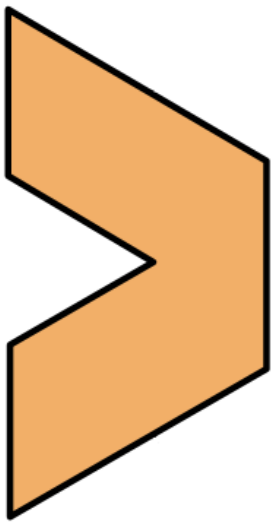
**UNIVERSITY OF OSLO**
**Department of Physics**



# Asynchronous Processing Units

## Dynamic trade-off between high-speed and low-power

Master thesis

Harald S. Furuseth

**January 2013**

# Asynchronous Processing Units

Harald S. Furuseth

January 2013

# Abstract

The purpose of this thesis is to explore the operation of clockless digital logic. Different methods of achieving clockless operation are examined. A chosen architecture is implemented on chip in a commercial 90nm process from TSMC.

Post-production testing is performed by means of a custom-built PCB, as well as a microcontroller with custom firmware. The performance and characteristics of the chip is then evaluated for varying supply voltages.

# Preface

This thesis has been both inspiring, challenging and greatly educational. During this master thesis, I have been fortunate enough to design a custom ASIC, as well as a custom PCB.

I would like to express my gratitude to my supervisor Philipp Häfliger, for his invaluable guidance during the course of this thesis work, and I would like to thank the members of his team of PhD candidates and postdocs, Juan Antonio Leñero-Bardallo, Ali Zaher and Thanh Trung Nguyen, for greatly appreciated input during our weekly meetings.

I would also like to thank fellow students Sindre Sørum, Lars Jørgen Aamodt, Kristoffer Amundsen and Anders Fritzell, for both on- and off-topic discussions and insight in the past few years.

Stein Stabelfelt Nielsen and the staff at the electronics lab in the physics department deserve a special thanks for their assistance with PCB design and component assembly.

On a personal note, I want to extend my thanks to my girlfriend Cecilie, for her continuous support and patience.

# Contents


**1 Introduction** **1**

**2 Why Asynchronous?** **3**
2.1 Asynchronous Methodologies . . . . . 5
2.1.1 Matched Delays . . . . . 5
2.1.2 Current-Sensing Completion Detection . . . . . 7
2.1.3 Differential Cascode Voltage Switch Logic . . . . . 7

**3 ALU Design** **13**
3.1 Instruction Decoder . . . . . 13
3.2 Opcodes . . . . . 17
3.2.1 ADD . . . . . 22
3.3 Transmission Gates and Standby Pull-up . . . . . 33
3.4 Completion Generation . . . . . 35
3.5 The Output Register . . . . . 37
3.6 Connecting the ALU to Padframe . . . . . 45

**4 Testing the ALU** **49**
4.1 PCB Design . . . . . 49
4.1.1 Schematic . . . . . 51
4.1.2 Layout . . . . . 52
4.2 Microcontroller Interaction . . . . . 53

**5 Results** **55**

**6 Discussion** **65**
6.1 Internal Request to the Output Register . . . . . 65
6.2 Single-bit Errors in the Output Data . . . . . 65
6.3 Schematic Errors in the Kogge-Stone Adder . . . . . 66
6.4 Measuring Current Consumption . . . . . 67
6.5 Irregularities in Layout Cell Size . . . . . 67
6.6 Lower limit of the Supply Voltage . . . . . 68

**7 Conclusion** **69**

**A Chip package with bonding diagram and padframe** **75**

**B PCB full schematics and layout** **81**

**C Olimex SAM7-H256 header board** **89**

**D Microcontroller C code and relevant header files** **93**

# List of Figures


2.1 Measured substrate noise for an array of synchronous (left) and asynchronous (right) pseudorandom number generators[17]. . . . . 5
2.2 A processing pipeline using delay elements to synchronize data. . . . . 6
2.3 A single-ended CVSL gate. . . . . 8
2.4 A DCVSL gate, using a differential signal path. . . . . 9
2.5 A DCVSL gate with NAND completion detection. . . . . 10
2.6 A CMOS NAND gate pull-down network. . . . . 11
2.7 A DCVSL NAND gate pull-down network. . . . . 12

3.1 Schematic of the instruction decoder. . . . . 14
3.2 A 30ns transient simulation of the instruction decoder, strobing the inputs at different rates. . . . . 15
3.3 Layout of the instruction decoder. Height = 15.78$\mu$m, width = 5.23$\mu$m. . . . . 16
3.4 Top: Schematic of a DCVSL NOT gate. Bottom: Layout of a DCVSL NOT gate. Height = 2.45$\mu$m, width = 2.99$\mu$m. . . . . 18
3.5 Top: Schematic of a DCVSL AND gate. Bottom: Layout of a DCVSL AND gate. Height = 2.54$\mu$m, width = 3.27$\mu$m. . . . . 19
3.6 Top: Schematic of a DCVSL OR gate. Bottom: Layout of a DCVSL OR gate. Height = 2.54$\mu$m, width = 3.27$\mu$m. . . . . 20
3.7 Top: Schematic of a DCVSL XOR gate. Bottom: Layout of a DCVSL XOR gate. Height = 8.11$\mu$m, width = 9.15$\mu$m. . . . . 21
3.8 Schematic of a DCVSL 8-bit Kogge-Stone adder with overflow output. . . . . 23
3.9 Layout of a DCVSL 8-bit Kogge-Stone adder with overflow output. Height = 72.81$\mu$m, width = 50.32$\mu$m. . . . . 24
3.10 Top: Schematic of the top cell in the Kogge-Stone adder. Bottom: Layout of the top cell in the Kogge-Stone adder. Height = 14.35$\mu$m, width = 10.71$\mu$m. . . . . 26
3.11 Top: Schematic of the mid cell in the Kogge-Stone adder. Bottom: Layout of the mid cell in the Kogge-Stone adder. Height = 6.56$\mu$m, width = 9.21$\mu$m. . . . . 28
3.12 Parallel prefix graph of a 16-bit Kogge-Stone adder [4]. . . . 29

3.13 Top: Schematic of the bottom cell in the Kogge-Stone adder. Bottom: Layout of the bottom cell in the Kogge-Stone adder. Height = 6.54$\mu$m, width = 13.47$\mu$m. . . . . 31
3.14 Top: Schematic of a transmission gate. Bottom: Layout of a transmission gate. Height = 2.47$\mu$m, width = 1.59$\mu$m. . . . . 34
3.15 Schematic of the completion generator. . . . . 36
3.16 Schematic of the output register. . . . . 38
3.17 Top: Schematic of an SR latch using NOR gates. Bottom: Layout of an SR latch using NOR gates. Height = 4.14$\mu$m, width = 2.15$\mu$m. . . . . 40
3.18 Layout of the output register. Height = 42.94$\mu$m, width = 4.39$\mu$m. . . . . 41
3.19 Cutout of the control logic in the schematic of the output register. . . . . 43
3.20 State diagram describing the output register control logic. . 44
3.21 Top level view of the ALU schematic. . . . . 46
3.22 Schematic view of output-to-pad digital buffers and analog voltage followers for testpoints. . . . . 47

4.1 Top: Photograph of the finished PCB. Bottom: Photograph of the finished PCB with microcontroller header board and lab equipment attached, during a testing run. . . . . 50
4.2 Top level sheet of the PCB schematic. . . . . 52
4.3 PCB layout. . . . . 53

5.1 Plot of pad- and core static current consumption vs. supply voltage. . . . . 59
5.2 Oscilloscope screenshot showing the REQ and ACK signals during an XOR operation at 1.2V. . . . . 61
5.3 Oscilloscope screenshot showing the propagation of Request during the very first XOR data set at 1.2V. . . . . 62
5.4 Oscilloscope screenshot showing the propagation of Request during the very last XOR data set at 1.2V. . . . . 63
5.5 Oscilloscope screenshot showing an error in bit 3 of the ALU output during an OR data set at 1.2V. . . . . 64

6.1 Schematic view of the Kogge-Stone adder with errors. . . . . 67

A.1 Full view of ALU layout, not including pad connections. . . 76
A.2 Full view of ALU layout inserted into the padframe. . . . . 77
A.3 Bonding diagram for the ALU chip. . . . . 78
A.4 Data sheet for the chip package showing physical dimensions. 79

B.1 Top level sheet of the PCB schematic. . . . . 82

B.2 PCB schematic, design sheet STEPDOWN_CONTR, showing potentiometers for the Request-, Acknowledge- and Select signals. . . . . 83
B.3 PCB schematic, design sheet STEPDOWN_A, showing potentiometers for ALU input vector A. . . . . 84
B.4 PCB schematic, design sheet STEPDOWN_B, showing potentiometers for ALU input vector B. . . . . 85
B.5 PCB schematic, design sheet STEPUP, showing the voltage comparators for the ALU outputs. . . . . 86
B.6 PCB layout. . . . . 87

C.1 Bottom view of the SAM7-H256 header board, showing the AT91SAM7S256 microcontroller, two pin rows for interconnect, and the USB port on the left. . . . . 90
C.2 Schematic of the header board. . . . . 91

# List of Tables


2.1 DCVSL output truth table. . . . . 9

3.1 List of supported ALU operations. . . . . 17
3.2 SR latch truth table. . . . . 39

5.1 ALU chip timing measurements for the NOT function. . . . 56
5.2 ALU chip timing measurements for the AND and NAND functions. . . . . 56
5.3 ALU chip timing measurements for the OR and NOR functions. . . . . 57
5.4 ALU chip timing measurements for the XOR and XNOR functions. . . . . 57
5.5 ALU chip timing measurements for the ADD function. . . . 58
5.6 ALU chip static current consumption. . . . . 58

# Nomenclature

$CV_{DD}$ Core $V_{DD}$. Supply voltage for the ALU core only, excluding everything else on the chip

$I_{DDmax}$ The maximum current drawn through the power rails on a digital integrated circuit

$PV_{DD}$ Pad $V_{DD}$. Supply voltage for everything on the chip that is not part of the ALU core

$V_{DD}$ Supply voltage on the power rails on a digital integrated circuit

ALU Arithmetic Logic Unit

ASIC Application-Specific Integrated Circuit

CMOS Complementary Metal-Oxide-Semiconductor

CPU Central Processing Unit

DC Direct Current

DCVSL Differential Cascode Voltage Switch Logic

FR-4 Flame Resistant, grade designation 4. Glass-reinforced epoxy laminate sheet.

GALS Globally Asynchronous, Locally Synchronous

GCC GNU Compiler Collection

I/O Input/Output

IC Integrated Circuit

JLCC J-Leaded Chip Chip Carrier

LSB Least Significant Bit

MOSFET Metal-Oxide-Semiconductor Field Effect Transistor

MSB Most Significant Bit

NMOS n-channel MOSFET

OF Overflow

PCB Printed Circuit Board

PMOS p-channel MOSFET

PWB Printed Wiring Board

SMD Surface Mount Device

TSMC Taiwan Semiconductor Manufacturing Company

UART Universal Asynchronous Receiver/Transmitter

USB Universal Serial Bus

VHDL Very-high-speed integrated circuits Hardware Description Language

# Chapter 1

# Introduction

The world of digital electronics today is dominated by synchronous logic[23]. In synchronous logic circuits, clock signals are used to control the data flow. Inputs are read at regular intervals, dictated by the clock signal. This means that all outputs must be stable and ready before the next clock cycle. To ensure correct operation, the designer must set the clock speed such that all components are able to generate stable output signals within a single clock cycle. This sets an upper limit on the clock frequency. Within each clock domain, the maximum clock frequency is limited by the slowest component in the circuit, also known as the critical data path[12]. The complexity and subsequent completion time for each component can vary greatly. As a consequence, the outputs from some components will be ready long before they are actually read by other components. The result is considerable timing overhead for faster calculations.

Asynchronous logic, on the other hand, operates without clock signals. In asynchronous circuits, output signals are accompanied by ready-signals[23], indicating to the receiving component(s) when the data can be read. Ideally, this would eliminate the wasteful idle time, allowing the circuit to operate as fast as the individual components allow. In reality, the ready-signals themselves will introduce some overhead.

In this thesis, different approaches to asynchronous data processing are explored. A limited instruction set asynchronous ALU is designed and produced, which will be presented in detail.

# Chapter 2

# Why Asynchronous?

A major motivation for switching from synchronous to asynchronous design is to get rid of the global clock signal. Clock generation and distribution can consume a large amount of power. For instance, in a modern CPU, up to 60% of $I_{DDmax}$ can be consumed by the clock network[25]. In order to reach all components on a chip, the clock is often implemented as a metal grid in layout, covering a large portion of the chip area[13]. Because the grid covers such a large area, it effectively becomes a large capacitor that charges and discharges with every clock cycle.

Standby power consumption is usually very low in asynchronous systems, especially when implemented in CMOS. This is not unique to asynchronous systems, but rather a general property of CMOS circuits. However, asynchronous systems are event-driven. This means that signals only transition when there is new data input to the system. When there is no work to be done, the system will lie completely dormant. In synchronous systems, the clock will still be running even if there are no new inputs to be processed. Larger synchronous systems can achieve the same effect by using sleep states to disable the clock. Clock gating can also be used, so that the clock is only active where it is needed at any given time[3].

Correct timing is essential in synthesis of synchronous systems. To ensure that all components generate stable output signals within a single clock cycle, a predefined supply voltage must be maintained. As supply voltage is lowered, logic gates will operate more slowly. If the voltage drops below a certain threshold, it is no longer guaranteed that output signals are valid when the next clock cycle arrives. Other parameters such as increased operating temperature, or transistor mismatch during chip production, can also negatively impact the overall speed of the system. When setting the clock speed of a synchronous system, these factors have to be taken into account. The clock speed must be set low enough to allow a certain tolerance for imperfections.

Asynchronous systems tend to be more robust in regards to variance in supply voltage and other parameters. In fact, the voltage can be

lowered intentionally. This will result in reduced speed, but also reduced power consumption. As long as the the data remains intact, i.e. all combinatorial functions are carried out correctly, the supply voltage can be set as low as desired. A good example of this robustness is the first fully asynchronous microprocessor, designed at Caltech in 1989[21]. The processor could handle a $V_{DD}$ ranging from 0.35V to 7V. During an experiment, the designers cooled the processor in liquid nitrogen, almost doubling the number of instructions per second.

All electronics generate electrical noise. Digital systems in particular, often exhibit sharp noise spikes because of the way charge carriers suddenly move back and forth when signals change. In CMOS logic, noise is generated primarily when signals transition from 0 to 1 or vice versa, which is when the transistors switch between cut-off and saturation. In clocked logic, this translates to a regular powerful noise spike for each clock edge. Clockless logic also exhibits noise, but the noise tends to be more evenly distributed in time. In figure 2.1 on the facing page we see a good example of this difference. On the left is the measured substrate noise on a chip with 36 clocked pseudorandom number generators. On the right is the measured substrate noise on the same chip, but now with 36 clockless pseudorandom number generators. The clock rate on the left is set such that the throughput of both the clocked and clockless circuits are the same.

So why is synchronous electronics still so dominant in the industry? The fact is that asynchronous circuits have many drawbacks. While it is true that getting rid of the clock signal offers many benefits, having a clock signal greatly simplifies the design process[23]. A regular clock allows much of the design and simulation to be performed in discrete time, decreasing the complexity and resource requirements during development. Simply put, the designer only needs to worry about signal levels at the rising or falling clock edge. In between clock edges, signals may take on intermediate values, but it doesn't matter as long as the signals settle before the next clock edge.

To this date, practically all commercially available high-level digital design tools assume clocked operation[18]. Some VHDL compilers, such as Xilinx ISE and Altera Quartus can be used to generate asynchronous circuits, but the software is not designed for it and may generate many warnings along the way. Avoiding hazards and race conditions is a challenge, as the software cannot easily guarantee correct timings when there are no clocked registers involved.

Asynchronous circuits tend to have a higher number of transistors than their synchronous counterparts, depending on the particular design topology being used. Without a clock, data must be synchronized by other means, such as handshaking[23]. This requires additional logic,

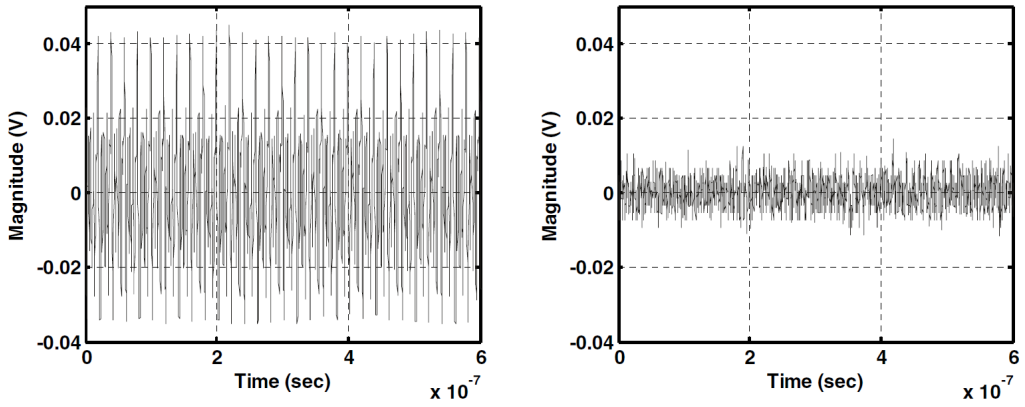



Figure 2.1: Measured substrate noise for an array of synchronous (left) and asynchronous (right) pseudorandom number generators[17].

which increases the transistor count and silicon area. Some asynchronous topologies also require hazard-free logic, pushing the transistor count even higher. The end result of this is increased total gate delays and reduced overall speed.

## 2.1 Asynchronous Methodologies

There are many ways to achieve asynchronous operation. Each approach has its pros and cons. In the following sections, some of the different alternatives are outlined.

It should be noted that for very large-scale systems, with millions of transistors, the asynchronous methodologies listed in this chapter can be used in conjunction with synchronous logic to produce globally asynchronous, locally synchronous (GALS) circuits[10]. In these circuits, each local clock signal drives a limited number of transistors. Asynchronous handshaking is only used to send and receive data between the clocked subcircuits. This approach limits the impact of clock skew, which is often a problem in large synchronous systems.

### 2.1.1 Matched Delays

A simple way to achieve asynchronous operation is to use preconfigured delays between circuit elements[16]. Consider the processing pipeline shown in figure 2.2 on the next page. In this methodology, logic blocks are accompanied by delay elements, which dictate when data is shifted from one block to the next. When data enters a block in the pipeline, the next block will not receive data until the previous delay element sends a ready-signal. The length of this wait time is determined by the delay

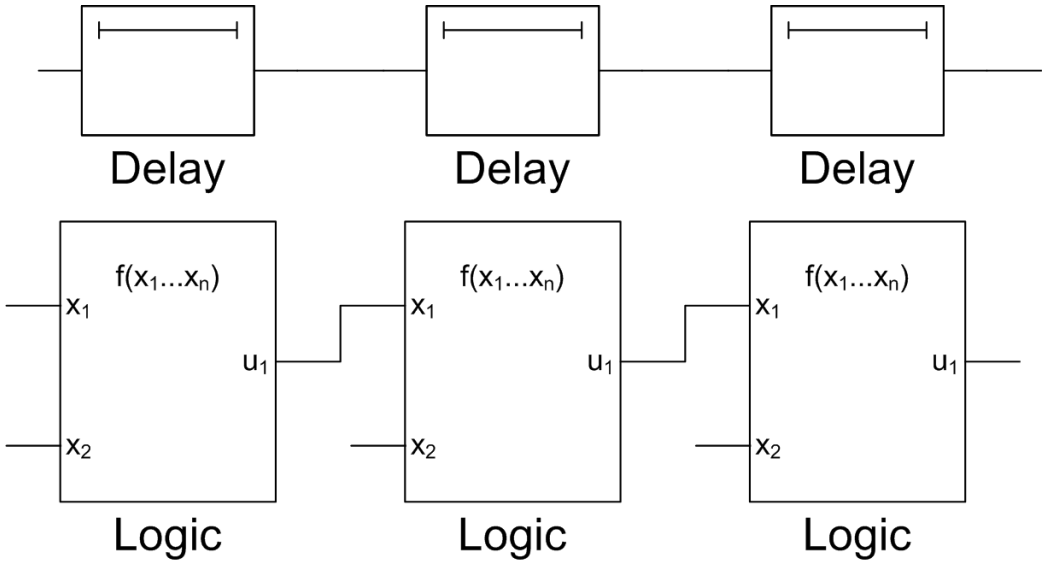



Figure 2.2: A processing pipeline using delay elements to synchronize data.

element. Each delay element is configured to produce a wait time equal to or longer than the processing time of its corresponding logic block.

With this approach, the logic blocks themselves can be constructed just as in a synchronous circuit. Glitch-free operation is not required, as long as the outputs settle within the preset delay time. The delay elements can be of very simple design, for example a capacitor discharging through a resistor. In this case, the delay time can be tuned by varying the size of the capacitor and/or the resistor.

The major downside of this approach is that all the delays must be separately configured during circuit design. This configuration can be very time-consuming, as precise timing simulations are required to determine the completion time of each logic block. The designer(s) must also take component mismatch, operating temperature and other variables into account, allowing a certain timing overhead. To allow for these imperfections, a higher delay time must be set.

The computing time for a logic block can vary a great deal, depending on the input data. A ripple adder, for example, can present valid outputs with minimal delay if there are no carries to be propagated. But if a carry must be propagated all the way from LSB to MSB, the processing delay increases substantially. Because of this variance in processing speed, the wait time in a delay element must be set long enough to allow for worst-case processing time in its corresponding logic block, just like in a clocked system.

Although an asynchronous circuit using matched delays can operate with varying supply voltage, the delay time may not be optimal over the entire voltage range. Changing the supply voltage can affect the logic

elements and the delay elements differently. If $V_{DD}$ is decreased, the processing time in a logic block will increase. However, the wait time in the delay element may not increase by the same amount, leading to timing errors. You could counter this by setting a longer delay time, but this could lead to unnecessary long delay time for higher $V_{DD}$.

### 2.1.2 Current-Sensing Completion Detection

The matched delay methodology outlined in the previous section uses preconfigured timing estimates to determine when output data from a logic block is valid. The delay elements are completely separate from the actual logic. Ideally, a delay element would be connected to its corresponding logic block and detect when the output data is valid. This detection can be accomplished by using current-sensing circuitry[16]. In conventional CMOS, the static current consumption is very low compared to the dynamic current consumption. Power is mostly dissipated during signal transitions. For a logic block receiving new data, current consumption will be high at first, when internal signals switch between high and low logic states. When processing is complete and all signals have settled, the current consumption will drop sharply.

A current-sensing completion detector connected to a logic block monitors the current consumption as data processing occurs. When the current drops, processing is complete, and the completion detector sends a ready-signal to the next stage in the pipeline.

Because CMOS logic primarily draws current during signal transitions, the current-sensing scheme works well as long as data keeps changing for each processing cycle. However, problems arise when two or more identical sets of data are processed in sequence. Suppose a logic block has finished processing a data set, and then receives a new, identical data set. Because the inputs are the same as before, the internal signals and outputs will not change, and the current consumption will not increase. In this case, the completion detector will not be able to detect that a new data set has been processed.

### 2.1.3 Differential Cascode Voltage Switch Logic

The logic family that is the main focus of this thesis is called Differential Cascode Voltage Switch Logic (DCVSL)[19]. Its method of operation is quite different from conventional static CMOS. In this section, we will explore DCVSL by first comparing static CMOS- and single-ended Cascode Voltage Switch Logic gates, before moving on to DCVSL gates.

In static CMOS circuits, logic gates consist of one or more NMOS transistors driving the output to 0 or ground (pull-down), and a complementary set of PMOS transistors driving the output to 1 or $V_{DD}$ (pull-up). It's called

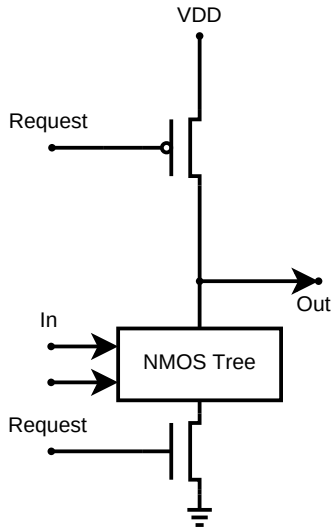



Figure 2.3: A single-ended CVSL gate.

*Static* CMOS because the outputs of a gate are at all times driven to either 0 or 1, meaning that a low resistance path always exists between the output and either ground or $V_{DD}$. The counterpart to static CMOS is dynamic CMOS, where outputs may temporarily be undriven, i.e. disconnected from ground and $V_{DD}$ by a transistor in cut-off[11].

In single-ended Cascode Voltage Switch Logic, the pull-down network (NMOS tree) is the same as for static CMOS, with an additional NMOS transistor to ground. The pull-up network is removed entirely, and the output is instead connected to $V_{DD}$ through a single PMOS transistor. A sketch of such a logic gate is shown in figure 2.3. Unlike static CMOS, the output from this gate is not driven at all times. The state of the output depends on the Request signal. As long as Request is 0, the output is pulled to 1 through the PMOS transistor, and the lowest NMOS transistor disconnects the rest of the circuit from ground. This is the idle, stable state of the gate. Because the NMOS tree is disconnected from ground, a change in the input at this time will not affect the output.

If we now set Request to 1, the output will no longer be pulled to 1, and the NMOS tree will be connected to ground. One of two things can happen at this point. Depending on the inputs, the output will either be pulled to 0 by the NMOS tree, or it will stay at 1. However, because $V_{DD}$ is disconnected, the output is a floating 1. Only parasitic capacitance is maintaining the voltage on the output, and because of leakage currents it will only stay valid for a short while[9].

Now let's say we make this gate differential instead of single-ended, as seen in figure 2.4 on the next page. In this gate, the NMOS tree is expanded to include a pull-down network for a second output, $\overline{Out}$. The second pull-down network is the boolean complement of the first pull-down network,

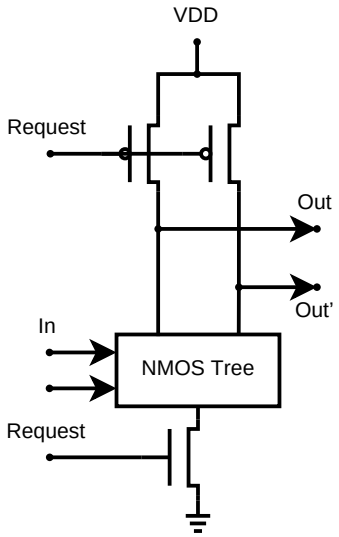



Figure 2.4: A DCVSL gate, using a differential signal path.

| Out | Out' | Logical value |
|---|---|---|
| 0 | 0 | Not used |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | Idle (not ready) |

Table 2.1: DCVSL output truth table.

such that when Out is 0, $\overline{Out}$ is 1 and vice versa.

We now have a DCVSL gate. This gate functions in the same way as the single-ended gate, except it generates a differential output signal. Instead of a single wire carrying the output data, we have two wires carrying opposite values when data is present. This is useful for clockless applications, because it allows us to see when the NMOS tree has finished evaluating the input. The possible states of the differential output are shown in table 2.1.

In the case of the single-ended gate, after Request is set to 1, we have to wait a certain amount of time to see whether or not the NMOS tree will pull the output down to 0. But how long do we have to wait? This depends on several factors, such as the complexity of the NMOS tree, rail voltage, transistor dimensions etc. With the differential gate on the other hand, one and only one of the output signals will be pulled down to 0. This means that as soon as Out and $\overline{Out}$ have differing values, we know that the output is ready. A simple way of detecting this is to connect a static NAND gate to the output, as shown in figure 2.5 on the next page. In the idle state, when $Out = \overline{Out}$, the NAND will generate a 0, indicating that the outputs are not ready. When Request goes to 1 and the NMOS tree evaluates the inputs, either Out or $\overline{Out}$ will be pulled to 0, and the NAND

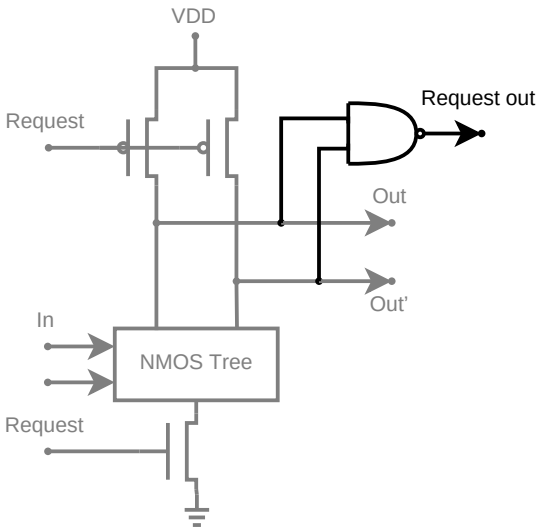



Figure 2.5: A DCVSL gate with NAND completion detection.

will generate a 1, indicating valid outputs.

Using differential outputs makes completion detection easy to implement, but it comes at a price. Using two wires to transmit the same amount of data as one wire is naturally inefficient by comparison. Because every logic gate has two pull-down networks instead of one, the transistor count and silicon area per gate increases. Two data outputs for each gate also means that wiring congestion can be a problem when interconnecting larger systems.

Hazard-free operation is a requirement when using DCVSL gates with NAND completion detection[27]. As we can see in figure 2.5, the NAND gate connected to the outputs will pull the Request out to 1 as soon as either Out or $\overline{Out}$ is pulled to 0. As such, the outputs must be ready and valid when the Request is generated.

We obviously have to make modifications to a conventional single-ended CMOS pull-down network in order to make it differential. Luckily, the modifications are easy to implement, and the procedure is the same for all logic gates. For any pull-down network used to generate an output Y, we need to add another pull-down network to generate the output $\overline{Y}$. De Morgan's theorem states that

$$\overline{A \cdot B} \equiv \overline{A} + \overline{B},$$
$$\overline{A + B} \equiv \overline{A} \cdot \overline{B}.$$

This theorem can be applied to any boolean expression in three simple steps:

1. Inverse all variables.

2. Replace all AND with OR, and vice versa.

3. Inverse the result.

The end result is a boolean expression that is equivalent to what we begin with. However, if we complete only the first two steps, we get a boolean expression that is the opposite of what we begin with, which is exactly what we need.

Let's look at an example. In figure 2.6 we see the pull-down network of a static CMOS NAND gate. The boolean expression for this gate is $Out = \overline{A \cdot B}$. If we create a new pull-down network by doing the first two steps above, and add this to the CMOS pull-down network, we get the DCVSL pull-down network shown in figure 2.7 on the next page. Note that the new transistors are connected in parallel instead of in series (changed NAND to NOR), and the inputs are inversed.

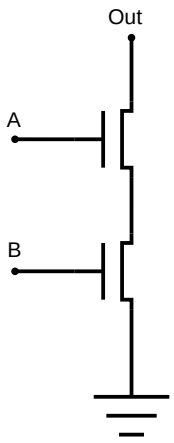



Figure 2.6: A CMOS NAND gate pull-down network.

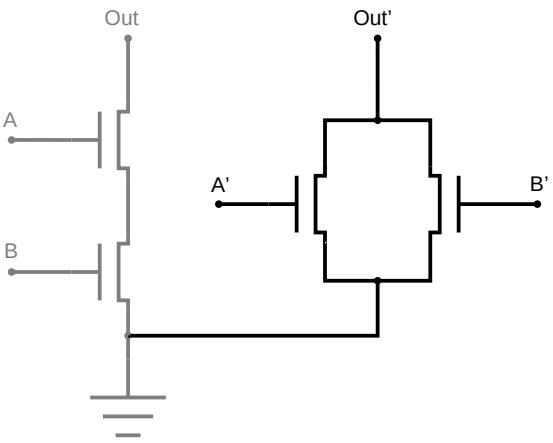



Figure 2.7: A DCVSL NAND gate pull-down network.

# Chapter 3

# ALU Design

In this chapter, the construction of an Arithmetic Logic Unit (ALU) in the low-power 90nm process from TSMC is presented. The ALU supports 8 different instructions or opcodes, and uses an 8-bit word length. There are 2 input channels $A_{7-0}$ and $B_{7-0}$, and one output channel $OUT_{7-0}$. In addition, there is an overflow (OF) output, used by the adder subcircuit to indicate data overflow in the output.

DCVSL logic is the basis of the ALU. All of the subcircuits that perform arithmetic and logic operations on the input data, are constructed using DCVSL gates.

All transistors, except line pull-up and input pad inverters, use minimum dimensions. In this particular process, the minimum transistor dimensions are $\frac{W}{L} = \frac{120nm}{100nm}$.

All schematics and layouts were made using Virtuoso Front to Back Design Environment version 5.1.0, from Cadence Design Systems.

## 3.1 Instruction Decoder

The ALU has a 3-bit control input, $S_{2-0}$ (Select). This signal, along with the request signal (REQ), is fed into the instruction decoder. A schematic is shown in figure 3.1 on the following page. The decoder forwards the incoming REQ signal to the appropriate subcircuit, which then starts processing the input data. In addition, the transmission gates following each subcircuit are enabled. The transmission gates require both an active-high and active-low enable signal, which is why the instruction decoder generates both REQ and $\overline{REQ}$ for each input combination. An example of a simulation run is shown in figure 3.2 on page 15. In this simulation, S is strobed at different rates, running through all eight combinations of control inputs. The request signals for the adder subcircuit are included as an example, showing that the adder will only receive the request signal when the ADD opcode (111) is present on S.

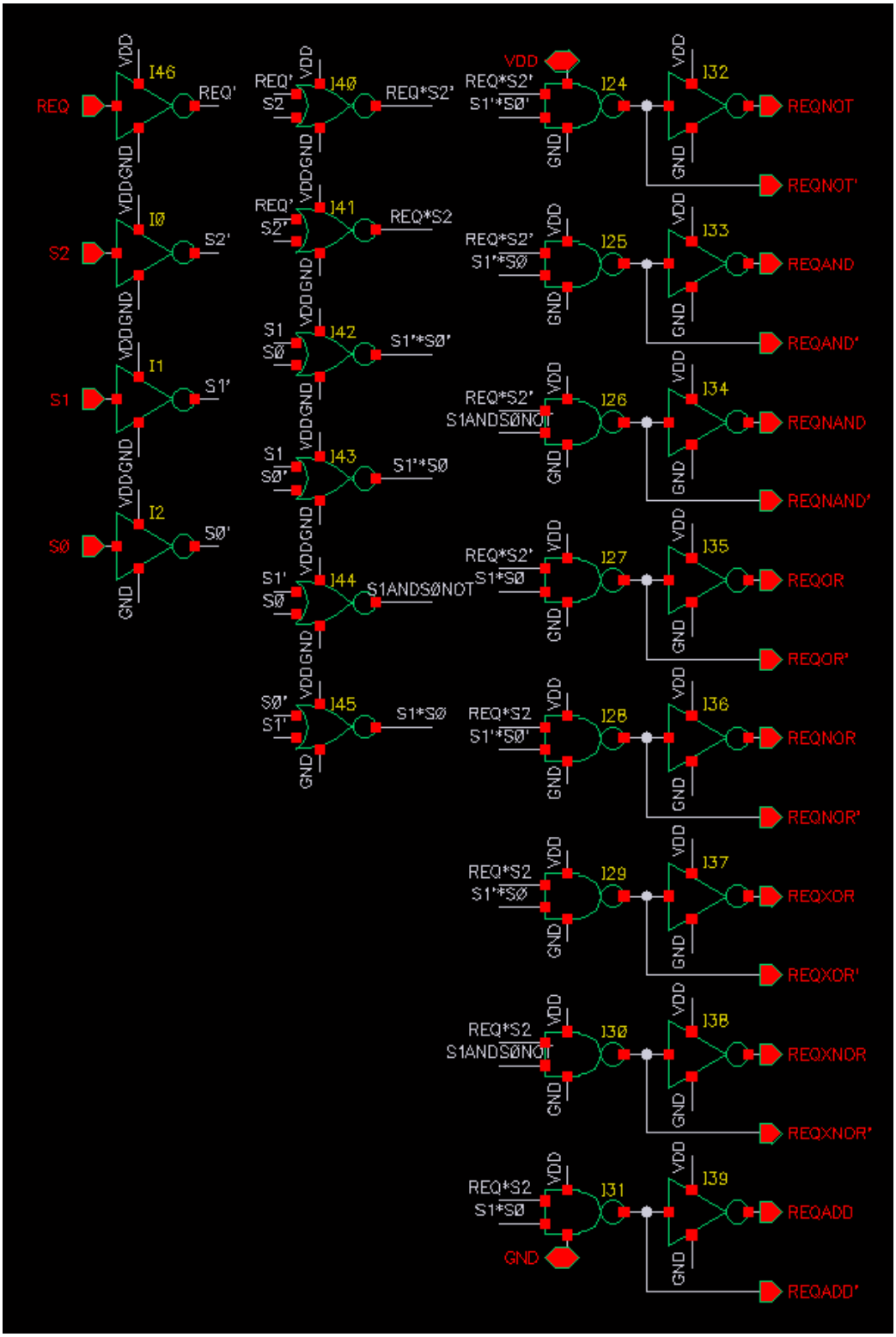



Figure 3.1: Schematic of the instruction decoder.

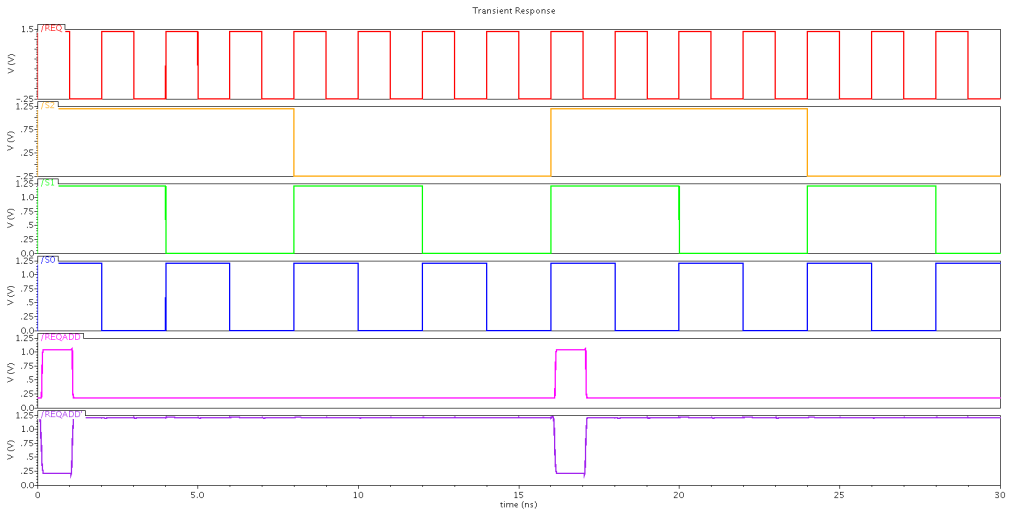



Figure 3.2: A 30ns transient simulation of the instruction decoder, strobing the inputs at different rates.

The layout of the instruction decoder is shown in figure 3.3 on the following page.

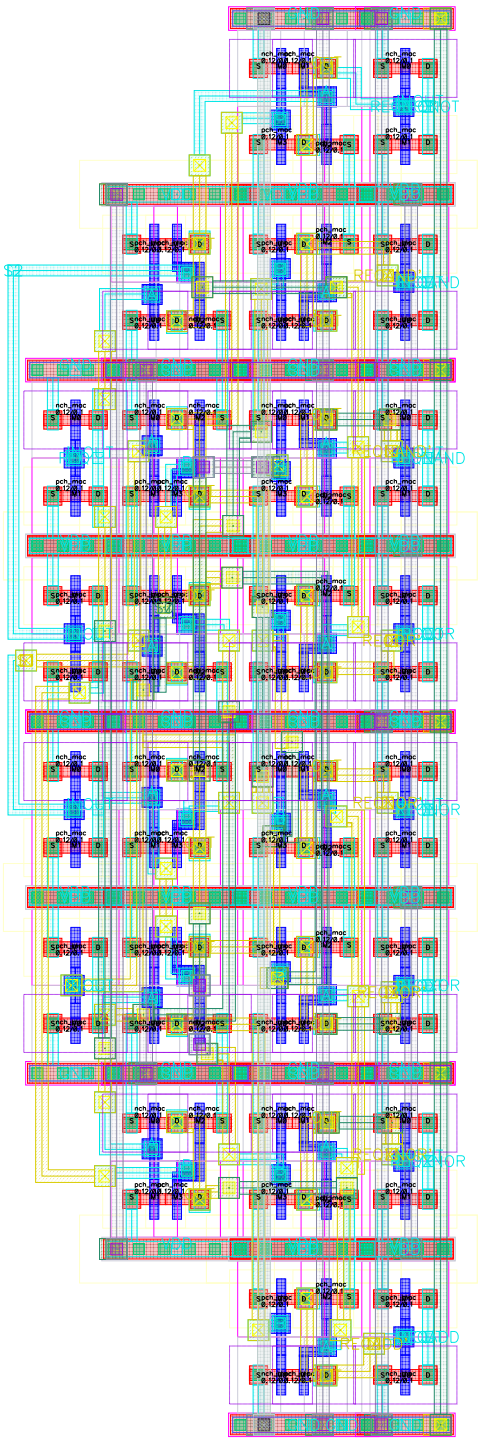


Figure 3.3: Layout of the instruction decoder. Height = 15.78$\mu$m, width = 5.23$\mu$m.

## 3.2 Opcodes

The ALU can perform 8 different operations on the input data. The operation to be executed at any given time is determined by the 3-bit Select input to the instruction decoder. The supported operations are listed in table 3.1.

The ALU uses 8-bit input- and output buses. The subcircuits that perform each operation consist of DCVSL gates stacked together in layout, where every other gate is flipped upside down, such that each gate shares $V_{DD}$- and ground rails with the gates directly above and beneath it.

In figure 3.4 on the following page we see the most basic DCVSL gate in the ALU: The NOT gate. Strictly speaking, there is no need for NOT gates when using differential data lines, because for each data bit X we already have $\overline{X}$. Nevertheless, it is included in this ALU design because it is a good indicator of the best-case performance of any DCVSL gate. It is the simplest DCVSL gate possible, which makes it a good reference when measuring the performance of other, more complex DCVSL gates.

Schematics and layouts of the AND and OR gates are shown in figures 3.5-3.6. NAND and NOR gates are not shown, as the only difference is that their outputs are swapped.

In all the DCVSL gates shown here, the PMOS transistors are abutted[2] since they share a common source. The same goes for the NMOS transistors. In the AND and OR gates, NMOS transistors connected in series are implemented as two gates covering a common channel.

The XOR gate is created by combining DCVSL NAND gates, as shown in figure 3.7 on page 21. In the schematic, REQ for the final DCVSL NAND gate is generated in the cell COMPLGEN2_v1, by static NANDing each differential output from the previous DCVSL NAND cells, and then static ANDing these results.

| Opcode | Operation |
|---|---|
| 000 | NOT |
| 001 | AND |
| 010 | NAND |
| 011 | OR |
| 100 | NOR |
| 101 | XOR |
| 110 | XNOR |
| 111 | ADD |

Table 3.1: List of supported ALU operations.

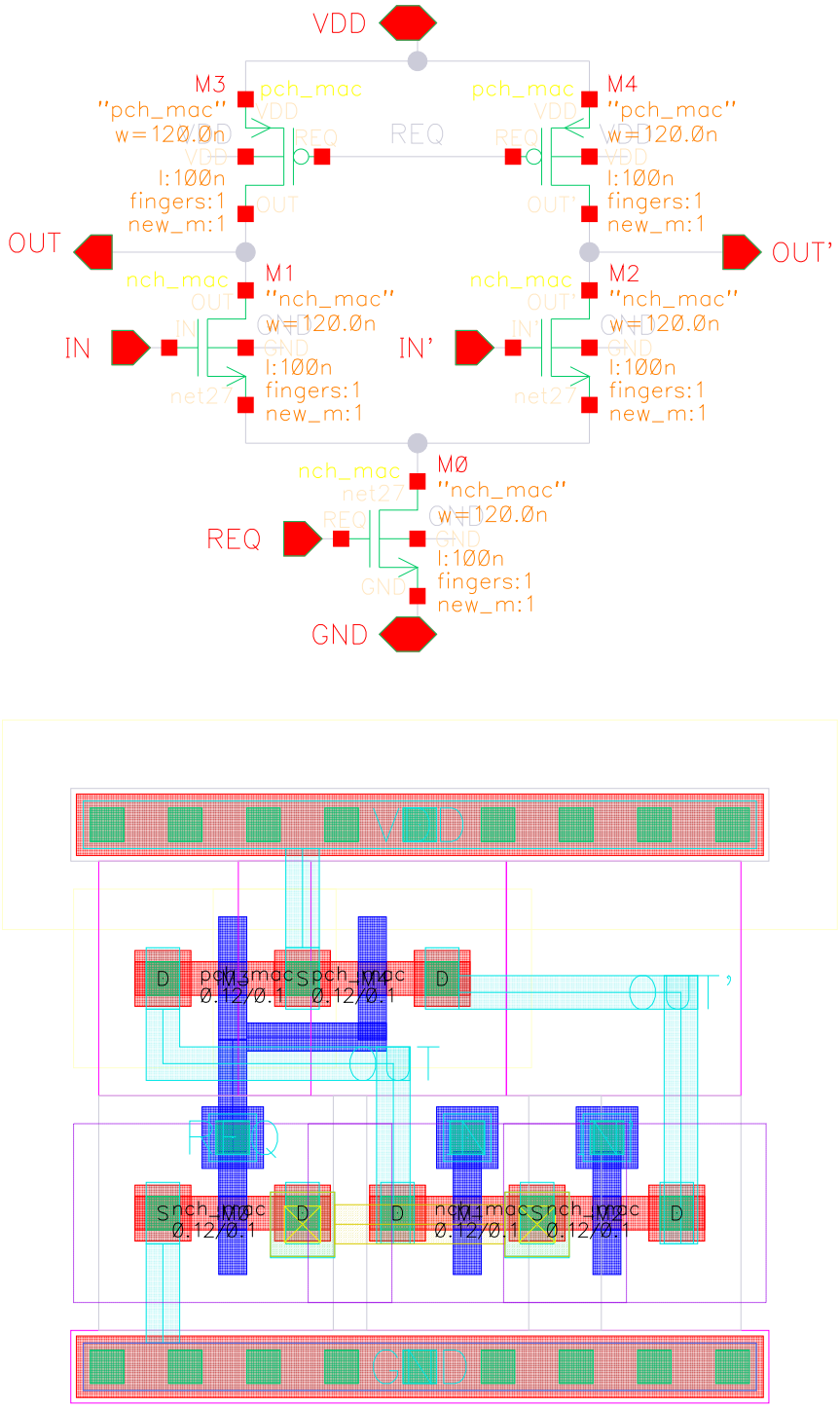



Figure 3.4: Top: Schematic of a DCVSL NOT gate. Bottom: Layout of a DCVSL NOT gate. Height = 2.45$\mu$m, width = 2.99$\mu$m.

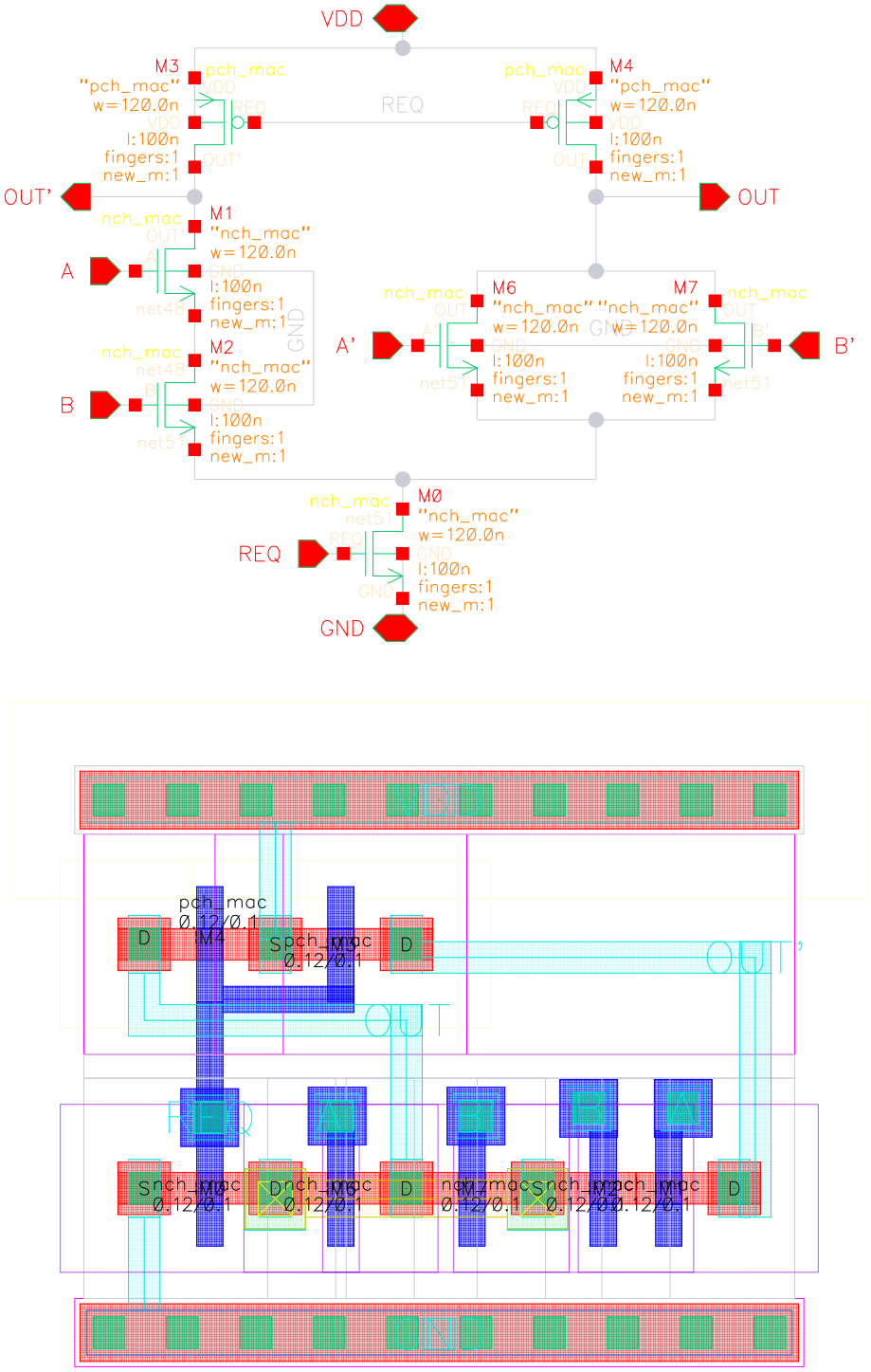



Figure 3.5: Top: Schematic of a DCVSL AND gate. Bottom: Layout of a DCVSL AND gate. Height = 2.54$\mu$m, width = 3.27$\mu$m.

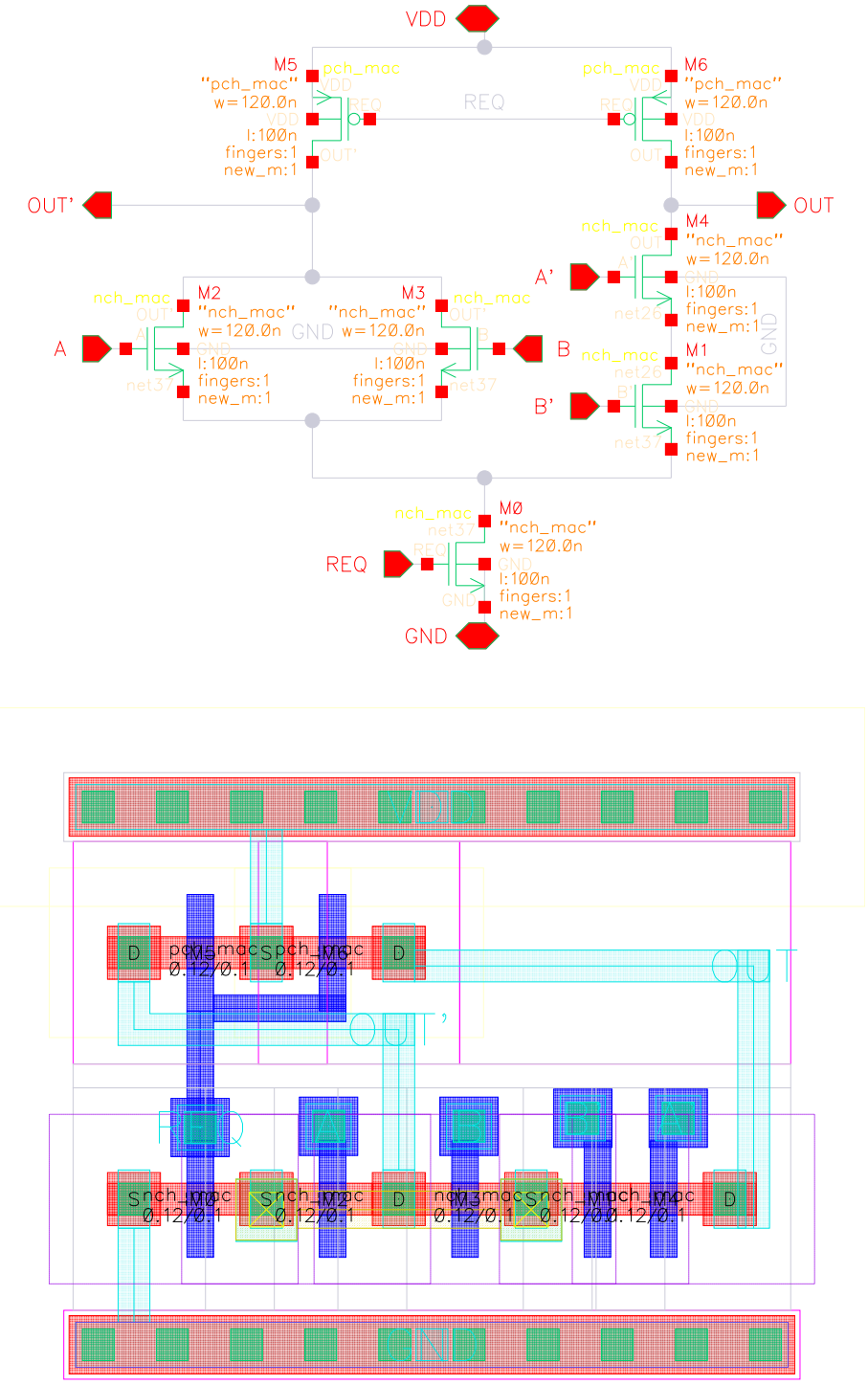



Figure 3.6: Top: Schematic of a DCVSL OR gate. Bottom: Layout of a DCVSL OR gate. Height = 2.54$\mu$m, width = 3.27$\mu$m.

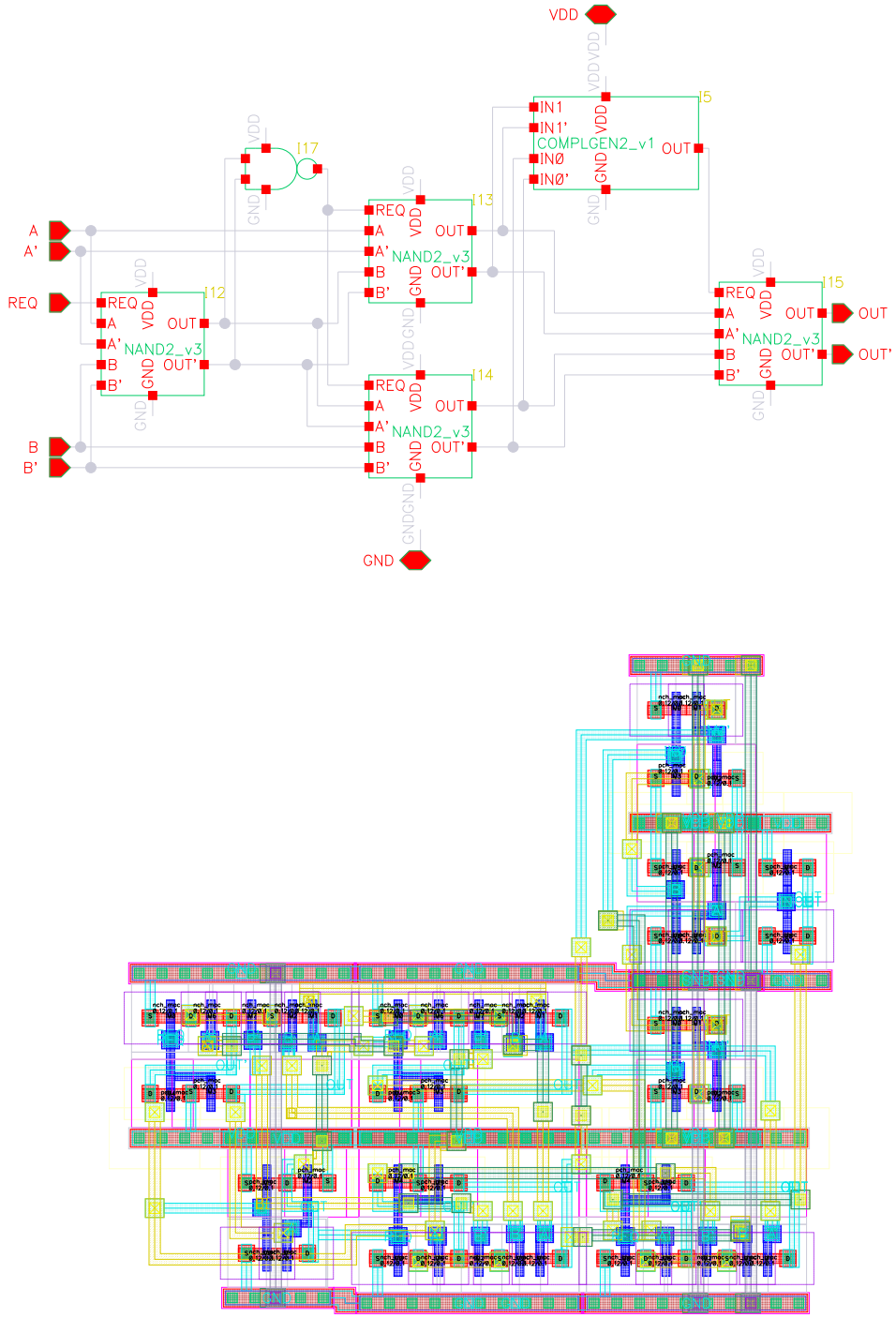



Figure 3.7: Top: Schematic of a DCVSL XOR gate. Bottom: Layout of a DCVSL XOR gate. Height = 8.11$\mu$m, width = 9.15$\mu$m.

### 3.2.1 ADD

The adder used in this ALU is a Kogge-Stone adder, a parallel prefix form carry look-ahead adder. The design was introduced by Peter Kogge and Harold Stone in 1973[14], and is widely used in modern processors because of its high performance and scalability[26]. As with any carry look-ahead adder, the principle idea is to calculate the carries for each bit before the preceding sum has been determined, greatly decreasing the time needed to calculate each bit result, especially for more significant bits.

The schematic for the Kogge-Stone adder is shown in figure 3.8 on the facing page. At first glance, the schematic seems quite complex, but we can divide the building blocks of the adder into three separate types of cells: top cells, mid cells and bottom cells. In the following sections, each of these cells will be explained in detail.

Figure 3.8: Schematic of a DCVSL 8-bit Kogge-Stone adder with overflow output.

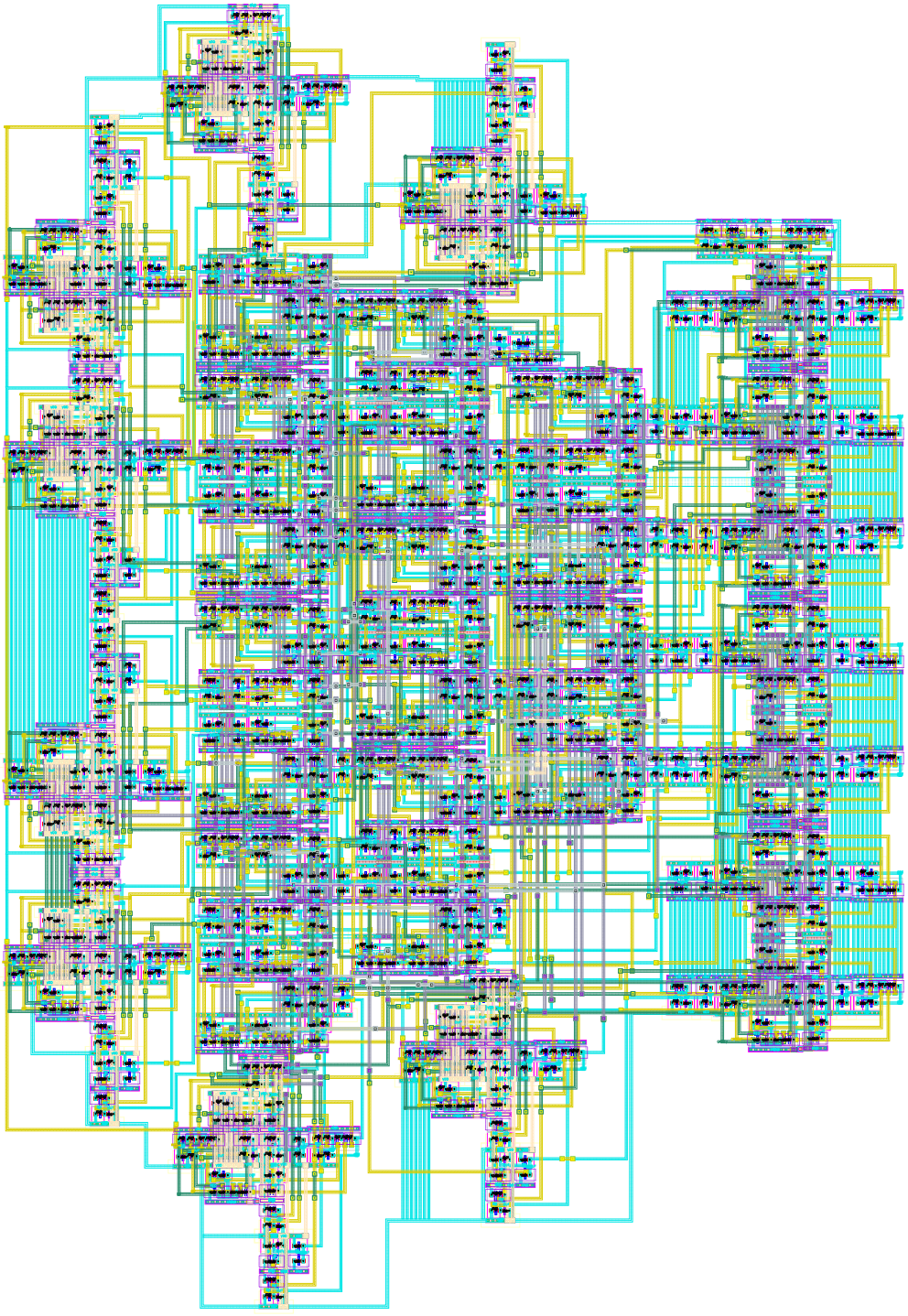


Figure 3.9: Layout of a DCVSL 8-bit Kogge-Stone adder with overflow output. Height = 72.81$\mu$m, width = 50.32$\mu$m.

### The top cell

In the schematic for the adder we see eight top cells, one for each pair of binary inputs. As the name suggests, the top cells are located in the top of the schematic. From right to left, input data $A_0$ and $B_0$ (LSB), $A_1$ and $B_1$, $A_2$ and $B_2$, ... up to $A_7$ and $B_7$ (MSB) enter the top cells from above.

In figure 3.10 on the next page, we see the schematic and layout for the top cell. The inputs and outputs are shown below.

**Inputs:**

- REQIN - Incoming request signal. Indicates new data input. Generated by the instruction decoder.
- A, A' - Data input, channel A. Differential line. Generated by circuitry external to the ALU.
- B, B' - Data input, channel B. Differential line. Generated by circuitry external to the ALU.

**Outputs:**

- REQOUT - Outgoing request signal. Indicates P and G outputs ready for retrieval by mid cells.
- P, P' - The "Propagate" signal. Differential line. Used by mid cells to calculate carries, or by a bottom cell to calculate a sum.
- G, G' - The "Generate" signal. Differential line. Used by mid cells to calculate carries.

The function of the top cell is to generate two signals from its input bits: A Propagate signal P and a Generate signal G. The Propagate signal denotes whether or not an incoming carry from the right would be propagated to the left, like the carry in a ripple adder. For bit position i, where position 0 is LSB, $P_i$ is 1 if a carry from position i-1 would be propagated to position i+1. The Generate signal on the other hand, denotes whether or not the bit position itself will generate a carry to the left, regardless of what is on the right. For bit position i, $G_i$ is 1 if a carry is generated in that position and sent to position i+1. Both of these signals are created only by looking at the data inputs $A_i$ and $B_i$. The expressions for P and G are shown below. Note that they are exactly the same as for a half adder, with P being the sum and G being the carry. The signals P and G are used further down in the schematic, by the mid and bottom cells.

$$P = A \oplus B \tag{3.1}$$

$$G = A \cdot B \tag{3.2}$$

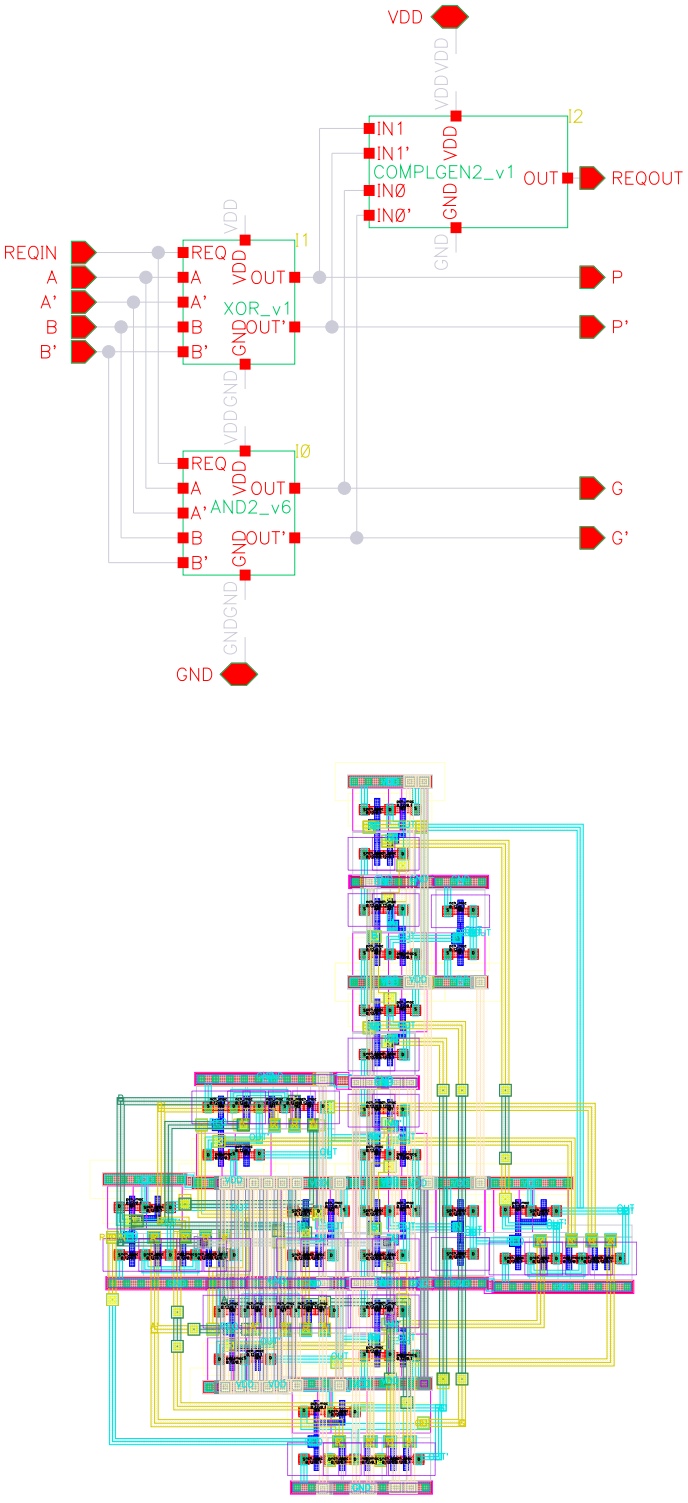



Figure 3.10: Top: Schematic of the top cell in the Kogge-Stone adder. Bottom: Layout of the top cell in the Kogge-Stone adder. Height = 14.35$\mu$m, width = 10.71$\mu$m.

### The mid cell and carry tree

The carry tree-structure in the adder is composed entirely of mid cells. These take up most of the total adder area and interconnect. The schematic and layout for the mid cell is shown in figure 3.11 on the following page, and the inputs and outputs are listed below.

**Inputs:**

- REQIN - Incoming request signal. Indicates that inputs PIN and GIN are ready. Generated by the top or mid cell directly above this one.
- REQPREV - Incoming request signal. Indicates that inputs PPREV and GPREV are ready. Generated by a top or mid cell above and to the right of this one.
- PIN, PIN' - Incoming "Propagate" signal. Differential line. Generated by the top or mid cell directly above this one.
- PPREV, PPREV' - Incoming "Propagate" signal. Differential line. Generated by a top or mid cell above and to the right of this one.
- GIN, GIN' - Incoming "Generate" signal. Differential line. Generated by the top or mid cell directly above this one.
- GPREV, GPREV' - Incoming "Generate" signal. Differential line. Generated by a top or mid cell above and to the right of this one.

**Outputs:**

- REQOUT - Outgoing request signal. Indicates P and G outputs ready for retrieval by mid or bottom cells.
- P, P' - The "Propagate" signal. Differential line. Used by mid cells below to calculate carries.
- G, G' - The "Generate" signal. Differential line. Used by mid cells below to calculate carries, or by a bottom cell to calculate a sum.

The function of the mid cell, just like the top cell, is to generate Propagate and Generate signals P and G. However, a mid cell in bit position i does not look at the input bits $A_i$ and $B_i$ directly. Instead, it combines Propagate and Generate signals from other cells above it in the schematic, in order to generate new Propagate and Generate signals. Depending on how far down the carry tree a particular mid cell is located, it can either get its P and G inputs from other mid cells, or directly from top cells. The expressions for P and G are shown below.

$$P = P_{in} \cdot P_{prev} \tag{3.3}$$

$$G = G_{in} + (P_{in} \cdot G_{prev}) \tag{3.4}$$

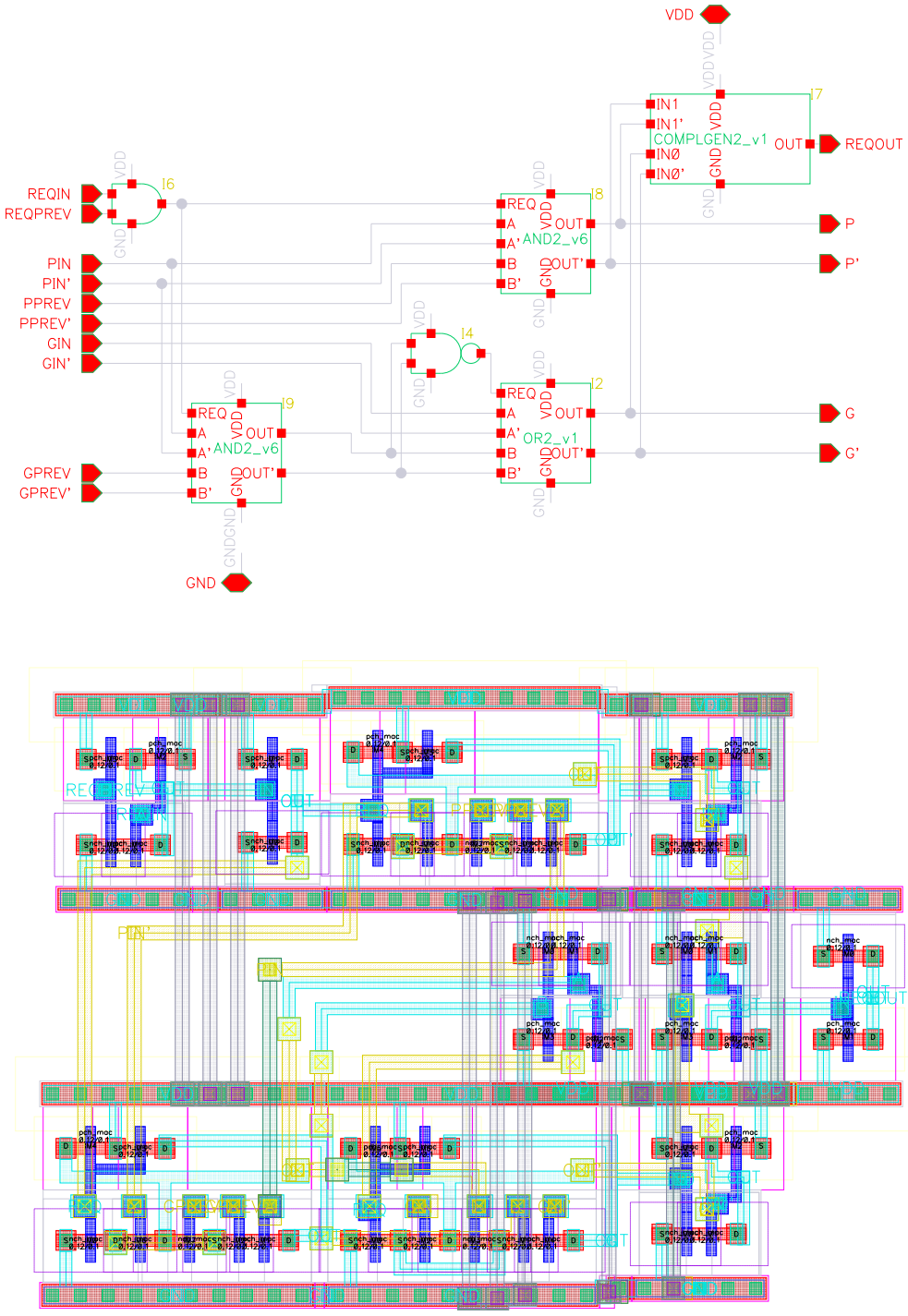



Figure 3.11: Top: Schematic of the mid cell in the Kogge-Stone adder. Bottom: Layout of the mid cell in the Kogge-Stone adder. Height = 6.56$\mu$m, width = 9.21$\mu$m.

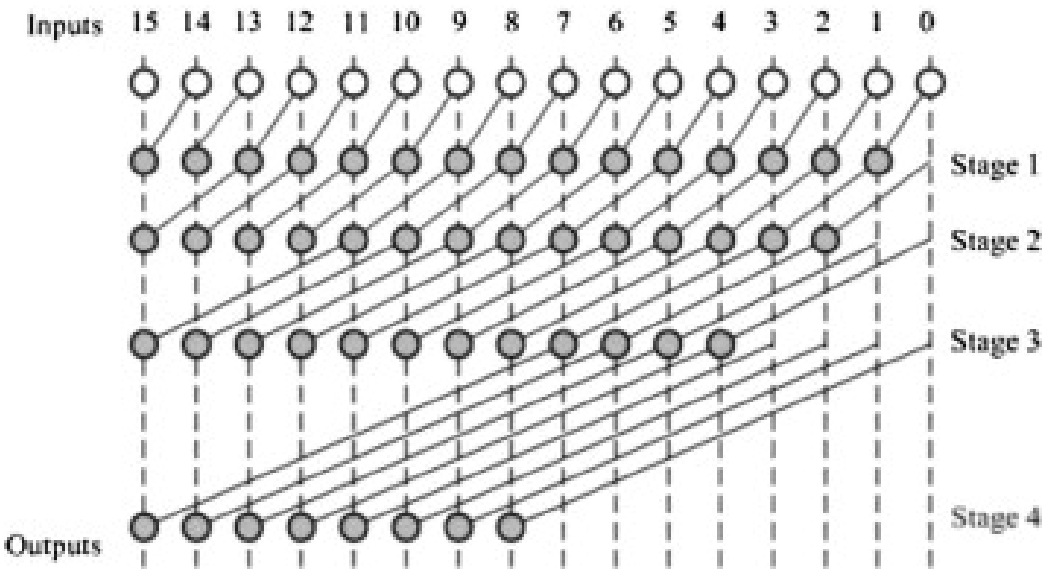



Figure 3.12: Parallel prefix graph of a 16-bit Kogge-Stone adder [4].

An illustration of the carry tree is shown in figure 3.12. The white circles represent top cells, and the grey circles represent mid cells. For all mid cells, $P_{in}$ and $G_{in}$ are read from the cell directly above it, accompanied by the request signal REQIN. $P_{prev}$ and $G_{prev}$ however, are a different story. The cells in stage 1 read REQPREV, $P_{prev}$ and $G_{prev}$ from bit position i-1, i.e. 1 to the right. The cells in stage 2 read REQPREV, $P_{prev}$ and $G_{prev}$ from bit position i-2, i.e. 2 to the right, and the cells in stage 3 read REQPREV, $P_{prev}$ and $G_{prev}$ from bit position i-4, i.e. 4 to the right.

As we can see, the horizontal distance between one cell's P or G output to its corresponding $P_{prev}$ or $G_{prev}$ input doubles at each vertical stage. This is the source of the Kogge-Stone adder's scalability. If we were to double the width of the data words, for instance go from 8-bit to 16-bit, we would only need one more vertical stage. Note that the carry-tree illustrated here is for a 16-bit adder, not an 8-bit one. Therefore it has one more vertical stage than our adder.

### The bottom cell

So far we have only generated Propagate and Generate signals. We still haven't calculated the final sum bits. This is accomplished by the bottom cell, shown in figure 3.13 on page 31. The inputs and outputs to the bottom cell are listed below.

**Inputs:**

- REQIN - Incoming request signal. Indicates that input P is ready. Generated by the top cell in the current bit position.
- REQPREV - Incoming request signal. Indicates that input GPREV is

ready. Generated by a top or mid cell above and to the right of this cell.

- P, P' - Incoming "Propagate" signal. Differential line. Generated by the top cell in the current bit position.
- GPREV, GPREV' - Incoming "Generate" signal. Differential line. Generated by a top or mid cell above and to the right of this cell.

**Outputs:**

- S, S' - The final sum for this bit position. Differential line.

The bottom cell produces a sum by performing a simple XOR operation:

$$S = P_{in} \oplus G_{prev} \tag{3.5}$$

The $P_{in}$ in this expression is the Propagate signal generated by the top cell in the current bit position, and $G_{prev}$ is the final Generate signal created by the last vertical stage in the previous bit position.

Considering that the top cell calculates P by XORing A and B, and that $G_{prev}$ is the carry-in for the current bit position, the expression above is equivalent to that of a full adder:

$$S = A \oplus B \oplus C_{in} \tag{3.6}$$

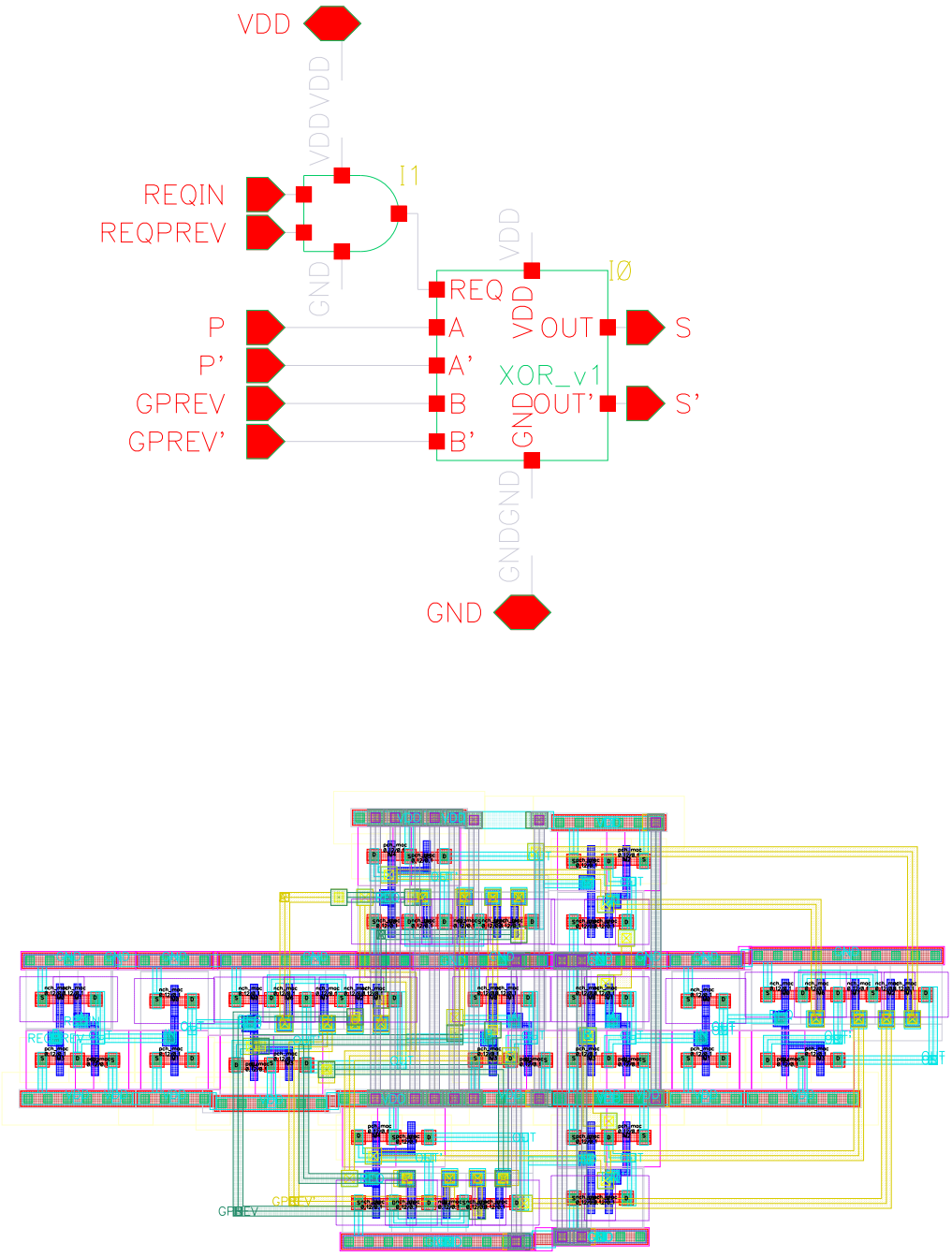



Figure 3.13: Top: Schematic of the bottom cell in the Kogge-Stone adder. Bottom: Layout of the bottom cell in the Kogge-Stone adder. Height = 6.54$\mu$m, width = 13.47$\mu$m.

### How it all works

Now that we have gone through the workings of each individual type of cell, let's see how it all fits together. We can do this by typing out the expressions for the final sums one by one. For P and G below, we're using two indices x and y to indicate which cell the signals originate from. x is the bit position, where x=0 is LSB and x=7 is MSB. y is the tree depth, e.g. stage number in figure 3.12 on page 29. For instance, y=0 is a top cell, and y=3 is a stage 3 mid cell. As an example, $P_{(4,3)}$ is the Propagate signal generated by the mid cell in bit position 4, in stage 3 e.g. third mid cell from the top.

For bit position 0, we have no carry-in, which means the sum is simply

$$\begin{aligned} S_0 &= A_0 \oplus B_0 \\ &= P_{(0,0)} \end{aligned}$$

For bit positions 1 and higher, we need to take the incoming carry into account. The sum in position 1 is

$$\begin{aligned} S_1 &= (A_1 \oplus B_1) \oplus (A_0 \cdot B0) \\ &= P_{(1,0)} \oplus G_{0,0} \end{aligned}$$

For bit position 2, the sum is

$$\begin{aligned} S_2 &= (A_2 \oplus B_2) \oplus (((A_1 \oplus B_1) \cdot (A_0 \cdot B_0)) + (A_1 \cdot B_1)) \\ &= P_{2,0} \oplus ((P_{(1,0)} \cdot G_{0,0}) + G_{1,0}) \\ &= P_{2,0} \oplus G_{1,1} \end{aligned}$$

For bit position 3, the sum is

$$\begin{aligned} S_3 &= (A_3 \oplus B_3) \\ &\quad \oplus ((((A_2 \oplus B_2) \cdot (A_1 \oplus B_1)) \cdot (A_0 \cdot B_0)) \\ &\quad + (((A_2 \oplus B_2) \cdot (A_1 \cdot B_1)) + (A_2 \cdot B_2))) \\ &= P_{3,0} \oplus (((P_{2,0} \cdot P_{1,0}) \cdot G_{0,0}) + ((P_{2,0} \cdot G_{1,0}) + G_{2,0})) \\ &= P_{3,0} \oplus ((P_{2,1} \cdot G_{0,0}) + G_{1,1}) \\ &= P_{3,0} \oplus G_{2,1} \end{aligned}$$

The expressions for the sums in bit positions 4 through 7 are not shown here, as the equations get increasingly complex. However, we can see that each expression boils down to an XOR operation on a Propagate and a Generate signal. For each new bit position, we only need to calculate one additional Generate signal.

## 3.3 Transmission Gates and Standby Pull-up

The data outputs from each DCVSL subcircuit all connect to the same output register (described in the next section). To ensure that only one subcircuit at a time can drive the ALU output register, transmission gates[20] are inserted at the outputs of each DCVSL subcircuit. The schematic and layout are shown in figure 3.14 on the following page. All data signals from the DCVSL subcircuits pass through a transmission gate before entering the output register. When a DCVSL subcircuit receives a request signal from the instruction decoder and starts processing input data, the same request signal is also sent to the transmission gates on the subcircuit output. The transmission gates will then be enabled and allow the data to pass through to the output register. All other transmission gates, connected to other DCVSL subcircuits, will still be disabled.

When the ALU is in standby, i.e. request in is low, no data processing occurs. In this case, no DCVSL subcircuit is driving the data lines to the output register. When request in is low, PMOS transistors are used to pull the internal data lines high.

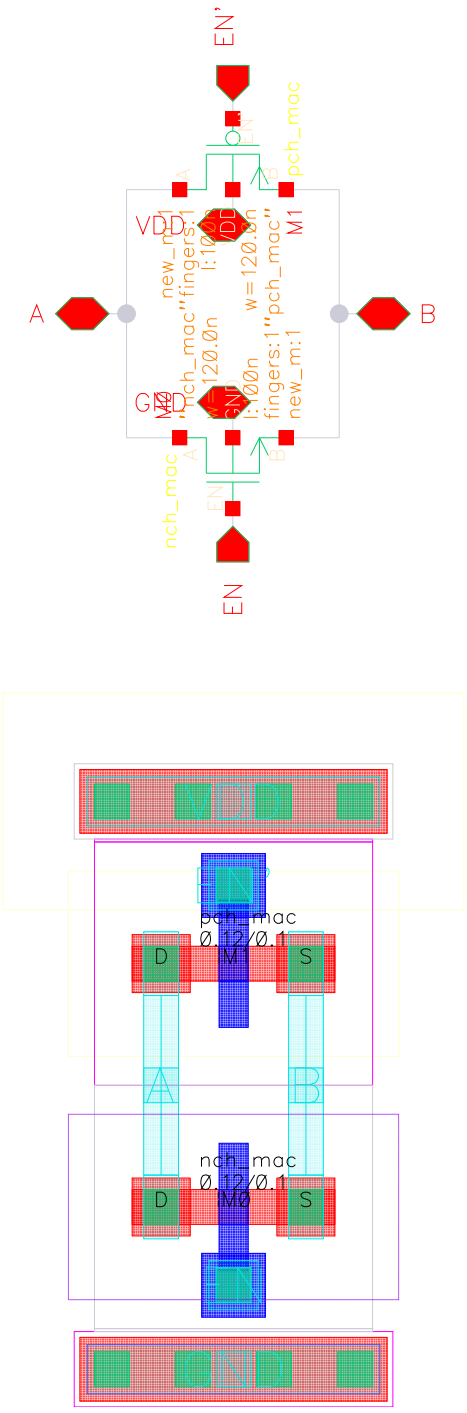



Figure 3.14: Top: Schematic of a transmission gate. Bottom: Layout of a transmission gate. Height = 2.47$\mu$m, width = 1.59$\mu$m.

## 3.4 Completion Generation

After a set of input data has been processed, a currently active DCVSL subcircuit sends a request signal to the output register. This internal request signal is generated by the completion generator, shown in figure 3.15 on the next page. Each DCVSL subcircuit has a completion generator connected to its data outputs.

The completion generator is composed of static NAND- and AND gates. Each differential data output bit from a DCVSL subcircuit is NANDed, and the 8 resulting values are then ANDed together by a tree of 2-input AND gates. When a DCVSL subcircuit is not processing data, all its output lines will be high. In this case, all 8 NAND gates will produce a 0, and the AND tree will output a 0. But when a DCVSL subcircuit has just finished processing data, all its differential outputs will be pairs of either 0/1 or 1/0. All the NAND gates in the completion generator will then switch from 0 to 1, and the AND tree will in turn switch to 1. This output is then passed through a transmission gate, like the rest of the processed data, and sent to the output register.

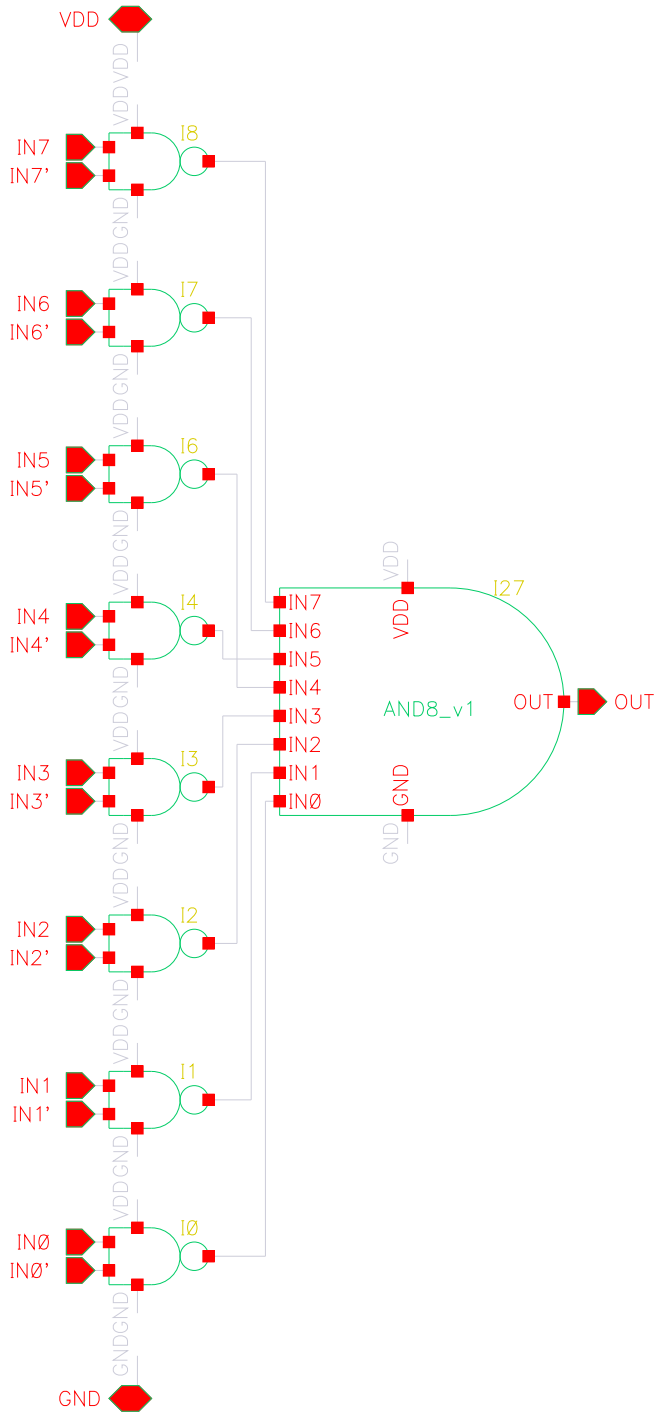



Figure 3.15: Schematic of the completion generator.

## 3.5 The Output Register

Because the ALU utilizes dynamic data paths, internal signals are only valid for a short period of time. When the ALU has finished an instruction and data is ready on the output, we need a way to maintain the data until it is read by an external circuit. The ALU is completely delay-insensitive, which means that per definition, it must be able to retain the data indefinitely. The circuit responsible for this is the output register, see figure 3.16 on the following page. The register is composed mainly of a series of SR latches (8 for data out, 1 for the overflow flag, and 2 for control logic). The inputs and outputs are shown below.

**Inputs:**

- REQ - Incoming request signal. Indicates new data input. Generated by the completion signal generator connected to the currently active DCVSL subcircuit.
- ACK - Acknowledge signal. Indicates data output successfully read. Generated by the external circuitry that communicates with the ALU.
- OF, OF' - Incoming overflow. Differential line. Indicates arithmetic overflow in the sum of an addition. Generated by the adder.
- IN[7-0], IN[7-0]' - Regular data inputs. Differential lines. Generated by the currently active DCVSL subcircuit.

**Outputs:**

- REQOUT - Outgoing request signal. Indicates new data output ready for retrieval by external circuitry.
- OFUT, OFOUT' - Outgoing overflow. Differential line. Indicates arithmetic overflow in the sum of an addition.
- OUT[7-0], OUT[7-0]' - Regular data outputs. Differential lines.

Static AND gates are connected to the S and R inputs of each data latch, coupling the input signal with REQ. For a data latch storing bit x, the S and R inputs are given by the logic expressions below.

$$S_x = IN_x \cdot REQ \tag{3.7}$$

$$R_x = \overline{IN_x} \cdot REQ \tag{3.8}$$

For reference, the truth table for an SR latch is shown in table 3.2 on page 39. As long as REQ is low, both S and R will be low, and the state of the latch will not change. When the ALU has finished an operation and REQ goes high, each latch will either enter the set state ($Q = 1$) if

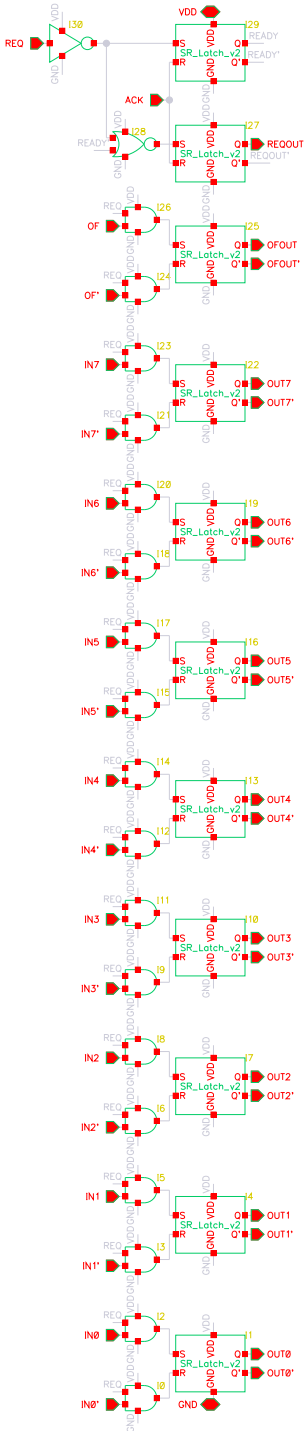



Figure 3.16: Schematic of the output register.

| S | R | $Q_{next}$ | Action |
|---|---|---|---|
| 0 | 0 | Q | hold state |
| 0 | 1 | 0 | reset |
| 1 | 0 | 1 | set |
| 1 | 1 | X | not allowed |

Table 3.2: SR latch truth table.

$S = 1$ and $R = 0$, or enter the reset state ($Q = 0$) if $S = 0$ and $R = 1$. The combination $S = 1, R = 1$ is usually considered not allowed for SR latches, because it would break the logical assertion $Q \neq \overline{Q}$. In this implementation, this would normally not be an issue because $IN_x$ and $\overline{IN_x}$ will always be opposite when REQ goes high. However, if the external circuitry fails to pull REQ low after data has been entered into the output registry, the voltages on the data input lines may begin to drift towards 0, eventually reaching inconsistent logical values, e.g. $IN_x = \overline{IN_x} = 0$. The latches are constructed using CMOS NOR gates, as seen in figure 3.17 on the following page.

The complete layout of the output register is shown in figure 3.18 on page 41.

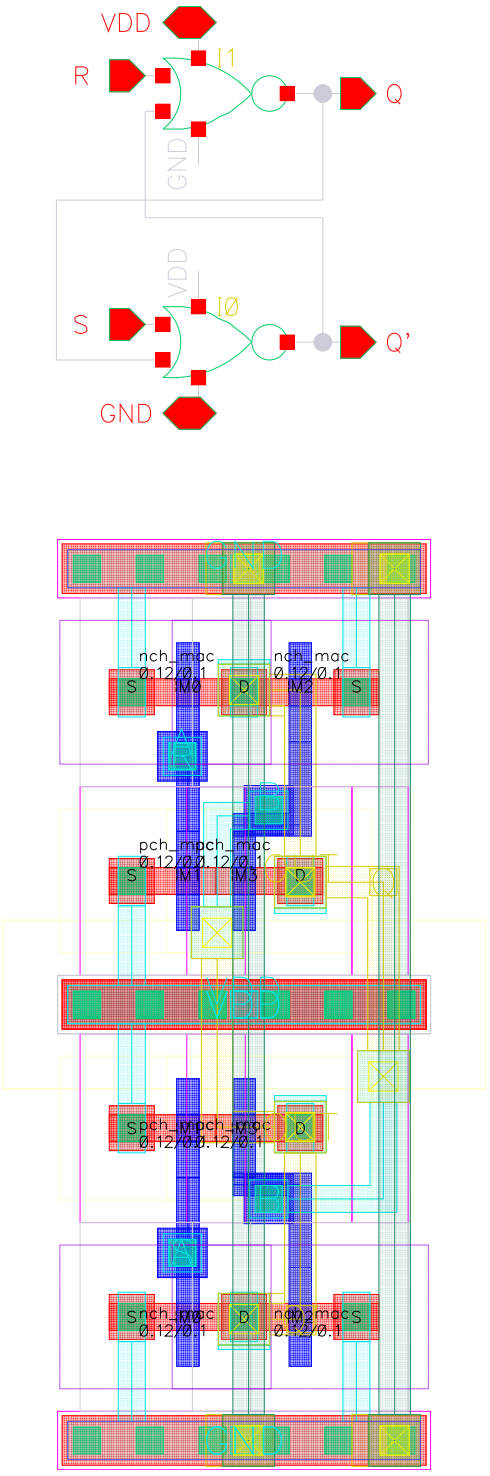



Figure 3.17: Top: Schematic of an SR latch using NOR gates. Bottom: Layout of an SR latch using NOR gates. Height = 4.14$\mu$m, width = 2.15$\mu$m.

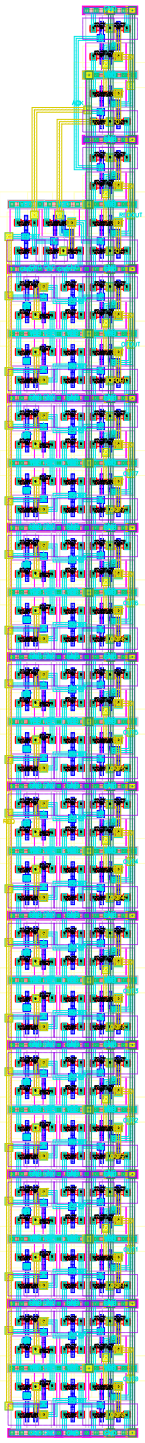


Figure 3.18: Layout of the output register. Height = 42.94$\mu$m, width = 4.39$\mu$m.

## Control Logic

In addition to simply storing the output data, the output register keeps track of the current operational state of the ALU. This is accomplished by the control logic in the top of the schematic. A cutout of this section is shown in 3.19 on the facing page.

In order for the output register to be able to receive data, two things must be checked. First, we have to be sure that data from a previous operation is not currently occupying the register, waiting to be read. If this is the case and we were to start a new operation, the data stored in the register would be overwritten without ever being read, which would naturally result in data loss. Second, the ACK input must not be high when the REQ input goes high. If ACK is high, it means that the previous read operation is not yet finished.

A simple state diagram illustrating the operation of the control logic is shown in figure 3.20 on page 44. Only transitions that cause the current state to change, are shown in the figure. Circular transitions, i.e. transitions that lead from one state back to the same state, have been omitted for readability.

The state in the middle of the diagram is the idle state. Here, the output register is empty, ready for input, and both REQ and ACK are low. Now let's say that REQ goes high. This causes a transition to the state on the left. This is the storage state, where input data is stored in the register, and REQOUT is set high to indicate available output data. The external circuitry that communicates with the ALU can choose to set REQ low at any time during this state. Setting REQ low will not affect the data, because the latches can only change values when REQ is high. However, if REQ stays high for too long, we run the risk of corrupting the data because of leakage currents in the dynamic DCVSL cells. If ACK is now set high, indicating that data is read out of the output register to the external circuitry, we move to the state on the right. This is the lock state. At this point in time, the register will not be able to process new data, and REQOUT will not go high. Before starting each operation, the ALU needs to be in the idle state for a certain amount of time in order to precharge the DCVSL cells. If REQ goes high while in the lock state, the storage latches will in fact be transparent and accept whatever data is on the input lines, but there is no guarantee that the data is valid, because the DCVSL cells may not yet have been precharged. When both REQ and ACK go low, the register returns to the idle state, ready for the next operation. If for some reason ACK goes high while in the idle state, the register would transition back to the lock state, and no data will be accepted before the register returns to the idle state.

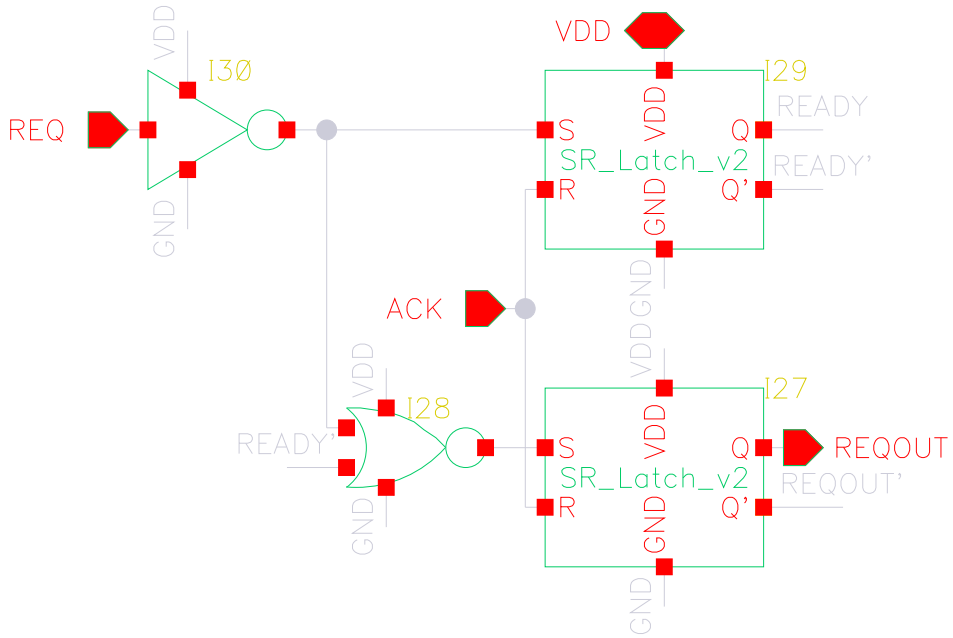



Figure 3.19: Cutout of the control logic in the schematic of the output register.

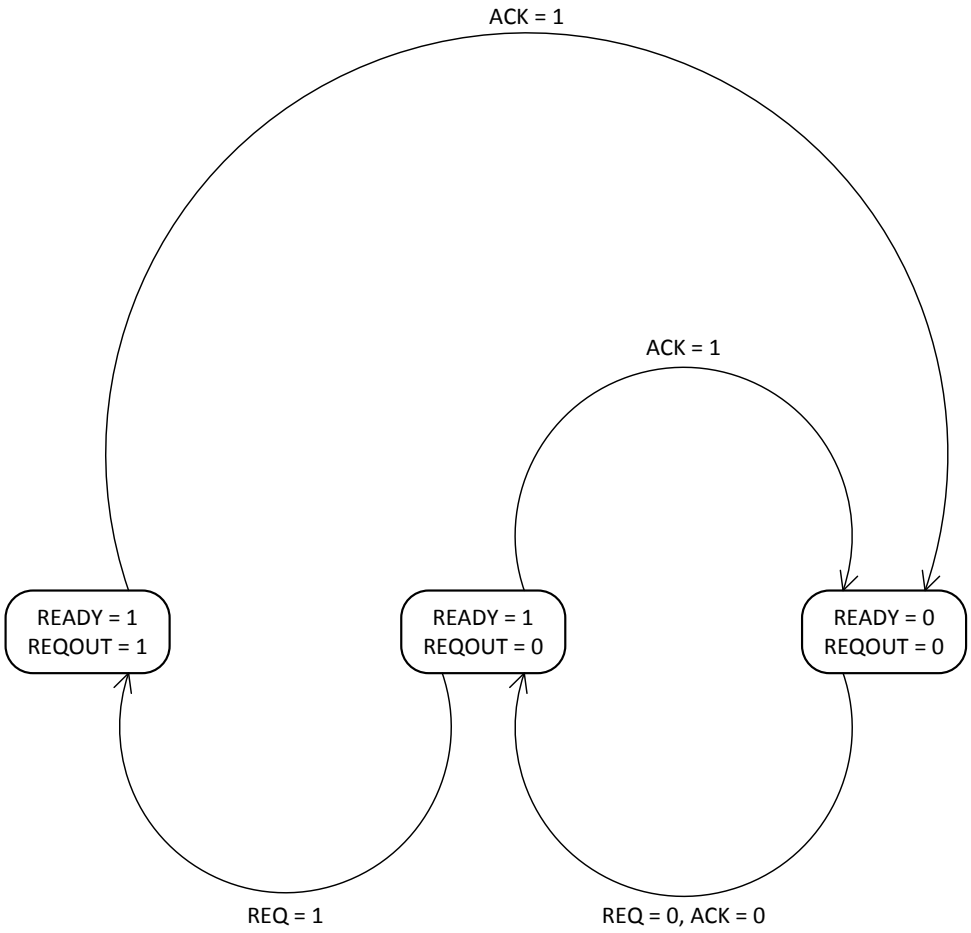



Figure 3.20: State diagram describing the output register control logic.

## 3.6 Connecting the ALU to Padframe

The physical chip package used for this ALU is an 84-pin ceramic JLCC package. For details on the chip package and padframe, as well as full ALU layout, see A on page 75.

In figure 3.21 on the next page we see a top level view of the entire ALU schematic. REQ and S[3-0] enter the instruction decoder in the top left of the schematic. Input vectors A and B are below, on the left-hand side. The output register and outputs to pads are in the upper right-hand corner.

Ordering custom microchips is a costly endeavour, especially in small quantities. Because of limited funds, the ALU in this thesis shares the die area and padframe with another circuit belonging to someone else. As such, the number of available input and output pads is limited. The ALU uses differential data paths, requiring two physical wires for each data bit. However, there are not enough pads available for differential inputs on the chip. Therefore, inverters are used at all data inputs, to create the inverse of all data bits. These inverters can be seen below each input vector in the top schematic.

The ALU output register does not drive the chip output pads directly. Instead, powerful digital buffers are inserted between the output register and the pads, in order to provide necessary current driving capacity. Each buffer consists of two static CMOS inverters connected in series, with transistor width/length ratios $\frac{W}{L} = \frac{5\mu m}{0.3\mu m}$. The transistors in the first inverter have 5 fingers[6], and the transistors in the second inverter have 10 fingers.

Four testpoints are inserted at the following critical locations in the ALU, to assist in testing:

- The non-inverted request signal from the instruction decoder to the adder
- The non-inverted sum bit 7 from the adder, before the transmission gate
- The non-inverted input bit 7 entering the output register
- The request signal entering the output register

These four signals are connected to analog voltage followers (design by Philipp Häfliger, Ph.D.). The outputs of these voltage followers are connected directly to pads on the chip. During ALU operation, these pads can be probed with an oscilloscope to get a direct view of the internal signal transitions. The voltage followers, as well as the previously mentioned digital pad drivers, can be seen in figure 3.22 on page 47. The resistors connected in series after each follower/buffer are a requirement of the process design rules by TSMC.

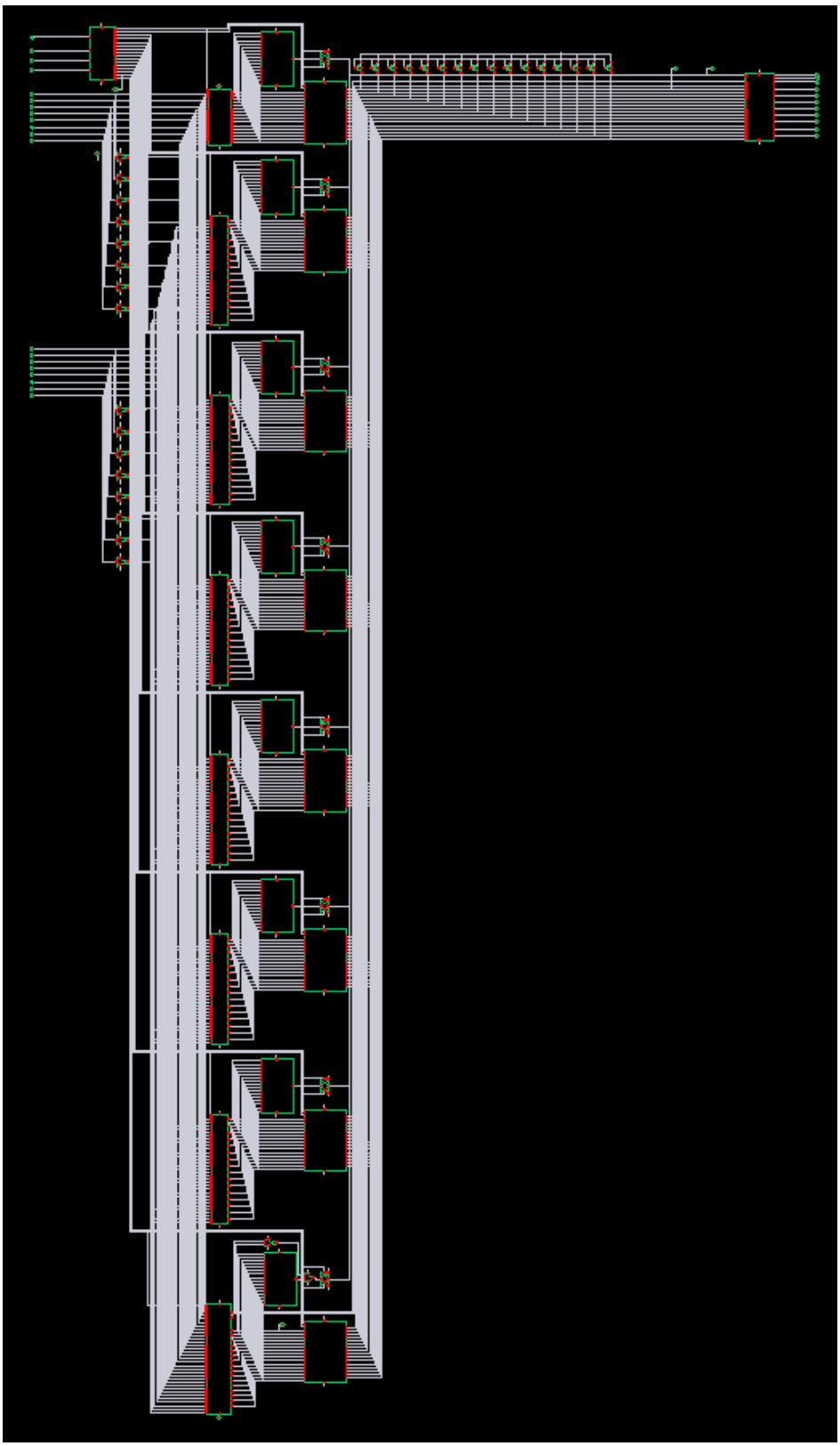


Figure 3.21: Top level view of the ALU schematic.

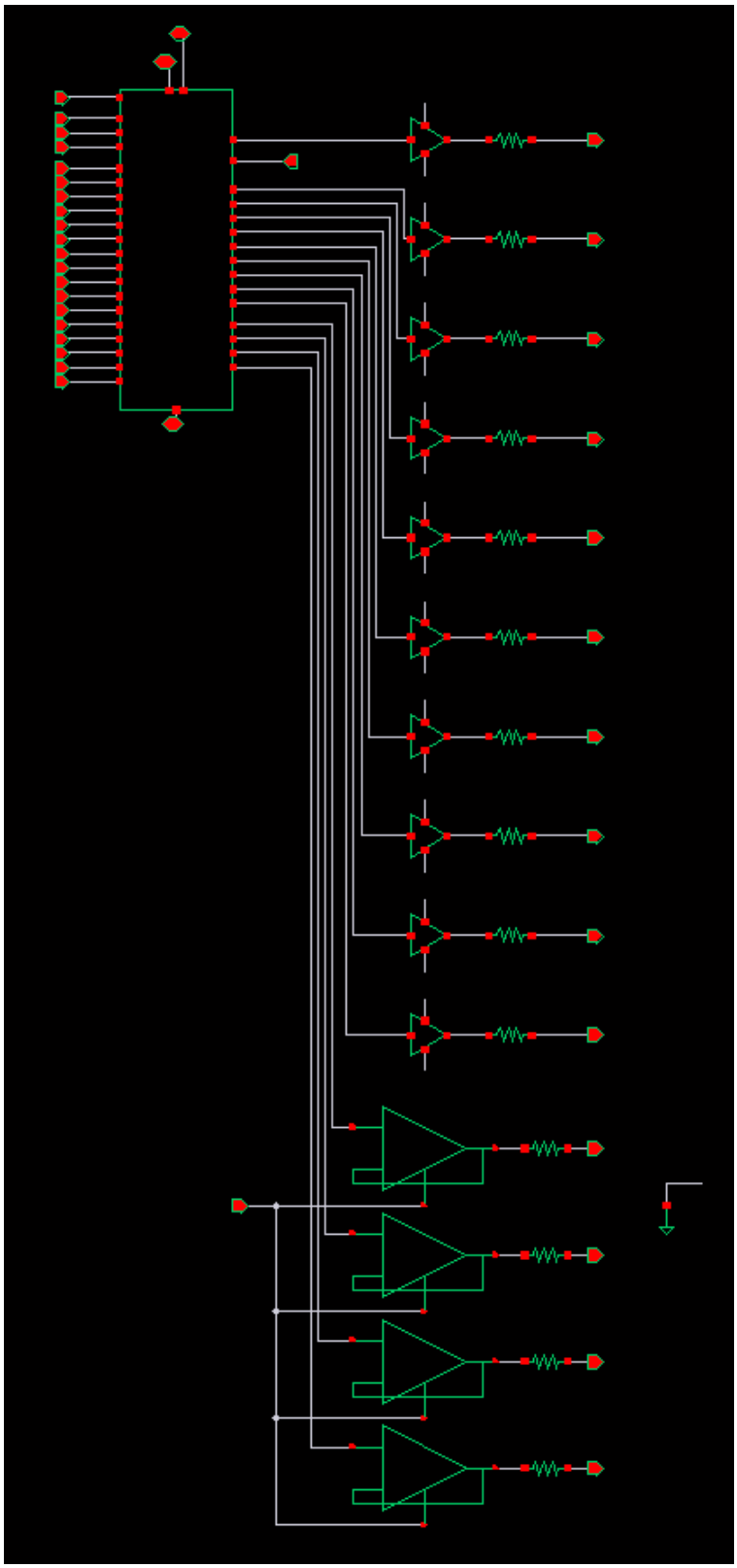


Figure 3.22: Schematic view of output-to-pad digital buffers and analog voltage followers for testpoints.

Compared to the small transistors of the ALU, the voltage followers and pad drivers consume quite a large amount of power. To be able to distinguish ALU power consumption from total chip power consumption during testing, we use two $V_{DD}$ references, $PV_{DD}$ and $CV_{DD}$. $CV_{DD}$ powers only the core, i.e. the ALU itself. $PV_{DD}$ powers everything else, such as the output buffers and voltage followers. On the die, a parallel plate decoupling capacitor is connected to each of these voltage references. The capacitors measure 50$\mu$m by 15$\mu$m, which gives a capacitance of approx. 4.6pF.

# Chapter 4

# Testing the ALU

Post-production testing of the ALU chip was performed by designing a custom PCB to connect the chip to an Atmel AT91SAM7S256 microcontroller on an Olimex SAM7-H256 header board. A testbench was programmed into the microcontroller using the C programming language. The microcontroller was used to supply the ALU with input vectors, and record the responses. The microcontroller itself was connected to a computer via USB, and the testbench program was controlled from the computer by issuing commands over the microcontroller's UART interface. Using the same interface, the microcontroller could transmit the test results back to the computer, displaying the results live on the computer screen.

The Thurlby Thandar PL330DP DC voltage source was used to provide varying supply voltages for the ALU. Signals on the PCB were probed using an Agilent Technologies DSO6034A oscilloscope. Current consumption was measured with an Agilent Technologies 34401A digital multimeter.

## 4.1 PCB Design

The PCB was designed using CADSTAR Design Editor version 13.0 and PREditor XR version 1.13.DB.33, by Zuken Ltd. The PWB was manufactured by an external supplier. SMD components were placed manually on the board, and soldered in place using a vapor phase reflow oven[15]. Through-hole mounted pin rows were soldered manually. The finished PCB is shown in figure 4.1 on the next page.

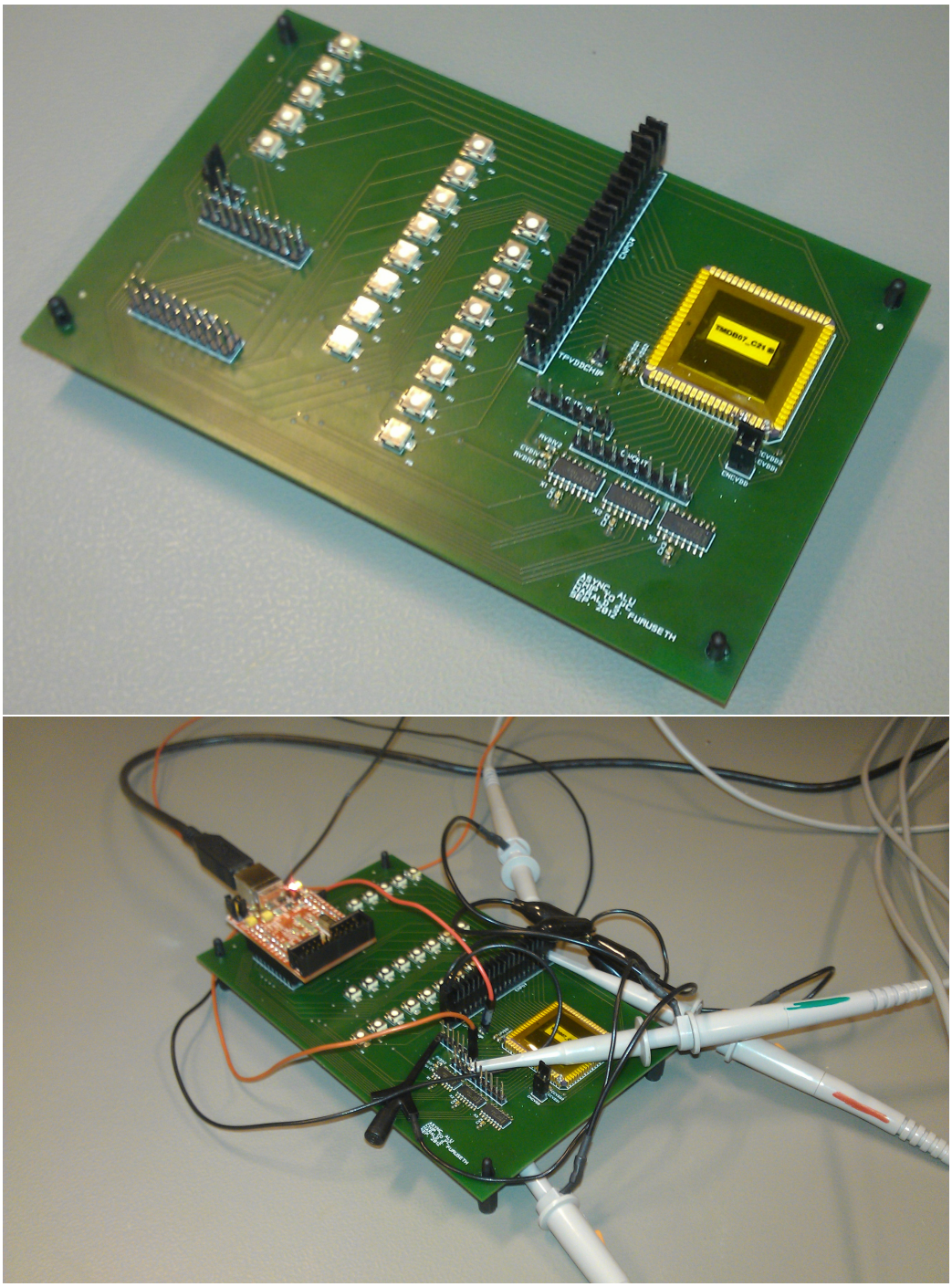


Figure 4.1: Top: Photograph of the finished PCB. Bottom: Photograph of the finished PCB with microcontroller header board and lab equipment attached, during a testing run.

### 4.1.1 Schematic

A top sheet view of the PCB schematic is shown in figure 4.2 on the following page. Because of the sheer number of signals, signal names have been omitted for visibility. For full page views of the entire schematic including subsheets, see appendix B on page 81.

The microcontroller header board (see appendix C on page 89 for details) is mounted on the two parallel pin rows CNEXT1 and CNEXT2, at the bottom left in the schematic. The microcontroller's I/O banks operate at 3.3V, whereas the ALU operates with voltages of 1.2V and below. Signals travelling from the microcontroller to the ALU chip are passed through 1kΩ potentiometers (design sheets STEPDOWN_CONTR, STEPDOWN_A, and STEPDOWN_B) before entering the pads on the chip. The potentiometers allow manual adjustment of the input voltage levels.

The voltage level on the return signals from the ALU chip to the microcontroller must also be adjusted, but this time it's the other way around. The output signals from the chip are fed into an array of high speed voltage comparators (design sheet STEPUP), which in turn drive the 3.3V microcontroller inputs. The chosen comparators are called MAX964, produced by Maxim Integrated. The MAX964 is a quad comparator IC, featuring a built-in 50mV hysteresis. The hysteresis reduces the susceptibility to signal noise on the comparator inputs, and helps to ensure glitch-free transitions between voltage levels[22].

The microcontroller header board has an on-board power regulator that draws power from the USB interface. The regulator provides supply voltage for both the header board itself and the MAX964 voltage comparators on the PCB. To provide a steady supply voltage during signal transitions, a 100nF decoupling capacitor is connected to the $V_{DD}$ input of each comparator IC[24].

$PV_{DD}$ and $CV_{DD}$ for the ALU are supplied by an external voltage source. On the PCB, these voltage references are connected to single-pin test points, to allow easy connectivity with the external equipment. Both $PV_{DD}$ and $CV_{DD}$ have decoupling capacitors to stabilize the voltage. Each power pin is connected to a 100nF and a 1nF capacitor in parallel. The 100nf capacitors act as charge reservoirs to minimize voltage loss during current transients, and the 1nF capacitors minimize potential noise[7].

The comparators require a reference voltage to act as a switching threshold. In our case, each comparator output should switch value when the respective ALU output voltage crosses $\frac{V_{DD}}{2}$. For example, if the ALU $V_{DD}$ is 1.2V, the switching threshold would be 0.6V. On the PCB, this switching threshold is provided by a voltage divider (design sheet STEPUP), composed of two 1kΩ resistors connected in series between ALU $PV_{DD}$ and ground. A 100nF capacitor is placed in parallel with the resistor to ground, stabilizing the reference voltage.

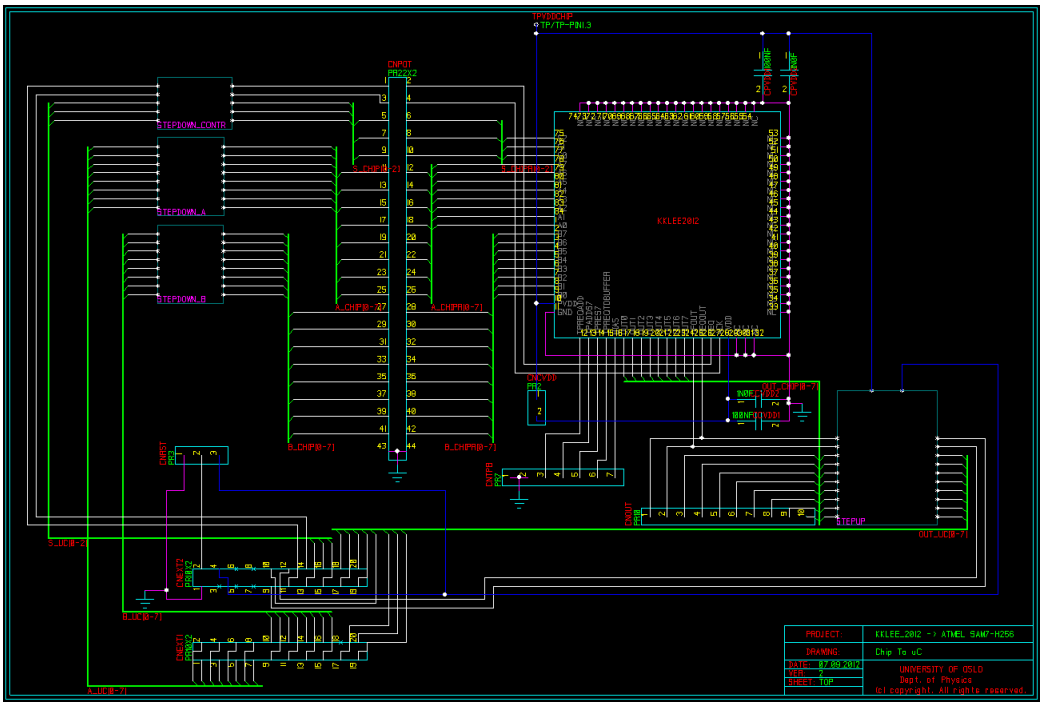



Figure 4.2: Top level sheet of the PCB schematic.

### 4.1.2 Layout

The layout of the PCB can be seen in figure 4.1.2 on the next page. For a full page view, see appendix B on page 81. There are 2 electrical layers, where a ground plane fills most of the bottom layer[8]. In the figure, the top layer is depicted in red, and the bottom layer is depicted in green. The PCB substrate is standard FR-4.

The board is designed for ease of testing, not high performance. As such, the board is quite large for its relatively small number of components, and there is a lot of unused space. Pin rows and jumpers are used to provide access points for the test equipment and voltage sources. The ALU input tracks, as well as $CV_{DD}$, can be physically disconnected if necessary, by removing the appropriate jumper.

The physical dimensions of the board are 160mm x 100mm. Nominal track widths are 1mm for power/ground, and 0.30mm for signals. Necked track width for power/ground is 0.25mm, for entry/exit to small soldering pads. In the design rules during layout, the copper-to-copper minimum distance was set to 0.40mm.

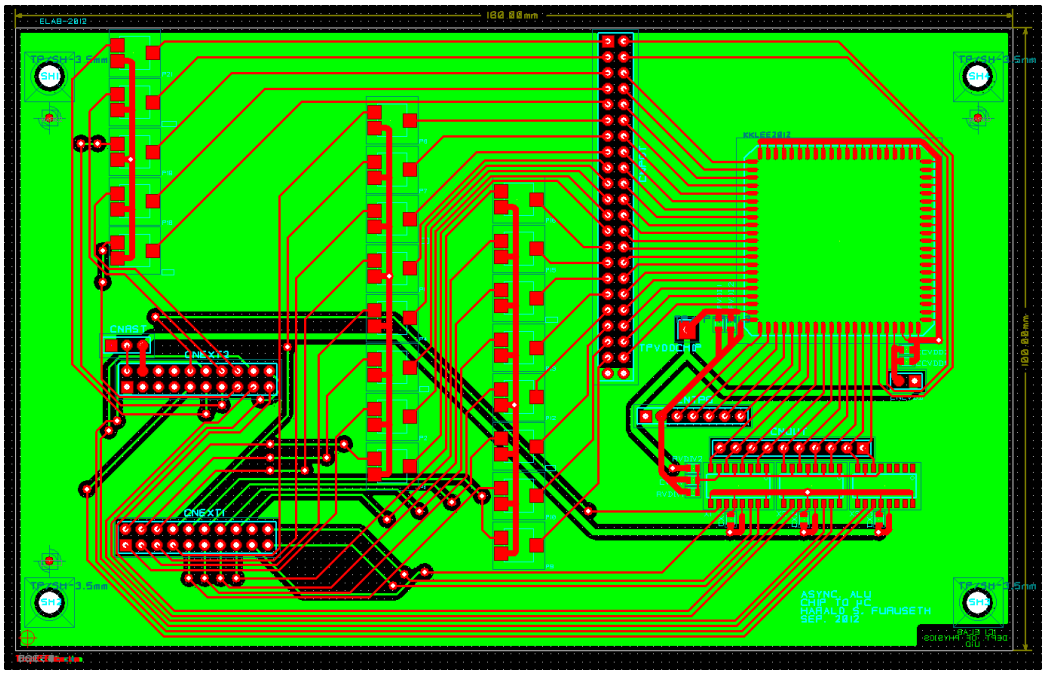


Figure 4.3: PCB layout.

## 4.2 Microcontroller Interaction

The function of the microcontroller is to provide the ALU chip with input data, record the generated outputs, and verify correct operation. In addition, visual feedback is presented to the user during run time. This is achieved by programming the microcontroller with testing procedures written in the C programming language.

Separate C methods are written to test each ALU function. These methods are implemented into a pre-existing microcontroller firmware, previously written by PhD candidate Håkon Hjortland at the UiO Department of Informatics. Hjortland's firmware already incorporates the computer communication over USB, allowing this author to concentrate on implementing the ALU test methods. For more information on the underlying firmware, please see http://haakoh.at.ifi.uio.no/sam7. All of the C source code used in the microcontroller was compiled under Linux Red Hat version 4.1.2-52, using GCC version 4.1.2 20080704.

The complete test method source code with relevant header file is listed in appendix D on page 93. The program flow of each test method is as follows:

- Initialize microcontroller I/O
- Set the ALU input S (Select) to the appropriate value. 000 for NOT, 001 for NAND, 010 for AND etc.

- Set ACK high and then low again to reset the ALU output register, in case the chip was power cycled
- Loop through ALU input A values
  - Loop through ALU input B values
    - Send current A and B values and REQ
    - Wait for REQOUT from the ALU. If REQOUT is received:
      - Read ALU output
      - Set REQ low
      - Set ACK high, wait until REQOUT goes low, and set ACK low
      - Check validity of output data, and report any errors to user
      - .
    - If REQOUT is not received:
      - Set REQ low
      - Increment wait timer and reset A and B to 0
      - Notify user
    - Wait a little while before next data set. Specified by the wait timer
  - Report number of successful runs for current A value to user

The test program is self-timing, and determines the maximum speed of the ALU during run time. When the program first executes, the wait timer is 0, which means the program will attempt to send a new data set as soon as the previous data set has been processed. In the event that the ALU has not had enough time after a data set to get ready for the next set (precharge time), REQOUT will not go high when REQ is sent to the ALU. In this case, the test program will abort the current data set, and try again from the start with a longer wait timer. This process may repeat many times, continually lowering the number of attempted data sets per second, until the maximum operating speed has been found.

# Chapter 5

# Results

By running the microcontroller test program repeatedly and varying the ALU voltage, the timing results shown in tables 5.1-5.5 were obtained. The voltages shown in the first columns indicate both ALU $V_{DD}$ and input voltage range, fine-tuned by adjusting the potentiometers. The columns marked *IN->OUT* indicate the time between a REQ was sent to the ALU, and the ALU responded with a REQOUT. The columns marked *Lap time* indicate the time between each data set. The final columns, marked *Errors* indicate whether or not bit errors were recorded during that particular test run. The timing intervals were measured using the oscilloscope, by probing the REQ and REQOUT signal tracks. Note that in table 5.3, no timing data is presented for the lowest voltage ranges. For these testing runs, denoted by *X* in the table, the ALU did not respond with a REQOUT when REQ was sent from the microcontroller.

The static current consumption of the chip for varying supply voltages is shown in table 5.6 on page 58. Pad- and core static current consumption plotted together is shown in figure 5.1 on page 59.

| | **NOT** | | |
|---|---|---|---|
| | IN->OUT | Lap time | Errors |
| **1.20V** | 26.0ns | 35.6$\mu$s | Yes |
| **1.15V** | 23.6ns | 66.0$\mu$s | Yes |
| **1.10V** | 25.4ns | 62.4$\mu$s | Yes |
| **1.05V** | 27.2ns | 63.0$\mu$s | Yes |
| **1.00V** | 30.6ns | 60.0$\mu$s | Yes |
| **0.95V** | 33.6ns | 55.0$\mu$s | Yes |
| **0.90V** | 36.4ns | 52.0$\mu$s | Yes |
| **0.85V** | 43.2ns | 42.4$\mu$s | Yes |
| **0.80V** | 53.2ns | 34.0$\mu$s | Yes |
| **0.75V** | 60.8ns | 34.2$\mu$s | Yes |
| **0.70V** | 78.0ns | 34.0$\mu$s | Yes |
| **0.65V** | 103.0ns | 34.4$\mu$s | Yes |
| **0.60V** | 155.0ns | 34.4$\mu$s | Yes |
| **0.55V** | 232.0ns | 34.4$\mu$s | Yes |
| **0.50V** | 416.0ns | 34.4$\mu$s | Yes |
| **0.45V** | 888.0ns | 34.6$\mu$s | Yes |
| **0.40V** | 2.42$\mu$s | 36.2$\mu$s | Yes |

Table 5.1: ALU chip timing measurements for the NOT function.

| | **AND** | | | **NAND** | | |
|---|---|---|---|---|---|---|
| | IN->OUT | Lap time | Errors | IN->OUT | Lap time | Errors |
| **1.20V** | 23.6ns | 123.0$\mu$s | No | 22.2ns | 119.0$\mu$s | No |
| **1.15V** | 25.0ns | 116.0$\mu$s | No | 23.2ns | 114.0$\mu$s | No |
| **1.10V** | 26.2ns | 116.0$\mu$s | No | 24.4ns | 112.0$\mu$s | No |
| **1.05V** | 28.6ns | 119.0$\mu$s | Yes | 27.4ns | 113.0$\mu$s | No |
| **1.00V** | 30.8ns | 121.0$\mu$s | Yes | 29.8ns | 112.0$\mu$s | Yes |
| **0.95V** | 35.2ns | 119.0$\mu$s | Yes | 32.4ns | 109.0$\mu$s | Yes |
| **0.90V** | 39.2ns | 128.0$\mu$s | Yes | 37.2ns | 121.0$\mu$s | Yes |
| **0.85V** | 44.8ns | 120.0$\mu$s | Yes | 44.0ns | 115.0$\mu$s | Yes |
| **0.80V** | 52.0ns | 118.0$\mu$s | Yes | 50.8ns | 110.0$\mu$s | Yes |
| **0.75V** | 64.8ns | 112.0$\mu$s | Yes | 62.8ns | 110.0$\mu$s | Yes |
| **0.70V** | 82.0ns | 112.1$\mu$s | Yes | 79.0ns | 111.0$\mu$s | Yes |
| **0.65V** | 113.0ns | 108.0$\mu$s | Yes | 108.0ns | 112.1$\mu$s | Yes |
| **0.60V** | 170.0ns | 102.0$\mu$s | Yes | 158.0ns | 109.0$\mu$s | Yes |
| **0.55V** | 274.0ns | 100.0$\mu$s | Yes | 240.0ns | 107.0$\mu$s | Yes |
| **0.50V** | 512.0ns | 94.0$\mu$s | Yes | 452.0ns | 102.0$\mu$s | Yes |
| **0.45V** | 1.1$\mu$s | 86.0$\mu$s | Yes | 956.0ns | 96.0$\mu$s | Yes |
| **0.40V** | 2.96$\mu$s | 80.0$\mu$s | Yes | 2.5$\mu$s | 89.0$\mu$s | Yes |

Table 5.2: ALU chip timing measurements for the AND and NAND functions.

| | **OR** | | | **NOR** | | |
|---|---|---|---|---|---|---|
| | IN->OUT | Lap time | Errors | IN->OUT | Lap time | Errors |
| **1.20V** | 24.0ns | 116.0$\mu$s | No | 22.6ns | 98.0$\mu$s | Yes |
| **1.15V** | 25.4ns | 112.0$\mu$s | Yes | 23.4ns | 93.6$\mu$s | Yes |
| **1.10V** | 27.4ns | 110.0$\mu$s | Yes | 25.0ns | 91.2$\mu$s | Yes |
| **1.05V** | 29.0ns | 110.0$\mu$s | Yes | 27.4ns | 91.0$\mu$s | Yes |
| **1.00V** | 31.4ns | 110.0$\mu$s | Yes | 29.4ns | 90.0$\mu$s | Yes |
| **0.95V** | 35.6ns | 109.0$\mu$s | Yes | 33.2ns | 89.0$\mu$s | Yes |
| **0.90V** | 39.6ns | 117.0$\mu$s | Yes | 37.6ns | 96.0$\mu$s | Yes |
| **0.85V** | 46.4ns | 111.0$\mu$s | Yes | 42.0ns | 93.0$\mu$s | Yes |
| **0.80V** | 55.6ns | 110.0$\mu$s | Yes | 50.8ns | 94.0$\mu$s | Yes |
| **0.75V** | 66.8ns | 111.0$\mu$s | Yes | 60.0ns | 97.0$\mu$s | Yes |
| **0.70V** | 90.0ns | 110.1$\mu$s | Yes | 78.0ns | 101.0$\mu$s | Yes |
| **0.65V** | X | X | X | 110.0ns | 105.1$\mu$s | Yes |
| **0.60V** | X | X | X | X | X | X |
| **0.55V** | X | X | X | X | X | X |
| **0.50V** | X | X | X | X | X | X |
| **0.45V** | X | X | X | X | X | X |
| **0.40V** | X | X | X | X | X | X |

Table 5.3: ALU chip timing measurements for the OR and NOR functions.

| | **XOR** | | | **XNOR** | | |
|---|---|---|---|---|---|---|
| | IN->OUT | Lap time | Errors | IN->OUT | Lap time | Errors |
| **1.20V** | 23.6ns | 40.6$\mu$s | No | 24.6ns | 31.6$\mu$s | Yes |
| **1.15V** | 24.6ns | 38.2$\mu$s | Yes | 26.0ns | 30.8$\mu$s | Yes |
| **1.10V** | 26.8ns | 37.0$\mu$s | Yes | 27.6ns | 27.0$\mu$s | Yes |
| **1.05V** | 29.0ns | 36.4$\mu$s | Yes | 29.0ns | 20.6$\mu$s | Yes |
| **1.00V** | 31.2ns | 34.2$\mu$s | Yes | 32.2ns | 15.0$\mu$s | Yes |
| **0.95V** | 34.8ns | 29.2$\mu$s | Yes | 36.8ns | 1.9$\mu$s | Yes |
| **0.90V** | 39.2ns | 26.0$\mu$s | Yes | 44.4ns | 6.5$\mu$s | Yes |
| **0.85V** | 45.6ns | 18.4$\mu$s | Yes | 46.0ns | 1.9$\mu$s | Yes |
| **0.80V** | 54.0ns | 8.6$\mu$s | Yes | 56.4ns | 1.9$\mu$s | Yes |
| **0.75V** | 66.4ns | 3.6$\mu$s | Yes | 72.8ns | 2.2$\mu$s | Yes |
| **0.70V** | 85.0ns | 2.1$\mu$s | Yes | 86.0ns | 2.2$\mu$s | Yes |
| **0.65V** | 115.0ns | 2.1$\mu$s | Yes | 119.0ns | 2.2$\mu$s | Yes |
| **0.60V** | 171.0ns | 2.1$\mu$s | Yes | 174.0ns | 2.2$\mu$s | Yes |
| **0.55V** | 264.0ns | 2.1$\mu$s | Yes | 264.0ns | 2.2$\mu$s | Yes |
| **0.50V** | 484.0ns | 2.4$\mu$s | Yes | 476.0ns | 2.5$\mu$s | Yes |
| **0.45V** | 1.2$\mu$s | 3.0$\mu$s | Yes | 1.1$\mu$s | 3.1$\mu$s | Yes |
| **0.40V** | 3.1$\mu$s | 5.0$\mu$s | Yes | 2.7$\mu$s | 4.8$\mu$s | Yes |

Table 5.4: ALU chip timing measurements for the XOR and XNOR functions.

| | **ADD** | | |
|---|---|---|---|
| | IN->OUT | Lap time | Errors |
| **1.20V** | 35.6ns | 96.0$\mu$s | Yes |
| **1.15V** | 38.0ns | 100.0$\mu$s | Yes |
| **1.10V** | 41.6ns | 98.0$\mu$s | Yes |
| **1.05V** | 45.0ns | 98.0$\mu$s | Yes |
| **1.00V** | 50.0ns | 100.0$\mu$s | Yes |
| **0.95V** | 56.4ns | 98.0$\mu$s | Yes |
| **0.90V** | 62.4ns | 100.1$\mu$s | Yes |
| **0.85V** | 74.0ns | 100.0$\mu$s | Yes |
| **0.80V** | 88.0ns | 100.0$\mu$s | Yes |
| **0.75V** | 107.0ns | 100.0$\mu$s | Yes |
| **0.70V** | 140.0ns | 100.0$\mu$s | Yes |
| **0.65V** | 190.0ns | 100.0$\mu$s | Yes |
| **0.60V** | 290.0ns | 100.0$\mu$s | Yes |
| **0.55V** | 464.0ns | 100.0$\mu$s | Yes |
| **0.50V** | 864.0ns | 100.0$\mu$s | Yes |
| **0.45V** | 2.0$\mu$s | 100.0$\mu$s | Yes |
| **0.40V** | 5.2$\mu$s | 100.4$\mu$s | Yes |

Table 5.5: ALU chip timing measurements for the ADD function.

| | Core (CVDD) | Pads (PVDD) |
|---|---|---|
| **1.20V** | 47.0$\mu$A | 696.9$\mu$A |
| **1.15V** | 41.8$\mu$A | 665.5$\mu$A |
| **1.10V** | 37.2$\mu$A | 620.9$\mu$A |
| **1.05V** | 32.8$\mu$A | 569.3$\mu$A |
| **1.00V** | 28.8$\mu$A | 480.6$\mu$A |
| **0.95V** | 24.7$\mu$A | 462.3$\mu$A |
| **0.90V** | 20.8$\mu$A | 452.8$\mu$A |
| **0.85V** | 17.3$\mu$A | 409.8$\mu$A |
| **0.80V** | 14.1$\mu$A | 386.0$\mu$A |
| **0.75V** | 11.5$\mu$A | 367.6$\mu$A |
| **0.70V** | 8.7$\mu$A | 332.5$\mu$A |
| **0.65V** | 6.4$\mu$A | 312.6$\mu$A |
| **0.60V** | 4.4$\mu$A | 279.8$\mu$A |
| **0.55V** | 2.7$\mu$A | 253.6$\mu$A |
| **0.50V** | 1.6$\mu$A | 229.3$\mu$A |
| **0.45V** | 0.8$\mu$A | 180.8$\mu$A |
| **0.40V** | 0.4$\mu$A | 73.1$\mu$A |

Table 5.6: ALU chip static current consumption.

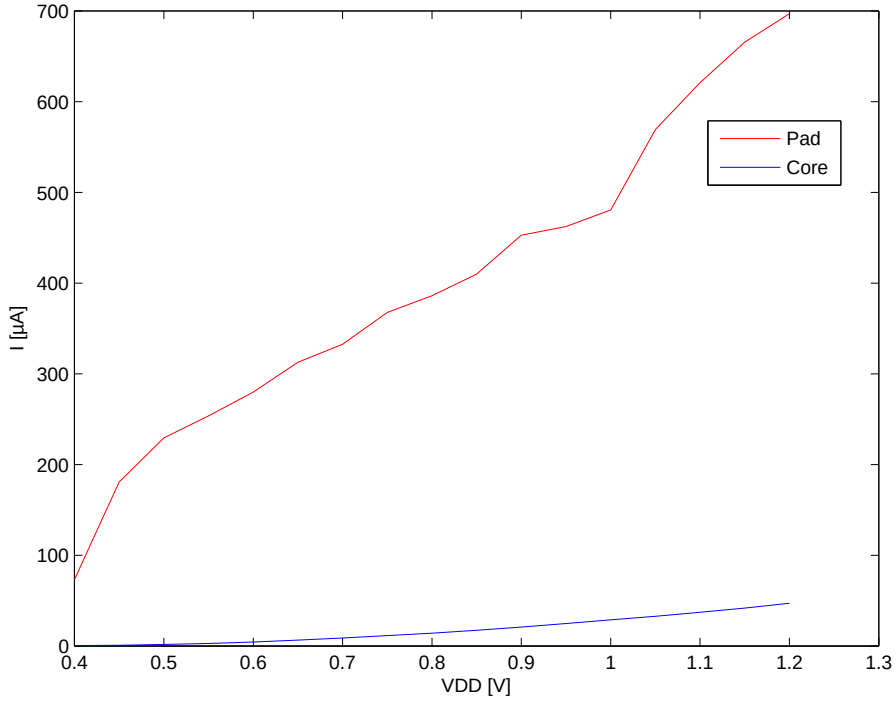



Figure 5.1: Plot of pad- and core static current consumption vs. supply voltage.

In figure 5.2 on the facing page we see a screenshot of the oscilloscope during an XOR operation, showing the REQ and ACK signals.

- Trace 1 (yellow): The REQIN signal from the microcontroller to the ALU.
- Trace 2 (green) : The REQOUT signal from the ALU to the comparator.
- Trace 3 (purple): The REQOUT signal from the comparator the microcontroller.
- Trace 4 (red) : The ACK signal from the microcontroller to the ALU.

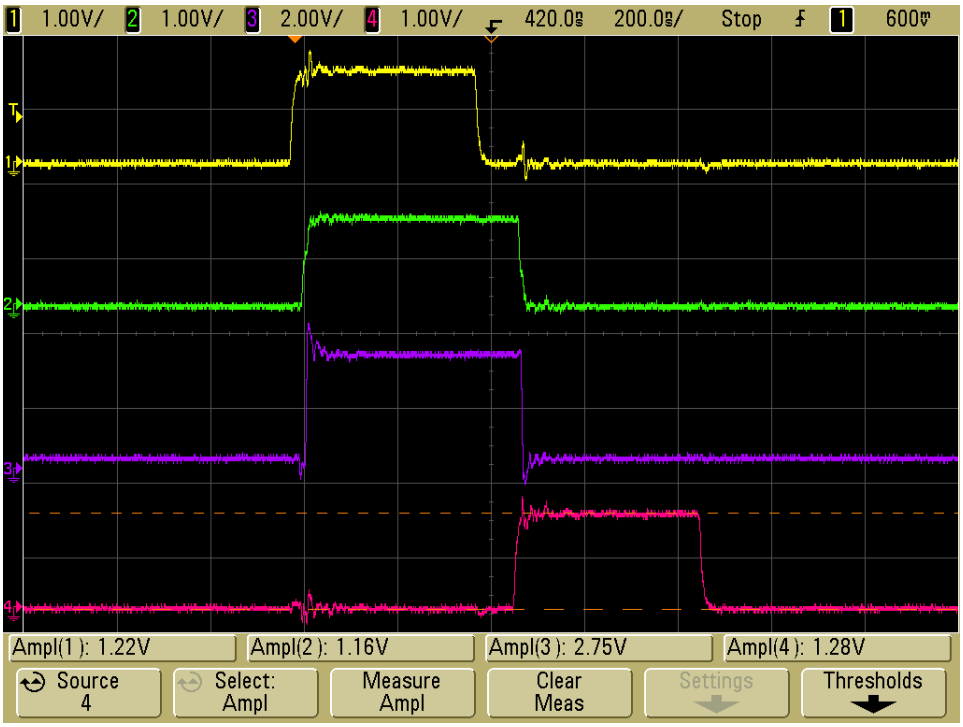



Figure 5.2: Oscilloscope screenshot showing the REQ and ACK signals during an XOR operation at 1.2V.

In figure 5.3 on the next page we see a screenshot of the oscilloscope during the very first XOR data set, where $A = B = 0$, showing the propagation of the REQ signal.

- Trace 1 (yellow): The REQIN signal from the microcontroller to the ALU.
- Trace 2 (green) : The internal REQ signal in the ALU, between the XOR cells and the output register.
- Trace 3 (purple): The REQOUT signal from the ALU to the comparator.
- Trace 4 (red) : The REQOUT signal from the comparator the microcontroller.

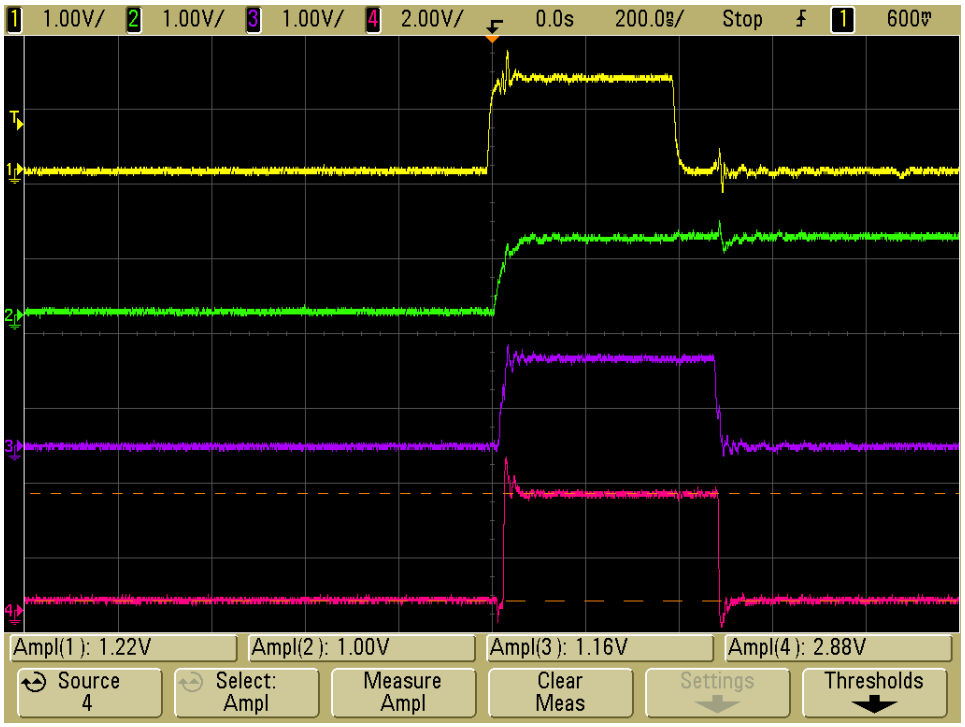



Figure 5.3: Oscilloscope screenshot showing the propagation of Request during the very first XOR data set at 1.2V.

In figure 5.4 on the facing page we see a screenshot of the oscilloscope during the very last XOR data set, where $A = B = 255$, showing the propagation of the REQ signal.

- Trace 1 (yellow): The REQIN signal from the microcontroller to the ALU.
- Trace 2 (green) : The internal REQ signal in the ALU, between the XOR cells and the output register.
- Trace 3 (purple): The REQOUT signal from the ALU to the comparator.
- Trace 4 (red) : The REQOUT signal from the comparator the microcontroller.

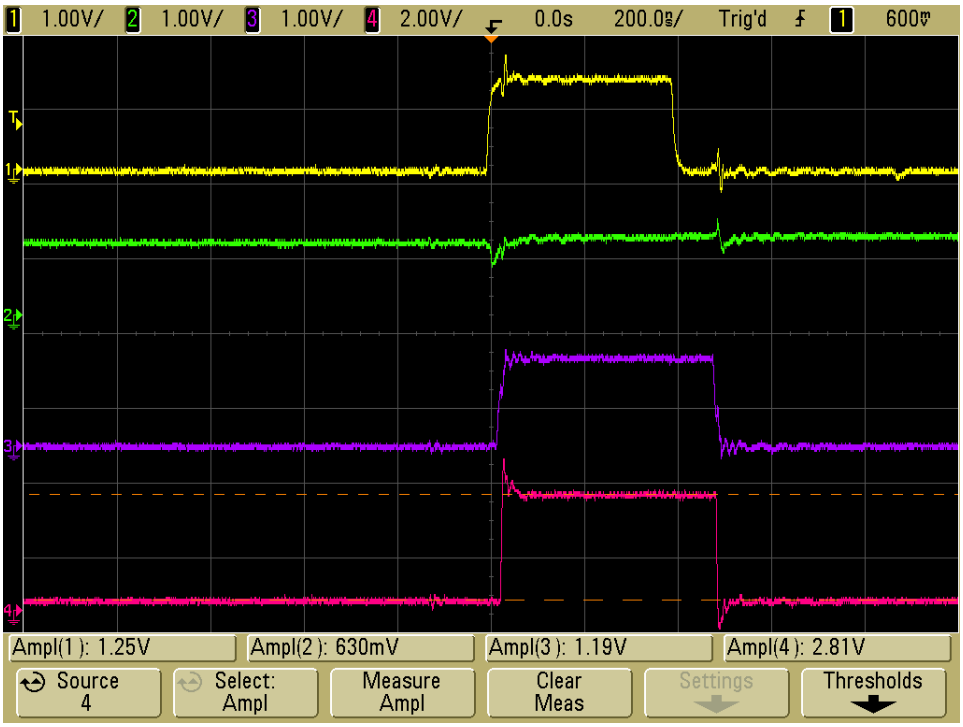



Figure 5.4: Oscilloscope screenshot showing the propagation of Request during the very last XOR data set at 1.2V.

In figure 5.5 on the next page we see a screenshot of the oscilloscope during an OR operation, where $A = 0$ and $B = 16$, showing an error in ALU output bit 3.

- Trace 1 (yellow): The REQIN signal from the microcontroller to the ALU.
- Trace 2 (green) : The OUT3 signal from the ALU to the comparator.
- Trace 3 (purple): The OUT3 signal from the comparator to the microcontroller.
- Trace 4 (red) : The ACK signal from the microcontroller to the ALU.

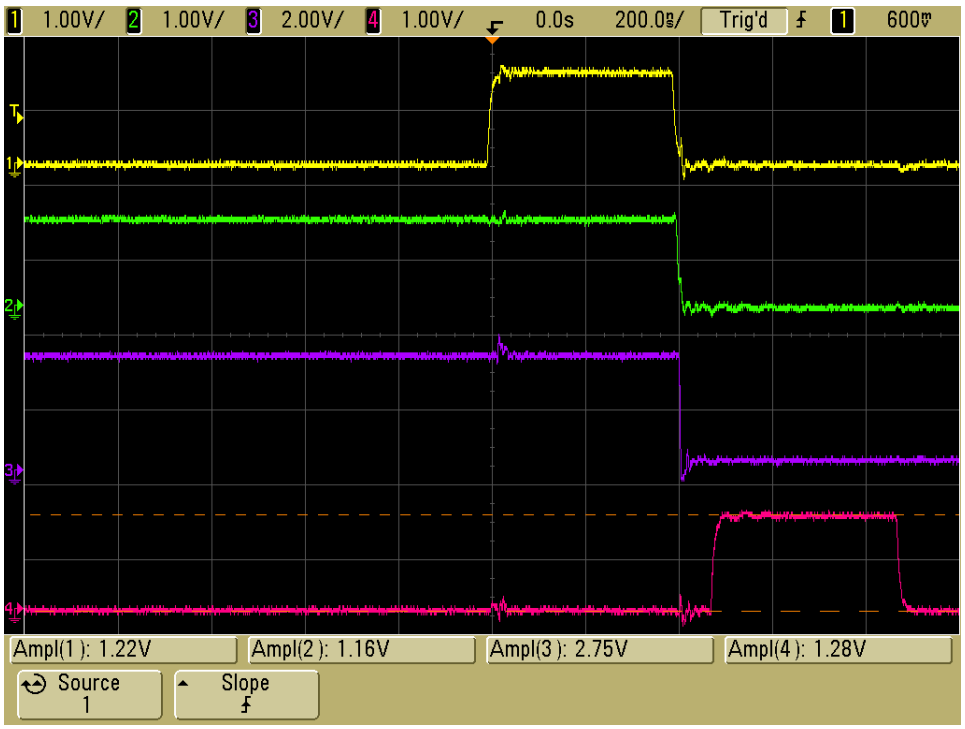



Figure 5.5: Oscilloscope screenshot showing an error in bit 3 of the ALU output during an OR data set at 1.2V.

# Chapter 6

# Discussion

## 6.1 Internal Request to the Output Register

When a REQIN signal is sent to the ALU, it will be propagated by the instruction decoder to one of the DCVSL subcircuits. When the subcircuit has processed the input data, the completion generator connected to it will send a request to the output register, which will then read the data into the latches, and trigger the REQOUT signal. This request propagation is shown in figure 5.3 on page 62. When REQIN goes low again, the same completion generator output will transition from 1 to 0. However, because REQIN is 0, the transmission gate connected to the completion generator will be disabled, and the completion generator will be virtually disconnected (high impedance) from the output register by the transmission gate. As a consequence, the request signal to the output register is never actively pulled low after a data set has been processed. The output register cannot accept new data until the request goes low again, which is why we see lap times in the order of microseconds instead of nanoseconds in tables 5.1-5.5. After each data set, the internal request line to the output register will eventually transition back to 0 because of leakage currents through the 8 transmission gates attached, but the operating speed of the ALU is greatly affected.

This design error could easily be corrected by connecting a standby pull-down NMOS transistor to the internal request line. The gate of this transistor would be connected to ALU $\overline{REQIN}$, which is already generated internally in the instruction decoder.

## 6.2 Single-bit Errors in the Output Data

In addition to the lack of an active pull-down on the internal request, there appears to be issues with the data lines to the output register. As we can see in tables 5.1-5.5, there are many bit errors in the output stream, especially at lower supply voltages. The errors manifest as single-bit

misses, seemingly uncorrelated to any specific pattern of input data. An example of such an error, during an OR operation, is shown in figure 5.5 on page 64. The second trace (green) shows the value of ALU output bit 3. During this particular instruction, A = '00000000' and B = '00010000', so output bit 3 should be 0 when REQ (yellow trace) goes high. Instead, output bit 3 stays high while REQ is high, and transitions to 0 when REQ goes low.

A falling edge on the REQ signal should in theory not affect the data outputs, as is the case whenever no output errors occur. It is possible that stray parasitics in the long internal conducting lines cause the glitches that are observed here[5].

## 6.3 Schematic Errors in the Kogge-Stone Adder

In the design of the Kogge-Stone adder, the P (Propagate) inputs to the bottom cells are connected to the mid cells directly above, instead of the top cells. A schematic with these errors present is shown in figure 6.1 on the facing page. Figure 3.9 on page 24 shows the correct schematic. This design error was uncovered during simulation, but unfortunately the chip layout was already delivered to TSMC for manufacturing. Because of this error, the adder will not produce the correct sum whenever carries are generated.

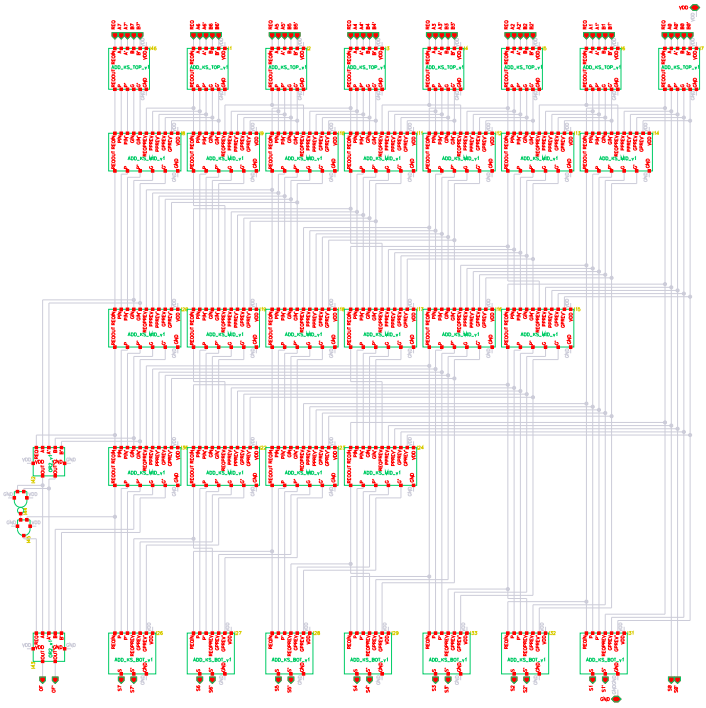


Figure 6.1: Schematic view of the Kogge-Stone adder with errors.

## 6.4 Measuring Current Consumption

In table 5.6 on page 58 and figure 5.1 on page 59 the static current consumption of the ALU chip can be observed. However, due to the limited sensitivity and temporal resolution of the lab equipment, dynamic current consumption could not be measured. A possible solution to this problem is to include current-sensing circuitry on the chip, as demonstrated in [1].

## 6.5 Irregularities in Layout Cell Size

When implementing layout blocks, especially larger ones such as the adder in 3.9 on page 24, it can be beneficial to first determine a standard size for the underlying cells and logic gates. If all gates are designed using the same height and width, fitting them together in larger blocks is much easier, and may in the end result in less area overhead. The adder designed in this thesis contains a lot of unused space between cells, because of small variations in size in the smaller cells. As bigger and bigger blocks are fitted together, the severity of these initially small variations increase, eventually leading to large offsets in cell placement.

## 6.6 Lower limit of the Supply Voltage

As shown in the results, no data is logged for ALU supply voltages lower than 0.40V. Attempts were made to lower the supply voltage further than 0.40V, but this resulted in uncontrollable oscillations on the input lines to the ALU. As we can see in figures 5.2-5.5, considerable ringing occurs on the request input when the voltage comparators switch output values. Steps taken to reduce the susceptibility to noise on the input lines, such as using lower resistance potentiometers, and better separation of high voltage and low voltage PCB tracks, could result in increased stability, enabling the ALU to be tested at lower voltages.

# Chapter 7

# Conclusion

In this thesis, an 8-bit differential asynchronous ALU designed in the 90nm process from TSMC has been constructed and presented. A test bench has been created, using a custom built PCB and microcontroller firmware.

Future work should include rigorous simulation runs of the ALU to determine the origin of the single-bit data errors. Parasitic extraction should be performed, to investigate the cause of potential crosstalk between signals.

The layout of the ALU should be optimized, to reduce the total occupied chip area. Design errors in the pathway of the request signal should be corrected, to realize the true performance potential of the ALU. The instruction set of the ALU could be extended with new arithmetic functions, such as multiplication and division.

A software test bench could be made, to verify and improve on the ALU design without the need for a physical chip.

# Bibliography


[1] J. Arguelles, M. Martinez and S. Bracho. 'Dynamic Idd test circuit for mixed signal ICs'. In: *Electronics Letters* 30.6 (Mar. 1994), pp. 485 –486. ISSN: 0013-5194. DOI: 10.1049/el:19940343.

[2] R. Arora and Shrivastava. 'Area Recovery by Abutted Cell Placement: Can Fillers be Killers? An Eye-opening Viewpoint!' In: *Circuits and Systems, 2006. APCCAS 2006. IEEE Asia Pacific Conference on*. Dec. 2006, pp. 805 –807. DOI: 10.1109/APCCAS.2006.342143.

[3] Wen-Yaw Chung, Jian-Ping Chang and F.R.G. Cruz. 'Clock-gated and low-power standard cell library for ISFET Two-Point Calibration processor chip'. In: *Circuits and Systems (APCCAS), 2010 IEEE Asia Pacific Conference on*. Dec. 2010, pp. 1163 –1166. DOI: 10.1109/APCCAS.2010.5774932.

[4] J.M. Colmenar et al. 'Characterizing asynchronous variable latencies through probability distribution functions'. In: *Microprocessors and Microsystems* 33.7-8 (2009), pp. 483 –597. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2009.09.005. URL: http://www.sciencedirect.com/science/article/pii/S0141933109000763.

[5] M. Cooperman and P. Andrade. 'CMOS gigabit-per-second switching'. In: *Solid-State Circuits, IEEE Journal of* 28.6 (June 1993), pp. 631 –639. ISSN: 0018-9200. DOI: 10.1109/4.217977.

[6] G.F. Formicone, W. Burger and B. Pryor. 'Analysis of distributed multi-finger high-power transistors using the FDTD method'. In: *Microwave Symposium Digest, 2005 IEEE MTT-S International*. June 2005, 4 pp. DOI: 10.1109/MWSYM.2005.1516864.

[7] Agnar Grødal. *Elektromagnetisk kompatibilitet for konstruktører*. Tapir forlag, 1997, p. 223. ISBN: 82-519-1271-7.

[8] L. Halbo and P. Ohlckers. *Electronic Components, Packaging and Production*. University of Oslo, 1995, p. 6.

[9] N. Hanchate and N. Ranganathan. 'LECTOR: a technique for leakage reduction in CMOS circuits'. In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 12.2 (Feb. 2004), pp. 196 –205. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2003.821547.

[10] A. Hemani et al. 'Lowering power consumption in clock by using globally asynchronous locally synchronous design style'. In: *Design Automation Conference, 1999. Proceedings. 36th*. 1999, pp. 873 –878. DOI: 10.1109/DAC.1999.782202.

[11] B. Jacob, S. W. Ng and D. T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, Oct. 2007, p. 270. ISBN: 978-0-123-79751-3.

[12] A. Jairath, B. Sivasubramanian and D. Velenis. 'Block placement for reduced delay uncertainty in high performance clock distribution networks'. In: *Circuits and Systems, 2005. 48th Midwest Symposium on*. Aug. 2005, 1454 –1457 Vol. 2. DOI: 10.1109/MWSCAS.2005.1594386.

[13] Kyung Ki Kim et al. 'Clock Grid Simulation using Transient S-parameter Modeling'. In: *Instrumentation and Measurement Technology Conference, 2006. IMTC 2006. Proceedings of the IEEE*. Apr. 2006, pp. 225 –228. DOI: 10.1109/IMTC.2006.328403.

[14] Peter M. Kogge and Harold S. Stone. 'A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations'. In: *Computers, IEEE Transactions on* C-22.8 (Aug. 1973), pp. 786 –793. ISSN: 0018-9340. DOI: 10.1109/TC.1973.5009159.

[15] O. Krammer and T. Garami. 'Investigating the mechanical strength of Vapor Phase soldered chip components joints'. In: *Design and Technology in Electronic Packaging (SIITME), 2010 IEEE 16th International Symposium for*. Sept. 2010, pp. 103 –106. DOI: 10.1109/SIITME.2010.5653622.

[16] T. Kumaran et al. 'Transient current sensing based completion detection with event separation logic for high speed asynchronous pipelines'. In: *TENCON 2009 - 2009 IEEE Region 10 Conference*. Jan. 2009, pp. 1 –6. DOI: 10.1109/TENCON.2009.5396051.

[17] J. Le et al. 'Comparison and Impact of Substrate Noise Generated by Clocked and Clockless Digital Circuitry'. In: *Custom Integrated Circuits Conference, 2006. CICC '06. IEEE*. Sept. 2006, pp. 105 –108. DOI: 10.1109/CICC.2006.321003.

[18] T. M. van Leeuwen. 'Implementation and automatic generation of asynchronous scheduled dataflow graph'. MA thesis. Delft University of Technology, Oct. 2010.

[19] T.H.-Y. Meng, R.W. Brodersen and D.G. Messerschmitt. 'Asynchronous design for programmable digital signal processors'. In: *Signal Processing, IEEE Transactions on* 39.4 (Apr. 1991), pp. 939 –952. ISSN: 1053-587X. DOI: 10.1109/78.80917.

[20] S.A. Mondal, S. Talapatra and H. Rahaman. 'Analysis, modeling and optimization of transmission gate delay'. In: *Quality Electronic Design (ASQED), 2011 3rd Asia Symposium on*. July 2011, pp. 246 –253. DOI: 10.1109/ASQED.2011.6111754.

[21] Chris J. Myers. *Asynchronous Circuit Design*. John Wiley & Sons, July 2001, p. 323. ISBN: 978-0-471-41543-5.

[22] K. Nandhasri and J. Ngarmnil. 'Hysteresis tunable FGMOS comparator'. In: *Semiconductor Electronics, 2000. Proceedings. ICSE 2000. IEEE International Conference on*. 2000, pp. 173 –177. DOI: 10.1109/SMELEC.2000.932458.

[23] J. Sparsø and S. Furber. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001, pp. 3,5–8. ISBN: 0-7923-7613-7.

[24] A. Todri et al. 'A study of decoupling capacitor effectiveness in power and ground grid networks'. In: *Quality of Electronic Design, 2009. ISQED 2009. Quality Electronic Design*. Mar. 2009, pp. 653 –658. DOI: 10.1109/ISQED.2009.4810371.

[25] A. Waizman. 'CPU power supply impedance profile measurement using FFT and clock gating'. In: *Electrical Performance of Electronic Packaging, 2003*. Oct. 2003, pp. 29 –32. DOI: 10.1109/EPEP.2003.1249993.

[26] S. B. Wijeratne et al. 'A 9-GHz 65-nm Intel reg; Pentium 4 Processor Integer Execution Unit'. In: *Solid-State Circuits, IEEE Journal of* 42.1 (Jan. 2007), pp. 26 –37. ISSN: 0018-9200. DOI: 10.1109/JSSC.2006.885055.

[27] J.L. Yang and C.W. Huang. 'A Power Aware Design Technique for High Performance Self-Timed Datapaths'. In: *Electron Devices and Solid-State Circuits, 2007. EDSSC 2007. IEEE Conference on*. Dec. 2007, pp. 851 –854. DOI: 10.1109/EDSSC.2007.4450259.

# Appendix A

# Chip package with bonding diagram and padframe

For access to Cadence Virtuoso source files, please contact this author or thesis supervisor Philipp Häfliger.

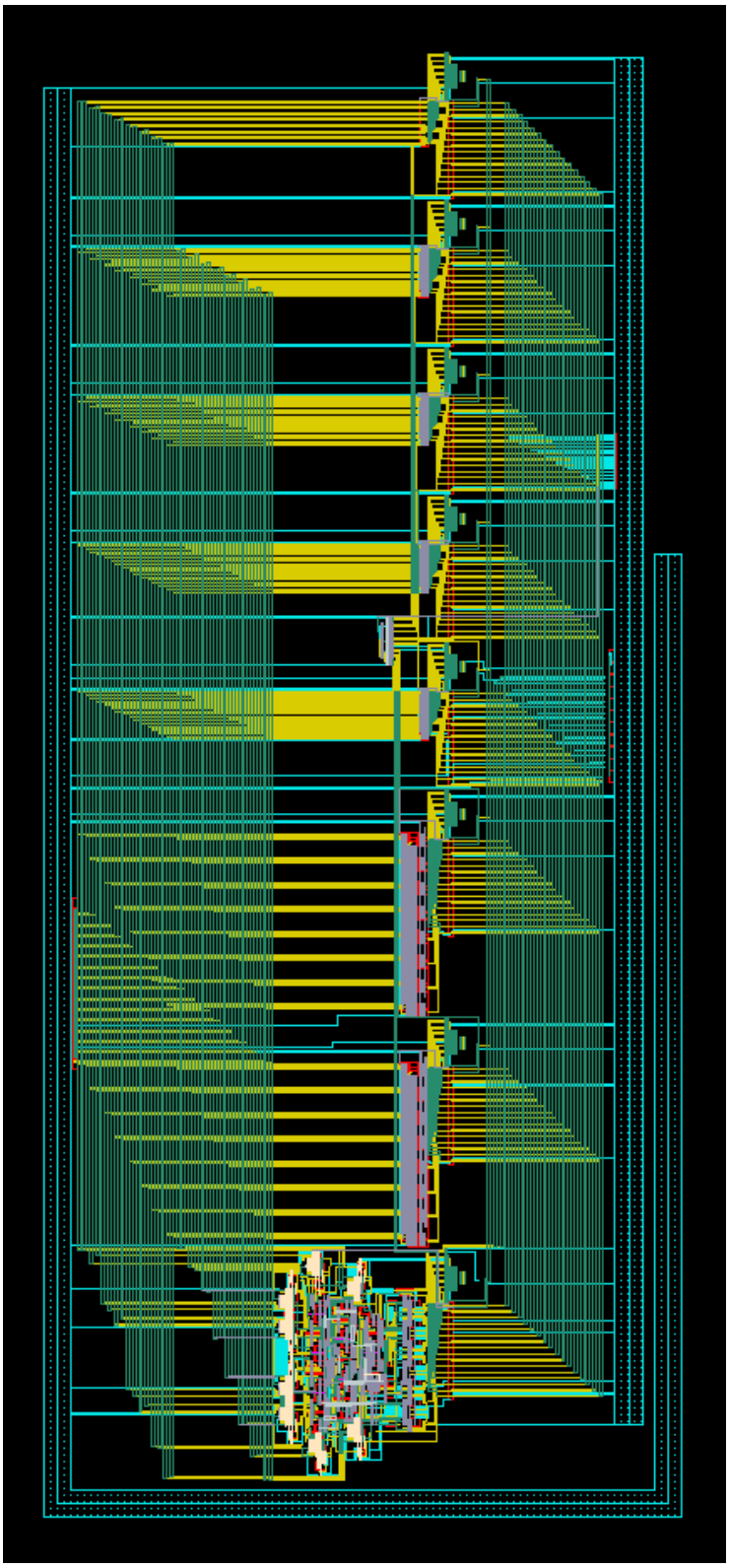


Figure A.1: Full view of ALU layout, not including pad connections.

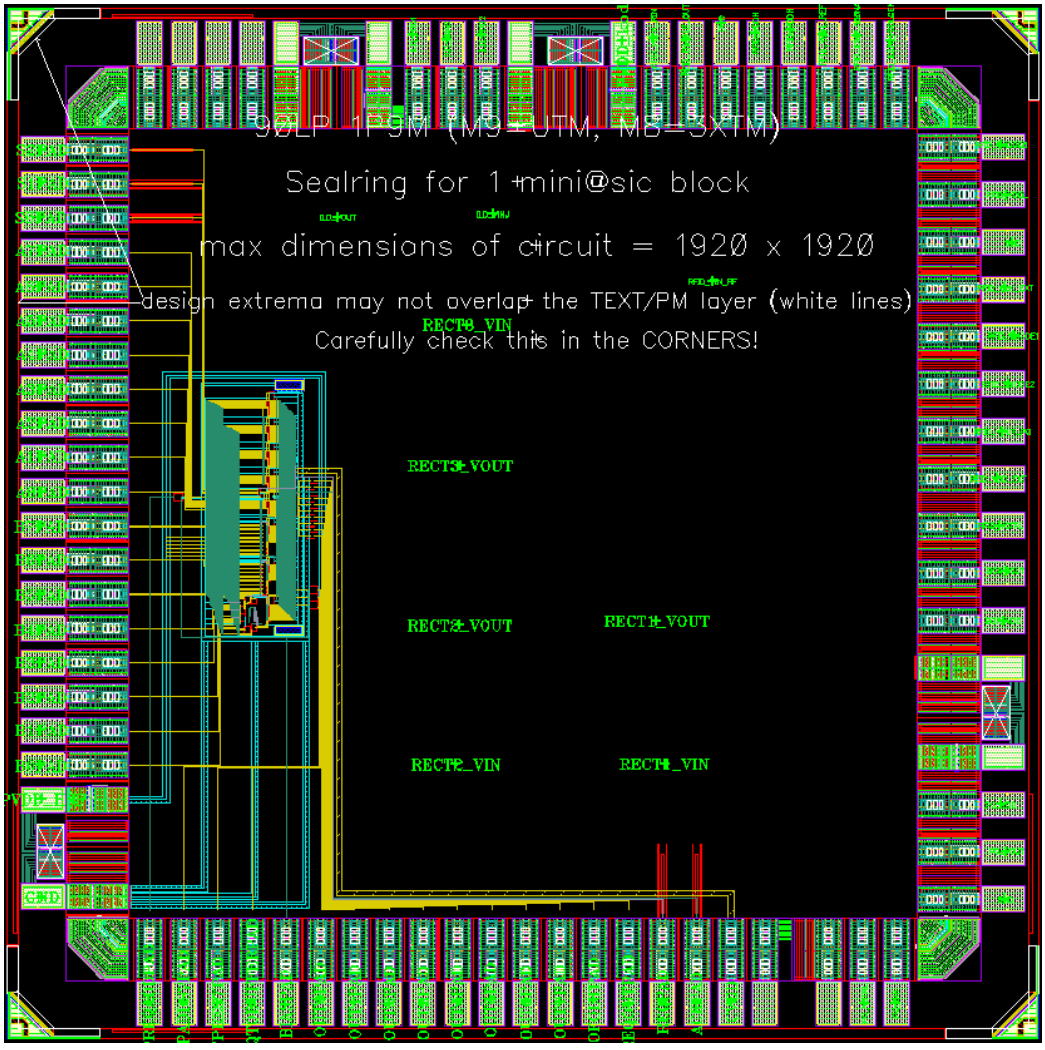



Figure A.2: Full view of ALU layout inserted into the padframe.

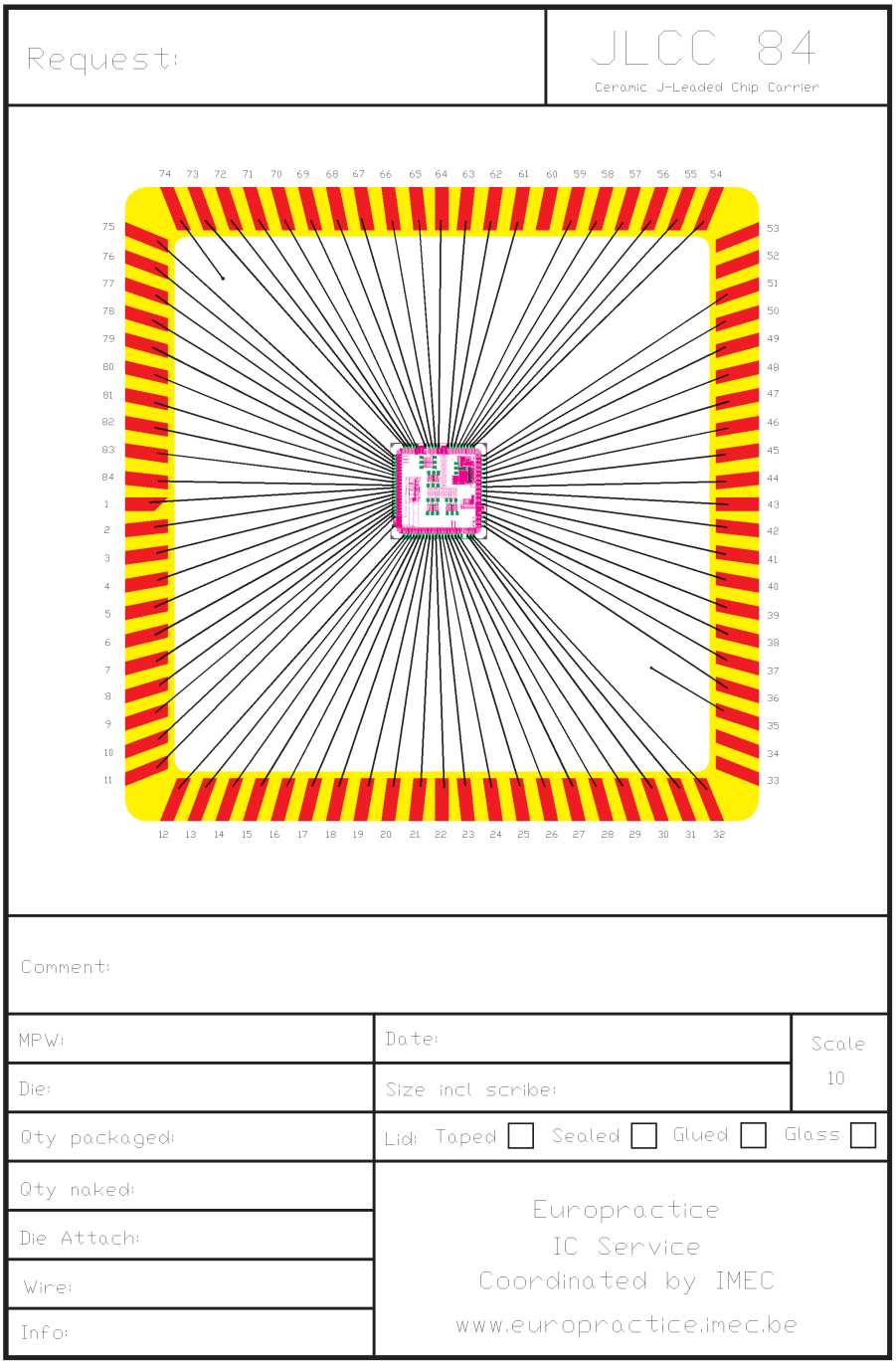



Figure A.3: Bonding diagram for the ALU chip.

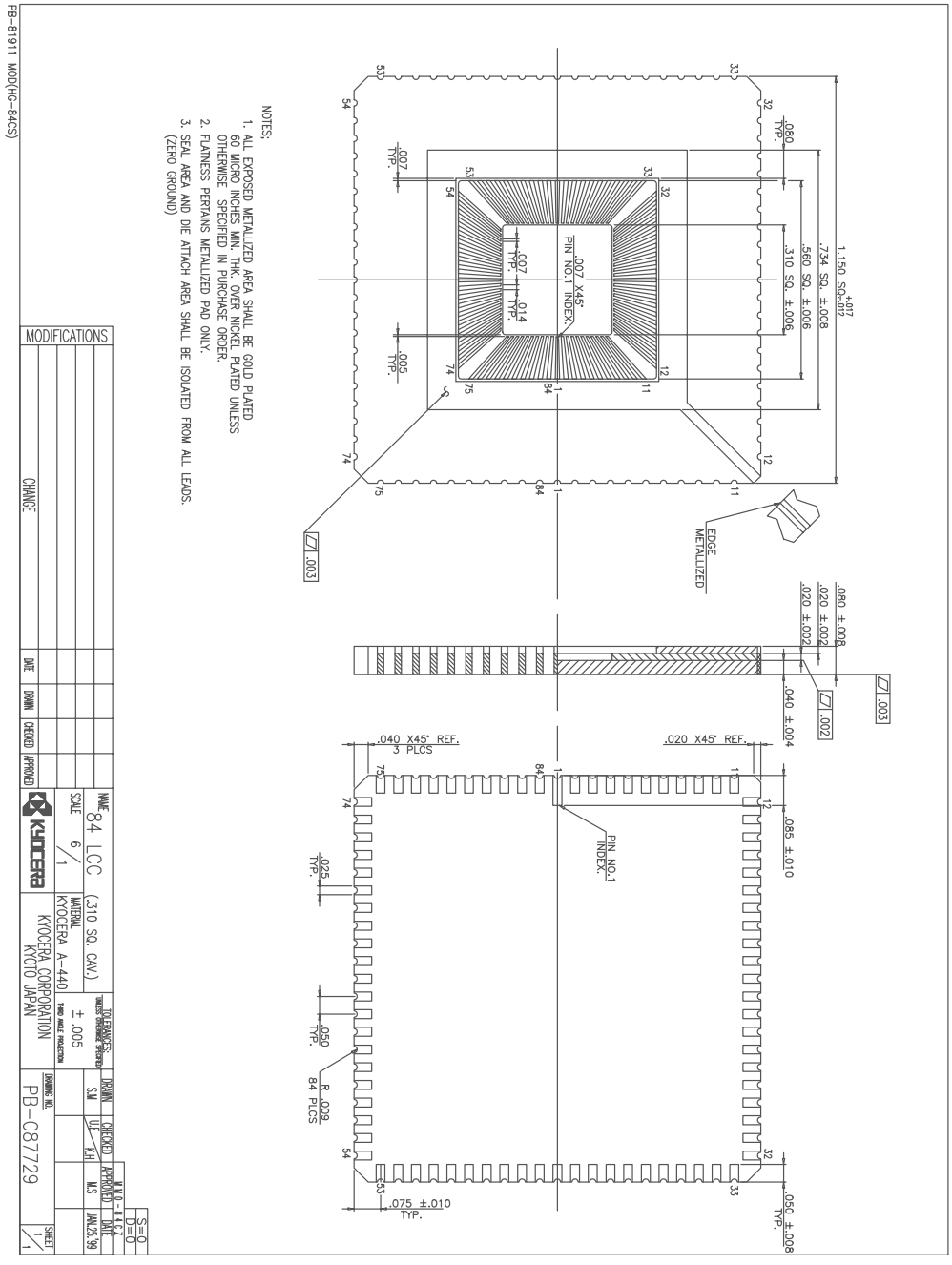



Figure A.4: Data sheet for the chip package showing physical dimensions.

# Appendix B

# PCB full schematics and layout

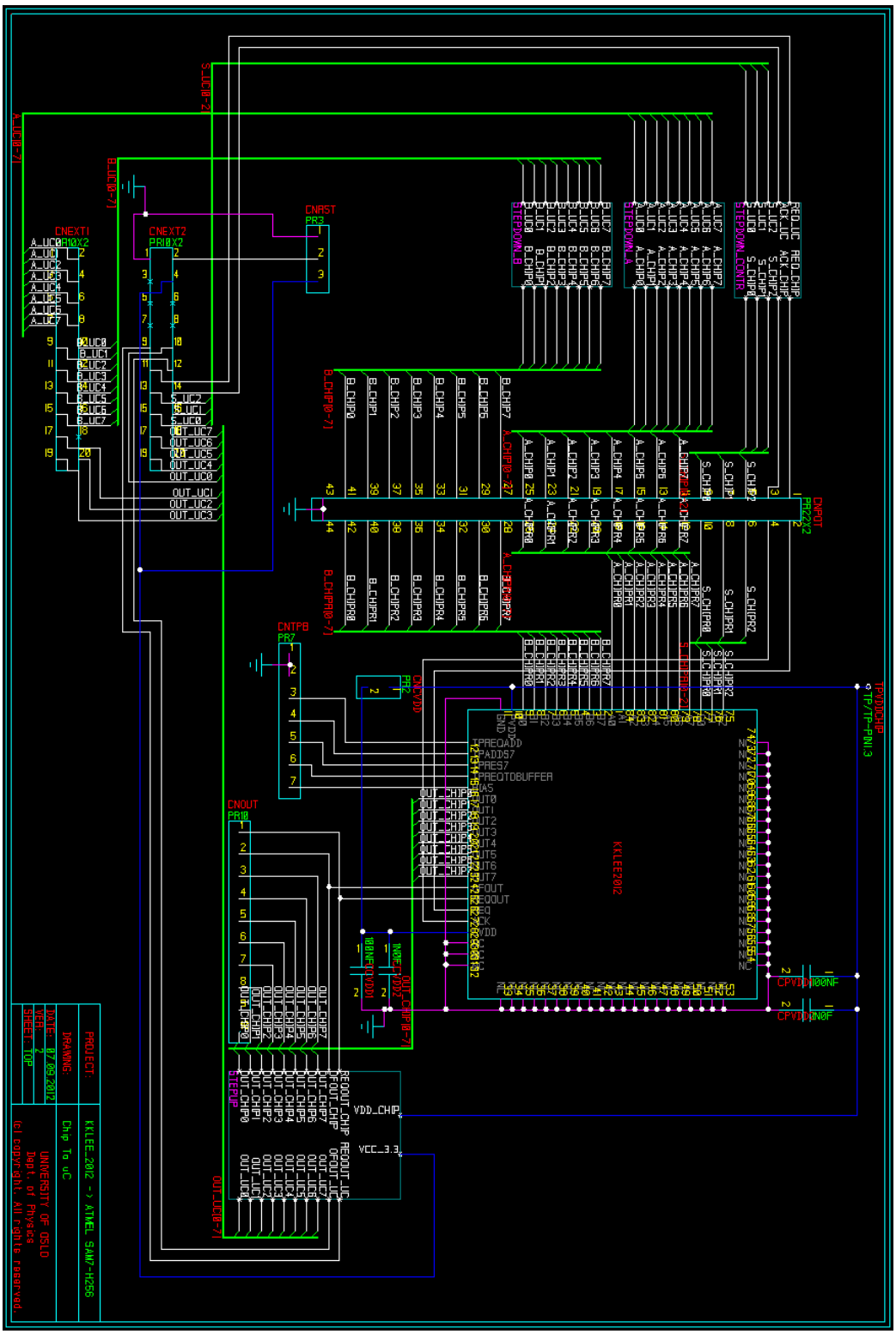


Figure B.1: Top level sheet of the PCB schematic.

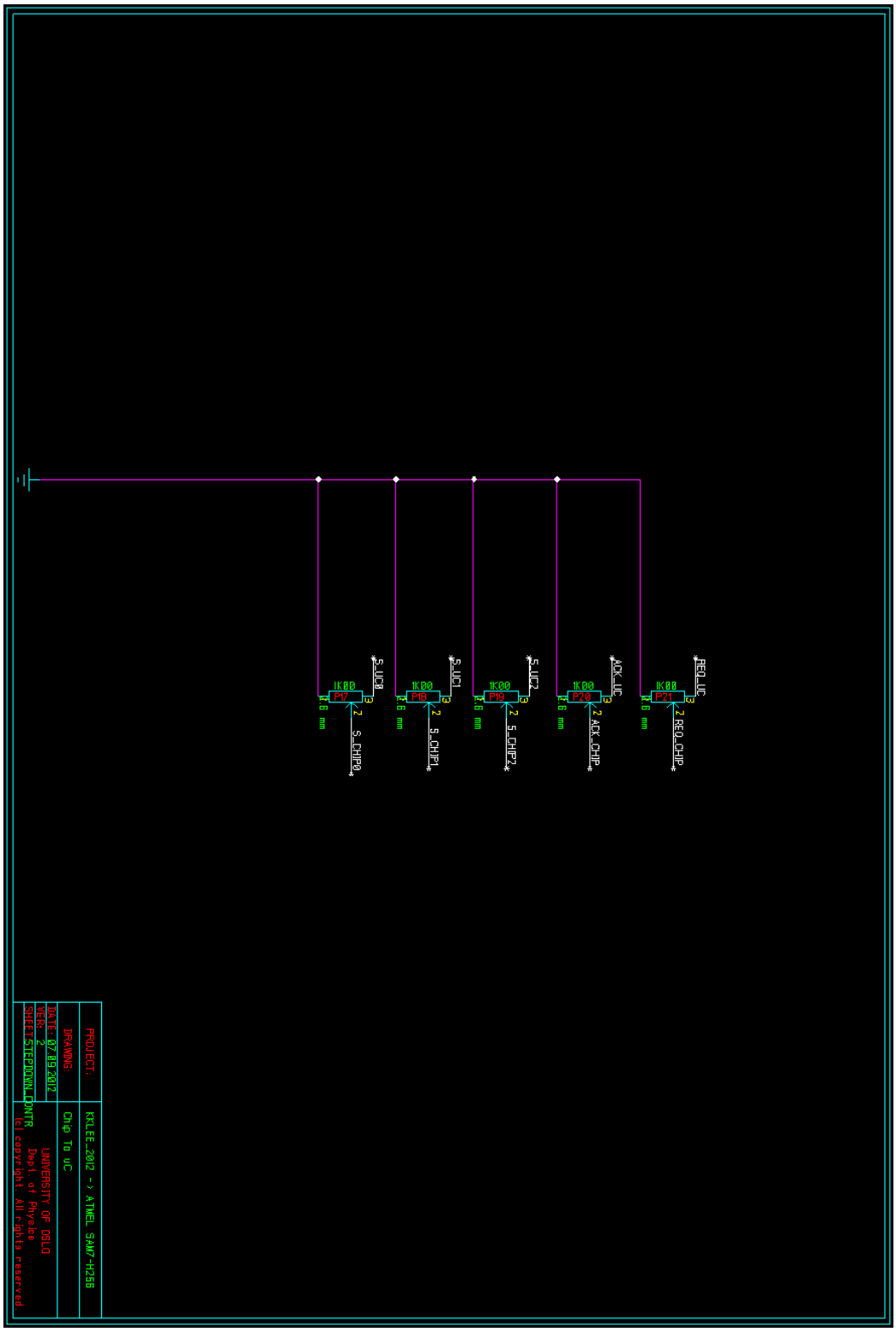



Figure B.2: PCB schematic, design sheet STEPDOWN_CONTR, showing potentiometers for the Request-, Acknowledge- and Select signals.

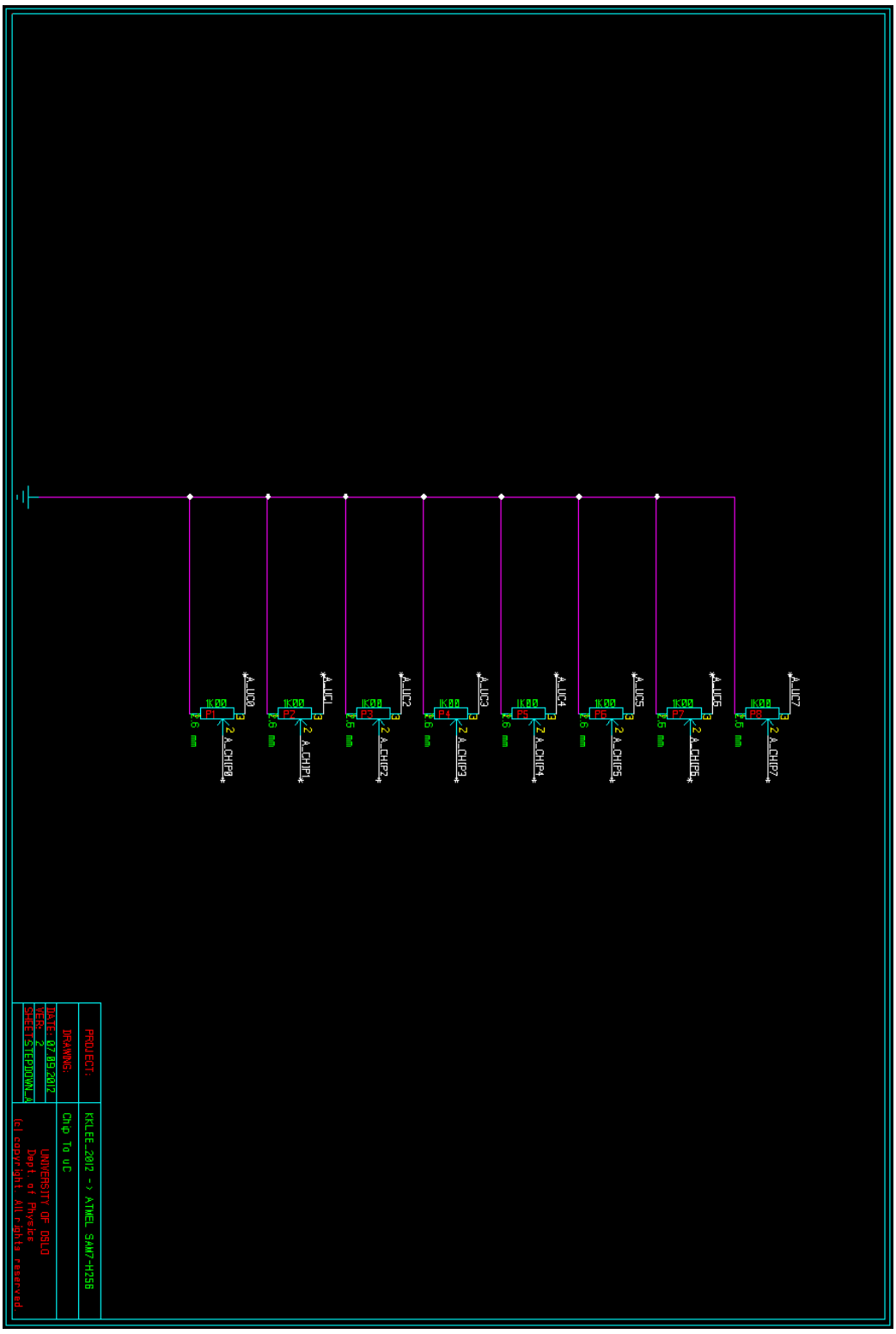



Figure B.3: PCB schematic, design sheet STEPDOWN_A, showing potentiometers for ALU input vector A.

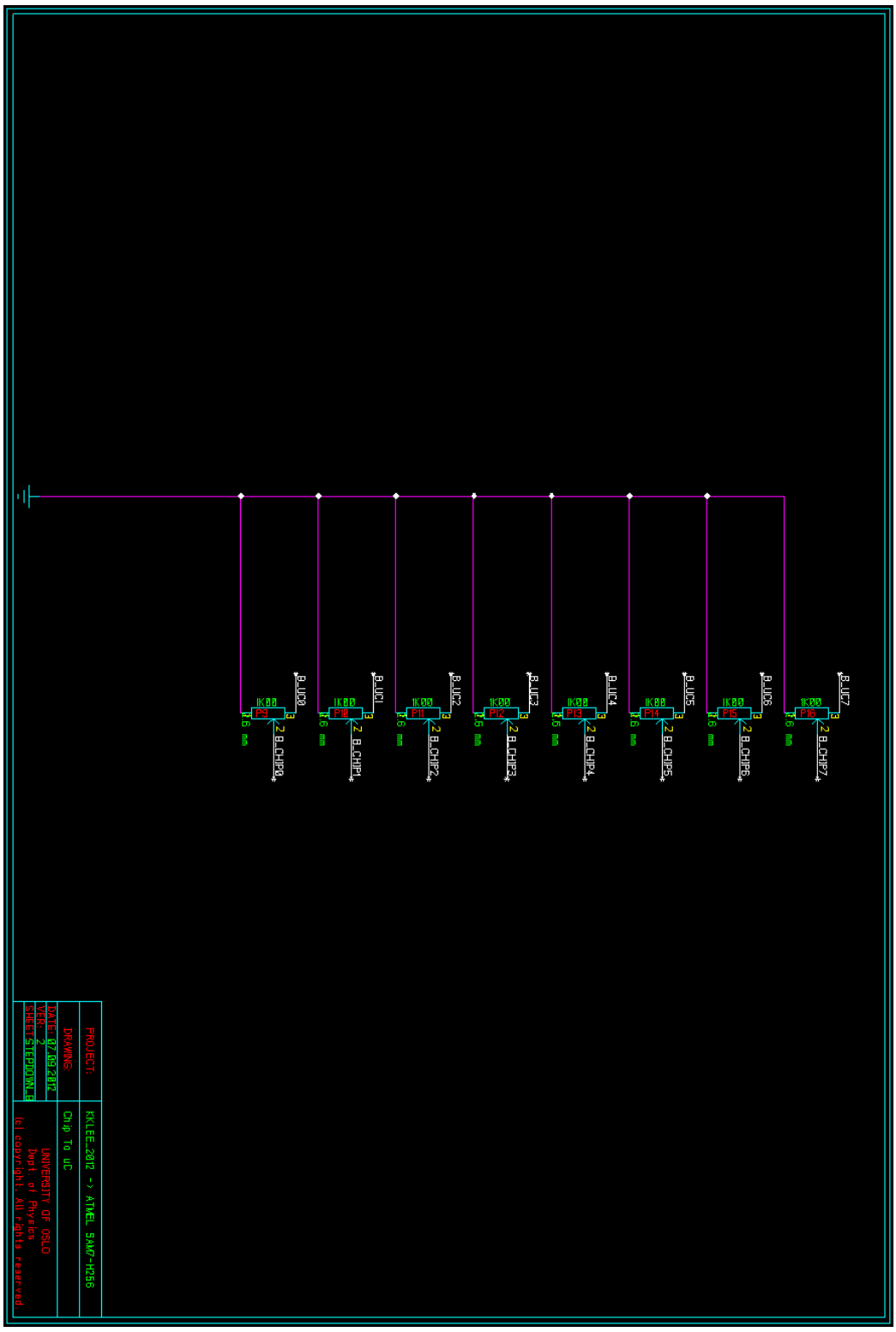



Figure B.4: PCB schematic, design sheet STEPDOWN_B, showing potentiometers for ALU input vector B.

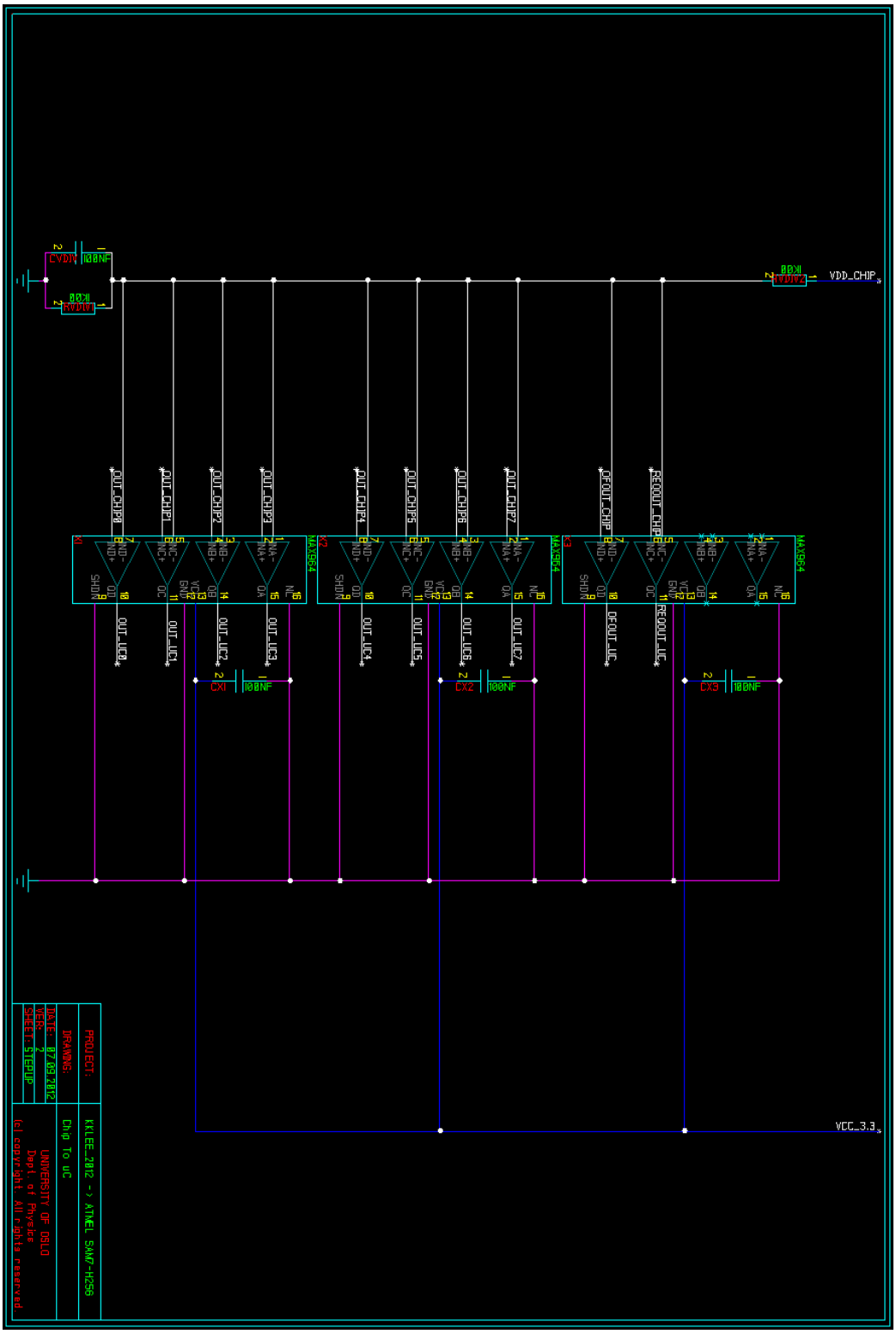



Figure B.5: PCB schematic, design sheet STEPUP, showing the voltage comparators for the ALU outputs.

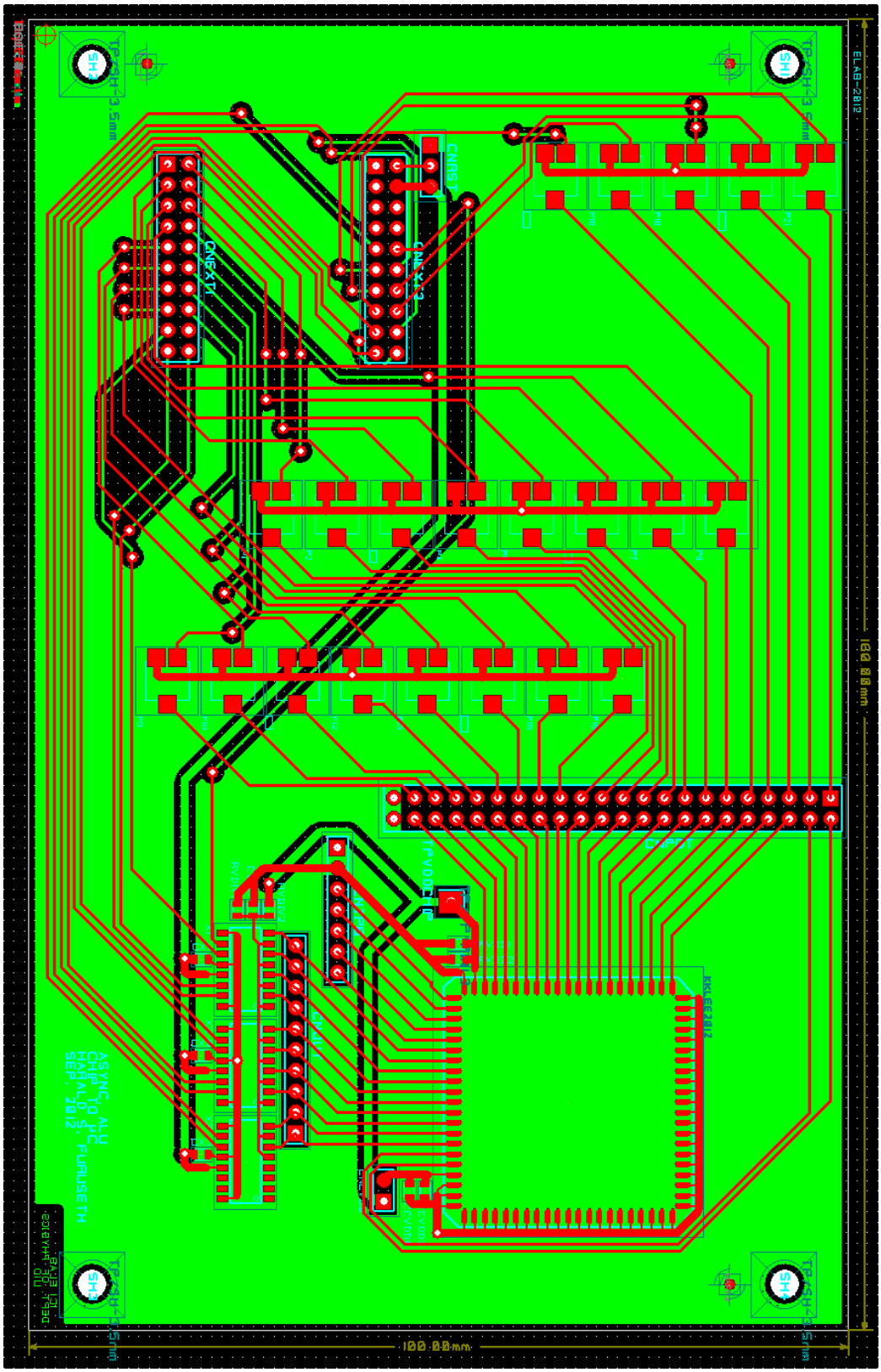


Figure B.6: PCB layout.

# Appendix C

# Olimex SAM7-H256 header board

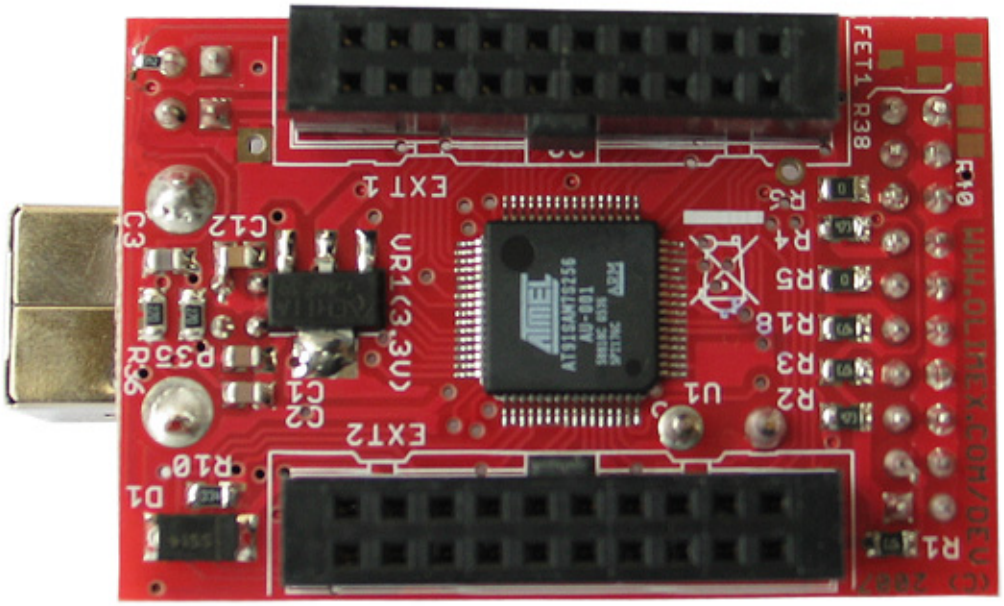



Figure C.1: Bottom view of the SAM7-H256 header board, showing the AT91SAM7S256 microcontroller, two pin rows for interconnect, and the USB port on the left.

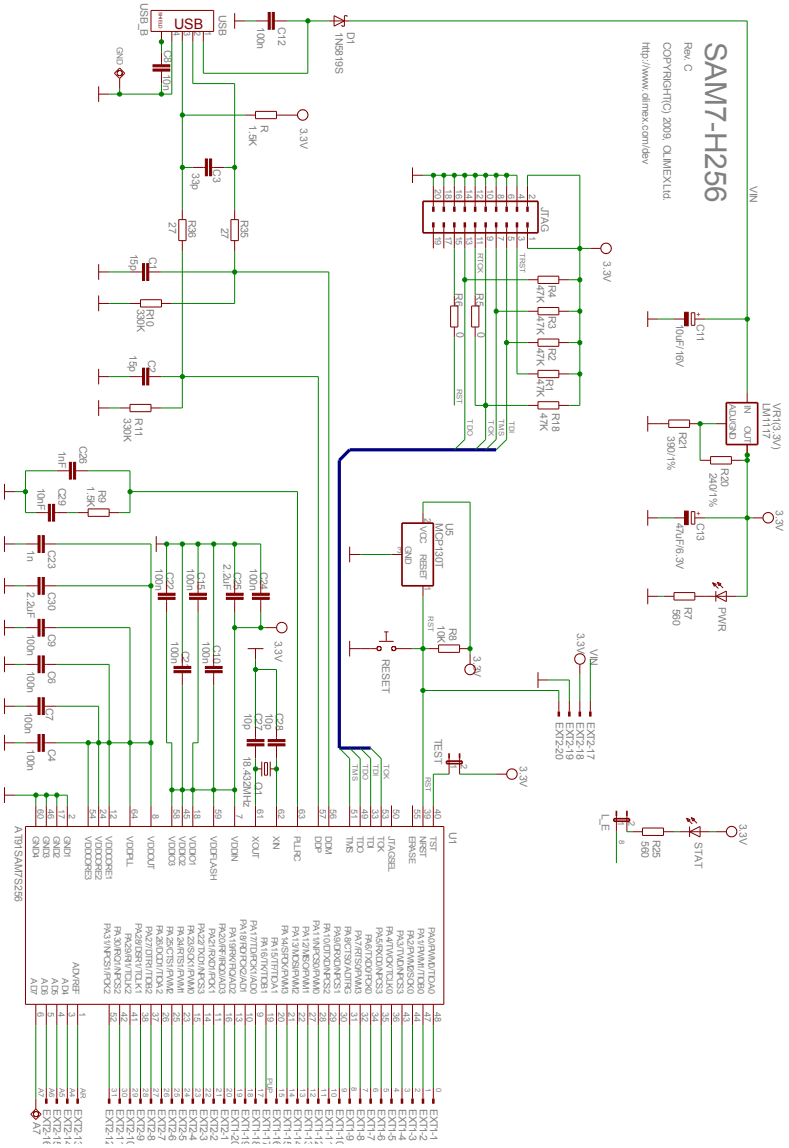



Figure C.2: Schematic of the header board.

# Appendix D

# Microcontroller C code and relevant header files

Below is Håkon Hjortland's header file for the Olimex SAM7-H256 board, modified by this author to include I/O addresses for use with the ALU chip.

```
1  /* ***********************************************************
2   * Copyright (C) 2008  Håkon A. Hjortland <haakoh@ifi.uio.no>
3   * You may use all or parts of this code any way you want. You
        can even remove
4   * the copyright notice and these conditions. There is NO
        WARRANTY, to the
5   * extent permitted by law.
6   *********************************************************** */
7
8  /* ********************************************
9   * Edited by Harald S. Furuseth for use in master thesis, fall
        2012
10  ******************************************** */
11
12 #ifndef _BOARD_H
13 #define _BOARD_H
14
15 #include "sam7.h"
16 #include "misc.h"
17
18 //////////////////////////////////////////////////////////
19 // Master Clock
20
21 #define EXT_OSC          18432000        // External oscillator
      MAINCK
22 #define MCK              48054857        // MCK (PLLRC div by 2)
23
24 //////////////////////////////////////////////////////////
25 // Peripherals in use
26
27 #define pPIO                             AT91C_BASE_PIOA
28 #define pSPI                             AT91C_BASE_SPI
```

```
29 #define pUSART                          AT91C_BASE_US0
30
31 //////////////////////////////////////////////////////////
32 // PIO pins in use
33
34 // Examples:
35 #define PIO_PA8_LED                     AT91C_PIO_PA8          /* LED
       on Olimex SAM7-H256 etc. */
36 #define PIO_PA5_Led1                    AT91C_PIO_PA5
37 #define PIO_PA6_Led2                    AT91C_PIO_PA6
38 #define PIO_PA7_Led3                    AT91C_PIO_PA7
39 #define PIO_PA18_Btn1                   AT91C_PIO_PA18
40 #define PIO_PA19_Btn2                   AT91C_PIO_PA19
41 #define PIO_PA20_Btn3                   AT91C_PIO_PA20
42 #define PIO_PA21_Btn4                   AT91C_PIO_PA21
43 #define PIO_PA28_PCBversionBit0         AT91C_PIO_PA28
44 #define PIO_PA29_PCBversionBit1         AT91C_PIO_PA29
45 #define PIO_PA30_PCBversionBit2         AT91C_PIO_PA30
46 #define PIO_PA31_PCBversionBit3         AT91C_PIO_PA31
47 #define PA17_PCK1_ClockOutput           AT91C_PA17_PCK1
48
49 //////////////////////////////////////////////////////////
50 // Memory information
51 #define SRAM_BASE   ((void *)0x00200000)
52 #define FLASH_BASE  ((void *)0x00100000)
53 #define TEXT_SECTION_SIZE ((int)&_etext)
54
55 //--------------------
56 // ALU Definitions
57 //--------------------
58 #define REQ_MASK      0x10000000 // PA28
59 #define ACK_MASK      0x08000000 // PA27
60 #define S_ALL_MASK      0x07000000 // PA26-24
61 #define A_ALL_MASK      0x000000FF // PA7-0
62 #define B_ALL_MASK      0x0000FF00 // PA15-8
63 #define REQOUT_MASK     0x40000000 // PA30
64 #define OFOUT_MASK      0x20000000 // PA29
65 #define OUT_7TO1_MASK 0x00FE0000 // PA23-17
66 #define OUT0_MASK     0x80000000 // PA31
67
68
69 #endif                                  // _BOARD_H
```

Below is the C code for the method to test the XOR function on the ALU. The other test methods are not shown in this appendix, as they are virtually identical to this one. The only difference is the C operator used in the code to verify the ALU output (Bitwise NOT, bitwise AND etc.). Also, the C method to test the NOT function uses a single for-loop instead of a double for-loop when generating input vectors, as the NOT function only reads ALU input channel A.

```
1 //Tests the XOR opcode by running through all input A values
     from 0 to 255, and for each A input, running through B = 0 to
```

```
    255, and reporting the results over UART
2 void testXOR() {
3
4   // Power Management Controller (PMC): Enable PIO clock
5   AT91F_PMC_EnablePeriphClock(AT91C_BASE_PMC, 1 << AT91C_ID_PIOA
        );
6
7   // PIO Output Enable Register - Enable output for REQ, ACK, S,
        A and B
8   pPIO->PIO_OER = REQ_MASK | ACK_MASK | S_ALL_MASK | A_ALL_MASK
        | B_ALL_MASK;
9
10  // PIO Clear Output Data Register - Set all outputs low
11  pPIO->PIO_CODR = 0xFFFFFFFF;
12
13  unsigned long j;
14
15  // Set S = '101' (XOR)
16  pPIO->PIO_SODR = 0x05000000;
17
18  // Set ACK high and then low again, to clear the ALU output
        register
19  for (j = 3000000; j != 0; j--);  // wait a while
20  pPIO->PIO_SODR = ACK_MASK;
21  for (j = 3000000; j != 0; j--);  // wait a while
22  pPIO->PIO_CODR = ACK_MASK;
23  for (j = 3000000; j != 0; j--);  // wait a while
24
25  // Initialize input and output data, success counter and
        Request OK boolean
26  unsigned long input_A = 0x00000000;
27  unsigned long input_B;
28  unsigned long output;
29  unsigned short nrSuccesses;
30  short REQOK;
31
32  // Loop through all A inputs from 0 to 255
33  while (input_A != 0x00000100) {
34    input_B = 0x00000000; // Reset input B
35    nrSuccesses = 0;
36    while (input_B != 0x00010000) {        // Loop through all B
          inputs from 0 to 255
37      pPIO->PIO_SODR = (input_A | input_B); // Set ALU input
          data
38      pPIO->PIO_SODR = REQ_MASK;          // Set REQ high
39
40      REQOK = 0;
41      for (j = 300000; j != 0; j--) {         // Wait for REQOUT
          to go high. Will time out eventually.
42        if ((pPIO->PIO_PDSR & REQOUT_MASK) != 0) {  // If REQOUT
            is high
43          REQOK = 1;
44          break;                    // Exit for-loop
45        }
```

```
46          }
47
48          if (REQOK) {                        // If a REQOUT was
                received from the ALU
49            output = pPIO->PIO_PDSR;              // Read ALU output
50            pPIO->PIO_CODR = REQ_MASK;              // Set REQ low
51            pPIO->PIO_SODR = ACK_MASK;              // Set ACK high
52            while ((pPIO->PIO_PDSR & REQOUT_MASK) != 0) {}  // Wait
                  until REQOUT goes low
53            pPIO->PIO_CODR = ACK_MASK;              // Set ACK low
54
55            if (((output >> 31) | ((output & OUT_7TO1_MASK) >> 16))
                  == (input_A ^ (input_B >> 8))) { // If ALU output is
                  bitwise A XOR B
56              nrSuccesses++;  // Increment success counter
57            }
58            else {
59
60              // Send great fail over UART
61              usb_print("Testing A = ");
62              usb_print_int(input_A);
63              usb_print(", B = ");
64              usb_print_int(input_B >> 8);
65              usb_print(": Fail! ALU output = ");
66              usb_print_int(((output >> 31) | ((output &
                    OUT_7TO1_MASK) >> 16)));
67              usb_print(", unprocessed output = ");
68              usb_print_hex(output);
69              usb_println(" ");
70            }
71
72            pPIO->PIO_CODR = input_B;       // Clear input B
73            input_B += 0x00000100;          // Increment input B
74            for (j = ALUWait; j != 0; j--);  // wait a while
75          }
76          else {                        // If a REQOUT was not received
                from the ALU
77            pPIO->PIO_CODR = REQ_MASK;          // Set REQ low
78            ALUWait++;                    // Increment wait timer
79            pPIO->PIO_CODR = (input_A | input_B); // Clear ALU input
                  data
80            input_A = 0;                  // Reset input A to 0
81            input_B = 0;                  // Reset input B to 0
82            nrSuccesses = 0;              // Reset success counter to
                  0
83
84            //Send notification over UART
85            usb_print("Wait timer too low, increasing to ");
86            usb_print_int(ALUWait);
87            usb_println(" ");
88
89            for (j = ALUWait; j != 0; j--);  // wait a while
90          }
91      }
```

```
92
93      // Send results over UART
94      usb_print("Successful runs for A = ");
95      usb_print_int(input_A);
96      usb_print(": ");
97      usb_print_int(nrSuccesses);
98      usb_println(" ");
99
100     pPIO->PIO_CODR = input_A; // Clear input A
101     input_A++;               // Increment input A
102   }
103 }
```