The Design of a Custom 32-Bit SIMD Enhanced Digital Signal Processor

Shashank Simha
ss7841@rit.edu

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Master's Project is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
The Design of a Custom 32-bit SIMD Enhanced Digital Signal Processor

by
Shashank Simha

Graduate Paper
Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Electrical Engineering

Approved by:

Mr. Mark A. Indovina, Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
December 2017
To my family and friends, for all of their endless love, support, and encouragement throughout my career at Rochester Institute of Technology
Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This paper is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Shashank Simha

December 2017
Acknowledgements

I would like to thank my advisor, professor, and mentor, Mark A. Indovina, for all of his guidance throughout the entirety of this project. The continuous feedback and motivation provided by him has been a major driving force to push myself beyond limits throughout my career at RIT, for which I am truly grateful. His passion for teaching, expertise in digital design, along with decades of industrial experience has established him as my role model in the field. His advice, methods of teaching, managing and cross-domain knowledge has been a huge inspiration for me to pursue a career in the VLSI and digital design.

I would like to thank Dr. Dorin Patru and Dr. Marcin Lukowiak for providing me valuable knowledge and feedback in topics of computer architecture and FPGA, which provided a firm foundation in my understanding of the topics.

I would like to thank my parents for their continuous support throughout my career at RIT, believing in me and my being biggest role models. They have always been my pillars of support and great motivators throughout my life, at and away from home.

I would also like to thank my roommates for being my brothers throughout the two years of graduate school.

I finally would like to thank all my classmates and TA’s for their invaluable guidance and support throughout my entire career at RIT.
Abstract

For a number of years, the hardware industry has seen a drastic rise in embedded applications. Thanks to the Internet of Things (IoT) revolution, a majority of these embedded applications are shifting towards the usage of simple hardware capable of running on batteries, while being able to handle complex data and implement complex algorithms. Translating these requirements to digital design terms, the hardware is expected to have high power efficiency, be tiny and simple enough, while being capable of meeting real-time constraints and process mathematical algorithms. Looking at some of the modern DSPs, most of them have been targeting high performance and wider applications, usually resulting in higher power consumption and complex hardware.

The main motivation of this paper was to implement a simple DSP design, optimized for power efficiency, while being capable of handling simple multimedia applications. Hence, an enhanced version of TMS32010 DSP is implemented with numerous modifications to the architecture, ISA, memory addressing and pipeline structure. The major enhancements include the addition of instruction level parallelism using SIMD instructions, use of a much larger data memory to be able to accommodate a larger amount of data in multimedia applications, and expansion of the data-word to 32-bits to be able support packed SIMD data and fully utilize the 32-bit ALU. The ISA, pipeline and memory access enhancements target higher power efficiency by using a single clock across the design.
# Contents

Declaration ii  
Acknowledgements iii  
Abstract iv  
Contents v  
List of Figures vii  
List of Tables viii  

## 1 Introduction  
1.1 DSP classifications ........................................... 2  
1.2 History of DSPs .................................................. 3  
1.3 Brief introduction to the DSP design and paper organization .... 6  

## 2 DSP architecture  
2.1 Top level block diagram ...................................... 10  
2.2 Internal blocks .................................................. 11  
2.2.1 Address decode unit ........................................ 12  
2.2.2 Execution unit ............................................... 13  
2.2.3 ALU .......................................................... 15  

## 3 Instruction Set Architecture of the DSP  
3.1 Instruction and data word expansion ........................ 18  
3.2 Addressing modes ............................................... 18  
3.2.1 Direct addressing .......................................... 21  
3.2.2 Indirect addressing ........................................ 22  
3.3 Instruction opcodes and operation ............................ 23  
3.3.1 List of instructions and corresponding opcodes ............ 23  
3.3.2 Description of the operation of each instruction ............. 27
## Contents

4 DSP Pipeline and Read/Write RAM buffer wrapper implementation 32
   4.1 Pipeline implementation 33
      4.1.1 Pipeline stages 33
      4.1.2 Pipeline design for non-branching instructions 35
      4.1.3 Pipeline design for unconditional branching instructions 37
      4.1.4 Pipeline design for conditional branching instructions 40
   4.2 Read/write RAM buffer wrapper 43
      4.2.1 RAM read/write problem description 44
      4.2.2 Design and implementation of read/write buffer wrapper 45

5 Median filter design 47
   5.1 Median filter overview 48
   5.2 Median filter design and implementation 48

6 Results 52
   6.1 Results 52

7 Conclusions and future work 54
   7.1 Conclusion 54
   7.2 Future work 54

References 56

I Source Code I-1
   I.1 RTL source code I-1
      I.1.1 DSP top level module I-1
      I.1.2 ALU I-25
      I.1.3 Input shifter I-32
      I.1.4 Output shifter I-35
      I.1.5 Compare select unit I-38
      I.1.6 Multiplier I-39
      I.1.7 Adder I-40
   I.2 Assembler designed in Perl I-41
   I.3 Assembly source code for testing and median filter I-55
      I.3.1 Assembly code used for basic level testing I-55
      I.3.2 Assembly code used for median filter algorithm I-57
# List of Figures

1.1 Fixed and floating point illustration ............................................. 2
2.1 Top-level block diagram .............................................................. 10
2.2 Address decode unit block diagram ............................................... 13
2.3 Execution unit block diagram ....................................................... 14
2.4 ALU block diagram .................................................................... 16
3.1 Instruction word expansion for various instructions ......................... 19
3.2 Data word expansion ................................................................... 20
3.3 Direct addressing illustration ....................................................... 22
3.4 Indirect addressing illustration ..................................................... 23
4.1 Pipeline stages and implementation ............................................... 34
4.2 Pipeline example for memory read instructions .................................. 36
4.3 Pipeline example for memory write instructions ............................... 38
4.4 Pipeline example for unconditional branching ................................. 40
4.5 Pipeline implementation example for conditional branch instruction, when condition is false ......................................................... 42
4.6 Pipeline implementation example for conditional branch instruction, when condition is true ......................................................... 43
4.7 Read/write RAM buffer wrapper state machine ................................. 45
5.1 Median filter working illustration .................................................. 49
5.2 Median filter algorithm .................................................................. 49
5.3 Median filter algorithm implementation illustration for a $3 \times 3$ window 51
## List of Tables

3.1 List of Instructions and their opcodes ........................................... 23
3.1 List of Instructions and their opcodes ........................................... 24
3.1 List of Instructions and their opcodes ........................................... 25
3.1 List of Instructions and their opcodes ........................................... 26
3.1 List of Instructions and their opcodes ........................................... 27
3.2 List of instructions and their operations ....................................... 28
3.2 List of instructions and their operations ....................................... 29
3.2 List of instructions and their operations ....................................... 30
3.2 List of instructions and their operations ....................................... 31
6.1 Synthesis results ........................................................................... 53
Chapter 1

Introduction

With advancement in technology, the world has been seeing exponential increase in the amount of data stored and processed ever since computers have been invented. A major part of this data represents multimedia, which is essentially either audio or image data [1]. To clearly compress, restore, process and understand image data, numerous mathematical algorithms have been implemented in computing, which are usually quite complex. After the invention of general purpose processors, there were many applications where a lot of its functions were not required by the application, or used by limited applications [2]. And, these processors took too much time to compute the mathematically intense algorithms in real time, which the hardware was simply not built to handle. This market was targeted by DSPs (Digital Signal Processors). DSPs have historically been used in such applications to increase the speed of computing by implementing complex hardware and parallel computing [3].
1.1 DSP classifications

DSPs are broadly classified into fixed and floating-point architectures. Fixed-point DSPs are designed to handle positive or negative integer data, while floating-point DSPs are designed to handle rational number data. The representation of data stored in each of these DSPs hence is different, which is the major reason behind the classification since it directly affects the amount of hardware required for each implementation. The fixed-point data is represented by the integer’s sign in the MSB (Most Significant Bit) followed by its value in the following bits. Floating-point data is represented by the rational number’s sign in the MSB, followed by its exponent, and later its mantissa. Fig. 1.1 illustrates fixed and floating point representations.[4]

Generally, fixed point implementations are faster, cheaper, more power efficient, simpler to design and verify, and require less time-to-market. Floating point implementation
1.2 History of DSPs

trades-off all these factors for better precision and faster computation of floating point data. Hence most of the times, either of the architectures is selected mainly based on the application. It is also worth noting that some DSPs are equally efficient in implementing either architectures like the SHARC DSP by Analog Devices.

Architecturally, the amount of hardware and design effort required to implement floating point precision is obviously much higher than fixed point precision. First, the data unit must be expanded from 16-bits to 32-bits at least along with the memory and registers. The ISA itself would have to be expanded significantly as almost all floating point DSPs usually support fixed point operations along with floating point ones [5].

From a compilation point of view, C language has an in-built type for floating-point to fully exploit the hardware capability of floating point DSPs. While the C compiler takes advantage of the floating point hardware, some rules and regulations are followed to ensure that the data fits within the 32-bit or 64-bit data word. Fixed point C compilation is implemented by mapping integers to fixed point data. The problem with fixed point though is that there is no ANSI standard for fixed point, hence it usually requires additional code for conversions and shifts. The efficiency of fixed point compilation takes another dip, since fixed point specific instructions are not built-in [6].

1.2 History of DSPs

The main motivation for DSP was to have powerful hardware with more application specific functions and instructions, when compared to a general-purpose processor. This is very evident throughout the evolution of DSPs looking at the various applications throughout the past few decades. Moreover, it is obvious that all early DSPs were fixed-point architectures mainly because there was no floating-point standardization until early 80’s. With
applications ranging from audio systems, speech processing, SONAR to medical imaging, 
RADAR, DSPs are used almost in every field today [7]. It is also interesting to note early 
applications of DSPs in personal computers like the Motorola 56000 used in the Atari 
Falcon, NeXT and SGI workstations.

DSPs have been produced by almost all major semiconductor companies including Intel, 
AMD, Texas Instruments, Motorola and Analog Devices at some point of time. Most of 
the early DSPs targeted audio processing, such as the Speak & Spell by Texas Instruments. 
Throughout the evolution of DSPs, they have grown more and more application specific 
over time, rather than the other way around [5].

Speak & Spell, an early toy used to teach kids to spell words, launched in 1976, was 
the earliest mass-produced DSP product in the market, powered by the Texas Instrument 
TMS5100 DSP [8]. Interestingly, in the late 70’s Intel unsuccessfully tried to enter the DSP 
market early with their 2920 analog processor, which failed mainly because of the absence 
of a true multiplier. The first attempts of DSP devices include the AT&T DSP1 and NEC 
µPD7720. It is worth noting that DSP1 introduced the historic MAC instruction to the 
world, this was one of the earliest steps of implementing instruction level parallelism in 
DSPs [9].

The first generation of DSPs started appearing in the market in the early 1980’s. Some 
key features of the DSPs of this generation were Harvard architecture and multiply-add-
accumulate instructions. The TMS32010, from this generation of DSPs, by Texas In-
struments was notably one of the most successful DSPs in history, as it pushed Texas 
Instruments to be the market leader in DSPs. Since it was based on Harvard architecture 
and with specialized ISA, it was the fastest DSP at the time [9].

The next generation DSPs, from late 1980’s to early 1990’s featured advanced archi-
tectures with capability of handling much complex applications. Motorola entered the
DSP market with their popular fixed-point DSP56000 featuring 24-bit program and data words. The second generations of DSPs featured further optimization in memory architecture, with architectures capable of accessing multiple data memories in a single instruction. This generation also brought floating-point DSP architecture into market. Examples for this include the SHARC series of DSPs by Analog Devices, calling the architecture Super Harvard architecture. Interestingly, shrinking fabrication technology also had a huge impact on this generation of DSPs, as more and more hardware could be fit into the chip while still keeping it tiny in size [9].

Late 90’s DSPs incorporated more application-specific instructions, as they were mostly used as coprocessors along with the main CPU. Many DSPs however lost market when CPUs became SIMD capable. Parallel processing capabilities were subsequently introduced with Single Instruction Multiple Data (SIMD) and Very Long Instruction Word (VLIW) instructions in the later DSPs. VLIW architectures take advantage of spatial parallelism along with temporal parallelism, since they utilize several functional units to concurrently execute multiple operations, while pipelining these functional units [10]. Parallelism was further boosted with adding multiple cores and threads in later DSPs [9].

Modern DSPs are functionally not very different from the late 90’s ones. Optimizations in the last decade though has been directed toward DSP compilation strategies. Designing and making DSP architecture more compiler friendly and making better DSP compilers has been a very crucial step in DSP evolution, mainly because its applications will tremendously increase since it is the bridge to the software world. As the software world starts utilizing DSPs effectively and understanding their capabilities, significant advantages in software productivity could be achieved [11]. Also, modern DSPs have been trying to incorporate as much parallelism as possible, with different approaches. The latest Texas Instruments DSP TMS320C64X is very good example of this, as it combines both VLIW and SIMD
1.3 Brief introduction to the DSP design and paper organization

The capabilities of DSPs have evidently evolved at a rapid pace over time, especially since the 90’s. With the introduction of concepts such as super Harvard architecture, VLIW and super-scalar architecture, the design complexity of DSPs has also risen at the same pace. Hence, the need for a simple embedded programmable processor with not only conventional instructions, but also DSP specific becomes desirable in some applications [12][13][14][15]. The paper [12] shows one such approach where, the TMS32010, a fairly simple monolithic DSP, is implemented on an FPGA platform.

The DSP design presented is very similar to the TMS32010 by Texas Instruments, but has major enhancements which are discussed in the later chapters. Looking at some applications of TMS32010 was a necessary step while deciding what enhancements could positively impact the DSP applications. Numerous applications of TMS32010 in the area of image processing involve design of a multiprocessor system using multiple DSPs to implement a complex algorithm. Paper [16] presents one such application where edge detection algorithm is implemented using eight TMS32010 DSPs in a multiprocessor configuration, involving parallel image processing architecture. Another interesting image processing application is presented in [17], where the TMS32010 is interfaced with a host processor and to speed up image processing algorithms. The paper also mentions limitations of the system, one of which is the lack of data memory in the TMS32010. This has been one of the
major enhancements in our DSP design.

The following chapters of the paper attempt to explain all details involved in designing the DSP. Chapter 2 talks about the architecture of the DSP, including the data flow within the DSP, types of operations and compares it other DSP architectures. Chapter 3 covers topics like instruction and data word expansions, opcodes of all instructions and addressing modes and assembler design, in an effort to describe the Instruction Set Architecture (ISA) of the DSP. Chapter 4 talks about pipeline design and RAM buffer wrapper design. Chapter 5 elaborates one application of the DSP which is the median filter design, and describes the merits of the DSP with respect to its implementation. Chapters 6 and 7 present the results and conclusion of the paper.
Chapter 2

DSP architecture

The DSP architecture is very different from that of a general-purpose CPU as discussed in the previous chapter. One of the biggest bottlenecks in executing DSP algorithms is transferring information to and from memory [5]. Things like Harvard architecture, direct memory access (DMA), multiply-accumulate unit (MAC) and barrel shifting are some features which distinguish DSPs from general purpose processors. Some DSPs have general purpose registers, like the SHARC ADSP-2106x, and others are accumulator based, such as the TMS32010 [5].

As noted earlier, the primary advantage of the DSP is its speed. This means its architecture needs to be capable of performing complex mathematical calculations within a single clock cycle. There are different techniques to achieve this architecturally. Firstly, pipelining the architecture ensures that most instructions are executed within a single clock cycle, but have a latency between the input instruction and its output equal to the number of pipelined stages. While this approach is attractive, one thing to remember is that the same resource can not be used in multiple stages. The second solution is parallel execution of multiple tasks. The important point to remember here that these tasks should not have
dependencies, and of course can not use the same resources. DSP architecture is usually designed by combining both techniques to accomplish the speed.

The DSP's architecture being its most important feature, its significant difference of the from that of conventional microprocessors comes obvious. The basic capability of integrating a multiplier/accumulator into its data-path has been proven to be revolutionary in computing multiple algorithms. Other factors such as preserving the precision of the product after multiplication, having shift capability while storing accumulator into the memory and handling overflow are crucial for DSP architecture, since most of its applications are usually complex arithmetic operations, requiring precise calculations [18].

While chapters 3 and 4 deal with how exactly each instruction is planned and executed, and pipeline design of the DSP respectively, this chapter starts by taking a brief look at the architecture of top-level design. Later in the chapter, all other lower level blocks of the DSP including functional blocks of the architecture are discussed in detail.
2.1 Top level block diagram

The top-level block includes the DSP, ROM, RAM and read/write buffer wrapper. Fig. 2.1 attempts to visually describe the top-level view of the DSP system along with an abstracted high-level view of the interconnections between these components. It is to be noted that all the modules within the DSP are functional blocks, not pipeline stages. And, the necessity and usage of read-write RAM buffer is explained in chapter 4.

The DSP sends a new value for program counter (PC) every cycle, which goes to the ROM. The ROM returns the instruction corresponding to the previous PC, back to the DSP. Later, depending on the instruction, the DSP then sends a read address to the read/write RAM buffer to fetch operand/value from the RAM. The read/write RAM buffer in turn communicates this address to the RAM and accordingly fetches data from the RAM. This data is sent to the DSP, which executes the instruction and computes the
result. Lastly, depending on the following instruction, this result is saved into the RAM via the read/write RAM buffer.

### 2.2 Internal blocks

As discussed in the previous section, the DSP needs to perform the following functions for every instruction in the same order:

1. Fetch the instruction from ROM,
2. Decode this instruction,
3. Fetch data from RAM if necessary,
4. Execution the instruction, and
5. Save the result back into the RAM if required.

While the pipeline takes care of distributing these functions across multiple clock cycles, it is necessary to carefully plan the hardware necessary to perform each function.

The address decode unit 2.2.1 section describes how the instruction is broken into pieces as soon as the DSP receives it from the ROM. As the logic remains same for almost all instructions, most its implementation is described within the small subsection. However, execution being a more complex task, as it is unique for every instruction, requires further planning. Hence, the ALU 2.2.3 is separately discussed, following a brief look at the execution unit 2.2.2.
2.2 Internal blocks

2.2.1 Address decode unit

Fig. 2.2 shows the block diagram of decode unit. The decode unit supports two addressing modes, namely direct and indirect addressing modes. As clearly shown in the figure, eight Auxiliary Registers (ARs) are used for indirect addressing. The AR pointer (ARP) is used to indicate which AR is to be used. Chapter 3 discusses addressing modes in detail.

Direct addressing does not require much logic to decode, as the instruction itself contains almost all required details to generate the data memory address. Here, the least significant or LSB 7-bits of the instruction is simply concatenated with the contents of the data page pointer (DPP) to generate the data memory address.

While using indirect addressing, the instruction specifies the following details: which AR will be used for the next indirect addressing or the next AR pointer (NARP), and whether the contents of the current AR is to be incremented/decremented by one or not. The AR however contains the address of data to be fetched.
2.2 Internal blocks

2.2.2 Execution unit

Fig. 2.3 describes the block diagram of the execution unit. The execution unit is broken down into three major functional blocks: ALU, barrel shifter and multiplier. Apart from these, all the other components in the figure are used to facilitate the interconnection
between these blocks, as directed by the instruction word decode logic.

Figure 2.3: Execution unit block diagram
Two 32-bit barrel-shifters are used in the DSP. The input shifter is used to shift one of the ALU inputs, while the output shifter is used to shift the result of the ALU. The input data to the first shifter is read from the RAM. The second shifter is however used only while storing or writing back the accumulator contents into the RAM.

A 16x16 bit multiplier is used for multiplication operations. While one input to the multiplier always comes from the T-register, the other input can either be read from the RAM, or directly read from the instruction. The output however is always stored in the P-register.

The DSP is designed such that one of the ALU inputs is always the accumulator, while the other input is loaded either from the output of the first shifter or from the P-register. The result of the ALU is always fed back into the accumulator.

### 2.2.3 ALU

To accommodate SIMD instructions into the ISA, the ALU is optimized for add/subtract SIMD instructions. Fig. 2.4 shows the block diagram of the ALU. The main goal while designing the ALU was to add minimal hardware to TMS32010 design, while also supporting SIMD instructions.

Four sets of 8-bit adders were used to implement 32-bit addition-subtraction operations, as well as 8-bit SIMD addition-subtraction operations. The only change internally to the adders was the additional multiplexers between the carry signals of the consequent adders, which were set to zero for SIMD add. For the subtraction operation though, 32-bit 2’s complement was fed to the adders in the case of 32-bit subtraction operation, while separate 2's complements were fed to the adders for each set of 8-bits in the case of SIMD subtractor.
Figure 2.4: ALU block diagram
Chapter 3

Instruction Set Architecture of the DSP

One of the biggest challenges while designing a DSP is to strike a fair balance between the hardware complexity for implementing a particular ISA (Instruction Set Architecture), and possible applications of almost every instruction. For the past few decades, major DSP manufacturers have been experimenting with different instructions and ISAs to maximize the applications of their DSPs across various fields. While having a complex ISA and hundreds of instructions looks like a clear winner, a significantly huge number of DSPs are used in embedded and real-time applications where power, size and cost are extremely important. Interestingly, with the advent of Internet of things (IoT), there has been an exponential increase in such applications. Here, ISA complexity needs to be traded off for flexibility and robustness.

Another very important factor to consider is the type of addressing, or how easy or flexible the address conversion logic is for a user. This part is even more crucial in DSPs as almost all DSPs fetch data from the RAM for each ALU operation, unlike CPUs where
numerous general-purpose registers are used as ALU operands. And, it is quite obvious that most of the DSP operations would be ALU operations. [19]

The proposed DSP architecture is very similar to TMS32010 in its ISA. In the next few sections, the instruction and data words of the DSP, addressing modes and instruction opcodes along with operations are briefly described.

3.1 Instruction and data word expansion

To be able to fit in all the instructions and corresponding data required for each instruction within 16-bits of instruction word, five types of instruction words are planned. Fig. 3.1 shows the expansions of these instruction words. It is to be noted that the expansions indicated in the figure are only for direct addressing mode. In the case of indirect addressing, the last 7-bits are used differently. Bits 0, 1 and 2 are used to indicate the value of next AR pointer (NARP), while bits 5 and 6 are used to indicate post increment/decrement operation for the current AR.

Data words can be stored in four different formats, depending on the type of instruction used to handle them. Fig. 3.2 shows all variants of data word, where D0, D1, D2 and D3 are 8-bit signed/unsigned integers. While all instructions handled by the ALU are 32-bit data words, SIMD instructions use the 8-bit variants.

3.2 Addressing modes

As indicated while describing the decode unit in chapter 2, two addressing modes are implemented in the DSP. These addressing modes function exactly like the TMS32010, except eight Auxiliary registers are used here instead of two. Also, since the DSP design
### Instruction Word Split

Width: 16-bits

1. **Instructions with shift**

<table>
<thead>
<tr>
<th>3-bits</th>
<th>5-bits</th>
<th>1-bit</th>
<th>7-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Shift</td>
<td>Mode</td>
<td>Memory</td>
</tr>
</tbody>
</table>

2. **Branch instructions**

<table>
<thead>
<tr>
<th>8-bits</th>
<th>1-bit</th>
<th>3-bits</th>
<th>4-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>Mode</td>
<td>111</td>
<td>Condition</td>
</tr>
</tbody>
</table>

3. **Instructions with constants**

<table>
<thead>
<tr>
<th>3-bits</th>
<th>5-bits</th>
<th>8-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Opcode</td>
<td>Constant</td>
</tr>
</tbody>
</table>

4. **Load/store AR instructions**

<table>
<thead>
<tr>
<th>5-bits</th>
<th>3-bits</th>
<th>1-bit</th>
<th>7-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>AR</td>
<td>Mode</td>
<td>Memory</td>
</tr>
</tbody>
</table>

5. **Other instructions**

<table>
<thead>
<tr>
<th>3-bits</th>
<th>5-bits</th>
<th>1-bit</th>
<th>7-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Opcode</td>
<td>Mode</td>
<td>Memory</td>
</tr>
</tbody>
</table>

Figure 3.1: Instruction word expansion for various instructions
3.2 Addressing modes

Figure 3.2: Data word expansion

Data Word expansion:

Width : 32-bits

1. Signed arithmetic data-

<table>
<thead>
<tr>
<th>1-bit</th>
<th>31-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign</td>
<td>Absolute data/ value</td>
</tr>
</tbody>
</table>

2. Unsigned arithmetic data-

<table>
<thead>
<tr>
<th>32-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer data/ value</td>
</tr>
</tbody>
</table>

3. SIMD signed data-

<table>
<thead>
<tr>
<th>1-bit</th>
<th>7-bits</th>
<th>1-bit</th>
<th>7-bits</th>
<th>1-bit</th>
<th>7-bits</th>
<th>1-bit</th>
<th>7-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3</td>
<td>D3</td>
<td>D2</td>
<td>D2</td>
<td>D1</td>
<td>D1</td>
<td>D0</td>
<td>D0</td>
</tr>
<tr>
<td>sign</td>
<td>value</td>
<td>sign</td>
<td>value</td>
<td>sign</td>
<td>value</td>
<td>sign</td>
<td>value</td>
</tr>
</tbody>
</table>

4. SIMD unsigned data-

<table>
<thead>
<tr>
<th>8-bits</th>
<th>8-bits</th>
<th>8-bits</th>
<th>8-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3</td>
<td>D3</td>
<td>D1</td>
<td>D0</td>
</tr>
</tbody>
</table>
3.2 Addressing modes

contains a much larger RAM, the data-page pointer here has been expanded to 8-bits from TMS32010’s single or double bit versions. [20].

Since multiplication is the only instruction which supports immediate addressing, the DSP does not really support immediate addressing when it comes to any other operations, including ALU operations. Therefore, immediate addressing is not claimed to be supported by the DSP.

### 3.2.1 Direct addressing

Though direct addressing seems fairly straightforward in implementation, it involves a two-step process. In the first step, the data-page pointer register needs to be loaded with the value of the most significant or MSB 8-bits of the address, using a separate load data-page pointer instruction. The second step is to specify the remaining 7-bits of the address in the least significant or LSB 7-bits of the instruction, while resetting the eighth bit of the instruction to indicate direct addressing mode.

Hence, the MSB 8-bits of the RAM address is considered the page. Or in other words, the RAM has 256 pages, each page consisting of 128 words. This means that, once we load the page address, accessing any data within the page could be done in a single instruction. The main drawback of this system though is that every time we need to access a different page, we must use an additional instruction to load the page address. The advantage though it that the instruction word remains small, and hence the power efficiency is high compared to a bigger instruction word. Fig. 3.3 illustrates the working of direct addressing.
3.2 Addressing modes

3.2.2 Indirect addressing

Indirect addressing is a multi-step process, since it accomplishes two things: selecting the next auxiliary register (AR) and incrementing/decrementing the current AR. The ARs hold address locations for the RAM data to be fetched, hence it is required to load them prior to using indirect addressing instructions.

The first step is to load one or more ARs with the value of the desired address location/s. Next, the AR pointer (ARP) needs to be set to the AR containing the next immediate address to be accessed. Later, the value of this AR can be incremented/ decremented, to be ready the next time the same AR is accessed.

Comparing indirect and direct addressing, indirect addressing is an attractive choice if the same data or its immediate neighbor is accessed multiple times. Since indirect addressing happens via the ARs, similar to direct addressing the ARs initially need to be loaded with address locations that are to be accessed. The advantage here is that these registers can be incremented/decremented every cycle, while also having the option of selecting which AR is to be accessed for the next operation.

Expanding the number of these registers is hence very helpful in numerous applications where consecutive data in multiple locations is required for the algorithm. The best example for such applications are image filters, where pixels within a 3x3 window of area are
3.3 Instruction opcodes and operation

The instruction set consists of a total of 50 instructions, including load/store, branching, data manipulation and SIMD instructions. Some instructions from the TMS32010 that are not implemented here are the table read and write, LTD, IN and OUT instructions. The Sections 3.3.1 and 3.3.2 list all the instructions and attempt to briefly describe their operations respectively.

3.3.1 List of instructions and corresponding opcodes

Table 3.1 shows the opcodes for all instructions. It is to be noted that all branching instructions will be followed by the jump address in the next instruction word, since it is not explicitly indicated in the table.

Table 3.1: List of Instructions and their opcodes

| Instr. | # | # | D | D | D | D | D | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|--------|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|
|        | Cyc | IW | 15 | 14 | 13 | 12 | 11 | 10 |    |    |    |    |    |    |    |
| 1 | MPY | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | M  | D  | D  | D  | D  | D  | D  | D  |
### Table 3.1: List of Instructions and their opcodes

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Cycles</th>
<th>#</th>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>D6</th>
<th>D7</th>
<th>D8</th>
<th>D9</th>
<th>IW</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Instr.**: Instruction name
- **Cycles**: Number of cycles required for execution
- **#**: Instruction number
- **D0** to **D9**: Destination register(s)
- **D10**: Destination register(s) (continued)
- **D11** to **D15**: Destination register(s) (continued)
- **IW**: Immediate word
- **15** to **10**: Instruction word

The instructions include:
- **MPYK**: Multiply
- **MAC**: Multiply Accumulate
- **OR**: OR Operation
- **XOR**: XOR Operation
- **SPAC**: Space Operation
- **SUB**: Subtract
- **SUBS**: Subtract with Source
- **ADD**: Add
- **ADDS**: Add with Source
- **AND**: AND Operation
- **BU**: Branch Unpredictably
- **BANZ**: Branch on Zero

The table entries represent the opcode for each instruction, with specific bits indicating the operation and destination registers.
### 3.3 Instruction opcodes and operation

#### Table 3.1: List of Instructions and their opcodes

<table>
<thead>
<tr>
<th>Instr.</th>
<th>#</th>
<th>Cyc</th>
<th>IW</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>0</td>
<td>AR2</td>
<td>AR1</td>
<td>AR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>BGEZ</td>
<td>2 2</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>BGZ</td>
<td>2 2</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>BLEZ</td>
<td>2 2</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>BLZ</td>
<td>2 2</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>BNZ</td>
<td>2 2</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
<td>0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>BV</td>
<td>2 2</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 1</td>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>BZ</td>
<td>2 2</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>LAC</td>
<td>1 1</td>
<td>0 1</td>
<td>1 S</td>
<td>S S</td>
<td>S S</td>
<td>M D</td>
<td>D D</td>
<td>D D</td>
<td>D D</td>
<td>M *</td>
<td>*</td>
<td>0 0</td>
<td>AR2</td>
<td>AR1</td>
<td>AR0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>LACK</td>
<td>1 1</td>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 K</td>
<td>K K</td>
<td>K K</td>
<td>K K</td>
<td>K K</td>
<td>K K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>LAR</td>
<td>1 1</td>
<td>1 1</td>
<td>0 0</td>
<td>0 0</td>
<td>AR AR AR M D D D D D D D D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>LARK</td>
<td>1 1</td>
<td>1 1</td>
<td>0 0</td>
<td>1 AR AR AR K K K K K K K K K K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>LARP</td>
<td>1 1</td>
<td>1 1</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>1 1</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>K K K K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>LDP</td>
<td>1 1</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>1 M D D D D D D D D D D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>LDPK</td>
<td>1 1</td>
<td>1 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>K K K K K K K K K K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>LT</td>
<td>1 1</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>1 0</td>
<td>M D D D D D D D D D D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>LTA</td>
<td>1 1</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>1 1</td>
<td>M D D D D D D D D D D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 3.3 Instruction opcodes and operation

#### Table 3.1: List of Instructions and their opcodes

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Cyc</th>
<th>IW</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>D9</th>
<th>D8</th>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30 LTP</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>M</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31 LTS</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>M</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 MAR</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>M</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33 PAC</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>34 ROVM</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>35 SAC</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>M</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36 SAR</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>AR</td>
<td>AR</td>
<td>M</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37 SOVM</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>38 NOP</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>39 ZAC</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>40 ZALH</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>M</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41 ZALS</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>M</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42 APAC</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 3.1: List of Instructions and their opcodes

<table>
<thead>
<tr>
<th>Instr.</th>
<th>#</th>
<th>#</th>
<th>D15</th>
<th>D14</th>
<th>D13</th>
<th>D12</th>
<th>D11</th>
<th>D10</th>
<th>D9</th>
<th>D8</th>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyc</td>
<td>IW</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>-</td>
<td>-</td>
<td>G/L</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPS-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>M</td>
<td>-</td>
<td>-</td>
<td>G/L</td>
<td>0</td>
<td>AR2</td>
<td>AR1</td>
<td>AR0</td>
</tr>
<tr>
<td>-IMD</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>*</td>
<td>*</td>
<td>G/L</td>
<td>0</td>
<td>AR2</td>
<td>AR1</td>
<td>AR0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>M</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>SUBS-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>M</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>-IMD</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>0</td>
<td>AR2</td>
<td>AR1</td>
<td>AR0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>M</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>ADDS-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>M</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>-IMD</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>0</td>
<td>AR2</td>
<td>AR1</td>
<td>AR0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>POP</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSH</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

3.3.2 Description of the operation of each instruction

The operation of all instructions is listed in Table 3.2. It can be observed from the table that each instruction has 4-stages of implementation, these are the four pipeline stages, which are explained in detail in Chapter 4.
### Table 3.2: List of instructions and their operations

<table>
<thead>
<tr>
<th>Sl no.</th>
<th>Instruction</th>
<th>Formula</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MPY</td>
<td>$\text{Treg} \ast [\text{dma}] \rightarrow \text{Preg}$</td>
<td>MPY dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MPY {\ast</td>
</tr>
<tr>
<td>2</td>
<td>MPYK</td>
<td>$\text{Treg} \ast \text{constant} \rightarrow \text{Preg}$</td>
<td>MPYK constant</td>
</tr>
<tr>
<td>3</td>
<td>MAC</td>
<td>$\text{Treg} \ast [\text{dma}] \rightarrow \text{Preg}$</td>
<td>MAC dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{Acc} + \text{Preg} \rightarrow \text{Acc}$</td>
<td>MAC {\ast</td>
</tr>
<tr>
<td>4</td>
<td>OR</td>
<td>$(\text{Acc} \mid [\text{dma}]) \land 0x00000000 \rightarrow \text{Acc}$</td>
<td>OR dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OR {\ast</td>
</tr>
<tr>
<td>5</td>
<td>XOR</td>
<td>$(\text{Acc} ^ \sim [\text{dma}]) \land 0x00000000 \rightarrow \text{Acc}$</td>
<td>XOR dma</td>
</tr>
<tr>
<td>6</td>
<td>SPAC</td>
<td>\text{Acc} - \text{Preg} \rightarrow \text{Acc}</td>
<td>SPAC</td>
</tr>
<tr>
<td>7</td>
<td>SUB</td>
<td>\text{Acc} - [\text{dma}]_{2\text{shift}} \rightarrow \text{Acc}</td>
<td>SUB dma, shift</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SUB {\ast</td>
</tr>
<tr>
<td>8</td>
<td>SUBS</td>
<td>\text{Acc} - [\text{dma}] \rightarrow \text{Acc}</td>
<td>SUBS dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SUBS {\ast</td>
</tr>
<tr>
<td>9</td>
<td>ADD</td>
<td>$\text{Acc} + [\text{dma}]_{2\text{shift}} \rightarrow \text{Acc}$</td>
<td>ADD dma, shift</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ADD {\ast</td>
</tr>
<tr>
<td>10</td>
<td>ADDS</td>
<td>$\text{Acc} + [\text{dma}] \rightarrow \text{Acc}$</td>
<td>ADDS dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ADDS {\ast</td>
</tr>
<tr>
<td>11</td>
<td>AND</td>
<td>$(\text{Acc} &amp; [\text{dma}]) &amp; 0x00000000 \rightarrow \text{Acc}$</td>
<td>AND dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AND {\ast</td>
</tr>
<tr>
<td>12</td>
<td>BU</td>
<td>\text{[pma]} \rightarrow \text{PC}</td>
<td>BU pma</td>
</tr>
<tr>
<td>13</td>
<td>BANZ</td>
<td>(Is AR(ARP) != 0); Yes =&gt; [pma \rightarrow PC]</td>
<td>BANZ pma</td>
</tr>
</tbody>
</table>
### Table 3.2: List of instructions and their operations

<table>
<thead>
<tr>
<th>Sl no.</th>
<th>Instruction</th>
<th>Formula</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>BGEZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No =&gt; [PC + 2 -&gt; PC]</td>
<td>BANZ pma, {*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Is (ACC) &gt;= 0]; Yes =&gt; [pma -&gt; PC]</td>
<td>BGEZ pma</td>
</tr>
<tr>
<td>15</td>
<td>BGZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No =&gt; [PC + 2 -&gt; PC]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Is (ACC) &gt; 0]; Yes =&gt; [pma -&gt; PC]</td>
<td>BGZ pma</td>
</tr>
<tr>
<td>16</td>
<td>BLEZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No =&gt; [PC + 2 -&gt; PC]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Is (ACC) &lt;= 0]; Yes =&gt; [pma -&gt; PC]</td>
<td>BLEZ pma</td>
</tr>
<tr>
<td>17</td>
<td>BLZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No =&gt; [PC + 2 -&gt; PC]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Is (ACC) &lt; 0]; Yes =&gt; [pma -&gt; PC]</td>
<td>BLZ pma</td>
</tr>
<tr>
<td>18</td>
<td>BNZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No =&gt; [PC + 2 -&gt; PC]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Is (ACC) != 0]; Yes =&gt; [pma -&gt; PC]</td>
<td>BNZ pma</td>
</tr>
<tr>
<td>19</td>
<td>BV</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No =&gt; [PC + 2 -&gt; PC]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Is OV == 1]; Yes =&gt; [[pma -&gt; PC] &amp;&amp; [OV -&gt; 0]]</td>
<td>BV pma</td>
</tr>
<tr>
<td>20</td>
<td>BZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No =&gt; [PC + 2 -&gt; PC]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Is ACC == 0]; Yes =&gt; [pma -&gt; PC]</td>
<td>BZ pma</td>
</tr>
<tr>
<td>21</td>
<td>LAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[dma]*2shift -&gt; Acc</td>
<td>LAC dma, shift</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAC {*</td>
</tr>
<tr>
<td>22</td>
<td>LACK</td>
<td>constant -&gt; Acc</td>
<td>LACK 8-bit positive constant</td>
</tr>
<tr>
<td>23</td>
<td>LAR</td>
<td>[dma] -&gt; AR</td>
<td>LAR AR, dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAR AR, {*</td>
</tr>
<tr>
<td>24</td>
<td>LARK</td>
<td>constant -&gt; AR</td>
<td>LARK AR, 8-bit positive constant</td>
</tr>
<tr>
<td>25</td>
<td>LARP</td>
<td>constant -&gt; ARP</td>
<td>LARP 3-bit constant</td>
</tr>
</tbody>
</table>
### Table 3.2: List of instructions and their operations

<table>
<thead>
<tr>
<th>Sl no.</th>
<th>Instruction</th>
<th>Formula</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>LDP</td>
<td>[dma] &amp; 0xff -&gt; data page pointer</td>
<td>LDP dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LDP {&quot;*&quot;</td>
</tr>
<tr>
<td>27</td>
<td>LDPK</td>
<td>constant -&gt; data page pointer</td>
<td>LDPK 8-bit constant</td>
</tr>
<tr>
<td>28</td>
<td>LT</td>
<td>[dma] -&gt; Treg</td>
<td>LT dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LT {&quot;*&quot;</td>
</tr>
<tr>
<td>29</td>
<td>LTA</td>
<td>[dma] -&gt; Treg</td>
<td>LTA dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Acc + Preg -&gt; Acc</td>
<td>LTA {&quot;*&quot;</td>
</tr>
<tr>
<td>30</td>
<td>LTP</td>
<td>[dma] -&gt; Treg</td>
<td>LTP dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preg -&gt; Acc</td>
<td>LTP {&quot;*&quot;</td>
</tr>
<tr>
<td>31</td>
<td>LTS</td>
<td>[dma] -&gt; Treg</td>
<td>LTS dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Acc - Preg -&gt; Acc</td>
<td>LTS {&quot;*&quot;</td>
</tr>
<tr>
<td>32</td>
<td>MAR</td>
<td>Modifies AR(ARP), and ARP as specified</td>
<td>MAR dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MAR {&quot;*&quot;</td>
</tr>
<tr>
<td>33</td>
<td>PAC</td>
<td>Preg -&gt; Acc</td>
<td>PAC</td>
</tr>
<tr>
<td>34</td>
<td>ROVM</td>
<td>0 -&gt; OVM status bit</td>
<td>ROVM</td>
</tr>
<tr>
<td>35</td>
<td>SAC</td>
<td>(Acc) *2 shift -&gt; [dma]</td>
<td>SAC dma, shift</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAC {&quot;*&quot;</td>
</tr>
<tr>
<td>36</td>
<td>SAR</td>
<td>AR -&gt; [dma]</td>
<td>SAR AR, dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SAR AR, {&quot;*&quot;</td>
</tr>
<tr>
<td>37</td>
<td>SOVM</td>
<td>1 -&gt; overflow mode (OVM status bit)</td>
<td>SOVM</td>
</tr>
<tr>
<td>38</td>
<td>NOP</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>39</td>
<td>ZAC</td>
<td>0 -&gt; Acc</td>
<td>ZAC</td>
</tr>
</tbody>
</table>
### Table 3.2: List of instructions and their operations

<table>
<thead>
<tr>
<th>Sl no.</th>
<th>Instruction</th>
<th>Formula</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>ZALH</td>
<td>0 -&gt; Acc[15:0]</td>
<td>ZALH dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[dma] -&gt; Acc[31:16]</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>ZALS</td>
<td>0 -&gt; Acc[31:16]</td>
<td>ZALS dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[dma] -&gt; Acc[15:0]</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>APAC</td>
<td>Acc + Preg -&gt; Acc</td>
<td>APAC</td>
</tr>
<tr>
<td>43</td>
<td>CMPSIMD</td>
<td>Acc[7:0] v/s dma[7:0] -&gt; Acc[7:0]</td>
<td>CMPSIMD dma</td>
</tr>
<tr>
<td>44</td>
<td>SUBSSIMD</td>
<td>Acc[7:0] - (dma[7:0]) -&gt; Acc[7:0]</td>
<td>SUBSIMD dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Acc[15:8] - (dma[15:8]) -&gt; Acc[15:8]</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>ADDSSIMD</td>
<td>Acc[7:0] + (dma[7:0]) -&gt; Acc[7:0]</td>
<td>ADDSIMD dma</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Acc[15:8] + (dma[15:8]) -&gt; Acc[15:8]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Acc[31:24] + (dma[31:24]) -&gt; Acc[31:24]</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>PUSH</td>
<td>Acc -&gt; Stack</td>
<td>PUSH</td>
</tr>
<tr>
<td>47</td>
<td>POP</td>
<td>Stack -&gt; Acc</td>
<td>POP</td>
</tr>
<tr>
<td>48</td>
<td>CALL</td>
<td>PC -&gt; Stack</td>
<td>CALL L2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[pma] -&gt; PC</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>RET</td>
<td>PC -&gt; Restore from Stack</td>
<td>RET</td>
</tr>
</tbody>
</table>
Chapter 4

DSP Pipeline and Read/Write RAM

buffer wrapper implementation

Speed of computation has been the biggest challenge for any digital processor since its invention, measured as the number of instructions that can be executed per second. In the processor world, it has been established from years of experimentation and observation there are only two factors which can significantly increase the speed computing. The first factor being evolution of fabrication technology and the second being better computer architecture, which has famously been marketed by Intel’s tick-tock processor model [21]. Fabrication technology understandably is of enormous complexity, since some of the topics involved are chemical reactions, photonics, material science and device physics.

CPU architecture planning makes a significant impact in its speed enhancement. The key factor in achieving computation speed is concurrency: performing as many operations as possible, simultaneously. Concurrency though has two very important implementations, namely pipelining, and parallelism. Although rooted in the same origins, often hard to distinguish in practice, the two terms are discernibly different in their general approach
Looking at a typical DSP MAC instruction, operands need to be fetched from the memory, multiplied, while the previous product is added to the accumulator and address register is post incremented/decremented. It is obvious that, to accomplish all these sequential functions it would take multiple clock cycles if the DSP is not pipelined [23].

While pipelining effectively speeds up the computation, programmability takes a serious hit, if not done properly. This may also result in loosing some instruction cycles due to data dependency hazards. When a programmer writes an assembly program, it is assumed that every instruction completes before the next instruction begins. This must be ensured by carefully designing the pipeline, such that the DSP should appear as if it were not pipelined even though it is [23].

### 4.1 Pipeline implementation

In a good pipeline design, extensive pipelining with parallel architecture capability has to be implemented, while ensuring programmability is not impacted due to dependency hazards. This requires a system-level understanding of the DSP along with careful planning of each pipeline stage [24].

In chapter 2, the DSP architecture had a brief look at parallelism with SIMD implementation in the ALU. Here, a clear description of pipelining is presented. This chapter explains how exactly each task is split into pieces, while being tackled simultaneously.

#### 4.1.1 Pipeline stages

Before planning the pipeline, it is very important to understand the sequence of events happening in the DSP and data arrival time. Some very important points to remember
4.1 Pipeline implementation

Figure 4.1: Pipeline stages and implementation

are:

1. It takes at least one clock cycle to fetch data from the ROM.

2. The DSP requires one clock cycle to decode the instruction coming from the ROM.

3. Data transfer to and from the RAM also takes at least one clock cycle.

4. Most instructions in the DSP use direct memory access (DMA), hence most of them will have to read data from the RAM.

5. Most instructions are to be executed within a single clock cycle.

Considering all the points mentioned above, the DSP pipeline has been divided into 4 stages. The name and function of each stage is described in Fig. 4.1.

1. The first stage is $FETCH$, where operand is fetched from the ROM. The program counter is updated by this stage, starting as soon as the DSP is switched on and reset.
2. The second stage is \textit{DECODE}, where the fetched instruction is decoded. This stage is responsible for generating the RAM address and corresponding handshake signals, if necessary.

3. The third stage is \textit{WAIT}, where the DSP waits for data to be fetched from the RAM, and feeds the read data into the execution unit. Also, updating AR and ARP is done at this stage.

4. The fourth stage is \textit{EXECUTE}, where all arithmetic operations are performed by the DSP.

Fig. 4.1.1 illustrates how the pipeline works in the DSP for the first 4 instructions. Assuming that the DSP is reset at $T_0$, it is observed that the DSP does not execute the first instruction until $T_4$. However, after $T_4$, for every cycle there will be an output. Hence, technically all single cycle instructions have a latency of 4 cycles, though they just take a single cycle to execute.

\subsection*{4.1.2 Pipeline design for non-branching instructions}

The pipelining of non-branching instruction is straightforward for all read instructions. Write instructions however need to be modified slightly because of the way the RAM memory works and DMA design of the DSP.

Fig. 4.2 illustrates the pipeline operation for DMA read instructions, where all four instructions are assumed to be DMA read instructions. The steps followed at each pipeline stage of the implementation of DMA read instruction are listed below:

1. Fetch stage: The fetch stage here reads the instruction from the ROM and stores it in an internal register for the next stage. It also increments the value of PC by one, so that the next instruction is fetched in the following cycle.
4.1 Pipeline implementation

Figure 4.2: Pipeline example for memory read instructions

2. Decode stage: The decode stage here decodes whether the instruction is direct addressing or indirect addressing. It is also responsible to setup the appropriate handshake signals for memory read and generate read address after decoding the instruction.

3. Wait stage: In the case of direct addressing, the wait stage does nothing. However, this stage takes care of updating AR and ARP registers for indirect addressing.

4. Execute stage: By this stage, the memory read operation would have finished. Hence, the fetched data is now used by the execution unit to compute. By the end of this cycle, the output is either stored in the Product register or the accumulator, depending on the instruction executed.

Fig. 4.3 illustrates the pipeline operation for 4 instructions, where the second and third
are DMA write instructions, while the first and fourth are DMA read instructions. The steps followed at each pipeline stage of the implementation of DMA write instruction are listed below:

1. Fetch stage: This stage is the exact same as DMA read fetch stage. Hence, the instruction is read from the ROM and passed on to the decode stage, while incrementing the value of PC by one for fetching the next instruction.

2. Decode stage: The instruction is decoded, setting up the write address according to direct/indirect addressing. Appropriate handshake signals for memory write are generated.

3. Wait stage: By the time this stage is completed, the appropriate write data needs to be ready. To accomplish this, appropriate changes in the architecture have been made at this stage to shift the output of the results of the fourth cycle of the previous instruction in the case of SAC or store accumulator instruction, in case the previous instruction is an ALU operation. The updating of AR and ARP for indirect addressing also is the responsibility of this stage.

4. Execute stage: During this stage, the memory write operation is performed by the read/write RAM buffer wrapper.

### 4.1.3 Pipeline design for unconditional branching instructions

As observed from Table 3.1, branching instructions are two-cycle instructions, and all except return or RET require two-instruction words. From Chapter 3, it is also established that the first instruction word contains the instruction op-code, while the second contains the jump or branch address.
Figure 4.3: Pipeline example for memory write instructions
During the execution of two-word branch instruction, all stages of the pipeline are stalled while reading the second word, since it is not an instruction. Appropriate changes are made at every pipeline stage to make sure that the second word is not read as an instruction, but stored as jump address.

There are two types of branching instructions, namely conditional and unconditional branching instructions. Unconditional branching instructions are BU (branch unconditional), CALL (call) and RET (return), where branch must always be taken. Conditional branching instructions are where the branching decision is made based on the valuation of a condition.

Figure 4.4 shows a pipeline implementation example of unconditional branch instruction. The steps followed at each pipeline stage of the implementation of unconditional branch instructions are listed below:

1. Fetch stage: The DSP during this stage reads the instruction from the ROM and increments PC by one, just like all other instructions. However, the fetch is stalled in order to read the jump or branch address.

2. Decode stage: The instruction is decoded, and call registers are accordingly modified for call and return instructions, while the jump address is read from the program memory and fed to the PC.

3. Wait stage: As the unconditional instruction has already been executed at this point, this stage does almost nothing. However, for call and return instructions, the stack pointer is stored and restored respectively at this stage.

4. Execute stage: This stage does performs no task since the instruction has already accomplished its purpose.
4.1 Pipeline implementation

4.1.4 Pipeline design for conditional branching instructions

Before looking at the implementation of conditional branching, it is necessary to understand how the instruction works in practice. Since almost all conditional branching instructions rely either on status flags resulting from an ALU operation, or the ALU result itself, timing and data dependency wise, the worst-case scenario of the previous instruction being an ALU operation is assumed before approaching to design the pipeline stages.

With the assumption that the previous instruction is an ALU operation, the outcome of the branching condition is not known until the operation is complete. From the pipeline design for ALU operations, or DMA read operations, discussed in Section 4.1.2, it is clear that execution happens only at the last pipeline stage. Hence, the branching decision cannot be made until after the fourth cycle of the previous instruction has been executed. However, after stalling the pipeline to read the branch address, before the third stage of the pipeline of the conditional branch instruction, the DSP should already know where to
fetch the next instruction from. In other words, during the second pipeline stage, the PC needs to be updated to the next program address to fetch from. This results in a dilemma, as the branching decision needs to be made at the second stage of the pipeline, however the decision is not available until the fourth stage.

There are two solutions to this problem:

Solution 1: Decide to not take the branch, and make necessary changes if the condition turns out to be true.

Solution 2: Predict the branch, and pay the penalty of two cycles if wrong by making necessary changes in the case of a wrong prediction.

Though solution 2 is a better option and has numerous methods of execution, even the simplest branch predictor requires a lot of additional hardware and planning. To keep the DSP design simple, solution 1 is considering in this design.

Figures 4.5 and 4.6 illustrate both cases of the working of pipeline for conditional branch instructions, the first case where the condition evaluates to be false, and the second case where the condition evaluates to be true. In both cases, the first instruction is assumed to be the evaluation instruction, hence the branch/jump condition is evaluated based on its outcome. The second instruction is the conditional branch instruction, while the last two instructions are unconditional instructions. The pipeline plan of action for each stage is listed below:

1. Fetch stage: The DSP during this stage reads the instruction from the ROM and increments PC by one, just like all other instructions. Fetching of the next instruction is stalled to read the jump or branch address.

2. Decode stage: The instruction is decoded, and the jump address is read from the program memory and stored until execution stage. However, the value of PC is
4.1 Pipeline implementation

Figure 4.5: Pipeline implementation example for conditional branch instruction, when condition is false

incremented by one for the pipeline to work smoothly, and not waste any cycles in case the branch evaluates to be false.

3. Wait stage: This stage does perform no task.

4. Execute stage: The branching condition is evaluated at this stage. In case the condition evaluates false, no change is made to the pipeline flow. However, if the condition evaluates to be true, the value of PC is updated to the jump address, resulting in the wastage of the computations in the previous two cycles. It is very important to undo any modifications done in the previous two cycles and also stall the pipeline accordingly, to make sure no unwanted data is propagated, in case the jump is taken.
4.2 Read/write RAM buffer wrapper

DSPs have much higher memory bandwidth and use lot more memory-to-memory instructions, when compared to traditional processors [25]. While most DSPs tackle this problem using small, fast and simple parallel memory banks, it is very difficult to design compilers and the power consumption increases significantly for such DSPs [18]. Since it has been established that data memory access is very important in DSPs, it is crucial to ensure that memory access is quick and effective, while keeping the power consumption low. Hence, both, data and address memories have been clocked at the same speed as the DSP, in an effort to keep the total power consumption low.

Taking a brief look at the pipeline implementation described in the Section 4.1.2, it can be observed that there exists a huge problem in the case of RAM memory writes, since the write data is provided a cycle after write address generation. Since the pipelining and data memory addressing of the DSP design implemented in this paper is very different from
TMS32010, even though the ISA is almost the same, this problem is not observed in the case of TMS32010. This is mainly because TMS32010 had its memory clocked to at least twice the speed of the DSP itself. This is evident from some of the instructions in its ISA, which have obviously not been implemented in this DSP. A good example for this is the LTA instruction, which featured multiple memory transactions within a single clock cycle.

Section 4.2.1 describes what problems were faced due to clocking the memory at the same speed as the processor, and Section 4.2.2 describes how the problem has been resolved using the read/write RAM buffer wrapper.

4.2.1 RAM read/write problem description

Looking at the pipeline implementation in the case of data memory or DMA write operations in Section 4.1.2, it is observed that write address and handshaking signal generation happens at stage 2 or decode stage, while the write data is sent to the data memory in the next stage, which is stage 3 or the wait stage. However, the RAM requires the address, handshaking signals, along with the data to be written, all within the same cycle. This is not possible with the pipeline design implemented in this paper, since the write data may be computed in last stage of the pipeline of the previous instruction.

Hence, since is in not possible to make sure that the RAM receives the write data at the correct cycle, a buffer layer has been designed to effectively facilitate the data-flow. The buffer layer is simple in design and implementation, using minimal hardware required to serve the purpose, since other easy solutions involve using a faster clock for the memory, leading to an increase in power consumption.
4.2 Read/write RAM buffer wrapper

4.2.2 Design and implementation of read/write buffer wrapper

The wrapper is designed with a simple goal: delay the data memory write operation by a single cycle, while seamlessly providing the correct data whenever necessary. Translating this to a plan of action, the following procedures were followed:

1. For every write operation, store the address and corresponding data.

2. For every read, check the address. If it matches the buffer address, transfer the contents of the buffer data as the output to the DSP. Else, make the necessary arrangements to fetch the data directly from the RAM, and send it to the DSP.

3. For every other write operation following the first, store the buffer data onto RAM and update the buffer address and data with the corresponding new values.

Fig. 4.7 illustrates the state-machine for the read/write buffer wrapper. The state machine consists of a total of four states depending on the type of operation involved. Since write is implemented as a two-stage operation in the pipeline, the following instruction also needs to be accounted for within the read/write buffer wrapper. A brief explanation of the implementation is described below, detailing the operation of each state:
1. **Read state:** Read state is also the idle state. If the read address matches the buffer address register contents, the buffer data is transferred to the DSP. However, if the read address is different from the buffer address register contents, the required data is fetched from RAM and transferred to the DSP within the next clock cycle.

2. **Write state:** A single bit flag is used to keep track of whether the buffer data has been transferred onto the RAM or not. Every time a new data arrives, if data is present in the data buffer register, it is transferred to the RAM address corresponding to the buffer address, which is retrieved from the buffer address register. This is followed by storing the write address in the buffer address register.

3. **RAW state:** In the RAW state or read-after-write state, the write data is stored in the buffer data register. Also, all functions in the read state are performed here as well.

4. **WAW state:** In the WAW state or the write-after-write state, the write data is directly sent to the RAM, at the address location corresponding to the buffer address register. The new write address is now stored in the buffer address register.
Chapter 5

Median filter design

Image processing and filtering is an area where DSPs have been used extensively since their invention. In the recent years however, more complex image processing have been handled by GPUs or graphical processing units mainly due to their hardware parallelism and enormous amount of data required to be processed. However, numerous image filtering applications are still use DSPs, but with multiprocessor type configuration.

Taking a brief look at image data, it is usually represented by the amount of Red, Green and Blue (RGB) colors over a fixed area of a preset number of very small points called pixels. The common representation of a standard dimension image is 24-bit RGB values per pixel, over an area of (720 x 576) pixels. Most simple DSPs are 16-bit fixed-point architectures, hence to handle image data they would require two data words per pixel. Expanding the data word to at least 24-bits hence could result in further applications in image handling and processing.

The following sections of this chapter present a simple application of the designed DSP, to showcase the merits of its enhancements over the TMS32010 by implementing a median filter. Section 5.1 presents an overview of the median filter by explaining how a median filter
works. Section 5.2 discusses the median filter algorithm design and its implementation.

5.1 Median filter overview

Median filters are non-linear digital filters used widely to get rid of salt and pepper noise. The implementation of the median is quite simple and straightforward. Considering a 3x3 window of pixels of an image, the following steps are followed to find the median:

*Step 1*: Arrange the pixels one after the other.

*Step 2*: Rearrange the pixels in an ascending or descending order.

*Step 3*: Pick the central value of the arranged pixels, which is the fifth pixel in this case. This will be the median.

While the median filter implementation looks like a simple two-step process, it takes a significant amount of effort to arrange the pixels in ascending or descending order, as every pixel needs to be compared to every other pixel, and this needs to be done sequentially to keep track of the order of their arrangement.

Fig.5.1 illustrates the working of a median filter. In the figure, P1, P2, P3, P4, P5, P6, P7, P8 and P9 are pixel values of the 3x3 window from the image. After step 2, note that the new pixel values P1’, P2’, P3’, P4’, P5’, P6’, P7’, P8’ and P9’ indicated in the figure represent the rearranged pixel values.

5.2 Median filter design and implementation

The median filter algorithm design for the 3 × 3 pixel window is explained in figure 5.2. The figure self-explanatory and hence clearly explains the algorithm which has been implemented in DSP assembly language.
5.2 Median filter design and implementation

Figure 5.1: Median filter working illustration

Figure 5.2: Median filter algorithm
The implementation of this algorithm on an image is done by moving the $3 \times 3$ window from the top-left corner of the image across all columns, and soon as the median for the first row has been computed, the window is moved to the next row. This is repeated until the last row is computed. Fig. 5.3 illustrates the implementation of algorithm on an image. The first part of the figure shows the $3 \times 3$ window placement while computing the first median, the second part shows the window placement while computing the second median and the third part shows the window placement while computing the median for the second row. This window placement pattern is repeated until all the medians are computed. It is worth noting that the output image will lose two rows and two columns, with this implementation.
5.2 Median filter design and implementation

1. **Computing the first median**

![Diagram showing the first median computation for a 3x3 window.]

2. **After computing the first median**

![Diagram showing the window moved one step to the right.]

3. **After computing medians for the first row**

![Diagram showing the medians computed for the first row.]

Figure 5.3: Median filter algorithm implementation illustration for a 3 x 3 window
Chapter 6

Results

This chapter discusses the results from this project, as well as future work that could be completed.

6.1 Results

The DSP design was synthesized using Synopsys Design Compiler at 180 nm technology nodes from TMSC. Cadence Virtuoso Suite was used for design, debugging and simulation of the design. Table 6.1 gives the synthesis results for the post-scan netlist of the design, when synthesized at 50MHz.

Due to time constraints, it was not possible to fully verify the DSP design. The DSP however has been verified to work at gate level, where most of its instructions and numerous branching dependencies have been tested. The basic median filter algorithm was designed in assembly language and verified to work on the DSP. Details including the Assembly code that was used for testing the DSP have been included in Appendix A.
### Table 6.1: Synthesis results

<table>
<thead>
<tr>
<th><strong>Area</strong> $(\mu m^2)$</th>
<th>Noncombinational area</th>
<th>181298</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Combinational area</td>
<td>167021</td>
</tr>
<tr>
<td></td>
<td>Buf/ Inv area</td>
<td>9027</td>
</tr>
<tr>
<td></td>
<td>Total cell area</td>
<td>348320</td>
</tr>
<tr>
<td><strong>Power</strong> $(mW)$</td>
<td>Internal Power</td>
<td>9.4111</td>
</tr>
<tr>
<td></td>
<td>Switching Power</td>
<td>1.6481</td>
</tr>
<tr>
<td></td>
<td>Leakage Power</td>
<td>1.4210</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>11.0607</td>
</tr>
<tr>
<td><strong>Timing</strong> $(ns)$</td>
<td>Data arrival time</td>
<td>18.1474</td>
</tr>
<tr>
<td></td>
<td>Slack</td>
<td>1.4799</td>
</tr>
<tr>
<td><strong>DFT Coverage</strong></td>
<td>Test coverage</td>
<td>99.92%</td>
</tr>
</tbody>
</table>

---
Chapter 7

Conclusions and future work

7.1 Conclusion

The DSP design has been successfully implemented, verified for the instructions mentioned in Appendix A and synthesized at 50MHz. The design was kept simple, since its ISA and architecture have been based on the TMS32010. Power efficiency was achieved by running the memory at the same speed as the DSP. A median filter algorithm was designed in assembly, simulated at gate-level and verified to work on the DSP within 100 instructions, demonstrating that the enhanced SIMD instructions could be used for median filter computation, hence proving that the DSP is capable of handling simple multimedia applications.

7.2 Future work

Since the DSP was designed in a very short span of time, testing the DSP thoroughly could not be completed. It is necessary to completely test the DSP before attempting to use it in an application, hence this would be the first thing to work on. The median
filter algorithm described in chapter 5, though successfully implemented, could not be tested with a noisy image due to time constraints. Doing this would demonstrate the capabilities of the DSP, and a comparison of the results with a similar implementation on the TMS32010 would prove the claims presented in the paper.

Designing a compiler would certainly be necessary and the next step to work on. Another interesting enhancement would be the design of a parallel-processor system using multiple DSPs for advanced imaging applications.
References


Appendix I

Source Code

I.1 RTL source code

I.1.1 DSP top level module

```vhdl
// Author : Shashank Simha
// Date : 12/12/2017
// University : Rochester Institute of Technology
// Description : This is a part of the DSP implemented for grad project

module DSP_Version1 (
    reset,
    clk,
    scan_in0,
    scan_en,
    test_mode,
    scan_out0,
    SW_pin, Display_pin,
    DM_out, CEN, wr_data, DM_Addr, DM_in, OEN,  //ram_buffer
    PC, PM_out  //rom
);
```
input [4:0] SW_pin;       // Four switches and one push–button
output [7:0] Display_pin; // 8 LEDs

input
reset,    // system reset
clk;      // system clock

input
scan_in0,   // test scan mode data input
scan_en,    // test scan mode enable
test_mode;  // test mode select

output
scan_out0;  // test scan mode data output

Ram ports
output[14:0] DM_Addr;
output[31:0] DM_in;
input[31:0] DM_out;

output reg wr_data, OEN, CEN;

Rom ports
input[15:0] PM_out;
output reg [15:0] PC;

Ram ports

--- 1 ISA Parameters

parameter [7:0] MPY = 8'h00000000; //1.
parameter [7:0] MPYK = 8'h10100000; //2.
parameter [7:0] MAC = 8'h02; //3.
parameter [7:0] OR = 8'h03; //4.
parameter [7:0] XOR = 8'h04; //5.
parameter [15:0] SPAC = 16'h0100; //6.
parameter [2:0] SUB = 3'h1; //7.
parameter [7:0] SUBS = 8'h05; //8.
parameter [2:0] ADD = 3'h2; //9.
parameter [7:0] ADDS = 8'h06; //10.
parameter [7:0] AND = 8'h07; //11.
parameter [15:0] BU = 16'h010F; //12.
parameter [15:0] BGEZ = 16'h0101; //14.
parameter [15:0] BGZ = 16'h0102; //15.
parameter [15:0] BLEZ = 16'h0103; //16.
parameter [15:0] BLZ = 16'h0104; //17.
parameter [15:0] BNZ = 16'h0105; //18.
parameter [15:0] BV = 16'h0106; //19.
parameter [15:0] BZ = 16'h0107; //20.
parameter [2:0] LAC = 3'h3; //21.
69 parameter [7:0] LACK = 8'b10100001; //22.
70 parameter [7:0] LAR = 5'b11000; //23.
71 parameter [7:0] LARK = 5'b11001; //24.
72 parameter [7:0] LARKH = 5'b11011; //25.
73 parameter [7:0] LARP = 8'b10100011; //26.
74 parameter [7:0] LDP = 8'h09; //27.
75 parameter [7:0] LDPK = 8'b10100100; //28.
76 parameter [7:0] LT = 8'h0a; //29.
77 parameter [7:0] LTA = 8'h0b; //30.
78 parameter [7:0] LTD = 8'h0c; //31.
79 parameter [7:0] LTP = 8'h0d; //32.
80 parameter [7:0] LTS = 8'h0e; //33.
81 parameter [7:0] MAR = 8'h0f; //34.
82 parameter [15:0] PAC = 16'h011F; //35.
83 parameter [15:0] ROVM = 16'h012F; //36.
84 parameter [2:0] SAC = 3'h4; //37.
85 parameter [7:0] SAR = 5'b11010; //38.
86 parameter [15:0] SOVM = 16'h013F; //39.
87 parameter [7:0] TBLR = 8'h11; //40.
88 parameter [7:0] TBLW = 8'h12; //41.
89 parameter [15:0] NOP = 16'h014F; //42.
90 parameter [15:0] ZAC = 16'h015F; //43.
91 parameter [7:0] ZALH = 8'h13; //44.
92 parameter [7:0] ZALS = 8'h14; //45.
93 parameter [15:0] APAC = 16'h016F; //46.
94 parameter [6:0] CMPSIMD = 7'b0001101; //47.
95 parameter [7:0] SUBSIMD = 8'h16; //48.
96 parameter [7:0] ADDSIMD = 8'h17; //49.
97 parameter [8:0] BANZ = 8'h18; //50.
98 parameter [15:0] PUSH = 16'h018F; //51.
99 parameter [15:0] POP = 16'h017F; //52.
100 parameter [15:0] CALL = 16'h01AF; //53.
101 parameter [15:0] RET = 16'h019F; //54.

// 2 Internal registers (& wires)
//
reg [15:0] AR [7:0]; // 7 Auxiliary Registers of width 16−bits each
reg [2:0] ARP, PARP; // 2 Registers to store current and previous AR pointers

reg [7:0] DPPTR;

reg [31:0] acc, Preg;
reg [15:0] Treg;
wire [15:0] SR_wire;
reg [15:0] SR; // Status register (4*CNVZ) for SIMD instructions
I.1 RTL source code

117    reg [4:0] SP;       // Stack pointer
118    reg [4:0] CSP;     // Call stack pointer;
119    //———————————————————
120    //— 3 Pipeline registers
121    //———————————————————
122    reg [4:0] sreg2, sreg3, sreg4;   // Used to temporarily Shift value between pipeline stages
123    reg [15:0] JAddr2, JAddr3, JAddr4, JAddr; // Used to temporarily store Jump Address between pipeline stages
124    reg [31:0] temp_acc;
125    reg [31:0] breg;
126    reg [15:0] PAR [7:0];  // 7 Auxiliary Registers of width 16−bits each
127    reg cnt;               //
128    reg JFlag, JFlag_del, JFlag_c, JFlag_uc;
129    reg stall_mc1, stall_mc2, stall_mc3, stall_mc4, stall_uc; //
130    reg [15:0] IR2, IR3, IR4, IR_del;
131    reg J_detect;
132    reg [15:0] DM_Addr_reg;
133    //———————————————————
134    //— 4 Memory clock setup
135    //———————————————————
136    // assign clk_n <=~clk;
137    reg get_DMAddr;
138
139    reg [31:0] stack [31:0];  // 32 stack registers
140    reg [15:0] call_stack [31:0]; // 32 call stack registers
141    reg [15:0] call_SR [31:0];  // 32 call SR registers
142
143    reg [15:0] alu_opcode, mpy_opcode, next_opcode;
144    //———————————————————
145    //                          
146    //———————————————————
147    reg DM_cnt;
148
149    wire [31:0] s1_in;
150    wire [31:0] s1_out;
151    wire [31:0] alu_out;
152    wire [15:0] mult_2;
153    wire [31:0] result;
154    wire [31:0] Preg_wire;
155    reg [31:0] s1_in_reg, buff;
156    reg [15:0] buff_mult_2;
157    reg updated_AR;
wire branch_predict;
reg check_condition;

assign branch_predict = (IR4[15:0]==BZ) ? ((acc==0)? 1 : 0) :
                 (IR4[15:0]==BV) ? ((SR[15]==0)? 1 : 0) :
                 (IR4[15:0]==BNZ) ? ((acc!=0)? 1 : 0) :
                 (IR4[15:0]==BLZ) ? ((acc< 0)? 1 : 0) :
                 (IR4[15:0]==BLEZ) ? ((acc<=0)? 1 : 0) :
                 (IR4[15:0]==BGEZ) ? ((acc >=0)? 1 : 0) :
                 (IR4[15:0]==BGZ) ? ((acc> 0)? 1 : 0) :
                 ((IR4[15:8]==BANZ) && (IR4[6:0]==0)) ? ((AR[ARP]=0)? 1 : 0) :
                 0;

                 (updated_AR)? AR[ARP][14:0] :
                 DM_Addr_reg;

assign mult_2= ((IR4[15:8]==MAC) || (IR4[15:8]==MPY)) ? buff_mult_2 :
                (IR4[15:8]==MPYK) ? {8'd0 , IR4[7:0]} :
                0;

assign DM_in = (IR4[15:13] == SAC) ?
                 (IR4[15:11] == SAR) :
                 AR[IR4[10:8]] :
                 32'h0;

                s1_in_reg :
                0;

//-- 5 Instantiation of components
multiplier ml (.scan_in0 (scan_in0),
               .scan_out0 (scan_out0),
               .scan_en  (scan_en),
               .test_mode (test_mode),
               .a       (Treg),
               .d       (d)
RTL source code

// Shifts input operand
shifter_input s1 (.scan_in0 (scan_in0),
   .scan_out0 (scan_out0),
   .scan_en (scan_en),
   .test_mode (test_mode),
   .shift_in (s1_in),
   .opcode (alu_opcode),
   .shift_out (s1_out));

ALU alu1 (.scan_in0 (scan_in0),
   .scan_out0 (scan_out0),
   .scan_en (scan_en),
   .test_mode (test_mode),
   //
   .a (acc),
   .b (s1_out),
   .opcode (alu_opcode),
   .result (alu_out),
   .carry (SR_wire[15]),
   .negative (SR_wire[14]),
   .ov (SR_wire[13]),
   .zero (SR_wire[12]),
   .carry_2 (SR_wire[11]),
   .negative_2 (SR_wire[10]),
   .ov_2 (SR_wire[9]),
   .zero_2 (SR_wire[8]),
   .carry_3 (SR_wire[7]),
   .negative_3 (SR_wire[6]),
   .ov_3 (SR_wire[5]),
   .zero_3 (SR_wire[4]),
   .carry_4 (SR_wire[3]),
   .negative_4 (SR_wire[2]),
   .ov_4 (SR_wire[1]),
   .zero_4 (SR_wire[0]);

// Shifts output operand
shifter_output s2 (.scan_in0 (scan_in0),
   .scan_out0 (scan_out0),
   .scan_en (scan_en),
   .test_mode (test_mode),
   .shift_in (alu_out),
   .opcode (next_opcode),
   .shift_out (result));

//
always@ (posedge clk or posedge reset)
begin
  if(reset)
  begin
    PC <= 16'h0;
    AR[0] <= 16'h0; AR[1] <= 16'h0; AR[2] <= 16'h0; AR[3] <= 16'h0;
    stall_mc1 <= 0; stall_mc2 <= 1; stall_mc3 <= 1; stall_mc4 <= 1;
    IR2 <= 16'h0; IR3 <= 16'h0; IR4 <= 16'h0; DPPTR <= 8'd0;
    wr_data <= 1'b1; //Read mode
    ARP <= 3'd0;
    SP <= 5'd0;
    CSP <= 5'd0;
    CEN <= 1'b0; OEN <= 1'b0;
    JFlag <= 1'b0; JFlag_uc <= 1'b0; JFlag_c <= 1'b0;
    Treg <= 16'h0;
    Preg <= 32'h0;
    acc <= 32'h0;
    breg <= 32'h0;
    alu_opcode <= 16'h0;
    mpy_opcode <= 16'h0;
    next_opcode <= 16'h0;
    DM_Addr_reg <= 15'h0;
  end
  //////////////////////////////////////////////////////////////////////////////////
  updated_AR <= 0;
  //////
  buff <= 32'h0;
  buff_mult_2 <= 32'h0;
  stall_uc <= 1'b0;
  cnt <= 0;
  JAddr <= 16'h0; JAddr2 <= 16'h0; JAddr3 <= 16'h0; JAddr4 <= 16'h0;
  //////
  stack[0] <= 32'h0; stack[1] <= 32'h0; stack[2] <= 32'h0; stack[3]
  <= 32'h0;
  <= 32'h0;
  stack[8] <= 32'h0; stack[9] <= 32'h0; stack[10] <= 32'h0; stack
  [11] <= 32'h0;
  stack[12] <= 32'h0; stack[13] <= 32'h0; stack[14] <= 32'h0; stack
  [15] <= 32'h0;
  stack[16] <= 32'h0; stack[17] <= 32'h0; stack[18] <= 32'h0; stack
  [19] <= 32'h0;
  stack[20] <= 32'h0; stack[21] <= 32'h0; stack[22] <= 32'h0; stack
  [23] <= 32'h0;
  stack[24] <= 32'h0; stack[25] <= 32'h0; stack[26] <= 32'h0; stack
  [27] <= 32'h0;
I.1 RTL source code

stack[28] <=32'h0; stack[29] <=32'h0; stack[30] <=32'h0; stack[31] <=32'h0;

////////
call_stack[0] <=16'h0; call_stack[1] <=16'h0; call_stack[2]
<==16'h0; call_stack[3] <=16'h0;
<==16'h0; call_stack[7] <=16'h0;
call_stack[8] <=16'h0; call_stack[9] <=16'h0; call_stack[10]
<==16'h0; call_stack[11] <=16'h0;
call_stack[12] <=16'h0; call_stack[13] <=16'h0; call_stack[14]
<==16'h0; call_stack[15] <=16'h0;
call_stack[16] <=16'h0; call_stack[17] <=16'h0; call_stack[18]
<==16'h0; call_stack[19] <=16'h0;
call_stack[20] <=16'h0; call_stack[21] <=16'h0; call_stack[22]
<==16'h0; call_stack[23] <=16'h0;
call_stack[24] <=16'h0; call_stack[25] <=16'h0; call_stack[26]
<==16'h0; call_stack[27] <=16'h0;
call_stack[28] <=16'h0; call_stack[29] <=16'h0; call_stack[30]
<==16'h0; call_stack[31] <=16'h0;

////////
call_SR[0] <=16'h0; call_SR[1] <=16'h0; call_SR[2] <=16'h0;
call_SR[3] <=16'h0;
call_SR[7] <=16'h0;
call_SR[8] <=16'h0; call_SR[9] <=16'h0; call_SR[10] <=16'h0;
call_SR[11] <=16'h0;
call_SR[15] <=16'h0;
call_SR[16] <=16'h0; call_SR[17] <=16'h0; call_SR[18] <=16'h0;
call_SR[19] <=16'h0;
call_SR[20] <=16'h0; call_SR[21] <=16'h0; call_SR[22] <=16'h0;
call_SR[23] <=16'h0;
call_SR[24] <=16'h0; call_SR[25] <=16'h0; call_SR[26] <=16'h0;
call_SR[27] <=16'h0;
call_SR[28] <=16'h0; call_SR[29] <=16'h0; call_SR[30] <=16'h0;
call_SR[31] <=16'h0;

end
else
begin
DM_Addr_reg <=DM_Addr;

// Fetch Data memory Address in Fetch stage or pipeline stage 2
I.1 RTL source code


6.1 Pipeline stage 4

333 //― 6.1 Pipeline stage 4

334 if (stall\_mc4 == 0) begin

335 \text{case (IR4[15:13])}

336 \text{ADD, SUB, LAC: begin}

337 \text{acc<= result;}

338 \text{SR <= SR\_wire; end case}

339 \text{case (IR4[15:11])}

340 \text{LAR: AR[IR4[10:8]] <=buff[15:0]; end case}

341 \text{case (IR4[15:9])}

342 \text{CMPSIMD: begin}

343 \text{SR<=SR\_wire;}

344 \text{acc<=result; end}

345 \text{endcase}

346 \text{case (IR4[15:8])}

347 \text{ADDS, SUBS, LTA, SUBS, LTS, SUBSIMD, ADDSIMD, AND, OR, XOR: begin}

348 \text{SR<=SR\_wire;}

349 \text{acc<=result; end}

350 \text{end}

351 \text{MAC: begin}

352 \text{Preg <= Preg\_wire;}

353 \text{SR<=SR\_wire;}

354 \text{acc<=result; end}

355 \text{end}

356 \text{MPY, MPYK: begin}
Preg <= Preg_wire;
end

LACK: begin
  acc <= IR4[7:0];
end

LT: begin
  if (SR[13] == 0) Treg[15:0] <= buff[15:0];  // if overflow is reset, then number is considered positive
  else begin
    // if overflow is set, then number is considered negative
    Treg[15] <= buff[31];
    Treg[14:0] <= buff[14:0];
  end
end

LTA, LTS: begin
  if (SR[13] == 0) Treg[15:0] <= buff[15:0];  // if overflow is reset, then number is considered positive
  else begin
    // if overflow is set, then number is considered negative
    Treg[15] <= buff[31];
    Treg[14:0] <= buff[14:0];
  end
end

LTP: begin
  acc <= Preg;
  if (SR[13] == 0) Treg[15:0] <= buff[15:0];  // if overflow is reset, then number is considered positive
  else begin
    // if overflow is set, then number is considered negative
    Treg[15] <= buff[31];
    Treg[14:0] <= buff[14:0];
  end
I.1 RTL source code

387 end
388 ZALH: begin
389 acc <= {16'd0, DM_out[15:0]};
390 end
391 ZALS: begin
392 acc <= {DM_out[15:0], 16'd0};
393 end
394 BANZ: begin
395 if (AR[ARP] == 0) JFlag <= 0;
396 else begin
397 JFlag <= 1'b1;
398 JFlag_c <= 1'b1;
399 JAddr <= JAddr4;
400 end
401 end
402 LARP: ARP <= IR4[2:0];
403 LDPK: DPPT <= IR4[7:0];
404 LDF: DPPT <= DM_out[7:0];
405 endcase
406 case (IR4[15:0])
407 APAC, SPAC: begin
408 acc <= result;
409 SR <= SR_wire;
410 end
411 BGEZ: begin
412 if (acc >= 0) begin
413 JFlag_c <= 1'b1;
414 JAddr <= JAddr4;
415 end
416 end
417 BGZ: begin
418 if (acc > 0) begin
419 JFlag_c <= 1'b1;
420 JAddr <= JAddr4;
421 end
422 end
423 BLEZ: begin
424 if (acc <= 0) begin
425 JAddr <= JAddr4;
426 JFlag_c <= 1'b1;
427 end
428 end
429 BLZ: begin
430 if (acc < 0) begin
431 JAddr <= JAddr4;
432 JFlag_c <= 1'b1;
BNZ: begin
    if (acc != 0) begin
        JAddr <= JAddr4;
        JFlag_c <= 1'b1;
    end
end

BV: begin
    if (SR[13] == 1) begin
        SR[13] <= 0;
        JFlag_c <= 1'b1;
        JAddr <= JAddr4;
    end
end

BZ: begin
    if (acc == 0) begin
        JFlag_c <= 1'b1;
        JAddr <= JAddr4;
    end
end

ROVM: SR[13] <= 0;
ZAC: acc <= 32'h0;
PAC: acc <= Preg;
PUSH: begin
    stack[SP] <= acc;
    SP <= SP + 1'b1;
end

POP: begin
    acc <= stack[SP-1'b1];
    SP <= SP - 1'b1;
end

CALL: begin
    call_SR[CSP] <= SR;
    CSP <= CSP + 1'b1;
end

RET: begin
    SR <= call_SR[CSP];
end

//--- 6.2 Pipeline stage 3
if (stall_mc3 == 0)
begin
    case (IR3[15:0])
APAC, SPAC: begin
    alu_opcode <= IR3;
end

s1_in_reg <= Preg;
endcase
end

CMPSimd: begin
    alu_opcode <= IR3;
s1_in_reg <= DM_out;
endcase

if (IR3[7] == 1) begin
    PAR[ARP] <= AR[ARP];
    PARP <= ARP;
    updated_AR <= 1;
    if (JFlag_c == 1)
        begin
            ARP <= PARP;
            AR[PARP] <= PAR[PARP - 1];
        end
    else begin
        case ({IR3[6], IR3[5]})
            2'b00: AR[ARP] <= AR[ARP];
            2'b01: AR[ARP] <= AR[ARP] + 1;
            2'b10: AR[ARP] <= AR[ARP] - 1;
        endcase
        end
end

end

endcase
end

OR, XOR, SUBS, ADDS, AND, SUBSIMD, ADDSIMD: begin
    alu_opcode <= IR3;
end
s1_in_reg <= DM_out;
if (IR3[7] == 1) begin
    PAR[ARP] <= AR[ARP];
    PARP <= ARP;
    updated_AR <= 1;
    if (JFlag_c == 1)
        begin
            ARP <= PARP;
            AR[PARP] <= PAR[PARP];
        end
    else begin
        case ({IR3[6], IR3[5]})
            2'b00 : AR[ARP] <= AR[ARP];
            2'b01 : AR[ARP] <= AR[ARP] + 1;
            2'b10 : AR[ARP] <= AR[ARP] - 1;
        endcase
        ARP <= IR3[2:0];
    end
end
end
LTA, LTS: begin
    buff <= DM_out;
    alu_opcode <= IR3;
    s1_in_reg <= Preg;
    if (IR3[7] == 1) begin
        PAR[ARP] <= AR[ARP];
        PARP <= ARP;
        updated_AR <= 1;
        if (JFlag_c == 1)
            begin
                ARP <= PARP;
                AR[PARP] <= PAR[PARP];
            end
    end
else
    begin
        case ({IR3[6], IR3[5]})
        2'b00: AR[ARP] <= AR[ARP];
        2'b01: AR[ARP] <= AR[ARP] + 1;
        2'b10: AR[ARP] <= AR[ARP] - 1;
        endcase
    ARP <= IR3[2:0];
end
end
LTP, MAR, ZALH, ZALS, LT, LDP: begin
    buff <= DM_out;
    if (IR3[7] == 1) begin
        PAR[ARP] <= AR[ARP];
        PARP <= ARP;
        updated_AR <= 1;
        if (JFlag_c == 1)
            begin
                ARP <= PARP;
                AR[PARP] <= PAR[PARP];
            end
    end
    else
        begin
            case ({IR3[6], IR3[5]})
            2'b00: AR[ARP] <= AR[ARP];
            2'b01: AR[ARP] <= AR[ARP] + 1;
            2'b10: AR[ARP] <= AR[ARP] - 1;
            endcase
        ARP <= IR3[2:0];
I.1 RTL source code

570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603

MPY: begin
    mpy_opcode <= IR3;
    buff_mult_2 <= DM_out;
    if (IR3[7] == 1) begin
        PAR[ARP] <= AR[ARP];
        PARP <= ARP;
        updated_AR <= 1;
        if (JFlag_c == 1)
            begin
                ARP <= PARP;
            end
         else
            begin
                case ({IR3[6], IR3[5]})
                    2'b00: AR[ARP] <= AR[ARP];
                    2'b01: AR[ARP] <= AR[ARP] + 1;
                    2'b10: AR[ARP] <= AR[ARP] - 1;
                endcase
            end
        ARP <= IR3[2:0];
    end
end

MPYK: mpy_opcode <= IR3;

MAC: begin
    mpy_opcode <= IR3;
    alu_opcode <= IR3;

s1_in_reg <= Preg;
buff_mult_2 <= DM_out;
updated_AR <= 1;
if (IR3[7] == 1) begin
    PAR[ARP] <= AR[ARP];
    PARP <= ARP;
end
if (JFlag_c == 1)
    begin
        ARP <= PARP;
        AR[PARP] <= PAR[PARP];
    end
else
    begin
        case ({IR3[6], IR3[5]})
            2'b00 : AR[ARP] <= AR[ARP];
            2'b01 : AR[ARP] <= AR[ARP] + 1;
            2'b10 : AR[ARP] <= AR[ARP] - 1;
        endcase
        ARP <= IR3[2:0];
    end
BANZ: begin
    if (IR3[7] == 1) begin
        PAR[ARP] <= AR[ARP];
        PARP <= ARP;
        updated_AR <= 1;
        if (JFlag_c == 1)
            begin
                ARP <= PARP;
                AR[PARP] <= PAR[PARP];
            end
        else
            begin
                case ({IR3[6], IR3[5]})
                    2'b00 : AR[ARP] <= AR[ARP];
                    2'b01 : AR[ARP] <= AR[ARP] + 1;
                endcase
            end
    end
end
2'b10: AR[
    ARP] <= AR
    [ARP] -
    1;
    endcase
    end
    end
endcase
    end
    endcase
    end
    end
    end
    endcase
    end
    endcase
    end
    end
    endcase
    end
    end
    endcase
    end
    endcase
    end
    endcase
    end
    end
    endcase
    end
    end
    end
    endcase
    end
    end
    end
    endcase
    end
    end

alu_opcode <= IR3;
updated_AR <= 1;
s1_in_reg <= buff;
if (IR3[7] == 1) begin
    PAR[ARP] <= AR[ARP];
    buff <= DM_out;
    s1_in_reg <= DM_out;
    PARP <= ARP;
    if (JFlag_c == 1) begin
        ARP <= PARP;
        AR[PARP] <= PAR[PARP];
    end
    end
else begin
    case ({IR3[6], IR3[5]})
        2'b00: AR[
            ARP] <= AR
            [ARP];
        2'b01: AR[
            ARP] <= AR
            [ARP] +
            1;
        2'b10: AR[
            ARP] <= AR
            [ARP] -
            1;
endcase
    end
    end
    SAC: begin
        alu_opcode <= IR3;
    end
    end
    end
    end
s1_in_reg <= DM_out;

updated_AR <= 1;
if (IR3[7] == 1) begin
  PAR[ARP] <= AR[ARP];
  PARP <= ARP;
  if (JFlag_c == 1) begin
    ARP <= PARP;
    AR[PARP] <= PAR[PARP];
  end
else begin
  case (IR3[6], IR3[5])
    2'b00: AR[ARP] <= AR[ARP];
    2'b01: AR[ARP] <= AR[ARP] + 1;
    2'b10: AR[ARP] <= AR[ARP] - 1;
  endcase
  ARP <= IR3[2:0];
end
end
endcase
endcase
end

buff <= DM_out;
if (IR3[7] == 1) begin
  PAR[ARP] <= AR[ARP];
  PARP <= ARP;
  updated_AR <= 1;
  if (JFlag_c == 1) begin
    ARP <= PARP;
    AR[PARP] <= PAR[PARP];
  end
else begin
  case (IR3[15:11])
    LAR: begin
      buff <= DM_out;
      if (IR3[7] == 1) begin
        PAR[ARP] <= AR[ARP];
        PARP <= ARP;
        updated_AR <= 1;
        if (JFlag_c == 1) begin
          ARP <= PARP;
          AR[PARP] <= PAR[PARP];
        end
      end
end
I.1 RTL source code

case ({ IR3[6], IR3[5] })
  2'b00: AR[ARP] <= AR[ARP];
  2'b01: AR[ARP] <= AR[ARP] + 1;
  2'b10: AR[ARP] <= AR[ARP] - 1;
endcase
end

ARP <= IR3[2:0];
end

SAR: begin
  if (IR3[7] == 1) begin
    PAR[ARP] <= AR[ARP];
    PARP <= ARP;
    updated_AR <= 1;
    if (JFlag_c == 1)
      begin
        ARP <= PARP;
        AR[PARP] <= PAR[PARP];
      end
  end
else begin
  case ({ IR3[6], IR3[5] })
    2'b00: AR[ARP] <= AR[ARP];
    2'b01: AR[ARP] <= AR[ARP] + 1;
    2'b10: AR[ARP] <= AR[ARP] - 1;
  endcase
  end
end

end
LARK: AR[IR3[10:8]][7:0] <= IR3[7:0];
LARKH: AR[IR3[10:8]][15:8] <= IR3[7:0];
endcase
end
//— 6.3 Pipeline stage 2
if (stall_mc2 == 0)
begin
    case (IR2[15:0])
        CMPSIMD: begin
            wr_data <= 'b1; // For read
        end
        BANZ: begin
            JAddr2 <= PM_out;
        end
MPY,MAC,OR,XOR,SUBS,ADDS,AND,LDI,LT,LSI,LTP,
LTS,MAR,ZALH,ZALS,SUBSIMD,ADDSIMD: begin
    wr_data <= 'b1; // For read
end
endcase
end
endcase
case (IR2[15:11])
    LAR: begin
        wr_data <= 'b1; // For read
    end
SAR: begin
    next_opcode <= IR2;
    wr_data <= 'b0; // For write
end
endcase
endcase
end
endcase
endcase
endcase
endcase
endcase
endcase
endcase
endcase
endcase
767 end
768 BU: begin
769 if (branch_predict == 0)
770 JAddr <=PM_out;
771 else begin
772 JAddr2<=PM_out; end
773 end
774 CALL: begin
775 call_stack[CSP] <=PC;
776 if (branch_predict == 0)
777 JAddr <=PM_out;
778 else begin
779 JAddr2<=PM_out; end
780 end
781 RET: begin
782 CSP <=CSP−1;
783 if (branch_predict == 0)
784 JAddr <=call_stack[CSP−1][14:0];
785 else begin
786 JAddr2<=call_stack[CSP−1][14:0];
787 end
788 end
789 endcase
790 //
791 if (stall_mc3 == 0)
792 begin
793 if (IR4[15:0] == BU ||
794 IR4[15:0] == BGEZ ||
795 IR4[15:0] == BGZ ||
796 IR4[15:0] == BLEZ ||
797 IR4[15:0] == BLZ ||
798 IR4[15:0] == BNZ ||
799 IR4[15:0] == BV ||
800 IR4[15:0] == BZ ||
801 IR4[15:0] == CALL ||
802 IR4[15:0] == RET ||
803 IR4[15:8] == BANZ )begin
804 stall_mc3 <=1'b1; IR4 <=16'hfffff;
805 end
806 else if (JFlag_c == 1) begin
807 stall_mc3 <=1;IR4 <=16'hfffff;
808 end
809 else begin
810 stall_mc4 <=0;
811 IR4<= IR3;
I.1 RTL source code

```
806 end
807 JAddr4 <= JAddr3;
808 end
809 //
810 if (stall_mc2 == 0)
811 begin
812 if (IR3[15:0] == BU ||
813 IR3[15:0] == BGEZ ||
814 IR3[15:0] == BGZ ||
815 IR3[15:0] == BLEZ ||
816 IR3[15:0] == BLZ ||
817 IR3[15:0] == BNZ ||
818 IR3[15:0] == BV ||
819 IR3[15:0] == BZ ||
820 IR3[15:0] == CALL ||
821 IR3[15:0] == RET ||
822 IR3[15:8] == BANZ) begin stall_mc2 <= 1; IR3 <= 16'hffff; end
823 else if (JFlag_c == 1) begin
824 stall_mc2 <= 1; IR3 <= 16'hffff;
825 end
826 else begin
827 stall_mc3 <= 0; IR3 <= IR2; end
828 JAddr3 <= JAddr2;
829 end
830 //
831 if (stall_mc1 == 0)
832 begin
833 if (check_condition) begin
834 JAddr <= JAddr2;
835 check_condition <= 0;
836 end
837 if (branch_predict)
838 PC <= JAddr;
839 else if ((IR2[13:0] == BU || (IR2[15:0] == CALL))
840 PM_out;
841 else if (IR2[15:0] == RET)
842 call_stack[ CSP - 1][ 14:0 ] + 2;
843 else
844 PC <= PC + 1'b1;
845 if (IR2[15:0] == BU ||
846 IR2[15:0] == BGEZ ||
847 IR2[15:0] == BGZ ||
```
```vhdl
I.1 RTL source code

IR2[15:0] == BLEZ  ||
IR2[15:0] == BLZ  ||
IR2[15:0] == BNZ  ||
IR2[15:0] == BZ  ||
IR2[15:0] == BV  ||
IR2[15:0] == CALL ||
IR2[15:0] == RET ||
IR2[15:8] == BANZ )
<=16'hffff;
else begin
stall_mc2 <=0; IR2 <=PM_out; end
end
```

I.1 RTL source code

I.1.2 ALU

1 //
2 ///////////////////////////////////////////////////////////////////////////}
3 // Author : Shashank Simha
4 // Date : 12/12/2017
5 // University : Rochester Institute of Technology
6 // Description : This is a part of the DSP implemented for grad project
7 //
8 ///////////////////////////////////////////////////////////////////////////}
9
10 module ALU ( reset, clk, scan_in0, scan_en, test_mode, scan_out0, a, b, opcode, result, ov, carry, negative, zero, ov_2, carry_2, negative_2, zero_2, ov_3, carry_3, negative_3, zero_3, ov_4, carry_4, negative_4, zero_4 );
11
12 input reset, // system reset
13 clk; // system clock
14
15 input scan_in0, // test scan mode data input
16 scan_en, // test scan mode enable
17 test_mode; // test mode select
output
  scan_out0;  // test scan mode data output
input [31:0] a, b;
input [15:0] opcode;
output [31:0] result;
output ov, carry, negative, zero, ov_2, carry_2, negative_2, zero_2, ov_3, carry_3, negative_3, zero_3, ov_4, carry_4, negative_4, zero_4;

parameter [7:0] MPY = 8'b00000000; // 1.
parameter [7:0] MPYK = 8'b10100000; // 2.
parameter [7:0] MAC = 8'h02; // 3.
parameter [7:0] OR = 8'h03; // 4.
parameter [7:0] XOR = 8'h04; // 5.
parameter [15:0] SPAC = 16'h0100; // 6.
parameter [2:0] SUB = 3'h1; // 7.
parameter [7:0] SUBS = 8'h05; // 8.
parameter [2:0] ADD = 3'h2; // 9.
parameter [7:0] ADDS = 8'h06; // 10.
parameter [7:0] AND = 8'h07; // 11.
parameter [15:0] BU = 16'h010F; // 12.
parameter [15:0] BGEZ = 16'h0101; // 14.
parameter [15:0] BGZ = 16'h0102; // 15.
parameter [15:0] BLEZ = 16'h0103; // 16.
parameter [15:0] BLZ = 16'h0104; // 17.
parameter [15:0] BNZ = 16'h0105; // 18.
parameter [15:0] BV = 16'h0106; // 19.
parameter [15:0] BZ = 16'h0107; // 20.
parameter [2:0] LAC = 3'h3; // 21.
parameter [7:0] LACK = 8'b10100001; // 22.
parameter [7:0] LAR = 5'b11000; // 23.
parameter [7:0] LARK = 5'b11001; // 24.
parameter [7:0] LARKH = 5'b11011; // 25.
parameter [7:0] LARP = 8'b10100011; // 26.
parameter [7:0] LDP = 8'h09; // 27.
parameter [7:0] LDPK = 8'b10100100; // 28.
parameter [7:0] LT = 8'h0a; // 29.
parameter [7:0] LTA = 8'h0b; // 30.
parameter [7:0] LTD = 8'h0c; // 31.
parameter [7:0] LTP = 8'h0d; // 32.
parameter [7:0] LTS = 8'h0e; // 33.
parameter [7:0] MAR = 8'h0f; // 34.
parameter [15:0] PAC = 16'h011F; // 35.
parameter [15:0] ROVM = 16'h012F; // 36.
parameter [2:0] SAC = 3'h4; // 37.
parameter [7:0] SAR = 5'b11010; // 38.
1.1 RTL source code

91 parameter [15:0] NOP = 16'h014F; //42.
92 parameter [15:0] ZAC = 16'h015F; //43.
93 parameter [7:0] ZALH = 8'h13; //44.
94 parameter [7:0] ZALS = 8'h14; //45.
95 parameter [15:0] APAC = 16'h016F; //46.
96 parameter [6:0] CMPSIMD = 7'b00001101; //47.
97 parameter [7:0] SUBSIMD = 8'h16; //48.
98 parameter [7:0] ADDSIMD = 8'h17; //49.
99 parameter [8:0] BANZ = 8'h18; //13.
100 parameter [15:0] PUSH = 16'h017F; //50.
101 parameter [15:0] POP = 16'h018F; //51.
102 parameter [15:0] CALL = 16'h01AF; //52.
103 parameter [15:0] RET = 16'h019F; //53.

104 wire [31:0] out_comp;
105 wire comp_en;
106 wire [31:0] b_sub;
107 wire [7:0] a1, a2, a3, a4;
108 wire [7:0] b1, b2, b3, b4;
109 wire [3:0] Cin;
110 wire [3:0] Cout;
111 wire [7:0] S1, S2, S3, S4;

          ? ((a[31] ~^ b[31]) && result[31]) :
          ? ((a[31] ~ b[31]) && result[31]) :
113          : 0;
114 assign ov_2 = (opcode[15:8]==ADDSIMD)? ((a[23] ~^ b[23]) && result[23]):
          (opcode[15:8]==SUBSIMD)? ((a[23] ~ b[23]) && result[23]):
          0;
          0;
          (opcode[15:8]==SUBSIMD)? ((a[7] ~ b[7]) && result[7]):
          0;
117 assign carry = Cout[0];
118 assign carry_2=Cout[1];
119 assign carry_3=Cout[2];
120 assign carry_4=Cout[3];
121 assign negative = result[31];
122 assign negative_2 = result[23];
123 assign negative_3 = result[15];
124 assign negative_4 = result[7];
           ? ((result[31:24] == 0)? 1:0) : (result==0)? 1: 0;
assign zero_2 = (result[23:16]==0)? 1:0;
assign zero_3 = (result[15:8]==0)? 1:0;
assign zero_4 = (result[7:0]==0)? 1:0;
  (opcode[15:8] == OR)
  ? (a | b):
  (opcode[15:8] == AND)
  ? (a & b):
  (opcode[15:8] == XOR)
  ? (a ^ b):
//
assign comp_en = (opcode[15:9] == CMPSIMD)? 1:0;
assign a4 = a[31:24];
assign a3 = a[23:16];
assign a2 = a[15:8];
assign a1 = a[7:0];
assign b_sub = (opcode[15:8] == SUBSIMD)? {(~b[31:24]) +1,(~b[23:16]) +1,(~b[15:8]) +1,(~b[7:0]) +1} : (~b) + 1; // 2's complement for subtraction

assign Cin[0] = 0;

adder A1 (.A (a1), .B (b1), .Cin (Cin[0]), .Cout (Cout[0]), .Sum (S1), .scan_en (scan_en), .scan_in0 (scan_in0), .test_mode (test_mode), .scan_out0 (scan_out0));

adder A2 (.A (a2), .B (b2), .Cin (Cin[1]), .Cout (Cout[1]), .Sum (S2), .scan_en (scan_en), .scan_in0 (scan_in0), .test_mode (test_mode), .scan_out0 (scan_out0));

adder A3 (.A (a3), .B (b3), .Cin (Cin[2]), .Cout (Cout[2]), .Sum (S3), .scan_en (scan_en), .scan_in0 (scan_in0), .test_mode (test_mode), .scan_out0 (scan_out0));

adder A4 (.A (a4), .B (b4),
I.1 RTL source code

201 .Cin (Cin[3]),
202 .Cout (Cout[3]),
203 .Sum (S4),
204 .scan_en (scan_en),
205 .scan_in0 (scan_in0),
206 .test_mode (test_mode),
207 .scan_out0 (scan_out0)
208 );
209
210 compare_select comp_4 (.scan_in0(scan_in0),
211 .scan_en(scan_en),
212 .test_mode(test_mode),
213 .scan_out0(scan_out0),
214 .A(a[31:24]),
215 .B(b[31:24]),
216 .flag(opcode[8]),
217 .C(out_comp[31:24]),
218 .en(comp_en)
219 );
220 compare_select comp_3 (.scan_in0(scan_in0),
221 .scan_en(scan_en),
222 .test_mode(test_mode),
223 .scan_out0(scan_out0),
224 .A(a[23:16]),
225 .B(b[23:16]),
226 .flag(opcode[8]),
227 .C(out_comp[23:16]),
228 .en(comp_en)
229 );
230 compare_select comp_2 (.scan_in0(scan_in0),
231 .scan_en(scan_en),
232 .test_mode(test_mode),
233 .scan_out0(scan_out0),
234 .A(a[15:8]),
235 .B(b[15:8]),
236 .flag(opcode[8]),
237 .C(out_comp[15:8]),
238 .en(comp_en)
239 );
240 compare_select comp_1 (.scan_in0(scan_in0),
241 .scan_en(scan_en),
242 .test_mode(test_mode),
243 .scan_out0(scan_out0),
244 .A(a[7:0]),
245 .B(b[7:0]),
246 .flag(opcode[8]),
247 .C(out_comp[7:0]),
248 .en(comp_en)
249 );
250
251   endmodule
I.1.3 Input shifter

```verilog
module shifter_input (scan_in0,
                      scan_out0,
                      scan_en,
                      test_mode,
                      shift_in,
                      opcode,
                      shift_out);

//--- 1 ISA Parameters
//
parameter [7:0] MPY = 8'h00000000; //1.
parameter [7:0] MPYK = 8'h10100000; //2.
parameter [7:0] MAC = 8'h02; //3.
parameter [7:0] OR = 8'h03; //4.
parameter [7:0] XOR = 8'h04; //5.
parameter [15:0] SPAC = 16'h0100; //6.
parameter [2:0] SUB = 3'h1; //7.
parameter [7:0] SUBS = 8'h05; //8.
parameter [2:0] ADD = 3'h2; //9.
parameter [7:0] ADDS = 8'h06; //10.
parameter [7:0] AND = 8'h07; //11.
parameter [15:0] BU = 16'h010F; //12.
parameter [8:0] BANZ = 8'h01; //13.
parameter [15:0] BGEZ = 16'h0101; //14.
parameter [15:0] BGZ = 16'h0102; //15.
parameter [15:0] BLEZ = 16'h0103; //16.
parameter [15:0] BLZ = 16'h0104; //17.
parameter [15:0] BNZ = 16'h0105; //18.
parameter [15:0] BV = 16'h0106; //19.
parameter [15:0] BZ = 16'h0107; //20.
parameter [2:0] LAC = 3'h3; //21.
parameter [7:0] LACK = 8'b10100001; //22.
parameter [7:0] LAR = 5'b11000; //23.
parameter [7:0] LARK = 5'b11001; //24.
parameter [7:0] LARKH = 5'b11011; //25.
parameter [7:0] LARP = 8'b10100011; //26.
parameter [7:0] LDP = 8'h09; //27.
```
parameter [7:0] LDPK = 8'b10100100; //28.
parameter [7:0] LT  = 8'h0a; //29.
parameter [7:0] LTA = 8'h0b; //30.
parameter [7:0] LTD = 8'h0c; //31.
parameter [7:0] LTP = 8'h0d; //32.
parameter [7:0] LTS = 8'h0e; //33.
parameter [7:0] MAR = 8'h0f; //34.
parameter [15:0] PAC  = 16'h011F; //35.
parameter [15:0] ROVM = 16'h012F; //36.
parameter [2:0] SAC = 3'h4; //37.
parameter [7:0] SAR = 5'b11010; //38.
parameter [7:0] TBLR = 8'h11; //40.
parameter [7:0] TBLW = 8'h12; //41.
parameter [15:0] NOP  = 16'h014F; //42.
parameter [15:0] ZAC = 16'h015F; //43.
parameter [7:0] ZALH = 8'h13; //44.
parameter [7:0] ZALS = 8'h14; //45.
parameter [15:0] APAC = 16'h016F; //46.
parameter [7:0] CMPSMD = 8'h15; //47.
parameter [7:0] SUBSIMD = 8'h16; //48.
parameter [7:0] ADDSIMD = 8'h17; //49.
parameter [15:0] PUSH = 16'h017F; //50.
parameter [15:0] POP  = 16'h018F; //51.
parameter [15:0] CALL = 16'h01AF; //52.
parameter [15:0] RET  = 16'h019F; //53.

input  scan_in0,
       scan_en,
       test_mode;
output scan_out0;

input [31:0] shift_in;
input [15:0] opcode;
output [31:0] shift_out;

// Shifts operand for ADD and SUB
  (((opcode[12:8]==5'b00000) ? shift_in :
    (opcode[12:8]==5'b00001) ? 1'h0,shift_in[31:1] :
    (opcode[12:8]==5'b00010) ? 2'h0,shift_in[31:2] :
    (opcode[12:8]==5'b00011) ? 3'h0,shift_in[31:3] :
    (opcode[12:8]==5'b00100) ? 4'h0,shift_in[31:4] :
    (opcode[12:8]==5'b00101) ? 5'h0,shift_in[31:5] :
    (opcode[12:8]==5'b00110) ? 6'h0,shift_in[31:6] :
    (opcode[12:8]==5'b00111) ? 7'h0,shift_in[31:7] :
    (opcode[12:8]==5'b01000) ? 8'h0,shift_in[31:8] :
):
I.1 RTL source code

(\text{opcode}[12:8]==5'b01001) \quad ? \{9'h0, \text{shift}_\text{in}[31:9]\} : \\
(\text{opcode}[12:8]==5'b01010) \quad ? \{10'h0, \text{shift}_\text{in}[31:10]\} : \\
(\text{opcode}[12:8]==5'b01011) \quad ? \{11'h0, \text{shift}_\text{in}[31:11]\} : \\
(\text{opcode}[12:8]==5'b01100) \quad ? \{12'h0, \text{shift}_\text{in}[31:12]\} : \\
(\text{opcode}[12:8]==5'b01101) \quad ? \{13'h0, \text{shift}_\text{in}[31:13]\} : \\
(\text{opcode}[12:8]==5'b01110) \quad ? \{14'h0, \text{shift}_\text{in}[31:14]\} : \\
(\text{opcode}[12:8]==5'b01111) \quad ? \{15'h0, \text{shift}_\text{in}[31:15]\} : \\
(\text{opcode}[12:8]==5'b10000) \quad ? \{16'h0, \text{shift}_\text{in}[31:16]\} : \\
(\text{opcode}[12:8]==5'b10001) \quad ? \{17'h0, \text{shift}_\text{in}[31:17]\} : \\
(\text{opcode}[12:8]==5'b10010) \quad ? \{18'h0, \text{shift}_\text{in}[31:18]\} : \\
(\text{opcode}[12:8]==5'b10011) \quad ? \{19'h0, \text{shift}_\text{in}[31:19]\} : \\
(\text{opcode}[12:8]==5'b10100) \quad ? \{20'h0, \text{shift}_\text{in}[31:20]\} : \\
(\text{opcode}[12:8]==5'b10101) \quad ? \{21'h0, \text{shift}_\text{in}[31:21]\} : \\
(\text{opcode}[12:8]==5'b10110) \quad ? \{22'h0, \text{shift}_\text{in}[31:22]\} : \\
(\text{opcode}[12:8]==5'b10111) \quad ? \{23'h0, \text{shift}_\text{in}[31:23]\} : \\
(\text{opcode}[12:8]==5'b11000) \quad ? \{24'h0, \text{shift}_\text{in}[31:24]\} : \\
(\text{opcode}[12:8]==5'b11001) \quad ? \{25'h0, \text{shift}_\text{in}[31:25]\} : \\
(\text{opcode}[12:8]==5'b11010) \quad ? \{26'h0, \text{shift}_\text{in}[31:26]\} : \\
(\text{opcode}[12:8]==5'b11011) \quad ? \{27'h0, \text{shift}_\text{in}[31:27]\} : \\
(\text{opcode}[12:8]==5'b11100) \quad ? \{28'h0, \text{shift}_\text{in}[31:28]\} : \\
(\text{opcode}[12:8]==5'b11101) \quad ? \{29'h0, \text{shift}_\text{in}[31:29]\} : \\
(\text{opcode}[12:8]==5'b11110) \quad ? \{30'h0, \text{shift}_\text{in}[31:30]\} : \\
(\text{opcode}[12:8]==5'b11111) \quad ? \{31'h0, \text{shift}_\text{in}[31]\} : \\
32'h0) : \\
\text{shift}_\text{in} ; \\
\text{endmodule}
I.1.4 Output shifter

module shifter_output (scan_in0,
   scan_out0,
   scan_en,
   test_mode,
   shift_in,
   opcode,
   shift_out);

//— 1 ISA Parameters

parameter [7:0] MPY = 8'h00000000; // 1.
parameter [7:0] MPYK = 8'h10100000; // 2.
parameter [7:0] MAC = 8'h02; // 3.
parameter [7:0] OR = 8'h03; // 4.
parameter [7:0] XOR = 8'h04; // 5.
parameter [15:0] SPAC = 16'h0100; // 6.
parameter [2:0] SUB = 3'h1; // 7.
parameter [7:0] SUBS = 8'h05; // 8.
parameter [2:0] ADD = 3'h2; // 9.
parameter [7:0] ADDS = 8'h06; // 10.
parameter [7:0] AND = 8'h07; // 11.
parameter [15:0] BU = 16'h010F; // 12.
parameter [8:0] BANZ = 8'h01; // 13.
parameter [15:0] BGEZ = 16'h0101; // 14.
parameter [15:0] BGZ = 16'h0102; // 15.
parameter [15:0] BLEZ = 16'h0103; // 16.
parameter [15:0] BLZ = 16'h0104; // 17.
parameter [15:0] BNZ = 16'h0105; // 18.
parameter [15:0] BV = 16'h0106; // 19.
parameter [15:0] BZ = 16'h0107; // 20.
parameter [2:0] LAC = 3'h3; // 21.
parameter [7:0] LACK = 8'b10100001; // 22.
parameter [7:0] LAR = 5'b11000; // 23.
parameter [7:0] LARK = 5'b11001; // 24.
parameter [7:0] LARKH = 5'b11101; // 25.
parameter [7:0] LARP = 8'b10100011; // 26.
parameter [7:0] LDP = 8'h09; // 27.
parameter [7:0] LDPK = 8'b10100100; //28.
parameter [7:0] LT = 8'h0a; //29.
parameter [7:0] LTA = 8'h0b; //30.
parameter [7:0] LTD = 8'h0c; //31.
parameter [7:0] LTP = 8'h0d; //32.
parameter [7:0] LTS = 8'h0e; //33.
parameter [7:0] MAR = 8'h0f; //34.
parameter [15:0] PAC = 16'h011F; //35.
parameter [15:0] ROVM = 16'h012F; //36.
parameter [2:0] SAC = 3'h4; //37.
parameter [7:0] SAR = 5'b11010; //38.
parameter [7:0] TBLR = 8'h11; //40.
parameter [7:0] TBLW = 8'h12; //41.
parameter [15:0] NOP = 16'h014F; //42.
parameter [15:0] ZAC = 16'h015F; //43.
parameter [7:0] ZALH = 8'h13; //44.
parameter [7:0] ZALS = 8'h14; //45.
parameter [15:0] APAC = 16'h016F; //46.
parameter [7:0] CMPSMD = 8'h15; //47.
parameter [7:0] SUBSIMD = 8'h16; //48.
parameter [7:0] ADDSIMD = 8'h17; //49.
parameter [15:0] PUSH = 16'h017F; //50.
parameter [15:0] POP = 16'h018F; //51.
parameter [15:0] CALL = 16'h01AF; //52.
parameter [15:0] RET = 16'h019F; //53.

input scan_in0,
    scan_en,
    test_mode;

output scan_out0;

input [31:0] shift_in;
input [15:0] opcode;

output [31:0] shift_out;

// Shifts output for SAC
    ((opcode[12:8]==5'b00000) ? shift_in
    
    (opcode[12:8]==5'b00001) ? {1'h0, shift_in[31:1]}
    
    (opcode[12:8]==5'b00010) ? {2'h0, shift_in[31:2]}
    
    (opcode[12:8]==5'b00011) ? {3'h0, shift_in[31:3]}
    
    (opcode[12:8]==5'b00100) ? {4'h0, shift_in[31:4]}
    
    (opcode[12:8]==5'b00101) ? {5'h0, shift_in[31:5]}
    
    (opcode[12:8]==5'b00110) ? {6'h0, shift_in[31:6]}
    
    (opcode[12:8]==5'b00111) ? {7'h0, shift_in[31:7]}
    
    (opcode[12:8]==5'b01000) ? {8'h0, shift_in[31:8]}
    
    : shift_in[31:0] :} :
    : ;
92 (opcode[12:8]==5'b01001) ? {9'h0, shift_in[31:9]} : 
93 (opcode[12:8]==5'b01010) ? {10'h0, shift_in[31:10]} : 
94 (opcode[12:8]==5'b01011) ? {11'h0, shift_in[31:11]} : 
95 (opcode[12:8]==5'b01100) ? {12'h0, shift_in[31:12]} : 
96 (opcode[12:8]==5'b01101) ? {13'h0, shift_in[31:13]} : 
97 (opcode[12:8]==5'b01110) ? {14'h0, shift_in[31:14]} : 
98 (opcode[12:8]==5'b01111) ? {15'h0, shift_in[31:15]} : 
99 (opcode[12:8]==5'b10000) ? {16'h0, shift_in[31:16]} : 
100 (opcode[12:8]==5'b10001) ? {17'h0, shift_in[31:17]} : 
101 (opcode[12:8]==5'b10010) ? {18'h0, shift_in[31:18]} : 
102 (opcode[12:8]==5'b10011) ? {19'h0, shift_in[31:19]} : 
103 (opcode[12:8]==5'b10100) ? {20'h0, shift_in[31:20]} : 
104 (opcode[12:8]==5'b10101) ? {21'h0, shift_in[31:21]} : 
105 (opcode[12:8]==5'b10110) ? {22'h0, shift_in[31:22]} : 
106 (opcode[12:8]==5'b10111) ? {23'h0, shift_in[31:23]} : 
107 (opcode[12:8]==5'b11000) ? {24'h0, shift_in[31:24]} : 
108 (opcode[12:8]==5'b11001) ? {25'h0, shift_in[31:25]} : 
109 (opcode[12:8]==5'b11010) ? {26'h0, shift_in[31:26]} : 
110 (opcode[12:8]==5'b11011) ? {27'h0, shift_in[31:27]} : 
111 (opcode[12:8]==5'b11100) ? {28'h0, shift_in[31:28]} : 
112 (opcode[12:8]==5'b11101) ? {29'h0, shift_in[31:29]} : 
113 (opcode[12:8]==5'b11110) ? {30'h0, shift_in[31:30]} : 
114 (opcode[12:8]==5'b11111) ? {31'h0, shift_in[31]} : 
115 32'h0 : 
116 shift_in ;
117 endmodule
I.1.5 Compare select unit

//
// Author : Shashank Simha
// Date : 12/12/2017
// University : Rochester Institute of Technology
// Description : This is a part of the DSP implemented for grad project
//
module compare_select ( scan_in0 , scan_en , test_mode , scan_out0 , A,B,flag ,C, en ) ;

input scan_in0 , scan_en , test_mode ;
output scan_out0 ;
input [7:0] A,B;
input flag , en ;
output [7:0] C;

assign C = en ? ( flag ? ((A>B)? A : B) :((A>B)? B : A) ) : 0 ; // if flag ==1 -> C = Greater

endmodule
I.1.6 Multiplier

```verilog
module multiplier (scan_in0, scan_out0, scan_en, test_mode, a, b, ov, product);

input scan_in0, scan_en, test_mode;
output scan_out0;
input [15:0] a, b;
output [31:0] product;
input ov;

wire [15:0] abs_a, abs_b;
wire [15:0] twos_comp_a, twos_comp_b;

parameter PSAT = 32'h7fffffff;
parameter NSAT = 32'h80000000;

wire [31:0] abs_result;

assign twos_comp_a = ((~a) + 1);
assign twos_comp_b = ((~b) + 1);

assign abs_a = a[15] ?
    (ov ? ((a == NSAT) ? PSAT : twos_comp_a) : twos_comp_a) : a;
assign abs_b = b[15] ?
    (ov ? ((b == NSAT) ? PSAT : twos_comp_b) : twos_comp_b) : b;
assign abs_result = abs_a * abs_b;
assign product = (a[15] ^ b[15]) ? ((~abs_result) + 1) : abs_result;

endmodule
```
I.1 RTL source code

I.1.7 Adder

```vhdl
//
phoneNumber

module adder (scan_in0, scan_en, test_mode, scan_out0, A, B, Cin, Cout, Sum);

input scan_in0, scan_en, test_mode;
output scan_out0;
input [7:0] A, B;
input Cin;
output [7:0] Sum;
output Cout;
wire [7:0] G;
wire [7:0] P;
wire [7:0] C;
assign {Cout, Sum} = A + B + Cin;
endmodule
```
I.2 Assembler designed in Perl

```perl
#!/usr/bin/perl

# Author : Shashank Simha
# Date : 12/12/2017
# University : Rochester Institute of Technology
# Description : This is a part of the ASSEMBLER for DSP implemented
# for grad project
#
use strict;
use warnings;

my $name = 'median_filter_first_try';
my $assembly_file = $name . '.txt';
my $mif_file = $name . '.mif';
my $hex_file = $name . '.hex';

my @code;
my @comments;
my @line_number;
my $line_count;
my $error_count;

my @code_rearr;
my @opcode_extracted;
my @non_opcode_extracted;
my %jmp_in_label_extracted;
my %jmp_out_label_extracted;
my @jmp_labels_called;
my $j = 0;

my @opcode_generated;
my @non_opcode_generated;
my @jmp_addr_generated;
my @jmp_addr_generated_nonconverted;
my $mode;
my $dir_addr;
my $ar;
my $narp;
my $constant;
my $incr_oper;
my $shift;
```
I.2 Assembler designed in Perl

```perl
# # # # # # # # # # # # # # # # # # # # #
my $greater_lesser;
# # # # # # # # # # # # # # # # # # # # #

my $jmp_label;

my %add_sub_ar = ( "\*" => "00", "\*+" => "01", "\*-" => "10" );

my %opcode_3bit = ( 'SUB' => "001", 'ADD' => "010", 'LAC' => "011", 'SAC' => "100" );

my %opcode_5bit = ( 'LARKH' => "11011", 'LARK' => "11001", 'LAR' => "11000", 'SAR' => "11010" );

my %opcode_7bit = ( 'CMPSIMD' => "0001101" );

my %opcode_8bit = (
    # KEY => 01234567
    'MPY' => "00000000", 'MPYK' => "10100000", 'MAC' => "00000010", 'OR' => "00000011", 'XOR' => "00000100", 'SUBS' => "00000101", 'ADDS' => "00000110", 'AND' => "00000111", 'LACK' => "10100001", 'LARP' => "10100011", 'LDP' => "00001001", 'LDPK' => "10100100", 'LT' => "00001010", 'LTA' => "00001011", 'LTD' => "00001100", # 2 cycle
    'LTP' => "00001101", 'LTS' => "00001110", 
);```

I.2 Assembler designed in Perl

```
96  'MAR' => '00001111',
97  'SOVM' => '00000001',
98  'TBLR' => '00010001',
99  'TBLW' => '00010010',
100 'ZALH' => '00010111',
101 'ZALS' => '00010100',
102 'SUBSSIMD' => '00010110',
103 'ADDSSIMD' => '00010111',
104 'BANZ' => '00011000',
105 ) ;
106 my %opcode_16bit = (
107     #KEY => 0123456789ABCDEF
108     'SPAC' => '0000000100000000',
109     'BU' => '0000000100001111',
110     'BGEZ' => '0000000100000001',
111     'BGZ' => '0000000100000010',
112     'BLEZ' => '0000000100000011',
113     'BLZ' => '0000000100000100',
114     'BNZ' => '0000000100000101',
115     'BV' => '0000000100000110',
116     'BZ' => '0000000100000111',
117     'PAC' => '0000000100011111',
118     'ROV' => '0000000101001111',
119     'SOVM' => '0000000101011111',
120     'NOP' => '0000000101101111',
121     'ZAC' => '0000000101111111',
122     'APAC' => '0000000110001111',
123     'POP' => '0000000110101111',
124     'PUSH' => '0000000110001111',
125     'RET' => '0000000110011111',
126     'CALL' => '0000000110101111'
127 ) ;
128
129 open ( my $in_file , '<:encoding(UTF-8)' , $assembly_file ) or die ' 	Error: Assembly input file not found!
"
130 while (<$in_file>) {
131     chomp $_;
132     $line_count++;
133     if ( ( $_ =~ /\(.*\) ; (.*\)/ ) & & ! ( $_ =~ /
134     my $code = $1 ;
135     $code =~ s/\s+//;
136     push (@code , $code ) ;
137     push ( @comments , $2 ) ;
138     push ( @line_number , $line_count ) ;
139     } else {
140         $line_count++;
141         if ( ( $_ =~ /
142         my $code = $1 ;
143         $code =~ s/\s+//;
144         push (@code , $code) ;
145         push ( @comments , $2 ) ;
146         push ( @line_number , $line_count ) ;
147         } else {
148         end of assembly file.
149     }
150     }
151     else { "$_" ;
152     }
153     # handle comments, labels, etc.
154     # handle label assignments
155     # handle operand stack
156     # handle instruction set
157     # translate assembly to machine code
158     # output machine code
159     	break
160     }
161 
162 # END OF assembler designed in Perl
```

---

The above code snippet is a Perl script designed to parse an assembly file and create a mapping between assembly instructions and their corresponding 16-bit codes. It includes handling of assembly instructions, comments, labels, and operand stack management. The script also translates assembly instructions to machine code and outputs the machine code.
I.2 Assembler designed in Perl

```
144   $error_count++;  
145   print "ERROR: Syntax error in line $line_count."
146 }
147 }

# Machine code generation

```

```
152   foreach (my $i = 0; $i < @code; $i = $i + 1){
153     $code_rearr[$j] = $code[$i];
154     if ($code[$i] =~ /^ (.*)[*:*](.*)$/){
155       $jmp_in_label_extracted{$1} = $j;
156       $code[$i] = $2;
157   }
158   }

159   if ($code[$i] =~ / (.*[*:*](.*)+)/){
160     chomp $1;
161     chomp $2;
162     $opcode_extracted[$i] = uc($1);
163     $non_opcode_extracted[$i] = $2;
164     if (exists $opcode_16bit{$opcode_extracted[$i]}{  
165       $opcode_generated[$j] = $opcode_16bit{  
166       $opcode_extracted[$i]};
167   } else {
168     if ($opcode_extracted[$i] eq "BU" || #1
169       $opcode_extracted[$i] eq "BGEZ" || #2
170       $opcode_extracted[$i] eq "BGZ" || #3
171       $opcode_extracted[$i] eq "BLEZ" || #4
172       $opcode_extracted[$i] eq "BLZ" || #5
173       $opcode_extracted[$i] eq "BNZ" || #6
174       $opcode_extracted[$i] eq "BV" || #7
175       $opcode_extracted[$i] eq "BZ" || #8
176       $opcode_extracted[$i] eq "CALL" || #9
177     }  
178     $j++;  
179     if ($non_opcode_extracted[$i] =~ / (.*)[*:*](.*)$/){
180       push (@jmp_labels_called, $1);  
181       push @{$jmp_out_label_extracted{$1}}, $j;
182       # print "Found $1 at $i array @{$  
183       $jmp_out_label_extracted{$1}} \n"
184     }
185   }
186   
187   # print "Direct address $1 found:  
188   # machine code= $opcode_generated[$i].$non_opcode_generated[$i]  
189   # found in line ".$line_number[$i - 1]."
190   
191 }
```
I.2 Assembler designed in Perl

```perl
184
185     else {
186         $error_count++;
187         print "ERROR : Invalid Instruction
188             $code[$i] in line ", $line_number[$i], "\n"
189         }
190     }
191 #
192 elif (exists $opcode_7bit{ $opcode_extracted[$i] }) {
193     $opcode_generated[$j] = $opcode_7bit{ $opcode_extracted[$i] };
194     if ($non_opcode_extracted[$i] =~ /^0x([\-9A-9A-9A-F])\s*([\-9A-9A-9A-F])\s*([GL])\s*$/) {
195         $dir_addr = $1;
196         $greater_lesser= $2;
197         $mode = "0";
198         if ($greater_lesser eq "G") {
199             $greater_lesser = "1";
200         }
201 else {
202             $greater_lesser = "0";
203         }
204     }
205     else {
206         $non_opcode_generated[$j] = $greater_lesser.$mode.sprintf ("%07b", hex($i));
207     }
208     # print "Direct address $1 found; machine code= $opcode_generated[$j].$non_opcode_generated[$j] found in line ", $line_number[$i - 1]."\n"
209     }
211     $incr_oper = $2;
212     $ar = $1;
213     $greater_lesser= $3;
214     $mode = "1";
215     if ($greater_lesser eq "G") {
216         $greater_lesser = "1";
217     }
218 }
```
I.2 Assembler designed in Perl

```perl
else {
    $greater_lesser = "0";
}

if ((exists $add_sub_ar{ $incrOper })
    && ($ar <= 7)) {
    $non_opcode_generated[$j] =
        $greater_lesser . $mode .
        $add_sub_ar{ $incrOper } .
        "00" . sprintf ('%03b', $ar);
} else {
    $error_count++;

    print 'ERROR : Invalid
    Instruction $code[$i] in
    line ".$line_number[$i]." 
    \n';
}

else {
    $error_count++;

    print 'ERROR : Invalid Instruction
    $code[$i] in line ".$line_number[$i]." 
    \n';
}

else if (exists $opcode_8bit { $opcodeExtracted[$i] }) {
    $opcode_generated[$j] = $opcode_8bit {
        $opcode_extracted[$i]};

    if ($opcode_extracted[$i] eq 'LDPK' ||
        $opcode_extracted[$i] eq 'LACK' ||
        $opcode_extracted[$i] eq 'MPYK') {
        if ($non_opcode_extracted[$i] =~ /^0x([0-9A-F][0-9A-F])$/i) {
            $constant = $1;
            $non_opcode_generated[$j] = sprintf
                ('%08b', hex($1));

            # print "constant $1 found; machine
            code= $opcode_generated[$j],
            $non_opcode_generated[$j] found
            in line ".$line_number[$i-1]." \n ";
        }
    } else {
        $error_count++;
```
I.2 Assembler designed in Perl

```perl
print 'ERROR : Constant $constant invalid $code[$i] in line ".
 lineWidth'[i]."\n";

elsif ($opcode_extracted[$i] eq "LARP") {
    if ($non_opcode_extracted[$i] =~ /\s*+ar\s*(.*)\s*/i) {
        $constant = $1;
        $non_opcode_generated[$j] = sprintf ('%08b', hex($1));
        # print "constant $1 found; machine code= $opcode_generated[$j] 
        $non_opcode_generated[$j] found in line ".$line_number[$i-1]."\n"
;
    }
else {
        $error_count++;
        print 'ERROR : Constant $constant invalid $code[$i] in line ".
        lineWidth'[i]."\n";
    }
}
elsif ($opcode_extracted[$i] eq "BANZ") {
    $j++;
    if ($non_opcode_extracted[$i] =~ /\s*+ar\s*(.*)\s*/i) {
        push (@jmp_labels_called, $1);
        push @{ $jmp_out_label_extracted{$1} }, $j;
        # print "Found $1 at $i array @{
        $jmp_out_label_extracted{$1} } \n"
;
    }
    $incr_oper = $3;
    $ar = $2;
    $mode = "1";
    if ((exists $add_sub_ar{$incr_oper})
        && ($ar <= 7)) {
        $non_opcode_generated[$j-1] = $mode, $add_sub_ar{
            $incr_oper },'00'.sprintf ('%03b', $ar);
        # print "Indirect address found; narp= ar$ar & 
        operation= $add_sub_ar{$
        $incr_oper} machine code= 
        $opcode_generated[$j]."\n"
;
```
Assembler designed in Perl

```perl
# Non-op code generated

270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295

$non_opcode_generated[$j]
found in line '.'.
$line_number[$i - 1]."\n":

else {

$error_count++;

print "ERROR: Invalid Instruction $code[$i] in line ".$line_number[$i].'
";
}

else {

push @{$jmp_out_label_extracted{$1}}, $j;
# print "Found $1 at $i array @{$jmp_out_label_extracted{$1}} \n ";

$mode = "0";
$non_opcode_generated[$j - 1] = $mode.
sprintf ("%07b", hex(0));
# print "Direct address $1 found; machine code= $opcode_generated[$j].$non_opcode_generated[$j] found in line ".$line_number[$i - 1]."\n":
}
else {

$error_count++;

print "ERROR: Invalid Instruction $code[$i] in line ".$line_number[$i]."\n":
}
else {

if ($non_opcode_extracted[$i] =~ /^0x([0-9A-F][0-9A-F][0-9A-F][0-9A-F][0-9A-F][0-9A-F][0-9A-F])\n\s*$i ){

$dir_addr = $1;
$mode = "0";
$non_opcode_generated[$j] = $mode.
sprintf ("%07b", hex($1));
# print "Direct address $1 found; machine code= $opcode_generated[$j].$non_opcode_generated[$j] found in line ".$line_number[$i - 1]."\n":

```
else if ($non_opcode_extracted[$i] =~ /\^\s*ar(.*?)[\s]*,\s*([0-9A-F]+[0-9A-F])\s*$/i) {
    $incr_oper = $2;
    $ar = $1;
    $mode = "1";
    if (($exists $add_sub_ar{ $incr_oper }) && ($ar <= 7)) {
        $non_opcode_generated[$j] = $mode . $add_sub_ar{$incr_oper} . "00 " . sprintf("%03b ", $ar);
# print "Indirect address found: narp= ar$ar & operation= $add_sub_ar{$incr_oper} machine code= $opcode_generated[$j]. $non_opcode_generated[$j] found in line ". $line_number[$i - 1]. \\
    } else {
        $error_count++;
        print "ERROR : Invalid Instruction $code[$i] in line ". $line_number[$i]. \\
    } else {
        $error_count++;
        print "ERROR : Invalid Instruction $code[$i] in line ". $line_number[$i]. \\
    } else {
        $error_count++;
        print "ERROR : Invalid Instruction $code[$i] in line ". $line_number[$i]. \\
    } else {
        $error_count++;
        print "ERROR : Invalid Instruction $code[$i] in line ". $line_number[$i]. \\
    }
} else if ($opcode_5bit{ $opcode_extracted[$i] }) {
    $opcode_generated[$j] = $opcode_5bit{ $opcode_extracted[$i] };
    if ($opcode_extracted[$i] eq "LARK") {
        if ($non_opcode_extracted[$i] =~ /^\s*ar(.*/\s*)*[0-9A-F][0-9A-F]\s*i$/i) {
            $ar = $1;
            $constant = $2;
            if ($ar <= 7) {
                $non_opcode_generated[$j] = sprintf("%03b ", $ar).
I.2 Assembler designed in Perl

```perl
I-50

```sprintf ('%08b', hex($constant));

# print 'constant $2 found
for ar$1 = machine code=
$opcode_generated[$j].
$non_opcode_generated[$j]
$code[$i] in line '.
$line_number[$i].'\n';

```325
326 else {
327     $error_count++;
328     print 'ERROR : Invalid Instruction $code[$i] in line '. $line_number[$i].' \n';
329 }
330 }
331 }
332 else {
333     if ((\$non_opcode_extracted[$i] =~ /^\s*ar .\s+\s+0x([0-9A-F][0-9A-F]\s*)$/i) && (ar <= 7) && (ar_n <= 7)) {
334         my $ar_n = $1;
335         $dir_addr = $2;
336         $mode = '0';
337         $non_opcode_generated[$j] = sprintf ('%03b', hex($ar_n)).$mode.
338         sprintf ('%07b', hex($2));
339     # print 'Direct address $1 found:
340     machine code= $opcode_generated[ $j ]. $non_opcode_generated[ $j ]
341     found in line '. $line_number[ $i ] -1].'\n';
342 }
343 elseif ($non_opcode_extracted[$i] =~ /^\s*ar .\s+\s+ar .\s+\s+$/i) {
344     my $ar_n = $1;
345     $incr_oper = $2;
346     $ar = $3;
347     $mode = '1';
348     if ((exists $add_sub_ar{\$incr_oper}) && ($ar <= 7) && ($ar_n <= 7)) {
349         $non_opcode_generated[\$j] = $mode.$add_sub_ar{\$incr_oper}.'00'.sprintf ('%03b', $ar_n).$add_sub_ar{\$incr_oper}.'00'.sprintf ('%03b', $ar);
```
# print 'Indirect address found; narp= ar$ar &
operation= $add_sub_ar{
$incr_oper} machine code=
$opcode_generated[$j].
$non_opcode_generated[$j] found in line',
$line_number[$i-1]."\n";

else {
$error_count++;
print 'ERROR : Invalid Instruction $code[$i] in line '.$line_number[$i].'
\n';
}

else {
$error_count++;
print 'ERROR : Invalid Instruction $code[$i] in line '.$line_number[$i].'
\n';
}
}
else {
$error_count++;
print 'ERROR : Invalid Instruction $code[$i] in line '.$line_number[$i].'
\n';
}

else if ( exists $opcode_3bit{$opcode_extracted[$i]} ) {
$opcode_generated[$j] = $opcode_3bit{$opcode_extracted[$i]};

if ( $non_opcode_extracted[$i] =~ /0x(\[0-9A-F]\[0-9A-F\])\[s\]*\[s\]*(\d+)\[s\]*$/i ) {

$dir_addr = $1;
$mode = '0';
$shift = $2;
$non_opcode_generated[$j] = sprintf ( '%05b',
$mode,$shift).$mode.sprintf ( '%07b', hex($1));

# print 'Direct address $1 found; machine code= $opcode_generated[$j].
$non_opcode_generated[$j] found in line '.$line_number[$i-1].'\n';

else if ( $non_opcode_extracted[$i] =~ /\[s\]*ar\(\.*\)\[s\]*\[s\]*(\.*\)\[s\]*\$i/ ) {

$incr_oper = $2;
$shift = $3;
$ar = $1;
$mode = '1';

if ( ( exists $add_sub_ar{$incr_oper} )&& ( $ar <= 7 ) ) {
$non_opcode_generated[$j] = sprintf(
    "%05b", $shift).$mode.
    $add_sub_ar{$incr_oper}.'00'.
    sprintf("%03b", $ar);

# print "Indirect address found;
    narp= ar$ar & operation=
    $add_sub_ar{$incr_oper} machine
    code= $opcode_generated[$j].
    $non_opcode_generated[$j] found
    in line ".$line_number[$i-1]."\n
;

else {
    $error_count++;
    print "ERROR : Invalid Instruction
        $code[$i] in line ".$line_number[$i]."\n";
}

} else { 
    $error_count++;
    print "ERROR : Instruction $code[$i] in line ".$line_number[$i]."\n";

}

else{
    $error_count++;
    print "ERROR : Opcode doesn't exist $code[$i] in
        line ".$line_number[$i]."\n";
}

# print $opcode_extracted[$i]."\t",
    $non_opcode_extracted[$i]."\t", $opcode_generated
    [$j]."\t", $non_opcode_generated [$j]."\n";

$j++;

elsif (exists $opcode_16bit{uc($code[$i])}){
    $opcode_generated[$j] = $opcode_16bit{uc($code[$i])};
    $j++;
}
else{
    $error_count++;
    print "ERROR : Instruction $code[$i] in line ".$line_number[$i]."
        DOES NOT EXIST\n";

}

# Jump address generation

############################################################

#non_opcode_extracted[0x0a56] = "00110000"
I.2 Assembler designed in Perl

```perl
foreach my $k (@jmp_labels_called){
    if (exists $jmp_in_label_extracted{$k}){
        my @temp_arr = @{$jmp_out_label_extracted{$k}};
        # print 'Label $k found in: $jmp_in_label_extracted{$k}\n';
        foreach my $jmp_addr ( @temp_arr ){
            $jmp_addr_generated{ $jmp_addr } = sprintf ('%016b ', $jmp_in_label_extracted{$k});
            $jmp_addr_generated_nonconverted{ $jmp_addr } = 'JA ='. $jmp_in_label_extracted{$k};
        }
    }
}
if ( $error_count == 0 ){
    #
    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

    open ( my $out_file , '>',$mif_file ) or die 'Error: Output MIF file not found and can't be created!\n';
    print $out_file 'WIDTH = 16;\nDEPTH = 65536;\nADDRESS_RADIX = DEC; % Can be HEX, BIN or DEC\nDATA_RADIX = BIN; % Can be HEX, BIN or DEC\n\nCONTENT BEGIN
    
    for (my $i =0 ; $i < $j ; $i++){
        my $int = $opcode_generated[$i].$non_opcode_generated[$i].$jmp_addr_generated[$i];
        $int = unpack('N' , pack('B32', substr('0' x 32 . $int , -32)) );
        my $hex = sprintf('%04x ' , $int);
        print $out_file $i . ': ' . $hex . ':'. $code_rearr[$i]. $jmp_addr_generated_nonconverted[$i].'\n';
    }
    print $out_file 'END; ';
    close($out_file);
    print '\nASSEMBLY SUCCESSFUL : Successfully assembled $assembly_file\n\nPlease check $mif_file for the output.';

    #
    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

    open( my $hex_file_out , '>',$hex_file ) or die 'Error: Output HEX file not found and can't be created!\n';
    for (my $i =0 ; $i < $j ; $i++){  
```
```
my $int = $opcode_generated[$i] . $non_opcode_generated[$i] .
    $jmp_addr_generated[$i];

    $int = unpack("N", pack("B32", substr("0" x 32 . $int , -32)))
    ;

    my $hex = sprintf("%04x", $int );

    print $hex_file_out $hex . "\n";

} else {
    print "\nASSEMBLY FAILED : Total of $error_count errors found. Please 
    check your code\n" ;
}
I.3 Assembly source code for testing and median filter

I.3.1 Assembly code used for basic level testing

```assembly
lack 0x01; // acc=1 arp=x ar0=0 ar1=0
sac 0x00, 0; // acc=1 arp=x ar0=0 ar1=0 mem[0]=1
lack 0x03; // acc=3 arp=x ar0=0 ar1=0 mem[0]=1
lark ar0, 0x00; // acc=3 arp=0 ar0=0 ar1=0 mem[0]=1
add ++,0,ar0; // acc=4 arp=0 ar0=1 ar1=0 mem[0]=1
sac --,0,ar0; // acc=4 arp=0 ar0=0 ar1=0 mem[0]=1
        mem[1]=4
sub ++,0,ar1; // acc=3 arp=0 ar0=0 ar1=0 mem[0]=1
        mem[1]=4
lark ar1, 0x10; // acc=3 arp=1 ar0=1 ar1=10 mem[0]=1
        mem[1]=4
```
sac *,0, ar0; // acc= 3  arp=1   ar0=1   ar1=10  mem[0]=1 ,
lt *,ar1;   // acc= 3  arp=0   ar0=1   ar1=10  mem[0]=1 ,
   mem[1]=4,mem[10]=3 ; Treg=4
mpy *, ar0; // acc= 3  arp=1   ar0=1   ar1=10  mem[0]=1 ,
   mem[1]=4,mem[10]=3 ; Treg=4  Preg=C
mac *-, ar0; // acc= f  arp=0   ar0=1   ar1=10  mem[0]=1 ,
   mem[1]=4,mem[10]=3 ; Treg=4  Preg=f (acc=c+acc ; Preg=4*mem[ ar0])
mpyk 0x02; // acc= f  arp=0   ar0=1   ar1=10  mem[0]=1 ,
   mem[1]=4,mem[10]=3 ; Treg=4  Preg=8
pac ;
L1: sub *,0, ar0;
bnz L1;
bz L2;
L3: or *, ar1;
xor *, ar1;
ret;
L2: call L3;
nop;
//
I.3.2 Assembly code used for median filter algorithm

LAR AR0, 0x0003;
LAR AR1, 0x0004;
LAR AR2, 0x0005;
LAR AR3, 0x0006;
LAR AR4, 0x0007;
LAR AR5, 0x0008;
LAR AR7, 0x0009;
LARP AR7;
LAC 0x0000, 0;
SAC AR7, *−, 0;
LACK 0x01;
SAC AR7, *+, 0;
MAR AR7, *+;
LAC 0x0001, 0;
SAC AR7, *+, 0;
LACK 0x01;
SAC AR7, *−, 0;
MAR AR0, *−;
L1: LAC AR0, *+, 0;
CMPSIMD AR0, *−, G;
PUSH;
PUSH;
PUSH;
LAC AR0, *+, 0;
CMPSIMD AR0, *+, L;
PUSH;
CMPSIMD AR3, *-, G;
SAC AR0, *, 0;
POP;
CMPSIMD AR5, *, L;
SAC AR3, *-, 0;
POP;
CMPSIMD AR3, *-, G;
SAC AR3, *+, 0;
POP;
CMPSIMD AR3, *, L;
SAC AR1, *+, 0;
LAC AR1, *+, 0;
CMPSIMD AR1, *-, G;
PUSH;
PUSH;
LAC AR1, *+, 0;
CMPSIMD AR1, *+, L;
PUSH;
CMPSIMD AR4, *-, G;
SAC AR1, *, 0;
POP;
CMPSIMD AR3, *, L;
SAC AR4, *, 0;
POP;
CMPSIMD AR4,*−,G;
SAC AR4, *+, 0;
POP;
CMPSIMD AR4,* ,L;
SAC AR2, *+, 0;
LAC AR2, *+, 0;
CMPSIMD AR2,*− ,G;
PUSH;
PUSH;
LAC AR2, *+, 0;
CMPSIMD AR2, *+ ,L;
PUSH;
CMPSIMD AR4,*−,G;
SAC AR2, *, 0;
POP;
CMPSIMD AR5,* ,L;
SAC AR4, *−, 0;
POP;
CMPSIMD AR5,* ,G;
SAC AR4, *+, 0;
POP;
CMPSIMD AR4,* ,L;
SAC AR3, *−, 0;
LAC AR5, *−, 0;
CMPSIMD AR5,*,+ ,G;
CMPSIMD AR5,*,− ,G;
SAC AR3, *,−, 0;
LAC AR4, *, 0;
CMPSIMD AR3,* ,G;
PUSH;
LAC AR4, *, 0;
CMPSIMD AR4,*,+,L;
CMPSIMD AR4,*,−,G;
SAC AR4 *, 0;
POP;
CMPSIMD AR4,*,L;
SAC AR3, *,−, 0;
LAC AR4, *,+, 0;
CMPSIMD AR5,*,+,L;
CMPSIMD AR3,*,+,L;
SAC AR3, *, 0;
LAC AR4, *, 0;
CMPSIMD AR3,*,G;
PUSH;
LAC AR4, *, 0;
CMPSIMD AR5,*,L;
CMPSIMD AR4,*,G;
SAC AR4 *, 0;
POP;
I.3 Assembly source code for testing and median filter

CMPSIMD AR6,*,L;
SAC AR7, *+, 0;
LAC AR7, *−,0;
SUB AR7, *+,0;
SAC AR0, *,0;
BNZ L1;
MAR AR0, *+,0;
MAR AR1, *+,0;
MAR AR1, *+,0;
MAR AR2, *+,0;
MAR AR2, *+,0;
MAR AR7,*+,0;
LAR AR7, 0x0000;
MAR AR7, *++; L2: LACK 0x00;
SAC AR6,*+,0;
BU L2;