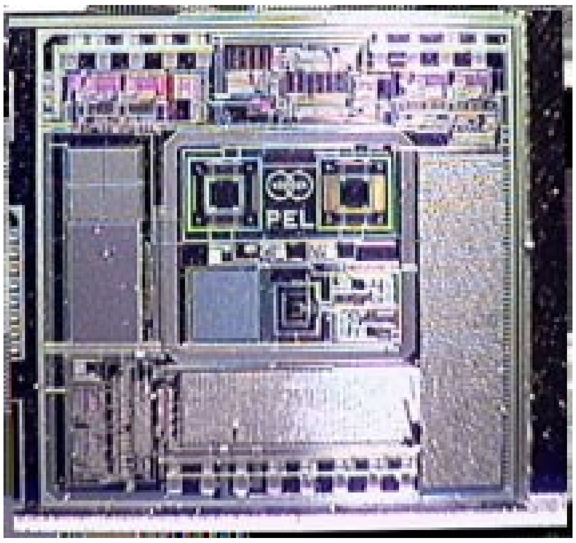


Fig. 1.1: Photograph of the SMC, flipchip version

them upside down on a ceramic substrate. The second version (Fig. 1.2) uses normal die attach and wedge bonding to connect the chips on the ceramic substrate. Then a glob-top is applied to seal the connections.

In Fig. 1.3 a schematic of all the parts of the SMC is shown. They are described in detail in the following chapters.

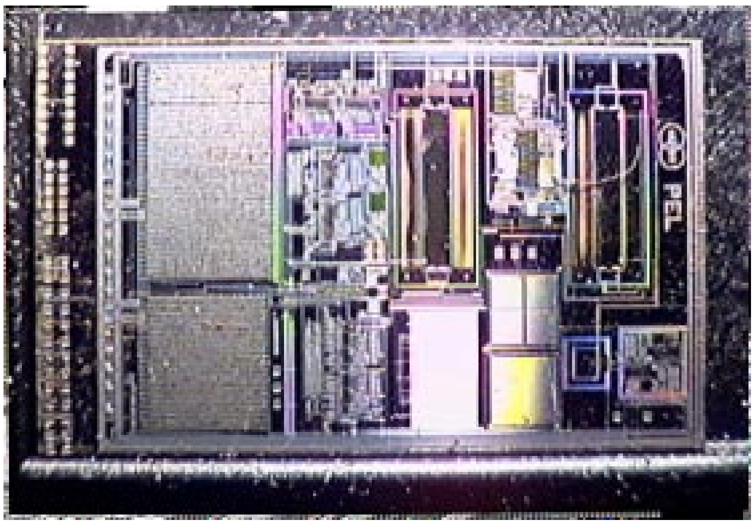


Fig. 1.2: Photograph of the SMC, globtop version

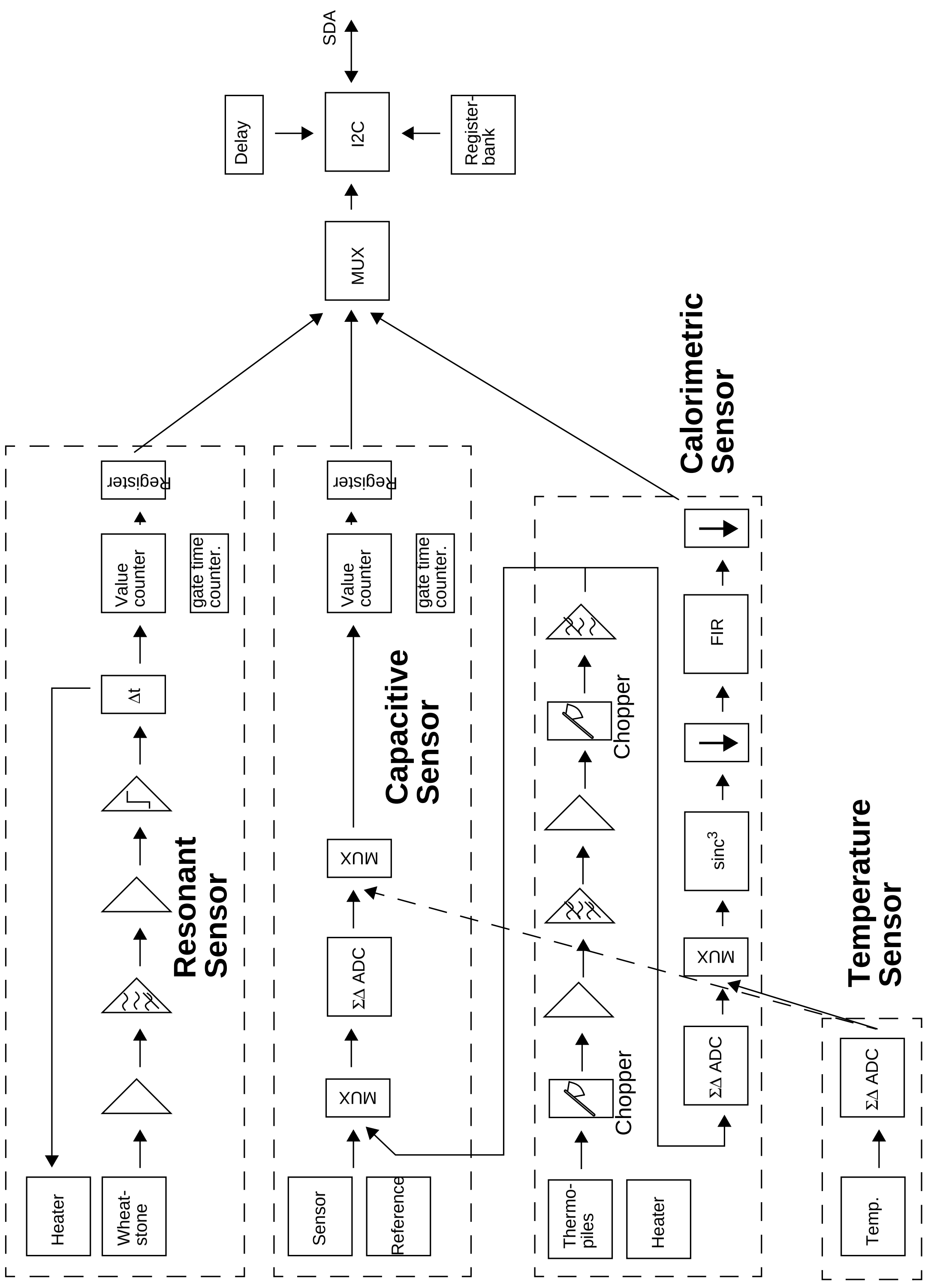


Fig. 1.3: Schematic overview for the SMC

2 DIGITAL PART

2.1 COMMUNICATION: FROM PC TO SMC AND BACK

While the final target of the project is to have a handheld device which includes six SMCs and a micro controller, for our testing purposes and the early gas measurements it is important to have a more flexible way of controlling the SMC and collecting the data. Therefore we chose the following approach [Tab. 2.1].

Tab. 2.1: The different stages of communication in the project

Device	Protocol	Device	Protocol	Device
PC	<--->	SMC	<--->	µC
	RS232		I2C	

On the PC we have a program running on a version of UNIX (in our case Linux) that takes the user's commands from an interface, assembles them into a simple protocol described below and sends them to the µC via a standard RS232 serial line. The µC receives them, converts them to the I2C protocol that the sensor is speaking, and sends them over the I2C line to the SMC. For the sensor values, it just goes the same chain backwards up to the PC, where the program logs the measurement data to a file and prints them to the screen. In this way, we could already develop the µC part of the project and later only replace the PC <-> µC routines by the desired µC user interface.

We now describe the two stages of communication and the associated protocols.

2.1.1 PC <-> UC

The protocol used for this part has changed considerably as the user interface on the PC and the program on the µC evolved. Earlier versions of the µC code polled the serial line for incoming data, but as explained in detail below, this caused problems and had to be fixed.

Reception on the PC was unreliable for the same reason. The ‘read serial’ routine was changed from polling to irq driven operation to resolve that problem. The basic code is very elegant in C; see Fig. 2.1. When *wait_flag* has been set to 0 by

```
int open_sdev(struct termios *old_pt)

{ int dsc;
struct termios new;
struct sigaction saio;

dsc = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NONBLOCK);

/* install the signal handler before making the device asynchronous */
saio.sa_handler = signal_handler_IO;
saio.sa_flags = 0;
saio.sa_restorer = NULL;
sigaction(SIGIO,&saio,NULL);

/* allow the process to receive SIGIO */
fcntl(dsc, F_SETOWN, getpid());

/* Make the file descriptor asynchronous (the manual page says only
O_APPEND and O_NONBLOCK, will work with F_SETFL...) */
fcntl(dsc, F_SETFL, FASYNC);
tcgetattr(dsc,old_pt);/* save current port settings */
memset(&new,0,sizeof(new));

/* set new port settings for canonical input processing */
new.c_cflag = 38400 | CLOCAL | CREAD | CS8;
new.c_iflag = IGNPAR | IGNBRK;
new.c_oflag = 0;
new.c_lflag = 0;
new.c_cc[VMIN]=1;
new.c_cc[VTIME]=0;
cfsetspeed(&new,B38400);/* set output speed */
tcflush(dsc, TCIOFLUSH);
tcsetattr(dsc,TCSANOW,&new);
return dsc;
}

================================================
void signal_handler_IO (int status)
{
wait_flag = 0;
}
```

Fig. 2.1: Signal handler and irq initialization of serial port

the interrupt routine, a loop takes care of reading out the data. There’s even a simple form of error management built into the RS232 protocol. The data format looks like this:

PACKET TYPES RECEIVED BY THE PC

data packet:

- flag byte (0x01)
- sensor address byte
- first parameter byte
- second parameter byte
- third parameter byte

PACKET TYPES SENT BY THE PC

two byte command packet:

- flag byte (0x02)
- target address byte

three byte command packet:

- flag byte (0x03)
- target address byte
- command byte

four byte command packet:

- flag byte (0x04)
- target address byte
- command byte
- register value byte

In the next version, we plan to include the flag byte, which indicates the type of the data packet, into some of the unused bits of another byte in order to save bandwidth on the serial line.

2.1.2 UC <-> SMC

The I2C (Inter IC bus) is a rather simple synchronous serial protocol developed by Philips in the 1970s. It uses three lines for communication: ground, SCL (a bidirectional clock line that is being pulled down with the clock frequency while data is being sent) and SDA (the bidirectional data line). The I2C definition by Philips has two different modes: the normal clock mode up to 100 kHz and the fast mode up to 400 kHz. For the SMC, the fast mode is used. SCL and SDA are realized as open-collector drivers. The I2C is a bus which can be accessed by several devices (up to 127). Each device has a unique bus address and can act as Master or Slave. A master can initiate a transfer which the addressed slave then answers. The I2C uses a sophisticated arbitration scheme in order to avoid bus collisions. Details can be found in [1] and [2]. In our case, we use the I2C to establish a communication between the SMC and the external micro controller. The I2C controller on the SMC receives the commands coming from the µC, passes them to the sensors and sends their answers back to the µC. The transmission protocol between SMC and µC can be divided into a hardware and a software part. The hardware protocol is explained in detail in [2], so it won't be covered here. The software part however has seen several modifications. The original version had severe timing problems and lost packets as it was polling both the serial and the I2C interface of the µC. The usual solution to such a problem is the use of interrupt driven reception routines. We proposed and implemented two irq routines. Of course, these two routines could disturb each other if the both get called on the same time. Therefore, an idea was necessary that prevents interaction. As the I2C bus has better arbitration and error handling capabilities than the serial line, we assigned a higher priority to serial communication. Therefore, the serial routines pull down the SCL line in order to avoid I2C traffic while they're busy. Currently, there are two suites of program versions available for the µC and the PC. The old version allows only one command to be transmitted to the µC, while the new PC software loops the input user interface routine. As the communications protocol has changed between the two revisions, be sure to download the right code to the flash memory of the µC using the 'flash' shell script.

The software I2C protocol between SMC and µC looks like this: Tab. 2.2

The bit order and the function of the bytes are still the same and can be found in [2].

Tab. 2.2: The software protocol

byte 0	byte 1	byte 1	byte 3	byte 4
SMC address	sensor address	com-mand/1st	2nd opt. byte	3rd opt. byte

2.2 THE REGISTER BANK

The SMC uses a register bank to control the settings of each sensor. The registers can be read and written via the I2C bus using commands that are specified in [2]. Each sensor can have up to 16 registers, called ComReg1..16. In the present design however, only some of them are used, so there is space for future enhancements. The registers can have different word lengths, depending on the desired value range. We now describe the bank for the capacitive, the calorimetric and the resonant sensor, and the way these registers change the settings of the sensors.

While working with the register bank of the SMC, an error was found in the design: normally, all registers can be set to a value, and this value can be read out afterwards. However, for one bit in ComReg3, the reading fails. After a redesign, we will see if the bug still exists in the second run.

2.2.1 THE CAPACITIVE SENSOR

ComReg1 (4 bits): specifies the Power-on delay. Values 0..15 correspond to time intervals of 320 µs to 10.5 s, according to the formula

$$T = \frac{256 \cdot 2^{x}}{f_{Clk}}$$

When one of the commands “on-read-off one” or “on-read many” is issued, the sensor waits for the time given by ComReg1 before the value is read. This enables the user to make sure that a stable signal is read after power has been turned on.

ComReg2 (4 bits): Sample delay. As in ComReg1, delays of 320 us to 10.5 s can be set which determine the sample frequency 1/T of the sensor. Therefore, sample frequencies of 0.095 up to 3125 Hz can be used. Commands “Read many” and “On-read many” send values according to this frequency.

ComReg3 (3 bits): gate time of the decimation counter. Values 0..7 result in gate times of 0.00512 to 0.655 s, according to

$$T = \frac{256 \cdot 16 \cdot 2^{x}}{f_{Clk}}$$

The gate time specifies how long the counter opens its gate before it returns the value. This allows scaling of the output and gives the possibility to choose between speed and accuracy. This setting is valid for the capacitive and the resonant sensor.

ComReg4 (6 bits): values of 0..63 determine the size of the interdigital compensation capacitance. It can also be used to scale the output.

ComReg5 (5 bits): parameters of the capacitive sensor.

Bit 0 and 1:	Bitstream multiplier. With these bits it is possible to clock the AD converter up to 4 times slower than the rest. This multiplies the output signal with 1, 2, 3 or 4.
Bit 2 and 3:	EnCap and EnCapFB. These bits toggle between the poly-poly capacitances for voltage measurement and the interdigital capacitors for capacitive measurement. The voltage input is connected to the output of the calorimetric sensor in order to provide a backup test. Note that for the normal capacitive use, these bits have to be turned on (they’re off after startup).
Bit 4:	Selects the input of the decimation counter. This bit does not control the power-on signal of the temperature sensor. To switch it on, bit 0 in ComReg3 of the calorimetric sensor has to be set to one. 0 selects the capacitive sensor, 1 selects the temperature sensor.

2.2.2 The calorimetric sensor

ComReg1 (4 bits) and **ComReg2 (4 bits)** have the same function as in the capacitive sensor. All notes made above apply without change.

ComReg3 (6 bits): parameters of the calorimetric sensor

Bit 0:	Sensor select. Select calorimetric sensor (low) or temperature sensor (high).
Bit 1:	G5_BP. The first amplifier and the bandpass amplifier in the sensor circuit have a gain which can be divided by 4. Bit 1 does that for the band pass amplifier. It can be used to avoid saturation in the amp.
Bit 2:	G5_PreAmp. This bit acts like bit 1, but in the preamp. With both bits high, gain is reduced by a factor of 16.
Bit 3:	Heater select. With this bit, the on-chip heating current can be toggled between the sensor and the reference thermopile. This is used for testing purposes.
Bit 4 and 5:	Current select. These two bits determine the heater current for testing the sensor. Bit patterns of 00, 01 and 11 lead to currents of 0, 15 and 30 μA, respectively. The combination 10 is identical to 00.

2.2.3 The resonant sensor

ComReg1 (4 bits) and **ComReg2 (4 bits)** are the same as in the other two sensors.

ComReg3 (7 bits): parameters of the resonant sensors

Bit 0, 1, 2:	Select delay. Each of these bits adds a delay of 200 ns. So 000, 001, 011 and 111 correspond to 0, 200, 400, 600 ns, respectively.
Bit 3 and 4:	Select current. Multiplies the current in the delay line with 1, 2, 3 or 4 and thus reduces the delay time by the same factor
Bit 5:	Divide current. Divides the supply current in the amplifier by two. This leads to a phase shift of +15 degrees.
Bit 6:	Double current. Doubles the amplifier supply current and shifts the phase by -15 degrees.

2.3 Digital part of each sensor

In the following section, we describe the digital part of each sensor. That is all the digital circuitry on the SMC that does not belong to the I2C interface and the register bank, as we already discussed them above. The general outline looks as follows: (Fig. 1.3). The outputs of the different sensors come together in a multiplexer which decides the output of which sensor is written into the DataOut register and then chopped into bytes to go over the I2C line. A difference between the digital parts of our sensors becomes important as soon as we talk about testing in Chapter 8: while the calorimetric sensor has a fully synchronous digital part, the capacitive and resonant sensors have asynchronous counters.

2.3.1 Capacitive and resonant sensor

The circuitry connecting to the multiplexer mentioned above is given in Fig. 2.2.

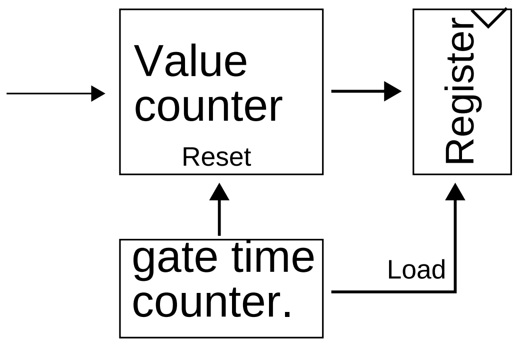


Fig. 2.2: Digital part of capacitive and resonant sensor

We see an asynchronous counter which counts the transitions of the sensor signal entering as a bit stream from the left and is reset by a second asynchronous counter, the gate time counter. The gate time counter counts down from the value given by ComReg3 in the register of the capacitive sensor. On the transition 0 -> -1, it loads the current counter value of the sensor counter into the output register and then resets the sensor counter. While this asynchronous design is very easy to understand, the calorimetric sensor needs a more sophisticated circuitry.

2.3.2 Calorimetric sensor

As can be seen in Fig. 2.3, the calorimetric sensor consists of a digital filter, a downsampling stage, another filter, a second downsampling stage, and the connection to the multiplexer. The bitstream entering from the left is first filtered by the sinc^3 filter with a frequency characteristic as in Fig. 2.4. In the following

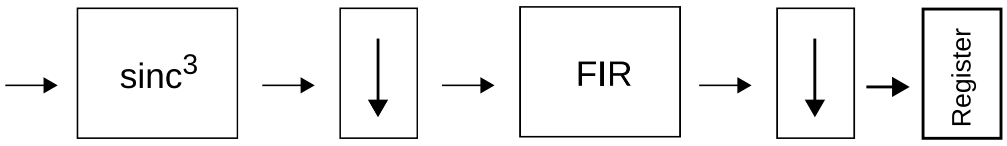


Fig. 2.3: Digital part of calorimetric sensor

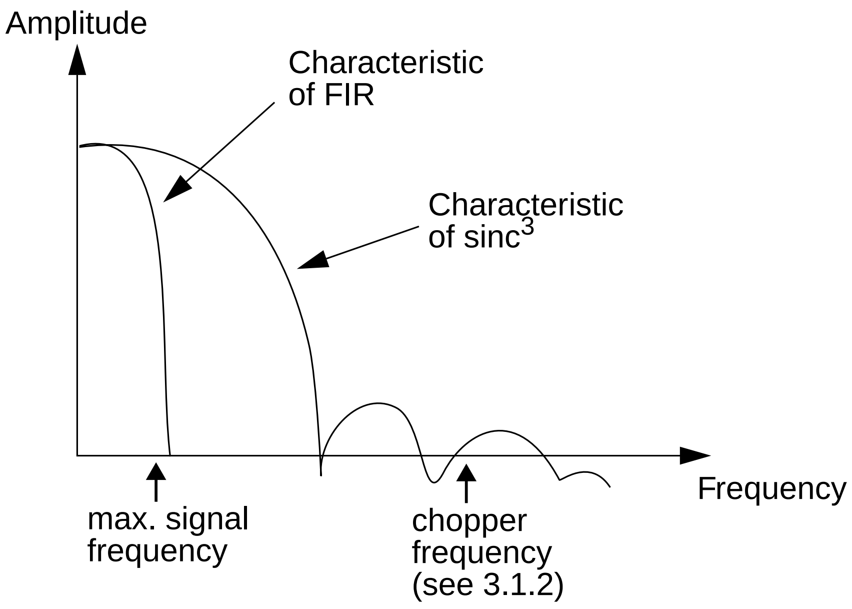


Fig. 2.4: Frequency characteristics of the two filters

downsampling stage, the sampling rate of the bitstream is reduced by a factor of 32. The higher spectral parts of the signal are therefore convoluted into the low frequency signal, but the high frequency suppression of the sinc^3 filter ensures that this is not a problem. The second filter, a FIR (finite impulse response), again acts as a lowpass with characteristic desired for the signal band. The second downsampling stage further reduces the sampling rate so that we have a reduction of 128 at the end of the filter bank. Thus, we at the output we have a frequency of

$$f = \frac{800kHz}{8 \cdot 128} = \frac{f_{\text{sampling}}}{128} = \frac{f_{\text{Clk}}}{\text{PreScaleFactor}} \cdot \frac{1}{128}$$

or about 800 Hz.

3 ANALOG PART

The analog circuitry on the SMC deals with the sensor outputs and the analog/digital converter before they are fed into the digital filter chain. We will describe the analog part of each sensor and the delay line which determines the timing of the I2C interface.

3.1 ANALOG PART OF EACH SENSOR

3.1.1 CAPACITIVE SENSOR

The only analog part of the capacitive sensor is the ΣΔ A/D converter, since the sensor and reference capacitance are directly incorporated into the converter. Fig. 3.1 shows a schematic drawing of the first integrator in the converter including the sensor and reference capacitor. ϕ and /ϕ are complementary nonoverlapping clocks [5]. For the theory of ΣΔ A/D converters, see [3]. Please also refer to Chapter 4 where we discuss the sensor in more detail.

3.1.2 CALORIMETRIC SENSOR

The basic job of the analog part of the calorimetric sensor is amplification. As the signals coming from the thermopiles are very weak, they have to be amplified roughly 10’000 times. With a traditional amplification scheme, we’d run into two problems:

- even the smallest DC offset at the input of the amplifier would overdrive the A/D converter at the output
- the 1/f noise, amplified by 10’000 in the first amplifier, would mask our signals

Therefore, an interesting amplification concept is used here: a chopper instrumentation amplifier [4]. A schematic of the amplification stage is shown in Fig. 3.2. The thermocouple voltage entering from the left is first chopped with a frequency of about 5 kHz, which means that it is modulated with a square wave of that frequency. This shifts our signal to higher frequencies. After the first amplifier, the signal is bandpass filtered. The filter eliminates two problems:

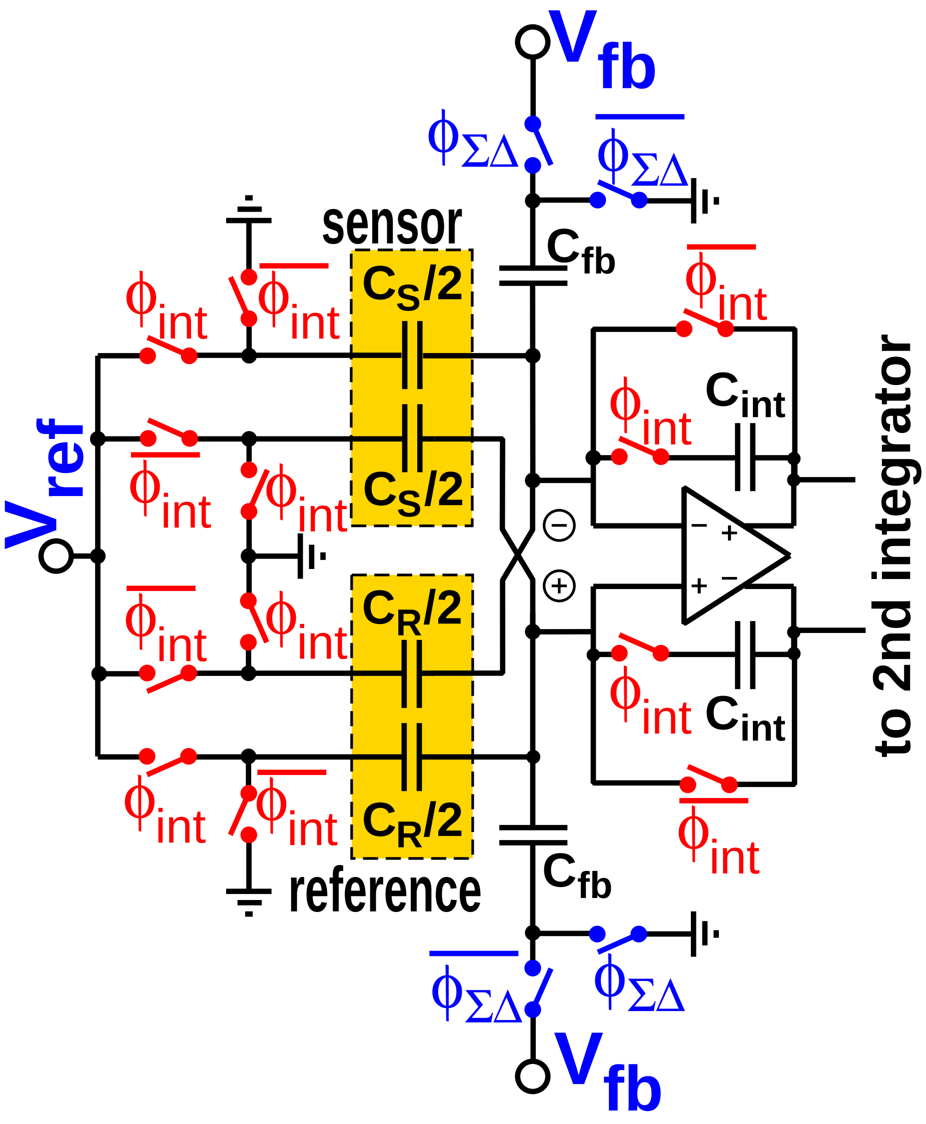


Fig. 3.1: *Schematic of the 1st integrator of the ΣΔ A/D converter*

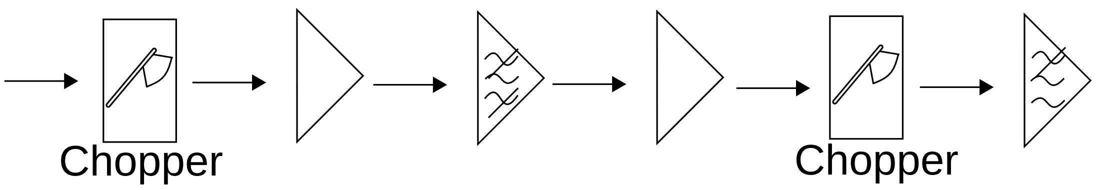


Fig. 3.2: *Analog part of the calorimetric sensor*

- saturation in the output is avoided as the DC offset is removed
- the so called residual offset (demodulated signals from higher harmonics of the chopper frequency), is reduced.

After the third amplifier, the signal is demodulated. This translates the sensor output back to DC, with 1/f noise and DC offset mostly removed.

Special attention has to be paid to the two important frequencies in this circuit: the center frequency of the bandpass filter and the chopper frequency. As they have to coincide (severe phase problems would occur otherwise), no divided clock signal is used for the chopper frequency. Instead, a second filter is used in feedback mode to generate the chopper frequency. After amplification, the signal passes an anti-aliasing filter which cuts off all spectral parts above half of the sampling frequency of the ΣΔ A/D converter.

3.1.3 Resonant sensor

The output of the resonant sensor is a sine. In order for the feedback loop to work (see chapter 6, The resonant sensor), we need to transform it into a digital signal with the same frequency. If we only amplified the sine and used it to drive the cantilever, we'd get no excitation at the base frequency but at the second harmonic ($P \sim U^2 \sim (\sin\omega t)^2 = \frac{1}{2}(1-\cos 2\omega t)$). The analog stage that does this is outlined in Fig. 3.3. The weak signal coming from the Wheatstone bridge is passing

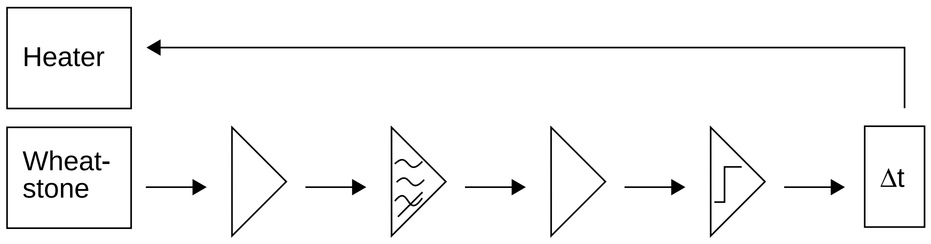


Fig. 3.3: Feedback loop of the resonant sensor

three amplifiers, all in fully differential design. The first and the third only amplify the amplitude, the second additionally performs a high pass filtering of the signal. The amplified signal then passes a comparator that converts it into a digital square wave. The delay which is required by the feedback is added by the circuit shown in Fig. 3.4.

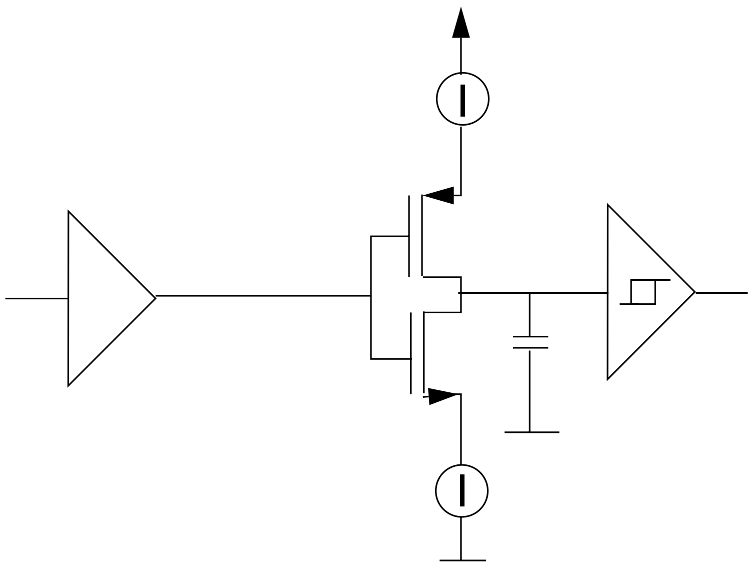


Fig. 3.4: Schematic of the delay line

First, a buffer decouples the signal from the comparator. Then, we have two current sources and two complementary transistors, one of which is closed and the other open, depending on the state of the input. Their outputs are connected to a capacitor and a Schmidt-Trigger. The hysteresis of the Schmidt-Trigger ensures that its output does not change until the capacitor has been loaded or emptied for a certain time. In this way, we can control the delay time by the current that is provided by the current sources.

The amplified, digitized and delayed signal then goes to the driver of the electro-thermal actuation stage which provides the resonant supply for the cantilever.

Apart from this delay line, we have another one which determines the timing of the I2C interface. According to the I2C specifications, it has to provide a delay of 300-600 ns.

4 The capacitive sensor

4.1 Sensor principle

The capacitive sensor is in a way the most direct one. It uses two capacitors and measures the change in their capacitance difference as one of them changes its value due to the analyte. For that reason, one of the capacitors is sealed with oxide while the other one is coated with the polymer. The polymer changes its dielectric constant upon absorption of analyte. Fig. 4.1 shows the two main

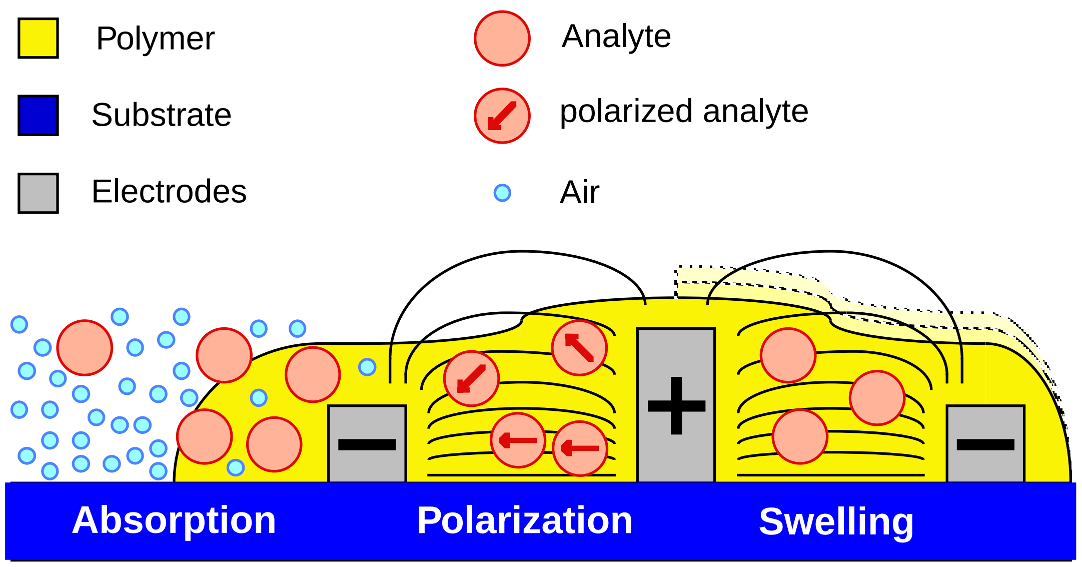


Fig. 4.1: Changes in polymer due to analyte

effects: polarization and swelling. In Fig. 4.2 we see the basic circuit diagram of the sensor capacitors and the ΣΔ A/D converter attached to it. While in earlier designs the difference in capacitance was detected directly by a differential amplifier, the SMC uses a different design where each capacitor is split into two parts. This avoids the main problem the old design had: a loss of sensor signal

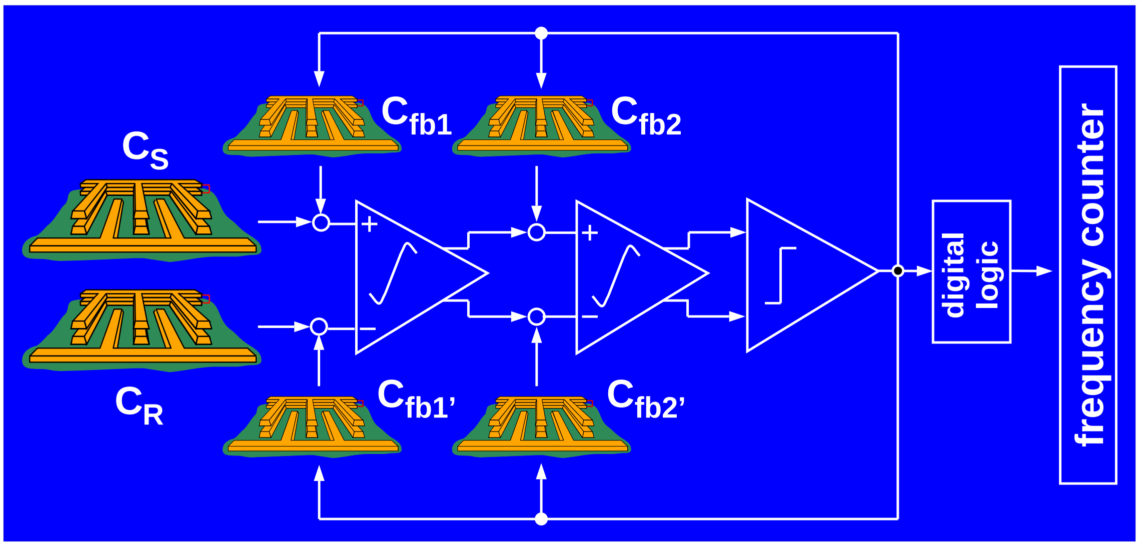


Fig. 4.2: Schematic of the ΣΔ converter, different capacitors

charge due to the common-mode voltage change at the input of the differential amplifier. Details can be found in [5].

The sensor and the reference capacitors are used as the input capacitors of the switched-capacitor integrator of the ΣΔ converter. Thus, we directly generate a bitstream proportional to the difference of the two capacitances and therefore the analyte concentration. Details on the capacitive sensor can be found in [6].

5 THE CALORIMETRIC SENSOR

5.1 SENSOR PRINCIPLE

The calorimetric sensor (Fig. 5.1) can give additional information. This can help to distinguish between different analytes that yield similar sensor responses on the other sensors. We have a thermally insulated structure, in our case an n-well island membrane of the CMOS process. This membrane is coated with the chemically sensitive polymer and contains a great number of polysilicon/aluminum thermopiles along its edges. When analyte is absorbed into or desorbed from the polymer, the enthalpy in this layer changes. This enthalpy change manifests as a temperature change on the island structure. The thermopiles convert a temperature difference between island and substrate into a voltage which is fed to the A/D converter. Thus, we get a transient signal which is proportional to the derivative of the concentration of the analyte as a function of time: dc/dt.

A typical measurement with the calorimetric sensor are given in Fig. 5.2. A peak in the positive y-direction occurs when the analyte is absorbed, and the peak is negative when it is desorbed. In order to get information about the concentration of the analyte, we have to integrate the thermovoltage over time. The area under a peak is a measure for the amount of analyte. Inspection of Fig. 5.2 reveals that most of the time the signal will be close to zero, but when gas is absorbed or desorbed, we have to get many samples in a short time in order to be able to integrate over the correct shape of the peak.

Furthermore, the island structure contains a polysilicon heater which can be used to test the device. The membrane itself is released in a post-processing KOH etching step with electrochemical etch stop. Details about calorimetric sensors can be found in [7].

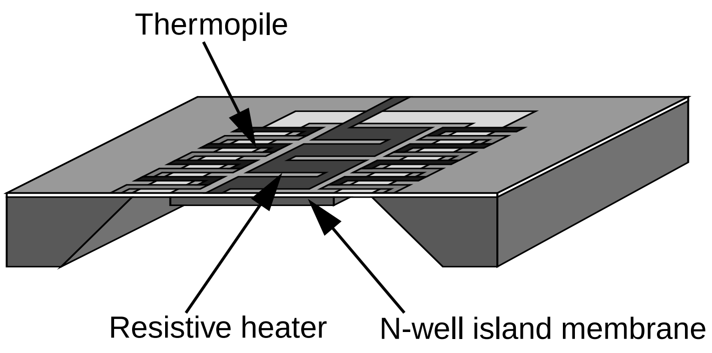


Fig. 5.1: Schematic drawing of the calorimetric sensor

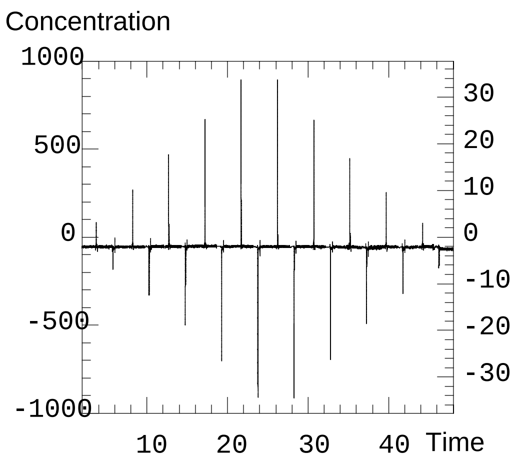


Fig. 5.2: Typical measurement graph of the calorimetric sensor

6 THE RESONANT SENSOR

6.1 SENSOR PRINCIPLE

The resonant sensor detects the shift of the resonance frequency in a cantilever in resonant vibration. When the analyte is absorbed into the polymer which coats the cantilever, the mass of the cantilever-polymer system increases and therefore the resonance frequency is decreased.

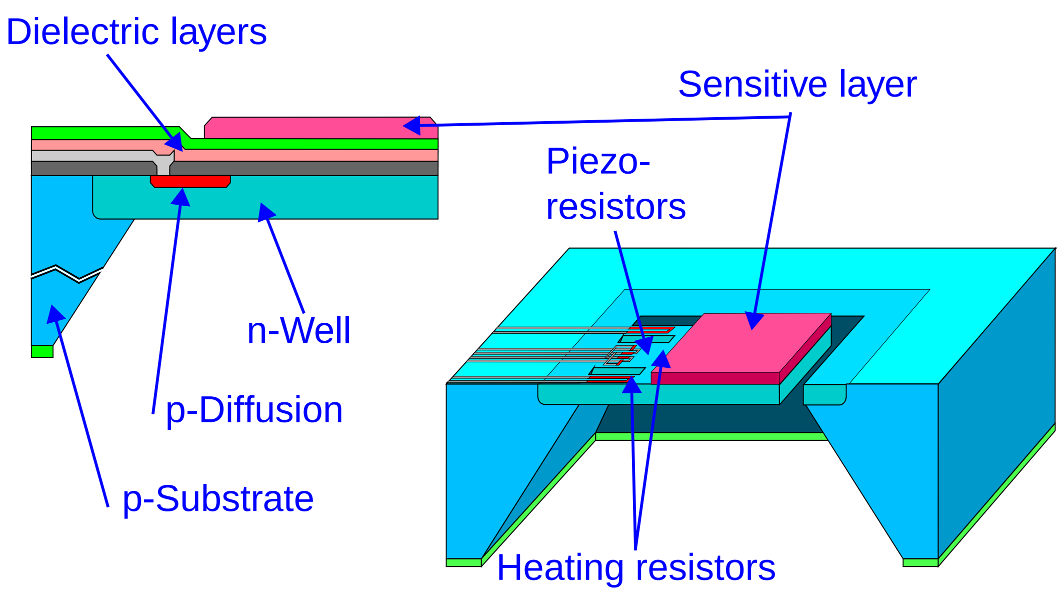


Fig. 6.1: Schematic drawing of the resonant sensor

In Fig. 6.1 the general setup for such a sensor is given. We have a rectangular cantilever consisting of a n-well and the dielectric layers of the CMOS process. The cantilever is coated with the polymer and after that has been released by a post-processing etch step. At its clamped edge, the cantilever features two heating resistors and four sensor resistors in Wheatstone configuration are built in. When the heating resistors are supplied with a heater current, they electrothermally excite the cantilever locally and cause it to bend. The Wheatstone bridge uses the piezoresistive effect to detect this displacement. If we now succeed in amplifying the output signal, delay it for half the resonance clock period of the cantilever, and feed it back to the actuation resistors, we can force the cantilever into resonance vibration.

As the resonance frequency is stable down to 0.05 Hz as measurements have shown [8], this system is very sensitive to changes in analyte concentration which reach the order of kHz. As an additional advantage, the quasi-digital output in the form of a frequency can directly be fed into the counter.

7 THE TEMPERATURE SENSOR

7.1 SENSOR PRINCIPLE

In addition to the three gas sensors, the SMC contains a temperature sensor. It can be used to compensate for the temperature dependence of the capacitive, calorimetric and resonant sensors. In the schematic overview of the SMC (Fig. 1.3) we see that the bitstream of the temperature sensor can be fed into the filter chain of either the capacitive or the calorimetric sensor. We will now give a short description of how this sensor works. Please refer to [9] for details.

The underlying measurement principle is to use the temperature dependence of the base-emitter-voltage of a bipolar transistor in the CMOS process. The base-emitter-voltage changes with temperature roughly like this (Fig. 7.1):

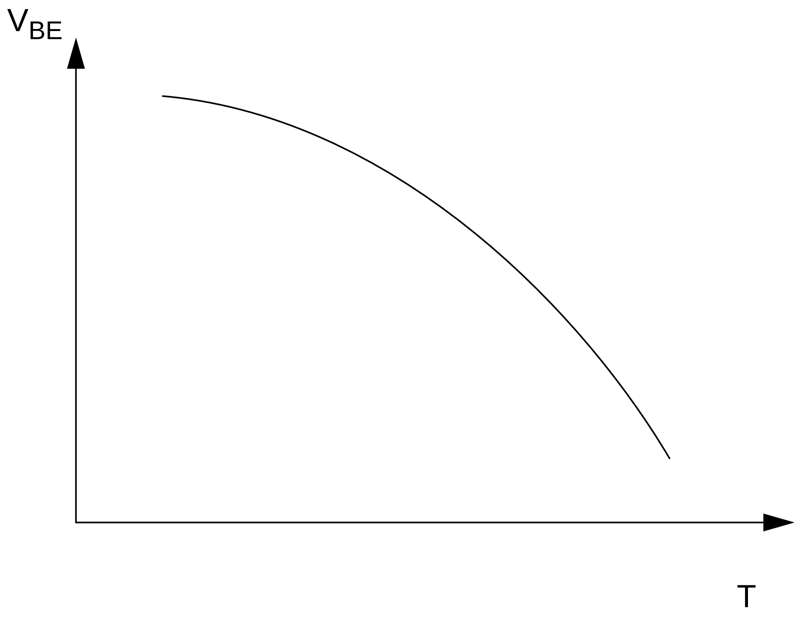


Fig. 7.1: Temperature dependence of the base-emitter-voltage of a bipolar transistor

Of course, we want a linear temperature dependence of the output signal. This can be achieved by a circuit sporting two bipolar transistors (Fig. 7.2). If we drive

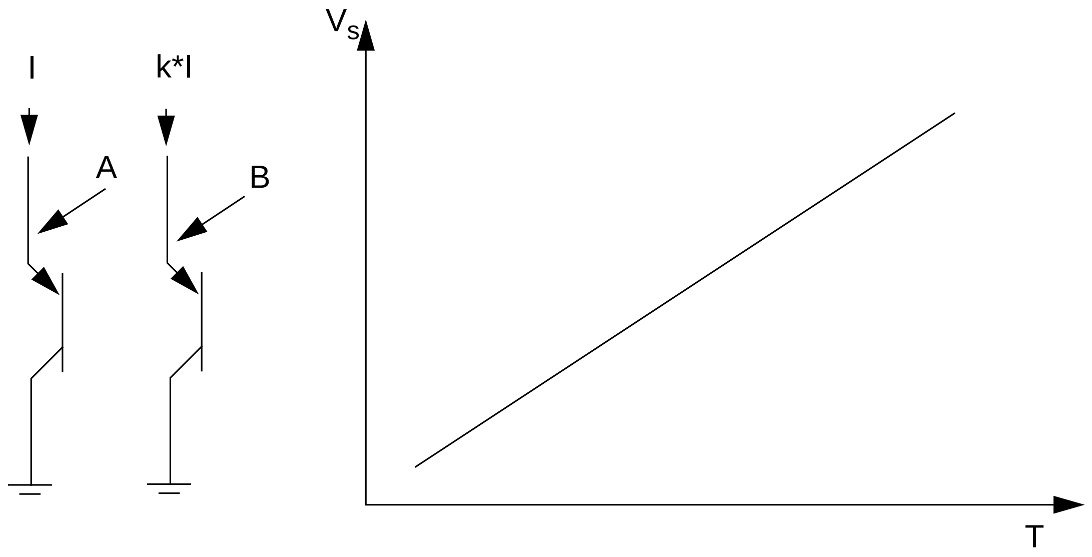


Fig. 7.2: Basic idea for the temperature sensor

one of them with the current I and the other one with k*I, we will get a temperature dependence of the difference of the two voltages taken at points A and B which is governed by

$$V \propto \ln(k) \times T$$

Thus, we have the desired linear dependence. Of course we have to convert this voltage into a digital signal. Our ΣΔ A/D converter needs a reference voltage to do this. Most on-chip references compensate the temperature dependence of the base-emitter voltage of a bipolar transistor in some way to realize a voltage that is constant over temperature. Thus, the measured voltage and the reference voltage are based on the same principle. A one-bit switched capacitor ΣΔ A/D-converter is a sampled system that feeds ($V_{in} \pm V_{ref}$) into its first integrator in every cycle. We can subdivide this cycle and generate the sensor and reference voltage from the same sensing transistor. This avoids separate circuitry for the reference voltage as well as matching problems.

The bitstream from the converter can be read out via the capacitive or calorimetric filter bank.

8 Testing of VLSI and mixed signal ICs

8.1 Theory

8.1.1 Introduction

As the complexity of VLSI circuits and microprocessors gets bigger and bigger, testing becomes an inevitable means for quality control. For today´s highly complex ICs, the cost for testing can already exceed that for production (Fig. 8.1), [10].

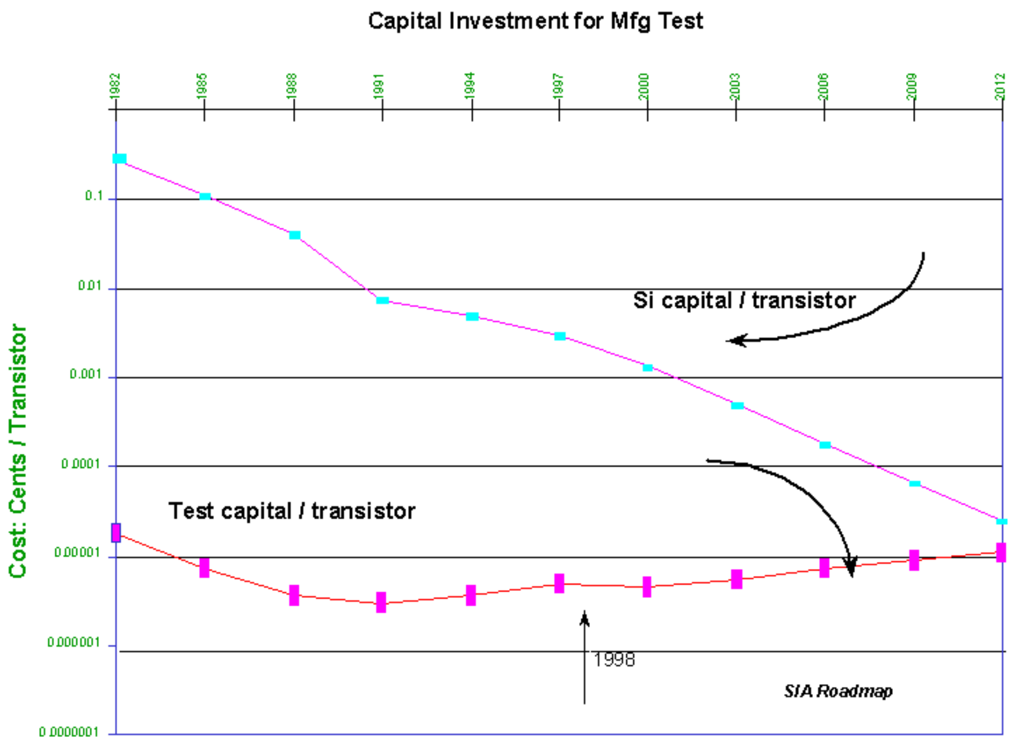


Fig. 8.1: Testing cost will soon exceed production cost

In our special case, where 6 SMCs are flip chip or globtop bonded on a ceramic substrate, we have to be absolutely sure to have 6 working chips as only one defunct sensor would render the whole module useless. Therefore, a thorough and exhaustive testing scheme for all parts of the SMC (digital part, analog part, sensors) was to be developed. Only small amounts of chips have to be tested in the current project (one wafer holds only 16 chips). However, for proof of concept, the basics of commercial VLSI testing were not to be forgotten: time is money. While the fabrication of the sensors in CMOS technology is an automated batch process where a huge number of chips can be manufactured in parallel, testing is a single chip job with highly specialized (and expensive) equipment. The plan was to use PEL´s wafer prober and a commercial pattern generator / logic analyzer to perform the testing.

An exhaustive introduction into testing theory is not given here - there's a lot of good literature [11][12][13][14]. But we have to get the basic ideas of controllability, observability, fault models, scan vector algorithms, and design for test.

8.1.2 FAULT MODELS

A circuit can be defect in many ways. In this first chapter, we restrict ourselves to digital logic. The analog part of the SMC will be dealt with separately. There's an arbitrary number of fault sources for logic as complex as this. So one makes the assumption that only one fault occurs at one time, and that this fault follows a simple idea: it is a stuck-at-fault. The corresponding location in our circuit has a short to either GND or VDD and therefore stays at that level all the time: stuck-at-0 (sa0) or stuck-at-1 (sa1). No bridges between neighboring lines, no time-dependent faults, no clustering of faults are taken into account. However, theoretical considerations show that this model, applied to any point in the logic, covers almost all of the possible faults.

The good thing about it is that it´s simple. Every point in a circuit belongs to one of three groups: it is either a primary input, an internal node, or a primary output. We can easily access primary in- and outputs, but the states of the internal nodes are basically hidden.

8.1.3 CONTROLLABILITY AND OBSERVABILITY

This is where controllability and observability come in: a node is controllable if we can force it to be either 0 or 1. For a primary input that's easy, but for an internal node it can be quite hard (depending on the logic leading from the primary inputs up to this node, it can be necessary to set many primary inputs), or even impossible. The same is true for observability: the question is if the logic can be forced into a state where the measurement of the primary outputs allows us to definitively determine the level of our internal node. This also can be a complex or impossible job.

As we now know about the stuck-at model and controllability/observability, we get the idea of our test: first, we assume the circuit to be fault-free. Then, for every point in our logic, we look for a way (via the primary inputs) to set it to 0 and 1. Then we compute the correct answer of the fault-free circuit to the primary outputs. This provides us with a set of vectors on the input and a set of corresponding vectors on the output which define the correct logic. If we now apply these vectors to our circuit under test and compare the circuit´s answer with the computed vectors, we are able to detect stuck-at faults in the logic as these two bit patterns will differ. But here we see that controllability and observability directly affect the fault coverage of our testing: only faults of controllable and observable nodes will be detected. Furthermore, the number of vectors necessary for a given fault coverage raises almost exponentially.

Fortunately, a reasonable number of vectors can deliver a fault coverage of > 98% in many cases. So far, we have only dealt with combinatorial logic. But what about sequential logic as in memories? As the state of this kind of circuitry depends not only on the state of the inputs, but also on the history of the logic, sequential logic is not testable with this kind of setup.

So we see two problems with the setup developed so far: observability and controllability of all nodes either requires a huge number of vectors or is not even possible at all, especially when there are only few inputs and outputs as is the case for the SMC, and we are blind to faults in sequential logic.

8.1.4 Scan chain

This is where the scan chain comes in. It’s an ingenious idea that changes the whole thing. We can guarantee perfect access to any node, and even sequential logic becomes testable, although we still may end with a lot of vectors.

The concept is the following: to each flip-flop in the circuit, we add a switch which toggles the input of the flip-flop from normal operation to test mode. In this test mode, all flip-flops in the logic are linked together input-output, they form the scan chain. In addition, their clock inputs are switched to a common test clock. Immediately, we have full control over every input and output in the circuit: we feed a sequential scan vector to the input of the first flip-flop, and operate the whole scan chain with the common test clock. For each clock cycle, a bit of the vector gets loaded into the shift register built by the flip-flops. When the scan chain is “full”, we switch the input multiplexers back to normal operation, go for one clock cycle, and have the circuit´s answer to the scan vector. In order to get this answer out of the chip, we toggle the input switches back into scan mode, and shift bit after bit out of the scan chain.

We are even able to test parts of the circuitry that are not directly under the control of the scan chain: if we ensure that all registers determining the state of the chip are indeed controlled by the scan chain, we can set the logic into a state we want, but then turn the test mode off not only for one but for many clock cycles, so that some analog or more mysterious digital parts of the chip can go for normal operation. When we have operated the circuitry for enough cycles in normal mode, we switch back to test mode and can read out the answer. This is how we are going to test our sensors without even using the I2C interface.

8.1.5 Algorithms

The next question of course is: where do those vectors come from? Fortunately, we don’t have to make them by hand. Literature knows several algorithms for automatic test pattern generation (ATPG), for example the boolean difference method, or Roth’s D-algorithm [11]. One can also use pseudo-random test pattern generation, which for complex circuits has a higher fault coverage than the algorithms if only small numbers of vectors are used. As synopsys, our logic synthesis suite, provides us with a rich set of vectors (about 880’000 lines), we won’t go into these algorithms and instead refer to literature [11].

8.2 The setup

The goal was clear: to provide a tool which is able to decide on wafer level which SMCs are functioning and which aren’t. We can break up this job into several parts: wafer handling and electrical contact, supply infrastructure for the test, the tester itself, and the controlling of it all.

8.2.1 WAFER HANDLING AND ELECTRICAL CONTACT

PEL already owns a Synatron Summit 12742SP wafer prober (Fig. 8.2). It is able to handle wafers up to 20 cm which can be stepped through very easily chip-to-chip once the geometry is entered. The pads on the wafer are contacted

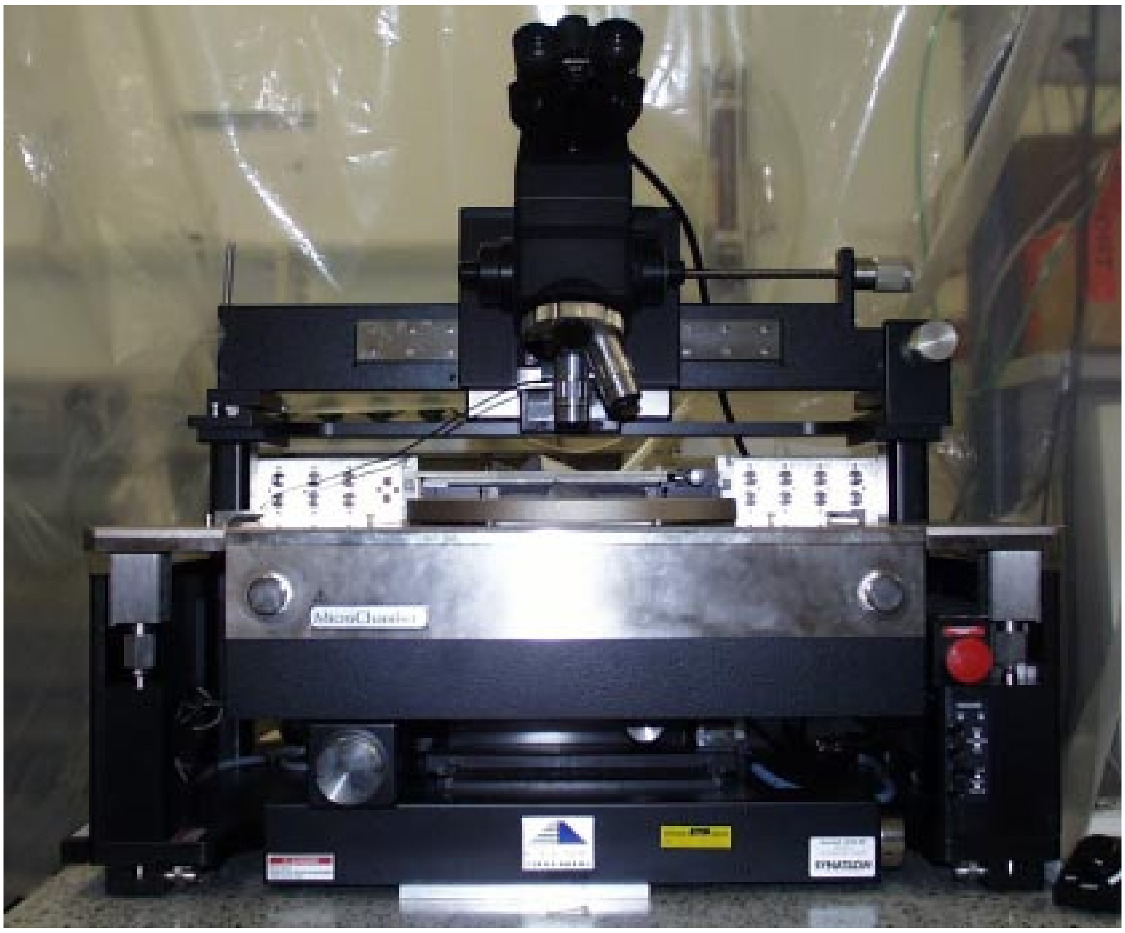


Fig. 8.2: PEL's wafer prober

by a probe card. It's a PCB which holds up to 70 tiny needles with µm precision (Fig. 8.3). When the probe card is lowered onto the wafer, each of the tips contacts one of the pads. In our specific case, we had to contact 20 pads on the glob-top version of the SMC (Fig. 8.4). They all lie in one line. From the probe card, a flat ribbon cable leads to the supply PCB.

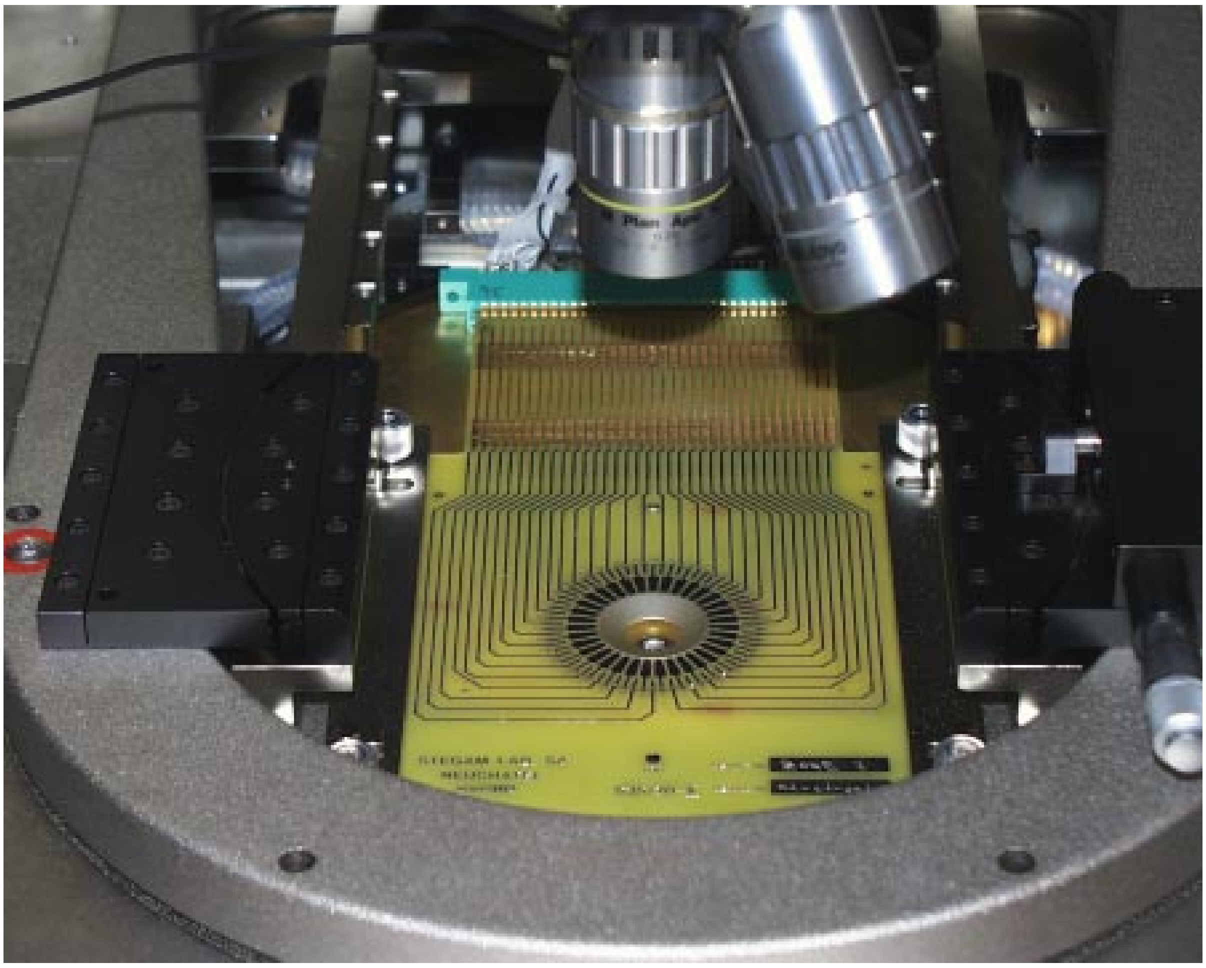


Fig. 8.3: The probe card

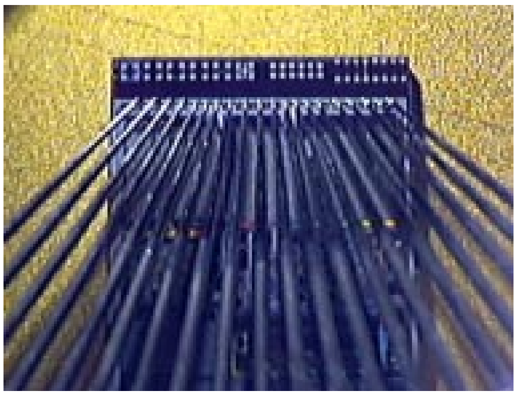


Fig. 8.4: Probecard lowered onto the SMC

8.2.2 The supply PCB

It’s a small circuit that has connectors for all the signals needed by the test setup and provides the chip under test with the appropriate supply voltages. In Fig. 8.5 one can see the LP2951, a controllable voltage regulator. It provides the 2.2 V

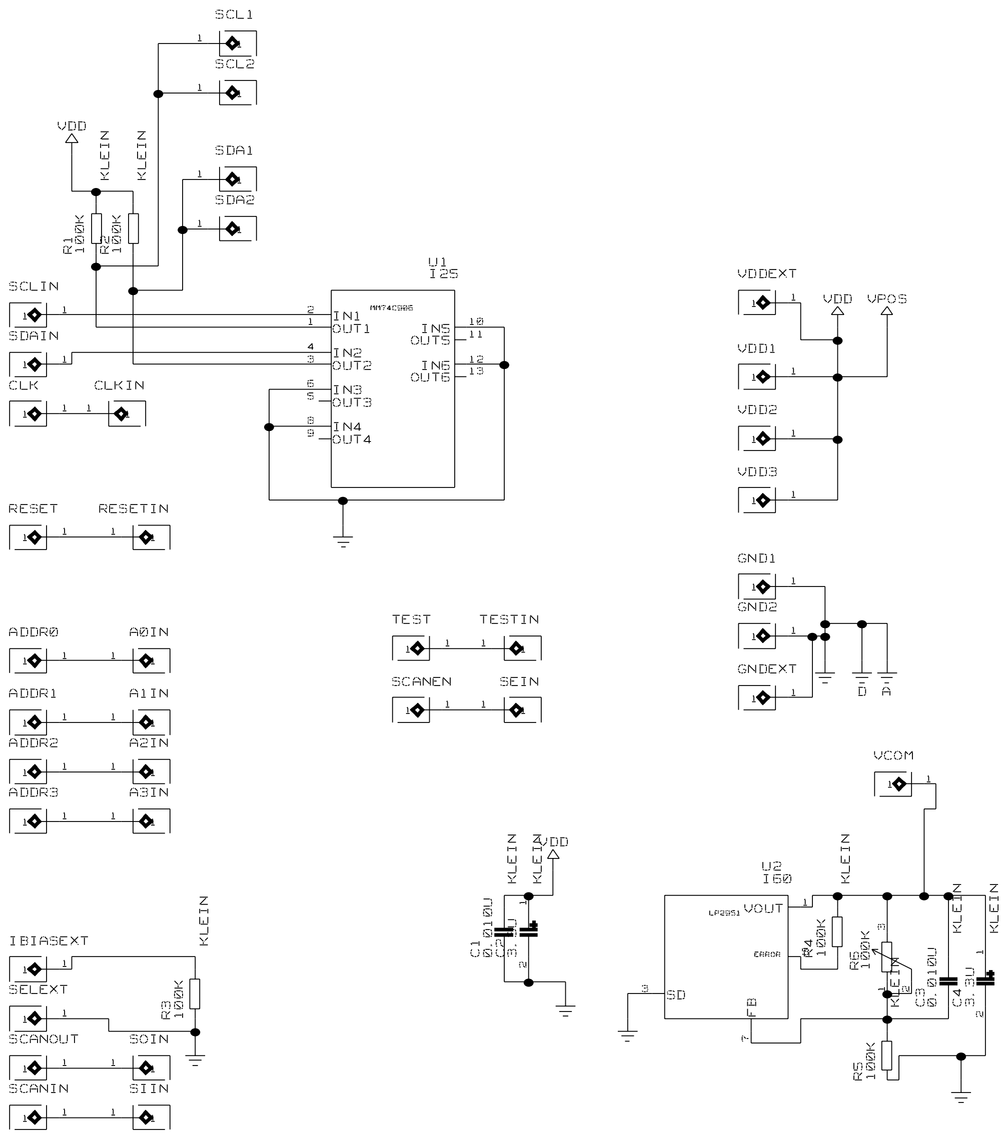


Fig. 8.5: Schematic of the PCB

needed for the V_{com} input of the SMC. The 74C906 is an Open Drain Buffer. This has the following reason: SCL and SDA are used as both in- and outputs in the test setup. In order to allow simultaneous access to those ports, we have to buffer the output of the test generator as long as there is no valid data applied. In Fig. 8.6 we see a photograph of the actual PCB.

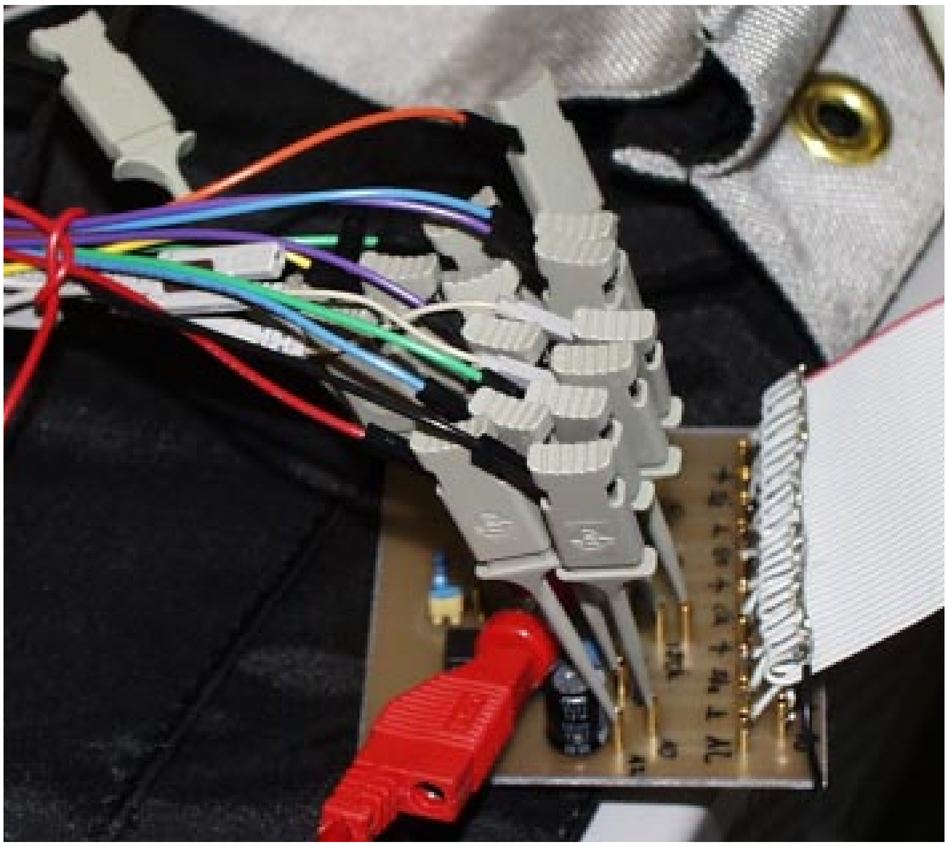


Fig. 8.6: The supply PCB

We use the testers's probe clamps to contact the plugs on the PCB. The next step would be to contact directly to the flat ribbon cable connectors that the prober is using.

Now we have some cables connected to some sockets, but which cables go where and why?

8.2.3 THE TEST VECTORS

To answer this question, we have to take a look at the test vectors generated by synopsys. The header for the vector file looks like this:

```
#==============================================================
# DESIGN NAME:  I2C_CYE500
# CUSTOMER:  Christoph H of PEL
# LIBRARY TYPE:  cyb
# REVISION:  1.00
# DATE:  07/23/2000
#==============================================================


Clock      1  I    # system clock; active rising-edge
XReset     2  I    # primary input
SCL        3  IO   # bidirectional inout port
SDA        4  IO   # bidirectional inout port
SMCAdr[3]  5  I    # primary input
SMCAdr[2]  6  I    # primary input
SMCAdr[1]  7  I    # primary input
SMCAdr[0]  8  I    # primary input
Test       9  I    # primary input
CapIn      10 I    # primary input
CalIn      11 I    # primary input
ResIn      12 I    # primary input
TempIn     13 I    # primary input
Scan_In    14 I    # scan_in port
Scan_Out   15 O    # scan_out port
Scan_En    16 I    # scan_mode_control port
```

There are 13 inputs, one output and the two combined signals SCL and SDA mentioned above. Each signal will be briefly described.

`Clock` is the system clock for the whole SMC. In normal operation, a 800 kHz square wave signal is applied here and is supplied to the different clock stages on the chip. There are many clock domains on the chip. When the test input is high, all these clock domains are connected together by a multiplexer. This way, clock is supplied to every flip flop in the scan chain.

`XReset` is the reset input for the SMC. While low, the chip is reset to a defined state. In normal operation, `XReset` has to be H.

`SCL` is the clock input/output of the I2C interface. It is used as both input and output for testing.

`SDA` is the data input/output of the I2C interface. It's also used in both directions for testing.

`SMCAdr0..3` determine the I2C address of the SMC in normal operation.

`Test` is one of the two inputs that determine whether the chip is in test mode. When `Test` is H, `Clock` becomes the test clock.

`CapIn, CalIn, ResIn, TempIn` are used in the following way: for the test, the output of the sensors are assumed to be zero. Otherwise, the result of the test would depend on the sensor output. In order to force the ATPG into an operation with all the sensor inputs zero, they are designed as primary inputs but then forbidden to be anything else but zero. Thus, they are not used in our testing.

`ScanIn` connects to the input of our first scan chain flip flop and therefore takes all the input bits to be fed into the chain.

`ScanOut` is the output of the last flip flop in the scan chain. This is where we can read our answer.

`ScanEn` determines if the chip is in normal (L) or test (H) operation.

Next, we have to get an idea how the test vectors look like. Here we have the beginning (remember, we have more than 880'000 lines) of the test vector file:

```
#=============================================================
# DESIGN NAME:  I2C_CYE500
# CUSTOMER:  Christoph H of PEL
# LIBRARY TYPE:  cyb dw01.sldb dw02.sldb dw03.sldb dw04.sldb
# REVISION:  1.00
# DATE:  07/23/2000
#==============================================================


# Synopsys Test Compiler, 1999.10 (Sep 02, 1999) was used to
generate this pattern set
# INPUT VECTOR FILE = I2C_CYE500.vdb was the source file for
this pattern set


40.0 ns         D N N N N N N N N N N N N N X N
55.0 ns         D N X X N N N N N N N N N N X N
105.0 ns        D U X X N N N N U D D D D N X N
```

```
# Pattern 0
205.0 ns        D U X X N N N N U D D D D D X U
240.0 ns        U U X X N N N N U D D D D D X U
255.0 ns        D U N N N N N N U D D D D D X U
340.0 ns        U U N N N N N N U D D D D D X U
355.0 ns        D U N N N N N N U D D D D D X U
405.0 ns        D U N N N N N N U D D D D U X U
440.0 ns        U U N N N N N N U D D D D U X U
455.0 ns        D U N N N N N N U D D D D U X U
540.0 ns        U U N N N N N N U D D D D U X U
555.0 ns        D U N N N N N N U D D D D U X U
605.0 ns        D U N N N N N N U D D D D D X U
640.0 ns        U U N N N N N N U D D D D D X U
```

We see 17 columns: the first one is the timing, followed by the 16 in- and outputs in the same order as above. We can see six different characters in the columns: D and U are used for inputs: D means down (0) and U means up (1). For output signals, we have L and H (low (0) and high (1)). N means ‘don’t know’ for inputs, X is ‘don’t care’ for outputs.

In the first two lines, we see a reset cycle. Then, different bit patterns are applied to the inputs, and the correspondent outputs can be seen in `SCL`, `SDA` and `ScanOut`.

So we have the vectors, and we have the connections. Now we need the tester.

8.2.4 THE TESTER

As we don’t have one of the real big machines (Fig. 8.7 and Fig. 8.8), we found a solution with the Agilent 16702 Logic Analysis System (Fig. 8.9). It features high-speed digital pattern generation with up to 208 channels, and can simultaneously analyze data on as many lines. As it is a full-featured Unix machine running HP-UX, it offers all advantages of a modern networking OS (redirectable X Windows frontend, network access, NFS..). Additionally, one can buy an oscilloscope card which allows the monitoring of analog data.

The general idea for using this machine for testing is the following: we load our vectors into the generator and connect all relevant signals to the probe card. The analyzer is also connected to the appropriate lines. Then we lock the timing of the analyzer to the generator. If we now run the system, we get an answer line for each line in the vector file.

Fig. 8.7: *The Teradyne L320 series*

Fig. 8.8: *The Schlumberger IHS1000*

In Fig. 8.10 we see the output pod allocation for our configuration. We use all bits of pod 5 and 6 bits of pod 4. One may notice that there are three bits we haven't mentioned so far: step, sync and read527. Step is used to tell the wafer prober to step to the next reticle. We pull it down at the end of a testing session to get the next chip. Sync is used as a one-bit alternating clock for the following rea-

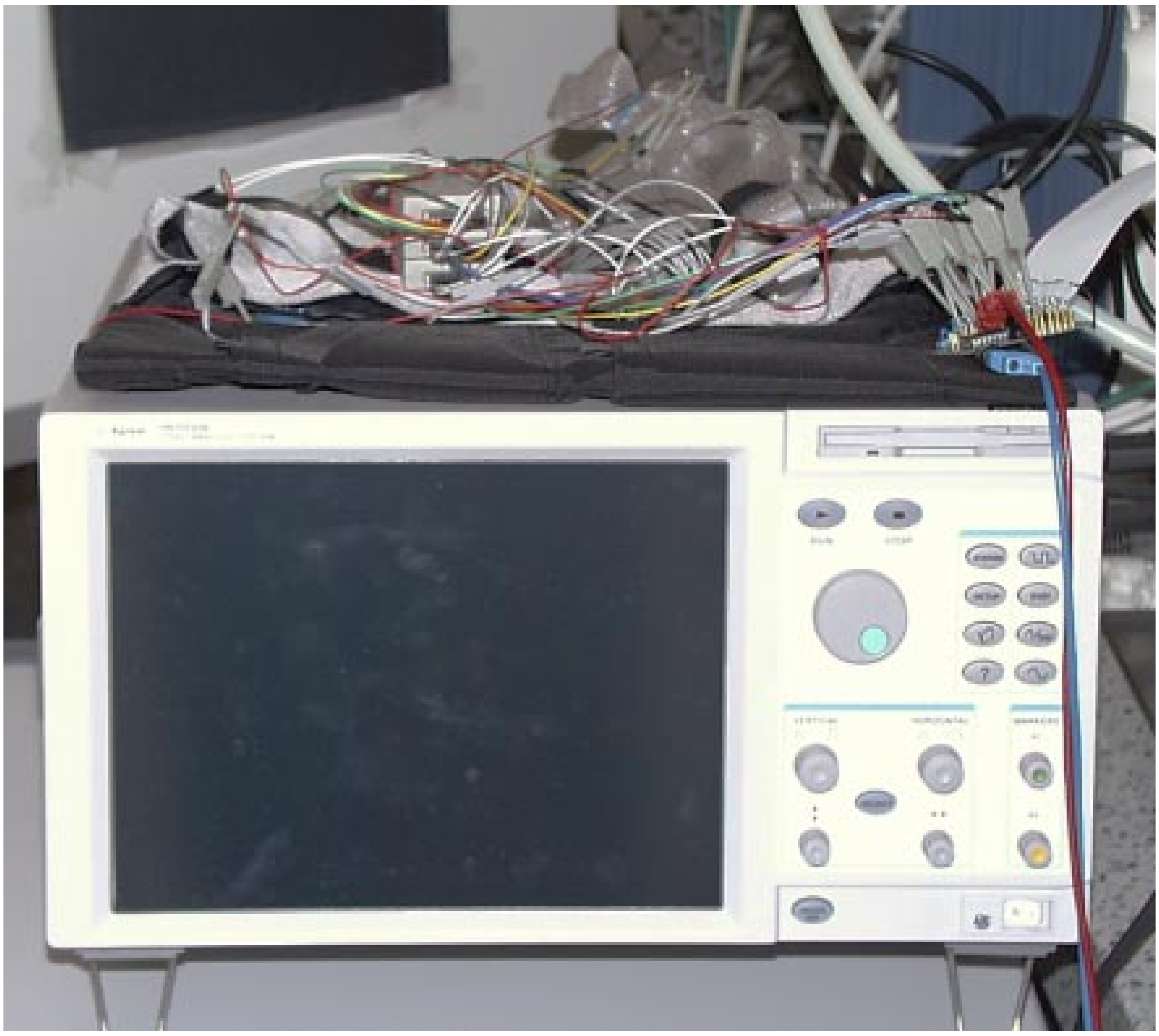
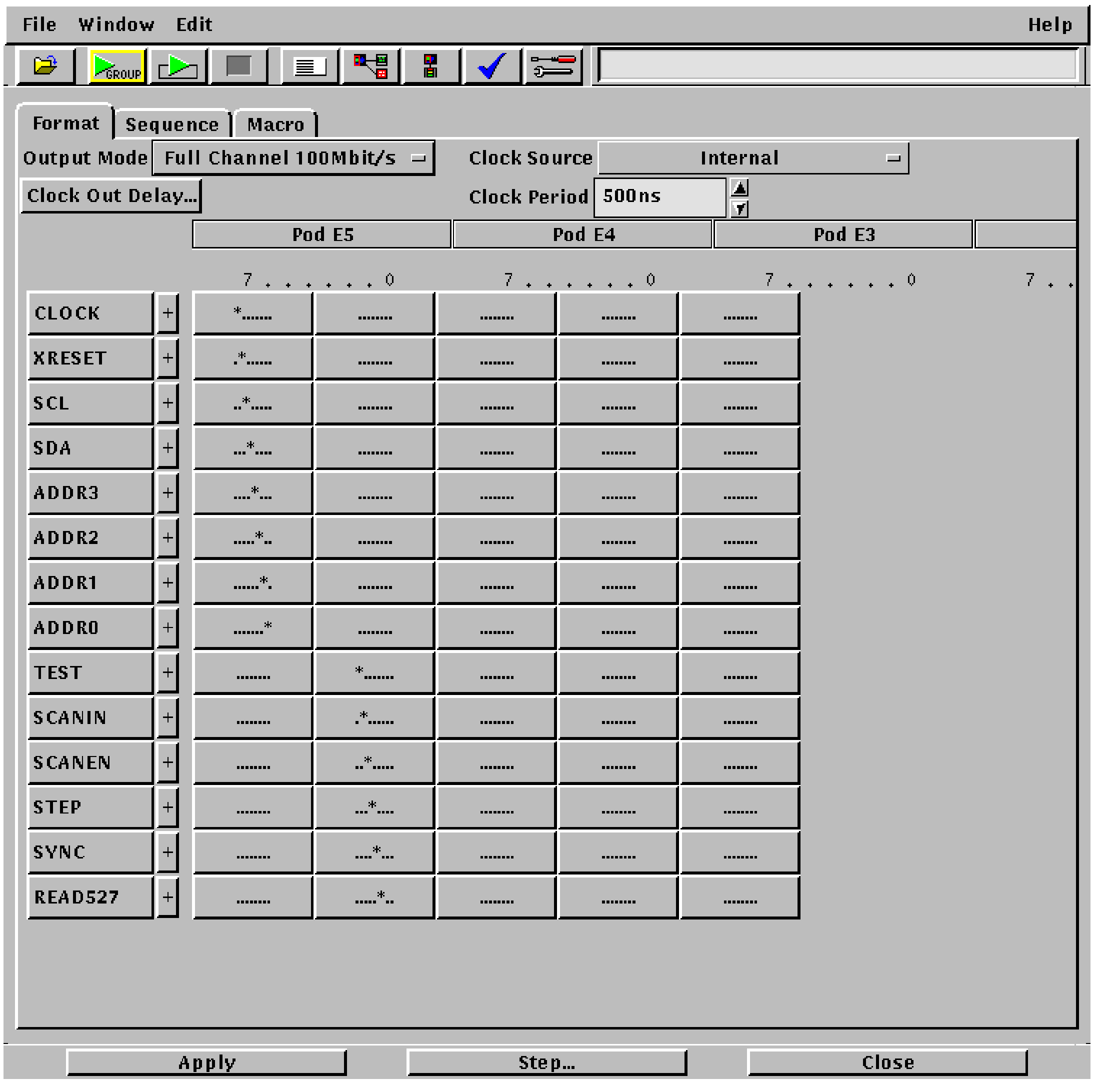


Fig. 8.9: *The Agilent 16702B*

son: the analyzer has a clock input, and it takes a sample on every (rising, falling or both) edge of this clock signal. If we look back to the example of our vector file, we see that the synthesized clock signal doesn’t change after every line. Quite often it stays low for two lines in order to guarantee a secure transition of some signal. Thus, we’d lose lines if we took the synthesized clock as the analyzer clock. So I computed an artificial clock signal which changes after every line: sync. Read527 is used in tests where we have regions of both scan chain filling/reading and normal operation. As we set it to 0 during normal operation, we make sure that we only sample data that is coming from the scan chain.



Output Mode: Full Channel 100Mbit/s

Clock Source: Internal

Clock Period: 500ns

Pod E5 | Pod E4 | Pod E3

7 0   7 0   7 0   7 . .

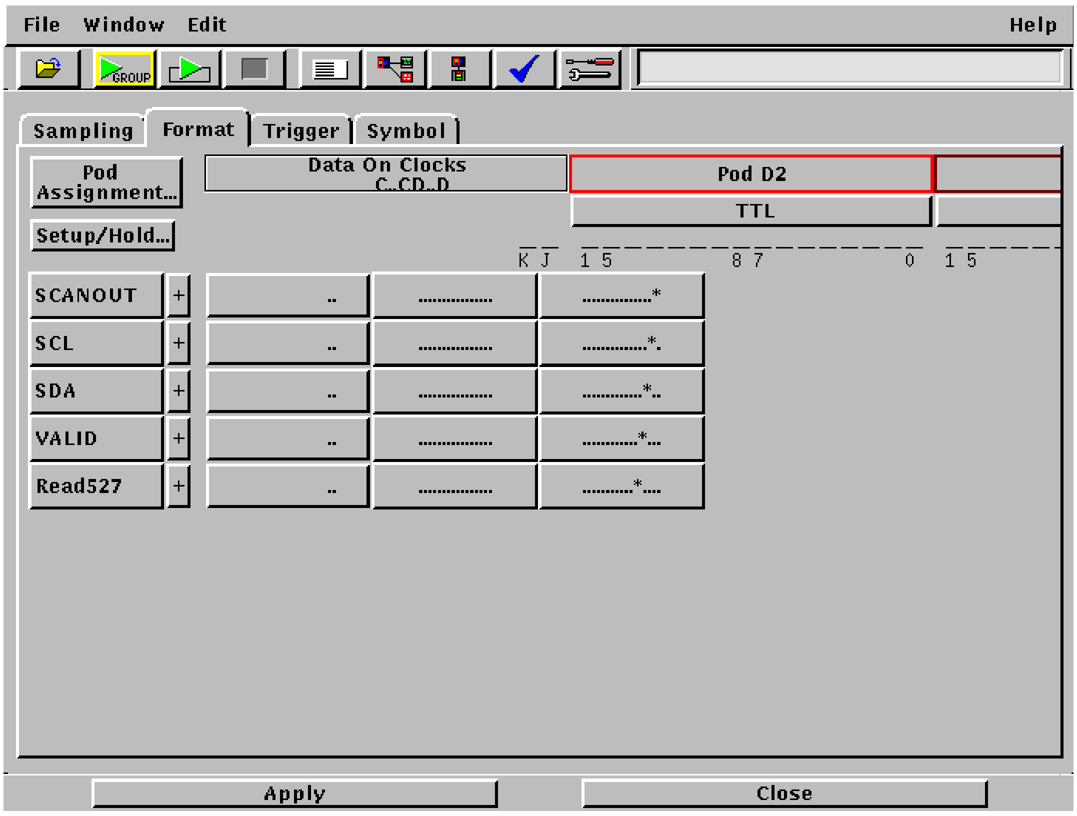
CLOCK	+	*.......	....	....	....	....
XRESET	+	.*......	....	....	....	....
SCL	+	..*.....	....	....	....	....
SDA	+	...*....	....	....	....	....
ADDR3	+	....*...	....	....	....	....
ADDR2	+	.....*..	....	....	....	....
ADDR1	+	......*.	....	....	....	....
ADDR0	+	.......*	....	....	....	....
TEST	+	....	*.......	....	....	....
SCANIN	+	....	.*......	....	....	....
SCANEN	+	....	..*.....	....	....	....
STEP	+	....	...*....	....	....	....
SYNC	+	....	....*...	....	....	....
READ527	+	....	.....*..	....	....	....

Fig. 8.10: Generator pod assignment

Fig. 8.11 shows the port allocation of the analyzer. We use 5 bits of pod D1. We also have two additional signals: valid has no specific function and exists only because we need a corresponding signal in the compare block. It can be used to monitor any signal of interest. Read527 connects to the generator output of the same name.

8.2.5 WHAT TO TEST?

Before we continue with a detailed description of the testing, we have to think about what we want to test at all. When we recall what we have learned about the



		Data On Clocks C..CD..D	Pod D2 TTL	
		K J	15 8	7 0
SCANOUT	+	..	....	.......*
SCL	+	..	....	......*.
SDA	+	..	....	.....*..
VALID	+	..	....	....*...
Read527	+	..	....	...*....

Fig. 8.11: *Analyzer pod assignment*

SMC in the preceding chapters, we see that we have three big parts: the digital circuitry, the asynchronous counters, and the analog parts, which means the sensors themselves.

8.2.6 Digital part

We have learned that whole digital part can be covered with the synthesized vectors, excluding the asynchronous counters. From the synthesized synopsys vectors, we know what the chip is supposed to answer to each line. Now the built-in compare function of the 16702 can be used which takes the input from the analyzer and compares it in a user-defined way with some preloaded reference file. If we then filter out the lines that differ, it can be seen if the digital part has errors.

In Fig. 8.12 all relevant parts can be seen: the generator, the analyzer, the compare tool, a filter and the file out tool. The generator and the analyzer are timing locked with what is called a group run in 16702's terminology.

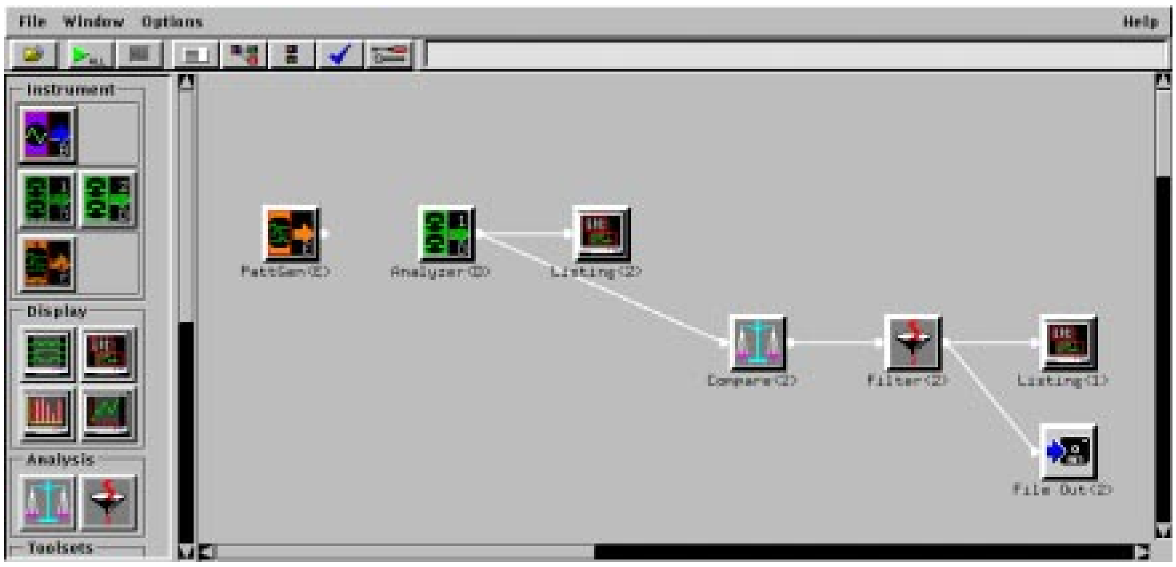
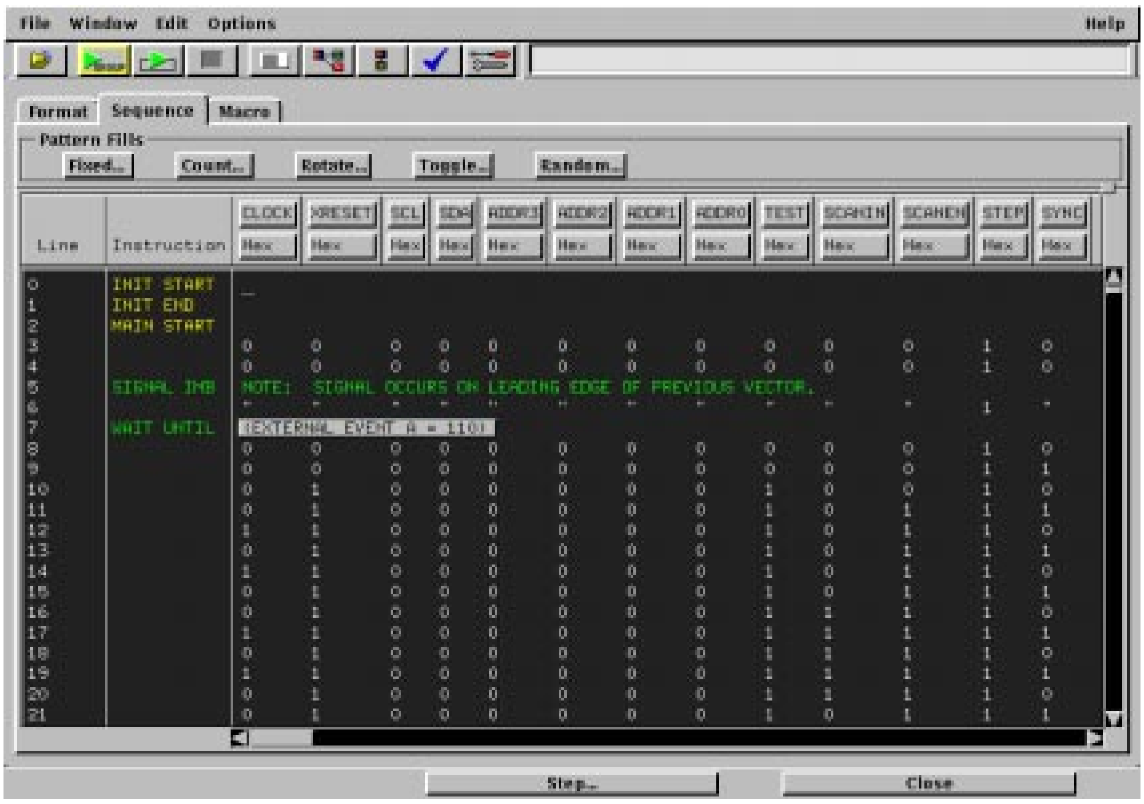


Fig. 8.12: The testing configuration

The generator is started as soon as it receives a signal from the wafer prober. For this purpose, we have a small Labview program on the computer which is controlling the prober. It waits for the step signal of the generator, steps the prober to the next reticle, and then pulls down a ‘ready’ line which signals the generator to start.

In Fig. 8.13 we see the beginning of the generator sequence. The Signal IMB event in line 5 tells the analyzer to become ready for input. The Wait Until in line 7 stops until the prober signals ‘ready’. The analyzer is armed by the IMB signal, and is set to trigger on each edge of the sync signal (Fig. 8.14). We have to put it into state mode as the timing mode will sample with an internal clock. The output of the analyzer then goes to the compare tool. This is configured in a way to compare SCL, SDA and ScanOut in each line to the corresponding values in the reference file (Fig. 8.15). Additionally, we tell it to display only those lines which have a difference in any of these three bits. So now we have all lines which could be proof of a potential fault in our circuit. But remember all those Xs in our vector file? For most of the lines, synopsys doesn’t care what the correct answer is. So we have to filter in a second step for only those lines with a valid test. This is what we need the valid flag for. The filter tool (Fig. 8.16) passes only those lines which have valid set to 1. Thus, at the input of the file out tool, we have only

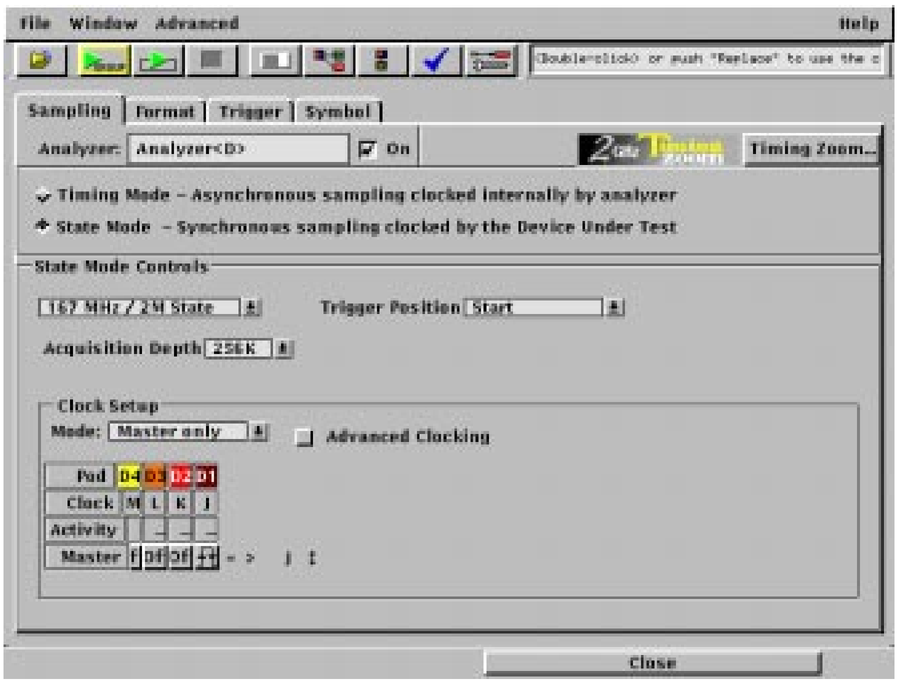


Line	Instruction	CLOCK	XRESET	SCL	SDA	ADDR3	ADDR2	ADDR1	ADDR0	TEST	SCANIN	SCANEN	STEP	SYNC
		Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex
0	INIT START	_												
1	INIT END													
2	MAIN START													
3		0	0	0	0	0	0	0	0	0	0	0	1	0
4		0	0	0	0	0	0	0	0	0	0	0	1	0
5	SIGNAL IMB	NOTE: SIGNAL OCCURS ON LEADING EDGE OF PREVIOUS VECTOR.												
6		"	"	"	"	"	"	"	"	"	"	"	1	"
7	WAIT UNTIL	(EXTERNAL EVENT A = 110)												
8		0	0	0	0	0	0	0	0	0	0	0	1	0
9		0	0	0	0	0	0	0	0	0	0	0	1	1
10		0	1	0	0	0	0	0	0	1	0	0	1	0
11		0	1	0	0	0	0	0	0	1	0	1	1	1
12		1	1	0	0	0	0	0	0	1	0	1	1	0
13		0	1	0	0	0	0	0	0	1	0	1	1	1
14		1	1	0	0	0	0	0	0	1	0	1	1	0
15		0	1	0	0	0	0	0	0	1	0	1	1	1
16		0	1	0	0	0	0	0	0	1	1	1	1	0
17		1	1	0	0	0	0	0	0	1	1	1	1	1
18		0	1	0	0	0	0	0	0	1	1	1	1	0
19		1	1	0	0	0	0	0	0	1	1	1	1	1
20		0	1	0	0	0	0	0	0	1	1	1	1	0
21		0	1	0	0	0	0	0	0	1	0	1	1	1

Fig. 8.13: The generator sequence

those lines left which differ in at least one bit from the reference and are marked as actual test lines in the vector file. They are written to disk by the file out tool.

But how do those files make it into the generator and compare tool, and who sets the valid bit accordingly? Note that we have D, U, N, L, H and X in our synopsys file, but only 0 and 1 in the files the 16702 is reading. The work is done by a Perl script that translates the synthesized vector files into two output files: one for the generator and one for the compare tool. The program (Appendix 1) first generates the headers for the output ASCII files, then takes each line of the input file, splits it up into the parts for the generator and the compare, translates the characters, and writes the output into two files. Additionally, it helps us to overcome a limitation of our current generator card: as it is a demo device, we have only memory for 256k lines. So we have to split our 880’000+ lines into several parts. Of course we cannot do this at an arbitrary position. So the program looks for a reset cycle in the file and uses it to begin a new set of output files. Note that this is not necessary in the final version of the testing as the new generator card will



Analyzer: Analyzer<D> ☑ On

Timing Mode – Asynchronous sampling clocked internally by analyzer

[✓ State Mode – Synchronous sampling clocked by the Device Under Test]

State Mode Controls

167 MHz / 2M State

Trigger Position: Start

Acquisition Depth: 256K

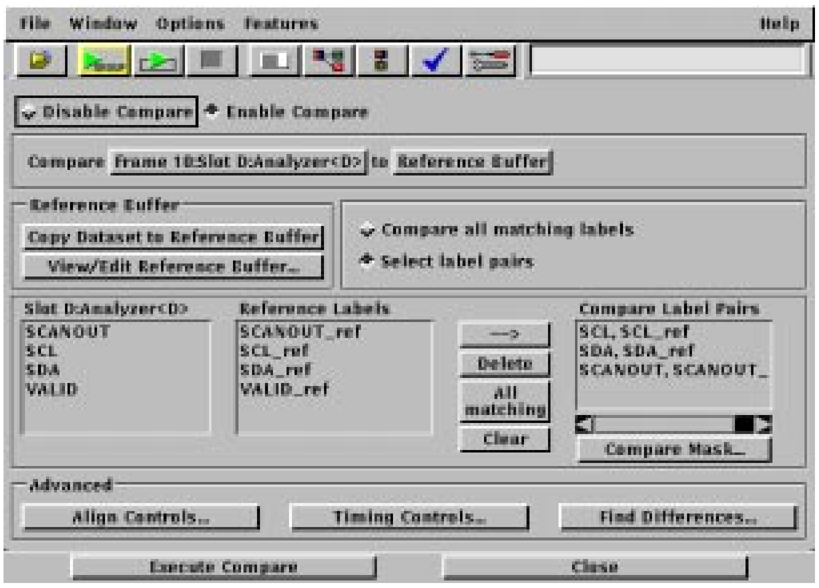
Clock Setup

Mode: Master only

☐ Advanced Clocking

Pod	D4	D3	D2	D1
Clock	M	L	K	J
Activity		–	–	–
Master	f	0f	0f	↑↓

Fig. 8.14: *Analyzer timing settings*



Disable Compare [✓ Enable Compare]

Compare Frame 10:Slot D:Analyzer<D> to Reference Buffer

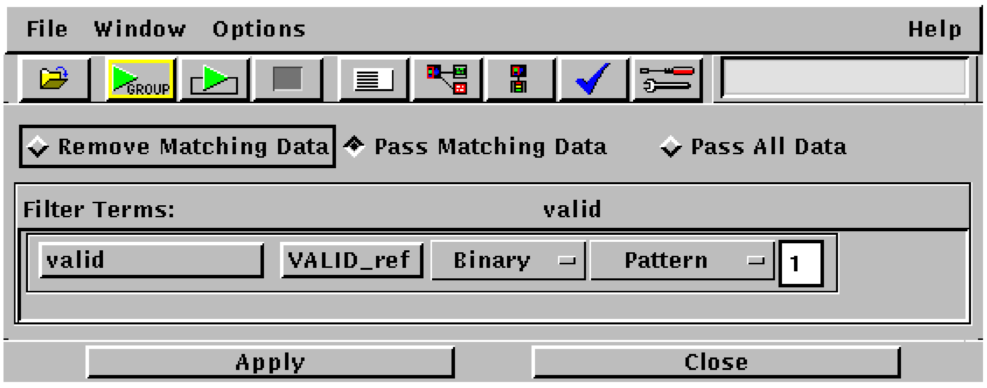
Reference Buffer

Compare all matching labels

[✓ Select label pairs]

Slot D:Analyzer<D>	Reference Labels	Compare Label Pairs
SCANOUT	SCANOUT_ref	SCL, SCL_ref
SCL	SCL_ref	SDA, SDA_ref
SDA	SDA_ref	SCANOUT, SCANOUT_
VALID	VALID_ref	

Fig. 8.15: *The compare tool*



Remove Matching Data [✓ Pass Matching Data] Pass All Data

Filter Terms: valid

valid | VALID_ref | Binary | Pattern | 1

Fig. 8.16: *The filter tool*

have 16M lines. Note that in that case you cannot use the ASCII file directly for the generator. Larger files have to be converted to a specific binary format. There is a C program to do that: asc2pgb.

Our translation program is invoked with the synopsys file and the name of the output as parameters. It then numbers the output files sequentially. Those files are then ftp'd to the generator and loaded into the compare tool and the generator. It is convenient to create a system setting save file for each output file so that testing can be automated.

In the end, if this system setting is executed, it yields an empty file if the test of the digital part went ok and a list of all failing lines if there were faults.

8.2.7 THE ASYNCHRONOUS COUNTERS

We have learned that asynchronous counters a not testable with the scan chain method. But why? An asynchronous counter (see Fig. 8.17) consists of a row of

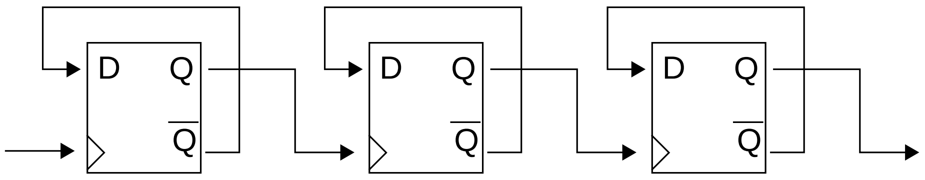


Fig. 8.17: *Asynchronous counter*

flip flops, wired in the way shown below. As such, it forms sort of a scan chain itself. Thus, we have to find other ways for testing it. But as there’s no general solution for this problem, we have to take a look at our circuit (Fig. 8.18).

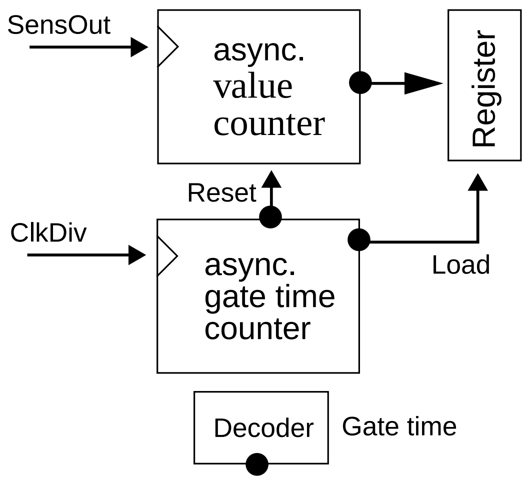


Fig. 8.18: The asynchronous counters on the SMC

We see that the gate time counter is clocked by ClkDiv, that is the system clock divided by 256. The clock input of the value counter is connected to the bitstream output of the sensor. All outputs from the asynchronous counters are synchronized. This way, uncontrollable signals coming from the counters can not reduce the testability of the rest of the circuitry. The small black circles mark flip flops that are in the scan chain. The gate time is normally set via the I2C interface, but is also accessible in the scan chain. So we might come up with the following idea for the testing of the gate time counter: using the scan chain, we set the gate time to the maximum value, 111. Then we switch to normal operation and run the system for as many cycles as are necessary for this gate time (524288 clock ticks). Then we have to check the output at Reset and Load. If they toggle after the exact number of cycles, we know that the counter has counted right. Unfortunately, it’s not that easy in practice: remember that we have the clock divider at the input. So we’d have to wait not 524288 but 524288 * 256 or 1.34E8 clocks. Even the big generator with 16M vectors is way too small for that. The only useful alternative I see would be to use an external clock for that part and use an additional counter which triggers the analyzer after the appropriate number of cycles. However, I haven’t tried that so far.

For the value counter, it's even more difficult as it is clocked by the sensor output to which access is not possible at the moment without changing the design. In my opinion the only possibility to test this counter is to combine it with the test of the sensors: if they deliver useful values all over the parameter space, we have to assume that the counter is working ok.

If these asynchronous counters should show to be a problem, they could be replaced by synchronous counters in a future design. They could be fully integrated into the scan chain and were therefore very easily testable. They do have a drawback though: they need about 20% more chip space.

8.2.8 The sensors

Before we begin with the test of the sensors, we perform one additional test, just to be on the safe side. We take two vectors of 527 0s or 1s, respectively, and clock them into the scan chain. Without performing the usual cycle(s) of normal operation, we immediately clock them out again. In this way we see if the scan chain is working all right. If both vectors don't have any bits toggled, the scan chain is fault free.

Now that we can rely on a working scan chain, which possibilities do we have to test the analog part of the SMC and the sensors? Most of the usual testing schemes for analog circuitry fail for one simple reason: we do not have the pins. As the SMC has only digital in- and outputs, we cannot feed in voltages, measure current consumptions, look at signals with an oscilloscope and the like. Instead, we have to find a way to test the hidden (behind the digital part) analog sensor circuitry solely by using the digital interface. This confinement almost inevitably leads us to a functional test. This means we can't measure the correctness of single parts of circuitry, we only can do dummy measurements and use their results to deduce that a sensor is ok. As these restrictions were already foreseen at the time of design, fortunately there are some features incorporated into the design that allow us to do cross and backup tests. Recalling Fig. 1.3, we see the following:

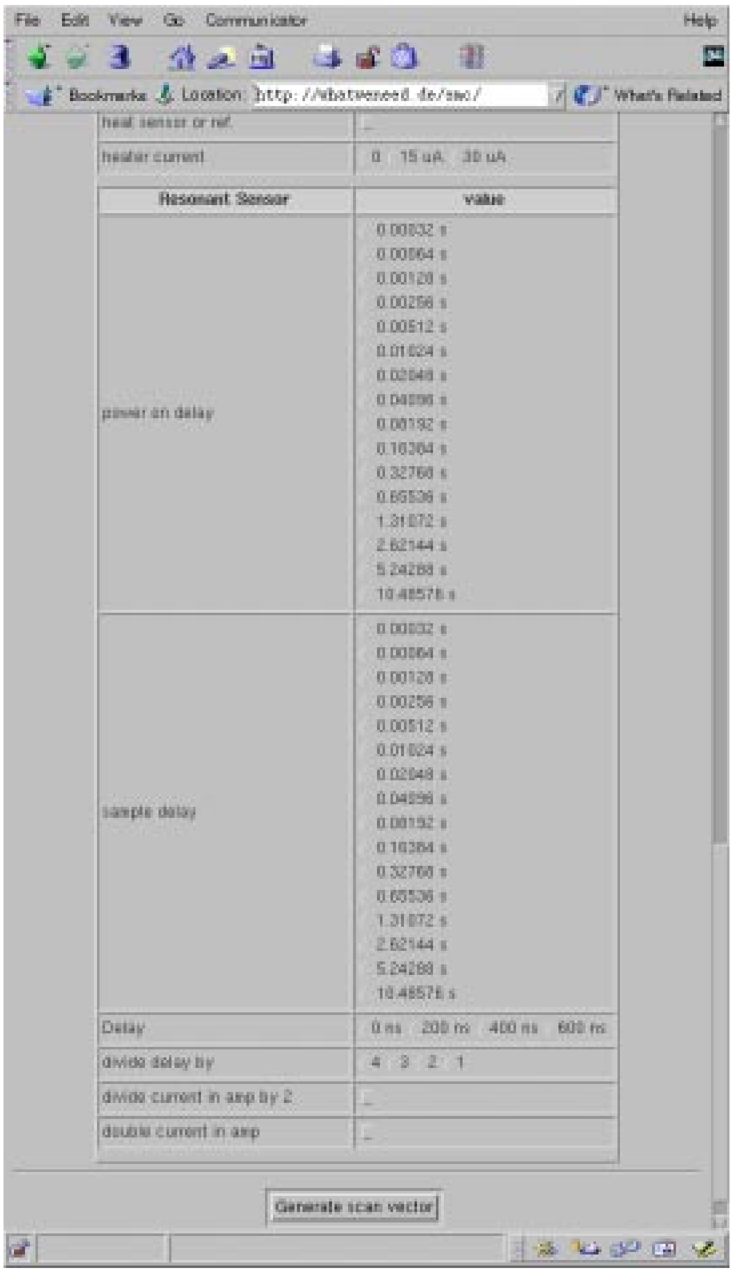
- the temperature sensor can be read out in two ways (via the calorimetric or the capacitive filter bank)
- the output voltage of the calorimetric sensor can be connected to the $\Sigma\Delta$ A/D converter of the capacitive sensor

- the calorimetric sensor has an integrated heater which can be used to simulate a measurement
- for all sensors, we can play with the different options in the respective ComRegs and watch the output for changes.

But how can we control all those features? The easiest way is again the scan chain. In Appendix 6 we see a complete list of all 527 bits. As the parameter registers of the sensors are included in the chain, we can just generate a vector with the desired attributes, clock it into the scan chain, operate the SMC in normal mode for some time and then use the sensor output registers to read out the sensor values. Now it's a rather boring job to set the right bits by hand every time. For that reason I wrote a bunch of PHP (an HTML-embedded server-side programming language) programs (Appendix 2) that provide a nice web interface to all sensor parameters (Fig. 8.19). When the desired values are entered, a click on the generate button compiles the corresponding sequence of 527 0s and 1s. Note that it is not enough to set just the right parameter bits. In order to get the SMC logic into a well-defined state, we have to turn on other bits in the scan chain (e.g. the sensor-power bits). To get this stream into the generator, there's another Perl script (well, actually there's two) that generates the according ASCII file (Appendix 3). The first one, xform527.1wait, takes one sequence, clocks it in, operates the chip in normal mode for one cycle, and reads out the answer. The second one, xform527.progwait, is an extended version. When we have a couple of vectors which we want to test in one run and which require different numbers of wait cycles, we can use xform527.progwait which will read in a list of input files (e.g. vec1, vec2, vec3...), each containing one vector and an integer that denotes the number of cycles to run in normal mode. In both cases, the output is written to a text file.

8.2.9 The capacitive sensor

We start with the capacitive sensor as it is the most simple one to test. If we again have a look at Chapter 2.2, the register bank, we see the possibilities we have for testing: if we have a nonzero output at all (which we hope), we can use ComReg 3 (the gate time), ComReg 4 (the reference capacity) and bits 0 and 1 of ComReg5 (bitstream multiplier) to influence the output. We might expect the fol-


File Edit View Go Communicator Help
Bookmarks
Location: http://whatweneed.de/swc/
What's Related
Generate scan vector

heat sensor or ref.	_
heater current	0 15 uA 30 uA

Resonant Sensor	value
power on delay	0.00032 s 0.00064 s 0.00128 s 0.00256 s 0.00512 s 0.01024 s 0.02048 s 0.04096 s 0.08192 s 0.16384 s 0.32768 s 0.65536 s 1.31072 s 2.62144 s 5.24288 s 10.48576 s
sample delay	0.00032 s 0.00064 s 0.00128 s 0.00256 s 0.00512 s 0.01024 s 0.02048 s 0.04096 s 0.08192 s 0.16384 s 0.32768 s 0.65536 s 1.31072 s 2.62144 s 5.24288 s 10.48576 s
Delay	0 ns 200 ns 400 ns 600 ns
divide delay by	4 3 2 1
divide current in amp by 2	_
double current in amp	_

Fig. 8.19: Web based scan vector generation

lowing behavior: Fig. 8.20. The output should depend on the gate time in the following way:

$$T = \frac{256 \cdot 16 \cdot 2^{x}}{f_{Clk}}$$

where x (0..7) is the gate time value set in the register.

So, using the PHP scripts mentioned above we generate a series of vectors with increasing gate times and feed them to the SMC.

For the reference capacitance, the dependence is expected to be linear, so an ASCII file with a sweep through different values is generated.

Finally, we step through the bitstream multipliers 1, 2, 3 and 4 in the file and hope to see also a linear answer.

Let's take a look at testing time. We assume that the time for clocking the vectors in and out of the scan chain is negligible compared to the testing time, so we do not include it into our estimation. In order to reduce testing time, we might want to test the reference capacitances and the bitstream multiplier with minimal gate time. Then we have 68 (64 for cap., 4 for multiplier) times 4096 (shortest gate time) clock cycles for testing, or about 280 milliseconds. If we want to test the different gate times, we have to wait longer though. As the longest gate time is about 0.5 seconds, we should perhaps not test all of them in order to avoid testing times in the multi-seconds range.

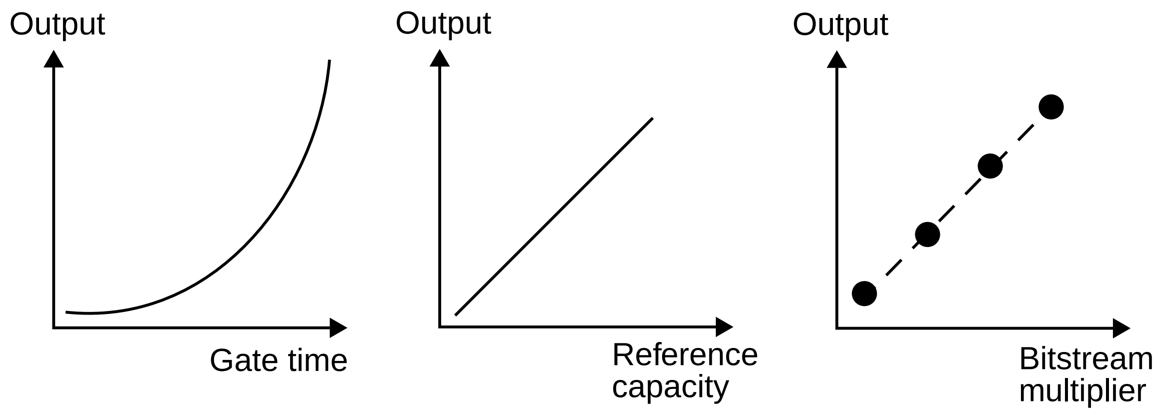


Fig. 8.20: Testing schemes for the capacitive sensor

The output of the capacitive sensor can be processed and verified with the read-out script described in Chapter 8.4, for example.

8.2.10 THE CALORIMETRIC SENSOR

Remember that we have extended analog and digital data processing between the thermopiles and the A/D converter (Chapter 2.3.2 and Chapter 3.1.2). The overall impact of those two filter stages on the sensor signal can be described as a low-pass filter with a maximum signal frequency of about 400 Hz. We can again play with the sensor parameters, but we also want to know if the filters are working. We can use some basic idea of signal processing to come up with a testing scheme: a step function on the input should be modified by the filter in the following way: Fig. 8.21. So we turn on the heater, and in the following several mil-

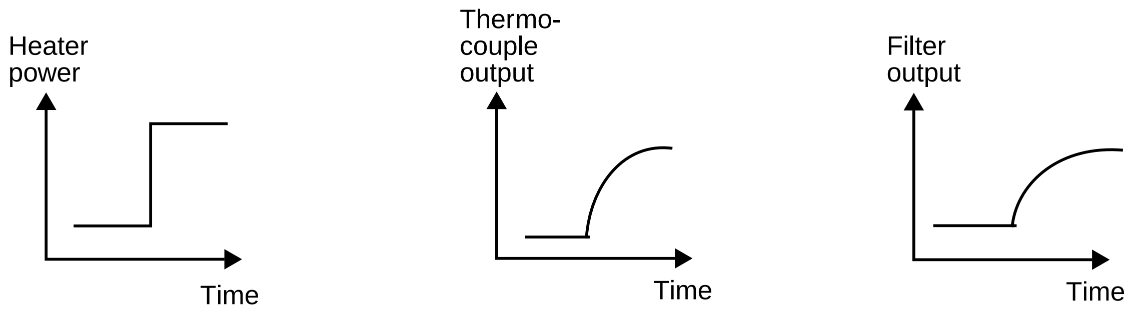


Fig. 8.21: Shape of the calorimetric signal at different points of the filter chain

liseconds read out a lot of values in order to make sure that the output looks something like the right shape in Fig. 8.21. We basically have two options to do so: we can use a long list of read-out vectors in the generator, or we delay that test and use the PC and the µC together with the *read continuous* command after the 16702 has done its job. Once we know that the filter is basically working, we can think of ways of influencing the output in order to see if the rest of the sensor is working: we can use bit 1 and 2 of ComReg3 to divide the output by 4 or 16 which should manifest in the output values. By heating not the sensor, but the reference capacitor via bit 3, the output can be inverted. Bit 4 and 5 allow us to set the heater current to two values. Again we should see the impact in the output values.

Testing time is not so much an issue for the calorimetric sensor as we can get a value every 1024 clock cycles, or about every millisecond. So even if we test some parameters, we are still deep in the millisecond range.

Additionally, it is possible to route the output of the analog filter stage of the calorimetric sensor to the input of the A/D converter of the capacitive sensor (Bit 2 and 3 of ComReg5). In this way, we have two possibilities to test the analog part of the calorimetric and the A/D converter of the capacitive sensor. In this setup, we scan through different gate times between 0 and several milliseconds and check the output of the capacitive sensor for the integral over time of the right curve in Fig. 8.21, which looks almost the same. In this case, the testing time considerations of the preceding sensor are valid of course.

8.2.11 The resonant sensor

The resonant sensor causes the biggest testing problems. As we have seen in Chapter 3.1.3, the feedback circuit should do its job alone. This is fine for normal operation, but it also severely restricts our testing possibilities. In the present design, about the only thing we can do is the following. We set up vectors for all the combinations of delay times (0, 200, 400, 600 ns, bit 0, 1 and 2 of ComReg3), delay currents (1, 2, 3, 4 with bits 3 and 4) and the half (bit 5) and double (bit 6) current. This gives us a 4-dimensional parameter space. For each point of this space, we read out the sensor several times and try to find a stable island where the sensor seems to be vibrating in resonance. Fig. 8.22 gives the idea.

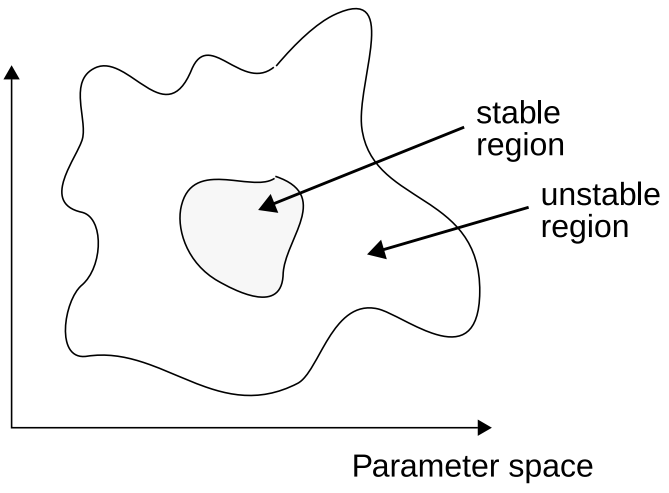


Fig. 8.22: Parameter space of the resonant sensor

The vast parameter space is bad news for testing time, of course. 4 delays x 4 current settings x 2 (double/divide current) x 50 (a reasonable number to guarantee reproducibility) yields 1600 test points. Even with the shortest gate time (4095 clocks), we end up with about 6.5 seconds.

Please refer to Chapter 8.5 for suggestions about how to improve the testing capabilities of the resonant sensor in a future design.

8.2.12 The temperature sensor

The temperature sensor can be read out in two ways: using the counter of the capacitive sensor (Bit 4 ComReg5) or via the filter chain of the calorimetric sensor (Bit 0 ComReg3). Our wafer prober has the nice feature that the temperature of the chuck the wafer is lying on can be controlled. Thus we can check the temperature sensor itself, the counter of the capacitive sensor and the digital filter of the calorimetric sensor. The last test of course implies that we are able to heat up fast enough to see the frequency response of the filter, which might be not possible. Otherwise, we can expect a linear temperature dependence of both output signals.

As the temperature sensor relies on the two other signal paths, the considerations made above apply here for testing time.

8.3 Summary

To sum up the preceding chapters, we provide a schematic flow diagram that quickly tells you which script is used where etc.: Fig. 8.23

8.4 Results

After the PCB was built and all the cables were wired correctly, the setup was ready for the first test. A Perl program allows us to control the testing process from a Unix machine. It uses the RPI (remote programming interface) of the 16702 to communicate via Unix network sockets. The script can be found in Appendix 5, the documentation of the RPI is at [16].

When testing of the first wafer was tried, a short circuit occurred. The current limiter avoided damage to the wafer, but testing was impossible. One of the vdd pads on the wafer wasn’t connected to the digital or analog circuitry, but instead

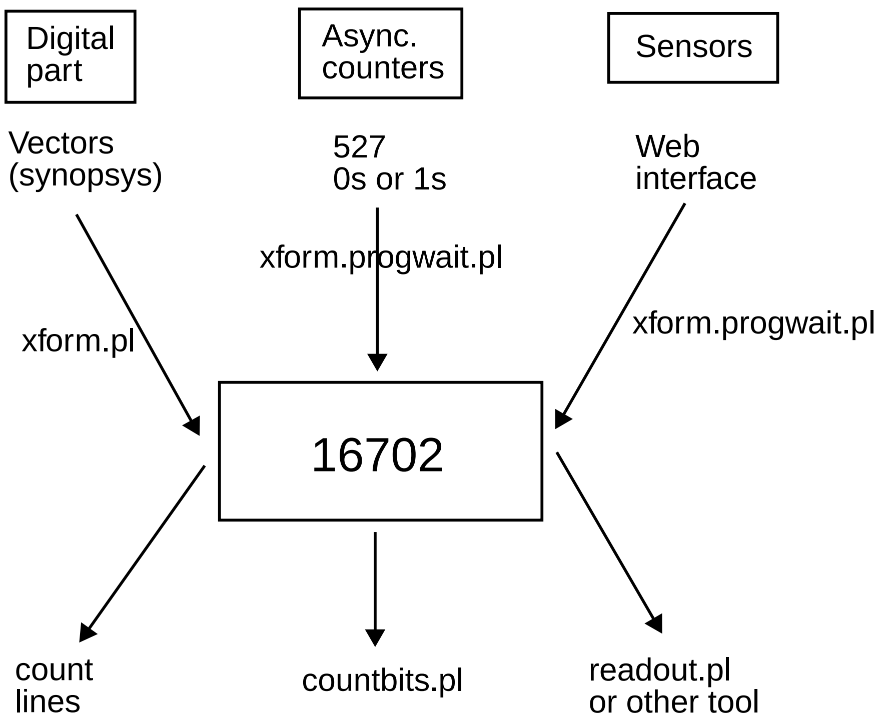


Fig. 8.23: *Testing overview*

contacted to the n-well of the whole wafer. This is necessary for the resonant sensor to work. But in our case, the wafer was not entirely etched, so the positive n-well and the grounded substrate built a nice diode over the whole wafer area. The obvious solution to this problem - to disconnect the n-well from vdd, for some reason didn't work. As a consequence, wafer tests had to be skipped for the beginning and single chips had to be tested while waiting for the second wafer run to be etched.

A later inspection revealed that the waver can be used if the supply voltage is increased slowly instead of turning it on abruptly.

The first digital test was performed with some 54'000 lines. The evaluation of the test is very easy: if the output file is empty, no error occurred. Unfortunately, about 130 of the 54'000+ lines failed. We might propose a possible explanation: the synopsys tool assumes the outputs of the sensors to be zero during digital test. In the current design, this is not ensured. We will see in the second run, where additional multiplexers ground the sensor outputs during digital test, if our idea was right. Other digital tests with longer vectors files showed even higher percentages of failing vectors. Several chips showed the same failing vectors, so we at least have reproducibility.

We discussed the theory of testing the asynchronous counters in Chapter 8.2.7. As we have seen there, our equipment is not able to handle the large amount of clock cycles necessary to test the gate time counter. We will talk about a possible future solution to this problem in the next paragraph. For the time being, we have to rely on the sensor testing.

The basic scan chain tests with 527 0s or 1s was working. The script countbits (Appendix 4) counts all the bits in the chain. If it's not 0 or 527, we're in trouble...

As it is not very convenient to parse the output of the sensor test by hand, a Perl script was used that reads in the output file, extracts chunks of 527 bits with valid sensor data, calculates the numeric sensor value for all sensors, and invokes gnuplot to provide a graphical output of the sensor values. The script can be found in Appendix 4 and a sample output is shown in Fig. 8.24.

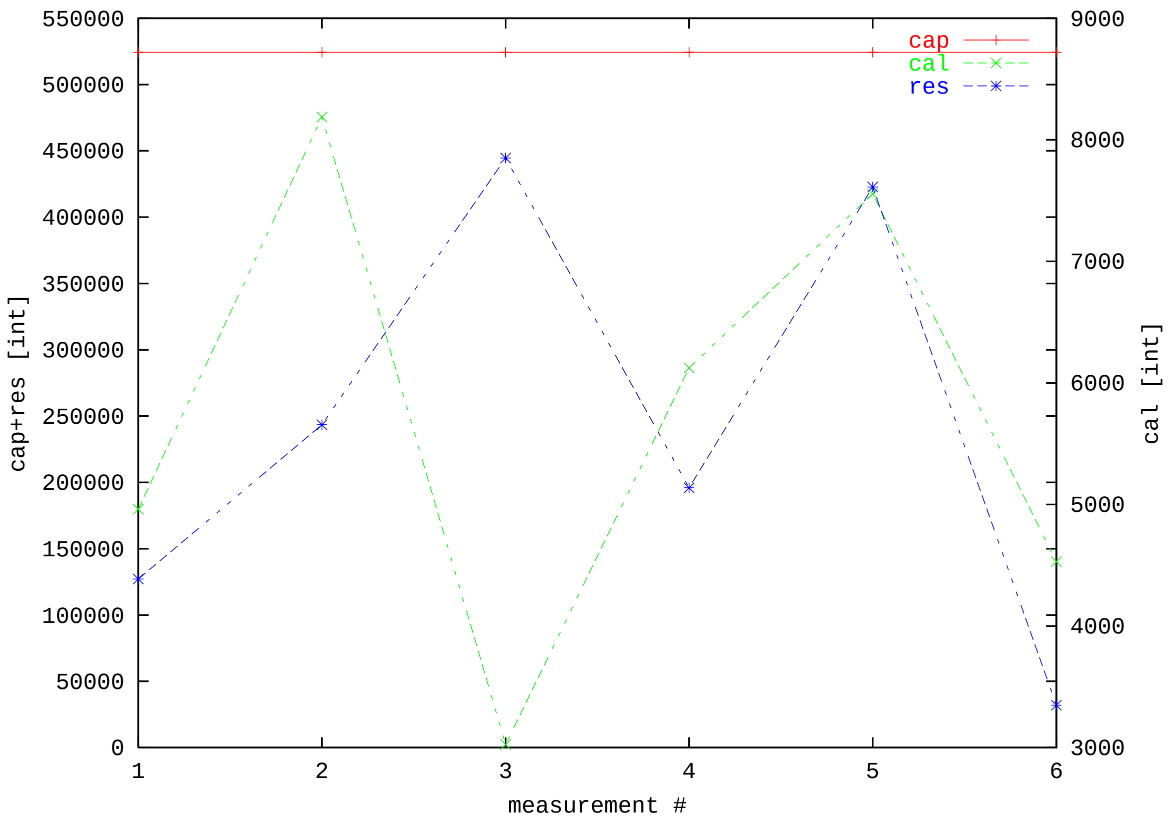


Fig. 8.24: Sample output of the readout script

8.5 Future improvements

We have seen that above all, there are two critical steps in the testing process: the asynchronous counters require too many lines, and the resonant sensor is not testable by design. Therefore I propose two possible design changes for future runs.

8.5.1 Asynchronous counters

The first idea to reduce the number of generator lines needed for testing is the implementation of a flip flop that allows to bypass the clock divider at the input of the gate time counter. This reduces the number of lines by a factor of 256, leaving only 524288 lines left. Our generator can handle that, and it's then only a question of testing time. As the ability to run in normal mode independently of the generator would be useful for other testing tasks too, we might also propose an external counter and clock generator. The generator clocks in the scan chain, tells the counter how many cycles to go, turns off the ScanEn, and triggers the counter. It then waits for another Wait_Until event from the counter. The counter uses the external clock to run the SMC in normal for the specified time and then signals the generator to go on with reading out the answer. Fig. 8.25 gives the idea. No external counter would be needed if the generator was able to loop for a

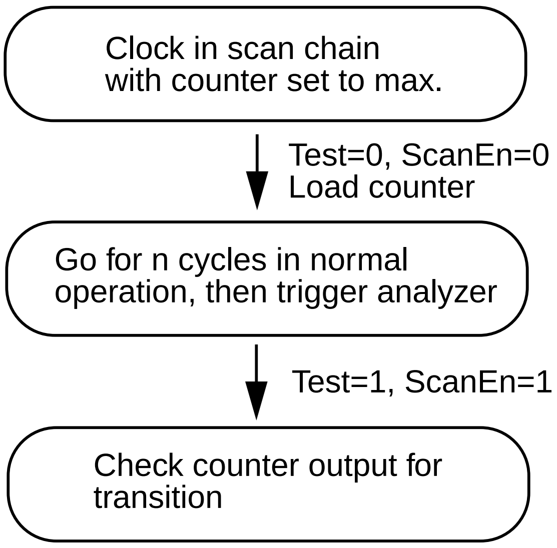


Fig. 8.25: Flowchart of the testing setup with an external counter

certain number of times. The 16702 does have a loop function, but it is expanded before running and therefore also causes the memory problem.

8.5.2 The resonant sensor

The problem with the resonant sensor is that there is no means to test the delay line. My first idea was to connect the delay line to the gate time counter and the system clock to the value counter. Then we insert a bit into the delay line and simultaneously reset the two counters. The value counter then counts the clock cycles until the delayed bit out of the delay line triggers the gate time counter and resets the value counter. In this way, we would be able to measure the delay time in multiples of clock cycles. Unfortunately, the delay is about 1.4 µs, and the clock is 1.25 µs. Thus, there’s no way in realizing this idea without overclocking the SMC considerably.

Fortunately, there is another idea that is very probable to work out. If we connect the output of the delay line to its input via an inverter, we get a system with positive feedback that is supposed to oscillate. We can directly use the existing counter setup to measure the frequency and use it to deduce the delay time.

If the delay line works, the rest of the analog circuitry can be verified by using the heater to generate the pulse. In this way, we can test the whole chain from heater to counter.

8.5.3 Other ideas

While we’re at it, we may propose another small design change that allows us to test the calorimetric sensor more thoroughly. We have seen that the chopper frequency is created by an oscillator on-chip. By the standard testing scheme developed above, we can tell if the chopper is working or not. However, we cannot make any statement about the chopping frequency. Therefore, we might just add another multiplexer that connects the output of the oscillator to the counter of the capacitive sensor, and the frequency can be determined.

9 GAS MEASUREMENTS

9.1 DESCRIPTION AND RESULTS

After all the discussions in the preceding chapters, we should not forget that the SMC is a gas sensor. Therefore gas measurements are interesting. Of course it's not convenient to use the tester to read out the sensor values in a gas measurement as that takes several hours. In Chapter 2, we discussed the I2C interface and also mentioned some µC and PC software that allows us to set the registers and read out the sensors over a long period of time. So all we need is some supply infrastructure for the SMC. A PCB was designed that holds the SMC in a DIL48

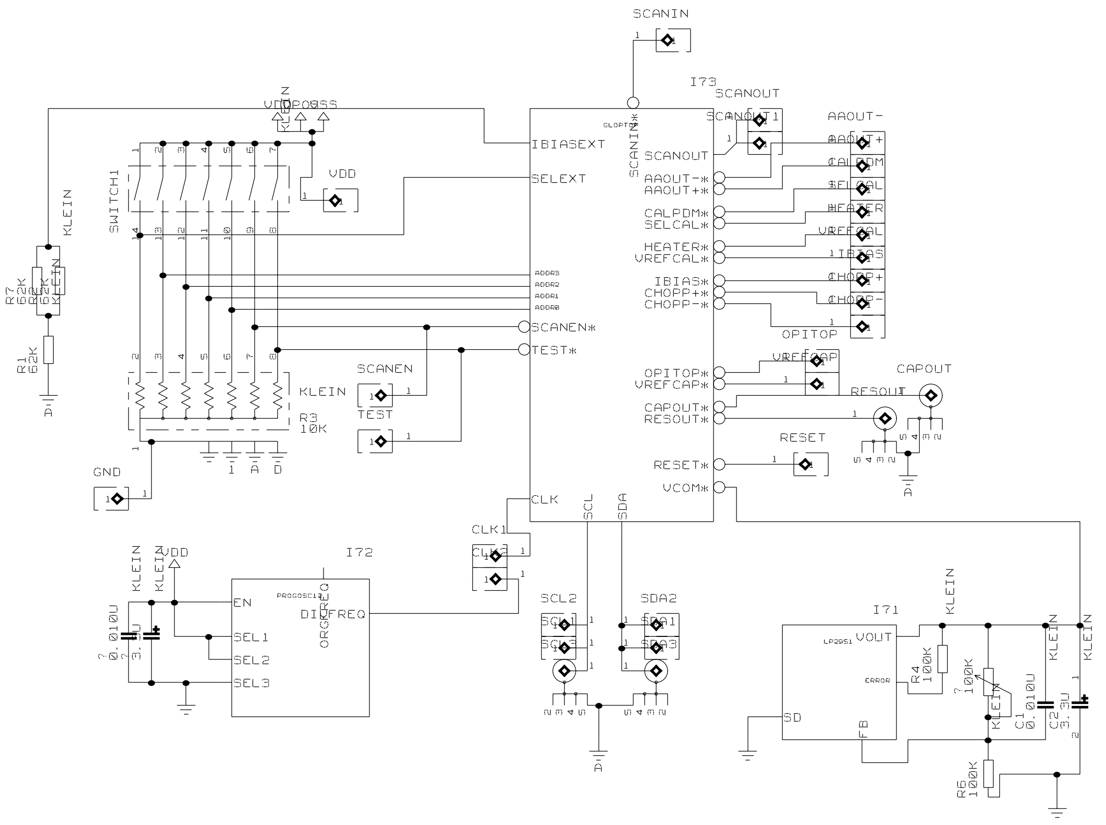


Fig. 9.1: Schematic of the supply PCB

package into which it is wire bonded. The schematic of the board (Fig. 9.1) contains the following main parts:

- the DIP switch in the upper left allows for control of some settings like I2C address or test mode
- the oscillator in the lower left provides a quartz-stabilized 800 kHz square wave as the main clock for the SMC
- in the lower right, we recognize the LP2951, the programmable voltage regulator that provides 2.2 V for V_{com}
- in the centre, there is the DIL package of the SMC with all the connectors that allow for access to signals for testing purposes.

The bonding diagram for the SMC can be seen in Fig. 9.2. The resulting pinout

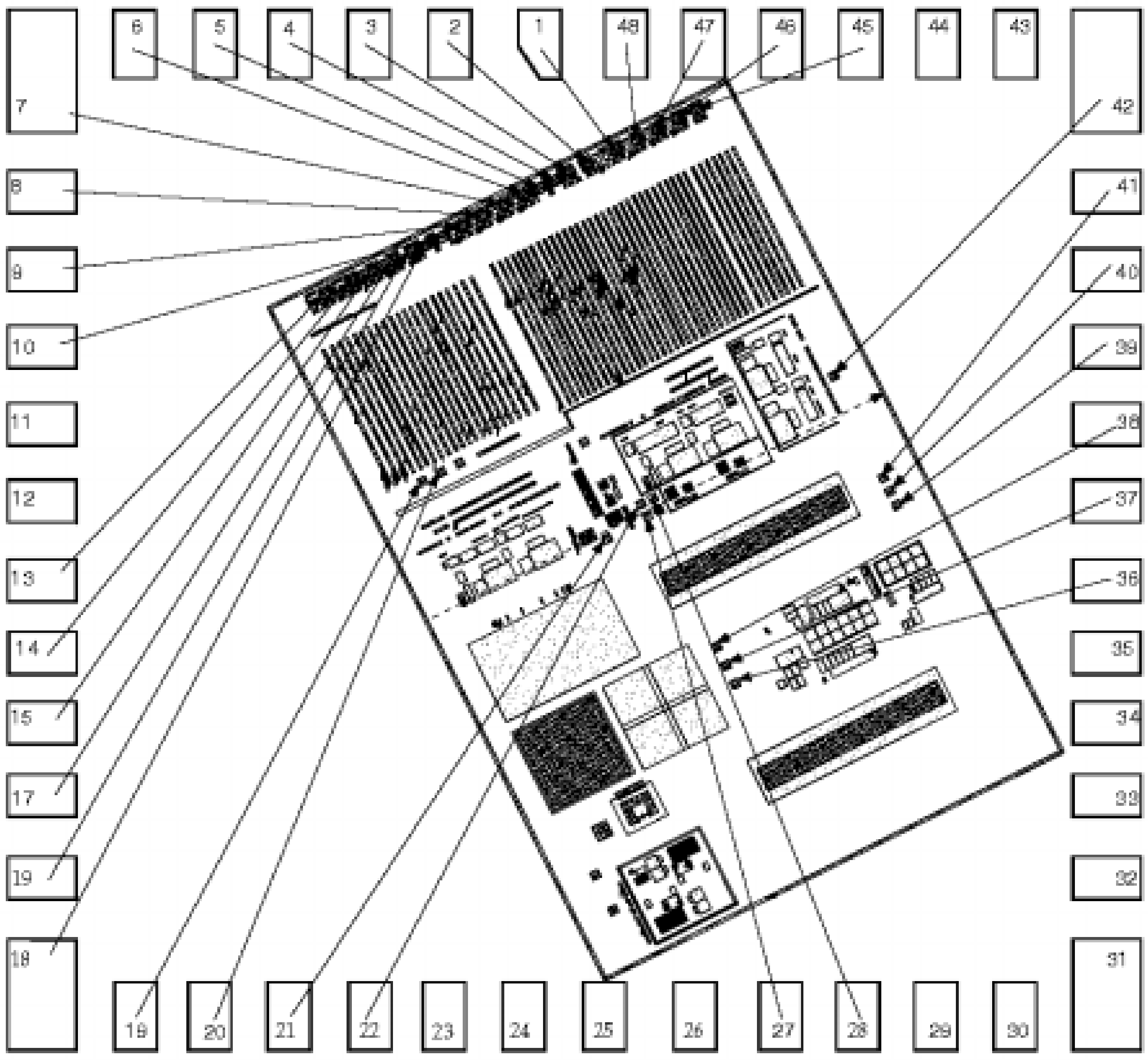


Fig. 9.2: Bonding diagram for the globtop version of the SMC

of the DIL48 package is shown in Tab. 9.1.

For the actual gas measurement, only few connections to the board are necessary: 5 V power of course, but then only SDA and SCL, the data and clock lines for the I2C bus. A photograph of the PCB connected to the µC can be found in Fig. 9.3.

Tab. 9.1: Pinout of the SMC in a DIL 48 package

Pin	Signal	Signal	Pin
1	ScanOut	ScanIn	48
2	SelExt	Test	47
3	IBiasExt	ScanEn	46
4	vdd	n-well	45
5	gnd		44
6	Addr3		43
7	Addr2	CalPDM	42
8	Addr1	AAout-	41
9	Addr0	AAout+	40
10	vdd	SelCal	39
11		Heater	38
12		Chopp-	37
13	SCL	Chopp+	36
14	SDA		35
15	Clk		34
16	Reset		33
17	Vcom		32
18	gnd		31
19	CapOut		30
20	ResOut		29
21	VRefCap	IBias	28
22	OPITop	VRefCal	27
23			26
24			25

When the PCB and the PC are connected to the µC, we are ready for gas measurements. PEL has a computer controlled measurement table that allows to mix several analytes [15] in exact proportions into a carrier gas and expose the sensor to that mixture (Fig. 9.4). A Labview program is used to step through a series of several analytes in different concentrations. Exhaustive measurements are

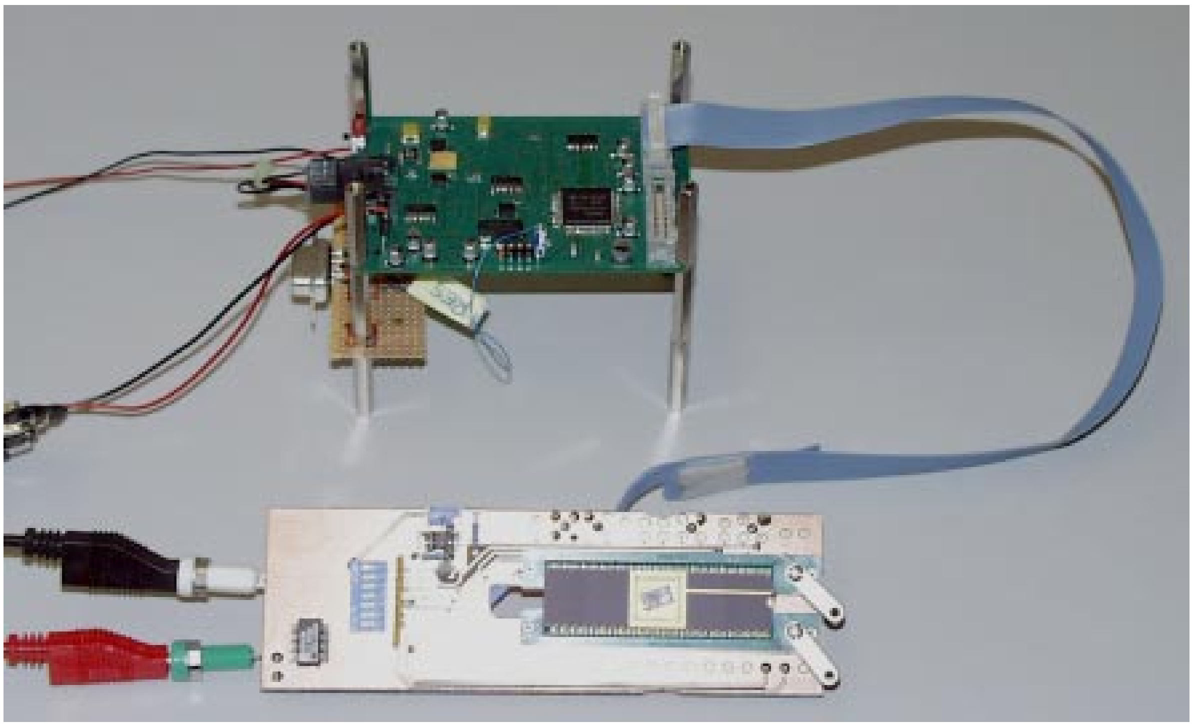


Fig. 9.3: The µC board and the PCB ready for gas measurement

beyond the scope of this thesis, but several publications are available from PEL that cover all aspects of the different sensors [6],[7],[8]. Due to design problems of the of the current version of the SMC (for the capacitive and the resonant sensor, the output value in the DataOut register is immediately overwritten by zeros), I could only read out the calorimetric sensor. Therefore for my gas measurements I used an older design that already has the same capacitive sensor and I2C interface as the SMC.

We set up a measurement with four different analytes (water vapor, ethanol, n-octane and toluene) where for each analyte we first increase the concentration in four steps and afterwards decrease it again in the same four steps. This is a typical measurement scheme used at PEL and yields results as shown in Fig. 9.5. The sensor was coated with the PEUT polymer. The measurement ran four about 6.5 hours. Each concentration is applied for 5 minutes and is then followed by five minutes of purging with carrier gas (synthetic air). The four organ shaped patterns can easily be recognized. Even without a detailed data analysis, we can note several things:

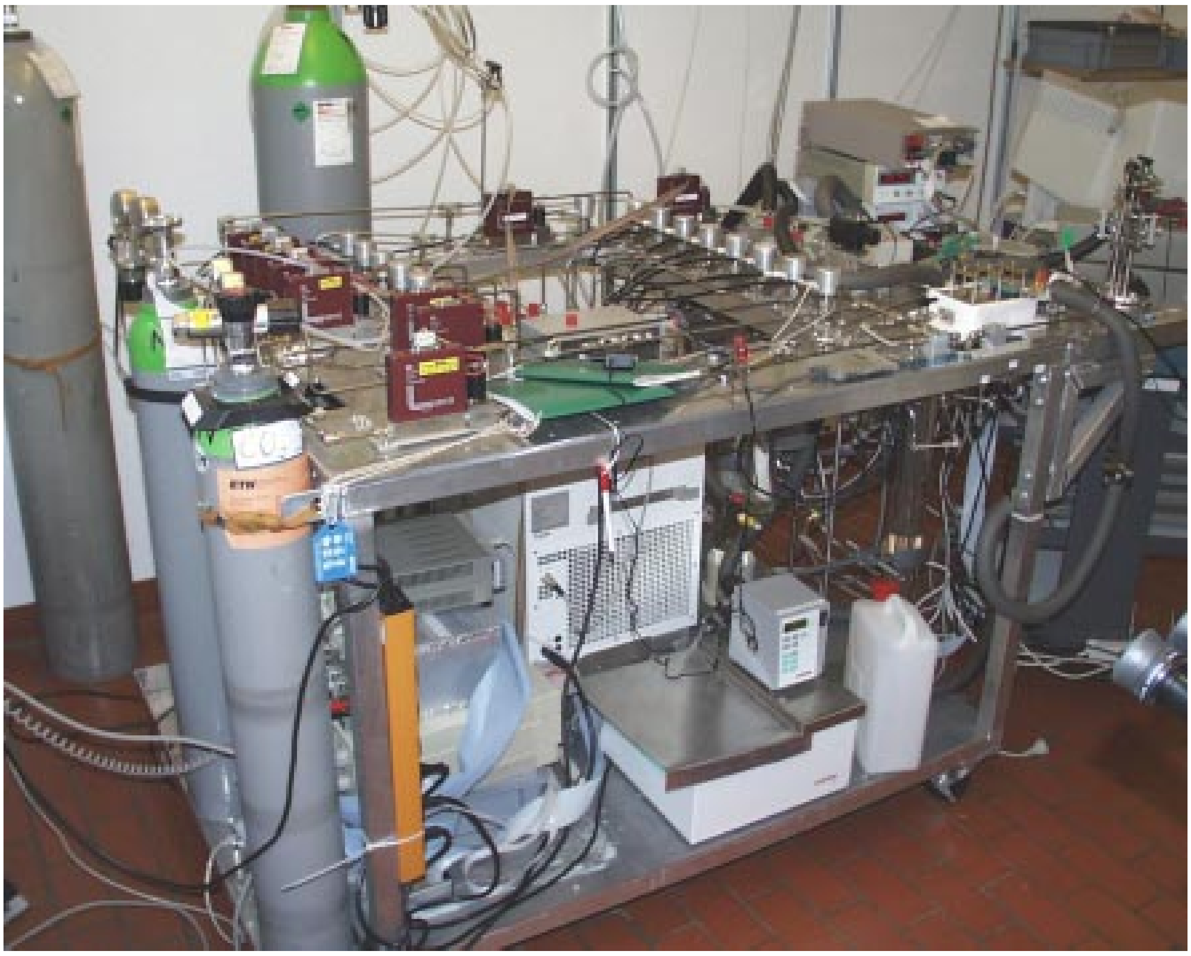


Fig. 9.4: PEL’s gas measurement table

- the signal to noise ratio, that is the height of the ‘pipes’ compared to the superimposed noise, looks very promising
- the baseline which is measured for the carrier gas is very flat. There is a linear decrease, most probably due to some temperature effects, that can be compensated.
- the different analytes can be distinguished not only by the height of the signal (which of course is useless as long as the concentration is not known), but also by the sign of the deviation from the baseline
- the spikes on the bars for the first analyte (water vapor) are believed to be caused by some problem of the gas measurement table as we were able to confirm them by other sensors.

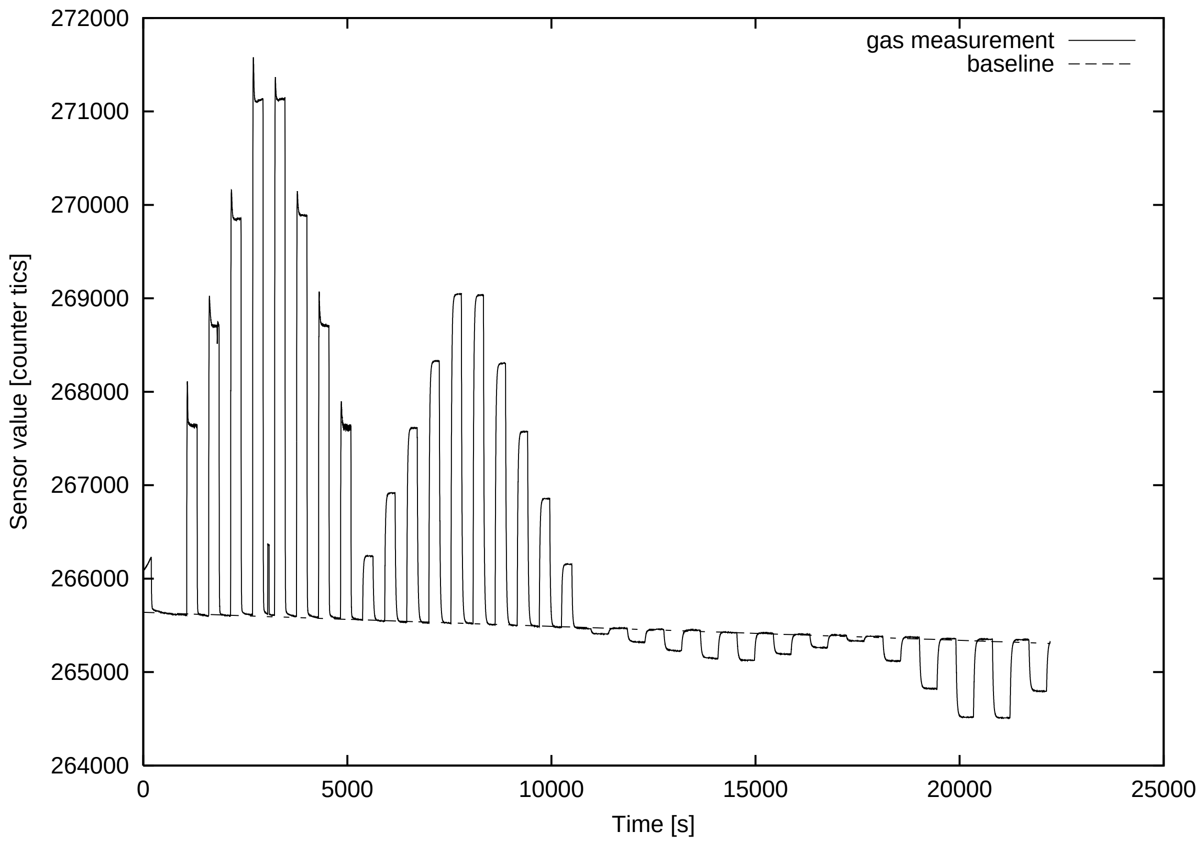


Fig. 9.5: *A gas measurement performed with the capacitive sensor of the SMC*

As the other sensors have been tested in separate designs before, we might be quite confident that the handheld prototype will be working well once the design problems are resolved.

Acknowledgements

I would like to thank the entire PEL team for their kind support. It is a pleasure to work there and I'd be glad to return to do my Ph.D. Special thanks go to Christoph Hagleitner who helped me solve all the problems I encountered during my work and Martin Zimmermann for his patient support in all questions concerning electronics.

REFERENCES

[1] Philips Semiconductors, PCF8584 IC-bus controller datasheet, http://www-us6.semiconductors.com/pip/pcf8584p#datasheet

[2] Michael Morent, Digitales Interface fuer chemische Mikrosensoren, Diploma Thesis 1999, PEL, ETH Zuerich

[3] Rudy van de Plassche, Integrated Analog-to-digital and Digital-to-analog converters, Kluwer Academic Publishers

[4] Christian Menolfi, Qiuting Huang, A Low Noise CMOS Instrumentation Amplifier for Thermoelectric Infrared Detectors, Technical Report No. 97/8, IIS, ETH Zurich

[5] Shoji Kawahito et al., Delta-Sigma Modulation Sensor Interface Circuits with Improved Conversion Gain for Capacitive Readout Chemical Sensors, T.IEE Japan, Vol. 119-E, No. 3, ‘99

[6] Adrian Kummer, Charakterisierung kapazitiver chemischer Mikrosensorren, Diploma Thesis 1999, PEL, ETH Zurich

[7] Nicole Kerness et al., N-well based CMOS Calorimetric Chemical Sensors, Proceedings of MEMS 2000

[8] Dirk Lange et al., CMOS Resonant Beam Gas Sensing System with On-Chip Self Excitation, Proceedings of MEMS 2001

[9] Flavio Heer, A Sigma Delta Temperature Sensor in CMOS technology, Diploma Thesis 2001, PEL, ETH Zurich

[10] Sengupta und Kundu, Intel Technology Journal 1/99 Defect Base Test, http://intel.com/technology/itj/q11999/articles/art_6.htm)

[11] Stanley L. Hurst, VLSI Testing, IEE Circuits, Devices and Systems Series 9

[12] Alfred L. Crouch, Design for Test, Prentice Hall

[13] Mark Burns, Gordon W. Roberts, An Introduction to Mixed-Signal IC Test and Measurement, Oxford University Press

[14] N. Felber, VLSI Design and Testing, Lecture Notes, ETH Zuerich

[15] Andreas Koll, CMOS Capacitive Chemical Microsystems for Volatile Organic Compounds, Ph.D. Thesis, PEL, ETH Zurich

[16] Agilent Technologies, RPI Documentation, `http://software.cos.agilent.com/Svyhlp21/helpRemCntl/ProCom.html`

[17] Larry Wall, Tom Christiansen, Jon Orwant, Programming PERL, O'Reilly

LIST OF FIGURES

APPENDIX 1

XFORM.PL

```
#!/bin/perl -w

use strict;

my ($i, $outfile, $infile, @vect, $f, $flag);
my ($sda, $scl, $clk);
my ($sdaref, $sclref, $scanoutref, $valid);

$f = 1;
$flag = 0;
$clk = 1;

$infile = $ARGV[0] || die “Please invoke with xform <infile> <outfile>!\n”;
$outfile = $ARGV[1] || die “Please invoke with xform <infile> <outfile>!\n”;

open (HEAD, “HEAD”) || die “could not read header\n”;
open (HEADref, “HEADref”) || die “could not read header\n”;
open (VECT, “$infile”) || die “could not open input file\n”;
open (OUT, “>$outfile”.$f) || die “could not create output file\n”;
open  (OUTref,  “>$outfile”.”ref”.$f++)  ||  die  “could  not  create  reference
file\n”;

while (<HEAD>) {print OUT;}#print ASCII file header
while (<HEADref>) {print OUTref;}#print ASCII reference file header


while ($i = <VECT>) {            #for each line
  if (!($i =~ m/^[#\s]/)) {#discard line if it starts with # or is empty
    $i =~ s/.*\s{3,6}//;         #get rid of the timing

    @vect = split(“ “, $i);
    $scl = $sclref = $vect[2];   #extract SCL
    $sda = $sdaref = $vect[3];   #.
    $scanoutref = $vect[14];     #.

    $scl =~ tr/XNDULH/000111/;   #translate chars
    $sda =~ tr/XNDULH/000111/;
    $sclref =~ tr/XNDULH/000101/;
    $sdaref =~ tr/XNDULH/000101/;

    if ($scanoutref eq “X”) {$valid=”0”} else {$valid=”1”}#scanout  is  only
valid if != X
    $scanoutref =~ tr/XLH/001/;                    #translate them
```

```
     $i =~ s/[LD]/0/g;           #make D and L to 0
     $i =~ s/[HU]/1/g;           #make U and H to 1
     $i =~ s/[NX]/0/g;           #make D and L to 0
     @vect = split(“ “, $i);

     if ($vect[1] eq “0” && $flag) {#if we encounter a reset cycle, go to a
new file: overcome the 256k line restriction of the generator
       print OUT “0 0 0 0 0 0 0 0 0 0 0 0 0 “.(($clk = !$clk)?1:0).”\n”;  #print
a 0 on STEP to trigger the waver prober
       print OUTref “0 0 0 0\n”;
       close (OUT);
       close (OUTref);
       close (HEAD);
       close (HEADref);
       open (HEAD, “HEAD”) || die “could not read header\n”;
       open (HEADref, “HEADref”) || die “could not read header\n”;
       open (OUT, “>$outfile”.$f) || die “could not create output file\n”;
       open (OUTref, “>$outfile”.”ref”.$f++) || die “could not create refer-
ence file\n”;
       while (<HEAD>) {print OUT;}                  #print ASCII file header
       while (<HEADref>) {print OUTref;}              #print ASCII reference
file header
       $flag = 0;
     } else {if ($vect[1] eq “1”) {$flag = 1;}}


     print OUTref join(“ “, $sclref, $sdaref, $scanoutref, $valid).”\n”;
#print result to reference file

     print OUT join(“ “, $vect[0], $vect[1], $vect[2], $vect[3], $vect[4],
$vect[5], $vect[6], $vect[7], $vect[8], $vect[13], $vect[15]).” 1 “.(($clk =
!$clk)?1:0).”\n”;#print result to file
   }                                                 #if
}  #VECT

print OUT “0 0 0 0 0 0 0 0 0 0 0 0 0 “.(($clk = !$clk)?1:0).”\n”;#print a 0 on
STEP to trigger the waver prober
print OUTref “0 0 0 0\n”;

print “File $outfile generated. Goodbye\n”;
close (VECT);
close (HEAD);
close (HEADref);
close (OUT);
close (OUTref);
```

APPENDIX 2

WEB BASED SCAN CHAIN GENERATOR

```
===========================================================================
index.php3 shows a table of all options which the user can choose from
===========================================================================

<!DOCTYPE  HTML  PUBLIC  “-//W3C//DTD  HTML  4.0  Transitional//EN”  “ht-
tp://www.w3.org/TR/REC-html40/loose.dtd”>
<HTML>
  <HEAD>
    <TITLE>ScanChain</TITLE>
<link rel=stylesheet type=”text/css” href=”wwn.css”>
  </HEAD>

  <BODY LINK=”#000000” VLINK=”#808080”><BR><BR><Center>
  <FORM ACTION=”generate.php3”>
  <TABLE BORDER=1 CELLSPACING=”2” CELLPADDING=”2”>

  <TR><TH>Capacitive Sensor</TH><TH>value</TH></TR>
  <TR><TD>power on delay</TD><TD>
  <?PHP
   for ($i=0; $i<16; $i++) {
     $time=256*pow(2,$i)/800000;
     echo  “<INPUT  TYPE=radio  NAME=poncap  VALUE=$i  SIZE=0>&nbsp;”.$time.”
s<BR>”;
   }
  ?>
  </TD></TR>

  <TR><TD>sample delay</TD><TD>
  <?PHP
        for ($i=0; $i<16; $i++) {
                $time=256*pow(2,$i)/800000;
                             echo  “<INPUT  TYPE=radio  NAME=sdcap  VALUE=$i
SIZE=0>&nbsp;”.$time.” s<BR>”;
        }
  ?>
  </TD></TR>

  <TR><TD>gate time dec. counter</TD><TD>
  <?PHP
        for ($i=0; $i<8; $i++) {
                $time=256*16*pow(2,$i)*0.00000125;
                             echo  “<INPUT  TYPE=radio  NAME=gtcap  VALUE=$i
SIZE=0>&nbsp;”.$time.” s<BR>”;
```

```
      }
  ?>
  </TD></TR>

  <TR><TD>compensation capacitance (0..63)</TD><TD>
  <INPUT TYPE=text NAME=compcap  SIZE=8>
  </TD></TR>

  <TR><TD>sigma delta reduction factor (0..3)</TD><TD>
  <INPUT TYPE=text NAME=sigdelcap  SIZE=8>
  </TD></TR>

  <TR><TD>EnCapFB (1 for cap!)</TD><TD>
  <INPUT TYPE=checkbox NAME=ecfbcap  SIZE=0>
  </TD></TR>

  <TR><TD>EnCap (1 for cap!)</TD><TD>
  <INPUT TYPE=checkbox NAME=eccap  SIZE=0>
  </TD></TR>

  <TR><TD>Sensor (1 for temp!)</TD><TD>
  <INPUT TYPE=checkbox NAME=tempcap  SIZE=0>
  </TD></TR>

  <TR><TD></TR></TD>

  <TR><TH>Calorimetric Sensor</TH><TH>value</TH></TR>
  <TR><TD>power on delay</TD><TD>
  <?PHP
   for ($i=0; $i<16; $i++) {
     $time=256*pow(2,$i)/800000;
     echo  “<INPUT  TYPE=radio  NAME=poncal  VALUE=$i  SIZE=0>&nbsp;”.$time.”
s<BR>”;
   }
  ?>
  </TD></TR>

  <TR><TD>sample delay</TD><TD>
  <?PHP
        for ($i=0; $i<16; $i++) {
                $time=256*pow(2,$i)/800000;
                             echo  “<INPUT  TYPE=radio  NAME=sdcal  VALUE=$i
SIZE=0>&nbsp;”.$time.” s<BR>”;
        }
  ?>
  </TD></TR>


  <TR><TD>Sensor (1 for temp!)</TD><TD>
  <INPUT TYPE=checkbox NAME=tempcal  SIZE=0>
  </TD></TR>

  <TR><TD>divide gain by 4 (BP)</TD><TD>
  <INPUT TYPE=checkbox NAME=g5bpcal  SIZE=0>
  </TD></TR>

  <TR><TD>divide gain by 4 (Amp)</TD><TD>
  <INPUT TYPE=checkbox NAME=g5pacal  SIZE=0>
  </TD></TR>

  <TR><TD>heat sensor or ref.</TD><TD>
  <INPUT TYPE=checkbox NAME=selhcal  SIZE=0>
  </TD></TR>

  <TR><TD>heater current</TD><TD>
  <INPUT TYPE=radio NAME=curcal VALUE=0 SIZE=0>0
  <INPUT TYPE=radio NAME=curcal VALUE=15  SIZE=0>15 uA
  <INPUT TYPE=radio NAME=curcal VALUE=30  SIZE=0>30 uA
  </TD></TR>


  <TR><TD></TR></TD>

  <TR><TH>Resonant Sensor</TH><TH>value</TH></TR>
  <TR><TD>power on delay</TD><TD>
  <?PHP
   for ($i=0; $i<16; $i++) {
     $time=256*pow(2,$i)/800000;
     echo  “<INPUT  TYPE=radio  NAME=ponres  VALUE=$i  SIZE=0>&nbsp;”.$time.”
s<BR>”;
   }
  ?>
  </TD></TR>

  <TR><TD>sample delay</TD><TD>
  <?PHP
        for ($i=0; $i<16; $i++) {
                $time=256*pow(2,$i)/800000;
                             echo  “<INPUT  TYPE=radio  NAME=sdres  VALUE=$i
SIZE=0>&nbsp;”.$time.” s<BR>”;
        }
  ?>
  </TD></TR>


  <TR><TD>Delay</TD><TD>
  <INPUT TYPE=radio NAME=delres VALUE=0 SIZE=0>0 ns
  <INPUT TYPE=radio NAME=delres VALUE=1 SIZE=0>200 ns
  <INPUT TYPE=radio NAME=delres VALUE=2 SIZE=0>400 ns
  <INPUT TYPE=radio NAME=delres VALUE=3 SIZE=0>600 ns
  </TD></TR>


  <TR><TD>divide delay by</TD><TD>
  <INPUT TYPE=radio NAME=curres VALUE=0 SIZE=0>4
  <INPUT TYPE=radio NAME=curres VALUE=1 SIZE=0>3
  <INPUT TYPE=radio NAME=curres VALUE=2 SIZE=0>2
  <INPUT TYPE=radio NAME=curres VALUE=3 SIZE=0>1
  </TD></TR>


  <TR><TD>divide current in amp by 2</TD><TD>
  <INPUT TYPE=checkbox NAME=halfres  SIZE=0>
  </TD></TR>


  <TR><TD>double current in amp</TD><TD>
  <INPUT TYPE=checkbox NAME=dblres  SIZE=0>
  </TD></TR>


  <TR><TD></TR></TD>

  </TABLE>
<hr>
  <input type=submit value="Generate scan vector">
  </FORM></center>
  </BODY>
</HTML>
```

```
================================================================
generate.php3 takes the choices via http get and compiles the scan vector
================================================================

<!DOCTYPE  HTML  PUBLIC  "-//W3C//DTD  HTML  4.0  Transitional//EN"  "ht-
tp://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
  <HEAD>
    <TITLE>ScanChain</TITLE>
<link rel=stylesheet type="text/css" href="wwn.css">
  </HEAD>

  <BODY LINK="#000000" VLINK="#808080"><BR><BR>

  <?PHP
   $b_poncap  =  decbin($poncap);  for  ($i=strlen($b_poncap);  $i<4;  $i++)
$b_poncap="0".$b_poncap;
   $b_sdcap  =  decbin($sdcap);  for  ($i=strlen($b_sdcap);  $i<4;  $i++)
$b_sdcap="0".$b_sdcap;
   $b_gtcap  =  decbin($gtcap);  for  ($i=strlen($b_gtcap);  $i<3;  $i++)
$b_gtcap="0".$b_gtcap;
   $b_compcap = decbin($compcap); for ($i=strlen($b_compcap); $i<6; $i++)
$b_compcap="0".$b_compcap;
   $b_sigdelcap  =  decbin($sigdelcap);  for  ($i=strlen($b_sigdelcap);  $i<2;
$i++) $b_sigdelcap="0".$b_sigdelcap;
```

```
  if ($ecfbcap==”on”) $b_ecfbcap=”1”; else $b_ecfbcap=”0”;
  if ($eccap==”on”) $b_eccap=”1”; else $b_eccap=”0”;
  if ($tempcap==”on”) $b_tempcap=”1”; else $b_tempcap=”0”;
  $b_cap                                                              =
$b_poncap.$b_sdcap.$b_gtcap.$b_compcap.$b_sigdelcap.$b_ecfbcap.$b_eccap.$b_
tempcap;
  echo $b_cap.” “.strlen($b_cap).”<br>”;

  $b_poncal  =  decbin($poncal);  for  ($i=strlen($b_poncal);  $i<4;  $i++)
$b_poncal=”0”.$b_poncal;
  $b_sdcal   =  decbin($sdcal);   for  ($i=strlen($b_sdcal);   $i<4;   $i++)
$b_sdcal=”0”.$b_sdcal;
  if ($tempcal==”on”) $b_tempcal=”1”; else $b_tempcal=”0”;
  if ($g5bpcal==”on”) $b_g5bpcal=”1”; else $b_g5bpcal=”0”;
  if ($g5pacal==”on”) $b_g5pacal=”1”; else $b_g5pacal=”0”;
  if ($selhcal==”on”) $b_selhcal=”1”; else $b_selhcal=”0”;
  if ($curcal==”0”) $b_curcal=”00”; else if ($curcal==”15”) $b_curcal=”01”;
else if ($curcal==”30”) $b_curcal=”11”;
  $b_cal                                                              =
$b_poncal.$b_sdcal.$b_tempcal.$b_g5bpcal.$b_g5pacal.$b_selhcal.$b_curcal;
  echo $b_cal.” “.strlen($b_cal).”<br>”;

  $b_ponres  =  decbin($ponres);  for  ($i=strlen($b_ponres);  $i<4;  $i++)
$b_ponres=”0”.$b_ponres;
  $b_sdres   =  decbin($sdres);   for  ($i=strlen($b_sdres);   $i<4;   $i++)
$b_sdres=”0”.$b_sdres;
  if ($delres==”0”) $b_delres=”000”; else if ($delres==”1”) $b_delres=”001”;
else   if   ($delres==”2”)   $b_delres=”011”;   else   if   ($delres==”3”)
$b_delres=”111”;
  $b_curres  =  decbin($curres);  for  ($i=strlen($b_curres);  $i<2;  $i++)
$b_curres=”0”.$b_curres;
  if ($halfres==”on”) $b_halfres=”1”; else $b_halfres=”0”;
  if ($dblres==”on”) $b_dblres=”1”; else $b_dblres=”0”;
  $b_res = $b_ponres.$b_sdres.$b_delres.$b_curres.$b_halfres.$b_dblres;
  echo $b_res.” “.strlen($b_res).”<br>”;


  echo “<BR><BR><BR>Resulting scan vector:<BR><BR>”;
  $vector = ““;
  for ($i=1; $i<=99; $i++) $vector=$vector.”0”;
  $vector = $vector.$b_cal;
  for ($i=1; $i<=6; $i++) $vector=$vector.”0”;
  $vector = $vector.$b_cap;
  for ($i=1; $i<=6; $i++) $vector=$vector.”0”;
  $vector = $vector.$b_ponres.$b_sdres.$b_delres.$b_curres.$b_halfres;
  for ($i=1; $i<=20; $i++) $vector=$vector.”0”;
  $vector = $vector.$b_dblres;
  for ($i=1; $i<=345; $i++) $vector=$vector.”0”;
  echo “length: “.strlen($vector).”<br><br>”.$vector.”<BR><BR>”;
  if (strlen($vector)!=527) echo “<font color=red>Attention: scan vector is
not 527 bits long. Pls check settings!”;

echo “<BR>&nbsp;”;
```

```
  ?>
  </BODY>
</HTML>




============================================================================
the stylesheet file
============================================================================

/* Seite */
body {font-family:Helvetica,sans-serif}

/* Links */
a    {font-family:Helvetica,sans-serif; text-decoration:none; color:#808080}

/* Text und Titel */
p    {color:#333333; margin-top:0px}
td   {font-family:Helvetica,sans-serif; font-size: 18pt; color:#333333}
th   {background:#cecece; font-family:Helvetica,sans-serif; color:#333333}
h1   {font-weight:bold; color:#333333; margin-top:0px; margin-bottom:0px}
h6   {font-wight:bold; color:#333333; margin-top:0px; margin-bottom:0px}
```

APPENDIX 3

SCAN CHAIN VECTOR GENERATION

```
============================================================================
xform527.1wait takes a 527bit sequence and compiles it into a 16702 ASCII file
============================================================================

#!/bin/perl

use strict;

my ($infile, $outfile, $vector, $i, $clk);

$infile = $ARGV[0] || die “Please invoke with xform527 <infile> <outfile>!\n”;
$outfile = $ARGV[1] || die “Please invoke with xform527 <infile> <out-
file>!\n”;


open(VECT, $infile) || die “could not open input file\n”;
$vector = <VECT>;
open (OUT, “>$outfile”) || die “could not create output file\ n”;
open (HEAD, “HEAD_scanin”) || die “could not read header\n”;

if (length($vector) != 528) {die “Vector is not 527 bits long: “.(length($vec-
tor)-1).”!\n”;}
$clk = 1;

while (<HEAD>) {print OUT;}                     #print ASCII file header

for ($i=0; $i<527; $i++) {#clock the scan vector into the scan chain (527
bits)
  print OUT ($clk=(not $clk)?1:0).” 1 0 0 0 0 0 0 1 “.substr($vector, $i,
1).” 1 1 “.$clk.” 0\n”;
  print OUT ($clk=(not $clk)?1:0).” 1 0 0 0 0 0 0 1 “.substr($vector, $i,
1).” 1 1 “.$clk.” 0\n”;
}


print OUT ($clk=(not $clk)?1:0).” 1 0 0 0 0 0 0 1 0 0 1 “.$clk.” 0\n”;#oper-
ate clk once w/ scanen=0 in order to process scan
print OUT ($clk=(not $clk)?1:0).” 1 0 0 0 0 0 0 1 0 0 1 “.$clk.” 0\n”;


for ($i=0; $i<527; $i++) {#read out the answer
  print OUT ($clk=(not $clk)?1:0).” 1 0 0 0 0 0 0 1 0 1 1 “.$clk.” 1\n”;
  print OUT ($clk=(not $clk)?1:0).” 1 0 0 0 0 0 0 1 0 1 1 “.$clk.” 1\n”;
```

```
}

close (VECT);
close (HEAD);
close (OUT);


===========================================================================
xform527.progwait takes several 527bit sequence and compiles them into a 16702
ASCII file
===========================================================================

#!/bin/perl

use strict;

my ($name, $infile, $outfile, $vector, $i, $clk, $counts, $filenumber);

$infile = $ARGV[0] || die “Please invoke with xform527 <infile> <outfile>!\n”;
$outfile = $ARGV[1] || die “Please invoke with xform527 <infile> <out-
file>!\n”;

$filenumber=1;

open (OUT, “>$outfile”) || die “could not create output file\ n”;
open (HEAD, “HEAD_scanin”) || die “could not read header\n”;
$clk = 1;

while (<HEAD>) {print OUT;}                     #print ASCII file header

$name = $infile.$filenumber;

while (-e $name) {#while there a files left
  open(VECT, $name) || die “could not open input file\n”;
  ($vector, $counts) = <VECT>;#read in vector and wait clocks
  chomp $counts;
  if (length($vector) != 528) {die “Vector is not 527 bits long:
“.(length($vector)-1).”!\n”;}

  print “in “.$name.” we will wait “.$counts.” clock cycles for the an-
swer.\n”;

  for ($i=0; $i<527; $i++) {#clock the scan vector into the scan chain (527
bits)
    print OUT ($clk=(not $clk)?1:0).” 1 0 0 0 0 0 0 1 “.substr($vector, $i,
1).” 1 1 “.$clk.” 1\n”;
    print OUT ($clk=(not $clk)?1:0).” 1 0 0 0 0 0 0 1 “.substr($vector, $i,
1).” 1 1 “.$clk.” 1\n”;
  }

  for ($i=0; $i<$counts; $i++) {#generate $counts clock cycles
    print OUT ($clk=(not $clk)?1:0).” 1 0 0 0 0 0 0 1 0 0 1 “.$clk.” 0\n”;
#operate clk once w/ scanen=0 in order to process scan
```

```
    print OUT ($clk=(not $clk)?1:0).” 1 0 0 0 0 0 0 1 0 0 1 “.$clk.” 0\n”;
  }


  close (VECT);
  $name = $infile.++$filenumber;
}

for ($i=0; $i<527; $i++) {#read out the answer
  print OUT ($clk=(not $clk)?1:0).” 1 0 0 0 0 0 0 1 0 1 1 “.$clk.” 1\n”;
  print OUT ($clk=(not $clk)?1:0).” 1 0 0 0 0 0 0 1 0 1 1 “.$clk.” 1\n”;
}

print OUT ($clk=(not $clk)?1:0).” 0 0 0 0 0 0 0 0 0 0 0 “.$clk.” 0\n”;#step
to the next chip
print OUT ($clk=(not $clk)?1:0).” 0 0 0 0 0 0 0 0 0 0 0 “.$clk.” 0\n”;

close (HEAD);
close (OUT);
```

APPENDIX 4

READOUT SCRIPTS

```
countbits.pl
===========================================================================
#!/bin/perl -w

use strict;

my ($i, $infile, @vect, $flag);
my ($n, $f);
my ($value);
my ($fn)=0;

$infile = $ARGV[0] || die “Please invoke with xform <infile> <outfile>!\n”;


open (VECT, “$infile”) || die “could not open input file\n”;



for ($i=0; $i<8; $i++) {#get rid of the 8 line header
  $f = <VECT>;
}


$n=0;

while ($vect[$n++] = <VECT>) {   #split file into parts of 527 lines, for
each part do display_result()
  if ($n == 527) {
    display_result();
    $n=0;
  }
} #VECT
if ($n != 1) {print $n.” The vector doesn’t consist of n*527 bits!!\n”;}


sub display_result {
  $value = 0;
  foreach (@vect) {#extract the values
    my ($bit, $state) = split(“ “, $_);
    $value += $bit;
  }
  print “The sum of all the bits in the vector is $value\n\n”;
}
```

```
close (VECT);


readout.pl
==========================================================================
#!/bin/perl -w

use strict;

my ($i, $outfile, $infile, @vect, $f, $flag);
my ($n);
my (@values, @states, $values, $states, $cap, $capval, $cal, $calval, $res,
$resval);
my ($fn)=0;

$infile = $ARGV[0] || die “Please invoke with xform <infile> <outfile>!\n”;
$outfile = $infile.”gnuplot”;


open (VECT, “$infile”) || die “could not open input file\n”;
open (OUT, “>$outfile”) || die “could not create output file\n”;
open (GNU, “>gnuplot.bat”) || die “could not create gnuplot file\n”;


print OUT “#count\tcap\tcal\tres\n”;#write into gnuplot data file


for ($i=0; $i<8; $i++) {#get rid of the 8 line header
  $f = <VECT>;
}


$n=0;

while ($vect[$n++] = <VECT>) {   #split file into parts of 527 lines, for
each part do display_result()
  if ($n == 527) {
    display_result();
    $n=0;
  }
} #VECT
if ($n != 1) {print $n.” The vector doesn’t consist of n*527 bits!!\n”;}


sub display_result {
  $values = ““;
  foreach (@vect) {#extract the values
    my ($bit, $state) = split(“ “, $_);
    $values = $values.$bit;
  }
  $cap = substr($values, 183, 20);#get bits 183-202 for the output register
  $capval = unpack(‘V’,pack(“b*”,”$cap”.(“0”x12)));
  $cal = substr($values, 206, 13);
  $calval = unpack(‘V’,pack(“b*”,”$cal”.(“0”x19)));
  $res = substr($values, 219, 19);
  $resval = unpack(‘V’,pack(“b*”,”$res”.(“0”x13)));

  print “Capacitive sensor says: $capval\nCalorimetric sensor says: $cal-
val\nResonant sensor says: $resval\n\n”;

  print OUT “$fn\t$capval\t$calval\t$resval\n”;#write into gnuplot data file
  $fn++;
}

print GNU “set term post color \”Arial\” 14\nset autoscale x\nset autoscale
y\nset autoscale y2\nset xlabel \”measurement #\”\nset ylabel \”cap+res
[int]\”\nset y2tics\nset y2label \”cal [int]\”\nplot [1:] \”$outfile\” using
2 title \”cap\” with linespoints, \”$outfile\” using 3 axes x1y2 title \”cal\”
with linespoints, \”$outfile\” using 4 title \”res\” with linespoints”;
#start gnupplot to generate gfx
close (GNU);
close (VECT);
close (OUT);

system(“/soft/gnu/bin/gnuplot <gnuplot.bat >$infile.ps”);
system(“ghostview $infile.ps”);
```

APPENDIX 5

RPI CONTROL PROGRAM

```
#!/bin/perl -w

use strict;
use IO::Socket;


my ($output, $command);
my ($sock, $i, $answer);


openConnection();
oadConfig(“/logic/pel/vector1”);

for ($i = 0; $i < 16; $i++) {    #step thru all 16 reticles
  putLine(“start\n”);
  waitfor(“->stopped”);
  print “fertig!!!!\n”;
}





sub openConnection {             #opens the network connection to 16702

my $host = shift || ‘iqe-mac-117.ethz.ch’;
my $port = shift || 6500;
$sock = new IO::Socket::INET(
                   PeerAddr => $host,
                   PeerPort => $port,
                   Proto    => ‘tcp’);
$sock or die “no socket :$!”;
}


sub closeConnection {
  close($sock);
}

sub waitfor {                    #waits until 16702 responds the argument
  my ($s)=$_[0];

        do {
```

```
                        putLine(“status\n”);
                       do { $answer=<$sock>; sleep 1} while ($answer eq ““);
        } until ($answer =~ m/$s/i);
}

sub putLine {                   #sends a command to 16702
  my ($command);
  ($command) = @_;
  print $sock ($command);
}


sub loadConfig {                #loads a given configuration
  my ($command);
  ($command) = @_;
  $command = ‘config -l ‘.$command.”\n”;
  putLine($command);
}
```

APPENDIX 6

SCAN CHAIN

```
1:   Core/I2CAsync/RepStart2_reg
2:   Core/I2CAsync/RepStart_reg
3:   Core/I2CAsync/RunInt_reg
4:   Core/SyncI2C/CalTC/U1/Current_State_reg[0] (*)
5:   Core/SyncI2C/CalTC/U1/Current_State_reg[1] (*)
6:   Core/SyncI2C/CalTC/U1/Current_State_reg[2] (*)
7:   Core/SyncI2C/CalTC/U1/Current_State_reg[3] (*)
8:   Core/SyncI2C/CalTC/U1/Finish_reg (*)
9:   Core/SyncI2C/CalTC/U1/Power_reg (*)
10:   Core/SyncI2C/CalTC/U1/RCnt_p/ZeroInt_reg (*)
11:   Core/SyncI2C/CalTC/U1/RegSel_reg
12:   Core/SyncI2C/CalTC/U1/XLoadRCnt_reg (*)
13:   Core/SyncI2C/CalTC/U2/Current_State_reg[0] (*)
14:   Core/SyncI2C/CalTC/U2/Current_State_reg[1] (*)
15:   Core/SyncI2C/CalTC/U3/Q1_reg (*)
16:   Core/SyncI2C/CalTC/U3/Q2_reg (*)
17:   Core/SyncI2C/CapTC/U1/Current_State_reg[0] (*)
18:   Core/SyncI2C/CapTC/U1/Current_State_reg[1] (*)
19:   Core/SyncI2C/CapTC/U1/Current_State_reg[2] (*)
20:   Core/SyncI2C/CapTC/U1/Current_State_reg[3] (*)
21:   Core/SyncI2C/CapTC/U1/Finish_reg (*)
22:   Core/SyncI2C/CapTC/U1/Power_reg (*)
23:   Core/SyncI2C/CapTC/U1/RCnt_p/ZeroInt_reg (*)
24:   Core/SyncI2C/CapTC/U1/RegSel_reg
25:   Core/SyncI2C/CapTC/U1/XLoadRCnt_reg (*)
26:   Core/SyncI2C/CapTC/U2/Current_State_reg[0] (*)
27:   Core/SyncI2C/CapTC/U2/Current_State_reg[1] (*)
28:   Core/SyncI2C/CapTC/U3/Q1_reg
29:   Core/SyncI2C/CapTC/U3/Q2_reg
30:   Core/SyncI2C/FIFOStack/InFIFO/Dack_Int_reg[0] (*)
31:   Core/SyncI2C/FIFOStack/InFIFO/Dack_Int_reg[1] (*)
32:   Core/SyncI2C/FIFOStack/InFIFO/Dack_Int_reg[2]
33:   Core/SyncI2C/FIFOStack/OutFIFO/Current_State_reg[0] (*)
34:   Core/SyncI2C/FIFOStack/OutFIFO/Current_State_reg[1]
35:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[0][0] (*)
36:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[0][1] (*)
37:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[0][2] (*)
38:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[0][3] (*)
39:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[0][4] (*)
40:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[0][5] (*)
41:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[0][6] (*)
42:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[0][7] (*)
43:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[1][0] (*)
```

```
44:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[1][1] (*)
45:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[1][2] (*)
46:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[1][3] (*)
47:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[1][4] (*)
48:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[1][5] (*)
49:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[1][6] (*)
50:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[1][7] (*)
51:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[2][0]
52:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[2][1]
53:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[2][2]
54:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[2][3]
55:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[2][4]
56:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[2][5]
57:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[2][6]
58:   Core/SyncI2C/FIFOStack/StackFIFO/Data_reg[2][7]
59:   Core/SyncI2C/FIFOStack/StackFIFO/Diff_Ptr_reg[0] (*)
60:   Core/SyncI2C/FIFOStack/StackFIFO/Diff_Ptr_reg[1] (*)
61:   Core/SyncI2C/FIFOStack/StackFIFO/Rd_Ptr_reg[0]
62:   Core/SyncI2C/FIFOStack/StackFIFO/Rd_Ptr_reg[1]
63:   Core/SyncI2C/FIFOStack/StackFIFO/Wr_Ptr_reg[0]
64:   Core/SyncI2C/FIFOStack/StackFIFO/Wr_Ptr_reg[1]
65:   Core/SyncI2C/Master/CntBit_reg[0]
66:   Core/SyncI2C/Master/CntBit_reg[1] (*)
67:   Core/SyncI2C/Master/CntBit_reg[2] (*)
68:   Core/SyncI2C/Master/CntBit_reg[3] (*)
69:   Core/SyncI2C/Master/CntByte_reg[0]
70:   Core/SyncI2C/Master/CntByte_reg[1] (*)
71:   Core/SyncI2C/Master/CntByte_reg[2] (*)
72:   Core/SyncI2C/Master/Current_State_reg[0] (*)
73:   Core/SyncI2C/Master/Current_State_reg[1]
74:   Core/SyncI2C/Master/Current_State_reg[2]
75:   Core/SyncI2C/Master/Current_State_reg[3] (*)
76:   Core/SyncI2C/Master/SCL_Out_reg (*)
77:   Core/SyncI2C/Master/SDA_Out_reg (*)
78:   Core/SyncI2C/RCnt_p/count_reg[0] (*)
79:   Core/SyncI2C/RCnt_p/y_int_reg
80:   Core/SyncI2C/ResTC/U1/Current_State_reg[0] (*)
81:   Core/SyncI2C/ResTC/U1/Current_State_reg[1] (*)
82:   Core/SyncI2C/ResTC/U1/Current_State_reg[2] (*)
83:   Core/SyncI2C/ResTC/U1/Current_State_reg[3] (*)
84:   Core/SyncI2C/ResTC/U1/Finish_reg (*)
85:   Core/SyncI2C/ResTC/U1/Power_reg (*)
86:   Core/SyncI2C/ResTC/U1/RCnt_p/ZeroInt_reg (*)
87:   Core/SyncI2C/ResTC/U1/RegSel_reg
88:   Core/SyncI2C/ResTC/U1/XLoadRCnt_reg (*)
89:   Core/SyncI2C/ResTC/U2/Current_State_reg[0] (*)
90:   Core/SyncI2C/ResTC/U2/Current_State_reg[1] (*)
91:   Core/SyncI2C/ResTC/U3/Q1_reg
92:   Core/SyncI2C/ResTC/U3/Q2_reg
93:   Core/SyncI2C/Master/LastSCL_reg (*)
94:   Core/SyncI2C/RegBank/RegBankCal/CmdReg_Int_reg[0] (*)
95:   Core/SyncI2C/RegBank/RegBankCal/CmdReg_Int_reg[1] (*)
96:   Core/SyncI2C/RegBank/RegBankCal/CmdReg_Int_reg[2] (*)
```

```
97:   Core/SyncI2C/RegBank/RegBankCal/CmdReg_Int_reg[3] (*)
98:   Core/SyncI2C/RegBank/RegBankCal/CmdReg_Int_reg[4] (*)
99:   Core/SyncI2C/RegBank/RegBankCal/CmdReg_Int_reg[5] (*)
100:   Core/SyncI2C/RegBank/RegBankCal/ComReg1_reg[0] (*)  register calsens
101:   Core/SyncI2C/RegBank/RegBankCal/ComReg1_reg[1] (*)  “
102:   Core/SyncI2C/RegBank/RegBankCal/ComReg1_reg[2] (*)  “
103:   Core/SyncI2C/RegBank/RegBankCal/ComReg1_reg[3] (*)  “
104:   Core/SyncI2C/RegBank/RegBankCal/ComReg2_reg[0] (*)  “
105:   Core/SyncI2C/RegBank/RegBankCal/ComReg2_reg[1] (*)  “
106:   Core/SyncI2C/RegBank/RegBankCal/ComReg2_reg[2] (*)  “
107:   Core/SyncI2C/RegBank/RegBankCal/ComReg2_reg[3] (*)  “
108:   Core/SyncI2C/RegBank/RegBankCal/ComReg3_reg[0] (*)  “
109:   Core/SyncI2C/RegBank/RegBankCal/ComReg3_reg[1] (*)  “
110:   Core/SyncI2C/RegBank/RegBankCal/ComReg3_reg[2] (*)  “
111:   Core/SyncI2C/RegBank/RegBankCal/ComReg3_reg[3] (*)  “
112:   Core/SyncI2C/RegBank/RegBankCal/ComReg3_reg[4] (*)  “
113:   Core/SyncI2C/RegBank/RegBankCal/ComReg3_reg[5] (*)  end
114:   Core/SyncI2C/RegBank/RegBankCap/CmdReg_Int_reg[0] (*)
115:   Core/SyncI2C/RegBank/RegBankCap/CmdReg_Int_reg[1] (*)
116:   Core/SyncI2C/RegBank/RegBankCap/CmdReg_Int_reg[2] (*)
117:   Core/SyncI2C/RegBank/RegBankCap/CmdReg_Int_reg[3] (*)
118:   Core/SyncI2C/RegBank/RegBankCap/CmdReg_Int_reg[4] (*)
119:   Core/SyncI2C/RegBank/RegBankCap/CmdReg_Int_reg[5] (*)
120:   Core/SyncI2C/RegBank/RegBankCap/ComReg1_reg[0] (*)  register capsens
121:   Core/SyncI2C/RegBank/RegBankCap/ComReg1_reg[1] (*)  “
122:   Core/SyncI2C/RegBank/RegBankCap/ComReg1_reg[2] (*)  “
123:   Core/SyncI2C/RegBank/RegBankCap/ComReg1_reg[3] (*)  “
124:   Core/SyncI2C/RegBank/RegBankCap/ComReg2_reg[0] (*)  “
125:   Core/SyncI2C/RegBank/RegBankCap/ComReg2_reg[1] (*)  “
126:   Core/SyncI2C/RegBank/RegBankCap/ComReg2_reg[2] (*)  “
127:   Core/SyncI2C/RegBank/RegBankCap/ComReg2_reg[3] (*)  “
128:   Core/SyncI2C/RegBank/RegBankCap/ComReg3_reg[0] (*)  “
129:   Core/SyncI2C/RegBank/RegBankCap/ComReg3_reg[1] (*)  “
130:   Core/SyncI2C/RegBank/RegBankCap/ComReg3_reg[2] (*)  “
131:   Core/SyncI2C/RegBank/RegBankCap/ComReg4_reg[0] (*)  “
132:   Core/SyncI2C/RegBank/RegBankCap/ComReg4_reg[1] (*)  “
133:   Core/SyncI2C/RegBank/RegBankCap/ComReg4_reg[2] (*)  “
134:   Core/SyncI2C/RegBank/RegBankCap/ComReg4_reg[3] (*)  “
135:   Core/SyncI2C/RegBank/RegBankCap/ComReg4_reg[4] (*)  “
136:   Core/SyncI2C/RegBank/RegBankCap/ComReg4_reg[5] (*)  “
137:   Core/SyncI2C/RegBank/RegBankCap/ComReg5_reg[0] (*)  “
138:   Core/SyncI2C/RegBank/RegBankCap/ComReg5_reg[1] (*)  “
139:   Core/SyncI2C/RegBank/RegBankCap/ComReg5_reg[2] (*)  “
140:   Core/SyncI2C/RegBank/RegBankCap/ComReg5_reg[3] (*)  “
141:   Core/SyncI2C/RegBank/RegBankCap/ComReg5_reg[4] (*)  emd
142:   Core/SyncI2C/RegBank/RegBankRes/CmdReg_Int_reg[0] (*)
143:   Core/SyncI2C/RegBank/RegBankRes/CmdReg_Int_reg[1] (*)
144:   Core/SyncI2C/RegBank/RegBankRes/CmdReg_Int_reg[2] (*)
145:   Core/SyncI2C/RegBank/RegBankRes/CmdReg_Int_reg[3] (*)
146:   Core/SyncI2C/RegBank/RegBankRes/CmdReg_Int_reg[4] (*)
147:   Core/SyncI2C/RegBank/RegBankRes/CmdReg_Int_reg[5] (*)
148:   Core/SyncI2C/RegBank/RegBankRes/ComReg1_reg[0] (*)  register ressens
149:   Core/SyncI2C/RegBank/RegBankRes/ComReg1_reg[1] (*)  “
```

```
150:   Core/SyncI2C/RegBank/RegBankRes/ComReg1_reg[2] (*)  “
151:   Core/SyncI2C/RegBank/RegBankRes/ComReg1_reg[3] (*)  “
152:   Core/SyncI2C/RegBank/RegBankRes/ComReg2_reg[0] (*)  “
153:   Core/SyncI2C/RegBank/RegBankRes/ComReg2_reg[1] (*)  “
154:   Core/SyncI2C/RegBank/RegBankRes/ComReg2_reg[2] (*)  “
155:   Core/SyncI2C/RegBank/RegBankRes/ComReg2_reg[3] (*)  “
156:   Core/SyncI2C/RegBank/RegBankRes/ComReg3_reg[0] (*)  “
157:   Core/SyncI2C/RegBank/RegBankRes/ComReg3_reg[1] (*)  “
158:   Core/SyncI2C/RegBank/RegBankRes/ComReg3_reg[2] (*)  “
159:   Core/SyncI2C/RegBank/RegBankRes/ComReg3_reg[3] (*)  “
160:   Core/SyncI2C/RegBank/RegBankRes/ComReg3_reg[4] (*)  “
161:   Core/SyncI2C/RegBank/RegBankRes/ComReg3_reg[5] (*)  ! we have 1 more
162:   Core/SyncI2C/Slave/U2/CountIntern_reg[0] (*)
163:   Core/SyncI2C/Slave/U2/CountIntern_reg[1]
164:   Core/SyncI2C/Slave/U2/CountIntern_reg[2]
165:   Core/SyncI2C/Slave/U2/CountIntern_reg[3] (*)
166:   Core/SyncI2C/Slave/U3/CountIntern_reg[0] (*)
167:   Core/SyncI2C/Slave/U3/CountIntern_reg[1]
168:   Core/SyncI2C/Slave/U3/CountIntern_reg[2] (*)
169:   Core/SyncI2C/Slave/U5/Selec_reg (*)
170:   Core/SyncI2C/Slave/U5/SenAdr_reg[0] (*)
171:   Core/SyncI2C/Slave/U5/SenAdr_reg[1] (*)
172:   Core/SyncI2C/Master/LastDat_reg (*)
173:   Core/SyncI2C/Slave/U4/DataReg_reg[0] (*)
174:   Core/SyncI2C/Slave/U4/DataReg_reg[1] (*)
175:   Core/SyncI2C/Slave/U4/DataReg_reg[2] (*)
176:   Core/SyncI2C/Slave/U4/DataReg_reg[3] (*)
177:   Core/SyncI2C/Slave/U4/DataReg_reg[4] (*)
178:   Core/SyncI2C/Slave/U4/DataReg_reg[5] (*)
179:   Core/SyncI2C/Slave/U4/DataReg_reg[6] (*)
180:   Core/SyncI2C/Slave/U4/DataReg_reg[7] (*)
181:   Core/SyncI2C/Slave/U6/Finish_reg
182:   Core/SyncI2C/RegBank/RegBankRes/ComReg3_reg[6]  register ressens[6]
183:   DecimationFilter/CapVal_reg[0]  output capsens
184:   DecimationFilter/CapVal_reg[1]  “
185:   DecimationFilter/CapVal_reg[2]  “
186:   DecimationFilter/CapVal_reg[3]  “
187:   DecimationFilter/CapVal_reg[4]  “
188:   DecimationFilter/CapVal_reg[5]  “
189:   DecimationFilter/CapVal_reg[6]  “
190:   DecimationFilter/CapVal_reg[7]  “
191:   DecimationFilter/CapVal_reg[8]  “
192:   DecimationFilter/CapVal_reg[9]  “
193:   DecimationFilter/CapVal_reg[10] “
194:   DecimationFilter/CapVal_reg[11] “
195:   DecimationFilter/CapVal_reg[12] “
196:   DecimationFilter/CapVal_reg[13] “
197:   DecimationFilter/CapVal_reg[14] “
198:   DecimationFilter/CapVal_reg[15] “
199:   DecimationFilter/CapVal_reg[16] “
200:   DecimationFilter/CapVal_reg[17] “
201:   DecimationFilter/CapVal_reg[18] “
202:   DecimationFilter/CapVal_reg[19] end
```

```
203:   DecimationFilter/ConvClk/ZeroInt_reg (*)
204:   DecimationFilter/GetCnt_reg
205:   DecimationFilter/XLoad_reg (*)
206:   DecimationFilter/CalFIR/DataOut_reg[0]  output calsens
207:   DecimationFilter/CalFIR/DataOut_reg[1]  “
208:   DecimationFilter/CalFIR/DataOut_reg[2]  “
209:   DecimationFilter/CalFIR/DataOut_reg[3]  “
210:   DecimationFilter/CalFIR/DataOut_reg[4]  “
211:   DecimationFilter/CalFIR/DataOut_reg[5]  “
212:   DecimationFilter/CalFIR/DataOut_reg[6]  “
213:   DecimationFilter/CalFIR/DataOut_reg[7]  “
214:   DecimationFilter/CalFIR/DataOut_reg[8]  “
215:   DecimationFilter/CalFIR/DataOut_reg[9]  “
216:   DecimationFilter/CalFIR/DataOut_reg[10] “
217:   DecimationFilter/CalFIR/DataOut_reg[11] “
218:   DecimationFilter/CalFIR/DataOut_reg[12] end
219:   DecimationFilter/ResVal_reg[0]  output ressens
220:   DecimationFilter/ResVal_reg[1]  “
221:   DecimationFilter/ResVal_reg[2]  “
222:   DecimationFilter/ResVal_reg[3]  “
223:   DecimationFilter/ResVal_reg[4]  “
224:   DecimationFilter/ResVal_reg[5]  “
225:   DecimationFilter/ResVal_reg[6]  “
226:   DecimationFilter/ResVal_reg[7]  “
227:   DecimationFilter/ResVal_reg[8]  “
228:   DecimationFilter/ResVal_reg[9]  “
229:   DecimationFilter/ResVal_reg[10] “
230:   DecimationFilter/ResVal_reg[11] “
231:   DecimationFilter/ResVal_reg[12] “
232:   DecimationFilter/ResVal_reg[13] “
233:   DecimationFilter/ResVal_reg[14] “
234:   DecimationFilter/ResVal_reg[15] “
235:   DecimationFilter/ResVal_reg[16] “
236:   DecimationFilter/ResVal_reg[17] “
237:   DecimationFilter/ResVal_reg[18] end
238:   DecimationFilter/CalSinc/INT1/DataReg_reg[0]
239:   DecimationFilter/CalSinc/INT1/DataReg_reg[1]
240:   DecimationFilter/CalSinc/INT1/DataReg_reg[2]
241:   DecimationFilter/CalSinc/INT1/DataReg_reg[3]
242:   DecimationFilter/CalSinc/INT1/DataReg_reg[4]
243:   DecimationFilter/CalSinc/INT1/DataReg_reg[5]
244:   DecimationFilter/CalSinc/INT1/DataReg_reg[6]
245:   DecimationFilter/CalSinc/INT1/DataReg_reg[7]
246:   DecimationFilter/CalSinc/INT1/DataReg_reg[8]
247:   DecimationFilter/CalSinc/INT1/DataReg_reg[9]
248:   DecimationFilter/CalSinc/INT1/DataReg_reg[10]
249:   DecimationFilter/CalSinc/INT1/DataReg_reg[11]
250:   DecimationFilter/CalSinc/INT1/DataReg_reg[12]
251:   DecimationFilter/CalSinc/INT1/DataReg_reg[13]
252:   DecimationFilter/CalSinc/INT1/DataReg_reg[14]
253:   DecimationFilter/CalSinc/INT1/DataReg_reg[15]
254:   DecimationFilter/CalSinc/INT2/DataReg_reg[0]
255:   DecimationFilter/CalSinc/INT2/DataReg_reg[1]
```

```
256:   DecimationFilter/CalSinc/INT2/DataReg_reg[2]
257:   DecimationFilter/CalSinc/INT2/DataReg_reg[3]
258:   DecimationFilter/CalSinc/INT2/DataReg_reg[4]
259:   DecimationFilter/CalSinc/INT2/DataReg_reg[5]
260:   DecimationFilter/CalSinc/INT2/DataReg_reg[6]
261:   DecimationFilter/CalSinc/INT2/DataReg_reg[7]
262:   DecimationFilter/CalSinc/INT2/DataReg_reg[8]
263:   DecimationFilter/CalSinc/INT2/DataReg_reg[9]
264:   DecimationFilter/CalSinc/INT2/DataReg_reg[10]
265:   DecimationFilter/CalSinc/INT2/DataReg_reg[11]
266:   DecimationFilter/CalSinc/INT2/DataReg_reg[12]
267:   DecimationFilter/CalSinc/INT2/DataReg_reg[13]
268:   DecimationFilter/CalSinc/INT2/DataReg_reg[14]
269:   DecimationFilter/CalSinc/INT3/DataReg_reg[0]
270:   DecimationFilter/CalSinc/INT3/DataReg_reg[1]
271:   DecimationFilter/CalSinc/INT3/DataReg_reg[2]
272:   DecimationFilter/CalSinc/INT3/DataReg_reg[3]
273:   DecimationFilter/CalSinc/INT3/DataReg_reg[4]
274:   DecimationFilter/CalSinc/INT3/DataReg_reg[5]
275:   DecimationFilter/CalSinc/INT3/DataReg_reg[6]
276:   DecimationFilter/CalSinc/INT3/DataReg_reg[7]
277:   DecimationFilter/CalSinc/INT3/DataReg_reg[8]
278:   DecimationFilter/CalSinc/INT3/DataReg_reg[9]
279:   DecimationFilter/CalSinc/INT3/DataReg_reg[10]
280:   DecimationFilter/CalSinc/INT3/DataReg_reg[11]
281:   DecimationFilter/CalSinc/INT3/DataReg_reg[12]
282:   DecimationFilter/CalSinc/INT3/DataReg_reg[13]
283:   DecimationFilter/CalFIR/DEC4/COUNT_reg[0] (*)
284:   DecimationFilter/CalFIR/DEC4/COUNT_reg[1]
285:   DecimationFilter/CalFIR/DEC4/DecClk_reg
286:   DecimationFilter/CalSinc/DataForDIF1_reg[0]
287:   DecimationFilter/CalSinc/DataForDIF1_reg[1]
288:   DecimationFilter/CalSinc/DataForDIF1_reg[2]
289:   DecimationFilter/CalSinc/DataForDIF1_reg[3]
290:   DecimationFilter/CalSinc/DataForDIF1_reg[4]
291:   DecimationFilter/CalSinc/DataForDIF1_reg[5]
292:   DecimationFilter/CalSinc/DataForDIF1_reg[6]
293:   DecimationFilter/CalSinc/DataForDIF1_reg[7]
294:   DecimationFilter/CalSinc/DataForDIF1_reg[8]
295:   DecimationFilter/CalSinc/DataForDIF1_reg[9]
296:   DecimationFilter/CalSinc/DataForDIF1_reg[10]
297:   DecimationFilter/CalSinc/DataForDIF1_reg[11]
298:   DecimationFilter/CalSinc/Diff1/DataReg_reg[0]
299:   DecimationFilter/CalSinc/Diff1/DataReg_reg[1]
300:   DecimationFilter/CalSinc/Diff1/DataReg_reg[2]
301:   DecimationFilter/CalSinc/Diff1/DataReg_reg[3]
302:   DecimationFilter/CalSinc/Diff1/DataReg_reg[4]
303:   DecimationFilter/CalSinc/Diff1/DataReg_reg[5]
304:   DecimationFilter/CalSinc/Diff1/DataReg_reg[6]
305:   DecimationFilter/CalSinc/Diff1/DataReg_reg[7]
306:   DecimationFilter/CalSinc/Diff1/DataReg_reg[8]
307:   DecimationFilter/CalSinc/Diff1/DataReg_reg[9]
308:   DecimationFilter/CalSinc/Diff1/DataReg_reg[10]
```

```
309:   DecimationFilter/CalSinc/Diff1/DataReg_reg[11]
310:   DecimationFilter/CalSinc/Diff2/DataReg_reg[0]
311:   DecimationFilter/CalSinc/Diff2/DataReg_reg[1]
312:   DecimationFilter/CalSinc/Diff2/DataReg_reg[2]
313:   DecimationFilter/CalSinc/Diff2/DataReg_reg[3]
314:   DecimationFilter/CalSinc/Diff2/DataReg_reg[4]
315:   DecimationFilter/CalSinc/Diff2/DataReg_reg[5]
316:   DecimationFilter/CalSinc/Diff2/DataReg_reg[6]
317:   DecimationFilter/CalSinc/Diff2/DataReg_reg[7]
318:   DecimationFilter/CalSinc/Diff2/DataReg_reg[8]
319:   DecimationFilter/CalSinc/Diff2/DataReg_reg[9]
320:   DecimationFilter/CalSinc/Diff2/DataReg_reg[10]
321:   DecimationFilter/CalSinc/Diff2/DataReg_reg[11]
322:   DecimationFilter/CalSinc/Diff3/DataReg_reg[0]
323:   DecimationFilter/CalSinc/Diff3/DataReg_reg[1]
324:   DecimationFilter/CalSinc/Diff3/DataReg_reg[2]
325:   DecimationFilter/CalSinc/Diff3/DataReg_reg[3]
326:   DecimationFilter/CalSinc/Diff3/DataReg_reg[4]
327:   DecimationFilter/CalSinc/Diff3/DataReg_reg[5]
328:   DecimationFilter/CalSinc/Diff3/DataReg_reg[6]
329:   DecimationFilter/CalSinc/Diff3/DataReg_reg[7]
330:   DecimationFilter/CalSinc/Diff3/DataReg_reg[8]
331:   DecimationFilter/CalSinc/Diff3/DataReg_reg[9]
332:   DecimationFilter/CalSinc/Diff3/DataReg_reg[10]
333:   DecimationFilter/CalSinc/Diff3/DataReg_reg[11]
334:   DecimationFilter/OutputVal_reg[0]
335:   DecimationFilter/OutputVal_reg[1]
336:   DecimationFilter/OutputVal_reg[2]
337:   DecimationFilter/OutputVal_reg[3]
338:   DecimationFilter/OutputVal_reg[4]
339:   DecimationFilter/OutputVal_reg[5]
340:   DecimationFilter/OutputVal_reg[6]
341:   DecimationFilter/OutputVal_reg[7]
342:   DecimationFilter/OutputVal_reg[8]
343:   DecimationFilter/OutputVal_reg[9]
344:   DecimationFilter/OutputVal_reg[10]
345:   DecimationFilter/OutputVal_reg[11]
346:   DecimationFilter/OutputVal_reg[12]
347:   DecimationFilter/OutputVal_reg[13]
348:   DecimationFilter/OutputVal_reg[14]
349:   DecimationFilter/OutputVal_reg[15]
350:   DecimationFilter/OutputVal_reg[16]
351:   DecimationFilter/OutputVal_reg[17]
352:   DecimationFilter/OutputVal_reg[18]
353:   DecimationFilter/OutputVal_reg[19]
354:   DecimationFilter/CalFIR/D_reg[1][0]
355:   DecimationFilter/CalFIR/D_reg[1][1]
356:   DecimationFilter/CalFIR/D_reg[1][2]
357:   DecimationFilter/CalFIR/D_reg[1][3]
358:   DecimationFilter/CalFIR/D_reg[1][4]
359:   DecimationFilter/CalFIR/D_reg[1][5]
360:   DecimationFilter/CalFIR/D_reg[1][6]
361:   DecimationFilter/CalFIR/D_reg[1][7]
```

```
362:   DecimationFilter/CalFIR/D_reg[1][8]
363:   DecimationFilter/CalFIR/D_reg[1][9]
364:   DecimationFilter/CalFIR/D_reg[1][10]
365:   DecimationFilter/CalFIR/D_reg[1][11]
366:   DecimationFilter/CalFIR/D_reg[1][12]
367:   DecimationFilter/CalFIR/D_reg[2][0]
368:   DecimationFilter/CalFIR/D_reg[2][1]
369:   DecimationFilter/CalFIR/D_reg[2][2]
370:   DecimationFilter/CalFIR/D_reg[2][3]
371:   DecimationFilter/CalFIR/D_reg[2][4]
372:   DecimationFilter/CalFIR/D_reg[2][5]
373:   DecimationFilter/CalFIR/D_reg[2][6]
374:   DecimationFilter/CalFIR/D_reg[2][7]
375:   DecimationFilter/CalFIR/D_reg[2][8]
376:   DecimationFilter/CalFIR/D_reg[2][9]
377:   DecimationFilter/CalFIR/D_reg[2][10]
378:   DecimationFilter/CalFIR/D_reg[2][11]
379:   DecimationFilter/CalFIR/D_reg[2][12]
380:   DecimationFilter/CalFIR/D_reg[3][0]
381:   DecimationFilter/CalFIR/D_reg[3][1]
382:   DecimationFilter/CalFIR/D_reg[3][2]
383:   DecimationFilter/CalFIR/D_reg[3][3]
384:   DecimationFilter/CalFIR/D_reg[3][4]
385:   DecimationFilter/CalFIR/D_reg[3][5]
386:   DecimationFilter/CalFIR/D_reg[3][6]
387:   DecimationFilter/CalFIR/D_reg[3][7]
388:   DecimationFilter/CalFIR/D_reg[3][8]
389:   DecimationFilter/CalFIR/D_reg[3][9]
390:   DecimationFilter/CalFIR/D_reg[3][10]
391:   DecimationFilter/CalFIR/D_reg[3][11]
392:   DecimationFilter/CalFIR/D_reg[3][12]
393:   DecimationFilter/CalFIR/D_reg[4][0]
394:   DecimationFilter/CalFIR/D_reg[4][1]
395:   DecimationFilter/CalFIR/D_reg[4][2]
396:   DecimationFilter/CalFIR/D_reg[4][3]
397:   DecimationFilter/CalFIR/D_reg[4][4]
398:   DecimationFilter/CalFIR/D_reg[4][5]
399:   DecimationFilter/CalFIR/D_reg[4][6]
400:   DecimationFilter/CalFIR/D_reg[4][7]
401:   DecimationFilter/CalFIR/D_reg[4][8]
402:   DecimationFilter/CalFIR/D_reg[4][9]
403:   DecimationFilter/CalFIR/D_reg[4][10]
404:   DecimationFilter/CalFIR/D_reg[4][11]
405:   DecimationFilter/CalFIR/D_reg[4][12]
406:   DecimationFilter/CalFIR/D_reg[5][0]
407:   DecimationFilter/CalFIR/D_reg[5][1]
408:   DecimationFilter/CalFIR/D_reg[5][2]
409:   DecimationFilter/CalFIR/D_reg[5][3]
410:   DecimationFilter/CalFIR/D_reg[5][4]
411:   DecimationFilter/CalFIR/D_reg[5][5]
412:   DecimationFilter/CalFIR/D_reg[5][6]
413:   DecimationFilter/CalFIR/D_reg[5][7]
414:   DecimationFilter/CalFIR/D_reg[5][8]
```

```
415:   DecimationFilter/CalFIR/D_reg[5][9]
416:   DecimationFilter/CalFIR/D_reg[5][10]
417:   DecimationFilter/CalFIR/D_reg[5][11]
418:   DecimationFilter/CalFIR/D_reg[5][12]
419:   DecimationFilter/CalFIR/D_reg[6][0]
420:   DecimationFilter/CalFIR/D_reg[6][1]
421:   DecimationFilter/CalFIR/D_reg[6][2]
422:   DecimationFilter/CalFIR/D_reg[6][3]
423:   DecimationFilter/CalFIR/D_reg[6][4]
424:   DecimationFilter/CalFIR/D_reg[6][5]
425:   DecimationFilter/CalFIR/D_reg[6][6]
426:   DecimationFilter/CalFIR/D_reg[6][7]
427:   DecimationFilter/CalFIR/D_reg[6][8]
428:   DecimationFilter/CalFIR/D_reg[6][9]
429:   DecimationFilter/CalFIR/D_reg[6][10]
430:   DecimationFilter/CalFIR/D_reg[6][11]
431:   DecimationFilter/CalFIR/D_reg[6][12]
432:   DecimationFilter/CalFIR/D_reg[7][0]
433:   DecimationFilter/CalFIR/D_reg[7][1]
434:   DecimationFilter/CalFIR/D_reg[7][2]
435:   DecimationFilter/CalFIR/D_reg[7][3]
436:   DecimationFilter/CalFIR/D_reg[7][4]
437:   DecimationFilter/CalFIR/D_reg[7][5]
438:   DecimationFilter/CalFIR/D_reg[7][6]
439:   DecimationFilter/CalFIR/D_reg[7][7]
440:   DecimationFilter/CalFIR/D_reg[7][8]
441:   DecimationFilter/CalFIR/D_reg[7][9]
442:   DecimationFilter/CalFIR/D_reg[7][10]
443:   DecimationFilter/CalFIR/D_reg[7][11]
444:   DecimationFilter/CalFIR/D_reg[7][12]
445:   DecimationFilter/CalFIR/D_reg[8][0]
446:   DecimationFilter/CalFIR/D_reg[8][1]
447:   DecimationFilter/CalFIR/D_reg[8][2]
448:   DecimationFilter/CalFIR/D_reg[8][3]
449:   DecimationFilter/CalFIR/D_reg[8][4]
450:   DecimationFilter/CalFIR/D_reg[8][5]
451:   DecimationFilter/CalFIR/D_reg[8][6]
452:   DecimationFilter/CalFIR/D_reg[8][7]
453:   DecimationFilter/CalFIR/D_reg[8][8]
454:   DecimationFilter/CalFIR/D_reg[8][9]
455:   DecimationFilter/CalFIR/D_reg[8][10]
456:   DecimationFilter/CalFIR/D_reg[8][11]
457:   DecimationFilter/CalFIR/D_reg[8][12]
458:   DecimationFilter/CalFIR/D_reg[9][0]
459:   DecimationFilter/CalFIR/D_reg[9][1]
460:   DecimationFilter/CalFIR/D_reg[9][2]
461:   DecimationFilter/CalFIR/D_reg[9][3]
462:   DecimationFilter/CalFIR/D_reg[9][4]
463:   DecimationFilter/CalFIR/D_reg[9][5]
464:   DecimationFilter/CalFIR/D_reg[9][6]
465:   DecimationFilter/CalFIR/D_reg[9][7]
466:   DecimationFilter/CalFIR/D_reg[9][8]
467:   DecimationFilter/CalFIR/D_reg[9][9]
```

```
468:   DecimationFilter/CalFIR/D_reg[9][10]
469:   DecimationFilter/CalFIR/D_reg[9][11]
470:   DecimationFilter/CalFIR/D_reg[9][12]
471:   DecimationFilter/CalFIR/D_reg[10][0]
472:   DecimationFilter/CalFIR/D_reg[10][1]
473:   DecimationFilter/CalFIR/D_reg[10][2]
474:   DecimationFilter/CalFIR/D_reg[10][3]
475:   DecimationFilter/CalFIR/D_reg[10][4]
476:   DecimationFilter/CalFIR/D_reg[10][5]
477:   DecimationFilter/CalFIR/D_reg[10][6]
478:   DecimationFilter/CalFIR/D_reg[10][7]
479:   DecimationFilter/CalFIR/D_reg[10][8]
480:   DecimationFilter/CalFIR/D_reg[10][9]
481:   DecimationFilter/CalFIR/D_reg[10][10]
482:   DecimationFilter/CalFIR/D_reg[10][11]
483:   DecimationFilter/CalFIR/D_reg[10][12]
484:   DecimationFilter/CalFIR/D_reg[11][0]
485:   DecimationFilter/CalFIR/D_reg[11][1]
486:   DecimationFilter/CalFIR/D_reg[11][2]
487:   DecimationFilter/CalFIR/D_reg[11][3]
488:   DecimationFilter/CalFIR/D_reg[11][4]
489:   DecimationFilter/CalFIR/D_reg[11][5]
490:   DecimationFilter/CalFIR/D_reg[11][6]
491:   DecimationFilter/CalFIR/D_reg[11][7]
492:   DecimationFilter/CalFIR/D_reg[11][8]
493:   DecimationFilter/CalFIR/D_reg[11][9]
494:   DecimationFilter/CalFIR/D_reg[11][10]
495:   DecimationFilter/CalFIR/D_reg[11][11]
496:   DecimationFilter/CalFIR/D_reg[11][12]
497:   DecimationFilter/CalFIR/D_reg[12][0]
498:   DecimationFilter/CalFIR/D_reg[12][1]
499:   DecimationFilter/CalFIR/D_reg[12][2]
500:   DecimationFilter/CalFIR/D_reg[12][3]
501:   DecimationFilter/CalFIR/D_reg[12][4]
502:   DecimationFilter/CalFIR/D_reg[12][5]
503:   DecimationFilter/CalFIR/D_reg[12][6]
504:   DecimationFilter/CalFIR/D_reg[12][7]
505:   DecimationFilter/CalFIR/D_reg[12][8]
506:   DecimationFilter/CalFIR/D_reg[12][9]
507:   DecimationFilter/CalFIR/D_reg[12][10]
508:   DecimationFilter/CalFIR/D_reg[12][11]
509:   DecimationFilter/CalFIR/D_reg[12][12]
510:   DecimationFilter/CalFIR/D_reg[13][0]
511:   DecimationFilter/CalFIR/D_reg[13][1]
512:   DecimationFilter/CalFIR/D_reg[13][2]
513:   DecimationFilter/CalFIR/D_reg[13][3]
514:   DecimationFilter/CalFIR/D_reg[13][4]
515:   DecimationFilter/CalFIR/D_reg[13][5]
516:   DecimationFilter/CalFIR/D_reg[13][6]
517:   DecimationFilter/CalFIR/D_reg[13][7]
518:   DecimationFilter/CalFIR/D_reg[13][8]
519:   DecimationFilter/CalFIR/D_reg[13][9]
520:   DecimationFilter/CalFIR/D_reg[13][10]
```

```
521:   DecimationFilter/CalFIR/D_reg[13][11]
522:   DecimationFilter/CalFIR/D_reg[13][12]
523:   DecimationFilter/CalSinc/DecimationClk/Count_reg[0]
524:   DecimationFilter/CalSinc/DecimationClk/Count_reg[1]
525:   DecimationFilter/CalSinc/DecimationClk/Count_reg[2]
526:   DecimationFilter/CalSinc/DecimationClk/Count_reg[3]
527:   DecimationFilter/CalSinc/DecimationClk/DecClkTmp_reg
```