ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

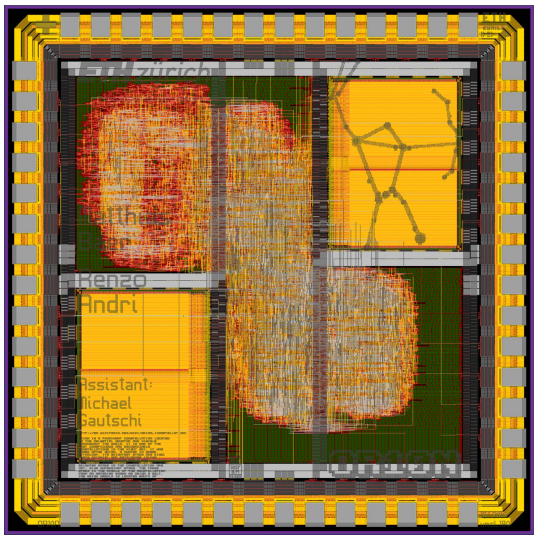
Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and
Electrical Engineering

Autumn Term 2013

Implementation and Optimization of the OpenRISC Processor

Semester Project



Renzo Andri
Matthias Baer
andrire@student.ethz.ch
baermatt@student.ethz.ch

February 2014

Supervisors: Michael Gautschi, gautschi@iis.ee.ethz.ch
Dr. Frank Kagan Gürkaynak, kgf@ee.ethz.ch

Professor: Prof. Dr. Luca Benini, lbenini@iis.ee.ethz.ch

Acknowledgements

We would like to thank our supervisors Frank Gürkaynak and Michael Gautschi for their great support and dedication which made this semester project possible. A special thank you also to Prof. Luca Benini for giving us the opportunity to contribute to the multicore cluster. Finally we express our gratitude to the Integrated Systems Laboratory (IIS) for providing us with industrial software licencies and for financing the ASIC production.

Abstract

Today's goal of high energy efficient computing can be achieved by splitting computation into clusters, which consist of several simple processor cores that can access shared memories. One possibility for those processor cores is the OpenRISC processor core, an open source RISC processor with a small area footprint. However, the current implementation can be improved a lot and this is where this semester thesis comes into play. We started a new and clean implementation of the OpenRISC architecture, focused on simplicity and throughput.

The result is a simplified OpenRISC processor with four well balanced pipeline stages. The achieved frequency of 364.96 MHz is 8.03% higher than the original implementation[1] and the resulting throughput is 67.5% better.

From the final design, an ASIC has been manufactured.

[1] Compared with the same technology and memories.

Declaration of Authorship

We hereby declare that the presented semester thesis is self composed and written only by means and sources as specified in the text and acknowledgements. The final thesis or similar versions of it have so far not been submitted to any other examination board.

Renzo Andri
Matthias Baer,
Zurich, February 2014

Contents

List of Figures

List of Tables

List of Acronyms

ALU . . . . . Arithmetic Logic Unit

ASIC . . . . . Application-Specific Integrated Circuit

CMOS . . . . . Complementary Metal Oxide Semiconductor

DFT . . . . . Design for Testing

DSP . . . . . Digital Signal Processor

EX . . . . . Execute stage

FPGA . . . . . Field Programmable Gate Array

GE . . . . . Gate Equivalent

GPR . . . . . General-Purpose Registers

IC . . . . . Integrated Circuit

ID . . . . . Instruction Decode stage

IF . . . . . Instruction Fetch stage

IIS . . . . . Integrated Systems Laboratory

IPC . . . . . Instructions per Cycles

LSB . . . . . Least Significant Bits

MOPS . . . . . Million Operations per Seconds

MSB . . . . . .Most Significant Bits

OpenRISC . . .Open(-source) Reduced Instruction Set Computing
OPS . . . . . .Operations per Seconds
ORBIS32 . . . .OpenRISC Basic Instruction Set (32bit)

PC . . . . . .Program Counter

RISC . . . . . .Reduced Instruction Set Computing
RTL . . . . . .Register Transfer Level

SoC . . . . . .System on (a) Chip
SPR . . . . . .Special-Purpose Registers

VHDL . . . . . .Very High Speed Integrated Circuit Hardware Description Language

WB . . . . . .Write Back stage

Naming conventions

Table 0.1 shows the used naming convention in this thesis.

Table 0.1.: Naming conventions

l.*	OpenRISC Basic Instruction Set (32bit) (ORBIS32) instruction
l.*s	The result of this instruction is sign-extended (exts).
l.*z	The result of this instruction is zero-extended (extz).
rX	General-Purpose Register X
SR	Supervision Register
spr	Special-Purpose Register
K	Immediate Operand is zero-extended.
I	Immediate Operand is sign-extended.
L	Length Immediate (zero-extended)
X	Don't Care Bits in Instruction Encoding
[*]	Address space
(*)	Memory entry at specified address
{*}	Operations in brackets are executed prior assignment.

Chapter 1

Introduction

1.1. Context and Motivation

Energy efficient computing can be achieved by splitting computation into several clusters. Each cluster consists of several simple processor cores, which are connected to data memories and an instruction cache which is connected by an instruction bus as seen in figure 1.1. One possible single core implementation for this setup is the OpenRISC processor, which is open source and provides a convenient reduced instruction set computing (RISC) software interface. The current implementation of the OpenRISC processor core

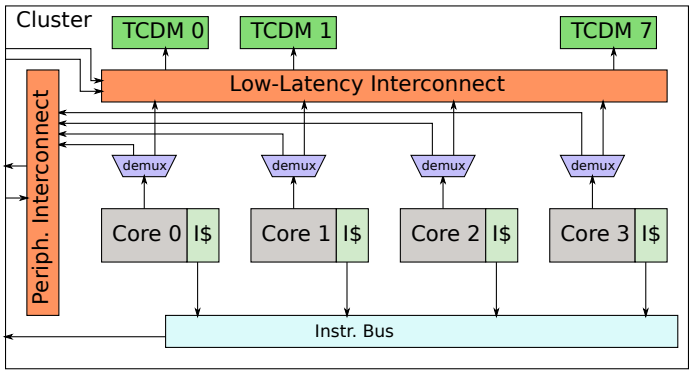


Figure 1.1.: Cluster with 4 processor cores and 8 shared memories.

works as intended, but still has some space for improvements especially in the given environment. Due to high usage of pipeline stalls to prevent hazards, the performance in terms of Instructions per Cycles (IPC) is pretty poor and absolutely improvable. This is the goal of this semester thesis, to improve the core's performance and to provide clean and well documented code of the implementation.

1.2. Achieved Results

We created our own implementation of the OpenRISC architecture from scratch and in terms of IPC, we managed to improve the original implementation by more than 50% in our performance tests. If we also take the clock frequency of the core into account, which is also slightly higher than the original, we can calculate the Million Operations per Seconds (MOPS) and compare them. Our design can perform 67.5% more operations with an area increase of only 3.12% and we therefore state that the optimizations payed off.

Chapter 2

Related Work

2.1. OpenRISC

Our thesis is based on the OpenRISC 1000 project from OpenCores[1][2], which aims to create free and open source computing platforms. The architecture specifies behavior for 32- and 64-bit RISC/DSP processors. It is designed for high performance, simplicity, low power consumption and scalability.

For the OpenRISC 1000 architecture exists an implementation called OpenRISC 1200 which is 32-bit with Harvard microarchitecture, 5 stage pipeline, virtual memory support and basic DSP capabilities (Figure 2.1). The implementation is written in Verilog and can be downloaded from the OpenCores website.[3]

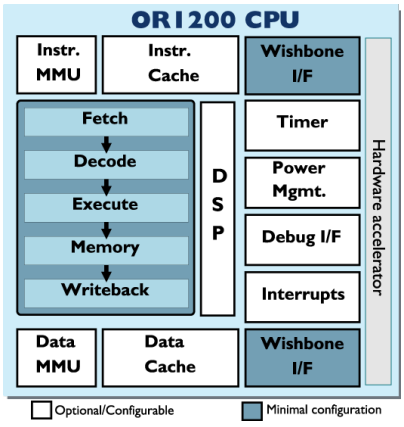


Figure 2.1.: OpenRISC 1200 block diagram[1]

[1] http://opencores.org/or1k/File:Or1200_blocks.png

In order to stay competitive, the OpenCores community called for OpenRISC Application-Specific Integrated Circuit (ASIC) donations, they would use to create a "super-low-cost" SoC ASIC component, but up to the date this document is released, they were not yet successful.

The community has also ported the GNU toolchain to OpenRISC, offering C and C++ support with static libraries. This allows us to create simple test programs for our implementation with very little effort.
It also exists an architecture simulator capable of emulating OpenRISC based computer systems at the instruction level.

Chapter 3

Hardware Architecture

The architecture of the project is hierarchically grouped from the chip level down to the multiplier inside the CPU. This chapter describes some of the most important concepts for each hierarchy level.

3.1. Top Design

One of our first design decisions, was how the memory is handled. It was clear to use a Von Neumann architecture which neither logically nor physically separates the instruction and the data storage, therefore we discussed to use a shared memory that would allow us to specify the size of both memory parts for each running program individually. In this case we would have needed a dual-port memory for accessing instructions and data in the same clock cycle, but there is a big downside for this setup: dual-port memories roughly require double the space for the same amount of data than single-port memories. So in order to keep chip size small and because the disadvantage (twice the size) overweighted the advantage (modular memory size), we decided to use two independent memories for instructions and data. Because of this the chip is basically a modified Harvard architecture.

As seen in figure 3.1 we placed both memories outside the processor itself and embedded them into wrappers in order to simplify the access for the CPU.

During our design process we also added several inputs and outputs to our top level design to be able to test the processor. One key component there is the *Mode_SI* input which specifies what we want our chip to do. There are four possible operating modes: Idle (the chip basically does nothing), Setup (memories are written), Run (processor executes the instructions) and Readout (memories/results are read out).

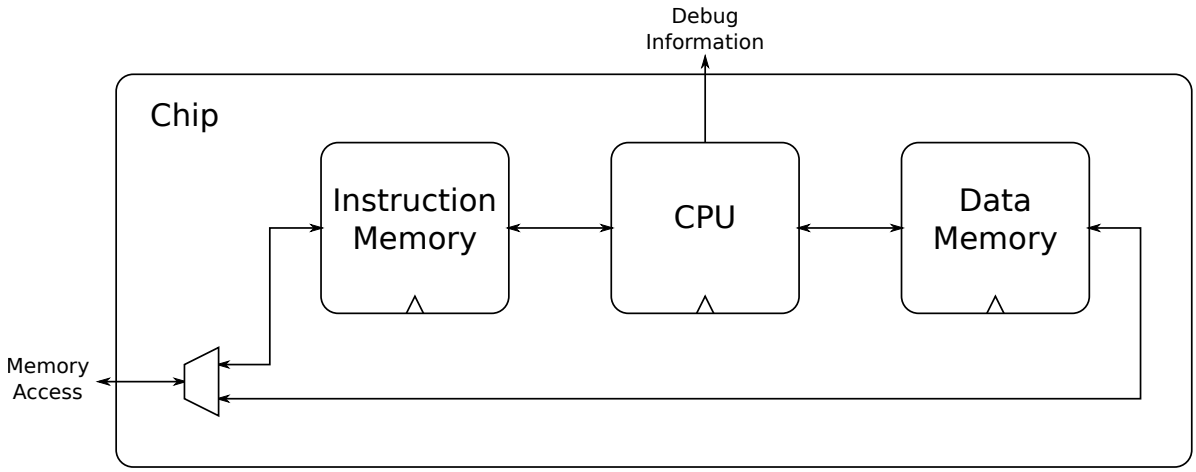


Figure 3.1.: Top Design simplified.

For testing purposes we also added some debugging output to the chip, namely the Program Counter (PC), the branch flag and information about the CPU being stalled.

3.2. Core Entity

3.2.1. Pipelining

Due to simplicity a single-cycle architecture without pipeline stages was implemented first. All signals are comprehensible during simulation and have a relative small control overhead, therefore it was a good joining up the processor design. However, this architecture has a drawback because the longest instruction which is a load word (l.lws) determines the longest path. To reach a higher operational frequency the longest path can be split it up in multiple stages. Table 3.1 presents this pipeline model. Figure 3.2 and Table 3.2 illustrate a program executed on a one-cycle architecture.

Table 3.1.: Pipeline stages.

IF	Instruction Fetch	The current instruction is read from the instruction memory.
ID	Instruction Decode	Instruction is decoded, all needed data is acquired and control signals are set.
EX	Execute	Operation is executed. Address and Data is applied to the data memory.
WB	Write Back	Result is written back to the General-Purpose Registers (GPR)

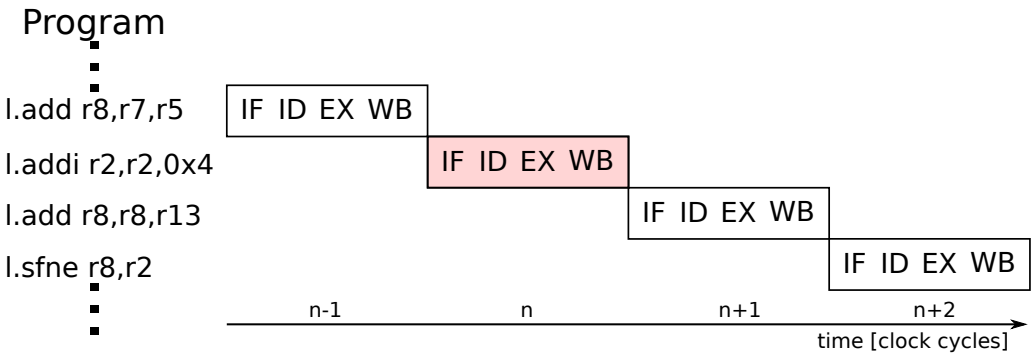


Figure 3.2.: Sample program execution for the single-cycle architecture.

The advantage of this separation is that all of this stages can be executed in separate hardware blocks. This means after the first instruction is fetched the next one can already be fetched while the first one is decoded. After four cycles each stage contains one different instruction. This architecture is referred to the pipelined architecture[4]. The maximal frequency of the core is determined by the longest path of all pipeline stages. In the optimal case the frequency is four times larger in a pipelined architecture with four pipeline stages than in the single-cycle architecture. This can only be reached if the stages are fully balanced.

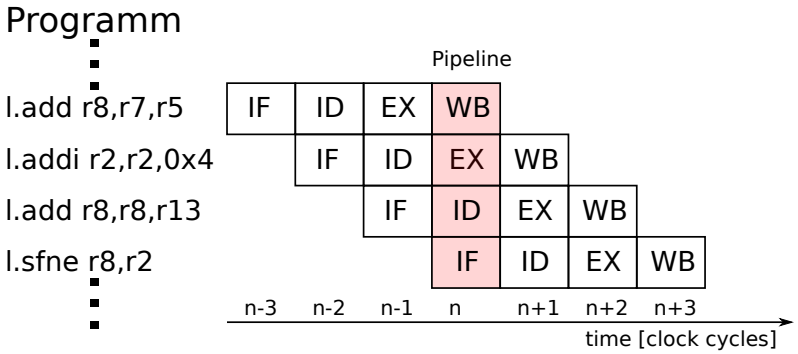


Figure 3.3.: Sample program execution for a pipelined architecture w/o hazard suppression.

Figure 3.3 and Table 3.3 show the execution of a sample program and illustrates the concurrency of four instruction at time n in the pipelined architecture. In each pipeline stage, there is one different instruction processed. The numbers in Figure 3.3 correspond to the register and branch flag values at according cycles.

Unfortunately, there is a data hazard in the pipelined architecture leading to errorneous behaviour. This can be seen if Table 3.3 was compared to 3.2 where no dependecies exist. The hazard occurs due to the fact, that the result is written in the WB stage and

Table 3.2.: Register values for the one-cycle architecture

time	r2	r5	r7	r8	r13	F[a]
$n-1$	6	3	4	0	3	0
n	-	-	-	7	-	-
$n+1$	10	-	-	-	-	-
$n+2$	-	-	-	10	-	-
$n+3$	-	-	-	-	-	0
end	10	3	4	10	3	0

[a](Branch) Flag

Table 3.3.: Register values for the pipelined architecture without hazard suppression

time	r2	r5	r7	r8	r13	F
$n-1$	6	3	4	0	3	0
n	-	-	-	-	-	-
$n+1$	-	-	-	7	-	-
$n+2$	10	-	-	-	-	-
$n+3$	-	-	-	3	-	-
$n+4$	-	-	-	-	-	1
end	10	3	4	3	3	1

accessible one cycle later. However, it is already needed in the ID stage of the following instructions. The first possibility to get rid of this problem is to stall the piepline and wait until the required data is available. This version is exhibited on Figure 3.4 and Table 3.4.

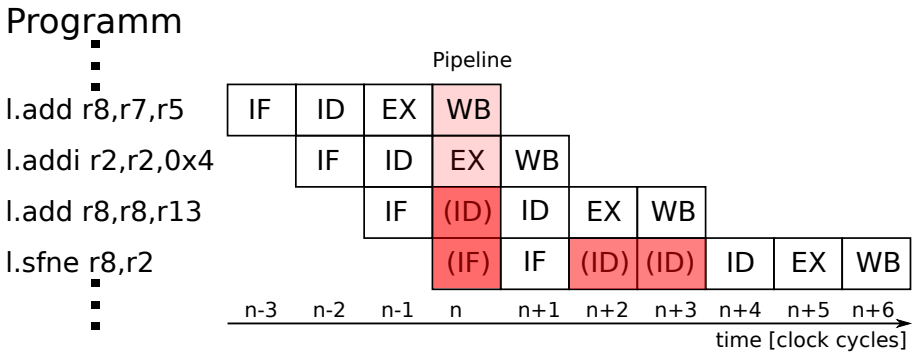


Figure 3.4.: Sample program execution for a pipelined architecture with stalls

3.2.2. Forwarding

A wide known technique to resolve this problem is forwarding. In this architecture variant register value A and/or B are replaced by the value calculated in the EX stage or aquired in the WB stage (e.g. data from data memory). A simplified forwarding block diagram is showed in Figure 3.5 and the resulting execution of the sample program is showed in Figure 3.6 and Table 3.5.

Table 3.4.: Register values for the pipelined architecture waiting for data

time	r2	r5	r7	r8	r13	F
$n-1$	6	3	4	0	3	0
n	-	-	-	-	-	-
$n+1$	-	-	-	7	-	-
$n+2$	10	-	-	-	-	-
$n+3$	-	-	-	-	-	-
$n+4$	-	-	-	10	-	-
$n+5$	-	-	-	-	-	-
$n+6$	-	-	-	-	-	-
$n+7$	-	-	-	-	-	0
end	10	3	4	10	3	0

Table 3.5.: Register values for the pipelined architecture with forwarding

time	r2	r5	r7	r8	r13	F
$n-1$	6	3	4	0	3	0
n	-	-	-	-↩	-	-
$n+1$	-	-	-	↬7⇑	-	-
$n+2$	10	-	-	↑	-	-
$n+3$	-	-	-	⇑10	-	-
$n+4$	-	-	-	-	-	0
end	10	3	4	10	3	0

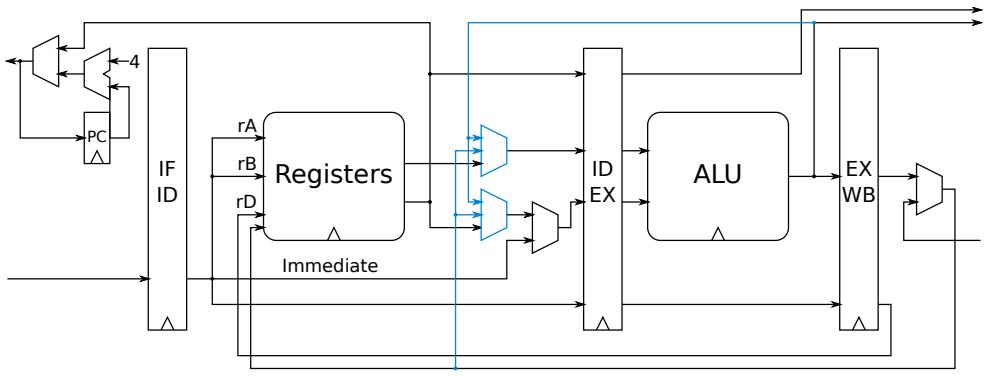


Figure 3.5.: Simplified block diagram with forwarding paths

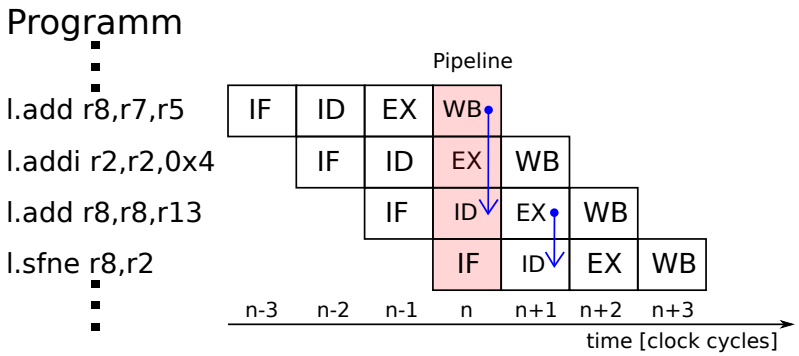


Figure 3.6.: Sample program execution for a pipelined architecture with forwarding

3.3. Controller

Dividing the processor architecture into a data path and a control path is one of the keys to keep the source code and schemata readable. The data path takes care of all data and processes it, while the control path decides which route of the data path will be taken. In our case, we strictly separated the data path (Appendix F.2) from the control path, which we call the controller.

The main purpose of the controller is to decode the current instruction (Instruction Decode stage) and set all signals in order to route the operands to the correct destination. These signals include switching off the PC for jumps, selecting the correct ALU operation or multiplier mode (see Section 3.6), enabling data write to the memory, setting required flags, etc.

Another important task of the controller is to detect hazards and to use forwarding or stalling in these cases accordingly. In most cases we tried to stall only where it was unavoidable and let operations run in parallel or forward data to prevent hazards.

3.4. ALU

The Arithmetic Logic Unit (ALU) is a substantial part of a processor. The ALU calculates all arithmetic operations needed by an instruction. Apart from the arithmetic operations, comparison and extension operations are also executed in the ALU. The ALU requires four inputs: ALU_Op_DI which indicates which operation has to be executed, the two 32 bits long operands $Op1\_DI$ and $Op2\_DI$, and CY_DI - the carry bit[1] of the last executed l.add, l.addc or l.sub instructions. The outputs are the 32 bit long result

[1] Carry bit is set if there was an unsigned overflow, e.g. $3'b101 + 3'b110 = 3'b011 \Rightarrow 5 + 6 = 3$

Result_DO, the carry flag *CY_DO* and the overflow[2] flag *OV_DO*. The two flags are saved in the supervision register which is part of the special-purpose registers.

The ALU supports five different types of operations which are explained in the following sections.

Adding Operations
The first ones are the adding operations. These include addition with and without carry (input) and subtraction. Primarily we need a 32 bit full adder and an addition with and without carry. Secondary, the subtraction can be calculated by this adder if the subtrahend is two's complemented and used as second summand, this is realized by inverting each bit of the second operand and setting the carry input of the full adder to one.

Finally all these operations need to set or unset the carry and overflow flags in the Special-Purpose Registers (SPR). The carry is already available from the full adder, it is the 33th bit of the result. Overflow may occur in two cases, the first one is a positive overflow when two positive numbers are added, but the result is negative and second one when two negative numbers are added and the result is positive. To check this the most significant bit of the operands and the result are compared. Figure 3.7 shows the adder part of the ALU.

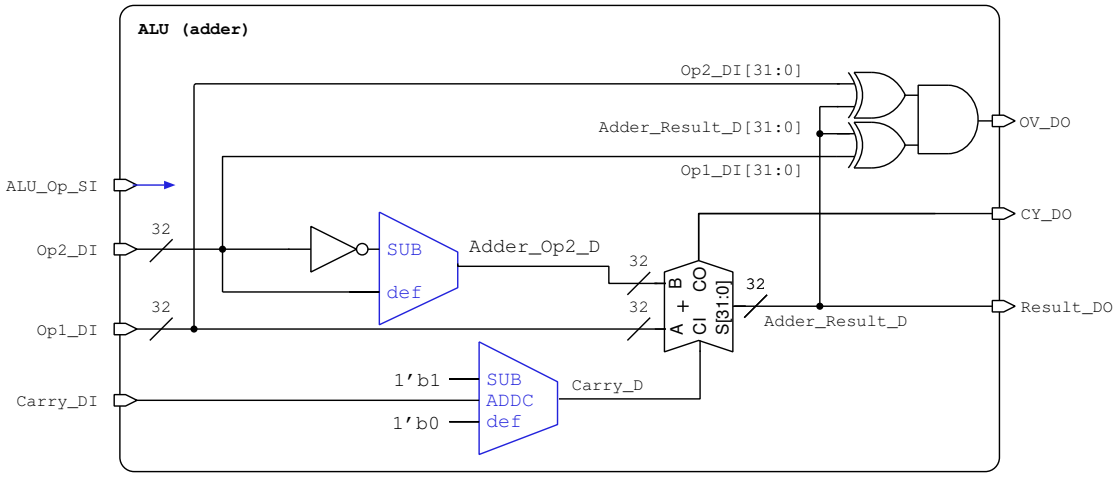


Figure 3.7.: ALU, adding operations.

Bitwise Operations
Bitwise operations execute boolean operations on each bit. The bitwise operations implemented in the ALU are AND, OR and the exclusive or XOR[3]

[2]Overflow bit is set if there was a signed overflow, e.g. $3'b011 + 3'b001 = 3'b100 \Rightarrow 3 + 1 = -4$
[3]The boolean exclusive or operator returns 1 if the boolean inputs are not equal. e.g. for the bitwise XOR: 4'b0101 XOR 4'b1100 = 4'b1001

Shift Operations

There are several shift operations which have to be supported. These operations have in common that they take the input and shift the binary representation observing a rule. Operand 1 is the number which has to be shifted and operand 2 how many positions the number has to be shifted. The first operation is *Shift Left Logical* which shift the bits to the left, bits, which are shifted out, are left out.[4] Consider that a shift of 1 positions to the left is equal to a multiplication with 2. For convenience the *MOVHI* operation is implemented with the Shift Left Logical where the second operand is set to constant 16. There are also bitwise right shifts including a logical and an arithmetical variant considering the fact that unsingend or signed numbers have to be shifted in a different way. *Shift Right Logical* shifts every bit some positions to the right dependent on the second operand. The Most Significant Bits (MSB) bits of the result are filled up with zeros such that the result is an unsigned integer division by a power of two. The signed alternative *Shift Right Arithmetical* fills the MSB with the sign bit[5] which is equivalent to a signed integer division. The last shift operation is called *Rotate Right Logical* which shifts the bits to the right. In contrast to the other right shift operators the MSB bits are filled up with the disappearing LSB bits such that a circular ring arises with invariant bit with regard to the right shift operator.[6] This operation can be built up with a combination of logical left and right shift operators. First all bits are shifted to the right and ored with the opposite left shifted bits. This can be formulated by equation 3.1. The block diagram of the shift operations is shown in Figure 3.8.

$$(Op1 << (32 - Op2)) \,|\, (Op1 >> Op2) \tag{3.1}$$

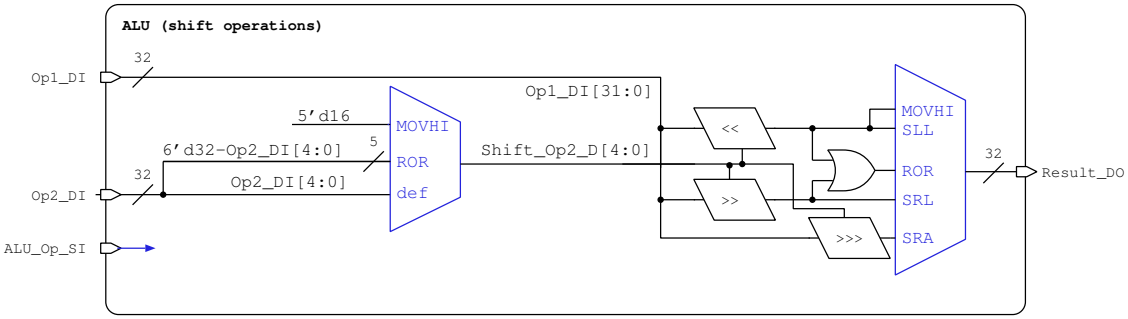


Figure 3.8.: ALU, shift operations.

Extension Operations

Extension operations take a number of different bit-length and converts it to a 32 bit wide integer. The conversion takes into account whether the number has to be interpreted as signed or unsigned number. Signed number are extended with the sign bit (sign-extend)

[4] The result's Least Significant Bits (LSB) bits are filled up with 0 e.g. $4'b1001 << 2 = 4'b0100$

[5] Shift Right Logical: $4'b1010 >> 2 = 4'b0010 \triangleq 10 >> 2 = 2$
Shift Right Arithmetical: $4'b1010 >>> 2 = 4'b1110 \triangleq -6 >>> 2 = -2$

[6] e.g. $8'b1011\ 1010\ ror\ 3 = 8'b0101\ 0111$

and the unsigned with zero (zero-extend). There are *EXTBZ* and *EXTBS* which extend bytes (8 bits) to words (32 bits), *EXTHZ* and *EXTHS* extend half word (16 Bits) to words (32 Bits). The word extension operations *EXTWZ* and *EXTWS* are implemented for reasons of compatibility which extend word wide number to double word in 64 bit architecture, but return operand 1 unchanged in 32 bit architecture.

Comparison Operations
The comparison operations compare operand 1 and operand 2. With the help of the three operators unsigned and signed "Is Greater As" and equality comparator a large set of comparison operators are implemented. There are always a signed and an unsigned version of each comparison operator. The operations are "Is Equal To" ($Op1 == Op2$), "Is Not Equal To" ($Op1 \neq Op2$), "Is Greater Than" ($Op1 > Op2$), "Is Greater Than Or Equal To" ($Op1 \geq Op2$), "Is Less Than" ($Op1 < Op2$) and "Is Less Than Or Equal To" ($Op1 \leq Op2$). Consider Figure 3.9 which illustrates how these operations are calculated. The one bit long result is finally zero extended.

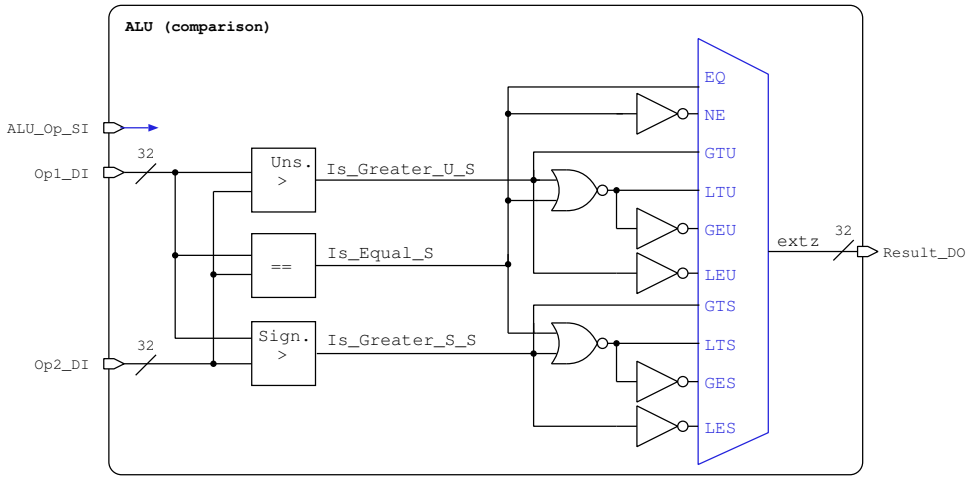


Figure 3.9.: ALU, comparison operations.

3.5. Memory & Wrapper

As mentioned in Section 3.1, we have two memories namely an instruction memory and a data memory. Both are embedded in a wrapper in order to simplify the usage for the processor.

The actual memory modules used are 4 KB in size and have 32-bit inputs and outputs, where the inputs are byte-write enabled in order to work as intended with our CPU.

For the instruction memory, the access for the CPU is simple, because we always want to read the whole 32-bit word as our instruction. But we also want to be able to store

a new program in the memory and because of our 8-bit interface, we need some input logic to write these bytes at the correct location.

In the data memory wrapper there is more to do, because the CPU needs the ability to store and load words, half-words and bytes. Last but not least we also want to be able to read out both, the data and the instruction memory. Therefore, we need an 8-bit output interface, where this logic is already present for the data memory because of the load byte operation.

3.6. Multiplier

In order to perform calculations efficiently, we decided to do multiplications on a separate unit in parallel to the ALU. But because a 32-bit multiplication takes about twice as long as other operations, we have to run it for more than one clock cycle. One design decision was therefore, how many cycles were optimal and how to implement this solution.

Our first attempt was to use FloPoCo[7], a generator of arithmetic cores (**Flo**ating-**Po**int **Co**res) which allowed us to generate Very High Speed Integrated Circuit Hardware Description Language (VHDL) modules with the desired characteristics. But because FloPoCo is optimized for Field Programmable Gate Arrays (FPGAs), the results were not suitable for our project.

By now we figured out that two clock cycles are enough for the multiplication to complete and we found two other possibilities how to achieve it: we could either use a DesignWare multiplier from Synopsys or try retiming of a standard multiplication. This works as follows: The multiplication is done as usual, but the result is not directly routed to the output, it is stored in a register. During the design compilation we then have the possibility to tell the compiler to take the whole multiplier and retime it, which means the register gets moved towards the middle of the multiplication, where we want it to be. The commands used for this are the following:

```
set_optimize_registers -designs mult
optimize_registers -only_attributed_designs
```

[7] `http://flopoco.gforge.inria.fr`

These were the results of the compilation for both possibilities with a clock period of 1.5 ns:

Table 3.6.: Comparison of DesignWare vs. Retiming multiplier

	DesignWare	Retiming
Input → Register	1.64 ns	1.70 ns
Register → Result	1.21 ns	1.16 ns
Register → Overflow	1.74 ns	1.75 ns
Area	133'713 μm^2	157'208 μm^2

As we can see in Table 3.6, the DesignWare multiplier is slightly faster than the retiming variant. But the difference is small and because the commercial virtual component does not fit the open source goal, we decided to go with the retiming multiplier anyways.

In order to increase the efficiency of the multiplication, one important goal was not to waste two whole cycles for each multiplication, but to run as many operations in parallel as possible.
For two multiplications one after the other, it is clear that they can run independently because they behave like normal operations in the pipeline and therefore they do not stall the CPU.
Operations on the ALU can follow the multiplication if they fulfill the following conditions: they do not write to the general or special-purpose registers and they do not require the result of the ongoing multiplication. If the operation after the multiplication would try to write to a register, a data conflict would occur as one can see in Figure 3.10.

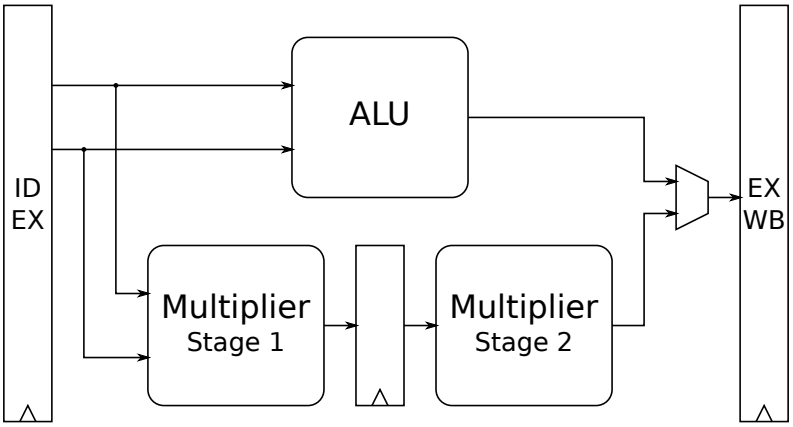


Figure 3.10.: Two-stage multiplier in parallel to the ALU.

3.7. Stalls

Even though with forwarding (see Section 3.2.2) most hazards can be avoided, there are still some cases, where data is not available when needed. One of those cases is the load operation where the data memory adds an additional latency to the data path and therefore we have to wait until the loaded data is finally available. For those cases we have to stall the entire CPU pipeline, i.e. stop the data flow for one cycle and continue execution in the next one. In the load operation case, we have to stall every stage of the pipeline, because the needed data gets inserted at the end of the pipeline and continuation of the previous stages would overwrite this data or vice versa.
Other cases where the whole pipeline has to be stalled are data or instruction cache misses, where we have the same problem as above.

There are also some cases where the later stages of the pipeline can continue with its operations, but the former ones have to stall. For example if there is a multiplication running and the following instructions requires its result, it has to wait due to the two-cycle multiplier. Or when an operation wants to write to some register directly after a multiplication operation, it has to wait too, because the results of the ALU and the multiplier would interfere with each other if there was no stall.

Slightly less sophisticated stalls are the ones from operations affecting the special-purpose register. These operations are rare and therefore each register write process (l.mtspr) causes a stall, because ALU operations could mess up the storing mechanism (overflow or carry). Reading from the special-purpose register (l.mfspr) causes a stall if the following operation requires the result of this operation, because it adds another latency cycle to the operation.

Chapter 4

Design Implementation

This chapter gives an overview of the implemented functionality and ASIC key data. These are the implemented instructions, timing, area, power and floorplaning.

4.1. Instruction Table

The following Table 4.1 gives an overview of the implementented ORBIS32 instructions. See Table D.2 for more detailled information about the functionality of each instruction and consider Table 0.1 for naming conventions.

Table 4.1.: Instructions

Instruction	Parameter	Description
l.bf	N	Branch if Flag
l.bnf	N	Branch if not Flag
l.j	N	Jump Immediate
l.jal	N	Jump and Link
l.nop	K	No Operation
l.jalr	rB	Jump and Link Register
l.jr	rB	Jump Register
l.movhi	rD, K	Move High
l.addi	rD, rA, I	Add Immediate
l.addic	rD, rA, I	Add Immediate Carry
l.andi	rD, rA, K	Bitwise And Immediate
l.lbs	rD, I(rA)	Load Byte Signed
l.lbz	rD, I(rA)	Load Byte Unsigned
l.lhs	rD, I(rA)	Load Halfword Signed

Table 4.1: Instructions (contin.)

Instruction	Parameter	Description
l.lhz	rD, I(rA)	Load Halfword Unsigned
l.lws	rD, I(rA)	Load Word Signed
l.lwz	rD, I(rA)	Load Word Unsigned
l.mfspr	rD, rA, K	Move from Special-Purpose Register
l.muli	rD, rA, I	Multiply Immediate
l.ori	rD, rA, K	Bitwise Or Immediate
l.xori	rD, rA, I	Bitwise Exclusive Or Immediate
l.rori	rD, rA, L	Rotate Right Logic Immediate
l.slli	rD, rA, L	Shift Left Logic Immediate
l.srai	rD, rA, L	Shift Right Arithmetic Immediate
l.srli	rD, rA, L	Shift Right Logic Immediate
l.mtspr	rA, rB, K	Move to Special-Purpose Register
l.sb	I(rA), rB	Store Byte
l.sh	I(rA), rB	Store Halfword
l.sw	I(rA), rB	Store Word
l.add	rD, rA, rB	Add
l.addc	rD, rA, rB	Add Carry
l.and	rD, rA, rB	Bitwise And
l.or	rD, rA, rB	Bitwise Or
l.sll	rD, rA, rB	Shift Left Logic
l.sra	rD, rA, rB	Shift Right Arithmetic
l.srl	rD, rA, rB	Shift Right Logic
l.sub	rD, rA, rB	Subtract
l.xor	rD, rA, rB	Bitwise Exclusive Or
l.mul	rD, rA, rB	Multiply
l.mulu	rD, rA, rB	Multiply Unsigned
l.muld	rA, rB	Multiply Double-Word
l.muldu	rA, rB	Multiply Double-Word Unsigned
l.sfeq	rA, rB	Set Flag if Equal
l.sfges	rA, rB	Set Flag if Greater Equal Signed
l.sfgeu	rA, rB	Set Flag if Greater Equal Unsigned
l.sfgts	rA, rB	Set Flag if Greater than Signed
l.sfgtu	rA, rB	Set Flag if Greater than Unsigned
l.sfles	rA, rB	Set Flag if Less Equal Signed
l.sfleu	rA, rB	Set Flag if Less Equal Unsigned
l.sflts	rA, rB	Set Flag if Less than Signed
l.sfltu	rA, rB	Set Flag if Less than Unsigned
l.sfne	rA, rB	Set Flag if not Equal
l.sfeqi	rA, I	Set Flag if Equal Immediate
l.sfgesi	rA, I	Set Flag if Greater Equal Signed Immediate

Table 4.1: Instructions (contin.)

Instruction	Parameter	Description
l.sfgeui	rA, I	Set Flag if Greater Equal Unsigned Immediate
l.sfgtsi	rA, I	Set Flag if Greater than Signed Immediate
l.sfgtui	rA, I	Set Flag if Greater than Unsigned Immediate
l.sflesi	rA, I	Set Flag if Less Equal Signed Immediate
l.sfleui	rA, I	Set Flag if Less Equal Unsigned Immediate
l.sfltsi	rA, I	Set Flag if Less than Signed Immediate
l.sfltui	rA, I	Set Flag if Less thanb Unsigned Immediate
l.sfnei	rA, I	Set Flag if not Equal Signed Immediate
l.extbs	rD, rA, rB	Extend Byte Signed
l.extbz	rD, rA, rB	Extend Byte Unsigned
l.exths	rD, rA, rB	Extend Half-Word Signed
l.exthz	rD, rA, rB	Extend Half-Word Unsigned
l.extws	rD, rA, rB	Extend Word Signed
l.extwz	rD, rA, rB	Extend Word Unsigned
l.eoc (cust1)		Set "End of Computation" (Custom Instruction 1)

4.2. Area

The colored Figure 4.1 and Table 4.2 give an overview of the occupied area. The multiplier is 19 % of the required area. This is due to the fact that it is a multiplier with long operator sizes (33x33 signed) and the timing constraints are ambitious for only two pipeline stages. In fact it turned out that on the Sir10us chip[1] the multiplier is only about $74'000\mu m^2$ in comparison to $159'000\mu m^2$ only because the timing constraints are more lax and two stages were kept.

[1] Sir10us is an additional chip we have done with another group. See Section 6.2 for more information.

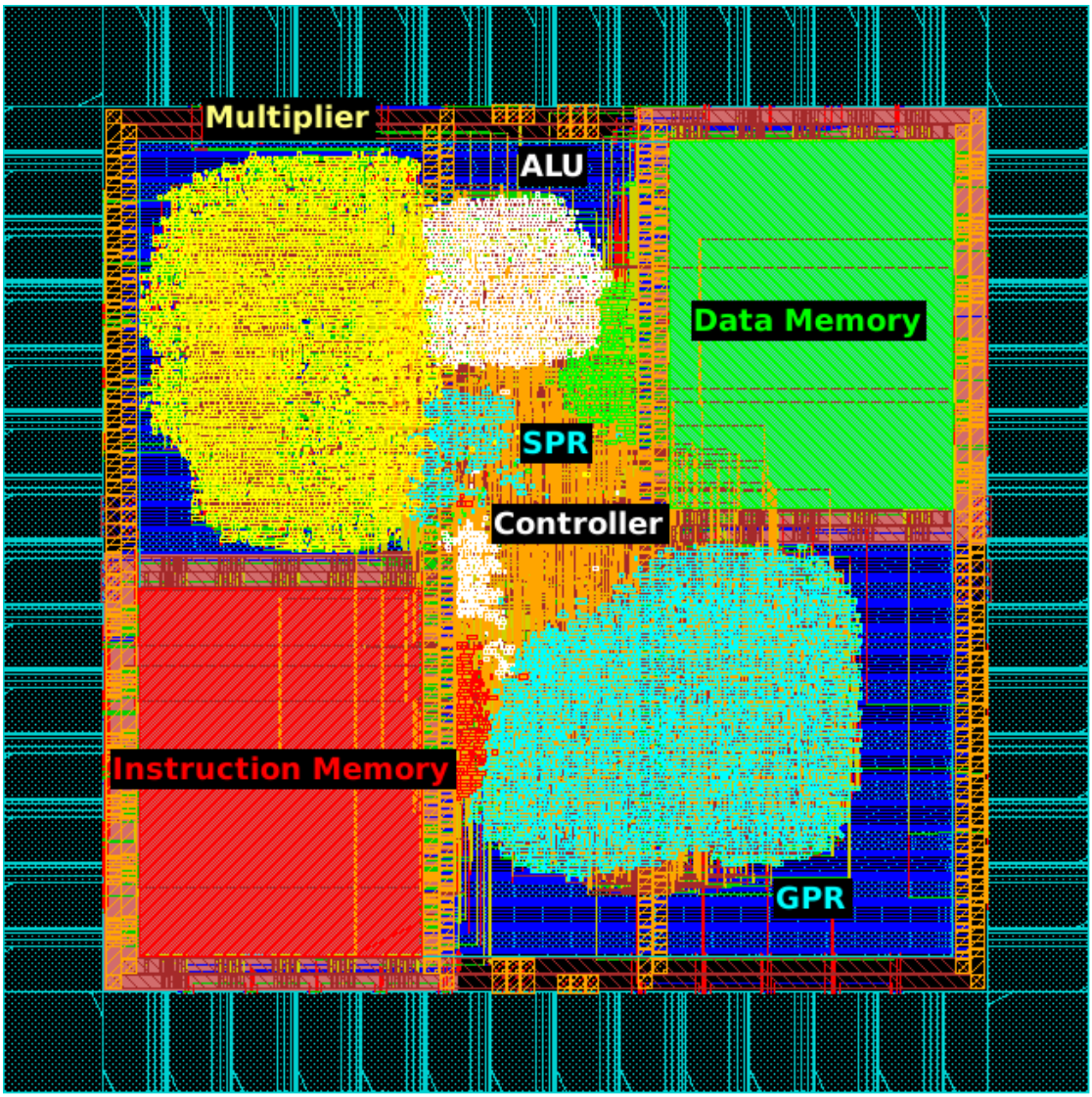


Figure 4.1.: Area Distribution

Table 4.2.: Area distribution

Name	Area [GE]	Cells	Area [μm^2]	A/A_{tot}	Core Area
chip	89174	15742	1′607′457	100.0 %	-
– pads and bondings	-	-	771′501	-	-
– top	88592	15601	830′497	99.3 %	68.7 %
— cpu	44393	15218	416′160	49.8 %	34.4 %
—– alu	5717	2283	53′597	6.4 %	4.4 %
—– controller	482	273	4′515	0.6 %	0.4 %
—– mult	16930	6192	158′709	19.0 %	13.1 %
—– registers	13797	4037	129′341	15.5 %	10.7 %
—– sp registers	1133	267	10′621	1.3 %	0.9 %
— instruction memory	21753	58	203′929	24.4 %	16.9 %
— data memory	22260	257	208′676	25.0 %	17.3 %

4.3. Timing

As we have seen in Section 3.2.1 - Pipelining is optimally used if the longest path of all pipeline stages are balanced. In Table 4.3 all longest paths of the four pipeline stages IF, ID, EX, WB and the two multiplier stages are listed. Detailled timinig diagram is shown in Figure C.3 where all paths between the pipeline stages are illustrated in an abstract way. The main stages are the same, but the Instruction Fetch stage is shorter than the others. Combining with ID or WB is not a feasible solution because they are already long enough.

Table 4.3.: Longest Paths obtained with Synopsys

Pipeline Stage	t_{pd}	t_{su}	Σt
Instruction Fetch	2.28 ns	0.12 ns	2.40 ns
Instruction Decode	2.54 ns	0.19 ns	2.73 ns
Execute	2.55 ns	0.19 ns	2.74 ns
- Execute ALU	2.55 ns	0.19 ns	2.74 ns
- Execute Mult	2.44 ns	0.17 ns	2.61 ns
- Execute Mult 2	2.43 ns	0.13 ns	2.56 ns
Write Back	2.55 ns	0.19 ns	2.74 ns
Maximum	2.59 ns		2.74 ns
Mean	2.48 ns		2.63 ns
Standard Deviation	0.15 ns		0.13 ns

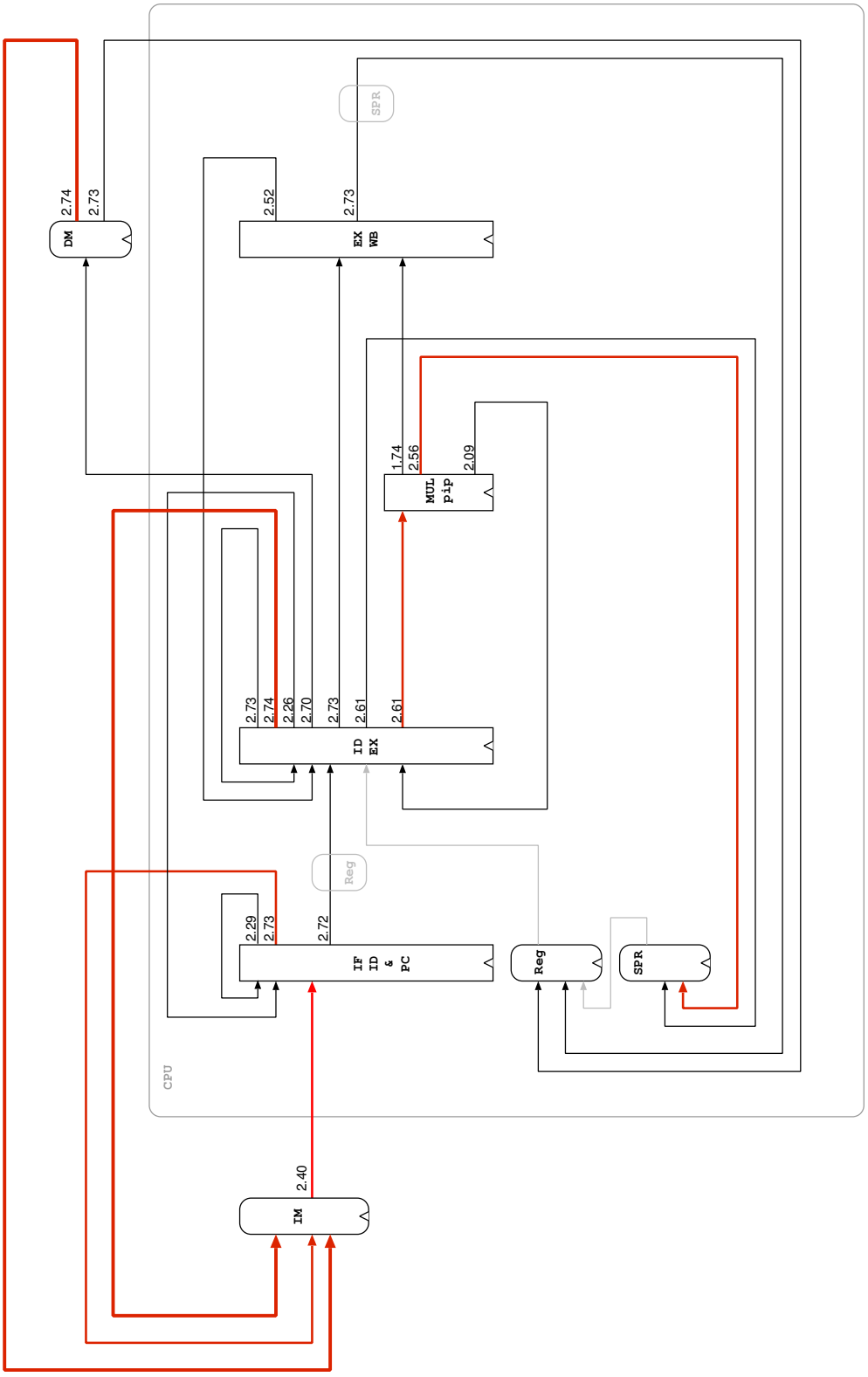


Figure 4.2.: Timing Diagram

4.4. Power

IR drops due to excessive resistance usually due to long metal lines may slow down the CMOS transistors, or even result in misinterpreted data. Simulations with SoC Encounter showed that the voltage drop is marginal in respect to the operating region of the used 180nm process. (See Table C.8) In the worst case it was only 10 mV, see Table 4.4 for more details.

Electromigration is anoher topic to consider. If the maximal current density exceeds some value metal ions are washed away in a long time process resulting in a malfunctioning chip. The power simulation showed that this is not the case and only 37 % of the maximal current density was reached.

Power consumption can be found on Table 4.4. Note: The missing number in read out mode are due to the fact that the calculated results are self-tested in software and only the boolean check result is read out and therefore not representative.

Table 4.4.: Power

program	P_{SETUP}	P_{RUN}	P_{OUT}	$V_{cc_{min}}$	I_{max}	$\frac{J}{J_{max}}$
full_coverage.S	53.9 mW	166.7 mW	64.8 mW	1.7943 V	6.97 mA	22 %
matrixMul.c	41.6 mW	161.2 mW	-	1.7905 V	11.42 mA	35 %
matrixAdd.c	43.5 mW	162.2 mW	-	1.7900 V	12.02 mA	37 %
fibonacci.c	44.5 mW	171.5 mW	-	1.7902 V	11.83 mA	35 %
mean/min/max	45.9 mW	165.4 mW	64.8 mW	1.7900 V	12.02 mA	37 %

Chapter 5

Verification and Testing

Testing is a substantial part in ASIC design. In comparison to FPGA design, where faults can be corrected later, this is not possible on ASICs because wire and gate placement is fixed after tape-out. This leads to a right the first time policy and a high testing effort during all design stages. But also faults in production may lead to faulty chips. Therefore, all procuded chips have to be tested efficiently to detect as many as possible of these faulty chips to avoid selling these faulty chips[5].

5.1. Testbench and Testing interface

In a usual test setup, stimuli are applied first and after or during execution responses are read and compared to expected responses[6]. This testing scheme was an inspiration for ours, but could not have been adapted directly because a processor behaves in an other way. The used test scheme is illustrated in Figure 5.1 and explained in the following sections.

5.1.1. Stimuli application, SETUP mode

In the SETUP mode instructions and data have to be loaded into the instruction and data memory, respectively. Therefore a stimuli file is read. Each stimuli entry consists of the byte address (first 13 bits[1]) and one word (32 bit instruction/ 32 bit data). Due to the small number of pins the data is then applied bytewise with the according byte address. See Ch. C.6 for more information about memory addressing.

[1] This number depends on the memory size. In the OR10N chip there are 8'192 bytes addressable, therefore address length has to be chosen as $l_{adr} = 13 = \log_2 8192$

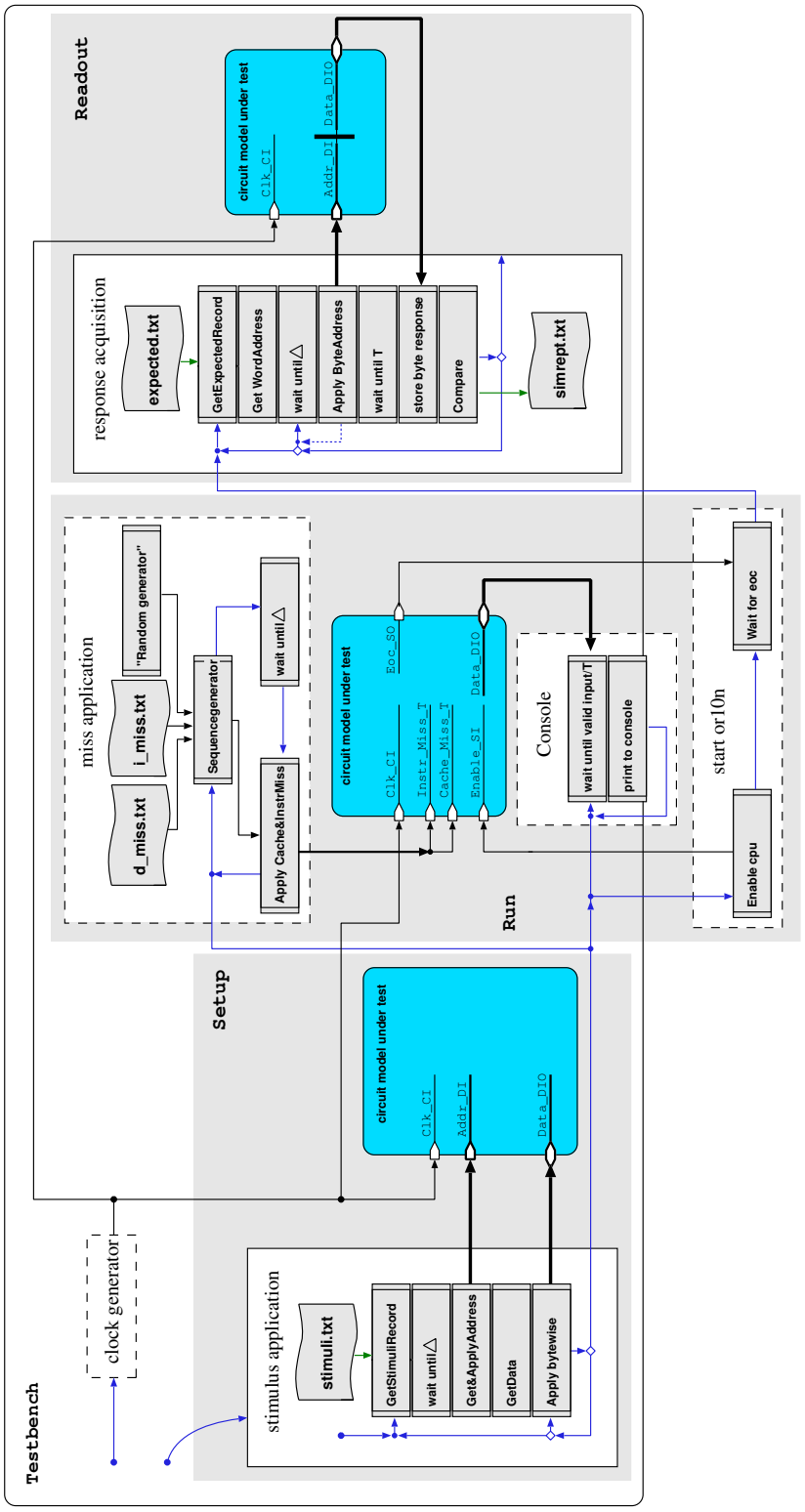


Figure 5.1.: Schema of the testbench

5.1.2. RUN mode

In RUN mode the cpu is enabled and the testbench waits on the end of computation signal. During running, the testbench applies the acknowledgement signal for the data and the instruction memory to simulate the behaviour of the processor in a multi-core system. This can be either from file or by random generator. See appendix B for more information how to use the testbench.

5.1.3. Response Acquisition, READOUT mode

The third mode checks whether the memory entries are as expected. For this reason a response file is read. Like in the SETUP mode the entries consist of the byte address (pointing to the MSB byte of the word) and the data word. Each byte address of the required word is then applied separately and the data can be read from the the *Data_DIO* output one cycle later and put together to compare the expected word.

5.2. Functional Verification

Functional verification is done with a few chips (prototypes) to check if the chips behave as expected. This includes in the ideal case the testing of the whole set of functionality and states of the chip. Faults detected in this design stage can have already a high economic impact because all chips behave in a wrong way. Therefore verification by simulation before tape-out is very important.

5.2.1. Verification by simulation

The simulation tool used in this thesis is ModelSim of MentorGraphics. Simulation allows us to check the chip functionality during early design steps but also provides the possibility to check if generated stimuli vectors have a high coverage in chip functionality. These stimuli with high coverage are later used to test chips after tape-out. In this thesis we used two possibilities to guarantee a high coverage: Code coverage and covergroups.

Code coverage is a feature of ModelSim which counts during simulation how often every line was executed and which branches were taken. It takes also into consideration whether signals had toggled during simulation. By means of this we developed the programm *full_coverage* which can be found in the directory */sw/testscripts/full_coverage.s*. Table 5.1 shows the coverage of this Assembler program. The relative bad Code Coverage for the top entity arises mainly by the fact that higher address bits (instruction and data memory) have never toggled. This is because we used only 4 kBit memory which can fully be addressed with 12 bits, these signals are cleaned out during synthesis anyway.

Table 5.1.: Code coverage of the programm *full_ coverage.S*

Module	Coverage
| Top	**89.1 %**
| CPU	96.3 %
| ALU	100.0 %
| Multiplier	99.9 %
| Special-Purpose Register	94.5 %
| Registers	99.7 %
| Data Memory	92.0 %
| Instruction Memory	96.3 %

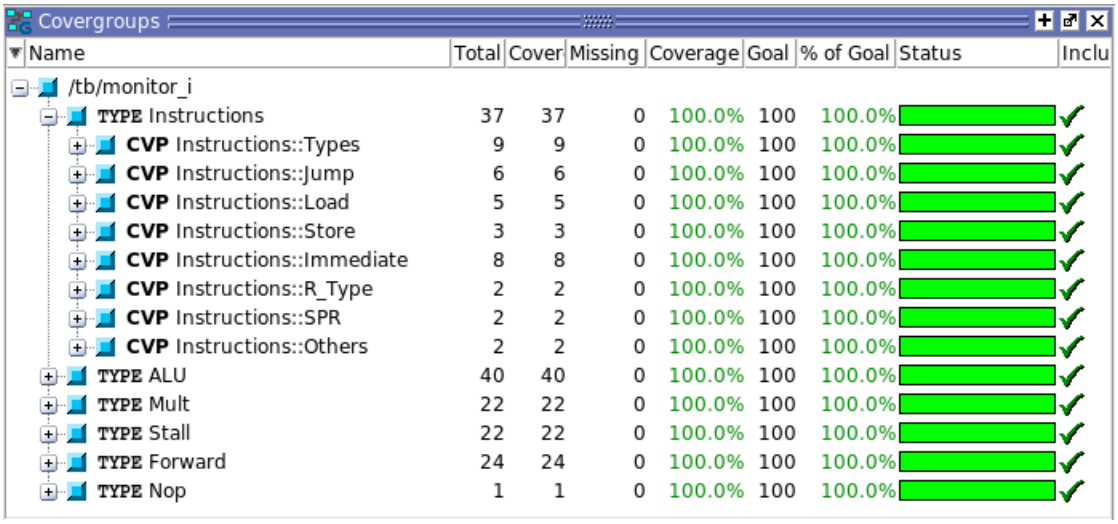


Figure 5.2.: Covergroups for *full_coverage* in Modelsim

Covergroups: Code coverage does already give a good insight in coverage but lacks the opportunity to check whether special critical cases occurr concurrently. Therefore we expanded our coverage analysis with so-called covergroups. Covergroup is a special class which is provided by SystemVerilog but not by VHDL. This class allows to select signals the simulator has to listen for, and to define bins containing signal values which belong together[7]. During simulation runs with enabled coverage option[2], ModelSim counts how often a signal matches the values defined in the bins and monitors them. Table 5.2 shows the covergroups defined for coverage analysis for this thesis and Figure 5.2 shows the resulting coverage analysis for the *full_coverage* program.

[2]To run ModelSim in coverage mode add *-coverage* to *vsim*. More detailled explanation on how to simulate the processor can be found in appendix B.

Table 5.2.: Covergroups

Covergroup	Coverpoint	Goal: Coverage of ...
Instructions	Types	all instruction types. (R-, I-, J-type, ...)
	Jump	all different jump instructions. (l.j, l.bf, ...)
	Load	all different load instructions. (l.lws, l.lbz, ...)
	Store	all differnet store instructions. (l.sw, l.sh, l.sb)
	Immediate	all different immediate instructions. (l.addi, l.movhi, l.ori, ...)
	R-type	ALU and Shift operations. (l.add, l.sll, ...).
	SPR	move to/from Special-Purpose register.
	Others	all other instructions. (l.nop, l.eoc)
ALU	ALU_Op	all ALU operations
	CY_Set	at least one (unsigned) carry overflow.
	OV_Set	at least one signed overflow.
	CompareNFlag	at least one SetFlag to 0 and 1.
	*_Set_CY	carry overflow for l.add, l.addc and l.sub.
	*_Set_OV	signed overflow/underflow for l.add, l.addc and l.sub.
Mult	DoubleWord	at least one double word multiplication.
	Mult_CY	at least one carry overflow multiplication.
	Mult_OV	at least one signed overflow multiplication.
	Mult_Stage1n2	at least one multiplication in both stages.
	Mult_ALU_Conc	at least one multiplication and ALU Operation concurrently.
Stall	lw_Stall	at least one stall due to dependency between load word data and data needed for subsequent instruction.
	CacheMiss	at leat one stall due to a cache miss (data memory).
	InstrMiss	at least one stall due to a cache miss (instruction memory).
	MultStall	at least one stall due to a multiplication.
Forward	EX2ID	at least one data dependency between EX and ID stage.
	WB2ID	at least one data dependency between WB and ID stage.
	EX2IDnWB	at least one data dependency between WB, EX and ID stage.

C programs: Although it is possible to write good Assembler codes, it is hard to write complex programs. OpenCores Community provides a C compiler which allows to translate C programs into Assembly. This allows us to write programs in the high level language C and generate Assembler code which can then ceasily be translated into machine code. Generating expected responses is simplified because such programs can be compiled and run on a Golden Model (computer's cpu). See appendix B for more information.

5.3. Production test

As mentioned in the introduction to this chapter production faults can play a decisive role. The yield[3] deviates mainly depending on circuit complexity and integration scale. Therefore, errornous chips have to be detected before selling. All produced chips have to be checked in reasonable time and all faulty chips should be rejected. There are several good possibilities to guarantee a high fault detection rate, the used Design for Testing (DFT) approaches are mentioned in the following lines.

5.3.1. Scan Chain

High testability needs a high controllability and high observability of as many nodes as possible. If internal complexity rises, these criteria cannot be fullfilled any more due to the lack of (output) pins. In this thesis, 1'712 flip-flops and 84 kGE are used but only 46 pins are available in total. Therefore, all flip-flops are replaced with scan flip-flops. A scan flip-flop has two additional singals: an enable signal (*Test_En_TI*) and a scan input signal. Synopsys then connects these Scan-FF to a shift register. To set up the FF to the test vectors' values, data can be passed through the whole scan chain. The scan enable signal can be driven to low and one clock period later all flip-flops have settled and can be read out the other way round using the scan chain.

Table 5.3.: Number of flip-flops connected to scan chains

Scan Chain 1-2	172
Scan Chain 3-10	171
Non-Scan-FF	0
Total	1'712

[3]Yield defines the ratio of error-free chip to produced chips.

5.3.2. Automated Test Pattern Generation (ATPG)

To get reasonable test vectors we use ATPG supported by Tetramax. First Tetramax builds a so-called fault dictionary where all possible detectable stuck-at[4] faults are collected. With this dictionary Tetramax is able to generate a test pattern which can be used with the available scan chains to detect possible flaws.

5.3.3. Testing the Reset signal

Stuck-at-one (s-a-1) fault somewhere in the (active-low) Reset signal tree can't be detected by the scan chain because the reset signal is not part of the chain. To get rid of these (ATPG) undetectable faults, we will use the reset signal to set all registers to their initial values and read out these values using the chains. Table 5.4 illustrates the consequent fault coverage.

Table 5.4.: Fault coverage with Scan-Chains using ATPG

fault class	**faults**
Detected	99'448
Detected (Rst/s-a-1)	1'712
Possibly detected	116
Undetectable	321
ATPG untestable	1'312
Not detected	2
total faults	102'910
test coverage	**98.30 %**

TetraMAX determined 102'910 nodes which could be affacted by a stuck-at faults. With ATPG TetraMAX was able to generate patterns such that 99'448 of these stuck-at faults are detectable, additional 1'712 are detectable with the initial sequence. The Defect Level (DF)[5] is 0.18 % if a yield of 90 % is assumed.

5.3.4. Testing the memories

The test coverages of the memories are low and it is not possible to improve this directly. Observability and controllability of the combinatoric circuits which connect the memories

[4]Stuck-at faults depend to a fault model where production faults causes the node to keep some value although it may be driven to some other value[5].

[5]The Defect Level ratioes the defective sold chips to all sold chips and can be approximated by the formula $DL = 1 - Y^{1-T}$ where Y is the yield and T the test coverage[5].

with the CPU cannot be fully realized, although a special bypass[6] was designed. For this reason the memories are fully accessible (read/write) from outside during setup mode. Memory testing will be done with well-known methods like walking-zeros or walking-ones.

[6] Inspired by block isolation, we bypassed the memories in Test mode such that the memory wrapper can be tested without being able to check the memories by ATPG directly. Consider schemata in appendix F of the instruction and data memory.

Chapter 6

Results

The goal of this thesis was a redesign of the already existing OpenRISC implementation[1]. This implementation will be called *"original"* in the following chapter. The original implementation has already been optimized and will be refered as the *"optimized"* implementation. Both of these implementations were synthesized with the same setup as our implementation to guarantee for comparable data.

6.1. Throughput Comparison

To compare different processor architectures different performance measures can be used. When comparing different processors the frequency can be compared. This criteria is highly dependent on the number of pipeline stages and does not consider how many and which instruction was executed in each cycle. Therefore, the first performance measure we used is IPC which ratioes the number of effective executed instructions to the number of cycles needed to execute them. An optimal IPC in a single-cycle RISC architecture would be 1, but this value cannot be reached by a pipelined architecture because at the end all final instructions have to pass all pipeline stages: With four stages as we used, there are always three additional cycles to finish execution. "No operation" (l.nop) which are usually injected by compiler[1] are not counted as effective executed instructions and decrease IPC. The main source of a decreased IPC are stalls. Equations 6.1[2] and 6.2

[1] An example why the OpenRISC compiler reasonably adds a l.nop is the delay slot after a jump instruction. The instruction which follows a jump instruction is executed in the so-called delay slot, usually the compiler tries to reorder the operations such that an expendient instruction is put into the delay slot. If this was not possible due to some dependency, the compiler adds a l.nop instead.

[2] $\mathbb{1}$ stands for the indicator function: $\mathbb{1}_{condition} = \begin{cases} 1 & \text{condition fullfilled,} \\ 0 & \text{else.} \end{cases}$

show a formal description of the used IPC calculation.

$$IPC = \frac{\#instructions}{\#cycles} = \frac{\sum_{k=k_{start}}^{k_{end}} \mathbb{1}_{(Instr[k] \neq 'nop' \wedge PC[k] \neq PC[k-1])}}{3 + (k_{end} - k_{start} + 1)} \tag{6.1}$$

$$IPC = \frac{\#instructions}{\#cycles} = \frac{(k_{end} - k_{start} + 1) - \#nop - \#stalls}{3 + (k_{end} - k_{start} + 1)} \tag{6.2}$$

Table 6.1 shows IPC values of OR10N for seven sample programs and Table 6.2 compares the resulting IPC with them from the original and optimized implementation. The sample programs were chosen such that all kinds of operations are covered: These are branch, computation and storage intensive functions.

Table 6.1.: IPC for sample programs on OR10N

program name	#instr.	#nop	#stalls	#cycles	IPC
fullCoverage.S	336	31	20	390	0.8615
matrixMul.c	5'537	10	512	6'062	0.9134
matrixAdd.c	5'341	15	0	5'359	0.9966
stencil.c	1'656	44	111	1'814	0.9129
towerofHanoi.c	88'209	56	2'048	90'316	0.9767
fibonacci.c	3'536	223	10	3'772	0.9374
bubblesort.c	70'704	101	4'950	75'758	0.9333

Table 6.2.: Comparison of IPC

program	original	optimized	OR10N	Improvement	
matrixMul.c	0.5458	0.9996	0.9134	+83.14%	+67.35%
matrixAdd.c	0.6697	0.9960	0.9966	+48.72%	+48.81%
stencil.c	0.6095	0.9572	0.9129	+57.05%	+49.78%
towerofHanoi.c	0.6143	0.9773	0.9767	+59.09%	+58.99%
fibonacci.c	0.5695	0.9101	0.9374	+59.81%	+64.60%
bubblesort.c	0.6484	0.9329	0.9333	+43.88%	+43.94%
Mean ø	0.6095	0.9622	0.9451	+57.86%	+55.04%

As expected the optimized and our implementation are 57 % and 55 % better than the original one. The main reason is the longer latency of the multiplication, which is implemented as a blocking multiplication and lasts four cycles in the original implementation instead of two, which results in more stalls than in the optimized and in our implementation. The comparison between our implementation and the optimized shows some benefit for the optimized implementation. Mainly because multiplications can be more efficiently executed due to aditional write ports at the GPR. This increases the area and setup delay of the GPR.

An even better performance measure is Operations per Seconds (OPS), or MOPS which indicates how many effective executed instructions are executed in one second. This measure can easily be determined with IPC and the maximal clock frequency. Equation 6.3 and 6.4 shows this relation.

$$\Theta_{OPS} = IPC \cdot f_{clk} = \frac{IPC}{T_{clk}} \tag{6.3}$$

$$\Theta_{MOPS} = \Theta_{OPS} \cdot \left(\frac{10^{-6}\ MOPS}{1\ OPS} \right) \tag{6.4}$$

Table 6.3 and diagram 6.1 compare the throughput in MOPS of the three implementations of the sample programs. The white part of the bars indicates the difference of the actual IPC to an optimal IPC of 1. The original architecture is shown in red, the optimized one in blue and the OR10N architecture in green.

Table 6.3.: Throughput comparison in MOPS

program	original	optimized	OR10N	Improvement	
matrixMul.c	184.39	305.69	333.36	+65.78%	+80.79%
matrixAdd.c	226.25	304.59	363.72	+34.62%	+60.76%
stencil.c	205.91	292.72	333.18	+42.16%	+61.80%
towerofHanoi.c	207.53	298.87	356.46	+44.01%	+71.76%
fibonacci.c	192.40	278.32	342.12	+44.66%	+77.82%
bubblesort.c	219.05	285.29	340.62	+30.24%	+55.50%
Mean ø	205.92	294.25	344.91	+42.89%	+67.49%

Due to the higher maximal frequency of the proposed implementation, an even higher throughput is achieved compared to the original implementation. OR10N achieves 17% higher throughput than the optimized implemenation because of the shorter longest path, a comparison of the maximal frequencies and areas are showed in Table 6.4. Section 4.2 shows detailed information about the area distribution of OR10N.

Table 6.4.: Area and frequency comparison

implementation	area		f_{max}	
original	755'608 μm^2	–	337.84 MHz	–
optimized	789'090 μm^2	+4.43 %	305.81 MHz	−9.48 %
OR10N	779'200 μm^2	+3.12 %	364.96 MHz	+8.03 %

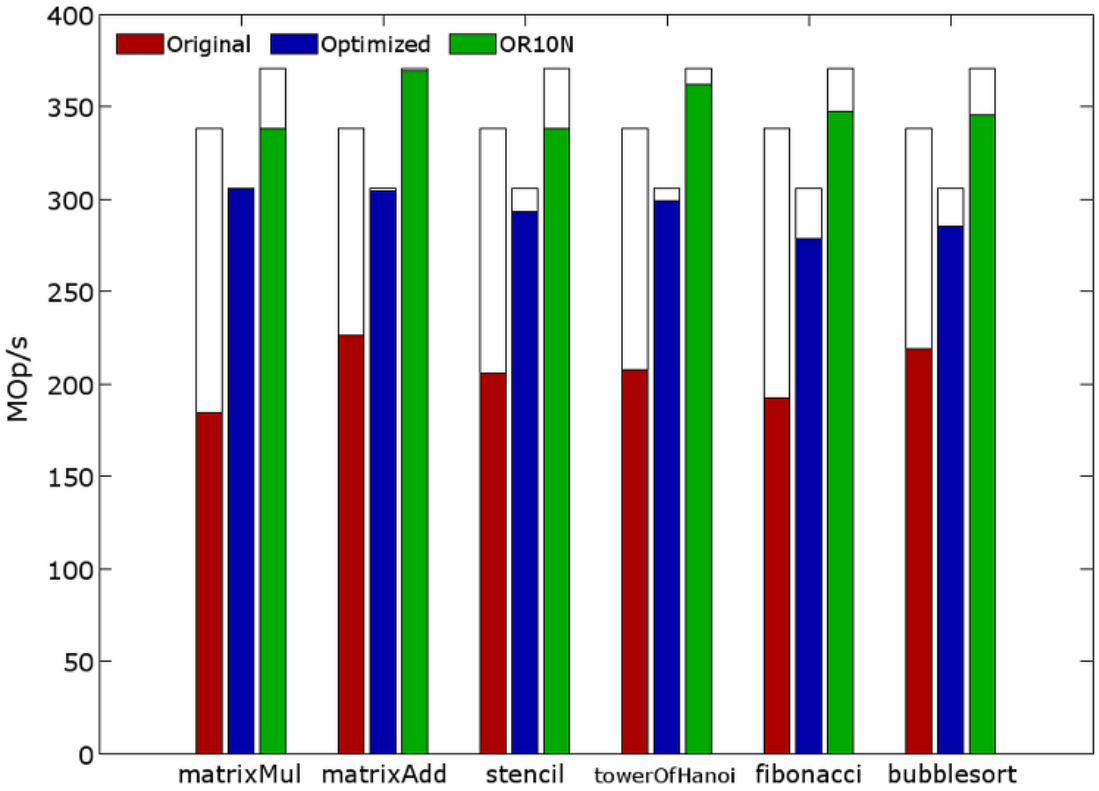


Figure 6.1.: Comparison between OR10N, original and optimized architecture.

6.2. Sir10us Project

As a proof of concept, we also did a collaboration with the semester thesis of Sven Stucki and Andreas Traber, where the idea was to attach an elliptic-curve cryptography accelerator as a co-processor to the CPU. Instead of using a CPU for elliptic-curve cryptography functions, an accelerator is used. This accelerator is attached to a general purpose processor, the OR10N. A combination of dedicated hardware and general-purpose processing offers a high flexibility while maintaining the ASIC's efficiency. Figure 6.2 illustrates how the integration was achieved: The accelerator basically acts like another memory unit and the control can all be done in software using the memory mapped status and command registers.

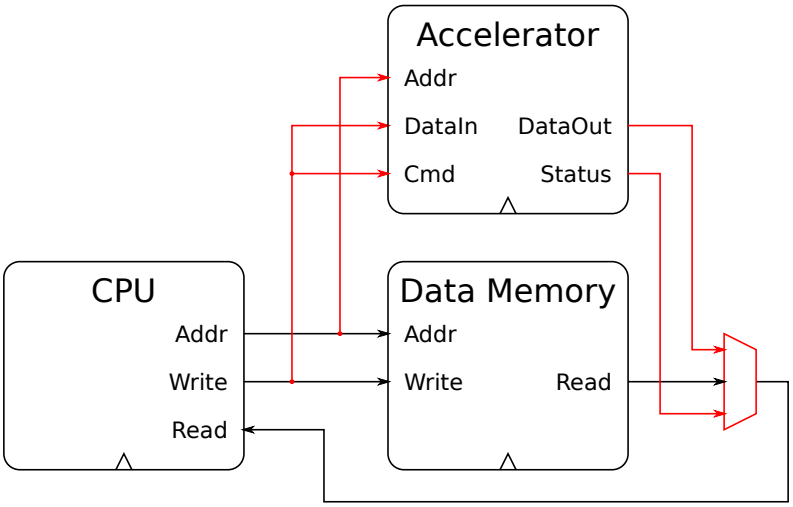


Figure 6.2.: Sir10us: Integration of the ECC co-processor using memory mapped registers.

The overhead of this combined design is very small: The initial phase used to load the data into the accelerator and the cycles used to check the status of the computation are negligible compared to the 35 - 35'000 cycles an accelerator operation takes.
In Figure 6.3 the final layout is presented with a 1 KB internal dual-port RAM for the accelerator, a 2 KB data memory and a 8 KB instruction memory for the CPU.

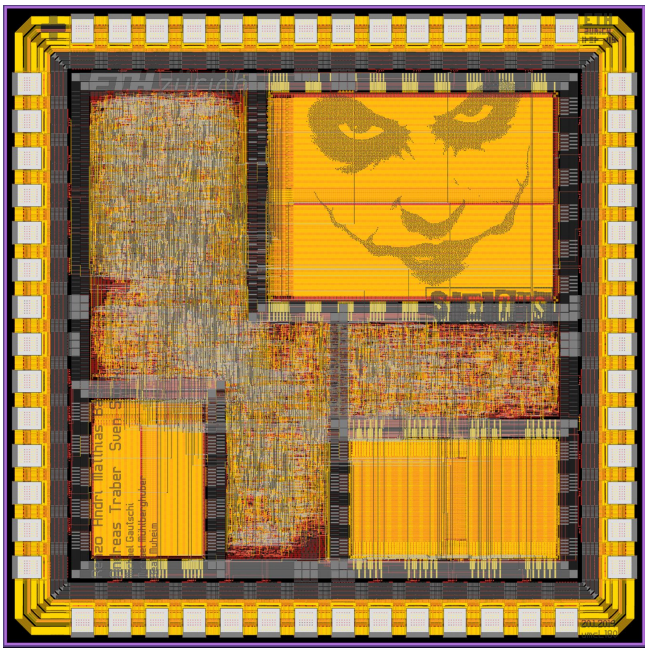


Figure 6.3.: Sir10us Layout

Chapter 7

Conclusion and Future Work

7.1. Conclusion

The current OpenRISC implementation got rewritten in System Verilog with a clean and well documented code and the redesigned core consists of four well balanced pipeline stages. The area of 779'200 μm^2 is only an increase of 3.12% compared to the original implementation. Increasing the throughput to 345 MOPS by increasing the frequency (364.96 MHz) and the IPC (0.945) leads to an advantage of 67.5% compared to the existing OpenRISC implementation.

7.2. Future Work

However, the current implementation is pretty basic and there are many additional components of the OpenRISC architecture which can be added, like exceptions, kernel mode or floating point operations. Hardware loops to accelerate small repeated code sequences or conditional instructions[1] may be additional techniques to further optimize IPC. But the modular structure of the code should allow an easy integration of such components in the future.

As mentioned in the introduction the single core processor will be embedded in a multiprocessor environment.

[1] E.g. used in ARM processors.

Appendix A

Task Description

This appendix presents in the following pages the initial task description.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Semester Project at the Departement of
Information Technology and Electrical Engineering

Autumn Semester 2013

Renzo Andri and Matthias Baer

Design and Implementation of an OpenRISC Processor

December 16, 2013

Advisors: Michael Gautschi (IIS), ETZ J69.2, Tel. +41 44 632 99 58, gautschi@iis.ee.ethz.ch
Frank K. Gürkaynak (DZ), ETZ J60.1, Tel. +41 44 632 27 26, kgf@ee.ethz.ch
Handout: September 16, 2013
Due: December 23, 2013

1 Introduction

1.1 Energy Efficient Computing using Multicore Systems

Ultra low voltage (ULV) computing circuits, where the supply voltage is near the threshold voltage of transistors, have emerged as an attractive approach for ultralow-power (ULP) embedded systems. The goal is to achieve a computing platform which is superior in energy efficiency.

A high energy efficiency in terms of GOPS/mW can be achieved by splitting computation among several clusters. Each cluster comprises several simple processor cores which are operated in ULP region. Such a cluster is shown in Figure 1. In this example four processing elements are used to cover the computational tasks and eight tightly coupled data memories(TCDM) are used as common storage for data, which can be accessed through a low-latency interconnect arbitration unit. Inter-cluster communication and direct memory access(DMA) can be added to improve the functionality.

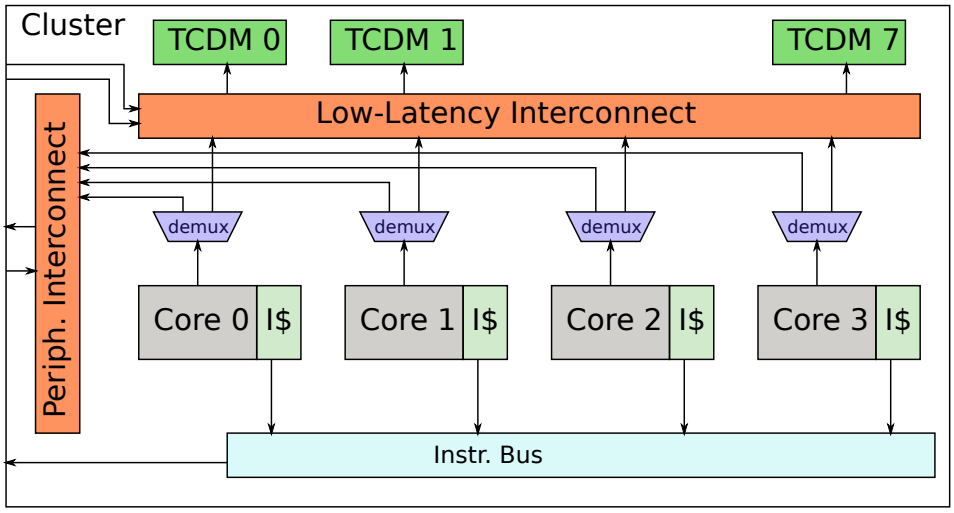


Figure 1: Cluster with 4 processing untis and 8 shared memories.

1.2 OpenRISC Processor

The main component of the cluster in Figure 1 is the processing element, which is typically a reduced instruction set computing(RISC) processor due to its simplicity. One possible candidate is the OpenRISC processor core, an open source RISC processor[3]. A stable implementation of the OpenRISC core, written in verilog exists and can be downloaded. The core consists of a four stage integer pipeline as shown in Figure 2. The four stages are Instruction Fetch(IF), Instruction Decode(ID), Execute(EX) and Write Back(WB). The core supports direct-mapped data and instruction cache, instruction and data management units(I/DMMU), power management units, etc. All this functionality can be enabled or disabled by the user‘s preferences.

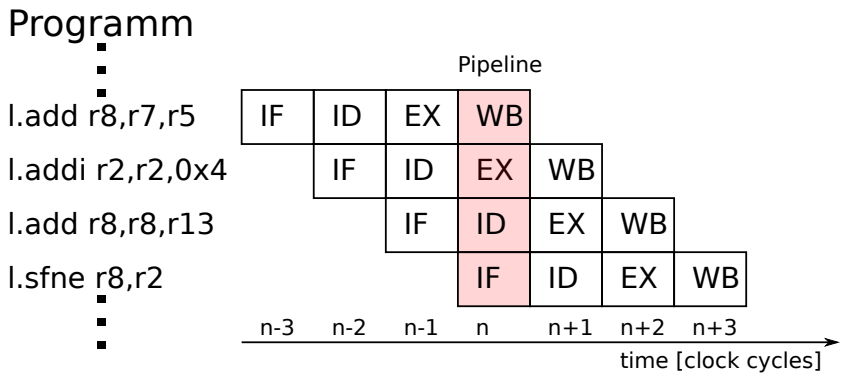


Figure 2: Pipeline of OpenRISC processor.

While beeing functionally correct, the microarchitecture of the core can be improved a lot. The verilog implementation of the core showed poor performance in terms of instructions per cycle(IPC). The reason for the poor performance was mainly due to stalls during load/store operations, stalls during multiplications and poor branch execution. These problems have already been resolved and the IPC as been improved significantly. The microarchitecture of the core is still far from perfect, one main thing which has to be improved is the pipeline itself. In the current implementation, the complete pipeline is stalled instead of one single stage of the pipeline. Another area of interest is the multiplier, which is implemented with a three cycle latency to break the critical path. While this might be optimal in a FPGA setup, it might be possible in an ASIC to perform a multiplication, which is done in the execute stage, in one or two cycles.

2 Project Description

We want to build a clean OpenRISC core which will later be used in the cluster. Throughout this project, the focus is on the implementation and optimization of one single core.

2.1 Configuration

In a single core environement the overall configuration differs from the one of a cluster. There is no need for shared memories, instruction caches or interconnects. A single memory to store data and instructions as illustrated in Figure 3 is enough. Whether this memory will be split in two single port memories to separate instruction and data, or implemented as a dual port memory with a separate, configurable address space for data and instructions will be decided throughout this project.

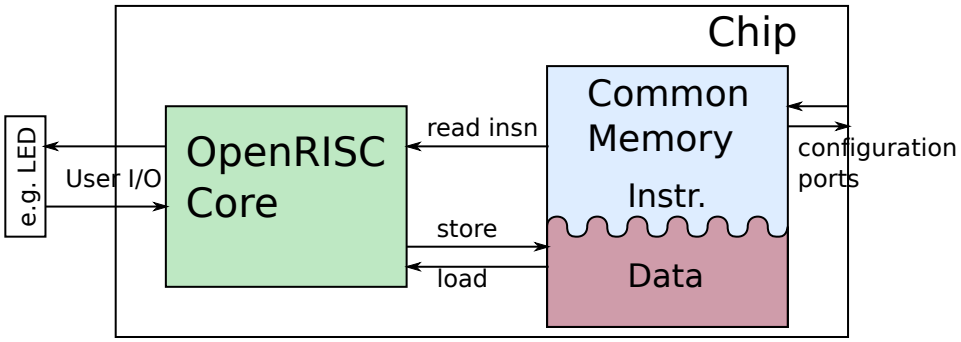


Figure 3: OpenRISC core in a single core environement with a common memory for instructions and data.

2.2 Implementation of the OpenRISC Processor

The original version of the verilog implementation of the OpenRISC core was already modified to improve the core in terms of IPC. This modified version will serve as golden model for your implementation. However, there is a lot of functionality which will never be used in the target system and it is therefore not necessary to maintain all this functionality. Removing caches, IMMU, DMMU, etc. will greatly simplify the design and increase readability of the code.

2.2.1 Clean and well-documented Implementation

The OpenRISC is an open source processor and it is important that the code of the processor is clean and well documented. The pipeline and the signals which influence the pipeline have to be clear and it has to be easy to add additional conditions to modify the behaviour of the pipeline. To obtain a good and clean design it is important to follow coding guidelines, as separation of combinational and sequential processes, using packages to define constants, adhering to naming conventions, no excessive use of defines, which make the code unreadable and the like. To achieve a good code, your supervisors will organize a code review after a few weeks of HDL coding.

2.2.2 Simulation of Instruction Cache and Data Cache Misses

In the future the core is intended to be used in a multicore environment similar to the one shown in Figure 1, where several cores use one or more shared memories. In this case it is possible that more than one core tries to access the same memory at the same time. In this case, at least one processor will have to wait(stall) until the memory becomes available. To simulate such a miss in a single core structure, an acknowledge will be added to each read/store operation. This acknowledge will be assigned in the testbench such that memory collisions can be simulated.

A similar modification has to be added to the instruction fetch unit of the processor to simulate instruction cache misses.

2.3 Design and Definition of a Test Interface

One of the most important things is the test interface. The main problems we will face are the limited number of I/O pins, no flash memory, and the speed. We will not be able to attach the final chip to an external memory, because the pin count will not be sufficient for addresses, 32 bit data and 32 bit instructions. Therefore we will use a structure similar to the one in Figure 3. The configuration ports will be used to initialize the memory with instructions and data. After the configuration is done the processor will start the program and in the end it will output a signal which indicates that the computation has finished. After this signal has been seen the memory can be read through the same configuration ports and compared to the expected responses. Note that for power measurements of the processor it is important to know the start and end point of a program.

2.4 Additional Tasks

Depending on the project progress and the student's interests one or more of the following subtasks can be implemented:

- **Implementation of Hardware Loops:** So far the performance of the core has been analyzed in terms of IPC. Other important measures are the code density and the number of cycles required to run a specific program. Both measures can be improved by replacing for-loops by hardware loops[5]. With hardware loops, the processor does no longer take care of incrementing and comparing loop variables, but uses a separate unit which takes care of this computations instead. Important information about each loop, such as entry and exit points, iteration count, etc. will be stored in a separate LUT, which can be accessed by the hardware loop execution unit. To store those information in the beginning, each loop requires a few initialization cycles. In [5] the authors have shown that the runtime of a benchmark test such as a matrix multiplication can be reduced up to 42% by replacing all the control computation of a loop with powerful hardware loop instructions. We believe that the impact on the OpenRISC core will be similar.
- **Adding User I/O to the Processor:** If the processor is not embedded in a system, it is like a blackbox which computes something and stores the result in the data memory. It will be possible to read the memory through the test interface, but it is also possible to add some specific I/O pins which are directly connected to the processor as illustrated on the left hand side of Figure 3. What the usage of these pins will be is up to the students and will be discussed once the basic functionality of the core is implemented.
- **Pipeline/Processor Optimizations:** Right now the processor is implemented with four pipeline stages. While increasing the number of pipeline stages also increases the overhead of the control and forwarding paths, fewer pipeline stages could even more simplify the design probably at the cost of a slightly increased critical path. We believe that it would be possible to combine the IF and ID stage and therefore reduce the number of pipeline stages to three. The impact on the critical path and complexity of the design will be elaborated in this task.

3 Goals

The goal of this thesis is to implement an OpenRISC processor in System Verilog or VHDL. The following tasks need to be accomplished during this project:

- Good understanding of the OpenRISC core and its implementation.
- Functionally correct, and proper implementation of the OpenRISC core in VHDL or System Verilog with a similar performance to the reference design.
- Developement of an environment to test the basic functionality of the core.
- Well-documented VHDL or System Verilog code.
- Perform a back-end design of the developped processor.
- Hand in a good report and give an interesting presentation.

4 Milestones

The following is a list of expected milestones of the project. Note that milestones 5 and 6 correspond to the additional tasks and have a lower priority then the others.

1. Study the reference design of the OpenRISC core and its implementation.
2. Draw clear block diagrams of the processor and its pipeline. Identify weaknesses and find better solutions if necessary.
3. Implement the basic functionality of the core in VHDL or System Verilog.
4. Define test interface and perform a first back-end run.
5. Study hardware loops, add them to your implementation and derive how much the gain is with enabled hardware loops.
6. Add user specific I/O to the processor according to your preferences.
7. Perform the final back-end design.
8. Write the report and prepare slides for your final presentation.

5 Project Realization

5.1 Project Plan

Within the first month of the project you will be asked to prepare a project plan. This plan should identify the tasks to be performed during the project and sets deadlines for those tasks. The prepared plan will be a topic of discussion of the first week's meeting between you and your advisors. Note that the project plan should be updated constantly depending on the project's status. Figure 4 illustrates a preliminary project plan which may serve as a starting point for you. Furthermore, it already highlights some important events of your project.

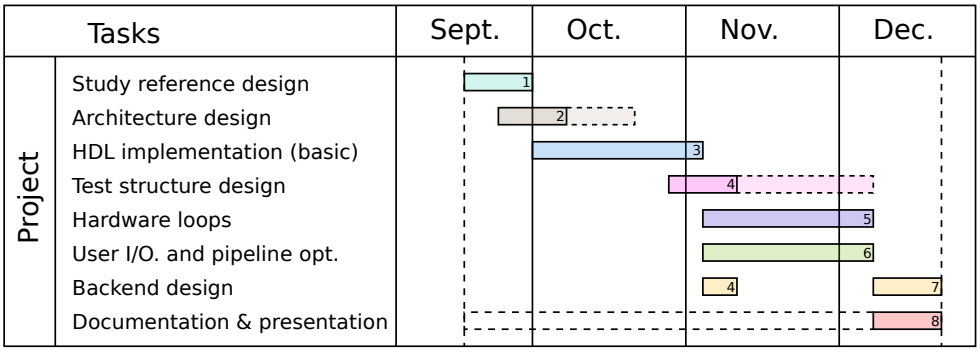


Figure 4: Preliminary project plan.

5.2 Meetings

Weekly meetings will be held between the student and the assistants. The exact time and location of these meetings will be determined within the first week of the project in order to fit the students and the assistants schedule. These meetings will be used to evaluate the status and progress of the project. Beside these regular meetings, additional meetings can be organized to address urgent issues as well.

5.3 VHDL/Syestem Verilog Guidelines

In the following some guidelines for coding HDL are given:

- **Naming Conventions:** Adapting a consistent naming scheme is one of the most important steps in order to make your code easy to understand. If signals, processes, and entities are always named the same way, any inconsistency can be detected easier. Moreover, if a design group shares the same naming convention, all members would immediately *feel at home* with each others code. The Microelectronics Design Zentrum [1] already proposed naming conventions for VHDL and is currently adopting them to System Verilog. Thus, try to maintain these naming convention in order to create readable and maintainable code.

- **Emacs Editor:** The preferred editor for writing HDL, either VHDL or System Verilog code, is the **emacs** editor, as it has a really advanced VHDL and System Verilog mode. Because of this, you should get comfortable with the idea of using emacs, even if you like a different editor.

5.4 Report

Documentation is an important and often overlooked aspect of engineering. One final report has to be completed within this project.

The common language of engineering is de facto English. Therefore, the final report of the work is preferred to be written in English. Any form of word processing software is allowed for writing the reports, nevertheless the use of LaTeX with Tgif[1] or any other vector drawing software such as inkscape (for block diagrams) is strongly encouraged by the IIS staff.

Final Report The final report has to be presented at the end of the project and a digital as well as one printed copy need to be handed in and remain property of the IIS. This report is only accepted if the keys for the student working room have been properly returned. Note that this task description is part of your report and has to be attached to your final report.

5.5 Design Review

Since the design developed throughout this project is supposed to be manufactured using the UMC 180 nm semi-conductor technology, a review of the chip design will be held during late November. The exact date of the review will be determined a few weeks in advance.

5.6 Presentation

There will be a presentation (15 min presentation and 5 min Q&A) at the end of this project in order to present your results to a wider audience. The exact date will be determined towards the end of the work (most probable on the last Thursday before Christmas).

6 Deliverables

Throughout the project, the following deliverables have to be submitted in order to finish the work successfully:

- Project plan
- Report
- HDL implementation
- Back-annotated netlist of final ASIC

[1] Tgif is a simple vector drawing software, quite useful for drawing block diagrams. For further information about Tgif we refer to `http://bourbon.usc.edu:8001/tgif/index.html` and `http://www.dz.ee.ethz.ch/en/information/how-to/drawing-schematics.html`.

References

[1] Design Zentrum website: `http://www.dz.ee.ethz.ch` and VHDL naming conventions: `http://www.dz.ee.ethz.ch/en/information/hdl-help/vhdl-naming-conventions.html`

[2] H. Kaeslin. "Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication". *Cambridge University Press*, 2008.

[3] OpenCores website: `http://opencores.org/or1k/Main_Page` and a tutorial about the OpenRISC core: `http://kth.diva-portal.org/smash/get/diva2:458625/FULLTEXT01`

[4] David Money Harris and Sarah L. Harris. "Digital Design and Computer Architecture". *Elsevier B.V.*, 2007.

[5] Kavvadias N. and Nikolaidis S.. "Elimination of Overhead Operations in Complex Loop Structures for Embedded Microprocessors". *IEEE Transaction on Computers*, 2008.

Zurich, December 16, 2013 Prof. Dr. Luca Benini

The project will not be accepted without returning the keys!

Appendix B

File Structure

The following chapter gives an overview of all project files and a guidance how to program an adequate C or Assembler program, how to use the OpenRISC tool chain, how to generate Stimuli and Expected Responses and how to simulate. Table B.1. shows the directory tree of the main project folder *./or10n/*.

Table B.1.: Directory and File Tree

```
./or10n/
├── README ..... Some general information about the project
├── cockpit.log ..... umcL180 Cockpit Log File
├── calibre ..... MentorGraphics' Calibre folder
│   ├── drc ..... Design Rule Check
│   │   └── *.drc* ..... DRC reports
│   └── lvs ..... Layout versus Schematic
│       ├── chip.lvs.report ..... LVS Report
│       ├── chip.lvs.report.ext ..... Circuit Extraction Report
│       └── verilog2spice ..... Verilog to Spice Conversion Scripts
├── docs ..... Documentations
│   ├── task.pdf ..... Task Description
│   ├── umcL180_*_rule.pdf ..... Design and Layout rules of umcL180
│   └── comparison ..... Comparison between architectures
│       ├── ArchitectureComparison.csv ..... Table with Data
│       ├── area_*.txt ..... Area Reports
│       ├── comp.m ..... Matlab Script to generate Comparison Diagram
│       ├── comp_table.csv ..... Table with MOPS for comp.m
│       ├── results.png ..... Comparison Diagram
│       └── timing_*.txt ..... Timing Reports
```

- presentation ..... Presentation files
  - OR10N.key ..... Original Keynote presentation
  - OR10N.ppt ..... Presentation as Powerpoint file
  - OR10N.pdf ..... Presentation as PDF
- report ..... Report Files
  - Makefile ..... Makefile for Report Files
  - README ..... Some Comments about the Report
  - report.pdf ..... Final Report File
  - report.tex ..... Main LaTeX File
  - content ..... All LaTeX Content Files
  - figures ..... All included Figures
- schematics ..... Schematics of the Project
- encounter ..... Cadence's Encounter Files
  - reports ..... Encounter Report Files
    - power ..... Encounter Power Analysis Files
    - rail ..... Encounter Rail Analysis Files
  - timingReports ..... Encounter Timing Report Files
  - src ..... Source Files for the Chip
  - scripts ..... Scripts for Building the Chip
  - save ..... Saves from the Different Stages of Building
  - out ..... The Final Encounter Output
- modelsim ..... Mentor Graphics' ModelSim Files
  - gate ..... Gate-Level Working Folder
  - reports ..... IPC reports
  - scripts ..... Scripts
    - compile.do ..... Normal Compile Script
    - compile_gate.sh ..... Gate-Level Compile Script
    - sim_postlayout.sh ..... Start Script for Gate-Level Simulation
    - *.do ..... ModelSim Wave Files
    - work ..... Main Working Folder
  - vcd ..... VCD Files
  - modelsim.ini ..... Library Paths
- setup ..... Setup Scripts
  - setup.csh ..... Setup Script
- simvectors ..... Simulation vectors
  - * ..... Programs
    - exp.txt ..... Expected Responses File
    - stim.txt ..... Stimuli File
  - misses ..... Miss Files
    - dataMiss.txt ..... Data Memory Miss File
    - instrMiss.txt ..... Instruction Memory Miss File
- sourcecode ..... Source Code
  - includes ..... Include Files

```
    │   └── defines.sv ..... Definition File
    ├── alu.sv ..... Arithmetic Logic Unit
    ├── chip.sv ..... Chip Entity
    ├── controller.sv ..... Controller
    ├── cpu.sv ..... CPU
    ├── data_mem.sv ..... Data Memory
    ├── instr_mem.sv ..... Instruction Memory
    ├── monitor.sv ..... Covergroup Monitor
    ├── mult.sv ..... Multiplier
    ├── registers.sv ..... General-Purpose Registers
    ├── sp_registers.sv ..... Special-Purpose Registers
    ├── SY180_*_wrapper.vhd ..... Memory Wrapper
    ├── tb.sv ..... Testbench for Normal Simulation
    ├── tb_chip.sv ..... Testbench for Gate-Level Simulation
    ├── top.sv ..... Top Entity
├── sw ..... Software to run on OR10N
    ├── bin2stim ..... Script to generate Stimuli from Binary File
    │   ├── bin2stim ..... Executable of bin2stim.
    │   └── bin2stim.c ..... Source Code
    ├── libs ..... Library functions
    │   ├── string_lib ..... String functions (qprintf)
    │   └── sys_lib ..... System functions (eoc)
    ├── mul2muld ..... Scripts which inserts Double-word multiplication
    │   ├── mul2muld ..... Executable of mul2muld
    │   └── muld2muld.c ..... Source Code
    ├── ref ..... Reference Files
    │   ├── common.mk ..... Makefile of the OpenRISC Tool Chain
    │   ├── cr0.* ..... Assembler Macros
    │   └── link.ld ..... Linker File
    ├── testscripts ..... Assembler Scripts
    │   └── *.S ..... Assembler Programs
    ├── *.bin ..... Binary Files
    ├── *.exe ..... Executables
    ├── *.read ..... Readable Binary File
    ├── *.c ..... Source Codes
    ├── *.h ..... Header Files
    ├── Makefile ..... Makefile of the OpenRISC Tool Chain
├── synopsys ..... Synopsys Files
    ├── reports ..... Synopsys Design Report Files
    ├── scripts ..... Scripts for Compiling the Chip
    ├── DDC ..... DDC Files of the Various Compilation Stages
    ├── netlists ..... Final Netlist of the Chip
├── tetramax ..... Synopsys' TetraMAX Files
```

```
reports ..... TetraMAX Fault Report Files
scripts ..... Scripts for Generating Test Patterns
pattern ..... TetraMAX Test Patterns
```

B.1. Assembler Coding

It is possible to create and compile assembler programs for the OR10N chip in a straightforward way. The assembler is capable of understanding all implemented operations except the `l.muld` and `l.muldu` command.

B.2. Stimuli Generation from Assembler programs

To create an assembler program executable on the OR10N chip, a Makefile in the `sw` directory is used. It links to the general `sw/ref/common.mk` Makefile. Simply put your assembler script in the `sw` folder and execute one of the following commands:

```
make [filename].exe    -- Compiles the *.S program into an executable
make [filename].bin    -- Creates a binary file from the executable
make [filename].stim   -- Converts the binary file into stimuli
make [filename].read   -- Creates a human readable version of the binary
```

B.3. C programing

For C programs the structure is also given, but it is slightly more complicated than the assembler part. In order to address each segment of the code to the correct memory location, there exists a linker file (`sw/ref/link.ld`) that matches the memory sizes used. The way it is set up at the moment, tells the linker to put the program itself into the instruction memory (address `0x0000 - 0x0fff`) and the stack and other data into the data memory (address `0x1000 - 0x1fff`). To put data explicitly into the data memory, one can use the following construct:

```
__attribute__ ((section(".tcdm"))) [type] [name];
```

To initialize the processor (registers, stack, jump to main) an assembler script takes care of this: `sw/ref/crt0.S`.
There are also some library functions that can be used in C programs and one of the most important is the `eoc()` (End of Computation) function. This is used to tell the processor when to finish execution and send a signal to the external interface. One debug

possibility we have is the `qprintf` function which essentially acts like the `printf` function and prints some string to the 8-bit interface of the chip.

B.4. Stimuli Generation from C programs

The same Makefile as for the assembler programs is used to compile and link C programs. It can be used in the exact same way.

B.5. RTL simulation

For compilation of the Register Transfer Level (RTL) model use the following script.

```
cd ./modelsim/
./scripts/compile.do
```

After this step you can start ModelSim with appropriate parameters:

```
vsim-10.2a -lib work -voptargs=+acc [-GPROG=default ...] tb &
```

Table B.2 gives an overview of the most important parameters. If you want to do some coverage analyse use `-coverage` this starts ModelSim in coverage mode where you can check code coverage and covergroups for the chosen program after simulation.

Table B.2.: Parameter for Simulation

param	example	default	Explanation
-coverage			Start simulation in coverage mode.
-GPROG=	matrixMul, ...	default	Program which should be simulated
-GMODE=	default, file	default	Misses are generated pseudo-randomly Misses are read from Miss Files.[1]
-GIMISS=	15, ...	5	Propability of an instruction miss. (def. mode)
-GIMISS=	90, ...	33	Propability of a data miss. (default mode)
-GRUNS=	10, ...	1	Numbers of runs to be executed (default mode)

[1] Misses are read cycle by cycle from the DataMiss.txt and InstrMiss.txt files which can be found and edited in the /or10n/simvectors/ folder.

B.6. Gate-Level simulation

B.6.1. Synopsys Compilation

To perform a gate-level simulation we first need the gate-level netlist from the Synopsys compiler. There are several steps to compile the chip and the scripts at `synopsys/scripts` help to achieve this. To compile the whole chip open the Synopsys command line and run the following command:

```
source all.tcl
```

If you want to perform each step individually, you can also do this by using the same command as above, but instead of `all` use one of the following:

- `1_analyze` - Performs some clean up and analyzes the source code.
- `2_elaborate` - Elaborates the design and checks the design.
- `3_constraints` - Sets some constraints for the compiler.
- `4_compile` - Compiles the design (special handling of multiplier).
- `5_reports` - Writes timing and area reports.
- `6_scanchain` - Setup and insertion of the scan chain.
- `7_netlist` - Finally writes the Verilog netlist.

B.6.2. Cadence Encounter

The next step is to generate the chip layout from the Synopsys netlist. This can be done similar to the previous compilation step. Open up Cadence Encounter and type this command in the console:

```
source all.tcl
```

These steps can be performed individually as well and the corresponding scripts are:

- `1_import_design` - Loads the configuration file and imports the design.
- `2_floorplanning` - Places the memories and adds the power rings/stripes.
- `3_placement` - Performs the placement of the standard cells.
- `4_timing` - Analyzes/optimizes the timing and inserts the clock tree.
- `5_signal_routing` - Routes the signals and optimizes the timing.
- `6_finishing` - Inserts filler cells and generates the output files.

B.6.3. ModelSim Simulation

Now we can perform the actual gate-level simulation with ModelSim. To compile the gate-level, the following commands can be executed in a terminal:

```
cd ./modelsim/
./scripts/compile_gate.sh
```

ModelSim can be started as follows:

```
./scripts/sim_postlayout.sh default &
```

The `default` parameter can be replaced by the program that should be run.

Appendix C

ASIC Datasheet (OR10N)

OR10N is an ASIC implementation of the OpenRISC architecture which was designed by Matthias Baer and Renzo Andri as a Semester Thesis at the IIS at ETH Zurich.

C.1. Features

- OpenRISC 1200 32-Bit architecture.
- 72 ORBIS32 instructions. (see Table 4.1 and D.2)
- Additional instruction for End of Computation. (l.cust1)
- 33-bit signed (hardware) multiplier.
- Debugging interface. Messages can be printed character-wise on Data Output.
- Test interface for cache misses.
- Full Scan-Chain (Flaw detection)

C.2. Packaging

Package QFN56 is used with 16 power pins and 40 user pins.

C.3. Bonding Diagram

The 16 power pins and the clock pin are arranged the same way as it is on the Standard QFN56 tester board. See figures C.1 and C.2 for more details. Pin 4-6 and 9-13 (*Data_ DIO*) are bidirectional.

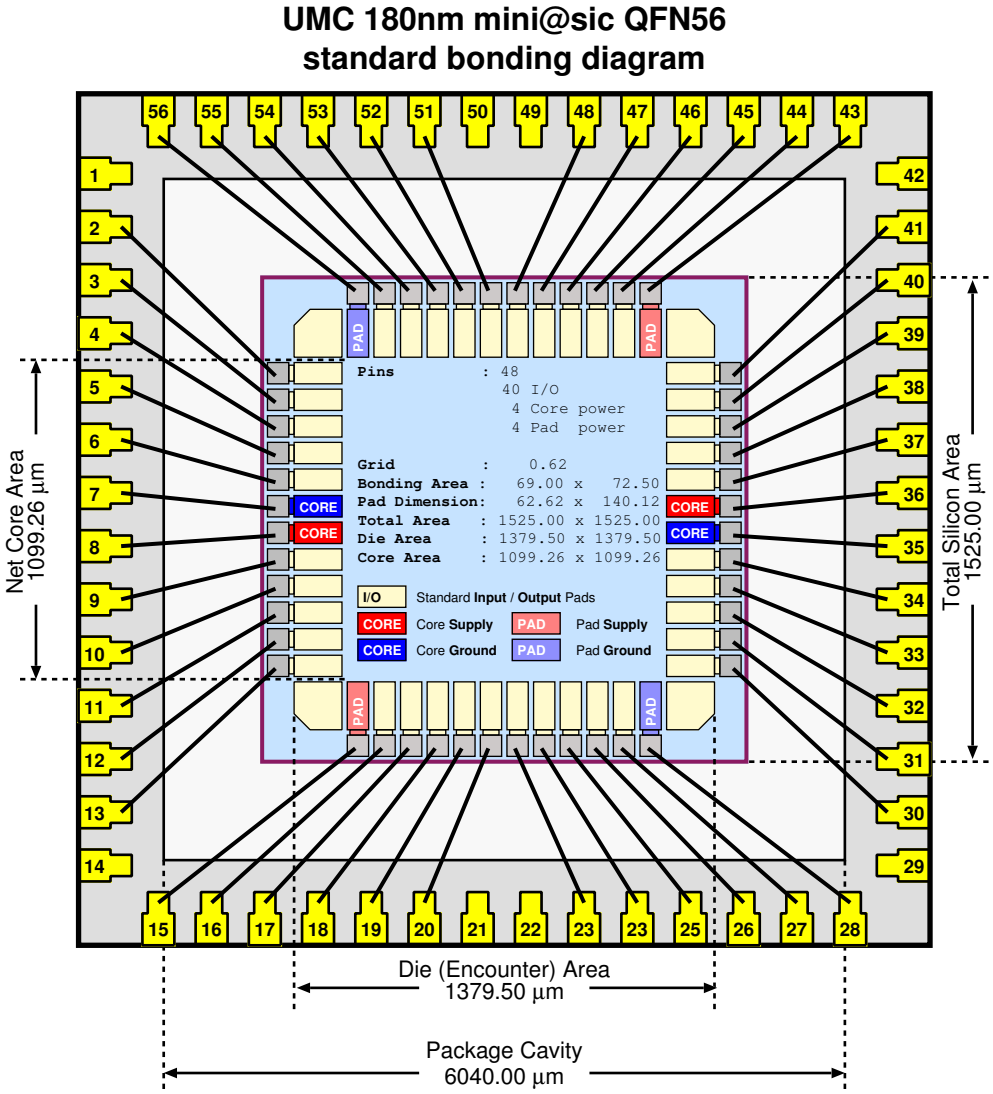


Figure C.1.: Bonding diagram.

C.4. Pin Map

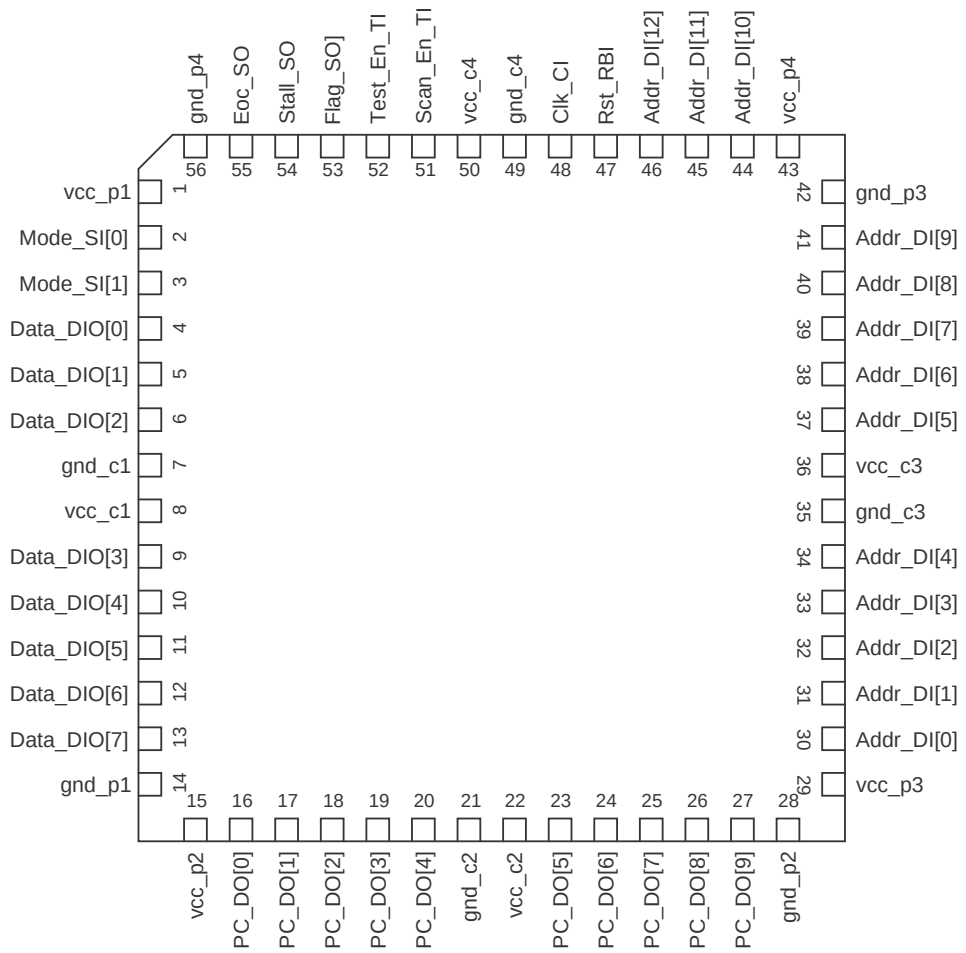


Figure C.2.: OR10N pinout.

C.5. Pin Description

Table C.1 describes pin usage. The first four are power pins, followed by the user-defined pins.

Table C.1.: Pin Description. (Power, IO, Clock, Reset, Test)

No.	Name	Setup	Run	Readout	Test
14; 28; 42; 56	gnd_p	Pad Ground			
01; 15; 29; 43	vcc_p	Pad Supply			
07; 21; 35; 49	gnd_c	Core Ground			
08; 22; 36; 50	vcc_c	Core Supply			
2-3	Mode_SI	Defines Mode. See C.7 for further information.			
4-6; 9-13	Data_DIO	Data Input to memories	Debugging Output (qprint)	Data Output from memories	-
16-20; 23-27	PC_DO	-	PC in ID stage	-	Scan_In_TI
30-34;37-41 44 45 46	Addr_DI[9:0] Addr_DI[10] Addr_DI[11] Addr_DI[12]	Memory Address	- - Instr_Ack_TI[1] Data_Ack_TI[2]	Memory Address	Scan_Out_TO - - -
47	Rst_RBI	Reset signal. (Active low)			
48	Clk_CI	Clock Signal			
51	Scan_En_TI	-	-	-	Scan Enable
52	Test_En_TI	-	-	-	Test Enable
53	Flag_SO	-	Flag Signal	-	-
54	Stall_SO	-	Stall Signal	-	-
55	Eoc_SO	-	End of Computation signal	-	-

[1] If set to 0 a cache miss from the instruction memory will occur.
[2] If set to 0 a cache miss from the data memory will occur.

C.6. Memory Addressing

Data can be written or read bytewise from chip. Internally, data is stored wordwise (4 Bytes). The memory addressing scheme is illustrated in Table C.2 and Table C.3 shows the byte addressing.

Table C.2.: Address scheme

Addr_DI[12]	0 points to the instruction memory 1 points to the data memory
Addr_DI[11:2]	word address
Addr_DI[1:0]	byte address

Table C.3.: Bit and Byte Ordering [2]

Bit 31 Bit 24	Bit 23 Bit 16	Bit 15 Bit 8	Bit 7 Bit 0
MSB			LSB
Byte Address 00	Byte Address 01	Byte Address 10	Byte Address 11

Table C.4 shows all cpu-side relevant addresses and their usage. For registers which are specially used by the compiler consider the OpenRISC 1000 Architecture Manual ([2], Ch. 16.2).

Table C.4.: Special Memory Mapping

memory	address	name	usage
data	0x1FFF	std_out	Bytes written to this address are mapped to the output *Data_DIO*. (qprint)
GPR	(r9) 0x9	LR	Link address used for l.jal(r)
SPR	0x11	SR	The Supervision Register contains branch, carry and overflow flags.
→SR	0x9	F	(Branch) Flag
→SR	0xA	CY	Carry Bit
→SR	0xB	OV	Overflow Bit
SPR	0x2801	MACLO	Contains lower 32 bits of l.muld(u) operations.
SPR	0x2802	MACHI	Contains higher 32 bits of l.muld(u) operations.

C.7. Operation Modes

OR10N runs in 4 different operational modes and 1 test mode. Table C.1 shows pin usage for all functional modes separately.

C.7.1. Functional Modes

IDLE

CPU is in IDLE mode if Mode_SI[1:0] is set to 00.
CPU is disabled, memories are not written. The chip drives the bidirectional signal Data_DIO to all zero.

SETUP

CPU is in SETUP state if Mode_SI[1:0] is set to 10.
In this mode the data can be written from outside to both of the memories. Data is laid bytewise to the bidirectional pin Data_DIO and is saved at address Addr_DI on rising edge. Section C.6 explains how the bytes are addressed.

RUN

CPU is in RUN mode if Mode_SI[1:0] is set to 01.
Main Mode. CPU runs until end of computation is reached. (special instruction l.cust1), then signal Eoc_SO is set to 1. During RUN state some outputs can be used for debugging.

- PC_DO: Current PC [12:2] in Instruction Decode stage (ID) stage (shifted two bits to the right).
- Data_DIO: Is either 0 or an ASCII character. Can be printed out using qprint (C programs) or writing to address 0x1FFF (Assembler).
- Flag_SO: Current Flag.
- Stall_SO: Indicates whether there is a stall at this time.
- Eoc_SO: Indicates that the end of the program were reached.

Although no misses can occur on this chip, there is a possibility to inject a delibarate miss to check cpu's ability for handling them. The following inputs are used for this option.

- Addr_DI[11]=Instr_Ack_TI: If this signal is set to 0, the instruction memory will cause a miss in the next cycle.
- Addr_DI[12]=Data_Ack_TI: If this signal is set to 0, the data memory will cause a miss in the next cycle.

READOUT

CPU is in READOUT mode if Mode_SI[1:0] is set to 11.
In the READOUT mode both memories can be read out using the address scheme from Section C.6. The byte at address Addr_DI is read from memory and put on Data_DIO with one latency.

C.7.2. Test Modes

If Test_En_TI is set to 1 both memory macrocells are bypassed. This allows a better test coverage using ATPG. See schemas of the instruction and data memory in Appendix F to see how these bypasses are realized.

C.8. Timing

OR10N is time-optimized for register to register allowing for a clock period of 2.75 ns (364 MHz) in RUN mode. If debugging outputs or qprint is used longest path lengthens to 4.14 ns (241 MHz) or 6.04 ns (165 MHz), respectively. Fig. C.3 and Table C.5 illustrate timing the maximum clocking frequency. As can be seen on Figure C.3 there is one in-to-out path: From mode input (*Mode_SI*) to data output (*Mode_DIO*). This leads not to a conflict with the clock period T_{SETUP}, T_{RUN} or $T_{READOUT}$.[3]

Table C.5.: Longest path and maximum frequency ∼ Mode

Mode	t_{lp}	f_{max}
SETUP	3.50 ns	285 MHz
RUN	2.75 ns	363 MHz
RUN+debug	4.14 ns	241 MHz
RUN+qprint	6.04 ns	165 MHz
READOUT	3.55 ns	282 MHz

[3] If the order SETUP-RUN-READOUT is not violated, there is at least one latency until inputs affect outputs such that the longer path does not matter.

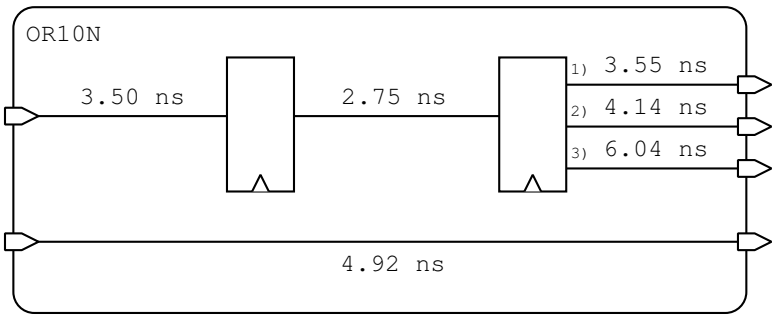


Figure C.3.: Timing diagram.
1) w/o debugging, 2) debugging information, 3) debugging incl. qprint

C.9. Electrical Specifications

Table C.6.: DC characteristics [8]

Symbol	Description	Min.	Max.	Unit
V_{Il}	Input low voltage	-	0.8	V
V_{Ih}	Input high voltage	2.0	-	V
V_{Ol}	Output low voltage	-	0.4	V
V_{Oh}	Output high voltage	2.4	-	V
I_O	Output driving		8	mA

C.9.1. Recommended Operating Regions

Table C.7.: Recommended Operating Conditions [8]

Symbol	Description	Min.	Typ.	Max.	Unit
vcc_c	Core power supply	1.62	1.8	1.98	V
vcc_p	Pad power supply	2.97	3.3	3.63	V
T_J	Operating junction temperature	-40	25	125	$°C$

C.9.2. Absolute Maximum Ratings

Table C.8.: Absolute Maximum Ratings [8]

Symbol	Description	Rating	Unit
vcc_c	Core power supply	-0.5 $\sim$ 2.5	V
vcc_p	Pad power suplly	-0.5 $\sim$ 4.6	V
V_{In}	Input voltage	-0.5 $\sim$ 4.6	V
I_{In}	DC input current	50	mA
I_{Out}	DC output current	50	mA

Appendix D

Instructions

Table D.2 complements Table 4.1 with a more detailled description about the functionality of the instruction set. Instruction description and naming conventions are adopted from the OpenRISC 1000 Architecture Manual and are explained in Table D.1.

Table D.1.: Naming conventions

l.*	ORBIS32 instruction
l.*s	The result of this instruction is sign-extended (exts).
l.*z	The result of this instruction is zero-extended (extz).
rX	General-Purpose Register X
SR	Supervision Register
spr	Special-Purpose Register
K	Immediate Operand is zero-extended.
I	Immediate Operand is sign-extended.
L	Length Immediate (zero-extended)
X	Don't Care Bits in Instruction Encoding
[*]	Address space
(*)	Memory entry at specified address
{*}	Operations in brackets are executed prior assignment.

Table D.2.: Instructions detailed

l.add rD,rA,rB	rD[31:0] ← rA[31:0] + rB[31:0] SR[CY] ← carry SR[OV] ← overflow

Table D.2: Instructions detailed (contin.)

l.addc rD,rA,rB	rD[31:0] ← rA[31:0] + rB[31:0] + SR[CY] SR[CY] ← carry SR[OV] ← overflow
l.addi rD,rA,I	rD[31:0] ← rA[31:0] + exts(Immediate) SR[CY] ← carry SR[OV] ← overflow
l.addic rD,rA,I	rD[31:0] ← rA[31:0] + exts(Immediate) + SR[CY] SR[CY] ← carry SR[OV] ← overflow
l.and rD,rA,rB	rD[31:0] ← rA[31:0] AND rB[31:0]
l.andi rD,rA,K	rD[31:0] ← rA[31:0] AND extz(Immediate)
l.bf N	EA ← exts(Immediate << 2) + BranchInsnAddr PC ← EA if SR[F] set
l.bnf N	EA ← exts(Immediate << 2) + BranchInsnAddr PC ← EA if SR[F] cleared
l.eoc=l.cust1	EOC ← 1
l.extbs rD,rA	rD[31:8] ← rA[7] rD[7:0] ← rA[7:0]
l.extbz rD,rA	rD[31:8] ← 0 rD[7:0] ← rA[7:0]
l.exths rD,rA	rD[31:16] ← rA[15] rD[15:0] ← rA[15:0]
l.exthz rD,rA	rD[31:16] ← 0 rD[15:0] ← rA[15:0]
l.extws rD,rA	rD[31:0] ← rA[31:0]
l.extwz rD,rA	rD[31:0] ← rA[31:0]
l.j N	PC ← exts(Immediate << 2) + JumpInsnAddr
l.jal N	PC ← exts(Immediate << 2) + JumpInsnAddr LR ← DelayInsnAddr + 4
l.jalr rB	PC ← rB LR ← DelayInsnAddr + 4
l.jr rB	PC ← rB
l.lbs rD,I(rA)	EA ← exts(Immediate) + rA[31:0] rD[7:0] ← (EA)[7:0] rD[31:8] ← (EA)[7]
l.lbz rD,I(rA)	EA ← exts(Immediate) + rA[31:0] rD[7:0] ← (EA)[7:0] rD[31:8] ← 0
l.lhs rD,I(rA)	EA ← exts(Immediate) + rA[31:0] rD[15:0] ← (EA)[15:0] rD[31:16] ← (EA)[15]

Table D.2: Instructions detailed (contin.)

l.lhz rD,I(rA)	EA ← exts(Immediate) + rA[31:0] rD[15:0] ← (EA)[15:0] rD[31:16] ← 0
l.lws rD,I(rA)	EA ← exts(Immediate) + rA[31:0] rD[31:0] ← (EA)[31:0]
l.lwz rD,I(rA)	EA ← exts(Immediate) + rA[31:0] rD[31:0] ← (EA)[31:0]
l.mfspr rD,rA,K	rD[31:0] ← spr(rA OR Immediate)
l.movhi rD,K	rD[31:0] ← extz(Immediate) << 16
l.mtspr rA,rB,K	spr(rA OR Immediate) ← rB[31:0]
l.mul rD,rA,rB	rD[31:0] ← rA[31:0] * rB[31:0] SR[OV] ← overflow SR[CY] ← carry
l.muld rA, rB	MACHI ← {rA[31:0] * rB[31:0]}[63:32] MACLO ← {rA[31:0] * rB[31:0]}[31:0]
l.muldu rA, rB	MACHI ← {rA[31:0] * rB[31:0]}[63:32] MACLO ← {rA[31:0] * rB[31:0]}[31:0]
l.muli rD,rA,I	rD[31:0] ← rA[31:0] * Immediate SR[OV] ← overflow SR[CY] ← carry
l.mulu rD,rA,rB	rD[31:0] ← rA[31:0] * rB[31:0] SR[OV] ← overflow SR[CY] ← carry
l.nop K	
l.or rD,rA,rB	rD[31:0] ← rA[31:0] OR rB[31:0]
l.ori rD,rA,K	rD[31:0] ← rA[31:0] OR extz(Immediate)
l.rori rD,rA,L	rD[31-L:0] ← rA[31:L] rD[31:32-L] ← rA[L-1:0]
l.sb I(rA),rB	EA ← exts(Immediate) + rA[31:0] (EA)[7:0] ← rB[7:0]
l.sfeq rA,rB	SR[F] ← rA[31:0] == rB[31:0]
l.sfeqi rA,I	SR[F] ← rA[31:0] ≠ rB[31:0]
l.sfges rA,rB	SR[F] ← rA[31:0] ≥ rB[31:0]
l.sfgesi rA,I	SR[F] ← rA[31:0] ≥ exts(Immediate)
l.sfgeu rA,rB	SR[F] ← rA[31:0] ≥ rB[31:0]
l.sfgeui rA,I	SR[F] ← rA[31:0] ≥ exts(Immediate)
l.sfgts rA,rB	SR[F] ← rA[31:0] > rB[31:0]
l.sfgtsi rA,I	SR[F] ← rA[31:0] > exts(Immediate)
l.sfgtu rA,rB	SR[F] ← rA[31:0] > rB[31:0]
l.sfgtui rA,I	SR[F] ← rA[31:0] > exts(Immediate)
l.sfles rA,rB	SR[F] ← rA[31:0] ≤ rB[31:0]

Table D.2: Instructions detailed (contin.)

l.sflesi rA,I	SR[F] ← rA[31:0] ≤ exts(Immediate)
l.sfleu rA,rB	SR[F] ← rA[31:0] ≤ rB[31:0]
l.sfleui rA,I	SR[F] ← rA[31:0] ≤ exts(Immediate)
l.sflts rA,rB	SR[F] ← rA[31:0] < rB[31:0]
l.sfltsi rA,I	SR[F] ← rA[31:0] < exts(Immediate)
l.sfltu rA,rB	SR[F] ← rA[31:0] < rB[31:0]
l.sfltui rA,I	SR[F] ← rA[31:0] < exts(Immediate)
l.sfne rA,rB	SR[F] ← rA[31:0] ≠ rB[31:0]
l.sfnei rA,I	SR[F] ← rA[31:0] ≠ exts(Immediate)
l.sh I(rA),rB	EA ← exts(Immediate) + rA[31:0] (EA)[15:0] ← rB[15:0]
l.sll rD,rA,rB	rD[31:rB[4:0]] ← rA[31-rB[4:0]:0] rD[rB[4:0]-1:0] ← 0
l.slli rD,rA,L	rD[31:L] ← rA[31-L:0] rD[L-1:0] ← 0
l.sra rD,rA,rB	rD[31-rB[4:0]:0] ← rA[31:rB[4:0]] rD[31:32-rB[4:0]] ← rA[31]
l.srai rD,rA,L	rD[31-L:0] ← rA[31:L] rD[31:32-L] ← rA[31]
l.srl rD,rA,rB	rD[31-rB[4:0]:0] ← rA[31:rB[4:0]] rD[31:32-rB[4:0]] ← 0
l.srli rD,rA,L	rD[31-L:0] ← rA[31:L] rD[31:32-L] ← 0
l.sub rD,rA,rB	rD[31:0] ← rA[31:0] - rB[31:0] SR[CY] ← carry SR[OV] ← overflow
l.sw I(rA),rB	EA ← exts(Immediate) + rA[31:0] (EA)[31:0] ← rB[31:0]
l.xor rD,rA,rB	rD[31:0] ← rA[31:0] XOR rB[31:0]
l.xori rD,rA,I	rD[31:0] ← rA[31:0] XOR exts(Immediate)

The following table illustrates the signal coding for all instruction which is being decoded by the controller.

										PC Sel	ALU Op1	ALU Op2	Imm Ext Sel	ALU Op	Mult Running	Sign Mode	Set Maacc lhoi	Data WE	Data Type	Data Ext	Data Reg	Result Sel	Reg W Addr	Reg W Data	Reg WE	SP Reg WE	SR FCO_YV	Eoc
									standard values	00	00	00	0	NOP	0	0	00	0	00	0	0	0	0	00	0	0	000	0
1	l.bf	N	000100	NNNNN	NNNNN	NNNNN	NNN	NNNN	NNNN	11																	
2	l.bnf	N	000011	NNNNN	NNNNN	NNNNN	NNN	NNNN	NNNN	11																	
3	l.j	N	000000	NNNNN	NNNNN	NNNNN	NNN	NNNN	NNNN	11																	
4	l.jal	N	000001	NNNNN	NNNNN	NNNNN	NNN	NNNN	NNNN	11	10	01		ADD								1		1			
5	l.nop	K	000101	01XXX	XXXXX	KKKKK	KKK	KKKK	KKKK																		
6	l.jalr	rB	010010	XXXXX	BBBBB	XXXXX	XXX	XXXX	XXXX	10	10	01		ADD								1		1			
7	l.jr	rB	010001	XXXXX	BBBBB	XXXXX	XXX	XXXX	XXXX	10																	
8	l.movhi	rD, K	000110	DDDDD	XXXX0	KKKKK	KKK	KKKK	KKKK		11		1	MOVHI										1			
9	l.addi	rD, rA, I	100111	DDDDD	AAAAA	IIIII	III	IIII	IIII			10		ADD										1		011	
10	l.addic	rD, rA, I	101000	DDDDD	AAAAA	IIIII	III	IIII	IIII			10		ADDC										1		011	
11	l.andi	rD, rA, K	101001	DDDDD	AAAAA	KKKKK	KKK	KKKK	KKKK			10		AND										1			
12	l.lbs	rD, I(rA)	100100	DDDDD	AAAAA	IIIII	III	IIII	IIII			10		ADD					10	1	1		11	1			
13	l.lbz	rD, I(rA)	100011	DDDDD	AAAAA	IIIII	III	IIII	IIII			10		ADD					10		1		11	1			
14	l.lhs	rD, I(rA)	100110	DDDDD	AAAAA	IIIII	III	IIII	IIII			10		ADD					01	1	1		11	1			
15	l.lhz	rD, I(rA)	100101	DDDDD	AAAAA	IIIII	III	IIII	IIII			10		ADD					01		1		11	1			
16	l.lws	rD, I(rA)	100010	DDDDD	AAAAA	IIIII	III	IIII	IIII			10		ADD							1		11	1			
17	l.lwz	rD, I(rA)	100001	DDDDD	AAAAA	IIIII	III	IIII	IIII			10		ADD							1		11	1			
18	l.mfspr	rD, rA, K	101101	DDDDD	AAAAA	KKKKK	KKK	KKKK	KKKK			10		OR									10	1			
19	l.muli	rD, rA, I	101100	DDDDD	AAAAA	IIIII	III	IIII	IIII			10			1							1		1		001	
20	l.ori	rD, rA, K	101010	DDDDD	AAAAA	KKKKK	KKK	KKKK	KKKK			10		OR										1			
21	l.xori	rD, rA, I	101011	DDDDD	AAAAA	IIIII	III	IIII	IIII			10		XOR										1			
22	l.rori	rD, rA, L	101110	DDDDD	AAAAA	XXXXX	XXX	11LL	LLLL			10		ROR										1			
23	l.slli	rD, rA, L	101110	DDDDD	AAAAA	XXXXX	XXX	00LL	LLLL			10		SLL										1			
24	l.srai	rD, rA, L	101110	DDDDD	AAAAA	XXXXX	XXX	10LL	LLLL			10		SRA										1			
25	l.srli	rD, rA, L	101110	DDDDD	AAAAA	XXXXX	XXX	01LL	LLLL			10		SRL										1			
26	l.mtspr	rA, rB, K	110000	KKKKK	AAAAA	BBBBB	KKK	KKKK	KKKK			11		OR											1		

										PC Sel	ALU Op1	ALU Op2	Imm Ext Sel	ALU Op	Mult Running	Sign Mode	Set Maaccl hoi	Data WE	Data Type	Data Ext	Data Reg	Result Sel	Reg W Addr	Reg W Data	Reg WE	SP Reg WE	SR FCO _YV	Eoc
									standard values	00	00	00	0	NOP	0	0	00	0	00	0	0	0	0	00	0	0	000	0
27	l.sb	I(rA), rB	110110	IIIII	AAAAA	BBBBB	III	IIII	IIII			11		ADD				1	10		1							
28	l.sh	I(rA), rB	110111	IIIII	AAAAA	BBBBB	III	IIII	IIII			11		ADD				1	01		1							
29	l.sw	I(rA), rB	110101	IIIII	AAAAA	BBBBB	III	IIII	IIII			11		ADD				1			1							
30	l.add	rD, rA, rB	111000	DDDDD	AAAAA	BBBBB	X00	XXXX	0000					ADD											1		011	
31	l.addc	rD, rA, rB	111000	DDDDD	AAAAA	BBBBB	X00	XXXX	0001					ADDC											1		011	
32	l.and	rD, rA, rB	111000	DDDDD	AAAAA	BBBBB	X00	XXXX	0011					AND											1			
33	l.or	rD, rA, rB	111000	DDDDD	AAAAA	BBBBB	X00	XXXX	0100					OR											1			
34	l.sll	rD, rA, rB	111000	DDDDD	AAAAA	BBBBB	X00	00XX	1000					SLL											1			
35	l.sra	rD, rA, rB	111000	DDDDD	AAAAA	BBBBB	X00	10XX	1000					SRA											1			
36	l.srl	rD, rA, rB	111000	DDDDD	AAAAA	BBBBB	X00	01XX	1000					SRL											1			
37	l.sub	rD, rA, rB	111000	DDDDD	AAAAA	BBBBB	X00	XXXX	0010					SUB											1		011	
38	l.xor	rD, rA, rB	111000	DDDDD	AAAAA	BBBBB	X00	XXXX	0101					XOR											1			
39	l.mul	rD, rA, rB	111000	DDDDD	AAAAA	BBBBB	X11	XXXX	0110						1							1			1		001	
40	l.mulu	rD, rA, rB	111000	XXXXX	AAAAA	BBBBB	X11	XXXX	1011						1	1						1			1		010	
41	l.muld	rA, rB	111000	XXXXX	AAAAA	BBBBB	X11	XXXX	0111						1		11					1						
42	l.muldu	rA, rB	111000	XXXXX	AAAAA	BBBBB	X11	XXXX	1100						1	1	11					1						
43	l.sfeq	rA, rB	111001	00000	AAAAA	BBBBB	XXX	XXXX	XXXX					EQ													100	
44	l.sfges	rA, rB	111001	01011	AAAAA	BBBBB	XXX	XXXX	XXXX					GES													100	
45	l.sfgeu	rA, rB	111001	00011	AAAAA	BBBBB	XXX	XXXX	XXXX					GEU													100	
46	l.sfgts	rA, rB	111001	01010	AAAAA	BBBBB	XXX	XXXX	XXXX					GTS													100	
47	l.sfgtu	rA, rB	111001	00010	AAAAA	BBBBB	XXX	XXXX	XXXX					GTU													100	
48	l.sfles	rA, rB	111001	01101	AAAAA	BBBBB	XXX	XXXX	XXXX					LES													100	
49	l.sfleu	rA, rB	111001	00101	AAAAA	BBBBB	XXX	XXXX	XXXX					LEU													100	
50	l.sflts	rA, rB	111001	01100	AAAAA	BBBBB	XXX	XXXX	XXXX					LTS													100	
51	l.sfltu	rA, rB	111001	00100	AAAAA	BBBBB	XXX	XXXX	XXXX					LTU													100	
52	l.sfne	rA, rB	111001	00001	AAAAA	BBBBB	XXX	XXXX	XXXX					NE													100	
53	l.sfeqi	rA, I	101111	00000	AAAAA	IIIII	III	IIII	IIII			10		EQ													100	

										PC Sel	ALU Op1	ALU Op2	Imm Ext Sel	ALU Op	Mult Running	Sign Mode	Set Maacc lhoi	Data WE	Data Type	Data Ext	Data Reg	Result Sel	Reg W Addr	Reg W Data	Reg WE	SP Reg WE	SR FCO_YV	Eoc
									standard values	00	00	00	0	NOP	0	0	00	0	00	0	0	0	0	00	0	0	000	0
54	l.sfgesi	rA, I	101111	01011	AAAAA	IIIII	III	IIII	IIII			10		GES													100	
55	l.sfgeui	rA, I	101111	00011	AAAAA	IIIII	III	IIII	IIII			10		GEU													100	
56	l.sfgtsi	rA, I	101111	01010	AAAAA	IIIII	III	IIII	IIII			10		GTS													100	
57	l.sfgtui	rA, I	101111	00010	AAAAA	IIIII	III	IIII	IIII			10		GTU													100	
58	l.sflesi	rA, I	101111	01101	AAAAA	IIIII	III	IIII	IIII			10		LES													100	
59	l.sfleui	rA, I	101111	00101	AAAAA	IIIII	III	IIII	IIII			10		LEU													100	
60	l.sfltsi	rA, I	101111	01100	AAAAA	IIIII	III	IIII	IIII			10		LTS													100	
61	l.sfltui	rA, I	101111	00100	AAAAA	IIIII	III	IIII	IIII			10		LTU													100	
62	l.sfnei	rA, I	101111	00001	AAAAA	IIIII	III	IIII	IIII			10		NE													100	
63	l.extbs	rD, rA, rB	111000	DDDDD	AAAAA	XXXXX	X01	00XX	1100					EXTBS											1			
64	l.extbz	rD, rA, rB	111000	DDDDD	AAAAA	XXXXX	X11	00XX	1100					EXTBZ											1			
65	l.exths	rD, rA, rB	111000	DDDDD	AAAAA	XXXXX	X00	00XX	1100					EXTHS											1			
66	l.exthz	rD, rA, rB	111000	DDDDD	AAAAA	XXXXX	X10	00XX	1100					EXTHZ											1			
67	l.extws	rD, rA, rB	111000	DDDDD	AAAAA	XXXXX	X00	00XX	1101					EXTW											1			
68	l.extwz	rD, rA, rB	111000	DDDDD	AAAAA	XXXXX	X01	00XX	1101					EXTW											1			
69	l.eoc (cust1)		011100	XXXXX	XXXXX	XXXXX	XXX	XXXX	XXXX		01																	1

Appendix E

Used Software

With this small list, we want thank all developer of the softwares we used in this thesis and additionally the IIS which enabled us to use industrial software.

Table E.1.: Used software

Calibre	MentorGraphics, Inc.
Drive	Google Inc.
Encounter	Cadence Design Systems, Inc.
Matlab	MathWorks
ModelSim	MentorGraphics, Inc.
Synopsys	Synopsys Inc.
TetraMAX ATPG	Synopsys Inc.
Tgif	William Chia-Wei Cheng

Appendix F

Schematics

This appendix contains all detailled schemata of this thesis.

Figures

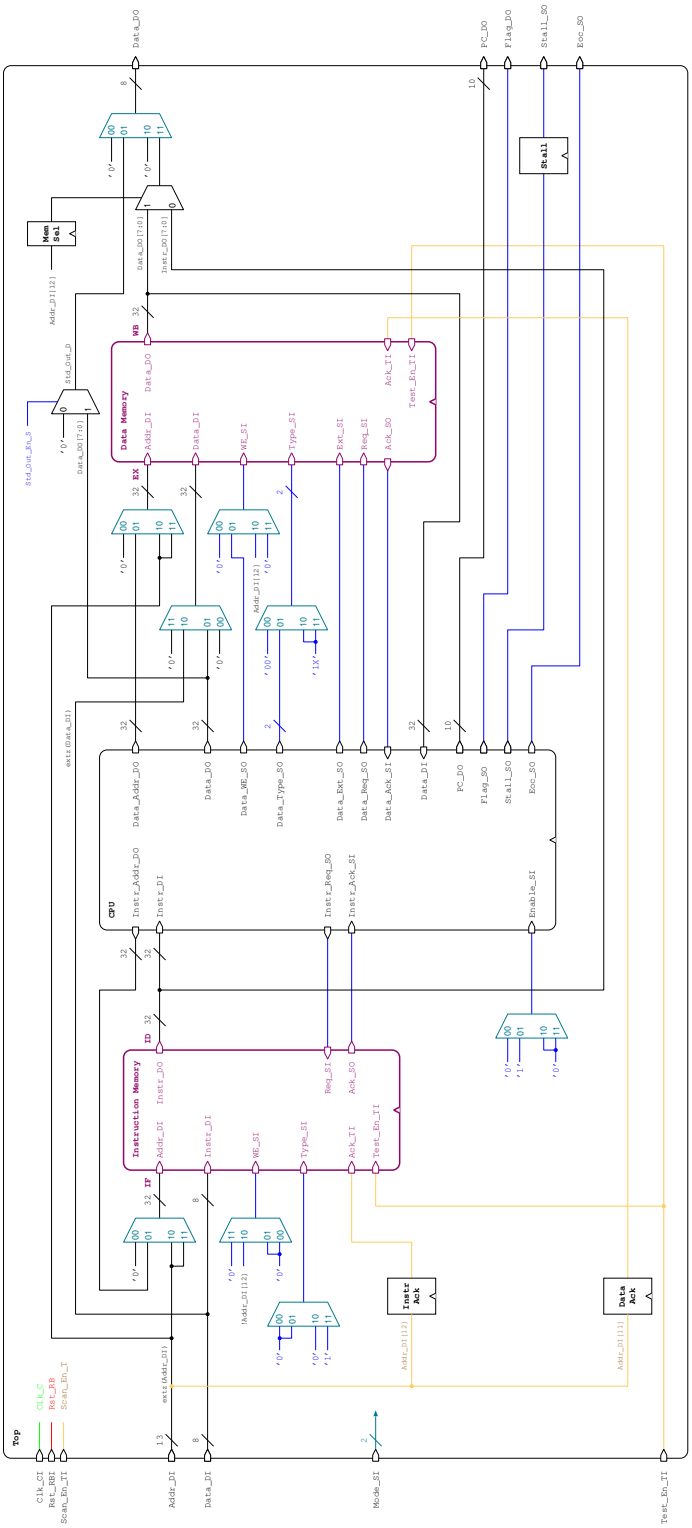


Figure F.1.: Main schema of the top level design

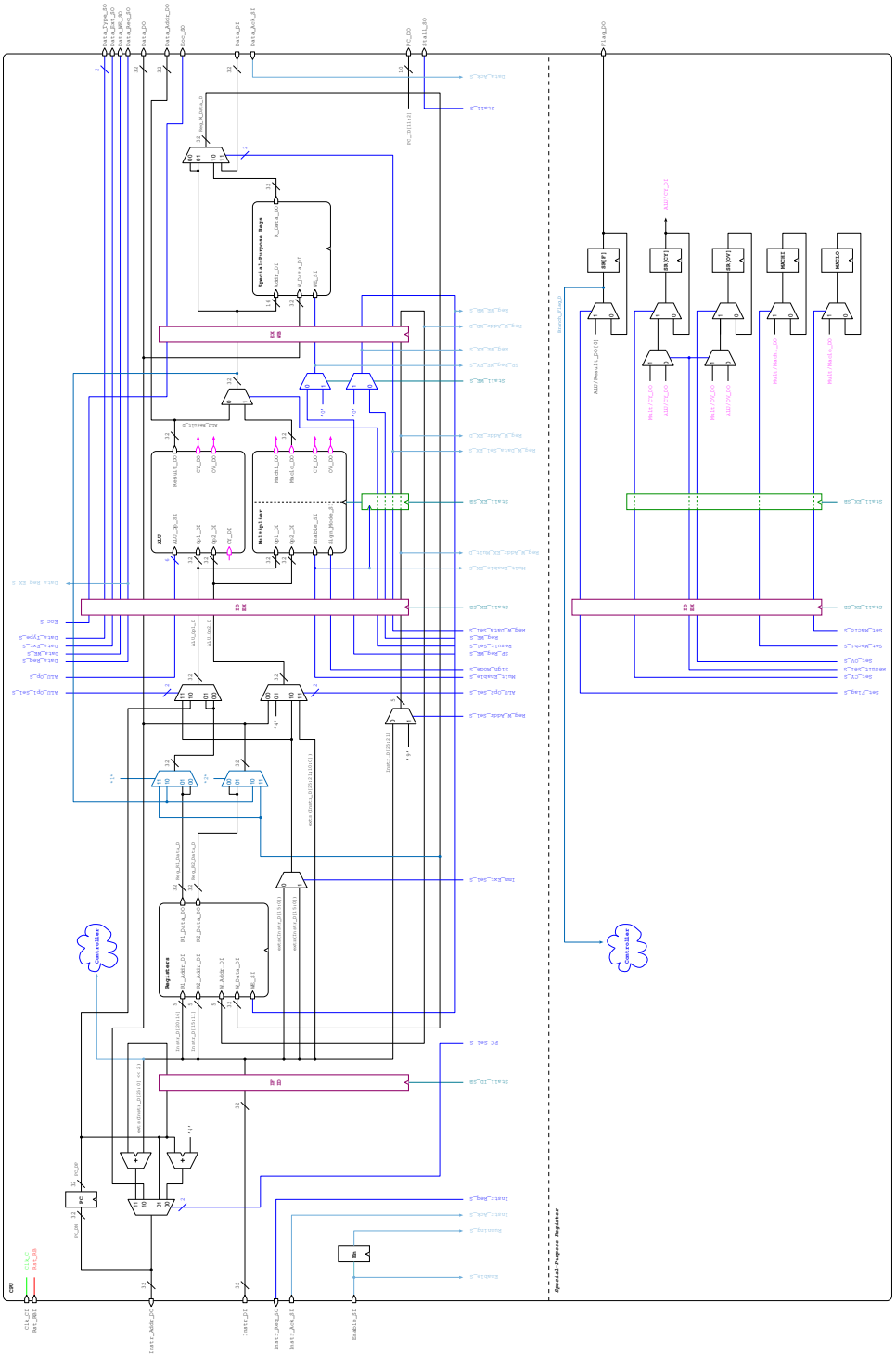


Figure F.2.: Main schema of the CPU

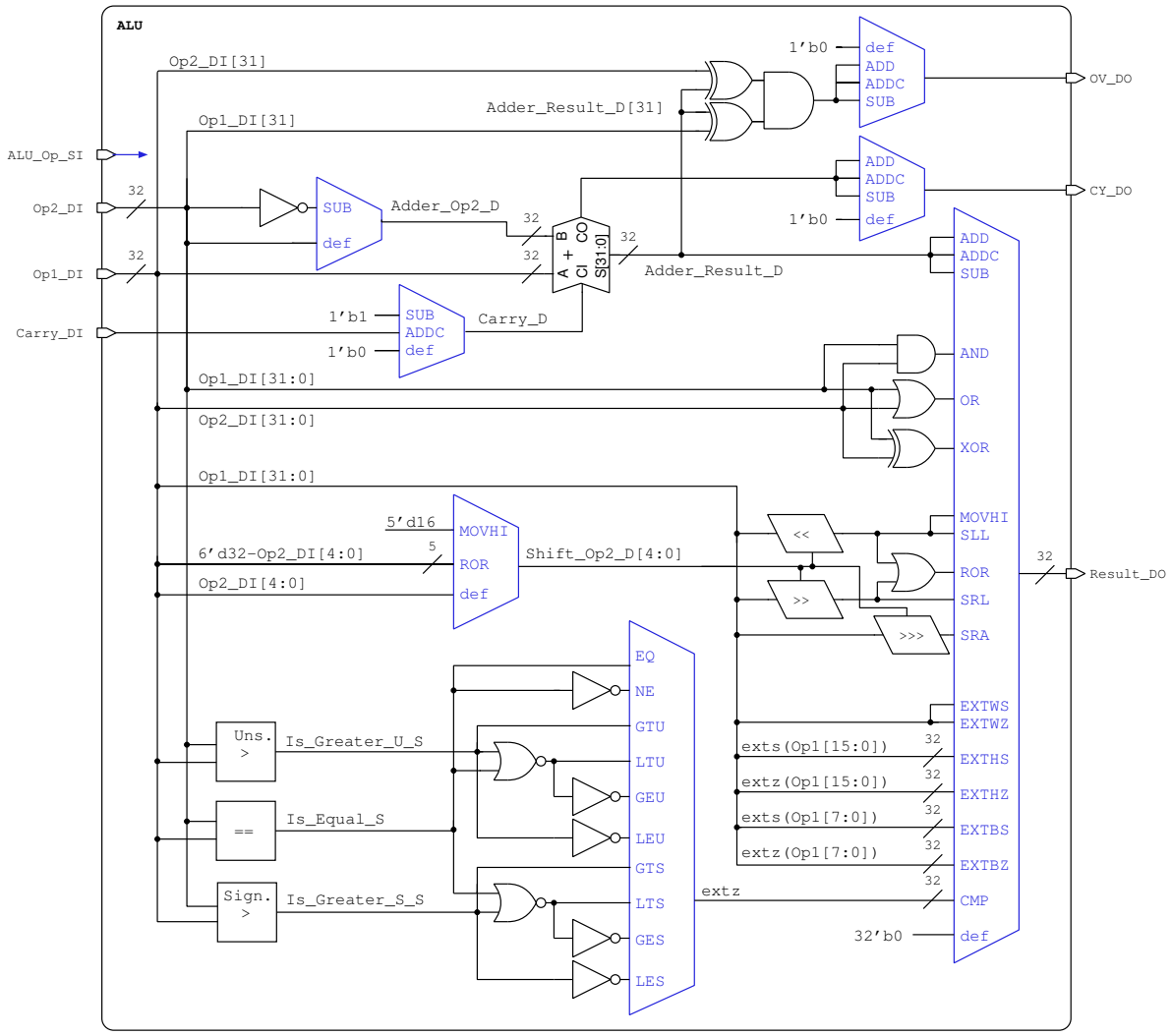


Figure F.3.: Main schema of the ALU

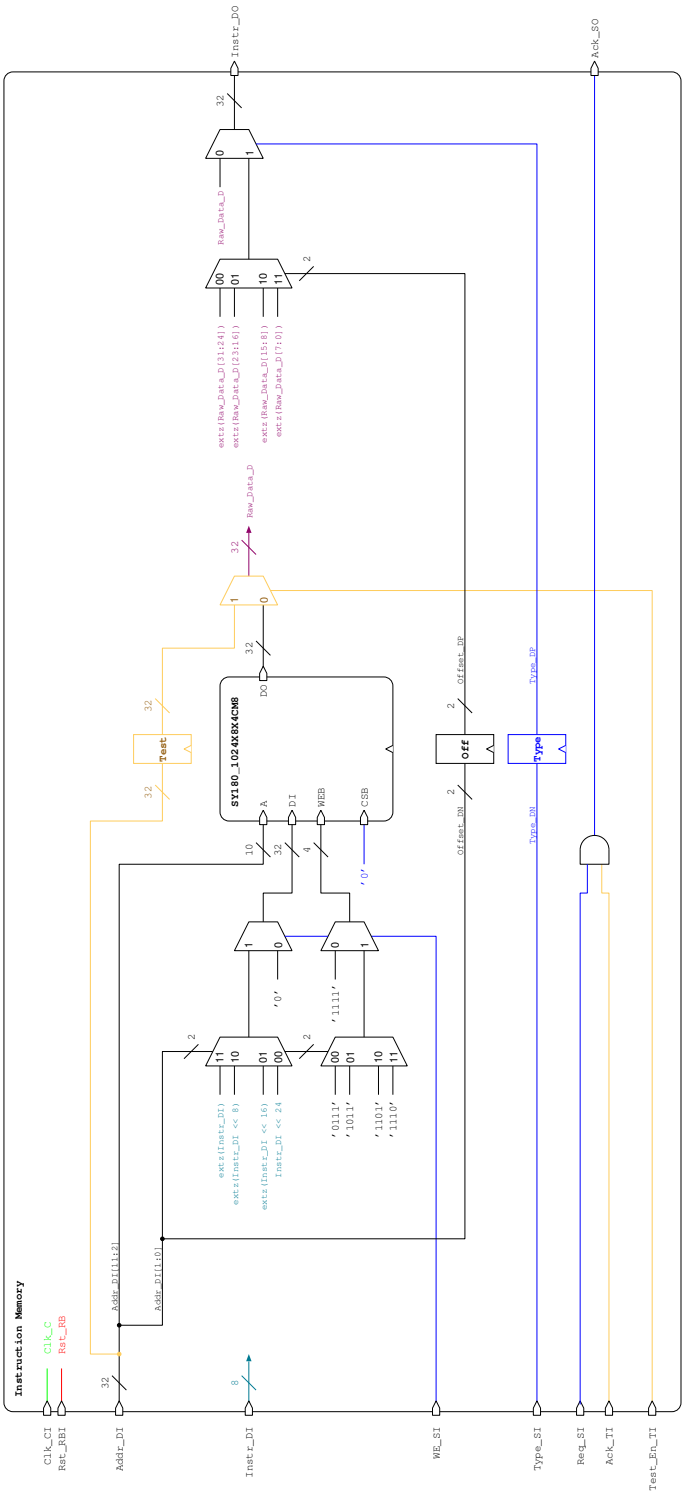


Figure F.4.: Main schema of the instruction memory

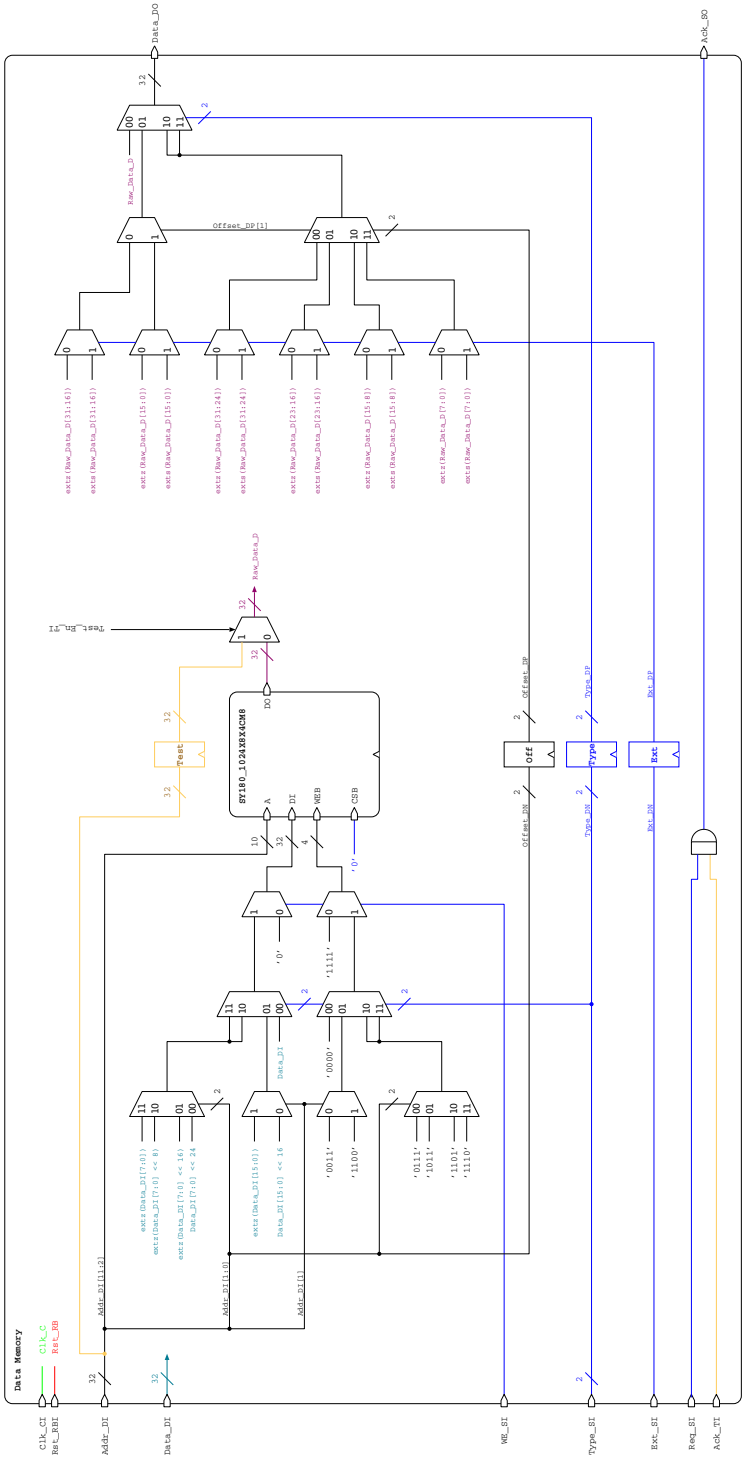


Figure F.5.: Main schema of the data memory

Glossary

Gate Equivalent Gate Equivalent is a measure for the complexity of a circuit which is independent of the technology. One gate equivalent is equal to one two-input NAND gate. In the used Complementary Metal Oxide Semiconductor (CMOS) technolgy this correspond to 4 transistors and an area of 9.3744 μm^2.[6].

GNU toolchain is a collection of tools used for developing applications and operating systems produced by the GNU Project.

Harvard microarchitecture is a computer architecture with physically separate storage and data path for instructions and data.

RISC **R**educed **I**nstruction **S**et **C**omputing is a CPU design strategy, where the instructions are kept simple and have the same size.

SoC **S**ystem **o**n a **C**hip is an integrated circuit that combines all components of a computer into a single chip.

Bibliography

[1] OpenCores, Jan. 2014. [Online]. Available: http://opencores.org

[2] D. Lampret and J. B. et al. (2012, Dec.) Openrisc 1000 architecture manual.

[3] D. Lampret and J. Baxter. (2012, Mar.) Openrisc 1200 ip core specification architecture manual.

[4] D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture.* Morgan Kaufmann, 2013.

[5] N. Felber and H. Kaeslin, *Lecture notes on Design for Testability.* IIS, ETH Zürich, 2013.

[6] H. Kaeslin, *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication.* Cambridge University Press, Apr. 2008.

[7] Doulos, *SystemVerilog Golden Reference Guide*, 2012.

[8] Faraday. (2009, Jan.) 0.18um standard i/o cell library.

[9] H. Kaeslin and N. Felber, *TBD.* IIS, ETH Zürich, 2013.