NTNU

Norwegian University of
Science and Technology

Design of a near-threshold Microcontroller

Åsmund Kvam Oma

Master of Science in Electronics

Submission date: June 2016

Supervisor: Snorre Aunet, IET

Co-supervisor: Trond Ytterdal, IET
Even Låte, IET
Ali Asghar Vatanjou, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Preface

This thesis is written for the degree of Master of science at Norwegian University of Science and Technology. Supervisors for this thesis were Phd candidate Even Låte, Professor Snorre Aunet, Phd candidate Ali Asghar Vatanjou and Professor Trond Ytterdal.

Trondheim, June 10, 2016

Åsmund Kvam Oma

Acknowledgment

I would like to thank Phd candidate Even Låte, Phd candidate Ali Asghar Vatanjou, Professor Trond Ytterdal, Professor Snorre Aunet and Sebastian Bøe for their help and contributions in this thesis.

The standard cell library used in this project was made and characterized by Ali Asghar Vatanjou. Custom SRAM blocks was created by Even Låte.

Å.K.O

Abstract

There is a strong interest in ultra low voltage digital design as emerging applications like Internet of Things, wearable biomedical sensors, radio frequency identification, sensor networks and more are gaining traction. This thesis describes the implementation, synthesis and testing of a microcontroller using a near-threshold library. The system has been described in VHDL and synthesized for near-threshold operation on 28 nm FDSOI production technology from STmicroelectronics. The microcontroller implements a 32 bit RISC-V subset compatible pipelined processor and has SPI connectivity. Two single port 2kB SRAM modules are used as RAM. A power gating technique that reduces the static power in an ALU during runtime has been implemented and compared to a traditional ALU. Traditional coarse grain power gating of the processor has also been implemented. Using a supply voltage of 350 mV and a clock speed of 1 MHz the schematic SPICE simulation reported an average power consumption of 4.42 μW during program execution. In power gated mode the microcontroller consumed 2.98 μW. In a sensor logging program the average energy per executed instruction was 4.91 pJ. Runtime power gating reduced the average energy consumption of the ALU with 58 - 57% with a propagation delay penalty of 346 - 143% depending of the sizing of the power gating transistors.

Sammendrag

Det er en stor interesse i digitale system med ultra lav spenningsforsyning ettersom nye applikasjoner som internett av ting, bærbare medisinske sensorer, radiofrekvensidentifikasjon, sensornettverk og mer blir mer populært. Denne avhandlingen beskriver implementasjon, syntese og testing av en mikrokontroller ved bruk av et nærterskelsbibliotek. Systemet har blitt beskrevet i VHDL og syntetisert for nærterskeloperasjon på en 28 nm FDSOI produksjonsteknologi fra STmicroelectronics. Mikrokontrolleren implementerer en 32 bit RISC-V undergruppekompatibel samlebåndsprosessor og har SPI tilkobling. To enkeltport 2 kB SRAM moduler er brukt som RAM. En strømportingsteknikk som reduserer statisk strømforbruk i en ALU under kjøretid har blitt implementert og sammenlignet med en tradisjonell ALU. Tradisjonell grov strømporting av prosessoren har også blitt implementert. Med en spenningsforsyning på 350mV og en klokkefrekvens på 1 MHz rapporterte en skjematikk SPICE simulering et gjennomsnittlig strømbruk på 4,42 μW under programkjøring. I strømportet modus brukte mikrokontrolleren 2,98 μW. I et sensorloggingsprogram var den gjennomsnittlige energibruken per instruksjon 4,91 pJ. Kjøretid-strømporting reduserte den gjennomsnittlige energibruken av ALUen med 58 - 57%, med en propageringsulempe på 346 - 143% avhengig av størrelsen på strømportingstransistorene.

Contents

Chapter 1

Introduction

Applications like sensor networks, radio frequency identification, biomedical equipment and other internet of things applications emerged due to tremendous advancements in digital circuits. These applications have a very small energy budget and data processing power efficiency is paramount. It has been shown that there is a tremendous opportunity to save energy by operating in or close to the subthreshold ($|V_T| > V_{DD}$) region [1]. Sub- and near-threshold operation can reduce the energy per operation by 5-10x over standard voltage operation [2]. Many sensor processing applications require a sensor processing bandwidth of less than 250 Hz [3] and therefore can take advantage of operating in this region. Because the static power is a relatively large component of the total power in low voltage circuits [1, p28], techniques to decrease static power is the most effective way to reduce overall power consumption. This thesis investigates the power savings from two different power gating techniques, runtime power gating and traditional coarse grain power gating. The microcontroller presented in this thesis operates in the near-threshold region at 350 mV using a modern 28 nm FDSOI fabrication technology. The microcontroller implements a 32 bit RISC-V subset compatible processor, SPI controller and 4 kB of SRAM. The design is simulated in RTL, at gate level and in schematic SPICE.

Chapter two include theory and background, the chapter covers theory behind the various optimization techniques used in the design and other relevant background.

Chapter three, the methodology chapter, is divided into two sections, Implementation and Verification and benchmarking. Implementation covers the design of the system while Verification and benchmarking cover the various tests that are performed on the system.

Chapter four lists all the results in the form of tables and figures. The chapter includes a brief description of the tests.

Chapter five, the results are interpreted and analyzed. This chapter also includes a section with suggestions for future work.

Chapter six is the conclusion chapter.

Appendix A lists acronyms used in the thesis and Appendix B contain various code that may be interesting to the reader.

Chapter 2

Background

2.1 Instruction set architecture

Instruction set architecture (ISA) determines the set of instructions supported by a processor. The ISA also dictates some of the memory architecture, interrupt handling and I/O. Instruction set architectures are divided into classes like CISC, RISC and VLIW. A reduced instruction set computer (RISC) in contrast to a complex instruction set computer (CISC) use simple instructions which require few cycles to perform, and often have a load store architecture. In a load store architecture the only instructions touching memory are the load and store instructions. RISC also has a smaller number of highly optimized instructions instead of a larger set of more versatile instructions like in CISC. CISC can match the complexity of high level languages like C more closely while RISC require a more sophisticated compiler to compile high level languages. RISC also often use a fixed instruction bit width which reduce the hardware implementation complexity.

2.1.1 RISC-V

RISC-V [4] is an open source ISA based on the principles of RISC and anyone is free to design and manufacture hardware and software supporting this ISA. The RISC-V toolkit includes a cross compiler, test programs, a simulator and documentation [5]. The user level ISA has several extension like multiplication, division, 64 bit integer, single and double precision floating point and more which can be implemented to increase performance or expand functionality of the design.

2.2 Power consumption in digital CMOS

From [6]:

$$P_{average} = P_{switching} + P_{short-circuit} + P_{leakage} \tag{2.1}$$

Power consumption in digital CMOS can be divided into three sources, switching, short-circuit and leakage. The switching, active or dynamic power is the power used to charge the capacitive load, C_L, in the CMOS through the PMOS network from 0V to VDD. From [6]:

$$P_{switching} = \alpha_{0\to1} C_L V_{dd}^2 f_{clk}$$

$\alpha_{0\to1}$ is the probability of a node switching from 0V to V_{DD} during a clock cycle. f_{clk} is the clock frequency of the CMOS circuit. The switching power is exponentially dependent on the supply voltage and its contribution is therefore greatly reduced in the sub- and near-threshold domain. From [6]:

$$P_{short-circuit} = I_{sc} \times V_{dd} \tag{2.2}$$

I_{sc} is the short-circuit current. Short-circuit or crowbar current is the current through CMOS that occur when the input of the of a logic unit is switching. The switching transition is not instantaneous and inputs of logic units will have an input between 0 V and VDD in the transition period, providing a direct path from supply to ground if VDD fulfills this equation: $V_{dd} > V_{Tn} + |V_{Tp}|$ where V_{Tn} is the threshold voltage of the NMOS transistors and V_{Tp} is the threshold voltage of the PMOS transistors [6]. If VDD is below the sum of the threshold voltages, like in sub- and near-threshold designs, the transistor networks will still provide a lower resistance from supply to ground when inputs are transitioning. From [6]:

$$P_{leakage} = I_{leakage} \times V_{dd} \tag{2.3}$$

$I_{leakage}$ is the leakage current. Six mechanisms are contributing to leakage current: reverse junction bias current and band-to-band tunneling, sub-threshold currents, tunneling through and into gate oxide, injection of hot carriers from substrate to gate oxide, gate induced drain leakage, punch-through current [7]. Because CMOS scaling has lowered the threshold voltage and reduced the channel length, subthreshold leakage has emerged as one of the dominant sources of leakage [7].

Subthreshold leakage is the current from source to drain while the transistor is in an off state.

2.3 Clock gating

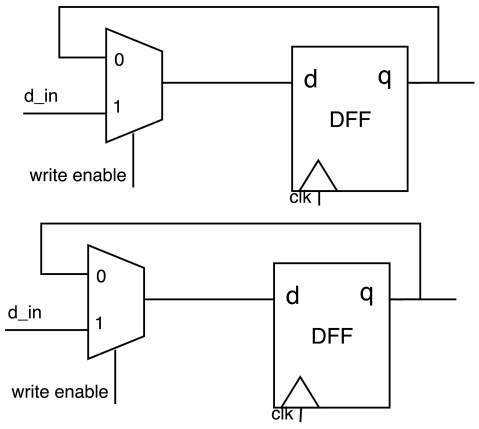

Figure 2.1: D-Flip-flops with write enable and no clock gate

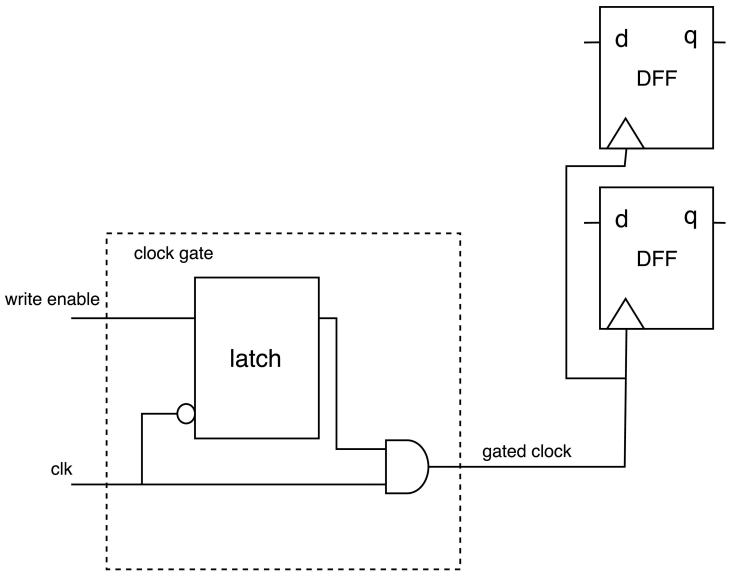

Figure 2.2: Clock gated D-flip-flops with write enable

Clock gating is a technique used to reduce active power. Instead of rerouting the output of the flip-flop into the input of the flip-flop when write is disabled, a latch clamps the local clock to zero. Figure 2.2 depicts flip-flops with a clock gate and figure 2.1 without clock gating. The active power saving comes from less switching activity in the clock tree, in the cells connected to the output of the registers and the registers itself. In flip-flop arrays with a shared write enable signal a clock gate can save area, because the individual flip flop don't need an input multiplexer and can use a shared clock gate.

2.4 Power gating

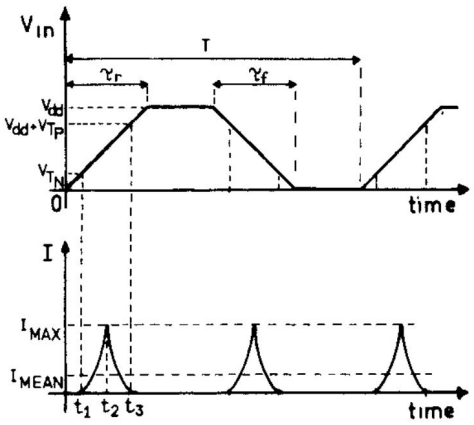

Figure 2.3: Current behavior of an inverter without load. The current spikes are the crowbar current. From [8]

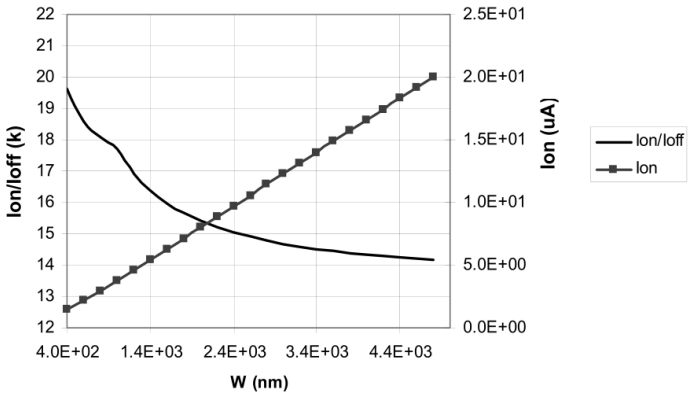

Figure 2.4: 90nm pMOS Ion/Ioff and Ion vs. W. From [9, p234]

Power gating is a power saving technique where logic blocks or cells are disconnected from VDD or GND. Power gating may reduce leakage power and active power in periods when unused cells are turned off. The power and ground planes which may be turned off are often called virtual GND and VDD. The act of turning on and off a virtual supply are not instantaneous because of the capacitance of the nodes connected to the virtual supply, the performance of the circuit may therefore be degraded when using this technique. A sudden surge in current may cause a voltage drop which in turn degrades the performance of the rest of the circuit. A way to mitigate the current surge issue is to slowly turn the virtual supply on, but this further increases switch-on time. Power gating is often classified into two classes, coarse grain power gating and fine grain power gating. In fine grain power gating the power transistors are encapsulated within a standard cell. With fine grain power gating the timing impact and IR drop is easy to characterize, and the size of the power transistors is therefore simpler to select [9, p39]. A cell with fine grain power gating is typically 2 to 4 times the size of a cell without fine grain power gating [9, p38]. In coarse grain power gating a group of cells shares a virtual supply. The worst case current in groups of cells can only be estimated and sizing of the power transistor network is therefore more difficult. Coarse grain power gating is more commonly used [9, p39] because the area overhead of coarse grain power gating is smaller than fine grain power gating. A block of cells or a sub-cell that shares a virtual ground is

called a power island or a power plane. Because small power planes have less capacitance, they require a smaller power gating network to switch on for a given switch-on time. Run-time power gating is a term used for very rapid power switching of a power island. A transistor switching VDD is called a header switch, a transistor switching GND is called a footer switch.

2.4.1 Isolation of power islands

Because outputs of power gated blocks are floating, they may cause crowbar current in adjacent powered on blocks. Crowbar current is a phenomenon that occurs when both the pmos and nmos networks are turned on when the gate voltage is between GND and VDD. Figure 2.3 depicts crowbar current in an inverter. The outputs may also cause functional problems in powered on blocks if the powered down block provide control signals to powered on blocks. The output of power gated blocks therefore needs to be isolated from other blocks. The isolating cell is called an isolation cell. An isolation cell must not cause crowbar current when the input is floating and not have a floating output. A simple method of isolating is to use an AND- or OR-gate as an isolation cell. AND-ing the isolation cell output with a power enable signal will clamp the signal to 0 when power is disabled. Figure 2.5 show an AND isolation cell on transistor level, notice how a floating input may not cause crowbar current due to the switched off NMOS network in the NAND gate. An OR gate can be used to clamp the signal of a power gated block to 1. A different approach is to use a pull up or pull down transistor as an isolation cell. When the block is on, the pull up/down transistors will fight the outputs of the block and waste power. Pull up/down transistors are also prone to metal migration [9] . Because isolation cells add extra delay, power gating delay sensitive blocks may therefore degrade performance. It is not necessary to isolate the inputs of a power gated block as it will not result in any crowbar current or functional issues.

2.4.2 Power transistors

Power transistors are the on/off switches for the virtual supply or ground rails. They are often evaluated in three metrics: switch efficiency, area efficiency and IR drop [9]. The switch efficiency is the ratio between off and on current. A good power transistor has a high ratio, meaning the current through the transistor is small compared to when it is turned on. Switch efficiency vary

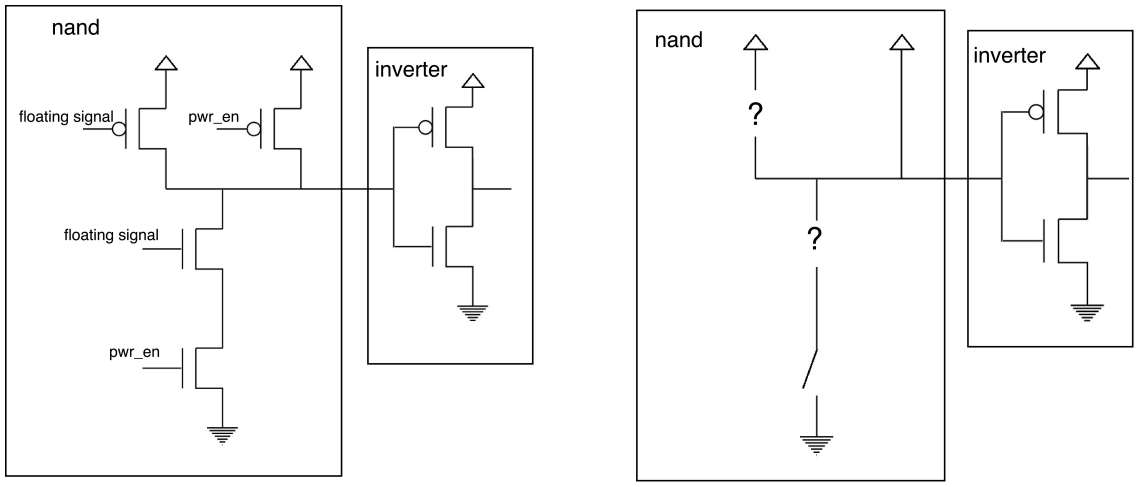

Figure 2.5: Isolation cell on transistor level. pwr_en is 0 to the right.

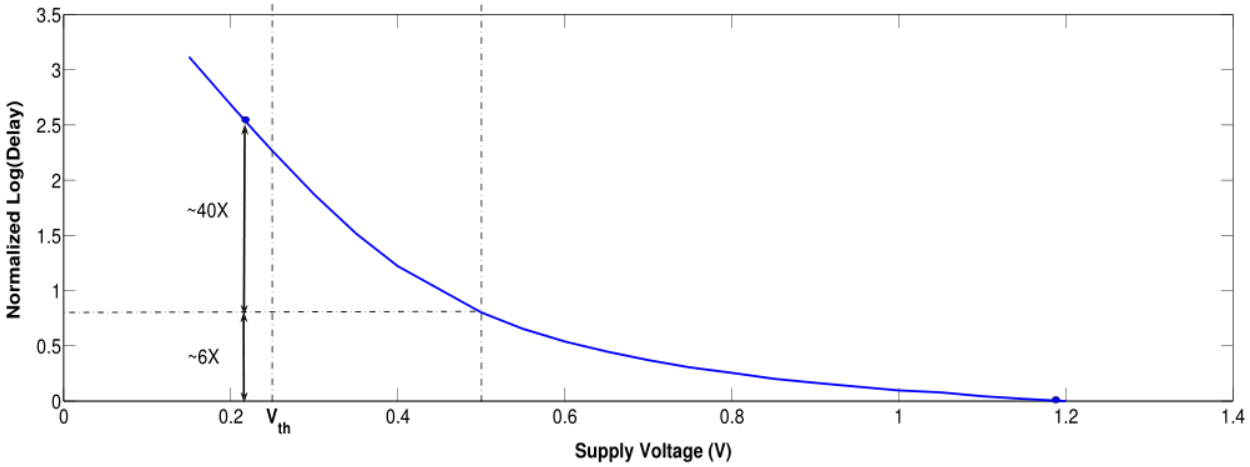

Figure 2.6: Delay vs supply voltage for a chain of 51 inverters in 90nm technology. From [10, p401]

with transistor type, gate width and length, substrate bias voltage and process technology. As can be observed in figure 2.4, the switch efficiency is reduced with larger gate widths for PMOS. The area efficiency is defined by the ratio between the drive strength and the area the transistor occupies. The IR drop is the resistive properties of the power transistor when it is turned on, or the voltage drop across it at a given current. A sub- or near-threshold circuit is more vulnerable to voltage drops because the propagation delay slope is steeper in this region. The relationship between supply voltage and propagation delay of an inverter chain in 90nm technology can be observed in figure 2.6.

2.5 SPI

Serial Peripheral Interface or SPI is a synchronous interface used primarily in embedded applications for off chip communication. SPI has a master-slave architecture and specify 3 wires plus one chip select wire for each slave as seen table 2.7. Communication is initiated when the master asserts a chip select. The master controls the serial clock which determines the rate which the data is transmitted over MOSI and MISO. Supported clock speed typically range from hundreds of kHz to a few MHz. SPI is a full duplex interface, master to slave through MOSI and slave to master through MISO. A simplification of a SPI controller can be observed in figure 2.8. SPI has two options, often referred to as CPOL and CPHA. CPOL, short for clock polarity determines the polarity of an idle serial clock and CPHA short for clock phase determines which serial clock edge data should be captured and which clock edge data should be output. Standards for higher throughput SPI exists, these protocols use more than one MOSI/MISO pair.

SCLK	Serial clock
MOSI	Master out, slave in
MISO	Master in, slave out
CS	Chip select

Figure 2.7: SPI pins

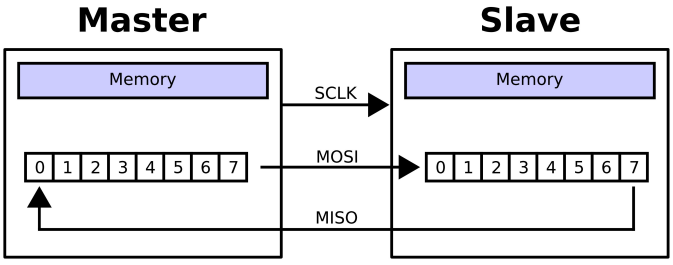

Figure 2.8: SPI implementation [11]

Chapter 3

Methodology

3.1 Implementation

3.1.1 System overview

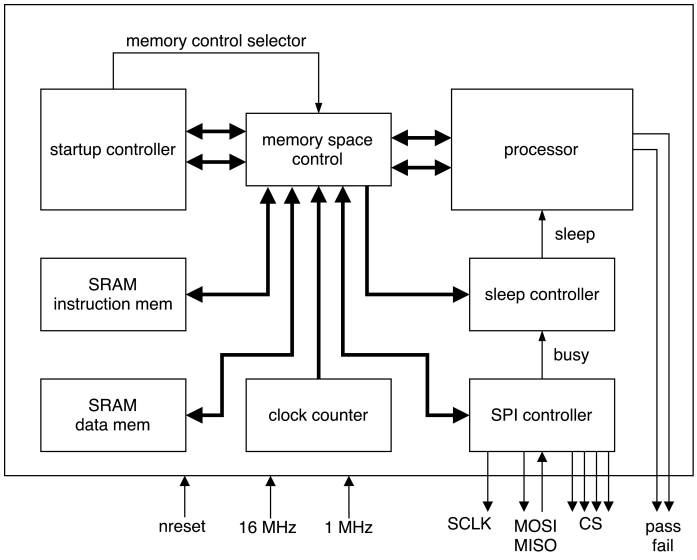

Figure 3.1: System architecture overview

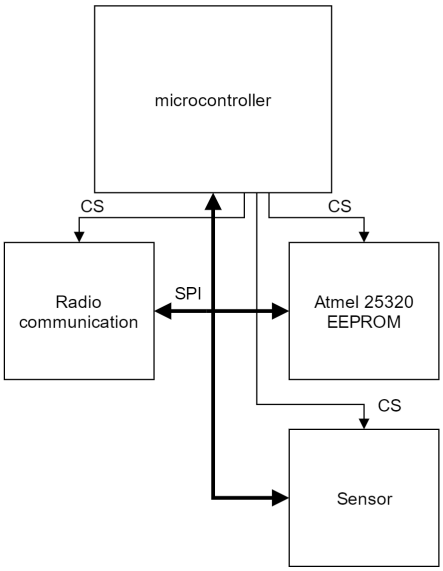

Figure 3.2: Usecase suggestion. The EEPROM chip is mandatory

The system consists of an SPI controller, RISC-V subset compatible pipelined processor, a startup controller, a sleep controller, a memory control unit, a clock counter and two single port 2 kB SRAMs. Listing B.9 contain the VHDL code for the top module. The system must be connected to an EEPROM programmed with a RISC-V program on the SPI bus to function. The startup

controller was designed to use an Atmel 25320 EEPROM chip, but changing this to a different SPI EEPROM is likely only to require changing some constants in the startup controller VHDL description. The startup controller takes control over the SPI and SRAMs when nreset is deasserted. The SPI bus is used by the startup controller to copy the content of the external EEPROM to the SRAMs, which are instruction memory and data memory for the RISC-V processor. Once the startup sequence is complete, the startup controller turns over the control of the SPI and SRAMs to the RISC-V processor. The microcontroller requires two clocks, 16 and 1 MHz. The SRAM is the only component using the 16 MHz clock and is used to control the internal state machine of the SRAM controller. The SRAM is only capable of one read or write per 16 clock cycles. The processor requires at least one read operation per processor clock and therefore the processor and the rest of the system are clocked at 1 MHz. 16 and 1 MHz was chosen because this is the highest frequency the SRAM is capable of. An overview of the system architecture can be viewed in figure 3.1. Because the startup controller and processor have a separate instruction memory interface and data memory interface, the interface between to the memory space control is illustrated with two separate buses. Figure 3.2 show an example of how the microcontroller can be used. The Atmel 25320 EEPROM chip is the non-volatile memory that contains the program. The system is designed and tested for a 350mV supply. 350mV was chosen because it was a realistic supply voltage for the SRAM to operate at. The processor was adapted from an earlier specialization project by the author. The rest of the system is implemented during this thesis with the exception of the SRAM and standard cell library which were implemented by the co-supervisors Even Låte and Ali Asghar Vatanjou respectively.

3.1.2 Near-threshold standard cell library

The standard cell library used in this thesis was created by Ali Asghar Vatanjou and was characterized at 350mV. The cells used low threshold transistors. The PMOS transistors had a threshold voltage of 390mV and the NMOS had a threshold voltage of 345mV. It was designed for 28 nm fully depleted silicon on insulator (FD-SOI) fabrication technology from STMicroelectronics. An overview of the cells in the library can be observed in table 3.1.

Name	Functionality	Transistors	dimensions (x,y) in μm	Size in μm^2
invx2	Inverter with twice the drive strength of invx1	2	0.8, 1.73	1.384
NAND2x1	Two input NAND-gate	4	1.2, 1,73	2.076
FAx1	One bit full adder	34	6.4, 1.73	11.072
invx1	Inverter	2	0.8, 1,73	1.384
RFDFFx1	D-flip-flop with no set	26	5.6, 1.73	9.688
NOR2x1	Two input NOR-gate	4	1.2, 1.73	2.076
RFDFFSx1	D-flip-flop with asynchronous set	29	6.4, 1.73	11.072
BUFx1	Buffer	4	1.2, 1.73	2.076

Table 3.1: Near-threshold cell library overview

3.1.3 RISC-V processor

The RISC-V processor used in the microcontroller implements support for a subset of RV32I. RV32I is the 32 bit integer base instruction set and do not include floating point, multiplication, division or atomic instructions. The supported instructions can be observed in table 3.2. This subset of instructions are required by the RISC-V cross compiler to compile a C program. The processor implements the classic RISC pipeline with 5 pipeline stages in a Harvard architecture fashion. The classic 5 stage RISC pipeline was chosen because of the simplicity its design and earlier experience with this kind of pipeline. It has also been found that a shallow pipeline may decrease energy per operation in subthreshold circuits [1, p47]. A classic RISC pipeline is illustrated in figure 3.3. Branch prediction and data forwarding techniques are used to increase pipeline utilization and possibly decrease processor energy consumption due to shorter program execution time.

Instruction fetch

This pipeline stage fetches the next instruction, keeps track of the program counter and contain a saturation branch predictor. A branch predictor predicts whether a branch should be taken or not taken.

Instruction decode

In this pipeline stage the control signals for the rest of the pipeline are generated. The register file is also in this pipeline stage. The RISC-V ISA specify a 32 times 32 bit register file, with 32 of the

Table 3.2: List of supported RISC-V instructions

Short name	**Full name**
LUI	load upper immediate
AUIPC	add upper immediate to pc
JAL	jump and link
JALR	jump and link register
BEQ	branch if equal
BNE	branch if not equal
BLT	branch if less than
BGE	branch if greater than
BLTU	branch if less than, unsigned
BGEU	branch if greater than, unsigned
LB	load byte
LH	load half word
LW	load word
LBU	load byte, unsigned
LHU	load half word, unsigned
SB	store byte
SH	store half word
SW	store word
ADDI	add immediate
SLTI	set on less than immediate
SLTIU	set on less than immediate, unsigned
XORI	exclusive or immediate
ORI	or immediate
ANDI	and immediate
SLLI	shift left logical immediate
SRLI	shift right logical immediate
SRAI	shift right arithmetic immediate
ADD	add
SUB	subtract
SLL	shift left logical
SLT	set on less than
SLTU	set on less than, unsigned
XOR	exclusive or
SRL	shift right logical
SRA	shift right arithmetic
OR	or
AND	and

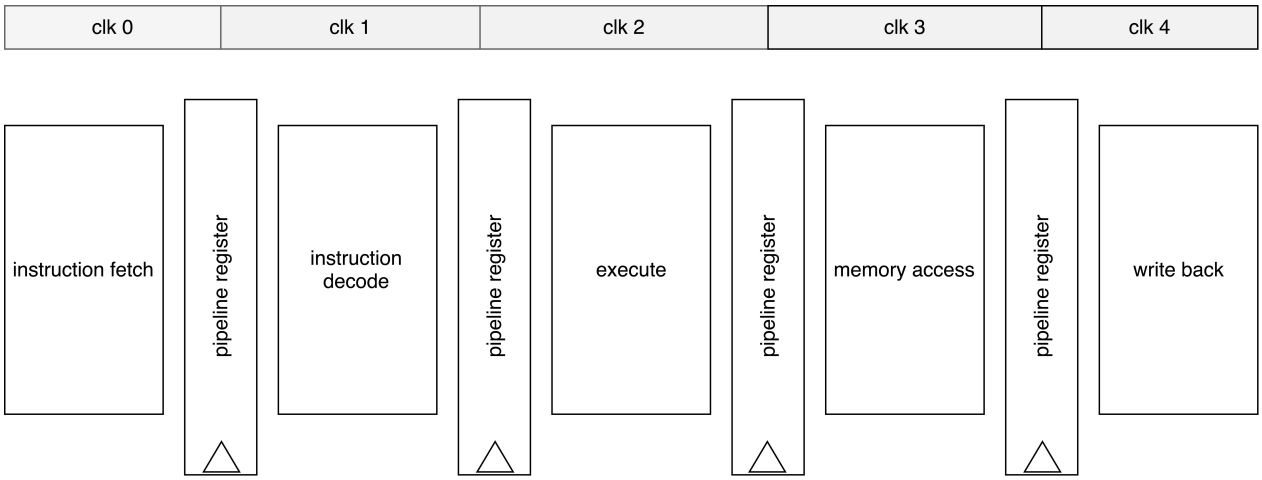

Figure 3.3: Classic RISC pipeline arrangement

registers wired to logical 0.

Execute

Execute is the stage with the ALU and is where arithmetic operations are performed on immediate or register values. The ALU also calculate the memory address for the next pipeline stage.

Memory access

The outcome of a branch is determined in this stage. If a branch has been incorrectly predicted, a control transfer signal is sent to the instruction fetch pipeline stage and the three pipeline register before memory is flushed. As the name implies, the data memory is accessed in this pipeline stage.

Register write back

This is the pipeline stage where data is written to the register file.

3.1.4 Branch prediction

The processor implements a simple local saturation counter branch predictor. A saturation counter is a state machine. A state diagram of a saturation counter is shown in figure 3.4. The saturation counter used in this branch predictor has four states: strongly not taken (00), weakly not taken (01),

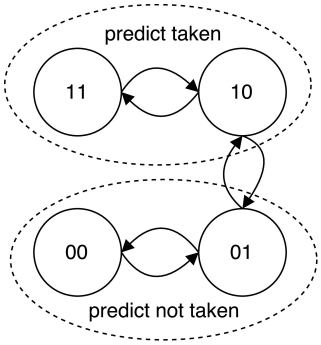

Figure 3.4: Saturation counter states

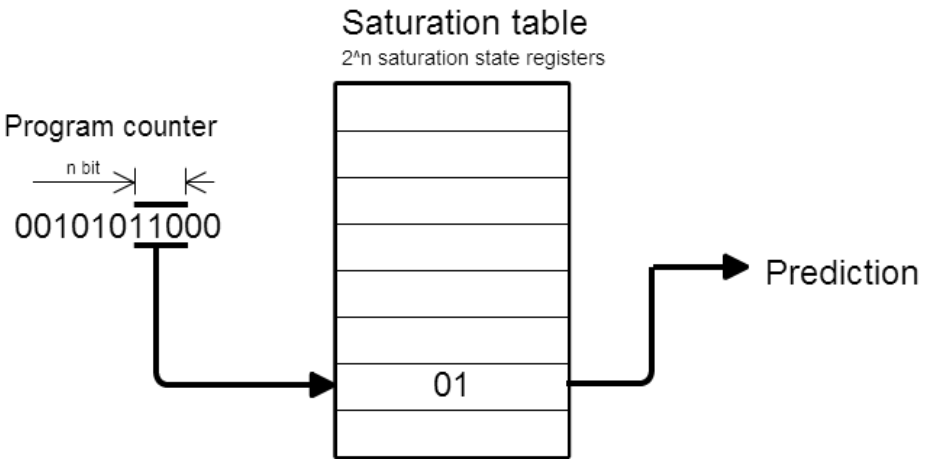

Figure 3.5: Saturation counter branch predictor illustration

weakly taken (10) and strongly taken (11). When a conditional branch is evaluated the corresponding saturation counter is moved one state towards strongly taken if the branch was taken or towards strongly not taken if the branch was not taken. The predictor has 32 saturation counters that use the LSBs of the PC as index. Because the number of saturation counters is much smaller than the maximum number of instructions in the instruction memory, two or more conditional branches may share the same saturation counter and cause a conflict. Figure 3.5 illustrate the prediction process.

3.1.5 Memory mapped functionality

Peripheral modules are accessed by the processor using a memory map. To accommodate the extra memory space these devices require, an extra bit has been added to the data memory address bus. The most significant bit of the data memory address is used to indicate that a memory access is not for the data memory. Because the extra bit add more address space than the peripheral modules require, a lot of the memory space is left unused. Table 3.3 show an overview of the memory space.

Clock counter

A simple clock counter unit has been implemented to enable the programmer to read the current time in cycles since startup. The clock counter is active as long as the chip is powered and does not go in low power mode. The counter is implemented using a 64 bit register array and will therefore never overflow. The value of the register is read using the memory interface of the processor.

000	Data memory
...	Data memory
7FF	Data memory
800	Unused
...	Unused
FF7	Unused
FF8	Timed sleep
FF9	Wait for SPI sleep
FFA	Clocks since startup (LSBs)
FFB	Clocks since startup (MSBs)
FFC	read SPI status registers
FFD	set SPI settings registers
FFE	SPI start
FFF	SPI clear

Table 3.3: Memory map

Because the register is twice the width of the data path of the memory interface, it must be read using two load operations instead of one. Because the LSBs may overflow in the period between the two load operations, the number of clocks since startup may seemingly decrease if the driver for this functionality is naively implemented. An example of this is found in listing 3.1. The probability of this error is very low, as the 32 LSBs of the counter will only overflow once every 72 minutes with a clock rate of 1 MHz. By reading the LSBs twice, once before the MSBs and once after, one can detect an overflow. If the LSBs decreased, an overflow has occurred and the whole operation should be performed again.

Listing 3.1: C code snippet of problematic read clocks implementation

```
clks  = read_lsbs();      //counter = 0x00000000FFFFFFFF
clks += read_msbs() << 32; //counter = 0x0000000100000000


//now the value of clks is 0x1FFFFFFFF
//later:


clks  = read_lsbs();      //counter = 0x00000001000000FF
clks += read_msbs() << 32; //counter = 0x0000000100000100
//now the value of clks is 0x10000000FF , which is less than before
```

Sleep controller

Low power mode is essential to most low power systems. The sleep controller controls the sleep signal to the processor. The processor initiates and configures the sleep mode by using the memory mapped interface to this module. The sleep controller implemented in this system has two sleep modes; timed sleep and SPI wait sleep. In timed sleep mode the sleep duration is set by the processor. A use case for this sleep mode is a realtime system which performs a task with a fixed interval. After checking how many clocks have passed since the task was started the system can go into low power mode for the rest of the interval. In SPI wait sleep mode the sleep controller halts the processor if the SPI controller is busy, and waits until it isn't busy. The SPI wait sleep mode is used by the SPI driver to stop the processor from reading unfinished SPI transmissions. A less power efficient way to perform this task is to use the processor to poll the SPI controller status registers in a loop until it is no longer busy. The sleep signal from this module is used to control power gates in the power gated design.

SPI controller

The SPI module is divided into two levels, a byte level and a word level controller. The byte level controller feeds a byte to the SPI bus while the word level controller can use the byte level controller to send and receive four bytes without any interaction with the processor. The controller is designed to be as autonomous as possible to reduce the instruction memory footprint of the SPI driver library. Because the SPI protocol requires data to be output and captured on different clock edges, an SPI module has to store one extra bit the half cycle between capture and output. This implementation uses an extra output register to hold the output when the shift register as seen in figure 2.8 is shifted. Because the SPI protocol require capturing or outputting data on negative SCLK edges, SCLK can not run on the same clock speed as the shift registers in the SPI controller. A four-state FSM controls each SCLK cycle. SCLK runs a fourth of the speed of the rest of the system, which translates into a maximum data transfer speed of 250kbit/s. The SPI controller requires the processor to initiate and end words that are to be transferred and the controller itself has a slight delay between the transfer of each byte. Therefore the sustained data bandwidth is slightly lower than the maximum transfer speed.

SPI controller interface

Control, status and data registers in the SPI controller can be accessed through the memory interface. A small section of the memory space is reserved for these registers, and can therefore be accessed using standard load and store instructions. It is important to tell the compiler to reserve the address space used by the controller interface, even when the SPI bus is not used, to avoid compiling a program that tries to use this space as RAM. Using only one load instruction the processor can initiate a 32 bit SPI full duplex data transfer. An additional instruction is needed when switching data transfer length or sending to a different slave. To check if a transfer is completed the processor can read the status registers of the SPI module using this interface.

3.1.6 Runtime power gating and traditional coarse grain power gating

As the supply voltage is decreased the leakage component of the total energy consumption becomes larger [1]. Power gating is a technique that can greatly reduce leakage power in unused blocks. Two different power gating techniques are implemented and tested in this design, runtime and traditional coarse grain power gating.

Power gating RTL model

All modules which implements power gating has a wrapper module which include isolation cells. In the case of the power gated ALU, the effects of power gating was modeled in the VHDL code. If power to the module is turned off, all signals is given 'X', unknown, to easily see if unisolated signals from a powered off module is used in RTL simulation. The entity declaration and architecture of the VHDL model of the power gated ALU adder is shown in listing 3.2

Listing 3.2: Power gated adder VHDL model

```
entity alu_adder is
port(
     pwr_en : in std_logic;
     A, B   : in  std_logic_vector(31 downto 0);
     C      : out std_logic_vector(31 downto 0)
   );
end alu_adder;

architecture behave of alu_adder is
signal addition : std_logic_vector(31 downto 0);
begin
C <= addition when (pwr_en = '1') else UNKNOWN_32BIT;
addition <= std_logic_vector(signed(A) + signed(B));
end behave;
```

Runtime power gating of the ALU

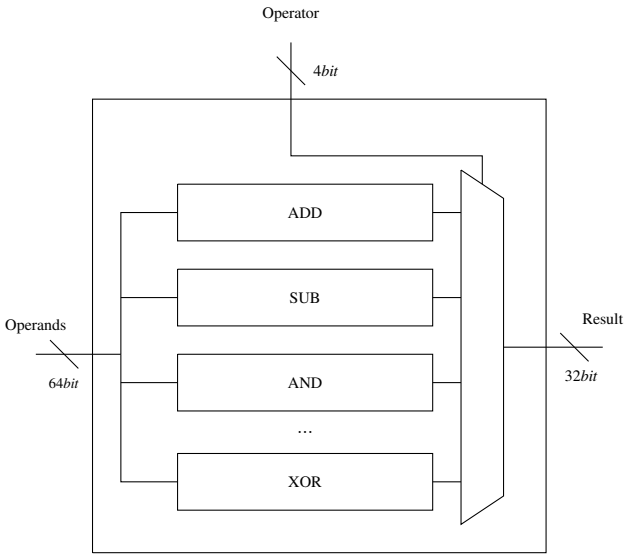

Figure 3.6: ALU without power gating

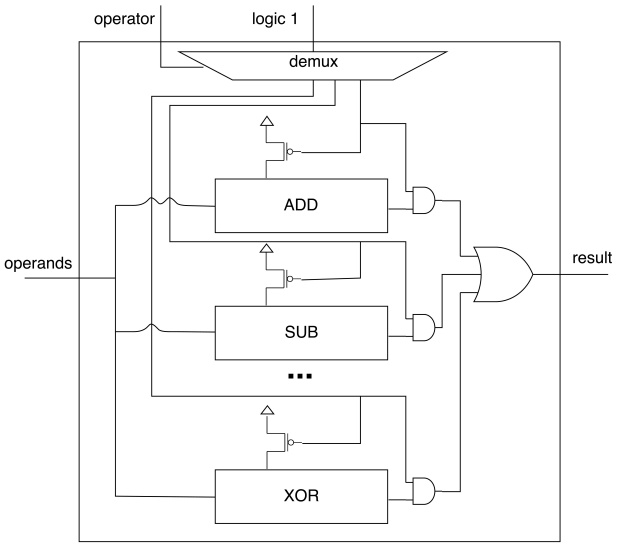

Figure 3.7: ALU with power gated functional units

The VHDL RTL description of the ALU has a separate result signal for each of the 11 operation the ALU can perform. The correct result is selected using a case statement. The VHDL code for the ALU without power gating is in Listing B.6. The RTL drawing of the ALU without runtime power gating can be observed in figure 3.6. The synthesizing tools may do optimizations which enable the ALU to reuse logic for different operations. The synthesized ALU may therefore not look like Figure 3.6. By separating each of the operations using VHDL submodules, one can force the synthesizer to not reuse logic. This strict separation is necessary to enable individual VDD supply for each operation. The power gated ALU will therefore result in an area penalty. By introducing one or more VDD or GND supply transistors, leakage from unused functional units can be greatly reduced. All outputs of the functional units must be isolated to avoid crowbar current in adjacent non-power gated blocks. Outputs of the functional units are clamped to '0' when they are powered down using AND gates as isolation cells. Because the isolated powered down functional units outputs only '0's, a MUX is not necessary to select the correct result signal, a network of OR gates is used instead. The operation control signal is demultiplexed to individual power enable signals for each functional unit. The size of the functional units varies, the optimal size for the power transistors may not be the same for all units. Figure 3.7 illustrate the design of the power gated ALU.

Sizing of runtime power transistor network

5 ALU power transistor network size configurations was created using 5 different cell area to gate width constants. As show in figure 2.4, PMOS transistors with shorter gate width have a higher switch efficiency. It was therefore decided to only use one gate width size in the PMOS transistor network, and if higher I_{on} was required, instead of widening the power transistor, use more than one. The transistor used had a gate width of 550 nm and a gate length of 30 nm. Table 4.2 show the combined gate width of the different functional units in the ALU. The cell area to gate width constants used was 2,3,4,5 and 6.

$$floor(\frac{C \times A}{T_W}) = N_T$$

The equation above is the equation used to select the number of power transistors for each module in the ALU. C is the cell area to power transistor gate width constant. A is the cell area in μm^2 of

the module which is being power gated, T_W is the width of the power transistor in nm and N_T is the number of transistors used to power the module.

$$floor(\frac{2 \times 429}{550}) = 1$$

The equation above show the calculation of the number of power transistors in the adder with a cell area to power gate width constant of 2.

Coarse grain power gating of processor

The sleep functionality of the chip provides an excellent opportunity to save power while the processor is sleeping. Footer power gates were chosen because header power gates are already used inside the processor, in the ALU. Adding an additional power gate network to the VDD will further increase the difference in rise and fall time in the ALU. NMOS transistors, which are used in footer gates, have a larger drive strength and therefore require less area which useful when powering a large portion of the design. The processor gating network consisted of 26 NMOS transistors with a gate width of 400 nm and length of 30 nm each for a total gate width of 10.4 μm.

State retention

The processor need to be able to continue the program execution after it has been temporary powered down and must therefore retain its state. There are three register arrays which must retain its data: the register file, the program counter and a sleep state register. In addition to these essential registers, the state of the saturation counters in the branch predictor is retained to stop a short sleep from reducing the branch prediction accuracy which in turn may increase execution time and task power consumption. A separate, always on, path to ground is supplied to these register banks. Figure 3.8 illustrate this arrangement. Because the inputs of the always powered on blocks are floating, the inputs of the always on block are isolated using AND-gates. The isolation of the register file is depicted in figure 3.9.

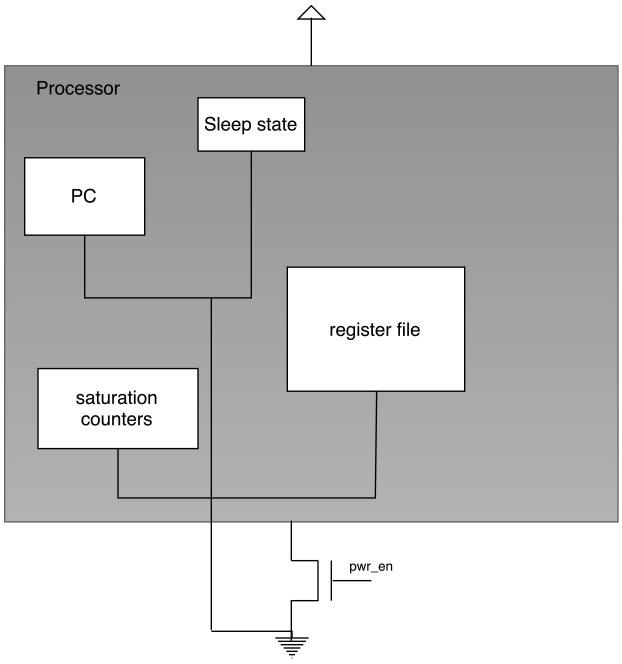

Figure 3.8: Processor state retention

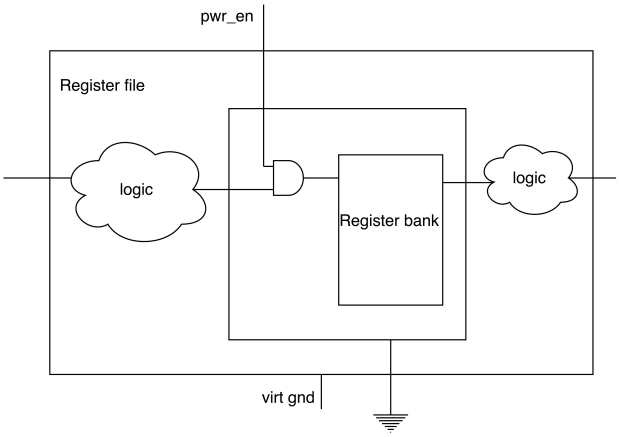

Figure 3.9: Register file state retention

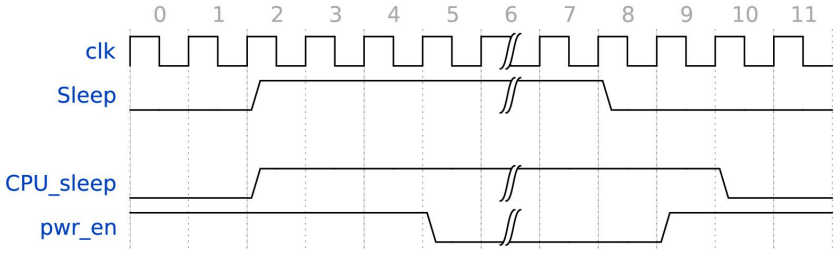

Figure 3.10: Processor power down and on transition

Power enable transition

When the processor initially is given a sleep signal, it saves the program counter of the instruction after the instruction that activated sleep. This instruction is always in the memory access pipeline stage, because the sleep controller is memory mapped. The rest of the pipeline is flushed, all dangerous signals, like register write enable are deasserted. The instruction before the initiate sleep instruction, which is in the writeback stage is allowed to finish. The processor therefore need 2 rising clock edges with sleep asserted to properly initiate sleep before the power can be turned off. Figure 3.11 show the wrapping module used in the power gated design. In the non-gated design, the sleep signal is connected directly to the processor. The processor power control module in the

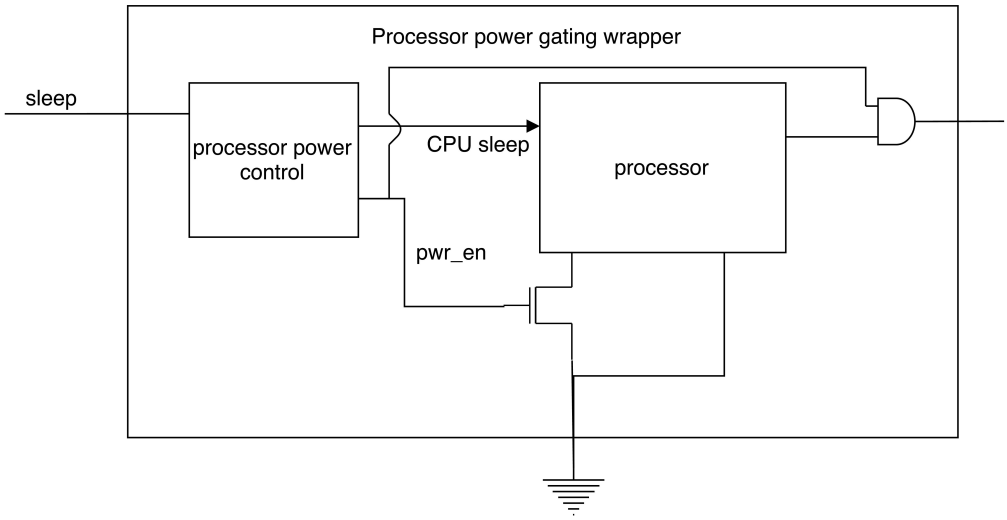

Figure 3.11: Processor power gating wrapper

processor power gating wrapper control the processor sleep signal and the power enable for the power gated processor. In figure 3.10 the power down and up transitions can be observed. In clock cycle 3 and 4 the processor is powered on to store all state data to the always on registers. In clock cycle 9 the processor is being powered on and a lot of dangerous signals have an unknown value. The processor is therefore given one more cycle with sleep asserted to set these signals to known values. The power-on sequence requires 2 cycles and is the total performance penalty per sleep period. If the sleep signal is deasserted within 2 cycles, the virtual ground is not turned off and therefore has no performance penalty.

3.1.7 C library

The RISC-V foundation maintains a C cross-compiler [5]. To enable usage of the peripheral modules on the microcontroller, a C driver library has been implemented.

Linker script and initialization code

The RISC-V cross compiler must be aware of the fact that the design is using a Harvard architecture and how large the memory of the target design is. A linker script was created and used to compile the C drivers and programs. The linker script is in Listing B.7. The C initialization code, or CRT0 was inserted before the program code to initiate the stack pointer. The CRT0 code is in Listing B.8.

SPI driver

A C SPI driver has been created to more easily use the SPI interface in a C-program. The library gives the programmer four functions; spi_write, spi_read, spi_settings and spi_status. Spi_settings is used to set the settings register in the spi controller, while spi_status returns the status register of the controller. spi_write initiate an SPI transfer and is non-blocking. Because it is non-blocking, other tasks can be performed while waiting for the SPI-transfer to be completed. A SPI transfer can take as much as 128 processor cycles. SPI_read is used to return the data registers in the SPI controller. All SPI driver function except the spi_status function check and wait if the spi controller is busy to avoid undefined usage of the SPI controller.

Clock counter driver

A C function was created to read the content of the 64 bit clock counter described earlier. The function returns the number of cycles since last reset as a uint64_t, a 64 bit unsigned integer.

Sleep controller driver

The system supports two sleep modes; timed sleep mode and SPI ready mode. In SPI ready mode, the processor is stalled until the SPI controller is no longer busy. This mode is used by spi_write, spi_read and spi_settings to ensure that SPI controller is ready when given a new task. In timed mode, the sleep controller halts the processor for a given amount of clock cycles. The sleep controller has a 32 bit clock counter and therefore the sleep function takes in a uint32_t value. To reduce instruction memory footprint, only a single function is defined in the sleep controller driver. If the sleep function is given the value 0 as input it starts a SPI ready sleep mode, otherwise a timed sleep mode is initiated.

3.1.8 Test programs

Sensor logger

To create realistic test scenario and to generate results for design comparisons a sensor logger program was written in C. The sensor logger test program simulates a program which reads data from a STMicroelectronics LIS3DH accelerometer, does some bit shifting and stores it on an Atmel

Phenomena	**Sample Rate** (in Hz)	**Sample Prec.** (in bits)
Low Frequency Band (< 100Hz)		
Ambient light level	0.017 - 1	16
Atmospheric temperature	0.017 - 1	16
Ambient noise level	0.017 - 1	16
Barometric pressure	0.017 - 1	8
Wind direction	0.017 - 1	8
Body temperature	0.1 - 1	8
Natural seismic vibration	0.2 - 100	8
Heart rate	0.8 - 3.2	1
Wind speed	1 - 10	16
Oral-nasal airflow	16 - 25	8
Blood pressure	50 - 100	8
Mid Frequency Band (100Hz - 1000Hz)		
Engine temperature & pressure	100 - 150	16
EEG (brain electrical activity)	100 - 200	16
EOG (eyeball electrical activity)	100 - 200	16
ECG (heart electrical activity)	100 - 250	8
High Frequency Band (> 1kHz)		
Breathing sounds	100 - 5k	8
EMG (skeletal muscle activity)	100 - 5k	8
Audio (human hearing range)	15 - 44k	16
Video (digital television)	10M	16
Fast A/D conversion	1G	8

Figure 3.12: Required sampling rate and data width for various phenomenon sensing From [3]

25320 EEPROM chip. LIS3DH is an ultra low power accelerometer with power consumption in the single digit micro watt region. This logging is performed with 20 and 2 ms period for a sampling rate of 50 and 500Hz. As seen in figure 3.12, these sampling rates cover a large range of possible applications. Three 16 bit values are read from the accelerometer, one for each accelerometer axis. The data is after some bit shifting stored on the EEPROM chip. It is is very light on processing, and the processor is mostly idle, waiting for the SPI transfer to finish. Even if it is very light on processing, it is still a realistic task for a processor in a sensor network. 2 ms is the shortest sampling period this microcontroller is capable of in this program, spending most of its time waiting for SPI transfers.

Short test

Short test is a test program designed to use time based sleep, do a few calculations and do a spi transfer in a short amount of execution time. It was created with SPICE simulation in mind, where

the simulation is very slow. The program finishes in a little over 200 μs and required 48 to 72 hours to simulate in SPICE. Longer tests were therefore not practical. With this test the power consumption in both sleep modes and during some general processor execution can be measured. The result of this test is used to estimate the power consumption of other test programs. The C code of this test is listed in Listing B.3.

3.1.9 Synthetisation

The flow of the synthetization and simulation of the microcontroller can be observed in figure 3.13. The VHDL files describing the microcontroller with an external memory interface was synthesized using the Synopsys design compiler (dc_shell) and the custom near-threshold library created by Ali. The synthetization script and design constraint file used with the power gated design can be observed in listing B.4 and listing B.5. The design compiler compiles the register transfer level VHDL code to a list of instantiated cells from the near-threshold library in a verilog format. As most of the cells are on a gate level, this can be described as a gate level netlist. The verilog netlist was imported by Virtuoso, a tool by cadence. Two voltage sources were manually added to the netlist in Virtuoso, VDD and PMOS substrate bias at 350 mV and -400 mV respectively. The netlist was combined with the transistor level description of the near-threshold library to generate a SPICE netlist in SPECTRE format. A python script was created to parse the SPICE netlist to extract cell count and cell area throughout the hierarchy of the design. A free data grapher javascript library was used to present the data in the web browser. Cell count or area was represented as area in a tree map. One could easily investigate further down the hierarchy by traversing down the tree graph. This tool is useful when selecting the size of the power gate for a block of cells, because the power requirement is dependent on number of cells it must supply. The tool is also useful in getting a overview of the design. ”How large portion of the design is the processor” and so forth. Table 4.1 show the number of cells and cell area of the synthesized design. Figure 4.1 and 4.2 shows the usage of the different standard cells with and without power gating respectively.

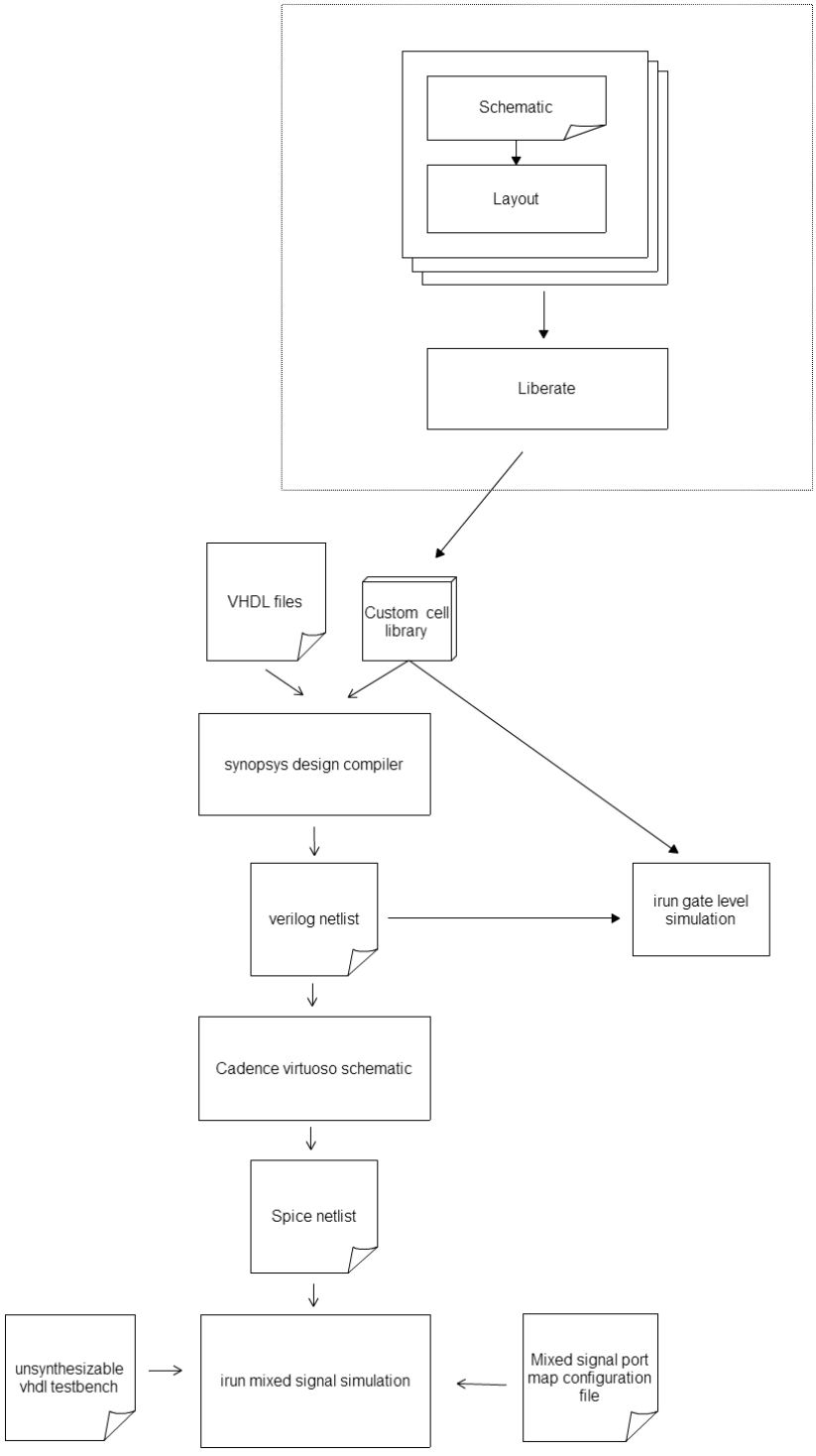

Figure 3.13: Synthetisation and simulation flow

3.2 Verification and benchmarking

3.2.1 Simulation

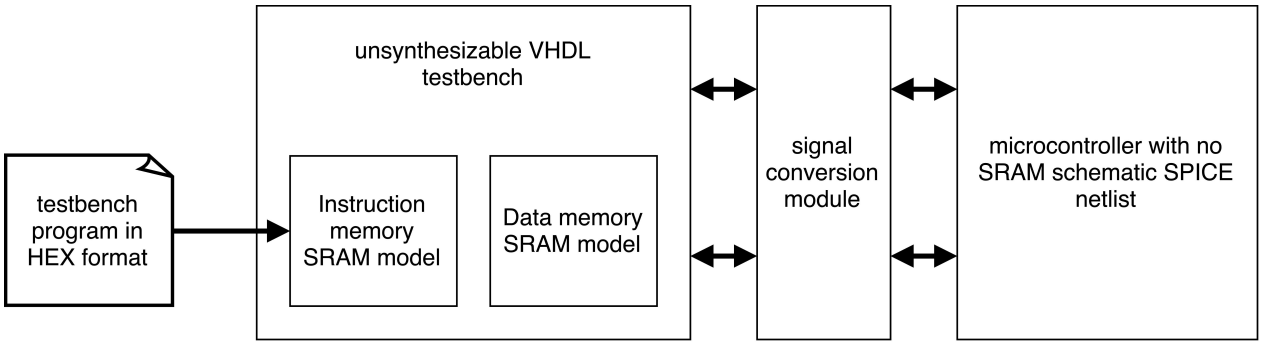

Figure 3.14: Schematic SPICE simulation test setup

The design was simulated at RTL level, gate level and schematic SPICE level. The full verification test suite included tests for the individual instructions supplied by the RISC-V foundation [12] and the custom tests that used the peripheral modules. The total simulation time for the verification test suite was a couple milliseconds. The simulation speed of the RTL and gate level is sufficient to perform a full run of the complete verification test suite in less than an hour. The SPICE simulation were relatively slow, and would require months to simulate the same tests. The SPICE simulation was therefore only used to measure power consumption of the design. The synthesized SPICE netlist was tested with an unsynthesizable VHDL testbench. Active HDL by Aldec was used to simulate the RTL. ncverilog was used to simulate the verilog netlist. A SPICE simulation requires a different simulator than the one running the VHDL testbench. The cadence command ”irun” is capable of executing two parallel simulations with some setup. irun was configured to use ultrasim for the SPICE simulation and ncvhdl for the VHDL simulation. ultrasim was used because it is faster than the other simulator SPECTRE. Because the SPICE simulation is analog and the VHDL is digital, conversion modules were created between each spice to VHDL and VHDL to SPICE connections (digital to analog and analog to digital connections). Figure 3.14 illustrate the test setup. The memory interface was routed to the output to enable this setup. A no startup signal was added to the microcontroller to force the startup controller to give control of the memory and SPI bus to the processor immediately. The startup sequence was not necessary because the program

was already loaded to the SRAM models in the VHDL testbench. A startup sequence is relatively time consuming and would not be feasible to simulate in SPICE.

The SRAM modules contain a large number of transistors, and required nearly two days of ultrasim simulation to write to all addresses. All the design tweaks performed in this project was performed on other parts of the design. The SRAM was therefore a constant in a changing design. It was therefore decided to simulate the SRAM by itself and measure power consumption of various usage patterns. This data was later combined with data from the RTL simulation to estimate the power consumption of the SRAM during realistic program execution.

3.2.2 Verification

The processor integrates a comparator which compares instructions in the instruction flow to a special pass and fail instruction. The pass and fail instruction is not part of the RISC-V instruction set and should only be added to test programs. If such an instruction is detected in the program flow, the processor is halted and a pass or fail signal is propagated to the output of the microcontroller. By integrating the comparator in the design, the microcontroller can do a self-test when a test program is loaded onto it. While testing it was found that a failing processor may ignore control transfers (ie branches) and increment the PC blindly. It is therefore important to place the pass instruction after the fail instruction in a test program and branch or jump past it, otherwise the processor may run in to the pass instruction while not functioning correctly. RISC-V has a set of tests that can be used to verify a RISC-V design [12]. These tests were used to verify the design both in behavioral simulation and in post-synthesis gate level simulation. The tests were not run on the SPICE netlist due to slow simulation speed as mentioned earlier. While the RISC-V project has tests that can verify if it conforms to the RISC-V standard, more tests were needed to verify the added functionality of the microcontroller (i.e. the SPI controller). The tests for the added microcontroller functionality used the C library.

3.2.3 Benchmarking

Runtime ALU power gating

A simple testbench was devised to test the different ALU configurations in SPICE. The test setup was similar to the SPICE simulation setup illustrated in Figure 3.14, but used only the schematic ALU SPICE netlist. The testbench generated random operands to the ALU, but the operators were cycled through in a non- random fashion. All tests used the same random seed to make the stimulus identical on all test runs. The output was checked to be correct and input was changed after 1 us, making it run on a 1 MHz clock frequency, the target frequency of this design. The testbench ran for 100 us. A script generated the power gates for the individual functional units in the ALU. The cell area was combined with a constant to select the appropriate power gate size. Five different constants were used to make 5 configurations. An always on footer power gate network was also inserted to include the effects coarse grain processor footer power gate network had the ALU. Because the smallest power network gave the lowest power consumption, it was used in the full chip simulation.

SRAM simulation

To test the power consumption of the SRAM a SPICE and VHDL co-simulation test was devised. The test setup was similar to the SPICE simulation of the rest of the microcontroller. An unsynthesizable VHDL testbench gave stimulus to the SRAM through a signal conversion unit. Because of the topology of the SRAM cell, SRAM cells may be floating and causing crowbar currents if they have not been written to. Therefore the testbench first wrote to all SRAM cells. When all cells had been written to, the testbench first stopped the clock for 10 clock periods, then it read 10 words, wrote 10 words and lastly it alternated between reading and writing words for 10 clock periods.

SPICE simulation of short test program

The design was simulated in SPICE without power gating, with coarse power gating of the processor and with both coarse power gating and runtime power gating of the ALU. The test setup is illustrated in Figure 3.14. Section 3.1.8 describes the short test. The power consumption data later presented in this report from timed sleep and SPI transfer sleep are from the lowest power state,

when the processor is power gated.

RTL simulation of sensor logger program

200 milliseconds of the sensor logger program were simulated in RTL with approximately 50 and 500Hz sampling rate to generate runtime statistics. The sampling rate is not precise because of how the sleep period is determined. These statistics are combined with clock by clock energy numbers from the SPICE simulation of the short test program to estimate the power consumption of this program running on this system. A classic RISC pipeline can execute up to one instruction per clock (IPC). By counting the empty pipeline stages (bubbles) during execution the number of instructions executed can be determined.

$$\frac{\text{energy per clock}}{\text{IPC}} = \text{energy per instruction} \tag{3.1}$$

By using Equation 3.1 the energy consumed per instruction can be calculated.

Chapter 4

Results

4.1 Synthesis cell count and area

Table 4.1 list the cell count and the estimated area of the synthesized design. The area numbers from the modules described in VHDL are the combined cell area. The core of the SRAM memory array was laid out by co-supervisor Even Låte, the layout of SRAM periphery elements remains to be implemented. To obtain an estimate of the total SRAM area including peripheral circuitry, a multiplication factor of 1.1 was used. Figure 4.1 and 4.2 show the standard cell usage of the processor with power gating and with no power gating respectively.

	with isolation cells		**without isolation cells**	
Module	**cell count**	**area [μm^2]**	**cell count**	**area [μm^2]**
Processor	18895	48358	17417	45701
SRAM (estimated)	N/A	77458	N/A	77458
Other (SPI controller, etc)	2910	7522	2915	7529
Total	21805	133338	20332	130688

Table 4.1: Cell count and cell area of synthesized design. The SRAM size is estimated from the size of the SRAM core size.

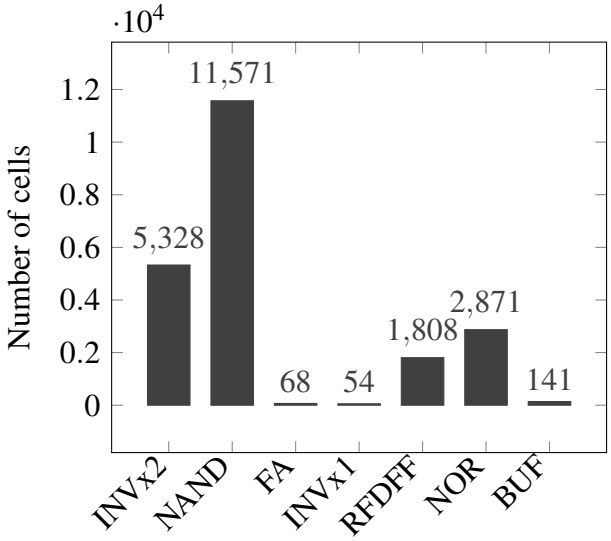

Figure 4.1: Distribution of cells in design with runtime and processor power gating. SRAM not included.

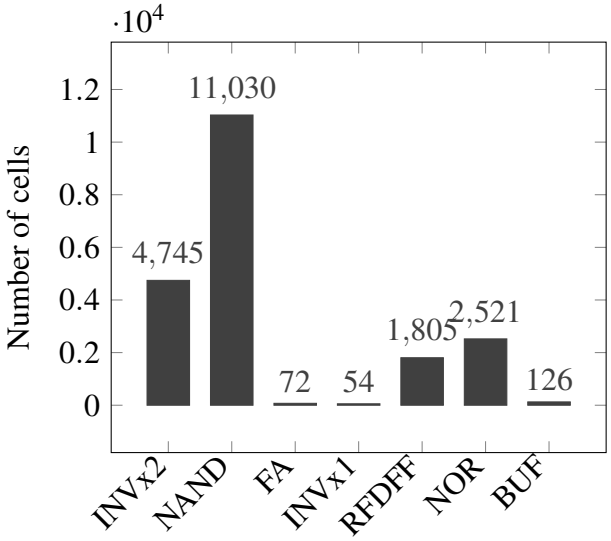

Figure 4.2: Distribution of cells in design with no power gating. SRAM not included.

4.2 Runtime ALU SPICE simulation

Table 4.2 list the gate width of the power gates used in the power gated ALU. How the gate sizes were selected is explained in Section 3.1.6. Size 1 corresponds to the smallest cell area to gate width constant, while size 5 is the largest. The gate length of the power transistors are 30 nm. Table 4.3 shows the average power consumption over 100 ALU operations with a 1 μs period. The operands are randomized, but because the same randomization seed was used in all simulation, the stimuli are identical in all tests. Figure 4.3 show the longest delay from the stimuli was changed to the correct data was on the output. In all tests this was the subtraction result. Figure 4.4 show the worst observed delay between a new operator and when the virtual VDD was 95% of the real VDD. Again, the subtraction unit was the slowest.

	area [μm^2]	**size 1** [μm]	**size 2** [μm]	**size 3** [μm]	**size 4** [μm]	**size 5** [μm]
adder	429	0.55	1.1	1.65	1.65	2.2
sra	1195	2.2	3.3	4.4	5.5	7.15
slt	344	0.55	0.55	1.1	1.65	1.65
sll	1123	2.2	3.3	4.4	5.5	6.6
subtractor	665	1.1	1.65	2.2	3.3	3.85
srl	1123	2.2	3.3	4.4	5.5	6.6
and	111	0.55	0.55	0.55	0.55	0.55
or	111	0.55	0.55	0.55	0.55	0.55
sltu	343	0.55	0.55	1.1	1.65	1.65
xor	288	0.55	0.55	1.1	1.1	1.65
total	5732	11	15.4	21.45	26.95	32.45

Table 4.2: Combined ALU power gate width, see Section 3.1.6 for more information

combined gate width [μm]	**average power consumption** [μW]	**relative**
No gating	0.433	1.000
11	0.1814	0.419
15.4	0.1826	0.422
21.45	0.1846	0.426
26.95	0.1863	0.430
32.45	0.1879	0.434

Table 4.3: Average ALU power consumption with random operands over 100 operations at 1μs interval

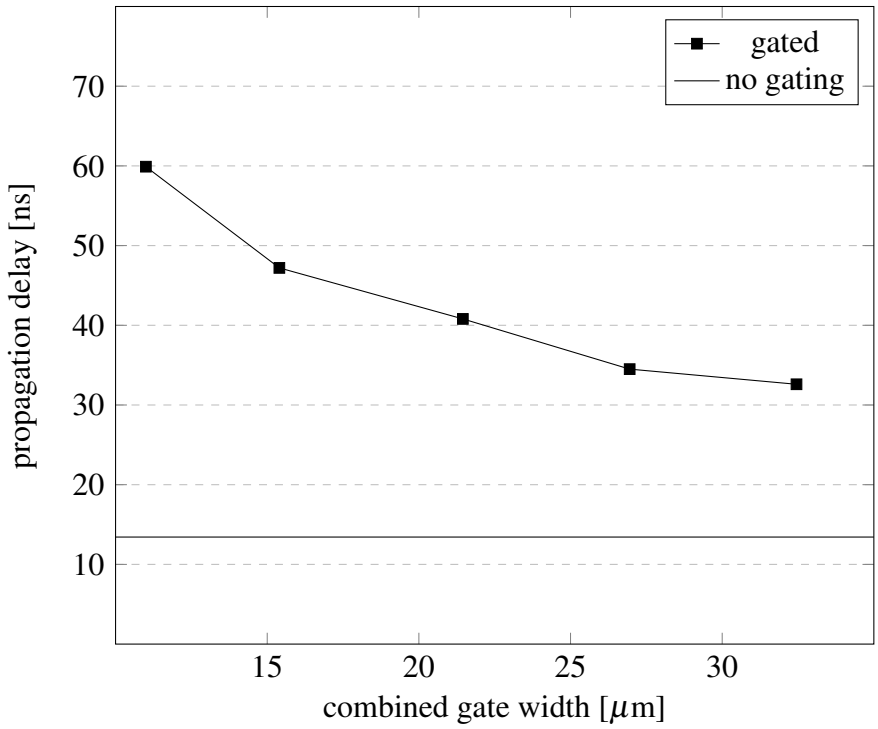

Figure 4.3: Longest propagation delay observed over 100 random operations

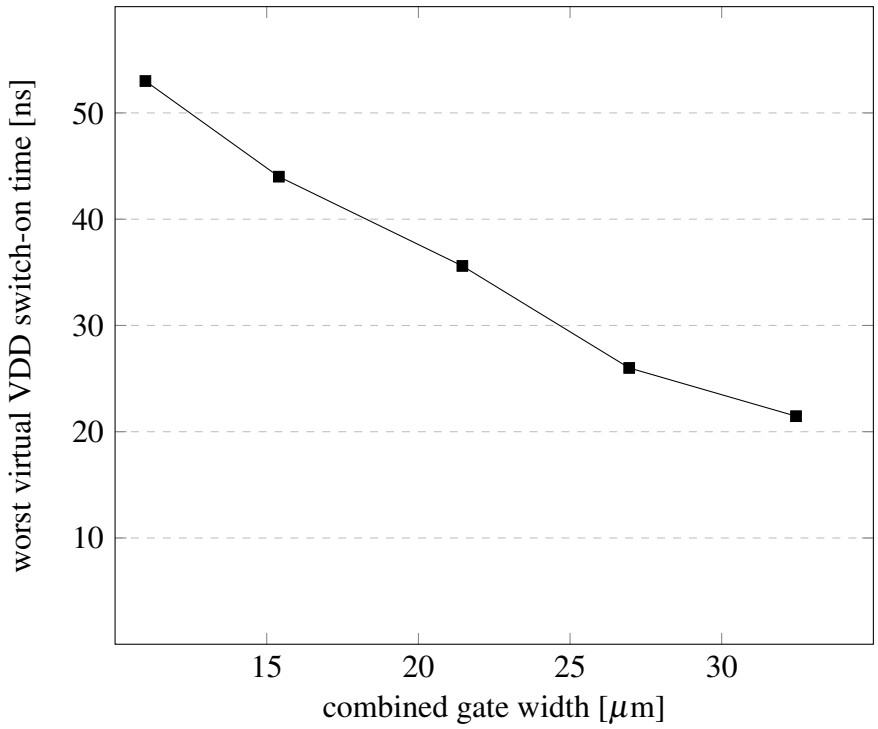

Figure 4.4: Worst observed virtual VDD switch-on (95% of VDD) time over 100 random operands

4.3 SRAM SPICE simulation

Table 4.4 list the power consumption of a single port 2kB SRAM memory with 32 bit word size. All power data is recorded after all memory locations has been written to, eliminating any crowbar current that may occur if any gates in the memory array are floating, due to the topology of the SRAM cell not revealed here. More information about this test in Section 3.2.3.

no clock [μW]	**reading** [μW]	**writing** [μW]	**alternating** [μW]
0.17	0.42	0.48	0.43

Table 4.4: Power consumption of a single port 2 kB SRAM memory with 32 bit word size

4.4 SPICE simulation of short test program

Figure 4.5 show the average power consumption during normal execution, timed sleep and in SPI sleep mode. In normal execution the processor is not entering sleep mode and is executing instructions as normal. In sleep mode the processor is waiting for the sleep controller to count to zero and in SPI transfer sleep the processor is waiting for an SPI transfer to complete. Figure 4.7 and 4.6 show the same test, but with processor gating only and processor gating and runtime ALU gating respectively. Figure 4.8 show the energy consumption per clock while the processor enters sleep, in sleep and when the processor wakes up from sleep. This data is used later to estimate power consumption of longer running, more complex programs.

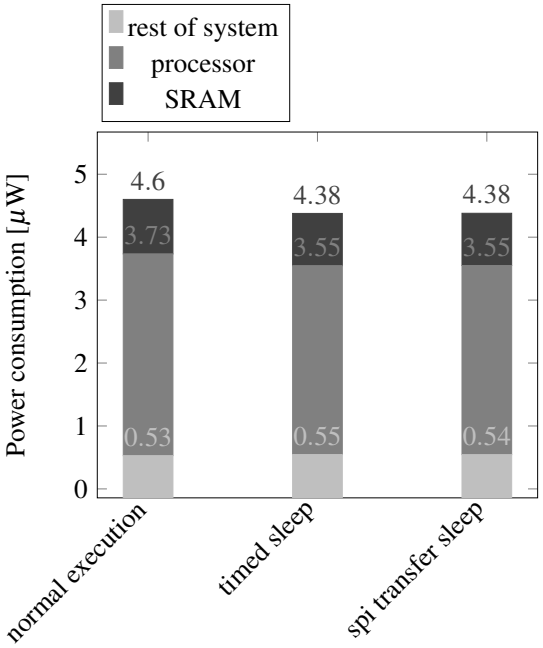

Figure 4.5: Power consumption reported during schematic SPICE simulation of SoC with no power gating running at 1 MHz

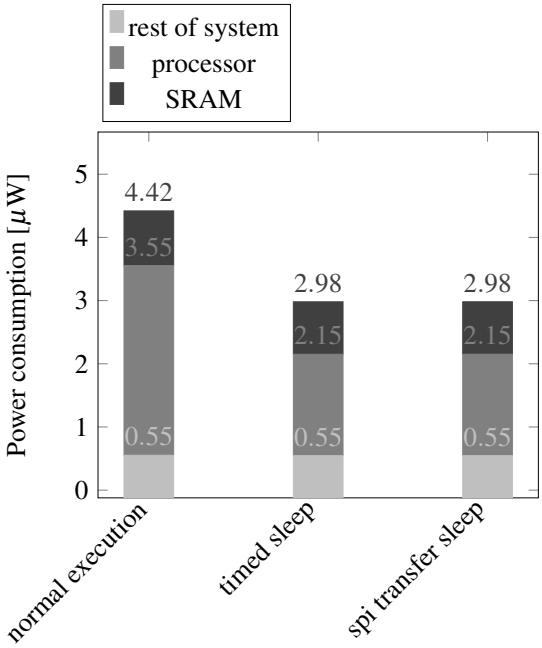

Figure 4.6: Power consumption reported during schematic SPICE simulation of SoC with both runtime and processor power gating running at 1 MHz

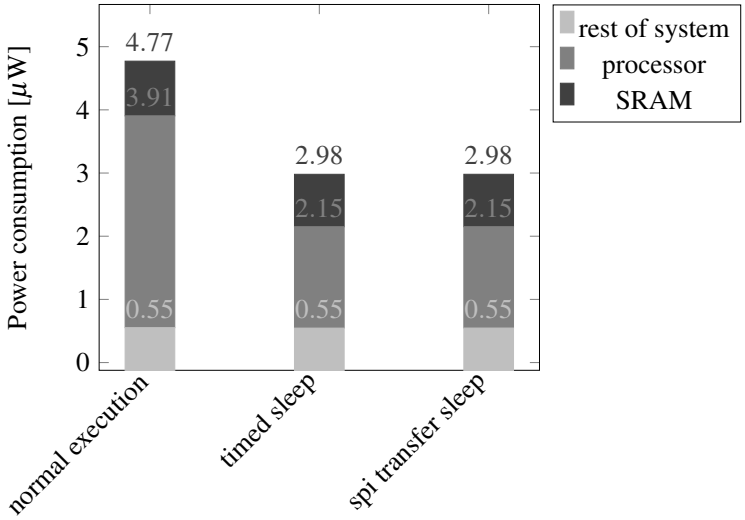

Figure 4.7: Power consumption reported during schematic SPICE simulation of SoC with only power gating of processor running at 1 MHz

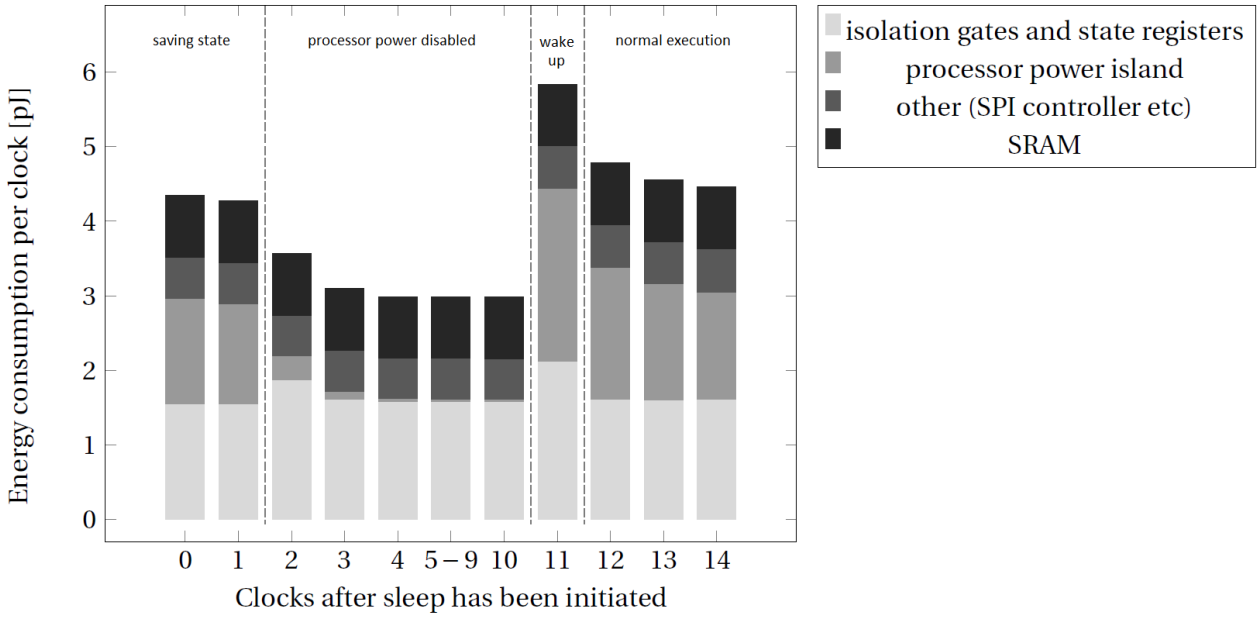

Figure 4.8: Energy consumption per clock during a complete sleep cycle in a system with both processor and ALU runtime gating running at 1 MHz.

4.5 RTL simulation of sensor logger test program

The runtime statistics of the sensor logging program can be observed in table 4.9 and 4.10 for 48.3 Hz and 483.8 Hz logging frequency respectively. Table 4.5 show the number of stalls, control transfers and the number of bubbles it creates in the pipeline during execution of the program. Figure 4.6 show that the average power consumption of the power gated microcontroller during program execution is 4.42 μW. With a clock speed of 1 MHz this translates into 4.42 pJ per clock. Equation 4.1 and 4.2 show the calculation of estimated energy per instruction during program execution of the low and high frequency sensor logging program. The equation used in the energy per instruction calculations is Equation 3.1. Figure 4.11 show the estimated average power consumption of these programs with both runtime ALU power gating and processor power gating.

Table 4.5: Pipeline statistics of sensor logging program RTL simulation

	49.8 Hz sensor logger		483.8 Hz sensor logger	
	count	resulting bubbles	count	resulting bubbles
control transfers	91	273	871	2613
stalls	387	387	3773	3773
total	478	660	4644	6346

Processor state	clocks	percentage
program execution	6641	3.3205
saving state 1	171	0.0855
saving state 2	171	0.0855
power gated 1	110	0.055
power gated 2	110	0.055
power gated 2+	192688	96.344
wake up	109	0.0545

Figure 4.9: 49.8 Hz sensor logger processor statistics over 200ms

Processor state	clocks	percentage
program execution	64120	32.06
saving state 1	1648	0.824
saving state 2	1648	0.824
power gated 1	1065	0.5325
power gated 2	1065	0.5325
power gated 2+	129390	64.695
wake up	1064	0.532

Figure 4.10: 483.8 Hz sensor logger processor statistics over 200ms

$$\frac{\frac{4.42\text{pJ}}{6641-660}}{6641} = 4.91\text{pJ per instruction} \tag{4.1}$$

$$\frac{\frac{4.42\text{pJ}}{64120-6346}}{64120} = 4.91\text{pJ per instruction} \tag{4.2}$$

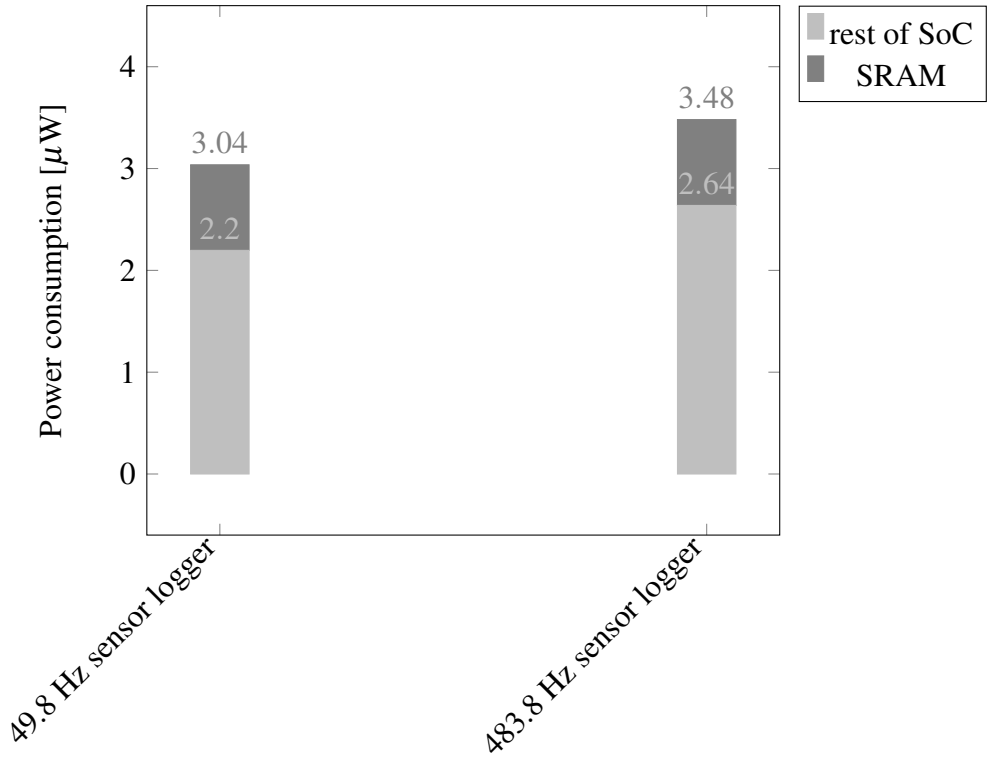

Figure 4.11: Estimated average power consumption of sensor logging program

Chapter 5

Discussion

5.1 Synthesis

As seen in Table 4.1 and 4.2 only 7 of the 8 cells in the standard cell library was used. The available cells in the standard library used are listed in Table 3.1. No flip-flops in the design used asynchronous set and therefore the standard cell with this functionality was not used. As seen in Figure 4.1 the addition of power gating isolation cells increased the cell area and cell count by 2% and 7% respectively. A small increase compared to the reduction in power as discussed later.

5.2 Runtime ALU power gating

The runtime power gating of the ALU were quite effective in reducing the runtime power consumption of the ALU, but it increased the propagation delay with a factor of more than 4 for the smallest power gating network. Because the SRAM limited the clock speed of the processor to 1 MHz the limiting factor in this design is not any critical path in the processor and therefore the added propagation is not an issue. The critical path reported by the Synopsys design compiler was only 30.39 ns when the effects of power gating are not considered. As the combined gate width of the power network decreases, the propagation delay increases. Although the propagation delay increases exponentially, the total delay through the ALU with the smallest power gating network is still less than 1/16 of the clock period. If the processor timings were more of an issue, a larger power gating network could be used to mitigate the effects of the power gating. The power enable

signal used by the power gates in the ALU are generated by a demultiplexer, and therefore the effect of increasing the power transistors will have an diminishing effect on the worst virtual VDD switch-on time as it approaches the delay through the demultiplexer. As seen when comparing figure 4.5, 4.7 and 4.6 the ALU runtime power gating enable the processor to consume less power during normal execution even with all the isolation cells added by the coarse grain power gating.

5.3 SPICE simulation of short test program

No power gating SPICE simulation

In Figure 4.5 the power consumption reported during SPICE simulation of the system without any power gating can be observed. There is less than 5% reduction in power consumption during sleep and normal execution. The difference is in how much power the processor is using. This can be attributed to lower switching activity in the processor as most registers in the processor have write disabled when in sleep mode.

Processor power gating SPICE simulation

In the simulation result using the microcontroller with processor power gating only, Figure 4.7, the average power consumption during normal execution is slightly higher than without power gating. This configuration used the same ALU operation separation as the runtime power gated configuration and therefore has no reuse of logic in the ALU, see Section 3.1.6 for more on this. Some of the increase in power consumption during normal execution should be attributed to this. The added isolation cells needed for the power gated processor also increase switching and leakage power during normal execution. When sleeping the power used by the processor is reduced by over 50%. This is because of the power footer network is switched off when the processor is sleeping. The power consumption in the rest of the system excluding SRAM is the same in all three scenarios. Under normal execution the memory address bus and memory space control has a higher switching activity. Under time based sleep the sleep counter registers are switching as they count to zero, and under SPI transfer sleep the SPI controller has a heightened activity.

Runtime ALU and processor gating SPICE simulation

In the SPICE simulation of the system with both runtime ALU and processor power gating, Figure 4.6, the power consumption during normal execution is reduced with 0.35 μW compared to the simulation with only processor power gating. The only difference between these two systems is the power gating of the ALU. The reduction in power consumption during normal execution is large enough to offset the added power consumption from the isolation cells, and uses less power during normal execution than the system with no power gating. The power consumption during timed sleep and SPI transfer sleep is identical to that of the system with power gating of the processor but no ALU power gating. This is because the ALU is using the same virtual ground as the rest of the processor and is powered off in both configurations during sleep periods.

5.4 Coarse grain processor gating

The number of registers and isolation gates needed to keep the state of the processor during deep sleep dampened the effect of the coarse grain power gating. The main power consumer in the processor while power gated is the register file. It contains 992 registers which state cannot be lost if the processor is to continue execution after being power gated. 992 registers is more than half of all registers in the design (Figure 4.1 and Figure 4.2). In figure 4.8 the energy consumed per clock can be observed. The first two clocks consume nearly as much energy as when the processor is running as normal. In the first clock cycle after turning off the processor footer power gates, the state register power consumption component rises slightly. It is likely that this is because the power enable signal to all isolation cells are switching at the same time. The power consumption quickly stabilizes and the current through the footer power gates approaches 0.1 μA. During the wake up cycle both the energy consumed by the state registers and the rest of the processor is higher. The increased energy consumption in the state registers is likely due to switching of the isolation cells. The increase in the rest of the CPU is due to now charged capacitance of the circuit being drained by the footer power gates.

5.5 Sensor logger power estimation

Because the sensor logger program uses the SPI bus a lot, the processor spends most of its time in a power gated state during program execution. In Table 4.9 the runtime statistics from the low sampling rate is listed. Only 3.32% of the time is spent executing instructions, while 96.34% of the cycles are spent in the lowest power state. Therefore the average power consumption is 31.1% lower than that of a microcontroller executing instructions (Figure 4.6). Table 4.10 list the runtime statistics of the higher frequency logging program. The time spent executing instructions are nearly ten times longer in the higher frequency logging program, but it still spends over half of the execution time in SPI sleep mode. As seen in equation 4.1 and 4.2 the energy per clock was identical in the two programs. Energy per instruction will vary with different programs as the number of control transfers and stalls do, and thus IPC will vary.

5.6 Clock tree and post-layout power consumption considerations

The clock tree power consumption has not been simulated in this thesis. The clock tree has been shown to account for 40% of the dynamic power [13]. Co-supervisor Ali Asghar Vatanjou has previous experience with post-layout simulation of sub and near-threshold circuits using STMicroelectronics 28 nm FD-SOI production technology, and by his accounts the post-layout simulations consume up to 70% more power than pre-layout schematic SPICE simulations. It is therefore difficult to compare the power consumption results from the schematic SPICE simulation to available microcontroller products.

5.7 Future work

Clock gating

Currently there is no clock gating in the microcontroller. The SRAMs has a control unit operating at a frequency 16 times higher than the rest of the system and therefore use a lot of dynamic power. As seen in table 4.4, the SRAM use 60% less power when clock gated over reading. In the programs

compiled in this project only a tiny fraction of the instruction flow are load or store instructions and should enable the data memory to disable the clock to memory for a significant energy saving. In sleep mode both the instruction memory and data memory can go in to no clock mode and save a significant share of the total power consumption. Clock gating the state retention registers in the processor while the rest of the processor is powered down could also save a significant amount of energy in sleep periods. The clock to the register file can be disabled when no data is written to it. This could save both area and power as the input mux to the registers require one less input.

Features

Currently the SPI controller only supports a single CHPA and CHPOL configuration, possibly making it incompatible with some SPI slaves. Implementing CHPA and CHPOL configurability would make the implementation more versatile.

Static program size can be reduced if the compressed instruction extension of RISC-V was implemented. It is as of this writing in the draft stage and claims to reduce the static code size with 20 to 30%. With a smaller static code size, larger programs can be stored in the instruction memory. Lowering the static code size will also reduce the average bandwidth required by the instruction memory. With a small instruction cache the processor could possibly operate at a higher clock frequency than the instruction memory, increasing performance.

The state data from the processor could be stored in SRAM when the processor is sleeping for longer periods. This technique could allow for a close to 1 μW power consumption during deep sleep.

Standard cell library

The standard cell library used only low threshold voltage transistors. Low threshold transistors have a lower propagation delay at a given voltage, but also a higher leakage current. The clock speed of this design was limited by the SRAM and high threshold transistors could have reduced power consumption and energy per operation. The critical path of the synthesized VHDL was much lower than the maximum operating speed of the SRAMs. By using higher voltage transistors, this imbalance could be reduced. Higher threshold transistors have lower leakage and would reduce power consumption. The standard library does not include a multiplexer, all multiplexers in the

design is made from other gates. Area and possibly energy consumption could be saved if this was included.

With a larger cell library the area and possibly power consumption of the synthesized design may have been lower. Cells with larger fan-in than two could also be beneficiary.

Chapter 6

Conclusion

In this thesis a 32 bit microcontroller has been implemented and synthesized using a near-threshold library. The microcontroller was verified at RTL and at post-synthesis gate level using a suite of verification programs from the RISC-V foundation and custom tests testing implementation specific functions. A linker script and C initialization code has been implemented to enable the compilation of C programs compatible with this microcontroller. The microcontroller has also been simulated in SPICE to obtain power figures. The power figures does not include the the clock tree and the power consumption is expected to be higher in post-layout simulations.

This thesis has shown that runtime power gating in near threshold digital circuits can provide significant power consumption savings, 58% in the case of this ALU, at the expense of propagation delay and area. It is therefore a promising technique to use in non-critical paths of a design. It has also been shown that scaling the power transistor network can greatly reduce the propagation delay penalty with only a slight increase in power consumption compared to a smaller power network.

A more traditional coarse grain power gating technique has been shown to reduce the power consumption during sleep periods. Because the RISC-V ISA require a large register file compared to the rest of the processor when implemented in a simplistic classic RISC pipeline, a large portion of the processor has to be powered on during sleep periods. The microcontroller used 32% less power when sleeping in the configurations using coarse grain gating of the processor. The coarse grain power gating was the largest contributor to reducing power consumption in the sensor logging program. In the sensor logging test program it was estimated that the microcontroller used 4.91 pJ per executed instruction.

Appendix A

Appendix A: Acronyms

I/O Input/Output

CMOS Complementary metal–oxide–semiconductor

PMOS p-channel MOSFET

NMOS n-channel MOSFET

CPI Clocks per instruction

RTL Register transfer level

VHDL Very high speed integrated circuit hardware description language

SPICE Simulation Program with Integrated Circuit Emphasis

SRAM Static Random Access Memory

CISC Complex Instruction Set Computer

RISC Reduced Instruction Set Computer

VLIW Very Long Instruction Word

ISA Instruction Set Architecture

CPU Central Processing Unit

FD-SOI Fully Depleted Silicon On Insulator

CPOL Clock Polarity

CPHA Clock phase

LSB Least Significant Bit

MSB Most Significant Bit

EEPROM Electrically Erasable Programmable Read-Only Memory

Appendix B

Appendix B: Code samples

Listing B.1: C driver implementation

```
#include "asoc.h"

#define SLEEPCYCLESADDR 0xFF8
#define SLEEPSPIFINISHEDADDR 0xFF9
#define RDTIMELOWERADDR 0xFFA
#define RDTIMEUPPERADDR 0xFFB
#define SPISETTINGSADDR 0xFFD
#define SPISTARTADDR 0xFFE
#define SPICLEARADDR 0xFFF
#define SPISTATUSADDR 0xFFC

uint32_t volatile * const sleep_cycles_p  = (uint32_t *) SLEEPCYCLESADDR;
uint32_t volatile * const sleep_spi_finished_p = (uint32_t *)
    SLEEPSPIFINISHEDADDR;
uint32_t volatile * const rdtime_lower_p  = (uint32_t *) RDTIMELOWERADDR;
uint32_t volatile * const rdtime_upper_p  = (uint32_t *) RDTIMEUPPERADDR;
uint32_t volatile * const spi_settings_p  = (uint32_t *) SPISETTINGSADDR;
uint32_t volatile * const spi_start_p     = (uint32_t *) SPISTARTADDR;
uint32_t volatile * const spi_clear_p     = (uint32_t *) SPICLEARADDR;
uint32_t volatile * const spi_status_p    = (uint32_t *) SPISTATUSADDR;


uint64_t rdcycles()
{
    uint64_t cycles = *rdtime_upper_p;
    cycles = cycles << 32;
    cycles = cycles + *rdtime_lower_p;
    return cycles;
}

inline void sleep(uint32_t cycles)
{
```

```
    if(cycles == TO_SPI_READY)
    {
        *sleep_spi_finished_p = cycles;
    }else
    {
        *sleep_cycles_p = cycles;
    }
}

void spi_write(uint32_t word)
{
    *sleep_spi_finished_p = 0;
    *spi_start_p = word;
}

void spi_settings(uint32_t setting)
{
    *sleep_spi_finished_p = 0;
    *spi_settings_p = setting;
}

uint32_t spi_read()
{
    uint32_t read_data;
    *sleep_spi_finished_p = 0;
    read_data = *spi_settings_p;
}

uint32_t spi_status()
{
    uint32_t read_data;
    read_data = *spi_status_p;
}

void spi_clear()
{
    *sleep_spi_finished_p = 0;
    *spi_clear_p = 0;
}
```

Listing B.2: Sensor logger program

```
#include "asoc.h"

#define BIT_CHECK(a,b) ((a) & (1<<(b)))

#define NEWDATA           2

#define READBYTE       0xc7 //0b11000111
#define SENSORSPISETTING 0x09 //0b00001001
#define EEPROMSPISETTING1 0x03 //0b00000010
#define EEPROMSPISETTING2 0x01 //0b00000001
#define EEPROMWRITEINSTR 0x02
#define PERIOD_MS        20
#define CLOCKPERIOD_US    1
#define EEPROM_SIZE    4096

uint16_t flip16(uint16_t word)
{
   return ((word & 0x00FF) << 8) | ((word & 0xFF00) >> 8);
}

void write_to_eeprom(uint16_t data[4], uint16_t address)
{
   uint8_t i;
   spi_settings(EEPROMSPISETTING1);
   spi_write((EEPROMWRITEINSTR << 16) | address);
   spi_settings(EEPROMSPISETTING2);
   for(i=0;i<4;i++)
   {
      spi_write(data[i]);
   }
   spi_clear();
}

void _start()
{
   const uint32_t PERIOD_CLKS = (1000 * CLOCKPERIOD_US) * PERIOD_MS;
   uint64_t clks;
   uint16_t sensor_status, eeprom_addr;
   uint16_t sensor_data[4];
   uint8_t i;

   eeprom_addr = 0;
   while(1)
   {
      clks = rdcycles();
      spi_settings(SENSORSPISETTING);
      spi_write(READBYTE);
      sensor_status = spi_read();
      if(BIT_CHECK(sensor_status,NEWDATA))
      {
```

```
        for(i=0;i<4;i++)
        {
           spi_write(0);
           sensor_data[i] = flip16(spi_read());
        }
        spi_clear();
        eeprom_addr = (eeprom_addr + 1) % EEPROM_SIZE;
        write_to_eeprom(sensor_data, eeprom_addr);
     }
     spi_clear();
     sleep(PERIOD_CLKS - (rdcycles() - clks));
   }
}
```

Listing B.3: Short test program

```
#include "asoc.h"

void fail();
void pass();

int _start()
{
   uint32_t a,b;
   uint64_t d;
   sleep(10);
   a = 5;
   b = 10;
   d = rdcycles();
   a = d + a;
   b = d + b;
   spi_settings(0);
   spi_write(a);
   spi_clear();
   sleep(0);
   pass();
   fail();
   return 0;
}
```

Listing B.4: Synopsys design compiler synthetization script

```
# Set parameter units
#set_units -time ns
#set_operating_conditions worst
#
set TOPLEVEL "soc_top"
set link_library "~/S28_coreLVTPB4_AFBB_ecsm_350mV.db"
set search_path ~/
set target_library "~/S28_coreLVTPB4_AFBB_ecsm_350mV.db"
```

```
set fileFormat vhdl

read_vhdl "~/newWorkdir/riscvsubthreshold/modules/sleep_reg_pg_wrap.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/clock_counter_pg_reg.vhd"
read_vhdl
    "~/newWorkdir/riscvsubthreshold/modules/instruction_decode/register_file_regs_wrap.vhd"
read_vhdl
    "~/newWorkdir/riscvsubthreshold/modules/instruction_decode/register_file.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/branch_predictor_reg_wrap.vhd"
read_vhdl
    "~/newWorkdir/riscvsubthreshold/modules/instruction_fetch/branch_predictor.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/instruction_decode/control.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/constants.vhdl"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/instruction_fetch/PC/PC_reg.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/branch_predictor_pg.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/clock_divider_cnt.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/write_back/clock_counter.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/spi_startup.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/SPI_top3.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/SPI_controller.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/activity_control.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/execute/ALU/alu_adder.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/execute/ALU/alu_and.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/execute/ALU/alu_bpt.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/execute/ALU/alu_or.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/execute/ALU/alu_sll.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/execute/ALU/alu_slt.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/execute/ALU/alu_sltu.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/execute/ALU/alu_sra.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/execute/ALU/alu_srl.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/execute/ALU/alu_subtractor.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/execute/ALU/alu_xor.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/execute/ALU/alu.vhdl"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/execute/forwarder.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/execute/execute.vhd"
read_vhdl
    "~/newWorkdir/riscvsubthreshold/modules/instruction_decode/hazard_detector.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/instruction_decode/imm_ext.vhd"
read_vhdl
    "~/newWorkdir/riscvsubthreshold/modules/instruction_fetch/branch_target_adder.vhd"
read_vhdl
    "~/newWorkdir/riscvsubthreshold/modules/instruction_fetch/PC/PC_increment.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/instruction_fetch/PC/PC.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/memory/branch_control.vhd"
read_vhdl
    "~/newWorkdir/riscvsubthreshold/modules/pipeline_registers/EXMEM_preg.vhd"
read_vhdl
    "~/newWorkdir/riscvsubthreshold/modules/pipeline_registers/IFID_preg.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/write_back/write_back.vhd"
read_vhdl
    "~/newWorkdir/riscvsubthreshold/modules/pipeline_registers/MEMWB_preg.vhd"
```

```
read_vhdl
    "~/newWorkdir/riscvsubthreshold/modules/pipeline_registers/IDEX_preg.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/data_mem_wrap.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/instr_mem_wrap.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/memory/memory.vhd"
read_vhdl
    "~/newWorkdir/riscvsubthreshold/modules/instruction_decode/instruction_decode.vhd"
read_vhdl
    "~/newWorkdir/riscvsubthreshold/modules/instruction_fetch/instruction_fetch.vhd"
read_vhdl
    "~/newWorkdir/riscvsubthreshold/modules/instruction_fetch/two_level_bp.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/top.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/top_pg_wrap.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/data_mem_sram_model.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/instr_mem_sram_model.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/sleep_controller.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/timer.vhd"
read_vhdl "~/newWorkdir/riscvsubthreshold/modules/soc_top.vhd"
read_sdc "~/riscvsubthreshold/constraints.sdc"

current_design soc_top

#create_clock -name $CLK -period 1000.0 -waveform {0 500} [get_ports clk]
set high_fanout_net_threshold 10000
uniquify
change_names -rules verilog -hierarchy

compile

check_design

all_high_fanout -nets
write -f verilog -o soc_top_pg.v -hierarchy
```

Listing B.5: Synopsys design compiler constraints file

```
# Set parameter units
set_units -time ns
set_units -capacitance pF

# Set clock conditions
create_clock -name {CLK} -period 62.5 -waveform {0 31.25} [get_ports {clk}]
create_clock -name {CLKMHZ} -period 1000.0 -waveform {0 500} [get_ports {d_clk}]

#link -all
# Set load capacitance to output node(s)
set_load 0.1 [all_outputs]
```

Listing B.6: VHDL of ALU with no power gating

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;

library work;
use work.constants.all;


entity alu is
port (
    A, B     : in std_logic_vector(31 downto 0);
    operation : in std_logic_vector(ALU_OPCODE_WIDTH - 1 downto 0);

    result   : out std_logic_vector(31 downto 0)
);
end alu;

architecture alu_arch of alu is
signal add_res, sub_res, and_res, or_res, xor_res,
     sll_res, sra_res, srl_res, slt_res, sltu_res,
     b_pass_through_res : signed(31 downto 0);
begin


combi: process(A, B)
begin
    add_res <= signed(A) + signed(B);
    sub_res <= signed(A) - signed(B);
    if(signed(A) < signed(B))  then slt_res <= x"00000001";
    else                           slt_res <= x"00000000";
    end if;
    if(unsigned(A) < unsigned(B)) then sltu_res <= x"00000001";
    else                           sltu_res <= x"00000000";
    end if;

    and_res <= signed(A and B);
    or_res <= signed(A or B);
    xor_res <= signed(A xor B);


    sll_res <= signed(std_logic_vector(shift_left(unsigned(A),
       to_integer(unsigned(B(4 downto 0))))));
    srl_res <= signed(std_logic_vector(shift_right(unsigned(A),
       to_integer(unsigned(B(4 downto 0))))));
    sra_res <= shift_right(signed(A), to_integer(unsigned(B(4 downto 0))));

   b_pass_through_res <= signed(B);
end process;

mux: process(add_res, sub_res, srl_res, sra_res, sltu_res, sll_res, slt_res,
    and_res, or_res, xor_res, operation, b_pass_through_res)
```

```
begin

    case operation is
        when ALU_ADD_OPCODE   => result <= std_logic_vector(add_res);
        when ALU_SUB_OPCODE   => result <= std_logic_vector(sub_res);
        when ALU_SRL_OPCODE   => result <= std_logic_vector(srl_res);
        when ALU_SRA_OPCODE   => result <= std_logic_vector(sra_res);
        when ALU_SLTU_OPCODE  => result <= std_logic_vector(sltu_res);
        when ALU_SLL_OPCODE   => result <= std_logic_vector(sll_res);
        when ALU_SLT_OPCODE   => result <= std_logic_vector(slt_res);
        when ALU_AND_OPCODE   => result <= std_logic_vector(and_res);
        when ALU_OR_OPCODE    => result <= std_logic_vector(or_res);
        when ALU_XOR_OPCODE   => result <= std_logic_vector(xor_res);
      when ALU_B_PASS_OPCODE => result <= std_logic_vector(b_pass_through_res);
        when others           => result <= x"00000000";
    end case;
end process;


end ALU_arch;
```

Listing B.7: Linker script

```
OUTPUT_ARCH( "riscv" )
ENTRY( _start )
MEMORY
  {
  imem(x) : ORIGIN = 0, LENGTH = 2K
  dmem(rw) : ORIGIN = 0x800, LENGTH = 2K
  }
SECTIONS
  {
 .lib 0x80 :
 {
 *(.lib)
 }
 /* text: code section */
 .text 0x200 :
 {
 *(.crt)
 *(.text)
 }
 /* data: Initialized data segment */
 .data 0x800 :
 {
 *(.data)
 *(.sdata)
 }
 /* End of uninitalized data segement */
 _end = .;
 }
```

Listing B.8: Linker script

```
   .file   "asoc_init.c"
       .section .crt
   .align 2
   .globl _init
   .type _init, @function
_init:
   li    sp, 2048
   j     _start

   .weak main
main:
   j     _start

   .weak _start
_start:
   j     main
```

Listing B.9: VHDL code describing the top module of the microcontroller with power gating and an external memory interface.

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

library work;
use work.constants.all;


entity soc_top is
    port(
        clk         : in std_logic;
        d_clk       : in std_logic;
        nreset      : in std_logic;

        --testbench
        pass        : out std_logic;
        fail        : out std_logic;
        --spi--
        sclk        : out std_logic;
        miso        : in  std_logic;
        mosi        : out std_logic;
        cs1         : out std_logic;
        cs2         : out std_logic;
        cs3         : out std_logic;
        cs4         : out std_logic;

        --reroute for fast sim--
        skip_startup                : in std_logic;
        d_clk_out                   : out std_logic;
        --instr-mem-reroute--
        instr_mem_write_en          : out std_logic;
        instr_mem_Address           : out std_logic_vector(INSTRUCTION_MEM_WIDTH
            - 1 downto 0);
        instr_mem_write_data_input  : out std_logic_vector(31 downto 0);
        instr_mem_read_data           : in std_logic_vector(31 downto 0);
        instr_mem_reset_pulse_generator : out std_logic;
        instr_mem_idle              : in std_logic;
        --instr-mem-reroute--
        data_mem_write_en           : out std_logic;
        data_mem_Address            : out std_logic_vector(INSTRUCTION_MEM_WIDTH
            - 1 downto 0);
        data_mem_write_data_input   : out std_logic_vector(31 downto 0);
        data_mem_read_data          : in std_logic_vector(31 downto 0);
        data_mem_be                 : out std_logic_vector(1 downto 0);
        data_mem_reset_pulse_generator : out std_logic;
        data_mem_idle               : in std_logic
        );
end entity;
```

```
architecture behave of soc_top is
    --test interface
    signal pass_i, pass_i_reg : std_logic;
    signal fail_i, fail_i_reg : std_logic;


    --SPI and startup
    signal spi_settings   :  std_logic;
    signal spi_data_in    :  std_logic_vector(31 downto 0);
    signal spi_data_out   :  std_logic_vector(31 downto 0);

    signal spi_busy       :  std_logic;
    signal spi_finished   :  std_logic;
    signal spi_clear      :  std_logic;
    signal spi_start      :  std_logic;

    signal startup_data_mem : std_logic_vector(31 downto 0);
    signal startup_address : std_logic_vector(DATA_MEM_WIDTH - 3 downto 0);
    signal startup_we     : std_logic;
    signal startup_done   : std_logic;

    signal clk_reset,
    reset_last            : std_logic;
    signal sync_reset     : std_logic;
    --memory
    signal instr_we       : std_logic;
    signal instr_data_w   : std_logic_vector(31 downto 0);
    signal instr_data_r   : std_logic_vector(31 downto 0);
    signal instr_addr     : std_logic_vector(INSTRUCTION_MEM_WIDTH - 1 downto 0);
    signal instr_re       : std_logic;
    signal instr_idle     : std_logic;


    signal data_we        : std_logic;
    signal data_mem_we    : std_logic;
    signal data_data_w    : std_logic_vector(31 downto 0);
    signal data_data_r    : std_logic_vector(31 downto 0);
    signal data_addr      : std_logic_vector(SPI_AND_DATA_MEM_WIDTH - 1 downto 0);
    signal data_be        : std_logic_vector(1 downto 0);
    signal data_re        : std_logic;

    signal data_mem_data_r : std_logic_vector(31 downto 0);
    signal data_mem_re    : std_logic;
    signal data_busy      : std_logic;
    signal data_idle      : std_logic;
```

```
signal startup_instr_addr : std_logic_vector(INSTRUCTION_MEM_WIDTH - 1 downto
    0);
signal startup_data_addr : std_logic_vector(SPI_AND_DATA_MEM_WIDTH - 1 downto
    0);
signal startup_instr_we : std_logic;
signal startup_data_we  : std_logic;

signal startup_spi_data_in : std_logic_vector(31 downto 0);
signal startup_spi_data_out : std_logic_vector(31 downto 0);
signal startup_spi_finished : std_logic;
signal startup_spi_start : std_logic;
signal startup_spi_clear : std_logic;
signal startup_spi_settings : std_logic;

signal cpu_instr_we      : std_logic;
signal cpu_instr_data_w  : std_logic_vector(31 downto 0);
signal cpu_instr_data_r  : std_logic_vector(31 downto 0);
signal cpu_instr_addr    : std_logic_vector(INSTRUCTION_MEM_WIDTH - 1 downto
    0);
signal cpu_instr_re      : std_logic;

signal cpu_data_we       : std_logic;
signal cpu_data_data_w     : std_logic_vector(31 downto 0);
signal cpu_data_addr     : std_logic_vector(SPI_AND_DATA_MEM_WIDTH - 1 downto
    0);
signal cpu_data_be       : std_logic_vector(1 downto 0);
signal cpu_data_re       : std_logic;

signal cpu_spi_data_in   : std_logic_vector(31 downto 0);
signal cpu_spi_data_out  : std_logic_vector(31 downto 0);
signal cpu_spi_finished  : std_logic;
signal cpu_spi_start     : std_logic;
signal cpu_spi_clear     : std_logic;
signal cpu_spi_settings  : std_logic;

signal data_to_spi                : std_logic_vector(31 downto 0);

signal mem_data_n, mem_data_seq    : std_logic_vector(31 downto 0);


signal mem_data_src, mem_data_src_n : std_logic;

signal sleep_we, sleep_type, sleep_ctl : std_logic;

signal timer                      : std_logic_vector(63 downto 0);

--CPU
signal cpu_sleep : std_logic;
--testsignal
signal mem_re_last : std_logic;
```

```
    --pragma synthesis_off
    signal sleep_cnt    : integer := 0;
    signal awake_cnt    : integer := 0;
    signal dmem_read_cnt : integer := 0;
    signal dmem_write_cnt : integer := 0;
    signal dmem_idle_cnt : integer := 0;
    --pragma synthesis_on
begin
    pass <= pass_i_reg;
    fail <= fail_i_reg;
    d_clk_out <= d_clk;

    cpu_sleep <= (not startup_done) or (pass_i_reg or fail_i_reg) or sleep_ctl;




    sync_reset_process : process(d_clk)
    begin
        if(d_clk'event and d_clk = '1') then
            sync_reset <= nreset;
        end if;
    end process;

    test_process : process(d_clk)
    begin
        if(d_clk'event and d_clk = '1') then
            if(nreset = '0') then
                pass_i_reg <= '0';
                fail_i_reg <= '0';
            else
                pass_i_reg <= pass_i;
                fail_i_reg <= fail_i;
            end if;
        end if;
    end process;

    startup_instr_addr <= startup_address(DATA_MEM_WIDTH - 3 downto 0) & "00";
    startup_data_addr <= '0' & startup_address(DATA_MEM_WIDTH - 3 downto 0) &
        "00";

    cpu_spi_data_in <= cpu_data_data_w;
    startup_spi_finished <= spi_finished;
    startup_spi_data_out <= spi_data_out;
    cpu_spi_data_out <= spi_data_out;

    memory_control_selector : process(startup_done, startup_instr_addr,
        startup_data_addr, startup_instr_we, startup_data_we, startup_data_mem,
        startup_data_mem, cpu_instr_addr, cpu_data_addr, cpu_instr_we,
        cpu_data_we, cpu_data_be, cpu_instr_data_w, cpu_data_re, cpu_instr_re,
        cpu_data_data_w, startup_spi_data_in, startup_spi_start,
```

```
        startup_spi_clear,
    startup_spi_settings, cpu_spi_data_in , cpu_spi_start, cpu_spi_clear,
        cpu_spi_settings)
begin
    case (startup_done) is
        when '0' =>
            instr_addr   <= startup_instr_addr;
            data_addr    <= startup_data_addr;
            instr_we     <= startup_instr_we;
            data_we      <= startup_data_we;
            data_be      <= "10";
            instr_data_w <= startup_data_mem;
            data_data_w  <= startup_data_mem;
            data_re      <= '0';
            instr_re     <= '0';
            data_to_spi  <= startup_spi_data_in;
            spi_start    <= startup_spi_start;
            spi_clear    <= startup_spi_clear;
        spi_settings <= startup_spi_settings;
        when '1' =>
            instr_addr   <= cpu_instr_addr;
            data_addr    <= cpu_data_addr;
            instr_we     <= cpu_instr_we;
            data_we      <= cpu_data_we;
            data_be      <= cpu_data_be;
            instr_data_w <= cpu_instr_data_w;
            data_data_w  <= cpu_data_data_w;
            data_re      <= cpu_data_re;
            instr_re     <= cpu_instr_re;
            data_to_spi  <= cpu_spi_data_in;
            spi_start    <= cpu_spi_start;
            spi_clear    <= cpu_spi_clear;
        spi_settings <= cpu_spi_settings;
        when others =>
        NULL;
    end case;
end process;

memory_map : process(mem_data_src, mem_data_seq, data_mem_data_r)
begin
    case (mem_data_src) is
        when '1'  =>
        data_data_r   <= mem_data_seq;
        when '0' =>
        data_data_r   <= data_mem_data_r;
        when others =>
            null;
    end case;
end process;
```

```
spi_status_regs : process(d_clk)
begin
if(d_clk'event and d_clk = '1') then
mem_data_seq <= mem_data_n;
mem_data_src <= mem_data_src_n;
mem_re_last <= data_mem_re;
end if;

end process;


mem_map_decode : process( data_addr, data_re, data_we, spi_data_out,
    data_mem_data_r, spi_busy, mem_data_src)

begin
    data_mem_we   <= data_we;
    data_mem_re   <= data_re;
    data_busy     <= '0';
    mem_data_src_n <= '0';

    cpu_spi_settings <= '0';
    cpu_spi_start  <= '0';
    cpu_spi_clear  <= '0';
    mem_data_n <= spi_data_out;

    sleep_we <= '0';
    sleep_type <= data_addr(0);


    case (data_addr(SPI_AND_DATA_MEM_WIDTH - 1 downto 4)) is
        when RESERVED_ADDR_SPACE =>
            data_mem_we   <= '0';
            data_mem_re   <= '0';
            mem_data_src_n <= '1';
            case (data_addr(2)) is
                when '1'  => --SPI address space
                    data_busy <= spi_busy;
                    case (data_addr(1 downto 0)) is
                        when "00" =>
                            mem_data_n <= x"0000000" & "00" & spi_finished &
                                spi_busy; --address: FFC
                        when SPI_settings_OP =>
                            cpu_spi_settings <= data_we;          --address: FFD
                        when SPI_START_OP  =>
                            cpu_spi_start   <= data_we;           --address: FFE
                        when SPI_clear_OP  =>
                            cpu_spi_clear   <= data_we;           --address: FFF
                        when others => NULL;
```

```
                end case;
            when '0' =>
                case(data_addr(1)) is
                    when '0' => --sleep control address space
                        sleep_we <= data_we;

                    when '1' => --timer address space
                        case (data_addr(0)) is
                            when '0' =>
                                mem_data_n <= timer(31 downto 0);
                            when '1' =>
                                mem_data_n <= timer(63 downto 32);
                            when others =>
                                null;
                        end case;
                    when others =>
                        null;
                end case;
            when others =>
                null;
            end case;
        when others =>
            null;
    end case;
end process;

sleep_controller : entity work.sleep_controller port map(
    clk         => d_clk,
    nreset      => sync_reset,
    data        => cpu_data_data_w,
    we          => sleep_we,
    spi_finished => spi_finished,
    spi_busy    => spi_busy,
    sleep_type  => sleep_type,
    sleep       => sleep_ctl
    );
time_module : entity work.timer port map(
    clk    => d_clk,
    timer  => timer,
    nreset => sync_reset
    );



spi_controller : entity work.SPI_controller port map(
    clk       => d_clk,
    nreset    => sync_reset,
    request   => spi_settings,
    data_in   => data_to_spi,
    data_out  => spi_data_out,
```

```
    busy      => spi_busy,
    finished  => spi_finished,
    clear     => spi_clear,
    start     => spi_start,

    --SPI interface
    miso      => miso,
    mosi      => mosi,
    sclk      => sclk,
    cs1       => cs1,
    cs2       => cs2,
    cs3       => cs3,
    cs4       => cs4
    );

startup_controller : entity work.spi_startup port map(
    clk       => d_clk,
    nreset    => sync_reset,
    skip_startup => skip_startup,

    --SPI controller interface--
    data_to_spi => startup_spi_data_in,
    data_from_spi => startup_spi_data_out,
    spi_finished => startup_spi_finished,
    spi_start => startup_spi_start,
    spi_clear => startup_spi_clear,
    spi_settings => startup_spi_settings,


    --Memory interface--
    data_mem => startup_data_mem,
    address => startup_address,
    instr_we => startup_instr_we,
    data_we => startup_data_we,

    done    => startup_done

    );

instruction_memory : entity work.instr_mem_wrap generic map(
    address_width => INSTRUCTION_MEM_WIDTH)
port map(
    d_clk => d_clk,
    clk   => clk,
    reset_pulse_generator => sync_reset,
    write_en => instr_we,
    Address  =>  instr_addr,
    write_data_input => instr_data_w,
    read_data => instr_data_r,
    idle => instr_idle,
    --rerouting for no startup--
```

```
    reroute_write_en => instr_mem_write_en,
    reroute_Address => instr_mem_Address,
    reroute_write_data_input => instr_mem_write_data_input,
    reroute_read_data => instr_mem_read_data,
    reroute_reset_pulse_generator => instr_mem_reset_pulse_generator,
    reroute_idle => instr_mem_idle
    );

data_memory : entity work.data_mem_wrap generic map(
    address_width => INSTRUCTION_MEM_WIDTH)
port map(
    d_clk               => d_clk,
    clk                 => clk,
    be                  => data_be,
    reset_pulse_generator => sync_reset,
    write_en            => data_mem_we,
    Address             =>  data_addr(DATA_MEM_WIDTH - 1 downto 0),
    write_data_input    => data_data_w,
    read_data           => data_mem_data_r,
    idle                => data_idle,
    --rerouting for no startup--
    reroute_write_en    => data_mem_write_en,
    reroute_Address     => data_mem_Address,
    reroute_write_data_input => data_mem_write_data_input,
    reroute_read_data   => data_mem_read_data,
    reroute_reset_pulse_generator => data_mem_reset_pulse_generator,
    reroute_idle        => data_mem_idle
    );


AAsmund_RISC : entity work.top_pg_wrap port map
    (
    clk => d_clk,
    nreset => sync_reset,

    sleep => cpu_sleep,

    --test interface
    pass  => pass_i,
    fail  => fail_i,

    --data memory interface
    data_memory_address    => cpu_data_addr,
    data_memory_read_data  =>   data_data_r,
    data_memory_be         =>   cpu_data_be,
    data_memory_write_data => cpu_data_data_w,
    data_memory_write_enable => cpu_data_we,
    data_memory_read_enable => cpu_data_re,

    --instruction memory interface
    inst_memory_address    => cpu_instr_addr,
```

```
        inst_memory_read_data  => instr_data_r,
        inst_memory_write_enable => cpu_instr_we
        );

    cpu_instr_data_w <= x"00000000";

    cpu_instr_re <= '1';

    --pragma synthesis_off
    process(d_clk)
    begin
        if(d_clk = '1' and d_clk'event) then
            if(sync_reset = '0') then
                sleep_cnt <= 0;
                awake_cnt <= 0;
                dmem_read_cnt <= 0;
                dmem_write_cnt <= 0;
                dmem_idle_cnt <= 0;
            else
                if(cpu_data_re = '1') then
                    dmem_read_cnt <= dmem_read_cnt + 1;
                elsif(cpu_data_we = '1') then
                    dmem_write_cnt <= dmem_write_cnt + 1;
                else
                    dmem_idle_cnt <= dmem_idle_cnt + 1;
                end if;
                if(cpu_sleep = '1') then
                    sleep_cnt <= sleep_cnt + 1;
                else
                    awake_cnt <= awake_cnt + 1;
                end if;
            end if;
        end if;

    end process;

    --pragma synthesis_on
end behave;
```

Bibliography

[1] Alice Wang, Benton Highsmith Calhoun, Anantha P. Chandrakasan, *SUB-THRESHOLD DESIGN FOR ULTRA LOW-POWER SYSTEMS*. Springer, 2006.

[2] Mingoo Seok, Gregory Chen, Scott Hanson, Michael Wieckowski, David Blaauw, and Dennis Sylvester, "Mitigating variability in near-threshold computing," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, pp. 42–49, March 2011.

[3] Leyla Nazhandali, Bo Zhai, Javin Olson, Anna Reeves, Michael Minuth, Ryan Helfand, Sanjay Pant, Todd Austin and David Blaauw, "Energy optimization of subthreshold-voltage sensor network processors," *32nd International Symposium on Computer Architecture (ISCA'05)*, pp. 197 – 207, June 2005.

[4] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovi, "The RISC-V Instruction Set Manual Volume I: User-Level ISAVersion 2.0." `http://riscv.org/download.html`, 2015. [Online; accessed 20-December-2015].

[5] "RISC-V software tools." `https://riscv.org/software-tools/`, 2016. [Online; accessed 08-June-2016].

[6] ANANTHA P. CHANDRAKASAN, ROBERT W. BRODERSEN, "Minimizing power consumption in digital cmos circuits," *Proceedings of the IEEE (Volume:83 , Issue: 4)*, pp. 498 – 523, Apr 1995.

[7] Zia Abbas, Mauro Olivieri, "Impact of technology scaling on leakage power in nano-scale bulk cmos digital standard cells," *Microelectronics Journal 45*, pp. 179 – 195, Feb 2014.

[8] HARRY J. M. VEENDRICK, "Short-circuit dissipation of static cmos circuitry and its impact on the design of buffer circuits," *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, vol. sc-19, pp. 468–473, Aug 1984.

[9] Flynn, D., Aitken, R., Gibbons, A., Shi, K, *Low Power Methodology Manual For System-on-Chip Design*. Springer, 2007.

[10] Mohammad Reza Kakoee, Ashoka Sathanur, Antonio Pullini, Jos Huisken, Luca Benini, "Automatic synthesis of near-threshold circuits with fine-grained performance tunability," *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pp. 401 – 406, Aug 2010.

[11] "A typical hardware setup using two shift registers to form an inter-chip circular buffer ." `https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus#/media/File:SPI_8-bit_circular_transfer.svg`, 2016. [Online; accessed 10-June-2016].

[12] "RISC-V tests repository." `https://github.com/riscv/riscv-tests`, 2016. [Online; accessed 31-May-2016].

[13] Nir Magen, Avinoam Kolodny, Uri Weiser, Nachum Shamir, "Interconnect-power dissipation in a microprocessor," *Proceeding SLIP '04 Proceedings of the 2004 international workshop on System level interconnect prediction*, pp. 7–13, 2004.