LUND UNIVERSITY

Department of Electrical and Information Technology

Master of Science Thesis

Low-power Microprocessor based on Stack Architecture

Author:
Girish Aramanekoppa Subbarao

Supervisors:
Prof. Joachim Rodrigues
Prof. Anders Ardö

Lund 2015

©

The Department of Electrical and Information Technology
Lund University
Box 118, S-221 00 LUND
SWEDEN

This thesis is set in Computer Modern 10pt,
with the LaTeX Documentation System

Printed in E-huset Lund, Sweden.
Sep. 2015

Abstract

There are many applications of microprocessors in embedded applications, where power efficiency becomes a critical requirement, e.g. wearable or mobile devices in healthcare, space instrumentation and handheld devices. One of the methods of achieving low power operation is by simplifying the device architecture.

RISC/CISC processors consume considerable power because of their complexity, which is due to their multiplexer system connecting the register file to the functional units and their instruction pipeline system. On the other hand, the Stack machines are comparatively less complex due to their implied addressing to the top two registers of the stack and smaller operation codes. This makes the instruction and the address decoder circuit simple by eliminating the multiplex switches for read and write ports of the register file. They are also optimized for procedure calls because they operate on stack instead of register, which reduces the memory size. All these factors make a stack machine power-efficient.

In this thesis project a Stack-based processor was designed in 65 nm CMOS technology. The area of the processor was 0.16 mm^2, which is very compact. The processor consumed about 20 uW/MHz when powered by a 0.6 V supply and 85 uW/MHz at 1.2 V. This is remarkably less than typical 250 to 450 uW/MHz consumed by the commercial grade low-power microcontrollers. This device was tested up to a speed of 50 MHz at 1.2 V and 20 MHz at 0.6 V.

Acknowledgement

I thank my thesis supervisors, Prof. Joachim Rodrigues and Prof. Anders Ardö for their guidance, support and encouragement during the course of this project. I would also like to thank Oskar Andersson for his help during the project.

I want to thank Hemant Prabhu for all the valuable discussions we had. I also wish to thank Prof.Liang Liu for his encouragement and inspiration. I also thank Rakesh Gangarajaiah, Ahmed Oudah and Jinan Shi for their help.

Finally, I want to specially thank my wife Jyoti, without whose support and encouragement, my Master course wouldn't have been possible.

Girish Aramanekoppa Subbarao
Lund, Aug, 2015

Contents

List of Tables

List of Figures

List of Acronyms

CISC	Complex Instruction Set Computer
CMOS	Complementary Metal-Oxide-Semiconductor
CTS	Clock Tree Synthesis
EDA	Electronic Design Automation
FF	Fast NMOS and Fast PMOS
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
IC	Integrated Circuit
IO	Input Output
ISR	Interrupt Service Routine
LEF	Layout Exchange Format
LIFO	Last In First Out
LPHVT	Low Power High Threshold Voltage
LPLVT	Low Power Low Threshold Voltage
LPSVT	Low Power Standard Threshold Voltage
NMOS	N-type Metal-Oxide-Semiconductor
NOP	No Operation

NOS	Next element Of Stack
PC	Program Counter
PMOS	P-type Metal-Oxide-Semiconductor
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SDC	Synopsys Design Constraints
SDF	Standard Delay Format
SPEF	Standard Parasitic Exchange Format
SS	Slow NMOS and Slow PMOS
TOR	Top element Of Return stack
TOS	Top element Of Stack
VCD	Value Change Dump
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WNS	Worst Case Negative Slack

Chapter 1

Introduction

My name is Sherlock Holmes. It is my business to know what other people do not know.

Arthur Conan Doyle

1.1 Overview

With the advancement of technology and the demand for advance features, Microprocessors are becoming increasingly complex and power hungry. There are many applications of microprocessors in embedded applications with limited resources, where power efficiency becomes a critical requirement, e.g. wearable or mobile devices in healthcare, space instrumentation and handheld devices. Hence, the need to develop an architecture, which focuses on power efficiency becomes important. One of the most important keys to reduce power consumption of any device is to reduce its complexity i.e. make things simple. The Stack machines with their simple architecture because of fewer numbers of transistors, are not only power efficient but also fast. Though this architecture may not be suited for every application, it can be targeted towards low power and real-time embedded applications.

The conventional RISC processors consume more power because of their complexity. The complexity comes from the fact that multiplexer system is used to connect the register file to the functional units. The complexity of such a system increases with the number of registers and functional unit [1]. Another factor, which complicates the hardware is the instruction pipeline system. The pipeline introduces the extra hardware and also brings in the problems called hazards. These hazards have

to be mitigated through techniques such as "forwarding" [2]. All these hardware results in the increased number of transistors, the wire length and the associated parasitic elements, which contribute to high power consumption [1].

On the other hand, the Stack machines are less complex compared to the RISC machines. Previous work on Stack machines highlight some of the reasons for their power efficiency. Implied addressing to the top two registers of the stack makes instruction and address decoder circuit simple and results in small operation codes [3]. Thus, less memory is needed to store the program, which needs less power. Multiplex switches are not required for read and write ports of the register file. Transferring data between the top of the stack register and the ALU is performed using the bus. So the energy is consumed only while data transfer occurs between stack registers and ALU functional units [1]. Also, the Stack machines are optimized for procedure calls because of very low penalty for the procedure calls [3]. Since, the Stack processors can be designed with fewer transistors, it results in not only a power efficient, but also a faster and reliable device.

1.2 Thesis Outline

The remaining chapters in this thesis are organized as follows.

In chapter two, the problem related to low-power processor architecture is introduced. The factors contributing to power consumption in RISC processor are analysed.

In chapter three, the proposed solution - the hardware architecture of the stack machine is discussed. Various modules of the processor, memory organization, program execution, instruction set architecture and programming are explained.

The IC layout of the stack processor, the design flow and the testing methodology are described in chapter four. In Chapter five, the power simulations performed on the IC are analysed. Chapter six concludes the report and discusses the future work.

Appendix A describes the instruction set architecture of the stack processor, appendix B tabulates the simulation results, while Appendix C gives a brief account of the history of stack machines.

Chapter 2

Problem statement

> Problems are not stop signs, they are guidelines.
>
> *Robert H. Schuller*

2.1 Issues with RISC processors

Microprocessors are becoming increasingly more complex resulting in consuming a considerable amount of power. Early microprocessors were based on the CISC architectures. These processors had complex hardware running moderately simple software. This is because, several instructions with high semantic content were capable of handling complex operations, thus simplifying the software that they execute[4]. Later, the RISC architecture was designed to address several issues with the CISC machines, mainly the throughput and performance [5]. The idea of the RISC was to simplify instruction set and thereby the hardware and transfer all the complexity to the software. Simplifying the instructions allowed them to be pipelined and hence increasing the throughput. With the advancement of RISC technology, these processors have ended up with complex hardware running complex software.

The complexity of any device considerably contributes to its power consumption. The complexity of RISC processors are mainly due to two factors - multiplexers and instruction pipeline [1]. The figure 2.1 shows the architecture of a simple RISC processor.

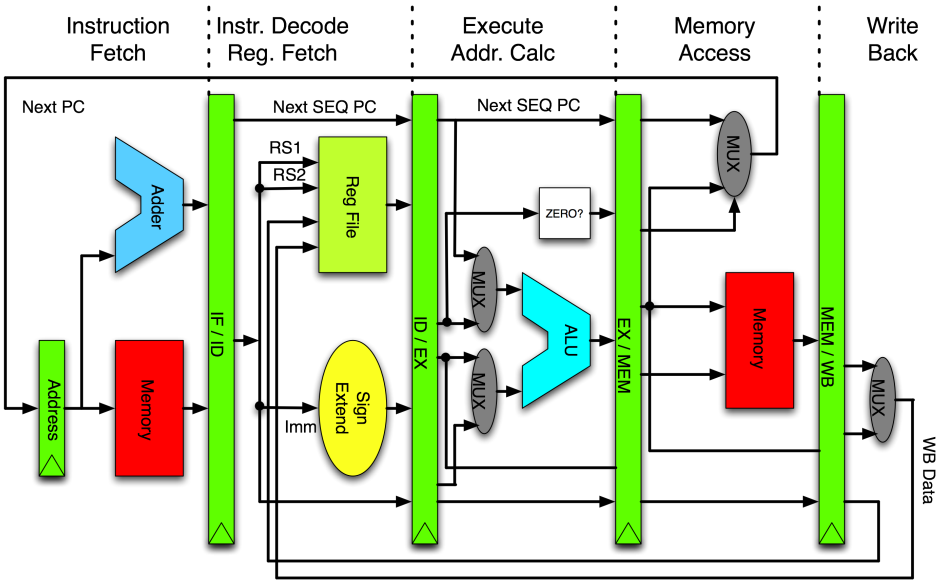

Image credit: *en.wikibooks.org*

Figure 2.1: RISC Architecture

2.1.1 Multiplexer

A register file is a set of registers used to store the data being operated upon by the processor. The functional units in the RISC processors get their operands from the register file [2]. The functional units read the operands from any of the registers in the register file, through a multiplexer circuit. Most of the functions need two operands and hence, two multiplexers are required at the input of the functional unit. Since, the output of the functional unit is stored back in one of the registers in the register file, another multiplexer is needed at the output. The size of the multiplexer increases exponentially with the number of registers [1]. Also, the multiplexers are large circuits. So, they contribute a large number of transistors, the interconnecting wires and the parasitic elements.

2.1.2 Instruction pipeline

One of the basic features of the RISC machines is the Instruction pipeline. This is accomplished by introducing registers in between different stages of execution. Besides, pipeline technique introduces conflicts called hazards. Some types of the hazards have to be overcome using the hardware called the forwarding circuit [2].

All these adds extra hardware to the processor circuit.

As can be seen, the hardware added by the multiplexer and the instruction pipeline complicates the processor circuit and hence contributes to the power consumption.

2.2 Proposed solution

There has been a great amount of research in improving the power efficiency of microprocessors leading to the development of several techniques. This project takes a slightly different approach to reduce the power consumption. One of the keys to reduce power consumption is to reduce the number of components in a device. Since, the power consumption is directly proportional to the component count, simplifying the circuit naturally reduces the power consumption. But, reducing its complexity would be a tedious task as the multiplexer and the instruction pipeline are integral parts of a RISC processor.

Another approach would be to design or find a simplified architecture. One such design is based on the Stack architecture. This architecture, not only eliminates the multiplexer at both sides of the functional units, it also removes the need for instruction pipeline. The architecture is discussed in detail chapter 3.

This project aims at verifying the assumption that the stack processor are power-efficient compared to the RISC processors and are suitable for low-power applications.

Chapter 3

Hardware Architecture

> Every solution to every problem is simple. It's the distance between the two where the mystery lies.
>
> *Derek Landy*

3.1 Stack Processor Overview

The Stack machines operate on a stack, rather than a register file as in the RISC processors. The operands to the functional unit is always fed from the top register or the top two registers of the stack. The result is always written back into the top register of the stack [3]. Hence, when an instruction is executed, the top element or the top two elements (depending on the type of instruction - unary or binary) is replaced by the result of the operation. This type of architecture, shown in the figure 3.1, has several advantages due to implied addressing, which are discussed below:

- Implied addressing to the top two registers of the stack simplifies the register address decoding as it eliminates the multiplexer at both sides of the functional unit. This also means that the signals have to pass through a shorter path and reduces the instruction execution time [1].

- Implied addressing also simplifies the instruction decoder circuit. Since, no address information is required, the instructions do not need the address fields. In contrast, the RISC processors should have register address fields to

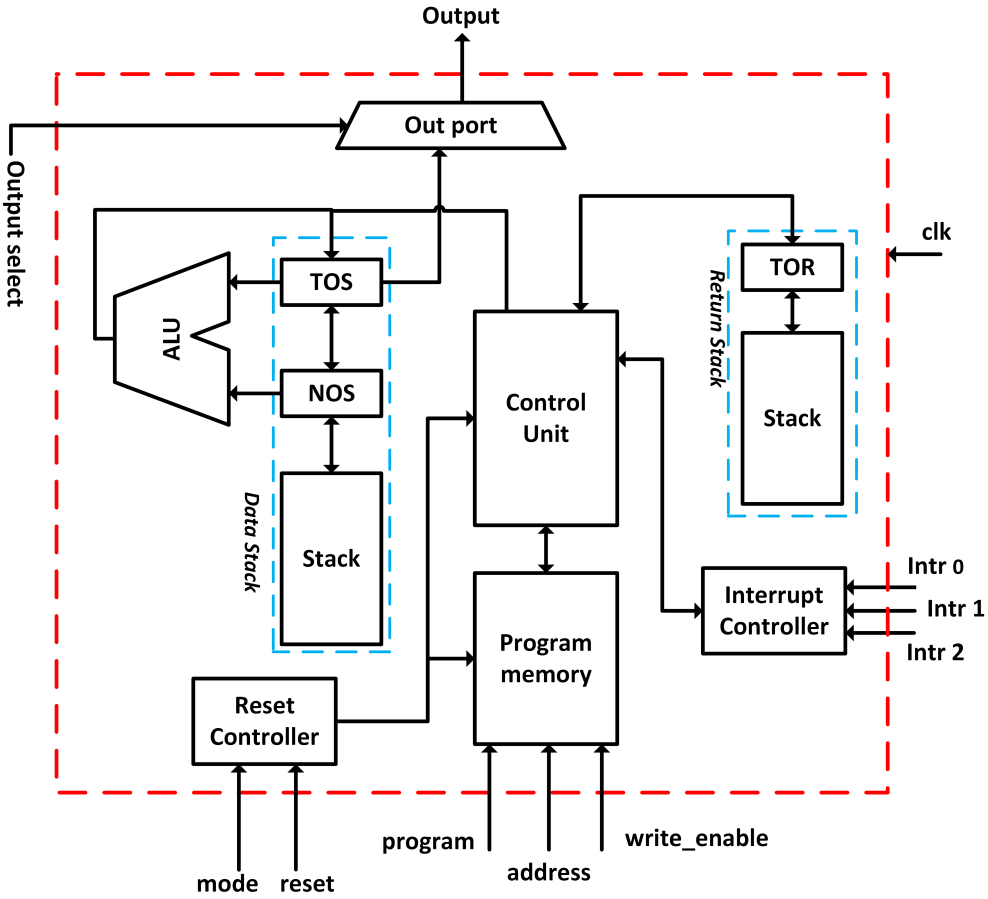

Figure 3.1: Architecture of Stack machine

read the operands and write back the result [3].

- Absence of address fields in the instructions shortens the instruction size. Short sized instructions need fewer bits for encoding, which results in small program size. Hence, the program memory size also reduces [3].

- There is no need to save the registers during the function call or servicing an interrupt, unlike in RISC processors. This is because the processor is already operating on a stack and hence, the calling function or an ISR has to just push the existing stack elements to use the registers. This means that the

function calls and the interrupts impose low clock cycle penalty, thus making the Stack machine more suitable for such operations. This encourages a modular program development [3].

- Low clock penalty for functions and ISRs also make Stack machine very deterministic and hence are more suitable for Real-Time applications [3].

3.2 Processor Specification

The specification of the stack processor designed in this project is summarised in table 3.1.

Table 3.1: Stack processor specifications

Instructions	38 8-bit instructions
Data stack	33 x 32-bit (Stores data)
Return stack	16 x 16-bit (Stores address)
Program memory	512 bytes (128 x 32-bit words)
Interrupts	3 external and 1 internal
Execution	3-stages (Fetch, Decode & Execute)

The instruction set consists of 38 instructions, which is summarized in table 3.2.

Table 3.2: Stack processor instruction set

Instruction type	**Description**
ALU	32-bit signed & unsigned
	16-bit signed & unsigned immediate data
	16-bit unsigned multiplication
	Logical & Shift
Branch	Conditional & unconditional jumps
	Function call
	Return from function & interrupt
Data stack	Data stack manipulation
Others	Software interrupt
	Interrupt enable & disable

3.3 Processor Architecture

The architecture of the stack processor shown in figure 3.1. The architecture contains the minimum set of modules that are required to function as a microprocessor. Various modules of the processor are explained in this section.

3.3.1 Control unit

The control unit is the heart of the processor. It decodes the instruction and issues signals to the appropriate modules in the processor.

Finite State Machine (FSM)

Its function is based on a 3-state finite state machine (FSM). The three states are `Interrupt`, `Decode` and `Execute` corresponding to three stages of the instruction execution cycle. Upon reset, the control unit enters the FSM. The FSM executes the interrupt state only if there is an interrupt signal from the interrupt controller. Otherwise, it continuously executes the other two states. The simplified form of the FSM is shown in figure 3.2. The figures 3.3, 3.4 and 3.5 show each state execution in detail. Each state takes once clock cycle to execute.

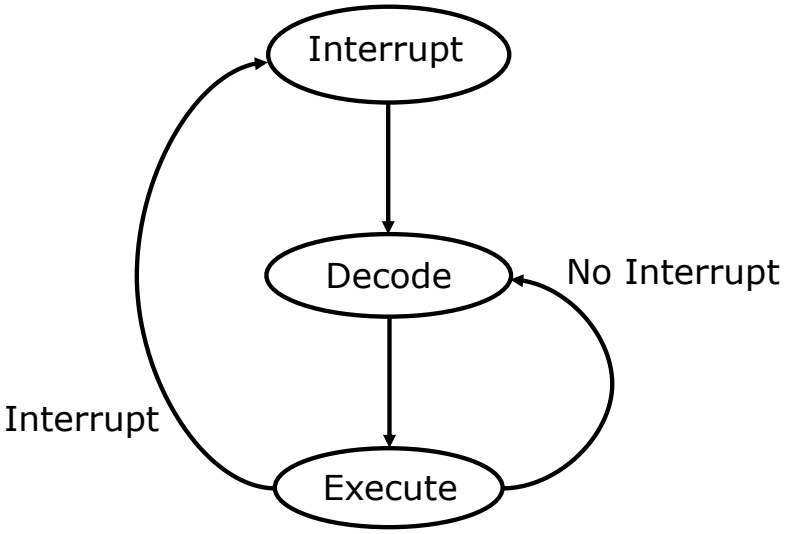

Figure 3.2: Finite state machine for the control unit

It takes four cycles to transfer the control to the ISR after the interrupt is acknowledged. The interrupt state in turn implements a small interrupt state machine

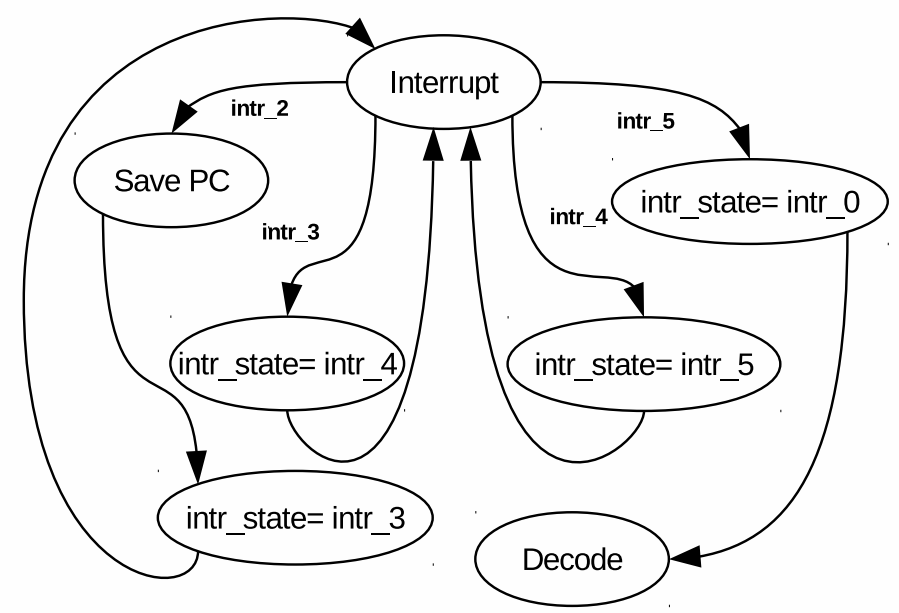

Figure 3.3: Interrupt state of the FSM

consisting of four states `intr_2` to `intr_5`. The state `intr_0` indicates that the processor has not received any interrupt and `intr_1` indicates that the processor has received an interrupt. The processor transits to state `intr_2` once the interrupt is acknowledged and the interrupt cycle begins. All the states of the interrupt cycle are summarised in table 3.3.

Table 3.3: Various states of Interrupt State Machine

Interrupt state	**Function**
intr_0	No interrupt
intr_1	Interrupt received
intr_2	Push PC into the Return stack & change PC to interrupt vector
intr_3	Wait for PC save in the Return stack and PC change
intr_4	Wait for Program to output new instruction word
intr_5	Initiate new instruction read & reset interrupt state to intr_0

In the decode state, the signals are issued to different modules of the processor based on the instruction. In the execute state, the modules that receive signals

produces the results.

If the instruction is a branching type such as jump, jump on a flag condition or a function call, a small branching state machine is executed. It consists of four states with decode and execute states executing two states each. It means that the decode and execute states are executed two times for a branching instruction. The states involving the branching process are shown in table 3.4.

Table 3.4: Various states of Branching instruction

Branching state	**Function**
branch_decode	Change PC to branch address.
branch_execute	If there is no interrupt, wait for PC change and start instruction fetch. If interrupt not asserted, don't fetch new instruction.
address_decode	Wait for the new instruction fetch cycle to complete.
address_execute	Read newly fetched instruction from the branching address.

Similarly for a return from function or an ISR instruction execution, another small return state machine is executed, consisting of six states. The decode and execute states execute three states each, which implies that they are executed three times for a return instruction execution. These states are described in table 3.5.

Table 3.5: Various states of Return instruction

Return state	**Function**
return_decode	Pop the return address from the Return stack.
pop_address	Wait for the Return stack to pop out the address.
change_pc	Wait for the PC to change to the return address.
fetch_instruction	If no interrupt asserted, start instruction fetch. If interrupt asserted, don't fetch new instruction.
wait_fetch	Wait for new instruction fetch to complete.
read_instruction	Read the newly fetched instruction.

The figures 3.4 and 3.5 shows various states of the branching instructions. Both the figures should be studied together to understand the state transitions as they are spread between the decode and execute states.

Instruction execution

An instruction-fetch process in the control unit fetches the instruction word from the memory. This process keeps track of the number of instructions remaining in

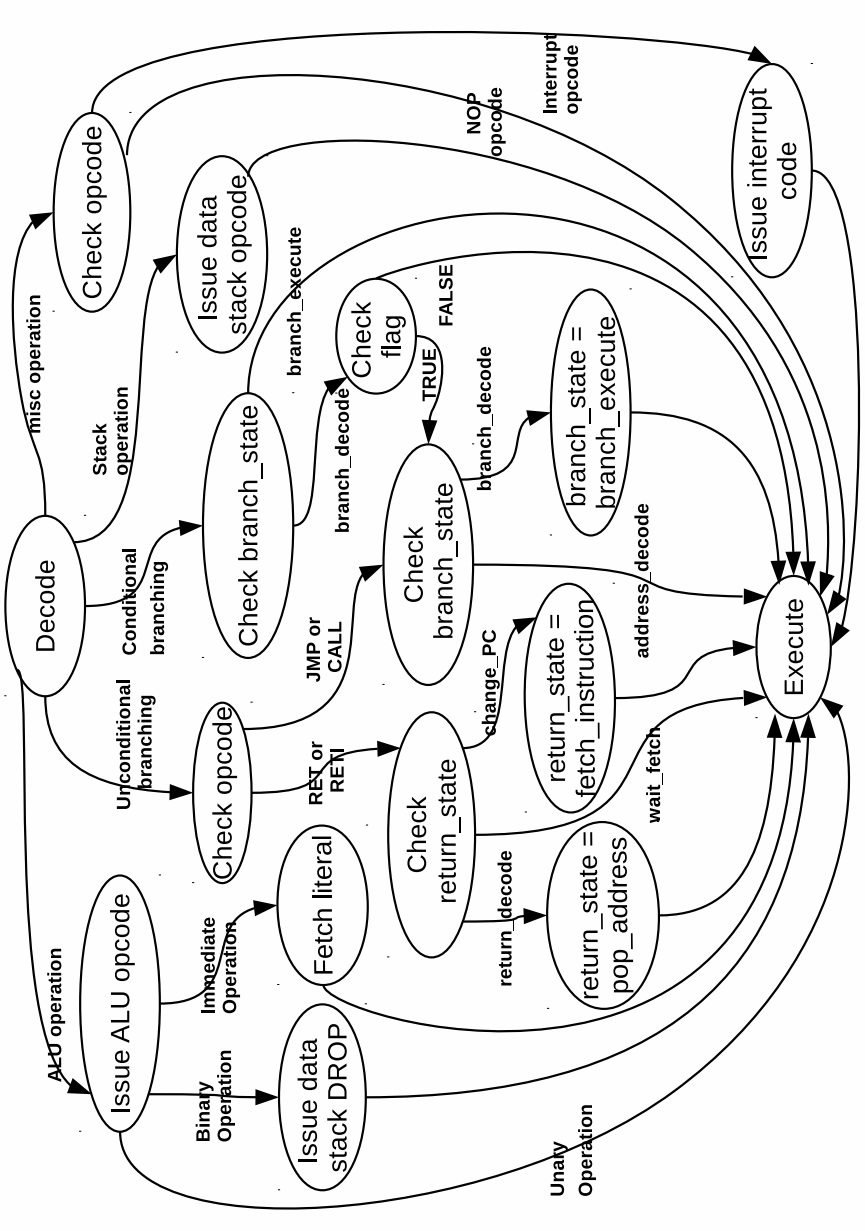

Figure 3.4: Decode state of the FSM

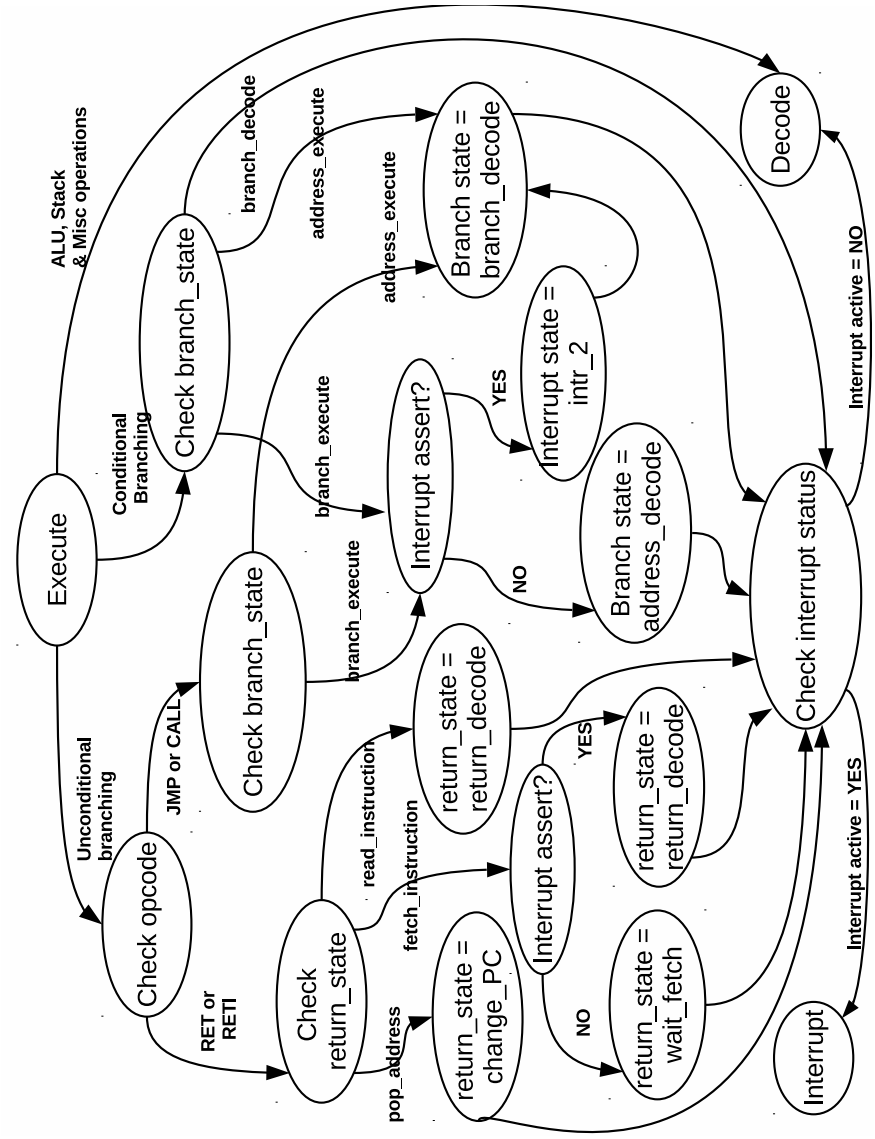

Figure 3.5: Execute state of the FSM

the instruction word that are to be executed. It initiates instruction fetch from the memory in advance in such a way that the next instruction fetch cycle overlaps with the execution of the last instruction of the current instruction word. Though the instruction size is 8-bit, some instructions which operate on immediate data and

jump instructions have 16-bit literal making them 24-bit or 3-byte instructions. As a result, each fetch can load two or four instructions from the memory depending upon the type of instructions. Figure 3.6 shows the examples of such instruction groups.

Figure 3.6: Examples of different instruction groups

The fetch-decode-execute cycle is shown in figure 3.7. The execution cycle appears as a two-stage non-pipelined process, though internally it operates as a semi-pipelined system. After the instruction word is fetched, one instruction is decoded and executed at a time. Instruction decode and the instruction execution take one clock cycle each. Hence, each instruction takes two clock cycles. During decode, the instruction bit map is analysed to determine the function that is represented and the signals are sent to the appropriate modules. In the execute cycle, the modules that are signalled by the control unit operates to produce the result.

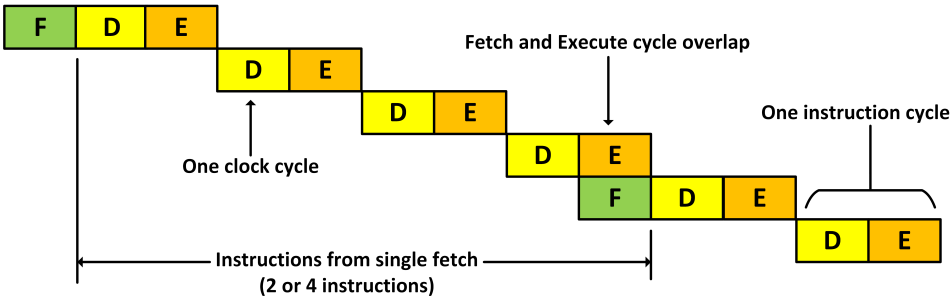

Figure 3.7: Instruction execution cycle of stack processor

The control unit interfacing with the memory and the return stack along with the associated signals are shown in figure 3.8. The waveforms of various signals during the program execution is shown in figure 3.9.

Braching execution

When the function-call instruction is executed, the current program counter (PC) is pushed into the return stack and the PC is changed to the address of the calling function. The return from the function is accomplished by retrieving the address

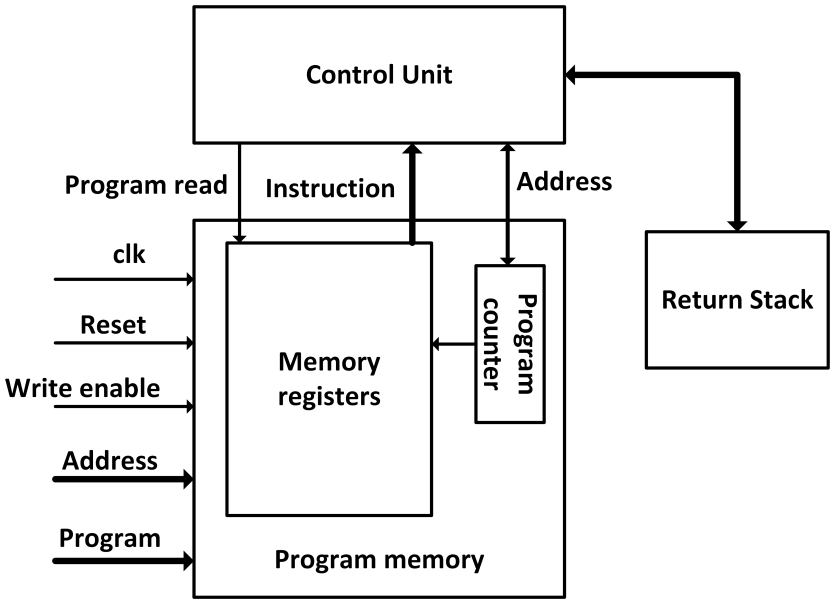

Figure 3.8: Control unit interface with the memory and return stack

from the return stack and loading it into the PC, so that the execution continues from the next address, where the function was last called.

The branching process, both conditional and unconditional are similar to the function call, except that the return to the calling point is not involved. The branching process takes four clock cycles, while the return from the function or ISR takes six clock cycles.

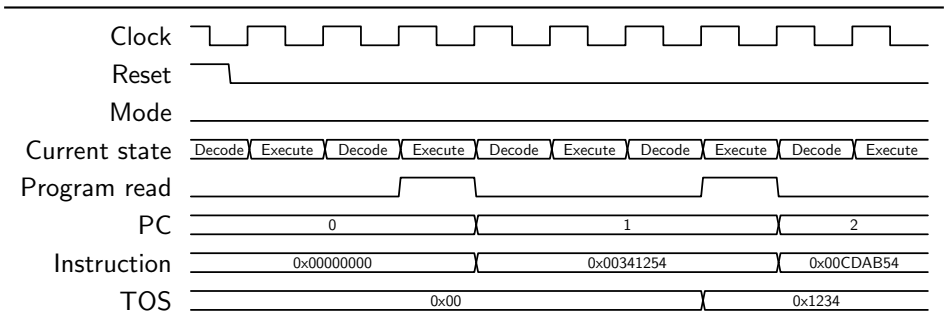

Figure 3.9: Waveforms of microprocessor execution

Interrupt execution

If an interrupt is signalled by the interrupt controller, then the control is transferred to an `interrupt` state of the FSM. This state saves the PC in the return stack and changes the PC to the interrupt vector. Then, the state changes to `decode` state and execution continues normally. At the end of the ISR, the return instruction retrieves the saved address of the program from the return stack, where the ISR started executing, and continuous with the main program execution.

3.3.2 Data stack

The data stack is used to store the operands of the program. This is the counterpart of registers in a RISC machine. It is an array of registers, with the top two elements `Top-element-Of-Stack (TOS)` and `Next-element-Of-Stack (NOS)` implemented as separate registers. This design will synthesize only one multiplexer, instead of two. Also, the TOS being the destination register will eliminate another multiplexer at the output of the ALU.

If the data stack overflows or underflows, a reset signal is generated, which resets the processor.

3.3.3 Return stack

The return stack is used to store the return address, when the control jumps to a different address due to a function call or execute an ISR. This is also an array of registers, with the `top element Top-element-of-Return-stack (TOR)` implemented as a separate register. Though not implemented in this design, TOR can also be used as temporary storage or as a loop counter. Hence, this stack cannot be directly accessed through any instruction, but accessed only indirectly.

If the return stack overflows or underflows, a reset signal is generated, which resets the processor.

3.3.4 Arithmetic and Logic unit

The ALU takes (pops out) the operands from the top of the data stack and performs the operation as signalled by the control unit. It pushes back the result into the data stack. Hence, the top one or two elements (depending upon the instruction type) of the data stack are replaced by the result after every ALU operation. The operations performed by the ALU are summarised in table 3.6.

Table 3.6: ALU operations

32-bit signed/unsigned addition
32-bit signed/unsigned subtraction
16-bit signed/unsigned immediate data addition
16-bit signed/unsigned immediate data subtraction
16-bit unsigned multiplication
Unsigned increment/decrement
Logical NOT, AND, OR, XOR
Logical and Arithmetic right/left shift

3.3.5 Program memory

Larger memory width decreases the memory bandwidth requirement as each instruction fetch loads greater number of instructions from the memory, but increases the memory bus width and vice versa. Also, the larger memory width enables handling of larger numbers. So, a trade-off between small and large memory width has to be made to optimize memory organization. In this processor, where the instruction size is 8-bit, the memory size of 32-bit was chosen. The program memory was implemented as an array of registers because of its small size. If the size of the memory is large, then memory module has to be used. The memory map is shown in table 3.7.

Table 3.7: Memory map

Address	**Description**
0x00	Reset vector
0x70	Interrupt 0 vector
0x74	Interrupt 1 vector
0x78	Interrupt 2 vector
0x7C	Software Interrupt vector

3.3.6 Reset controller

The reset controller issues two types of reset signals to the modules within the processor - `program reset` and `device reset`. Figure 3.10 shows the reset system. The type of reset signal that is released depends on the logic level of the input pin `mode` when the processor is reset. If the processor is reset with `mode` pin at logic high, the `program reset` is released. If the processor is reset with `mode` pin at logic low, the `device reset` is released.

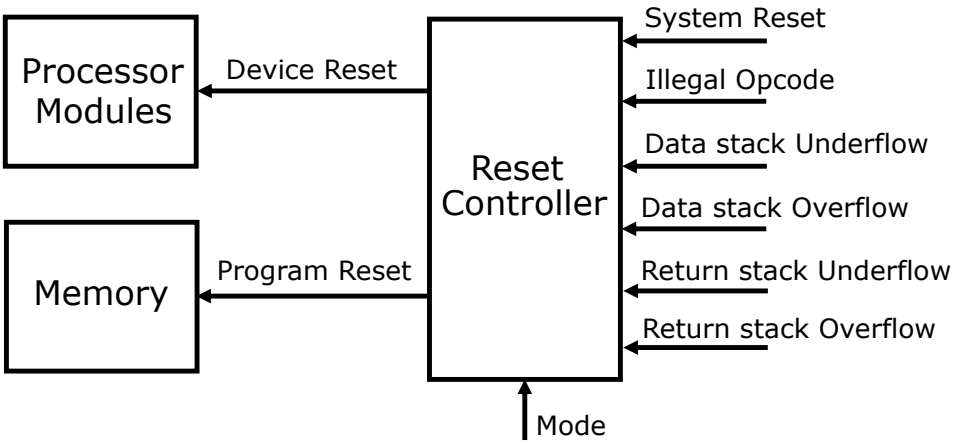

Figure 3.10: Reset controller

When the `program reset` is released, the address, program and memory-write-enable pins are enabled. These pins are used to program the memory from the external device. In this condition, the rest of the modules within the processor are in the reset state. Figure 3.11 shows the waveforms of programming the microprocessor.

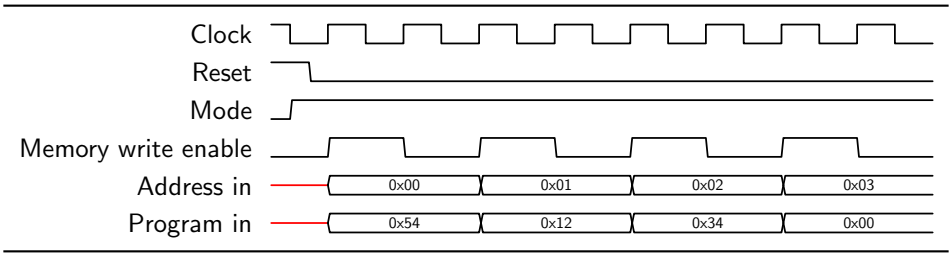

Figure 3.11: Waveforms of programming the microprocessor

When the `device reset` is released, all the modules within the processor come out of the reset state and starts functioning. In this condition, the address, program and memory-write-enable pins are disabled and the processor cannot be programmed.

The processor can be reset due to reasons other than toggling the reset pin. The underflow or overflow of the data or return stack and illegal instruction can also cause the processor reset. Figure 3.10 shows various sources of reset through the reset controller.

3.3.7 Interrupt controller

The interrupt controller interfaces the three external interrupts and one internal (software) interrupt to the processor. The external interrupt is sensed when the signal on any of the interrupt pins changes logic level from low to high. The internal interrupt is sensed when the instruction `SWI` is executed. The interrupt service can be enabled or disabled by executing the instructions `EI` and `DI` respectively. The interrupts are disabled when the processor is reset and hence, have to enabled every time the processor is reset, if they have to be serviced.

When the interrupt is enabled and any of the interrupts asserts, the interrupt controller will signal the control unit by sending the interrupt number. The control unit will then execute the interrupt as explained in section 3.3.1. Once the control enters the ISR, the invoked interrupt has to be disabled; otherwise, the control will enter into an infinite loop of entering into the ISR. In normal processors, this is done by resetting the interrupt source bit in the interrupt control register. Since this processor doesn't have such registers, the interrupt disabling is done in hardware by sending an acknowledgement from the control unit to the interrupt controller. On receiving the interrupt acknowledgement, the interrupt controller disables the invoked interrupt. The interface of the interrupt controller and various associated signals are shown in figure 3.12.

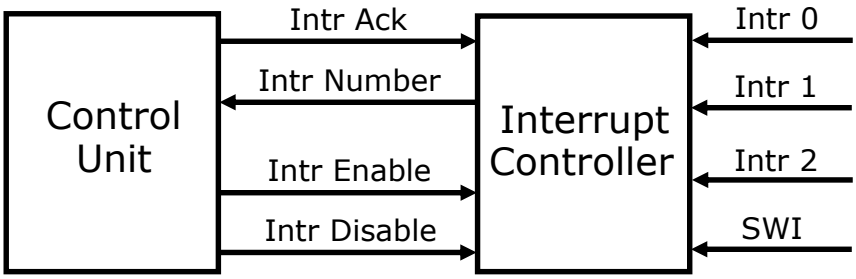

Figure 3.12: Interrupt controller interface

Upon receiving an interrupt, the control unit changes PC to appropriate interrupt vector as shown in table 3.7. The required ISR should be programmed at respective the interrupt vectors. The size of the ISR for each interrupt is limited to 16 bytes in this implementation.

3.3.8 Output port

This project aims at determining the power efficiency of the processor core. The input/output (IO) port is not a part of the core and also there is no standard way of implementation. Hence, adding the IO port does not contribute towards the project goal.

It is not possible to observe the result of any program execution because of the absence of IO ports. Hence, a set of output pins is connected to the TOS register. The result to be observed is stored in TOS, which appears on the pins. Since the size of TOS is 32-bit, connecting the pins to all the bits of TOS is not economical. Hence, only eight pins are connected to this register through a multiplexer. This arrangement needs only ten pins in total (Eight pins for output and two pins for multiplexer). Switching of multiplexer pins allows observation of all the four bytes in TOS. If the result has more than one word, then they can be pushed into the data stack. An ISR can be programmed to pop out the data stack on every trigger of an external interrupt. This way, all the values can be observed at the output.

It should be noted that this output module is only used in place of an IO port and should be removed when the actual ports are implemented.

3.4 Program

Since the operands on a stack are available in a Last-In-First-Out (LIFO) sequence, the program uses a Reverse-Polish or a Post-fix notation. In this technique, the operands precede the operators [3]. Every operation is done on the top two elements of the stack, e.g. (12 + 34) * 56 is represented as `56 34 12 + *`. Figure 3.13 shows the program to evaluate this expression. The literals 56, 34 and 12 are pushed into the data stack first. Then, the operation + is applied to the top two elements of the stack, which are 56 and 34. These two literals are popped out of the stack and the result, 46 is pushed in. Then the operation * is applied to the top two stack elements, which are 46 and 12. Again, both of them are popped out and the result, 2576 is pushed into the stack. Figure 3.14 shows the snapshots of the data stack when these instructions are executed.

Apart from the normal instructions, the stack processor has to provide some extra instructions to perform stack manipulations. Such operations, which are implemented in this processor are `PUSH`, `SWAP`, `DROP`, `OVER` and `DUP` [3]. The instruction PUSH pushes a literal into the stack, SWAP exchanges TOS and NOS, DROP pops out TOS, OVER pushes NOS over TOS and DUP duplicates TOS, i.e. pushes TOS on top of TOS. These operations are demonstrated in figure 3.15.

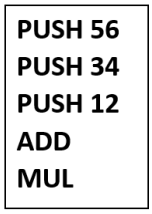

Figure 3.13: Example of a short program

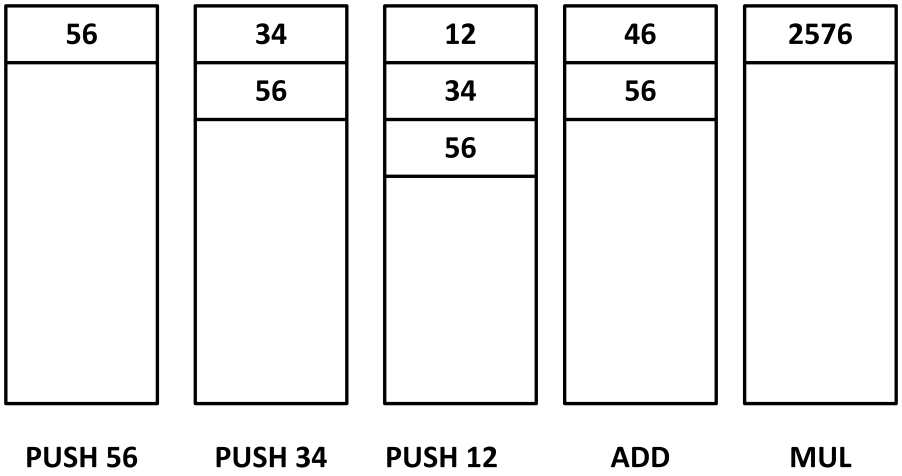

Figure 3.14: Snapshot of the data stack

The figure shows the instructions executed and the state of the stack after the execution of each instruction. The instructions PUSH 0x78, PUSH 0x56, PUSH 0x34 and PUSH 0x12 pushes the literals 0x78, 0x56, 0x34 and 0x12 into the stack in a LIFO sequence. The instruction SWAP swaps the top two literals 0x12 and 0x34, DROP pops out the top literal 0x34, OVER pushes the NOS literal 0x56 into the stack and DUP pushes the TOS literal 0x56 again into the stack.

3.5 Design improvement opportunities

In the course of designing this processor, several compromises were made, in order to simplify the architecture and complete the design in a limited time period. Though the designed stack processor works according to the specification, there are several opportunities for improvements. The compromises and the opportunities for improvement of the processor are discussed in this section.

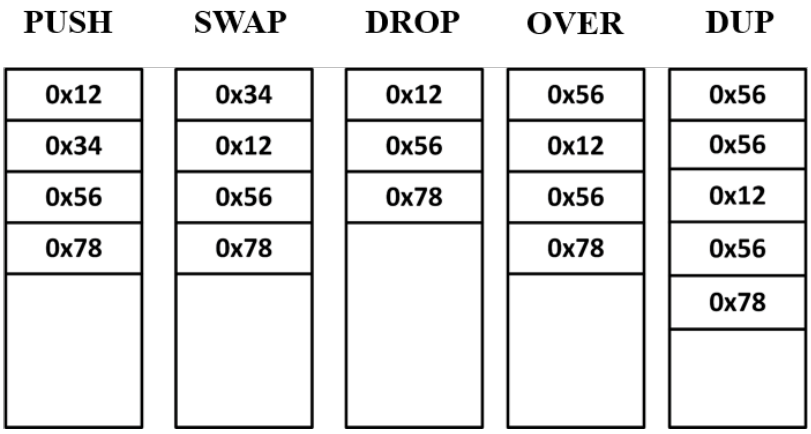

Figure 3.15: Examples of stack manipulation operations

3.5.1 ALU operations

Size of the literal

Immediate data operations perform addition or subtraction of a literal with TOS. The literal is a 16-bit number, while TOS stores a 32-bit number. If the literal is made 32-bit, then it has to be stored in the next instruction word because of its size. This will need an extra fetch for the literal. If the immediate data instruction is at the beginning of the instruction word or if there are more than one such instruction, it not only wastes the memory, but also complicates decoding scheme. This will also make the hardware complex. Hence, the literal size was limited to 16-bit. Table 3.8 demonstrates this fact.

Multiplication

The multiplication is performed on 16-bit unsigned numbers. This allows 32-bit result to be conveniently placed in a single register. A 32-bit multiplication will result in a 64-bit result, which needs two 32-bit registers to save. This argument also holds good for the `PUSHI` instruction, which pushes a literal into the data stack.

3.5.2 Position of instructions

Position of immediate data instructions

Though the immediate instructions are 8-bit wide, they are always combined with the literal of 16-bit wide, making them 24-bit instructions. A restriction is imposed on their position in the instruction word to simplify fetching them together. These

instructions can appear only on the first or the second byte position as shown in table 3.8. In such conditions, the remaining one byte in that instruction word has to be filled with `NOP` or any other instruction which doesn't affect the program logic.

Table 3.8: Immediate data instruction positions

Byte 3	Byte 2	Byte 1	Byte 0
Opcode	Literal		Immediate data Opcode
Literal		Immediate data Opcode	Opcode

If these instructions appear at the third or fourth position, then the literal spills to next instruction word and makes fetching process complex.

Position of branching instructions

The branching instructions `JMP`, `JZ`, `JNZ`, `JC`, `JNC`, `JNEG`, `RET` and `RETI` also have the restriction of being placed in the first or the second position of the instruction word. The reason is the same as that for the immediate data operations.

It should be noted that the issues explained here and in section 3.5.1 to are due to the type of memory organization chosen for the processor and not due to the stack architecture.

3.5.3 Special purpose registers

Microprocessors have special purpose registers to control the peripherals and read their status. The current implementation of the stack processor does not have these registers. These registers also need instructions to read and write values into them.

3.5.4 Interrupts

Interrupt control and status registers

Interrupt control and status registers are also part of special purpose registers and hence not present in this processor. A minimal support for control of interrupt controller is provided in the form of the global interrupt enable and disable instructions `IE` and `ID`. Whenever there is an interrupt, on entering the corresponding ISR, that particular interrupt has to be disabled in order to prevent infinite interrupt re-entrance. The automatic interrupt disable on entering the ISR is done automatically in hardware by the interrupt acknowledge signal sent from the control unit to the interrupt controller. Figure 3.12 demonstrates this process.

Interrupt priority and nesting

Interrupt priority is not implemented in this processor. Priority is resolved only if more than one interrupt occurs during the same clock cycle. In such a case, `Interrupt 0` has the highest priority and `Software interrupt` has the lowest priority.

Interrupt nesting is also not implemented. This requires interrupt priority scheme and also the flag register to be saved (by pushing into the stack), whenever an ISR is interrupted by a higher priority interrupt. At present, if an interrupt occurs during the execution of an ISR, it has to wait until the current interrupt service is complete.

Interrupt latency

At present, the performance of this processor is not very good as far as the interrupt latency is concerned. The interrupt latency has to be small and also deterministic. In this processor, the interrupt is serviced after all the instructions of the current instruction word are executed, which is a maximum of four instructions. Hence, the number of clock cycles elapsed before the ISR starts executing varies depending upon the instance at which interrupt arrives with respect to the instruction number in the current instruction word that is being executed. The latency varies from 9 to 15 clock cycles.

Latency of return instruction

The return from the function or ISR takes six clock cycles as the return address has to be retrieved and then change the PC.

3.5.5 Functions

Address of function

Since each instruction word has up to four instructions, the PC increment by one effectively changes the byte address by 4. Hence, the starting address of the function has to be a multiple of 4. The gaps in the program due to this limitation have to be filled by the `NOP` instructions.

3.6 Limitations

Though the stack architecture appeals due to its simplicity, it has several limitations.

- There is a limit to the speed performance as any two consequential instructions always have dependence on the TOS register [1].

- It has a limited ability for parallel execution because of the sequential nature of the stack [1].

- It is required to keep track of the stack as it might spill. This can be mitigated by spilling the operands into the RAM [3].

- Iterative process is poorly performed as the index might be buried inside the stack [6]. This can be overcome by providing separate index registers.

- Stack manipulation overhead. Extra operations are required to rearrange the values in the stack [7].

- Compact instruction size and large operand pose challenge for efficient memory organization.

- The instruction pipelines, if implemented, might lead to hazards.

Chapter 4

IC Layout Design, Implementation and Simulation

All code is guilty, until proven innocent.

Anonymous

4.1 Tools and Design flow

The IC layout is the product of several EDA tools, which executes scripts describing user-defined constraints. The process involves Behavioural simulation, RTL synthesis, Place and route and Power simulation. Each of these stages are carried out by different tools. Figure 4.1 shows the various steps in the IC design flow [8].

4.1.1 Behavioural simulation

The RTL coding for the Stack processor was done using VHDL. The behaviour of the design was simulated using the HDL simulation tool **Modelsim**. This tool compiles and executes the VHDL program and displays the waveforms. These are ideal waveforms, which are used only for the functional analysis of the design. Various modules, which made up the processor and the whole processor design were validated using several test benches, also written in VHDL. The test bench provides the stimulus to the processor and reads the output of the processor.

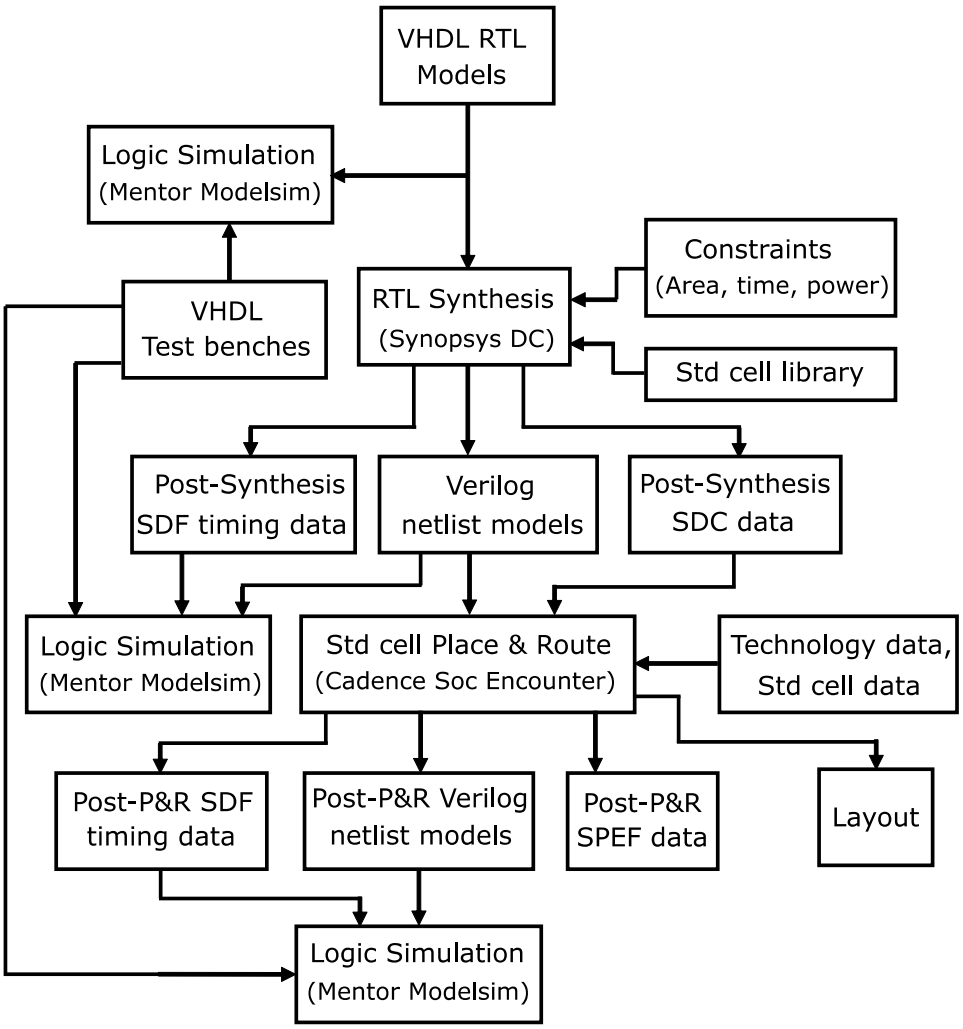

Figure 4.1: Design flow

4.1.2 RTL Synthesis

The RTL code was synthesized using the synthesis tool **Design compiler**. RTL synthesis creates a gate-level circuit based on the RTL model, which meets the design constraints such as area, timings and power consumption. The constraints are provided to the tool through a script file, which contains commands specific to the tool. The tool synthesises the circuit using the standard cells, which are by themselves transistor circuits providing a defined functionality such as logical

operations, multiplexer, adder, flip-flop, buffer, etc. The standard cells are provided in the form of libraries by the foundry. The synthesis outputs a Verilog gate-level netlist, Standard Delay Format (SDF) description and Synopsys Design Constraints (SDC) files. The various steps involved in synthesis are [8]:

- Analyze - Compile the VHDL models and check whether the VHDL codes are synthesizable.
- Elaborate - Perform generic pre-synthesis of the analysed models. Inferred registers (flip-flops and latches) are identified.
- Define the design environment such as operating conditions, wire load models and system interface characteristics.
- Define design constraint such as clock, area and timing.
- Design mapping and optimization (Compilation) - Logic gates from the standard cell library are assigned to the generic gates in the elaborated design such that the defined constraints are met.
- Generate Verilog netlist, SDF and SDC files for post-synthesis simulations.
- Generate various reports such as violated constraints, area, timing, critical path and resource usage.

Timing constraint is one of the most important constraints that has to be met during synthesis. For the circuit to operate correctly, the data at every node should arrive before it is required at that node. The difference between the required time and the arrival time of the data is known as the `Slack`. The slack defines the timing margin. A timing violation is indicated by a negative slack value. The timing report, shown in table 4.1, generated for all the three designs of the stack processor shows that this constraint has been met.

Table 4.1: Timing report of synthesis

	Transistor type		
Timing parameter	**LPLVT**	**LPSVT**	**LPHVT**
Clock period (ns)	20.00	20.00	40.00
Data required time (ns)	19.54	19.52	39.12
Data arrival time (ns)	-3.81	-4.62	-5.32
Slack (ns)	15.72	14.90	33.80

The high positive value of slack indicates that this design could be synthesised for higher clock speed. However, this was not done for three reasons:

- Low-power processors are rarely used above 50 MHz.
- Lowering the supply voltage decreases the slack and hence decreases the frequency of operation.
- This design was synthesised with fewer timing constraints. Hence, with actual constraints the slack is expected to decrease.

The synthesis report indicated that the critical path is provided by the multiplier. This is expected as it is the most complicated module between two registers.

Post-synthesis gate-level simulations were performed in **Modelsim** using the Verilog netlist with the same set of test benches that were used for behavioural simulation. The standard cells were back-annotated with timing delays from the SDF file. The behaviour of the post-synthesis model was found to confirm with that of behavioural model.

4.1.3 Place and route

The tool **Soc Encounter** was used to perform the place and route of the Stack processor integrated circuit. This tool takes several files as input:

- The Verilog gate-level netlist generated by the synthesis tool.
- Layout Exchange Format (LEF) files containing the information on the technological process such as metal and via layers, via generate rules and the cell library.
- Library files containing information on cell timings such as delays, setup and hold times.
- Design constraint file (SDC) generated by the synthesis tool, containing the information about timing, area and power constraints.
- Pad configuration file defining the pads in the design.
- Power net names that defined in the LEF technology file.

The place and route process involves the following steps:

- Create a configuration file with all the input files listed earlier in this section.
- Floorplan the design, which defines the size of the layout.
- Place and route the memories (if used).
- Power planning, which involves adding the power rings and the power stripes.

- Place the standard cells.
- Clock tree synthesis (CTS).
- Place the IO filler cells.
- Routing the power nets.
- Routing the design.
- Timing analysis.
- Fixing timing violations through pre-CTS, post-CTS and post-route optimizations.
- Post-route Verilog netlist generation for post-layout simulations.
- Extract post-route timing data in the form SDF and SPEF files for post-layout simulations.

The figure 4.2 shows the IC layout of the stack processor that was produced by the place and route process. The features of the IC are shown in table 4.2.

The post-layout simulations were performed using **Modelsim** with the same set of test benches that were used for behavioural and post-synthesis simulation. The standard cells were back-annotated with timing information from the SDF and SPEF files. The behaviour of the post-layout model was found to confirm with that of behavioural and post-synthesis model.

Table 4.2: Features of Stack processor IC

Technology	65 nm CMOS
Dimension	400 um x 400 um
Area	0.16 mm^2
Core utilization	0.58
WNS	23.4 ns
Critical path	Multiplier

4.2 Implementation

In order to assess the power performance of the processor, separate layouts for the processor were designed with three types of low-power transistors having different threshold voltages. Transistors with lower threshold voltage switch faster than those

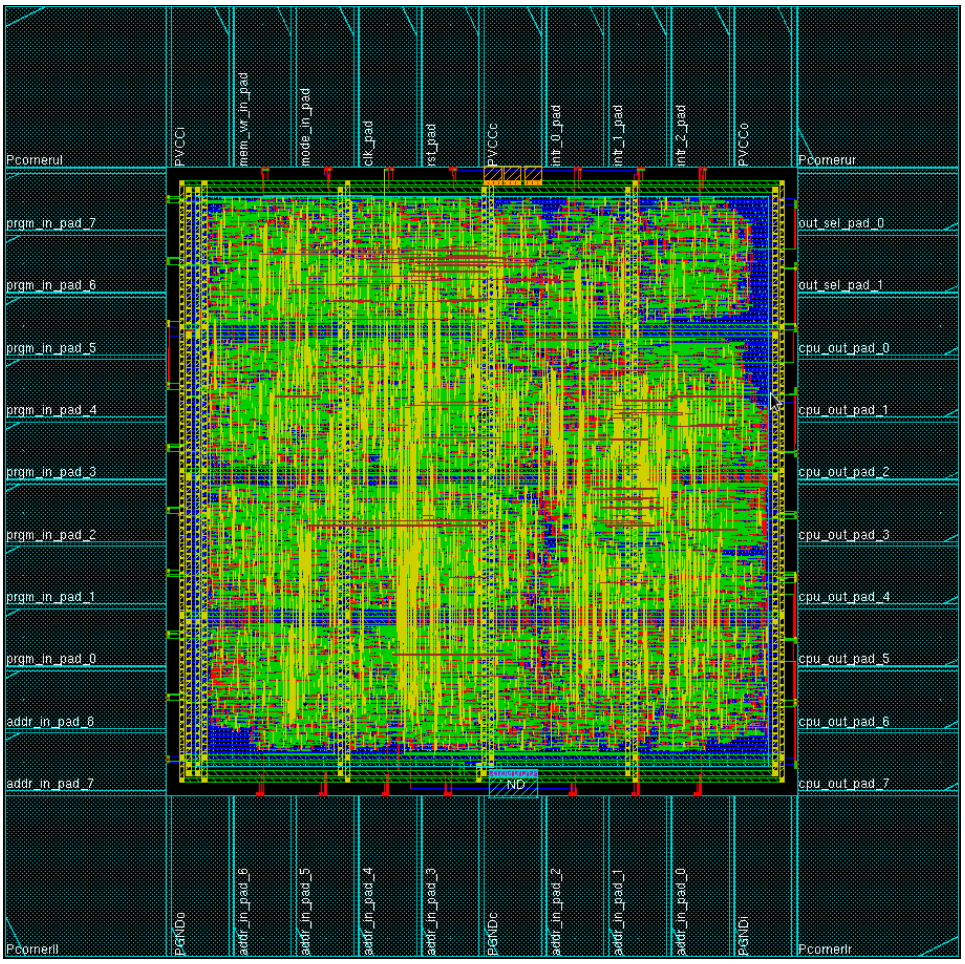

Figure 4.2: IC Layout

with higher threshold voltage [9]. But they have higher leakage current compared to the higher threshold voltage transistors. Transistors with higher threshold voltage exhibit opposite characteristics. The simulations on these different layouts provide information, which helps make a trade-off between the microprocessor speed and power efficiency, and design the processor with the required characteristics.

The processor IC was implemented using the STMicroelectronics 65 nm standard cell libraries - CORE65LPLVT, CORE65LPSVT and CORE65LPHVT. The CORE65LPLVT provides the standard cells with transistors having low threshold

voltage (about 0.3 V), while CORE65LPHVT provides the standard cells with transistors having high threshold voltage (about 0.6 V) and CORE65LPSVT is between the two (about 0.5 V).

These libraries are available for various supply voltages ranging from 0.4 V to 1.35 V. Three voltages 0.6 V, 0.8 V and 1.2 V were chosen for testing. The IC layout was done for 1.2 V supply voltage with all three types of standard cells. Then the IC design was reopened with the standard cells of lower voltage for testing. This method was chosen as it was quicker to design for 1.2 V compared to lower supply voltages. Only the layout with LPHVT was done with pads, while the other two layouts were without pads. This was done because of the non-availability of suitable pads for LPSVT and LPLVT designs.

4.3 Gate count

The number of gates used in each layout was determined through **Soc Encounter**. They are tabulated in table 4.3. Though all the layouts were based on the same design, the number of gates varies because of the differences in the extent of optimizations performed during place and routing. Each gate is equivalent to 2-input NAND gate, which is typically made up of four transistors. Therefore, the transistor count is four times the value shown in the table. This amounts to approximately 200,000 transistors.

Table 4.3: Gate count

Module	**Gate count**		
	LPHVT	**LPSVT**	**LPLVT**
ALU	5278	4530	4642
Control Unit	2113	1692	2147
Data Stack	8275	8151	8412
Return Stack	2195	2149	2243
Program memory	27936	30596	30744
Reset controller	27	24	26
Interrupt controller	86	84	117
Total	45910	47226	48330

4.4 Power simulations

The power simulations on the post-layout design was carried out using the tool **Primetime**. The most effective way of testing a microprocessor is by executing

several types of programs which represent the actual scenario. Since, the time to develop such programs was not available, a program which tested all the instructions, including the interrupt service, was developed. The same program was used to perform testing on all the IC layouts designed with different transistors types.

The program was written in the assembly language of the processor using the postfix notation since a high-level language compiler was not available for this processor at the time of testing.

The sequence of operations involving power simulation is shown in figure 4.3. After the successful completion of the place and route of the processor design using **Soc Encounter**, the Verilog netlist, the Standard Delay Format (SDF) and the Standard Parasitic Exchange Format (SPEF) files were extracted from the tool. The SDF file stores the timing data generated by the tool. The data contains information about various delay and timing parameters. The SPEF file stores the information about chip parasitic elements.

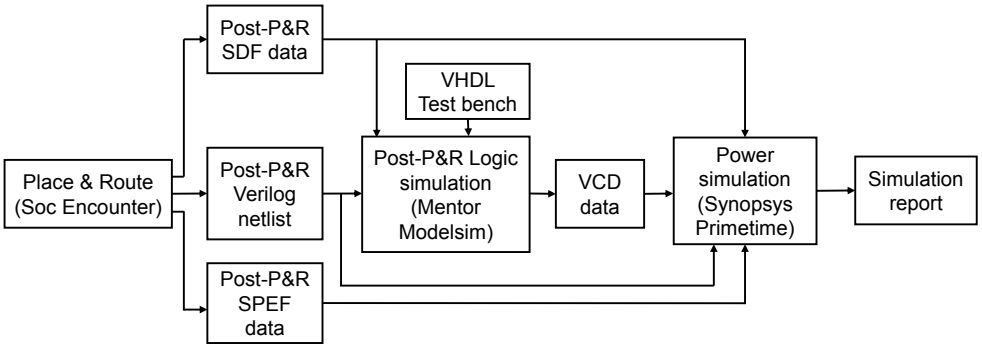

Figure 4.3: Power simulation sequence

The simulation was run in **Modelsim** using the test-bench, the Verilog netlist and the SDF data. This generated the VCD data, which contains the information about the value changes on variables in a design.

The tool **Primetime** read the Verilog netlist, VCD, SDF and the SPEF files and reported the power consumption for the given design. The report provides the break-up of power for different components like leakage, internal and switching. It also provides the power consumption by different modules in the IC. The same procedure was repeated for all three processor designs. The power simulations were performed at three different supply voltages of 0.6 V, 0.8 V and 1.2 V to analyse its effect on the speed and the power consumption by the processor. The simulation results are compiled and analysed in detail in chapter 5.

Chapter 5

Simulation results and Analysis

> If you torture the data long enough, it will confess.
>
> *Ronald Coase*

Power simulations were performed on the IC designed separately with three different transistor types - LPHVT, LPSVT and LPLVT. This was done to analyse the power performance of the Stack processor with different switching speeds and leakage current.

The power consumption by the transistor is given by the equation [10]:

Total power = Static power + Internal power + Switching power

$$\boldsymbol{P_t = P_{st} + P_{int} + P_{swi}} \tag{5.1}$$

$$\boldsymbol{P_{st} = V_{DD} I_{leakage}} \tag{5.2}$$

$$\boldsymbol{P_{int} = V_{DD} I_{SC} + \left(\frac{1}{2}\right) C_{in} V_{DD}^2 f} \tag{5.3}$$

$$\boldsymbol{P_{swi} = \left(\frac{1}{2}\right) C_L V_{DD}^2 f} \tag{5.4}$$

where

$$
\begin{aligned}
V_{\mathrm{DD}} &= Supply\,Voltage \\
I_{\mathrm{leakage}} &= Leakage\,current \\
I_{\mathrm{SC}} &= Short\,circuit\,current \\
C_{\mathrm{in}} &= Internal\,cell\,capacitance \\
C_{\mathrm{L}} &= Capacitive\,load\,at\,the\,cell\,output \\
f &= Clock\,frequency
\end{aligned}
$$

The leakage current $\mathrm{I_{leakage}}$ is due to the sub-threshold leakage current from source to drain of the transistor and the reverse-biased diode between the diffusion layers and substrate. The short-circuit $\mathrm{I_{current}}$ is due to the NMOS and PMOS transistors, within the gate, conducting simultaneously during switching.

The static power is due to the leakage current, whereas the switching power is due to the charging of the capacitive load at the output of the cell. The internal power is due to the charging of cell internal load capacitor and the short-circuit current through the NMOS and the PMOS transistors.

From the equations 5.1 - 5.4, it is evident that the power consumed by the transistor depends on the supply voltage, leakage current, short-circuit current, capacitive load and the clock frequency. Power consumption can be reduced by reducing any one or more dependent parameter. While the supply voltage and the clock frequency can be controlled externally, others are the characteristics of the transistor.

The power simulations were performed with different supply voltages, frequencies and transistors with different threshold voltages. The results of the simulations are discussed in the next few sections. Though most of the graphs show the performance of LPLVT devices, the performance of the other two types are similar with different numbers. The simulation results are tabulated in appendix B.

5.1 Simulation with different supply voltages

Power simulations were performed at three different supply voltages - 0.6 V, 0.8 V and 1.2 V. The power consumed due to switching and leakage current, the power consumed by the different modules of the processor and the maximum frequency of operation for each supply voltage were analysed.

5.1.1 Dependence of different power components on the supply voltage

The power simulations provide the information about the switching power and the leakage power consumption separately. As can be seen in the graph 5.1, both the switching power and the leakage power increases with the supply voltage.

5.1.2 Dependence of power consumption by different modules on the supply voltage

The different modules of the processor consumes power depending upon the number of transistors that it contains. The graph 5.2 shows the power consumed by each module at different supply voltages. Program memory consumes the highest power of all the modules as it is made up of registers. Since, the memory is accessed regularly to fetch the instruction, it is expected to consume a significant amount of power. This power can be reduced by using memory modules instead of registers. Memory module usage is justified when larger memory is used as it has the overhead of the address decoder logic circuit. Similarly the data stack, which is also made up of registers, also consumes considerable power. The reset controller, which operates only during reset, consumes the least amount of power.

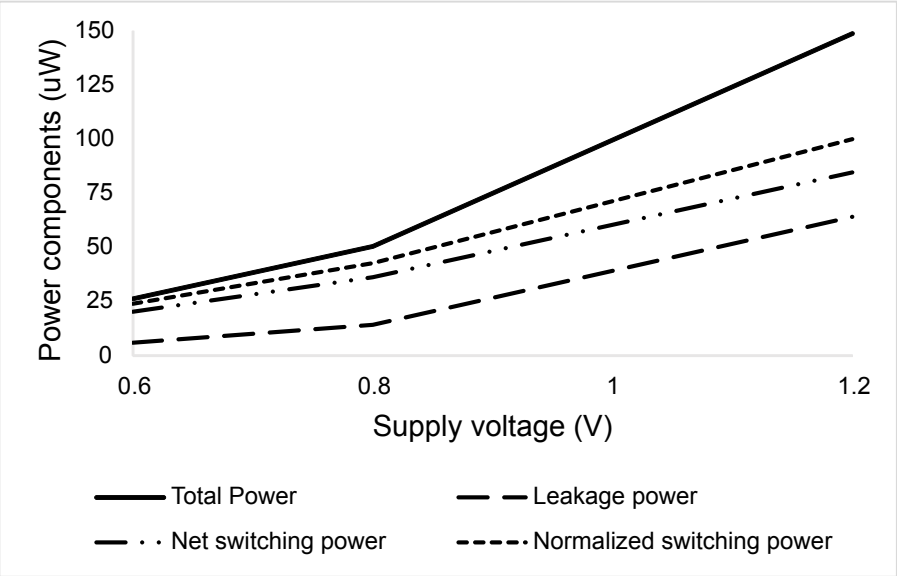

Figure 5.1: Different power components v/s Supply voltage

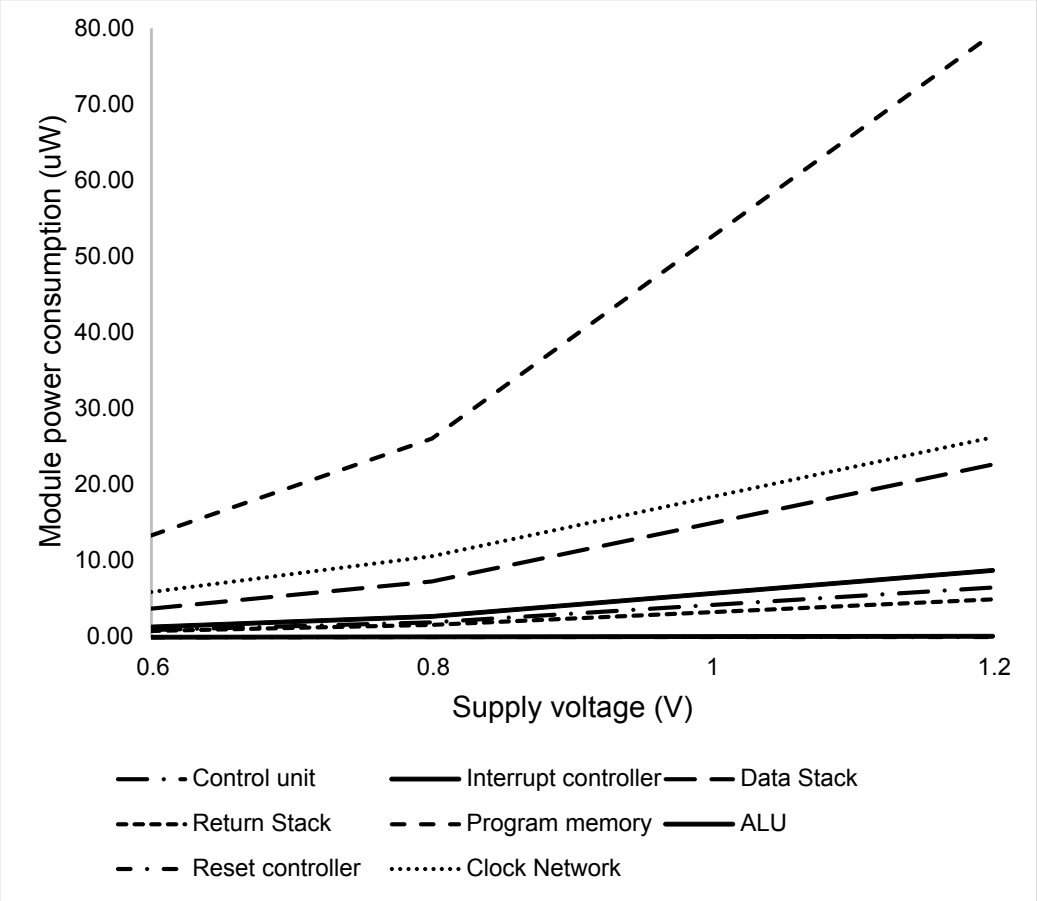

Figure 5.2: Power consumption by different modules v/s Supply voltage

5.1.3 Dependence of speed on the supply voltage

The slack is dependant upon the supply voltage. The slack increases as the supply voltage decreases. Hence, the maximum clock frequency has to be reduced at lower voltages. The graph 5.3 shows the maximum frequency that was attained at different voltages for different transistors. Though the frequency of LPLVT transistor could have been increased by a few tens of megahertz, it was not done as the low power microprocessors are rarely operated above 50 MHz. The maximum frequency of the processor with HPLVT transistors was half of that of other types.

The low voltage operation, at 0.6 V, clearly shows the difference in maximum

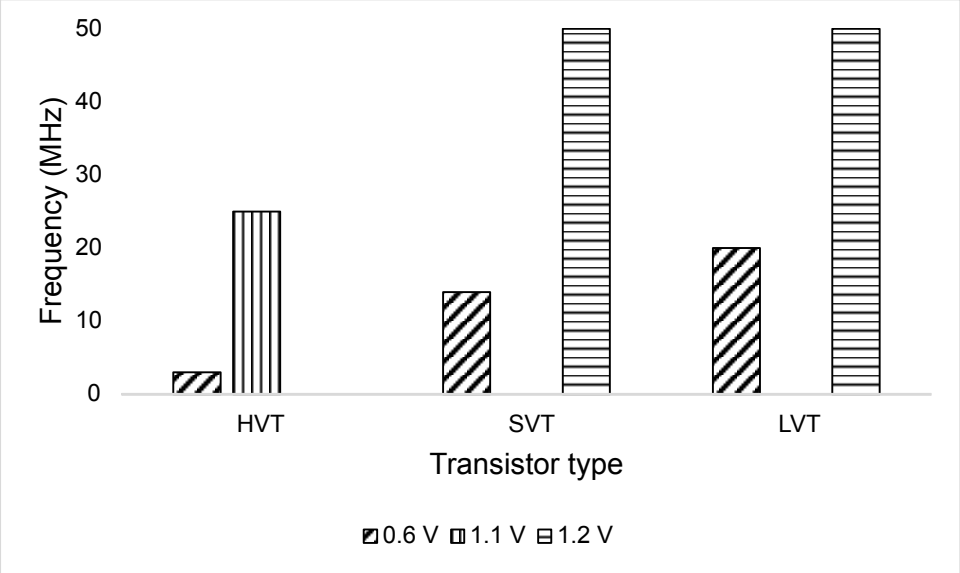

Figure 5.3: Maximum frequency v/s Supply voltage

frequency of operation among the transistors with different threshold voltages. The LPLVT design, with the minimum threshold of all three types, can be operated at highest frequency, while the LPHVT design, with the maximum threshold could only be operated seven times slower.

5.2 Simulation with different clock frequencies

The power consumption due to switching and leakage current at different frequencies were analysed. The graph 5.4 shows the way different components of power vary with the clock frequency. The net switching power and the internal power increased linearly with the frequency with different slopes as expected. However, the leakage power was constant for a given supply voltage (0.6 V), irrespective of the frequency.

5.3 Simulation with different transistor threshold voltages

The power consumption by the processors designed with transistors having different threshold voltages was analysed. The three different power components vary differently with the transistor threshold voltage. LPHVT transistors, with lowest

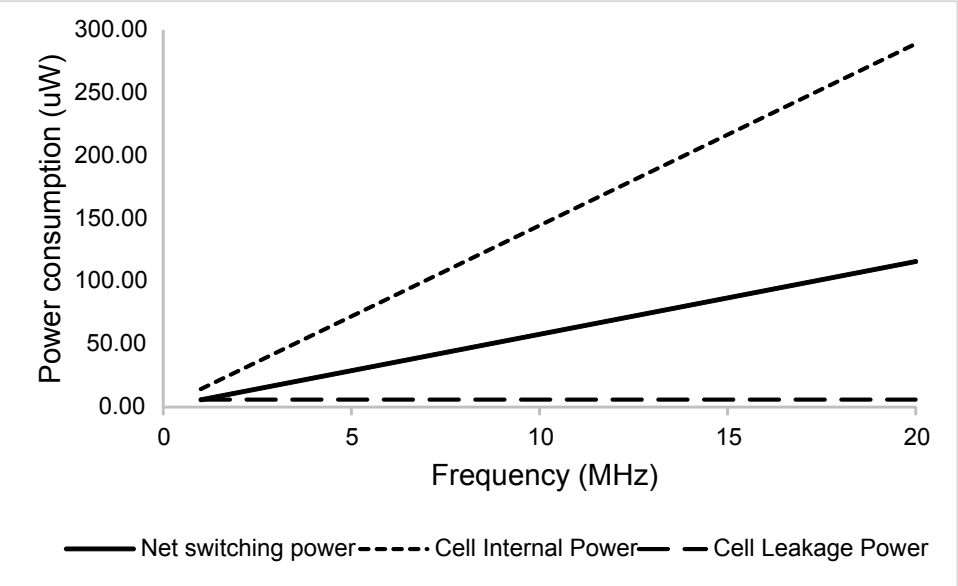

Figure 5.4: Total power consumption v/s Frequency

leakage current consumes least static power, while LPLVT transistors consume the highest static power. Net switching power is almost the same in all three cases as the capacitive load at the output is same.

The internal power consumption is highest for the LPHVT transistor and least for the LPLVT transistors. This component is dependent on both the short-circuit current, the transition time of the input signals and the internal capacitive load. Transistors with slow transition time dissipate excessive short-circuit power as both PMOS and NMOS transistors are switched on for a longer period of time. The summation of all the power components shows that the least power is consumed by the LPSVT transistors as they have the leakage power closer to LPHVT devices and the internal power closer to that of LPLVT devices, thus possessing the advantages of both types. All these behaviours are seen in the graph 5.5.

The processor with LPHVT transistors was designed with pads, while the rest was designed without pads due to the non-availability of suitable pads. The power consumption by the pads might slightly affect the readings.

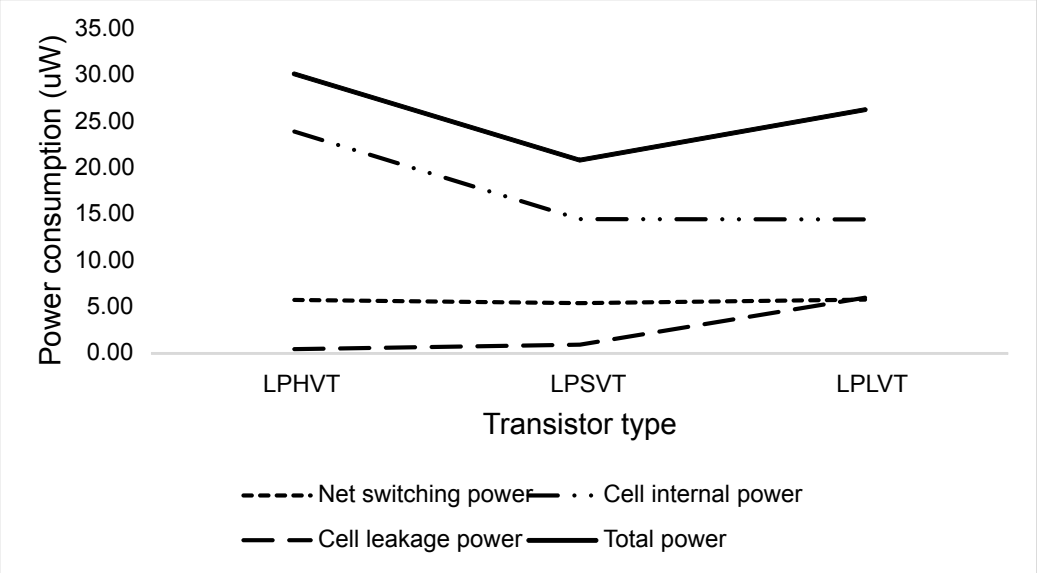

Figure 5.5: Total power consumption v/s Threshold voltage

5.4 Simulation at different corners

The power consumption by the processor at two corners, i.e. with Slow-Slow (SS) and Fast-Fast (FF) NMOS and PMOS transistors is seen in the graph 5.6. The FF configuration consumes more power compared to SS, as it operates at supply voltage of 1.3 V. Though the leakage current for FF corner is much lower compared to SS due to higher operating temperature of 125°C, the higher switching power at FF more than offsets the lower leakage power.

5.5 Simulation with different instructions

The effect of instruction type on the switching and the leakage power and on the power consumption by different modules of the processor were analysed.

5.5.1 Effect of instructions on different power components

The power consumption is expected to vary with the type of instructions executed. Programs with instructions, which access different modules in the processor, consume more power as it has to switch several transistors. The program with only the NOP (No Operation) instructions, doesn't access any module other than the control unit and hence consumes minimum power. This is seen in the graph 5.7.

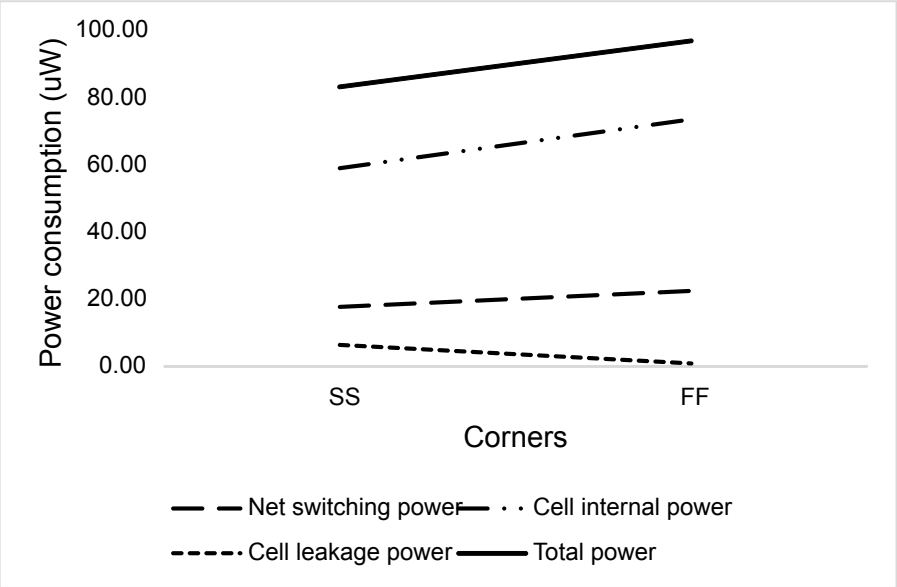

Figure 5.6: Total power consumption at different corners

5.5.2 Effect of instructions on power consumption by different modules

As seen earlier, other than the program memory and the clock network, the modules which consume considerable power are the data stack and the ALU. In the programs that have intensive interrupt service or function calls, the return stack also adds to the power consumption. The comparison of two programs - one with all types of instructions and the other with only NOPs - demonstrates this fact. This is shown in the graph 5.8.

5.6 Analysis

The power simulations, which were performed in the earlier sections help us to design the processor with the required performance and power efficiency. There is always a trade-off between the speed and the power efficiency. The LPLVT transistors help build a faster processor, but consume higher power, while the processor with LPSVT transistors is slower than that with LPLVT, but consume lesser power. On the other hand, the processors with LPHVT transistors consume higher power and also slowest. A better approach would be to build a processor with all types of transistors, each being used according to its strength. The modules with lesser

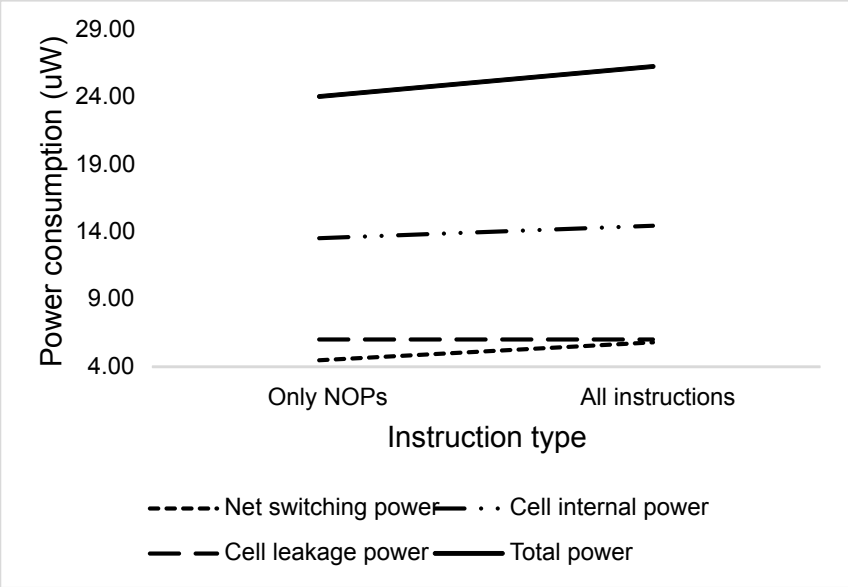

Figure 5.7: Power components v/s Instructions

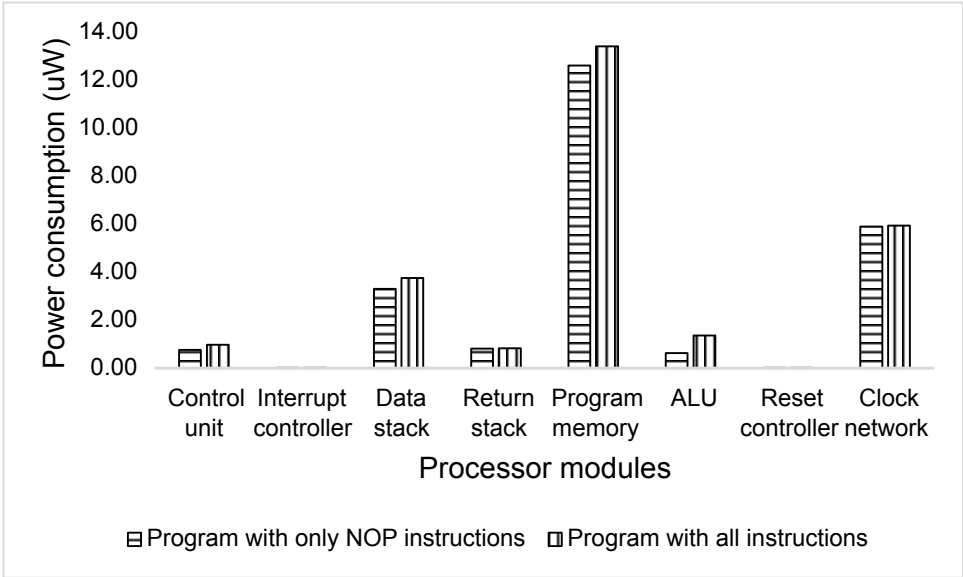

Figure 5.8: Power consumption by different modules v/s Instructions

switching can be built with LPHVT transistors while those with higher switching can be built with either LPLVT or LPSVT transistors.

5.7 Comparison

There is always a need to benchmark the processor to assess its performance relative to other devices. Benchmarking needs a suite of programs, developed in high level language such as C, to be executed on the target processor and performance assessed. Since, the stack processor developed in this project doesn't have a compiler developed yet, this benchmarking process couldn't be carried out. Hence, a general comparison of the power figures of this processor was made with a few selected commercial processors.

Microcontrollers from some of the well-known vendors were selected for comparison. The power ratings of these devices were obtained from their data sheets and tabulated in table 5.1. Care was taken to make a reasonable comparison:

- The entry-level devices from the low-power families of the controllers were selected because, they have a bare minimum hardware and hence consume very low power. It was thought that these devices would provide a fairly good comparison with the stack processor, which also has basic hardware modules.
- The power consumption at 1 MHz or power consumed per MHz for every microcontroller (as available from the data sheets) was used for comparison.

Comparing the power figures of the Stack processor, whose power ratings range from 80-100 uW/MHz at 1.2V and 20-30 uW/MHz at 0.6V, with those in the table it can be seen that the Stack processors can be highly power efficient at the same clock speed. Though the performance of the Stack processors may be slightly lesser than the RISC processors due to the absence of a full fledged instruction pipeline, it can be made up by increasing the clock frequency. The resulting increased power consumption would still be considerably lower than that of RISC processors and hence providing a significant power and performance advantage.

However, this comparison methodology has several drawbacks:

- The figures provided in the table were obtained by the vendors under different circumstances, i.e. by executing different suites of programs, with different compiler settings.
- Though the power figures are normalised in the table for comparison purpose, they were measured at different voltages and frequencies.

Table 5.1: Comparison table

Processor	**Data size**	**Power rating**	**Power efficiency**
PIC12F1571 (Microchip)	8-bit	30 A/MHz at 1.8V	54 W/MHz
STM8L101x1 (ST Microelectronics)	8-bit	150 A/MHz at 1.65V	247 W/MHz
MSP430C1101 (Texas Instruments)	16-bit	160 A at 1 MHz and 2.2 V	352 W/MHz
ATtiny4 (Atmel)	8-bit	200 A at 1 MHz and 1.8V	360 W/MHz
MAXQ613 (Maxim Integrated)	16-bit	271 A/MHz at 1.7V	460 W/MHz

- The microcontrollers used for comparison have a different set of peripherals like timers, communication ports, IO pins, etc. and hence hardware of each device is different.
- The data word size is not the same for all the devices.

Though this is not a fair method for power performance assessment, this just gives an idea of where the implemented stack processor stands.

Chapter 6

Conclusion and Future work

> A conclusion is simply the place where you got tired of thinking.
>
> *Dan Chaon*

6.1 Conclusion

There is an increasing demand for power efficient microprocessors as they are an integral part of mobile devices. The RISC processors are complex machines which results in significant power consumption. The main factors, which contribute to power consumption in a RISC processors are multiplexers on either side of the register file and the instruction pipeline. The stack processor eliminates these two circuits and considerably reduces the hardware and hence the power consumption. The implied addressing removes the need for extra bits for register addresses and hence results in smaller program size. Also, the stack architecture incurs very low penalty for function calls and interrupts as the operations are always performed on a stack instead of the register file.

A stack processor was designed to verify the theoretical understanding. The processor was based on 65 nm CMOS technology. Separate layouts using three types of transistors - LPLVT, LPSVT and LPHVT, were made based on the same architecture to assess its speed and power performance. The testing was done with different supply voltages of 0.6 V, 0.8 V and 1.2 V and at frequencies ranging from 1 to 50 MHz. Various components of power, such as switching, internal and leakage power were analysed. Also, the power consumed by different modules of the processor was determined. All the analysis was based on the behaviour, post-synthesis and

post-layout simulations using the EDA tools.

The simulations showed that, when powered by a 0.6 V supply voltage, the processor based on the LPLVT transistors could run up to a speed of 20 MHz with a power efficiency of about 20 uW/MHz, while that with LPHVT transistors ran up to a speed of 3 MHz with a power efficiency of 30 uW/MHz. The performance of LPSVT processor was closer to LPLVT, but at a lower speed. The LPHVT processor was found to have considerably lower leakage power compared to other two, while it consumed a much higher switching power. The gate count of the processor ranged from 46K to 48K depending upon the type of transistors and optimization used during place and routing.

A comparison of the power performance figures of few selected entry-level low power processors was done with that of the stack processor. The stack processor showed a remarkable power performance in this comparison. Benchmarking was not performed for the stack processor due to the non-availability of C compiler. Though not verified, stack processor is expected to suffer from a slight speed performance loss due to the absence of an instruction pipeline. But this loss can be made up by running the processor at a higher clock rate without significantly increasing the power consumption.

Though the stack processor may not be suitable for every application, it can provide a power-efficient solution for certain application such as process control, data acquisition and instrumentation.

6.2 Future Work

The stack processor developed in the thesis project has only basic modules of a processor. It can be made more powerful by adding peripherals such as input/output ports, peripherals such as timers, communication interface and sensor interface.

The current design uses a registers store the program. This is justified for low memory size. When the processor has to run larger programs, then the registers have to replaced by a memory module. It is more efficient in terms of size and power for larger size.

Though the processor serves interrupts, it doesn't recognize priority or allow interrupt nesting. Also, the interrupt latency is quite high and varies between 9 to 15 cycles, depending upon the interrupt arrival with respect to the current instruction that is executed. The Interrupt controller function has to be improved for a better program execution.

A power manager can also improve the power performance by switching the peripherals and memory on and off as required by the program and also by changing the clock frequency.

Appendix A

Instruction set architecture

I call architecture frozen music.

Johann Wolfgang von Goethe

A.1 Instruction set

Table A.1: Instruction set

Sl.no	Instruction type	Instruction	Description
1	ALU - Arithmetic	ADD	Add TOS and NOS with sign
2		ADDI	Add TOS and immediate literal with sign
3		ADDU	Add TOS and NOS unsigned
4		ADDIU	Add TOS and immediate literal unsigned
5		SUB	Subtract NOS from TOS with sign
6		SUBU	Subtract NOS from TOS unsigned
7		SUBI	Subtract immediate literal from TOS with sign
8		SUBIU	Subtract immediate literal from TOS unsigned
9		INC	Increment TOS by 1
10		DEC	Decrement TOS by 1
11		MUL	Multiply TOS and NOS unsigned
12	ALU - Logic	NOT	Invert TOS
13		AND	Logic AND of TOS and NOS

Continued on next page

Table A.1 – Instruction set

Sl.no	Instruction type	Instruction	Description
14	ALU - Logic	OR	Logic OR of TOS and NOS
15		XOR	Logic XOR of TOS and NOS
16	ALU - Shift	SLL	Logical Shift Left TOS
17		SRL	Logical Shift Right TOS
18		SLA	Arithmetic Shift Left TOS
19		SRA	Arithmetic Shift Right TOS
20	Branch	JMP	Unconditional jump
21		JZ	Jump on Zero flag
22		JNZ	Jump on no Zero flag
23		JC	Jump on Carry flag
24		JNC	Jump on no Carry flag
25		JNEG	Jump on Negative sign flag
26		CALL	Call function
27		RET	Return from function
28		RETI	Return from an ISR
29	Stack	PUSHI	Push literal into the Data stack
30		DROP	Pop TOS
31		DUP	Duplicate TOS
32		SWAP	Swap TOS and NOS
33		OVER	Push NOS over TOS
34	Interrupt	IE	Enable Interrupt
36		ID	Disable Interrupt
37		SWI	Software Interrupt
38	Others	NOP	No Operation

A.2 Instruction codes

The instruction encoding scheme shown in table A.2 is based on a previous work on stack processor [11].

Table A.2: Instruction codes

Instruction Group code	111	110	101	100	011	010	001	000
Instruction Function code	**Binary**	**Immediate**	**Unary**	**Unconditional branching**	**Conditional branching**	**Stack**	**Reserved for future use**	**Miscellaneous**
000	ADD	ADDI	INC	JMP	JZ	DROP		NOP
001	ADDU	ADDIU	DEC	CALL	JNZ	DUP		
010	SUB	SUBI	NOT		JC	SWAP		
011	SUBU	SUBIU			JNC	OVER		
100	AND		SRL	RET	JNEG	PUSHI		IE
101	OR		SLL	RETI				ID
110	XOR		SRA					SWI
111	MUL		SLA					

Table A.3: Auxiliary codes

Auxiliary codes (Bits 7 & 6)	**Instruction type**
00	ALU, Miscellaneous operations
01	Immediate data operations
10	Unconditional branch operations
11	Conditional branch operations

A.3 Instruction bit map

The opcode size is 8 bits, whose bit map is shown in table A.4. The bits 0 to 2 are provided by the Function code and the bits 3 to 5 are provided by the Group code, which are shown in table A.2. The bits 7 and 8 are provided by the auxiliary code as shown in table A.3. The code for any given instruction is formed by concatenating the auxiliary code, the group code and the function code as shown in table A.5.

Table A.4: Operation code bit map

Bit number	7	6	5	4	3	2	1	0
Description	Auxiliary code		Group code			Function code		

Table A.5: Operation code example

Instruction	**Auxiliary code**	**Group code**	**Function code**	**Op code**
ADD	00	111	000	00111000 (0x38)
SUBIU	01	110	011	01110011 (0x73)
RETI	10	100	101	10100101 (0xA5)
JC	11	011	010	11011010 (0xDE)

Appendix B

Simuation results

> A mind is a simulation that simulates itself.
>
> *Erol Ozan*

The simulation results are tabulated in this section. Though the simulations were performed on all three layouts, only one of them is produced here because the results are similar. The graphs of all the simulations are discussed in chapter 5.

Table B.1: Total power v/s Supply voltage (LPLVT)

Voltage (V)	**Power (uW)**			
	@20 MHz	@10 MHz	@5 MHz	@1 MHz
0.6	411.10	208.70	107.40	26.29
0.8	738.90	376.60	195.40	50.46
1.2	1757.00	910.60	487.40	148.80

Table B.2: Leakage and Switching power v/s Supply voltage at 1 MHz (LPLVT)

Voltage (V)	**Total power (uW)**	**Leakage power (uW)**	**Switching power (uW)**	**Normalized switching power (%)**
0.6	26.29	6.02	20.27	23.95
0.8	50.46	14.23	36.23	42.80
1.2	148.82	64.18	84.64	100.00

Table B.3: Power consumption by different modules v/s Supply voltage at 1 MHz (LPLVT)

Voltage (V)	Power (uW)							
	Control Unit	Interrupt Controller	Data Stack	Return Stack	Program memory	ALU	Reset controller	Clock network
0.6	0.98	0.02	3.76	0.83	13.40	1.36	0.01	5.94
0.8	1.95	0.04	7.33	1.62	26.10	2.73	0.02	10.67
1.2	6.56	0.15	22.7	4.99	79.30	8.78	0.05	26.28

Table B.4: Different power components v/s Frequency at 0.6 V (LPLVT)

Frequency (MHz)	Net switching power (uW)	Cell internal power (uW)	Cell leakage power (uW)
1	5.80	14.47	6.02
20	116.00	289.40	6.02

Table B.5: Power consumption by different transistor types at 0.6 V and 1 MHz

Transistor type	Net switching power (uW)	Cell internal power (uW)	Cell leakage power (uW)	Total power (uW)
LPHVT	5.76	23.92	0.46	30.14
LPSVT	5.42	14.50	0.93	20.85
LPLVT	5.80	14.47	6.02	26.29

Table B.6: Power consumption at different corners at 1 MHz (LPHVT)

Transistor configuration	Net switching power (uW)	Cell internal power (uW)	Cell leakage power (uW)	Total power (uW)
SS	17.77	59.01	6.44	83.22
FF	22.49	73.54	0.91	96.94

Table B.7: Effect of instruction on different power components at 0.6 V and 1 MHz (LPLVT)

Program	Net switching power (uW)	Cell internal power (uW)	Cell leakage power (uW)	Total power (uW)
Only NOPs	4.49	13.54	6.02	24.05
All instructions	5.80	14.47	6.02	26.29

Table B.8: Power consumption by different modules v/s Supply voltage at 0.6 V and 1 MHz (LPLVT)

Voltage (V)	**Power (uW)**							
	Control Unit	**Interrupt Controller**	**Data Stack**	**Return Stack**	**Program memory**	**ALU**	**Reset controller**	**Clock network**
Only NOPs	0.77	0.02	3.30	0.82	12.60	0.64	0.01	5.90
All instructions	0.98	0.02	3.76	0.83	13.40	1.36	0.01	5.94

Appendix C

History of Stack machines

> Study the past if you would divine the future.
>
> *Confucius*

Stack computer is one of the first computers that were produced. It is quite important to understand their evolution in the computer history, when a device of its type is designed. The information documented in this appendix is based on a bachelor thesis [7].

Stack computers are one of the first computers that were produced. Burroughs Corporation produced one of the earliest stack machines B5000, which was a mainframe computer, in 1961. It produced several other series up to B7900 and A series machines. The programs that ran on them were written in COBOL, ALGOL and FORTRAN languages. Another company, The English Electric, manufactured a stack machine, KDF9, in 1963. This also ran programs written in ALGOL. In 1974, International Computers Limited, announced ICL2900 series of mainframe computers. Hewlett Packard's HP3000 series minicomputer produced in 1972 was another machine based on stack architecture. It also ran COBOL programs.

Though the stack computers were one of the most sophisticated machines that were built at that time, they couldn't continue their run due to several reasons. Their hardware was designed to directly support high-level languages such as ALGOL and COBOL, through micro-coding. Hence, a machine optimized for one language performed poorly for another language. In 1980s, the RISC processors were designed to improve the throughput and performance, which was solidly backed by the advancement of C compiler technology. The earlier stack machines were no

match to these modern processors. There were other issues like single stack for both parameters and return address, which made parameter access by the function difficult. Also, the performance of iterative code on the stack machine is poor. All these made the stack machines loose popularity.

Though stack based microprocessors were developed in 1980s, they couldn't generate enough interest due to the tags attached by the first generation and some distorted historical facts. Most of the work on stack computers has emphasised on their simplicity, suitability for real-time applications and faster execution rate. But, very little has been written about its power efficiency, except that it is a low power device. One such study describes in detail the reasons for the stack computer to be power efficient. It can be understood why very less importance was given to its power efficiency, as most of the work in this area was done in the 1980s and 1990s. This was the time when the supremacy of the microprocessors was attributed to its speed rather than other characteristics.

In 1970, Charles (Chuck) Havice Moore developed a stack based language called `Forth`. All the stack microprocessors that were built from 1980s were the hardware implementation of this language. The first single-chip stack microprocessor that was designed was NC4016, by NOVIX. It was an 8 MHz, 16-bit processor, designed with about 16000 transistors in 3 um High-speed CMOS technology and ran Forth language programs. It differed from the earlier stack architectures in that it had two separate stacks for data and return address, both implemented outside the main memory.

Harris Corporation (now Intersil Corporation) produced RTX2000, which was derived from NC4016 in 1988. It was a 10 MHz, 16-bit processor built in 2 um CMOS standard cell technology. RTX2010RH is a radiation-hardened version of RTX2000. It was built using 1.2 um CMOS technology and mainly used in space applications. Some of the several missions where this processor was used are IMAGE (for imaging the Earth's magnetosphere), Rosetta and Philae (for studying the comet 67P/ChuryumovGerasimenko), Deep Impact (for studying the comet 9P/Tempel) and the Advanced X-ray Astrophysics Facility. The main reason for using this processor in the Rosetta and Philae mission was that this was the only radiation-hardened processor that was available with the lowest power budget.

A 32-bit dual-stack processor named Sh-BOOM was designed by Chuck Moore in 1988, which also had 16 general-purpose registers for temporary storage and IO operations. It was marketed by Patriot Scientific as IGNITE I processors. In 1995, it was followed by MuP21, a 21-bit processor manufactured in 1.2 um with 7000 transistors and running at 20 MHz. This was the first processor with Minimal Instruction Set Computer (MISC) design and was targeted towards video applications. There were other stack processors named F21 AND c18 designed and

prototyped. Besides, there were several designs prototyped on FPGA, but nothing was commercialized.

In 2009, Charles Moore founded a company GreenArrays, Inc [12]. It has produced a multi-computer chip GA144, which is an 18x8 array of identical, independent, F18A computers, each of which operates asynchronously.

Stack processors has generated a good amount of interest also in the academia. Prof. Philip Koopman of Carnegie Mellon University has done extensive study on Stack processors and also had worked on RTX2000 processors [13]. Prof. Chris Crispin-Bailey of the University of York also has done considerable amount of work on stack processors [14]. There also have been some amount of work in the universities in the form of research, doctoral, masters and bachelor thesis.

Bibliography

[1] A. Chapyzhenka, D. Ragozin, and A. Umnov, "Low-power architecture for CIL-code hardware processor," *Programming Problems*, vol. 4, pp. 20–38, 2005.

[2] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2nd ed., 1998.

[3] P. J. Koopman, Jr., *Stack Computers: The New Wave*. Hemel, Hemstead, Herts, UK: Ellis Horwood, Ltd., 1st ed., 1989.

[4] P. J. Koopman, Jr., "The WISC Concept," *BYTE*, 1987.

[5] C. H. Squin and D. A. Patterson, "Design and Implementation of RISC I," 1982.

[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2011.

[7] C. E. LaForest, "Second-generation Stack Computer Architecture," Master's thesis, University of Waterloo, 2007.

[8] A. Vachoux, "Top-down digital design flow." `http://lsm.epfl.ch/files/content/sites/lsm/files/shared/Resources%20documents/TopdownDF.pdf`, 2011. Accessed: September 8, 2015.

[9] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2nd ed., 2003.

[10] G. Yip, "Expanding the Synopsys PrimeTime Solution with Power Analysis." `https://www.synopsys.com/Tools/Implementation/SignOff/CapsuleModule/ptpx_wp.pdf`, 2006. Accessed: September 8, 2015.

[11] K. Schleisiek, "Microcore an Open-Source, Scalable, Dual-Stack, Harvard Processor Synthesisable VHDL for FPGAs." http://www.microcore.org, 2004. Accessed: September 8, 2015.

[12] C. H. Moore, "Greenarrays, Inc.." http://www.greenarraychips.com/, 2009. Accessed: September 8, 2015.

[13] P. J. Koopman, Jr. http://users.ece.cmu.edu/~koopman/. Accessed: September 8, 2015.

[14] C. Crispin-Bailey. http://www-users.cs.york.ac.uk/~chrisb/. Accessed: September 8, 2015.