

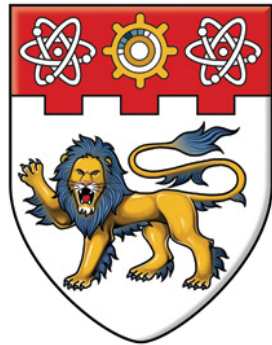
NANYANG
TECHNOLOGICAL
UNIVERSITY

**UNDERSTANDING AND
PROFILING A CNN APPLICATION
ON DIFFERENT PLATFORMS
USING OPENCL**

by
Shuvam Nandi
U1322990B

School of Computer Science and Engineering

27th March, 2017



**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SCE16-0101
Understanding and Profiling a
Convolutional Neural Network Application
on Different Computing Platforms using
OpenCL

by
Shuvam Nandi

Submitted in Partial Fulfillment of the Requirements for the Degree of
Bachelor of Computer Engineering of the Nanyang Technological
University

School of Computer Science and Engineering

27th March, 2017

Abstract

Acknowledgment

First and foremost, I would like to offer my sincerest gratitude to my supervisor, Assoc Prof Douglas Maskell, for ensuring that this project was a valuable and enriching experience in my final year.

Moreover I would also like to thank Abhishek Kumar Jain for his professional guidance, continuous support, effective suggestions, constructive criticism and timely help. I am also thankful to him for carefully reading and commenting on countless revisions of this report.

I am grateful to Prashant Ravi, an MSc student under Prof. Douglas, for his professional guidance, continuous support, and teachings in the beginning, which were essential for me to understand important concepts and topics driving the project.

Thanks to Mr. Jeremiah Chua in Hardware and Embedded Systems Lab (HESL) for his technical support and the facilities.

I would also like to extend my thanks to my examiner, Dr. Sharad Sinha, for taking out time to evaluate my project.

The list of acknowledgements will not remain complete without mentioning the encouragement and motivation given by my family and friends throughout the year.

Contents

1	Introduction	2
1.1	Motivation	2
1.2	Organization	2
2	Background	3
3	Literature Review	4
4	The OpenCL Programming Model	5
4.1	Introduction	5
4.1.1	Platform Model	6
4.1.2	Memory Model	7
4.1.3	Execution Model	8
4.1.3.1	Execution Strategy	9
4.1.3.2	Host API	10
4.1.4	Programming Model	12
4.1.4.1	Data Parallel	12
4.1.4.2	Task Parallel	13
4.2	Experiments	13
5	Convolutional Neural Networks	16
5.1	Working	16
5.2	MNIST Database	17
5.3	Experiments	17
6	RISC-V	20
6.1	Introduction	20
6.1.1	About	21

6.1.2	ISA Specifications	22
6.2	Software Tools and Setup Required	23
6.2.1	Software Tools	23
6.2.1.1	GNU Toolchain	24
6.2.1.2	Front End Server	24
6.2.1.3	Proxy Kernel	24
6.2.1.4	ISA Simulator	25
6.2.1.5	Opcodes	25
6.2.2	Setup Required	25
6.2.2.1	Toolchain	26
6.2.2.2	Testing	27
6.3	PicoRV32	28
6.3.1	About	28
6.3.2	Setup Required	29
6.3.2.1	Toolchain	29
6.3.2.2	PicoRV32 Source Code	30
6.3.2.3	Icarus Verilog	30
6.3.3	Execution of Simple C Programs	31
6.4	Implementation of RISC-V Processor	33
6.4.1	Objective	33
6.4.2	Design	33
6.4.2.1	Requirements	33
6.4.2.2	Components	34
7	Conclusions and Future Work	37
7.1	Conclusions	37
7.2	Future work	37
	Appendix A OpenCL	38
	Appendix B RISC-V	39

List of Figures

4.1	Platform Model in OpenCL	6
4.2	Memory Model in OpenCL [3]	8
4.3	C and OpenCL Function Calls [3]	9
4.4	Handling multi-dimensional work-group sizes	10
4.5	NDRange multi-dimensional index space example	11
6.1	Base instruction formats in RISC-V	22
6.2	Software stack for RISC-V tools	24
6.3	Hello World program execution on Spike	28
6.4	Successful execution of PicRV32 testbench	31
6.5	Formats for RISC-V instructions to be implemented	35
6.6	Symbolic representation of an ALU	36

List of Tables

4.1	Execution time (in μs) of matrix multiplication on AMD Radeon HD 8730M GPU	14
4.2	Execution time (in μs) of matrix multiplication on NVIDIA GeForce 405 GPU	14
5.1	Execution time (in μs) for different operations in MNIST Application on different devices	18

Chapter 1

Introduction

1.1 Motivation

Silicon technology will continue to provide an exponential increase in the availability of raw transistors.

Shuvam Nandi

1.2 Organization

The remainder of the report is organized as follows: Chapter 2 presents background information on Chapter 3 studies current state of the art overlays and techniques for placement and routing. Chapter 4 Chapter 5 throws light Chapter 6 shows We thereby conclude in chapter 7 and discuss future work.

Chapter 2

Background

Chapter 3

Literature Review

Chapter 4

The OpenCL Programming Model

OpenCL is a framework for writing application programs that run across heterogeneous platforms, comprising Graphics Processing Unit (GPU)s, Central Processing Unit (CPU)s, Digital Signal Processor (DSP)s, Field Programmable Gate Array (FPGA)s, and other processors or hardware accelerators[1]. A framework suited for parallel programming, it has been standardized by the Khronos group, made up of companies like AMD, Apple, Intel, NVIDIA, Qualcomm, Sony, Xilinx, etc. [2].

4.1 Introduction

OpenCL is truly the first open programming standard for general-purpose computations on heterogeneous systems, allowing programmers to target their source code on multi-core CPUs, GPUs, and other devices. It specifies a programming language (centered on the C99 standard) for programming these devices and provides Application Processing Interface (API)s, to control the platform and execute programs on various computing devices. OpenCL is vendor independent and thus not specialized for any device. Therefore, OpenCL is a powerful way to write parallel programs capable of running on a wide range of devices.

The framework includes a language, API, libraries and a runtime system to support software development. The architecture of OpenCL can be de-

scribed using a hierarchy of models [3]:

- Platform Model
- Memory Model
- Execution Model
- Programming Model

4.1.1 Platform Model

In this model, a host device is connected to one or more OpenCL devices, which is divided into one or more compute units (CUs), further divided into one or more processing elements (PEs). All computations on a device are executed on the PEs [4].

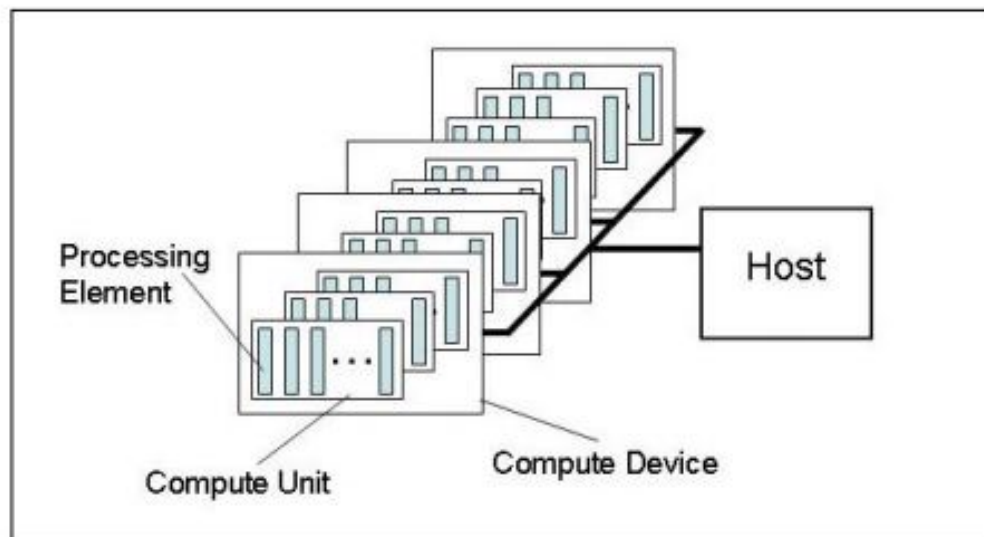


Figure 4.1: Platform Model in OpenCL

The host device holds the OpenCL application running, from where the commands are submitted to the device, to be executed on the PEs within. The PEs inside a CU execute a single set of instructions as Single Instruction Multiple Data (SIMD) units in parallel.

4.1.2 Memory Model

Memory directly available to kernels executing on OpenCL devices is known as device memory. The device memory consists of the four distinct memory regions as follows:

- **Global Memory:** This memory region allows read/write access to each work-item in all the work-groups executing on any device inside a context. Work-items are capable of reading from or writing to any part of memory objects. Accesses to global memory may be cached, depending on the device capabilities.

Global memory is shared with all processing elements, and is also available to the host. The host utilises this memory to copy data to/from the device.

- **Constant Memory:** This memory region contains content that remains constant throughout the execution of a kernel.

Constant memory is also shared between all processing elements, but it is read-only. It provides an efficient way to share data with all processing elements.

- **Local Memory:** A region of memory local to a work-group. This region can be used to allot variables that are common to all work-items within the work-group.

Each CU has its own local memory, only shared with the processing elements within the compute unit. It cannot be accessed by other compute units.

- **Private Memory:** A region of memory private to a work-item. Variables declared in one work-item's private memory are not seen to another work-item.

Private memory belongs to a sole processing element. Each processing element has its own private memory, thereby unable to access all other private memories within or outside the CU.

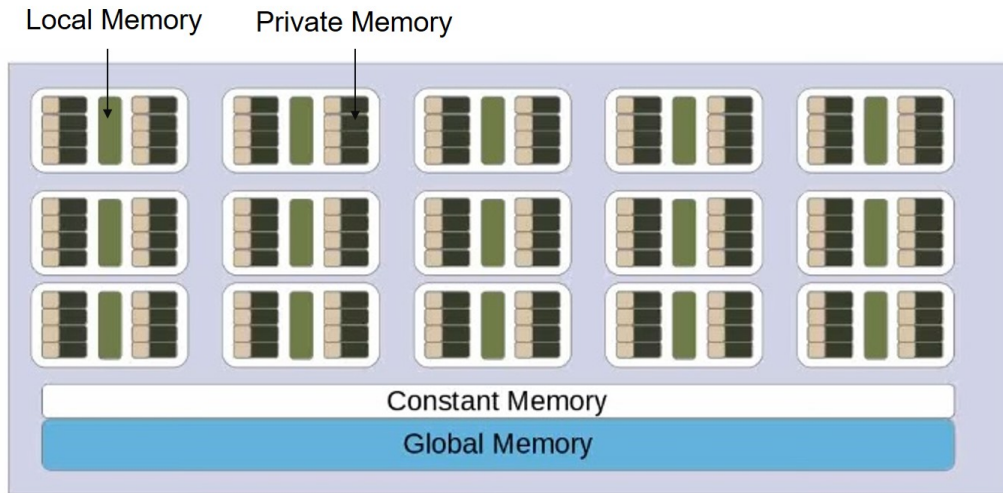


Figure 4.2: Memory Model in OpenCL [3]

Global memory is the only memory which is persistent between kernel calls. Constant, local, and private memories are just scratch spaces, which get reset after every kernel call.

4.1.3 Execution Model

The core of the OpenCL execution model is based on how kernels execute on the device. The execution of an OpenCL program happens in two parts: kernels that execute on the OpenCL devices, and a host that runs on the host device. The host is responsible for executing kernel functions on the device [4]. These are ordinary functions, albeit with special signatures. The kernel call comprises two parts: an ordinary argument list, and external execution parameters that control the parallelism. OpenCL provides direct support for parallel computing.

The host coordinates the execution of the kernels on the OpenCL device. It does not get involved in the computations itself; it only provides execution parameters to launch the kernel and the arguments which the kernel requires for carrying out the operations.

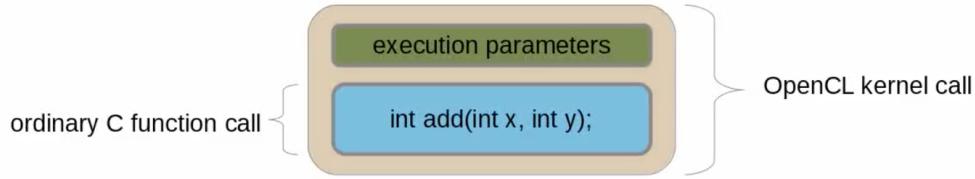


Figure 4.3: C and OpenCL Function Calls [3]

4.1.3.1 Execution Strategy

The kernel function will be invoked many times, the argument list being identical for all the invocations. This means that the same function gets called repeatedly, based on the execution parameters specified by the host prior to launching the kernel.

There exists an index space provided by an N-dimensional index space. The index space supported in OpenCL is called an NDRange. It can be one, two or three dimensional. Each function invocation can access its index; this is how it is identified. A work-item is a kernel invocation for a particular index, and the global id is a unique identity for every work-item belonging to the index space. Every work-item is provided with a global ID. The global work size is the number of work-items in every dimension. The work dimension is the dimension of the index space.

From the Device Model, the processing elements are the ones which execute the instructions. To be able to execute the kernel multiple times on the device, the work-item needs to be mapped to Processing Elements (PEs). Multiple work-items are assigned to each PE, so that the case where number of PEs is less than the global work size can be taken care of.

Now, every Compute Unit (CU) has a dedicated local memory which is shared by all PEs within. Local memory provides huge advantages in performance. Thus, work-items must be mapped to the PEs in such a way that a group of them can access the local memory of a CU for effective execution. The partitioning of the global work size into smaller pieces leads to the formation of work-groups. The work-groups provide a more coarse-grained decomposition of the index space. Work-groups are assigned a unique work-group ID with

the same dimension as the index space used for the work-items. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID [4].

Work-groups execute on CUs and share their local memory. All the work-items in the work-group share local memory and are mapped to PEs within a CU. These work-items are capable of identifying which work group they are parts of, work-group ID, size of work-groups, their global IDs and the global work size.

The work-group size has a device specific physical meaning. The maximum work-group size is a device characteristic. This can be determined by querying the device. Also, it is an integer value; thus, n-dimensional work-groups have to be handled in a special way. Work-groups are launched by the host device.

The device particular work-group size is a scalar, but work-groups can have multiple dimensions. For example, if the maximum work group size is 32, work-groups could be launched in 3 dimensions like (8, 2, 2). The following must be true in order to launch the work-groups successfully:

$$\text{Work-group size} = (w_1, w_2, w_3, \dots, w_k)$$

$$w_1 * w_2 * w_3 * \dots * w_k \leq \text{max work-group size}$$

Figure 4.4: Handling multi-dimensional work-group sizes

The figure below shows an example of how the local size, number of work-groups and global size vary for different dimensions of `NDRange`:

4.1.3.2 Host API

The host must call certain APIs in order to execute the kernels, which take care of the following:

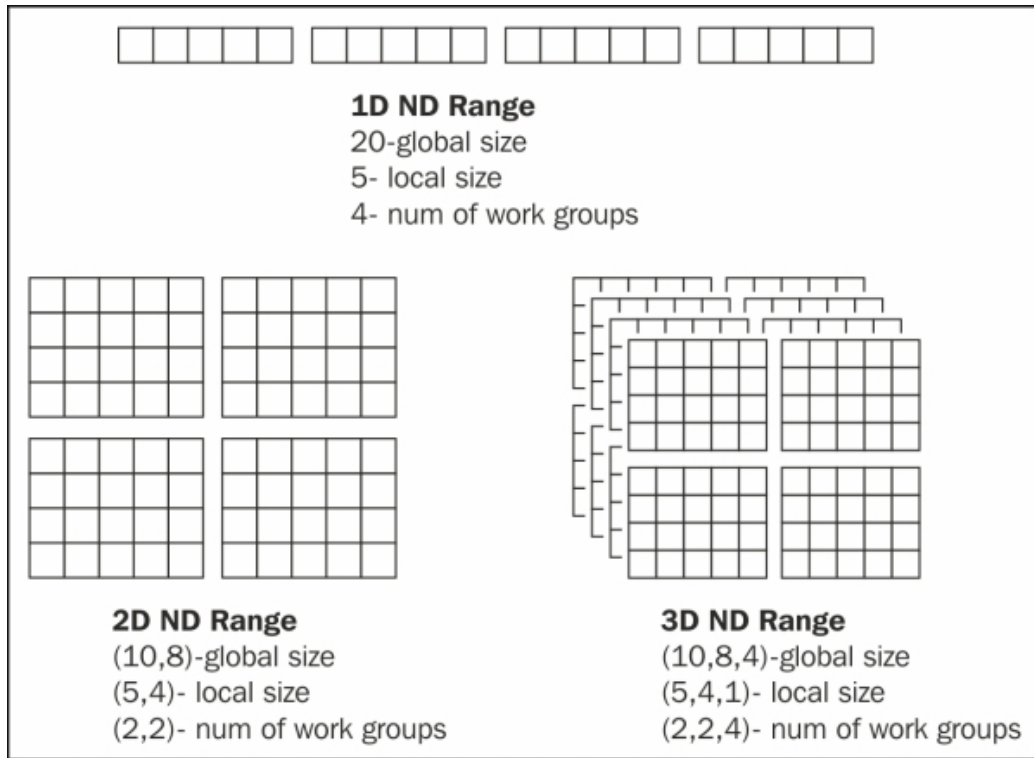


Figure 4.5: NDRange multi-dimensional index space example

- Platform
 A platform is an implementation of OpenCL. These are the drivers for specific devices which support the execution of OpenCL kernels. Platforms enables supported devices to be exposed to the programmer. For example, if a system consists of 1 AMD GPU, 1 NVIDIA GPU and an Intel Xeon Phi CPU, all capable of running OpenCL code, there will be three platforms corresponding to each of these devices. The platform is used to discover devices available.
- Context
 A context is created by the host for each platform on which the kernel executes. Thus, a context cannot have multiple platforms. A context acts as a container for devices and memory. Most kernel operations are related to a context.
- Program

A program is a collection of kernels. A program is loaded by the host device and kernels are extracted from the program to be invoked. The host could either directly compile the OpenCL C code or load a binary representation. Programs are device specific.

- **Asynchronous Device Calls**

The host manages devices asynchronously. Host issues commands to the device, telling the device to perform some work. Devices act as slaves and do as the instructed command. The host waits for the commands to complete, meaning that the device has completed the action. Commands can have dependencies on other commands. OpenCL commands are issued by `clEnqueue*` calls. A `cl_event` returned by `clEnqueue*` calls is used for dependencies.

4.1.4 Programming Model

The OpenCL execution model supports data parallel and task parallel programming models. In OpenCL, the primary model is data parallel.

4.1.4.1 Data Parallel

Data parallel programming model (equivalent to SIMD) defines computations in terms of a sequence of instructions applied to multiple elements of a dataset. In a strict data parallel model, a one-to-one mapping exists between a work-item and an element in the dataset over which a kernel can be executed in parallel. However, a strict one-to-one mapping is not required in OpenCL.

A hierarchical data programming model exists in OpenCL, where the hierarchical subdivision can be achieved either explicitly or implicitly. In the explicit model, the total number of work-items to execute in parallel and how work-items are divided among work-groups must be defined by the programmer. On the other hand, the implicit model only requires the total number of work-items to execute in parallel to be defined, while the division into work-groups is taken care by the OpenCL implementation.

4.1.4.2 Task Parallel

Task parallel programming model in OpenCL defines a model in which a single instance of a kernel is executed independent of any index space. It is equivalent to executing a kernel on a compute unit with a work-group containing a single work item. It is the simultaneous execution of many different tasks across the same or different memory objects.

4.2 Experiments

A few simple experiments were conducted on different devices – an AMD GPU and an NVIDIA GPU, to see how performance varied on using the same OpenCL kernel code but with different execution parameters. A small kernel which performs 2-dimensional matrix multiplication was written in OpenCL and was launched with different work-group sizes on the specific platform to see how its execution varied.

The host device selects the platform and the device, creates the context and command queue, initializes the matrices, creates memory buffers in the device, and transfers the data to the device memory. The two input matrices and one output matrix form the data passed to the kernel function as arguments, on which the matrix multiplication was performed in parallel. The execution is controlled by the `clEnqueueNDRangeKernel` parameters, which define the work-group size by specifying the local size and global size.

$$\text{Work} - \text{group size} = \text{Local size} \quad (4.1)$$

The work-group size must be perfectly divisible by the specified global work size, failing which the kernel would not be launched and an error would be thrown.

The resultant matrix was stored in a buffer and transferred back to the host asynchronously. The execution time was calculated and the result was compared for correctness by running the matrix multiplication sequentially on the host device.

The following tables show the average timing results on AMD Radeon HD 8730M on platform OpenCL 2.0 AMD-APP (1800.8) and NVIDIA GeForce 405 on OpenCL 1.1 CUDA 4.2.1.

Table 4.1: Execution time (in μ s) of matrix multiplication on AMD Radeon HD 8730M GPU

2-D Matrix Multiplication Timings on AMD GPU					
Global Size Local Size	128	256	512	1024	2048
Not Specified	6121.25	48282.5	401073.58	3270198.86	16901530.39
2	8400.23	67250.57	530250.24	1082663.32	5365565.08
4	2181.05	16945.09	142875.45	298480.89	1703353.98
8	3359.17	25594.24	206148.51	265741.50	1362683.18
16	6123.93	48253.26	385895.55	291104.51	1243956.77
32	Error	Error	Error	Error	Error
64	Error	Error	Error	Error	Error
128	Error	Error	Error	Error	Error

Table 4.2: Execution time (in μ s) of matrix multiplication on NVIDIA GeForce 405 GPU

2-D Matrix Multiplication Timings on NVIDIA GPU					
Global Size Local Size	128	256	512	1024	2048
Not Specified	6138.05	48312.10	401356.14	5773783.80	Error
2	10721.54	58040.95	621084.80	4527345.84	Error
4	3488.40	18654.32	232133.67	1776133.69	Error
8	2969.30	17499.01	229669.94	1729808.40	Error
16	8382.89	34673.16	486371.19	3369363.59	Error
32	Error	Error	Error	Error	Error
64	Error	Error	Error	Error	Error
128	Error	Error	Error	Error	Error

Table 4.1 and Table 4.2 show the execution time (in μs) taken by the two-dimensional matrix multiplication kernel in OpenCL on AMD and NVIDIA GPUs, for different work group sizes (denoted by local size). The global work-size was increased from 128 to 2048 and the execution times were measured and noted.

As observed in the results, the AMD Radeon 8730M GPU overpowers the NVIDIA GeForce 405 if their best execution times are compared for different global sizes. Both the devices follow a similar trend for maximum performance (or least execution time). For both the devices, there exists a trend of having a least execution time at a particular work-group size, beyond which the increasing the local work size leads to higher execution times (lower performance). In the case of AMD Radeon 8730M, the best performance is achieved when the work-group size is 4, while for NVIDIA GeForce 405 it is 8. These best performing work-group size values are device specific, as seen from the results.

Upon further increasing the work-group size, it was noticed that the execution of OpenCL kernel was failing due to error in launching it. This work-group size was 32 for both AMD Radeon 8730M and NVIDIA GeForce 405. This falls in line with the Section 4.1.3.1 Execution Strategy, which mentions that the maximum work-group size is device specific. Thus, we can conclude that the maximum work-group size for both the GPUs is 16.

Chapter 5

Convolutional Neural Networks

Convolutional Neural Network (CNN) is a classification of multi-layered neural networks [5]. CNNs are quite similar to regular neural networks – they are composed of neurons possessing learnable weights and biases; neurons get inputs, perform dot products and follow it up with non-linearity (optionally). The network expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. There is a loss function on the last, fully-connected which generates the final output. The difference exists for the type of input CNN architectures explicitly assume, which are images, allowing certain properties to be encoded into the architecture. These then make the forward function more efficient to implement and hugely decrease the number of parameters in the network [6].

CNNs have excellent performance in areas such as image recognition and classification. These are widely used in the detection and identification of faces, objects and animals, apart from powering robot vision and driverless cars [7]. LeNet-5 is a convolutional network designed for handwritten and machine-printed character and digit recognition.

5.1 Working

Hello World

5.2 MNIST Database

The Mixed National Institute of Standards and Technology (MNIST) database is an enormous database of handwritten digits which is frequently utilized for training different image processing procedures [8]. It is also extensively used for training and testing in the field of machine learning.

The MNIST database consists of 60,000 training and testing images [9]. Samples from National Institute of Standards and Technology's (NIST) original training and testing dataset were remixed to generate the MNIST database. The digits have been size-normalized and centered in a fixed-size image of 20x20 while preserving their aspect ratio [10]. The resulting images contain grey levels because of anti-aliasing methods used by the normalization algorithm. The images were centred in a 28x28 image by calculating the centre of mass of the pixels and rendering the image to place this point at the centre of the 28x28 field.

The database is a good starting point for individuals who desire to learn machine learning techniques and pattern recognition methods on real-world data while spending marginal efforts on preprocessing and formatting. When one learns how to program, there's a tradition to print "Hello World" in his first program. Similar to how programming has Hello World, machine learning has MNIST.

A number of scientific papers have been written to attempt achieving the lowest error rate in predicting the digits – one paper manages a rate as low as 0.23 percent using a hierarchical system of convolutional neural networks.

5.3 Experiments

An existing real word application was obtained and executed to further gauge the performance capabilities of OpenCL. An MNIST handwritten digits prediction application written by Gopala Krishna Hegde based on LeNet-5 convolutional neural networks was cloned from GitHub for this purpose [11].

The MNIST handwritten digit recognition application was executed on different devices - two CPUs and one GPU, and average timings were calculated

for different parts in the execution of kernels. There is a total of eight kernels which execute on the device, launched by the host one by one. The kernels launched have a work dimension of 3 and their global sizes are (12, 12, 20).

The following results were obtained on an Intel(R) Core(TM) i5-5200U CPU, Intel(R) Xeon(R) CPU E5-1650, and NVIDIA Quadro 600 GPU.

Table 5.1: Execution time (in μ s) for different operations in MNIST Application on different devices

MNIST Application Execution Times			
Device	i5-5200U CPU	E5-1650 CPU	Quadro 600 GPU
Initialize Application	0.9	1.3	1.5
Allocate Host Memory	16.2	16.9	16.5
Initialize Device	62198.5	78408.9	36985.2
Build Kernel	132807.2	130634.2	16914.6
Allocate Device Memory	833.2	1025.2	1364.7
Kernel Execution	99.1	394.7	3903.5

The results above in Table 3 show the time taken on three different processors for initializing the application, allocating memory in the host, initializing the device on which the kernels will execute on, building the kernels, allocating memory on the device, and the execution of all kernels.

From the results, it is evident that both the CPUs, i5-5200 and Xeon E5-1650 take much lesser time than the NVIDIA Quadro 600 GPU in the kernel execution – their performance is 10.6 and 9.89 times better than the GPU’s. This could be reasoned by making the point that OpenCL parallelizes the code on both the CPUs and GPU, but the execution is not parallel enough to exploit the full usage of the GPU’s performance capabilities. Also, the kernel execution time is for the completion of all the kernels, between which data transfers take up more time as compared to the CPU. The host and the target being CPU and GPU, respectively, increase the communication bottleneck drastically and leads directly to having a greater execution time for the GPU. In case of the CPUs, the communication requires hardly any time, leading to a smaller execution time.

To further understand this, the timings for the completion of individual kernels were measured. It was shown that the CPUs were only 2 times faster than the GPU on an average, as compared to 10 times for the aggregated execution time for the eight kernels. These experiments conclude that the GPU execution is much faster if the communication bottleneck can be avoided.

Chapter 6

RISC-V

6.1 Introduction

The performance derived from a computer and the amount of energy that is consumed in the process hugely depends on algorithms, application code, compiler, microarchitecture, Instruction Set Architecture (ISA), physical design, circuit design and fabrication process. This is the order of abstraction from a user working on the computer, starting from the top most layer exposed to programmer, down to the bottom most layer which depends on how the chip is fabricated.

The Central Processing Unit (CPU) is one of the most significant parts of any computing device, as it is the brain of the computer [12]. It performs arithmetic and logical operations and moves data from one place to another. This is done by the processor according to ISA it is based on. An ISA is a well-defined interface linking computer software to the hardware its running on, enabling the independent development of the two computing realms. It is strictly an interface specification, not an implementation.

In today's age, technology has been able to evolve much more because of the existence of open standards and open source software – Linux as an open source operating system for example. This builds the case for an ISA which is not proprietary and is powerful enough to drive custom processing chips.

Historically, ISAs have been proprietary for business reasons, but there is

no good technical reason for the absence or deficiency of open source ISAs. Having a free and open source ISA specification would hugely benefit the industry, just like how open source software has added great value. It would directly lead to a rise in innovation via free-market competition, with several designers coming with various implementations, which comprise both the open source and proprietary ISA versions. More shared open core designs would exist, reducing the time taken to hit the market, lowering cost from reuse, and making the designs more transparent and error-free. Moreover, an open ISA would lead to processors becoming affordable for a greater number of devices, helping in expanding the Internet of Things (IoTs) [13].

6.1.1 About

RISC-V is an ISA originally designed to support computer architecture research and education, now reaching the stage to become a standard open architecture for industry implementations, under the governance of the RISC-V foundation. It was developed in the Computer Science Division of the EECS department at University of California, Berkeley [14]. Its development was inspired by ARM’s IP restrictions together with the lack of 64-bit addresses.

RISC-V has learned from various mistakes in different ISAs, making its design superior to other ISAs incorporating a better mix of capabilities [13]. It provides a small core instruction set that compilers and operating systems can depend on, as well as optional standard extensions. It has a compact instruction set encoding – smaller code is desirable as it reduces the memory requirement. It offers single, double and quadruple precision floating point capabilities, in addition to being 32, 64 and 128-bit addressable.

RISC-V was defined with a goal to be completely open and freely open to academic researchers and industries. Designed with small, fast and low-power real-world implementations in mind, it does not over-architect for a particular style of microarchitecture. It is a real ISA suited for direct native hardware implementation, not for just simulation or binary translation. It provides support for the revised 2008 IEEE-754 floating-point standard [15]. RISC-V simplifies experiments with new supervisor-level and hypervisor-level ISA designs.

RISC-V’s capabilities make it extremely powerful, yet simple and clean to

implement. In contrast to most ISAs, it is available for all types for use, empowering anyone to design, manufacture and sell RISC-V chips and software. It supports small embedded systems, Personal Computers (PCs), supercomputers with vector processors, and warehouse-scale rack mounted parallel computers. Thus, the widespread usage of RISC-V would allow software developers to target an open hardware target, while making commercial processor designers compete on implementation superiority.

6.1.2 ISA Specifications

RISC-V is specified as a base integer ISA, which is a must in any implementation, with additional extensions to the base ISA. Based on the width of the integer registers, the base instruction set has two variants, RV32I and RV64I. There is scope for a future variant RV128I for 128-bit address support [16]. It provides support for extensive customization and specialization, along with extension beyond the base instruction set, but doesn't allow redefining these base integer instructions.

There are four core instruction formats defined in the base instruction set, as shown follows:

0000000	rs2	rs1	000	rd	0110011	ADD
imm[11:0]		rs1	000	rd	0010011	ADDI
0000001	rs2	rs1	000	rd	0110011	MUL
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[20:10:11:19:12]				rd	1101111	JAL
imm[12:10:5]	rs2	rs1	001	imm[4:1:11]	1100011	BNE

Figure 6.1: Base instruction formats in RISC-V

Furthermore, there are two variations of the instruction formats (SB/UJ) based on how immediate values are taken care of. A few sets of standard extensions have been pre-defined to cater to general software development. These comprise integer multiplication/division (extension “M”), atomic operations (extension “A”), single-precision floating-point (extension “F”), and double-precision floating-point (extension “D”). A combination of these 4 standard extensions (denoted by “IMAFD”) is given a general abbreviation

of “G”, which results in a general-purpose scalar instruction set. RV32G and RV64G are currently set as default targets of the RISC-V compiler toolchains. The base set and extensions have been kept separated to have a simple instruction set.

The base integer instruction set for the 32-bit and 64-bit variants (RV32I/RV64I) comprises 40 instructions, while the general-purpose instruction set (RV32G/RV64G) supports 122 instructions. There are 31 general purpose registers x1-x31 capable of storing integer values, register x0 being hardwired to the constant value 0. The register width is 32 bits for RV32I and 64 bits for RV64I. RISC-V is based on the load-store architecture, where arithmetic operations only function on the data stored the registers and only the load and store operations access memory. The user address space is byte-addressed and little-endian.

There also exists a reduced version of the base integer instruction set (RV32I), designed specifically for embedded systems (denoted by “E”). The major difference is that the number of registers in RV32E is reduced to 16 from 31 in RV32I. A draft proposal for the RISC-V standard compressed instruction set has been made which reduces static and dynamic code size by adding short 16-bit instruction encodings for conventional operations. Denoted by the extension “C”, it could be added to any of the base ISA variants (RV32I, RV64I, and RV128I).

6.2 Software Tools and Setup Required

6.2.1 Software Tools

Several software tools and libraries are required to be downloaded and setup before any development on RISC-V can begin. These comprise the riscv-tools infrastructure and are available on GitHub. It is a meta-repository with Git submodules containing every stable component of the RISC-V software toolchain [17]. The image below shows the software stack of riscv-tools:

	Applications			
Distributions	OpenEmbedded	Gentoo	BusyBox	
Compilers	clang/LLVM		GCC	
System Libraries	newlib		glibc	
OS Kernels	Proxy Kernel		Linux	
Implementations	Rocket	Spike	ANGEL	QEMU

Figure 6.2: Software stack for RISC-V tools

6.2.1.1 GNU Toolchain

The RISC-V GNU Toolchain is a RISC-V C and C++ cross compiler utility. It contains binutils, gcc, newlib, glibc, and Linux UAPI headers. It supports two build modes: a generic ELD/Newlib toolchain and a more sophisticated Linux-ELF/glibc toolchain. The `riscv-unknown-elf` command is used to compile a program while `riscv64-unknown-elf-gcc` is used to assemble and link with gcc/binutils.

6.2.1.2 Front End Server

RISC-V front end server library facilitates communication between the host machine and the RISC-V target device on the Host-Target Interface (HTIF). It also provides a virtualized console and disk device [18].

6.2.1.3 Proxy Kernel

RISC-V proxy kernel is responsible for servicing system calls generated by code built and linked with the RISC-V Newlib port [18]. It handles system calls like `open`, `close` and `printf`. Abbreviated as `pk` (or `riscv-pk`), it is a lightweight application execution environment that can host statically-linked RISC-V ELF binaries. It is designed to support tethered RISC-V implementations with limited I/O capability and thus handles I/O-related system calls by proxying them to a host computer [19].

6.2.1.4 ISA Simulator

RISC-V ISA simulator consists of a functional simulator known as Spike [18]. It implements a functional model of one or more RISC-V processors [20]. Since there is no proper operating system, it executes the generated binary running on top of the proxy kernel.

Spike takes the path of the binary to run as its argument, which is `pk`, located at `$RISCV/riscv-elf/bin/pk` and finds it automatically. The name of the program to be run is the argument for `riscv-pk`, which then executes the program. An interactive debug mode can be invoked in Spike with the `-d` command line flag or `SIGINT`, which enables single-stepping through instructions, setting break point conditions, printing register and memory contents, etc.

6.2.1.5 Opcodes

RISC-V Opcodes is a subcomponent of `riscv-tools` which enumerates all standard RISC-V instruction opcodes executable by the simulator, and control and status registers (CSRs). It also has a script to convert them into different formats (C, Scala, LaTeX) [21].

6.2.2 Setup Required

There are many possible combinations to pick for the different layers of the stack shown in Figure 4, but there exist 2 of the most common workflows for RISC-V software development [17]. These are as follows:

- **Spike + pk**

The use case for following this workflow are embedded or single applications.

- Distributions: None
- Compilers: clang/LLVM or GCC
- System Libraries: newlib
- OS Kernels: Proxy Kernel (pk)
- Implementations: Spike

- **QEMU + Linux**

The use case for following this workflow is Simple POSIX environment.

- Distributions: BusyBox
- Compilers: GCC
- System Libraries: glibc
- OS Kernels: Linux
- Implementations: QEMU

In this case, the former workflow was followed, the aim being familiarization with the RISC-V development phase. The selected workflow was easier to set up and took lesser time. This requires the components riscv-gnu-toolchain, riscv-fesvr, riscv-isa-sim, and riscv-pk.

The binaries generated against newlib system library will not be running on a full-blown operating system, but will require access to few crucial system calls [18]. The setup will thus require the installation of riscv-newlib. Newlib is a C library intended for embedded systems. It has the edge of not being unnecessarily complicated over Glibc, as well as having sufficient support. Also, its porting process is much simpler than that of Glibc because it only requires a few stubs of glue code. These stubs of code include the system calls that are supposed to call into the operating system you're running on.

6.2.2.1 Toolchain

The following steps were followed to obtain and compile the RISC-V toolchain sources for the above-mentioned workflow:

1. To build GCC, several other Ubuntu packages like **flex**, **bison**, **autotools**, **libmpc**, **libmpfr**, and **libgmp** are required. Obtain the required Ubuntu packages required for installation with this command:
sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc
2. Set the environment variables TOP as the directory where the tools will be installed, i.e. the current working directory:
export \$TOP = \$(pwd)

3. Clone riscv-tools repository from GitHub:
git clone https://github.com/riscv/riscv-tools.git
4. Change into the `riscv-tools` directory:
cd \$TOP/riscv-tools
5. Instruct Git to update its submodules:
git submodule update --init --recursive
6. Set the `RISCV` environment variable to point to the path where new tools will be installed:
export \$RISCV = \$TOP/riscv
7. The `PATH` environment variable also needs to be set to point to the directory specified by `RISCV`:
export \$PATH = \$PATH:\$RISCV/bin
8. After everything has been set up, run the `build` script:
./build.sh

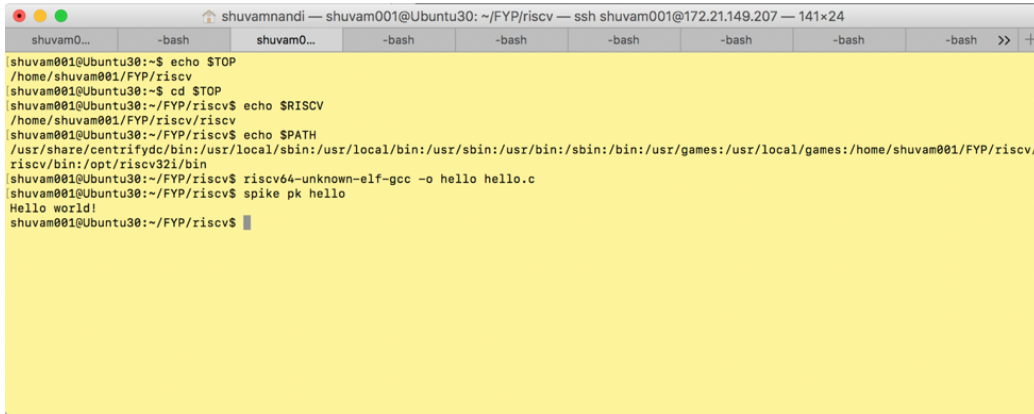
A successful build marks the end of the toolchain installation process.

6.2.2.2 Testing

After the installation procedure is finished, testing was done to make sure that there were no issues with the set up.

1. Change to the directory of toolchain installation:
cd \$TOP
2. Write a simple “Hello World” program in a C file
3. Build your program using the `riscv64-unknown-elf-gcc` command:
riscv64-unknown-elf-gcc -o hello hello.c
4. Run the binary on the ISA Simulator (Spike):
spike pk hello

The output generated should be “Hello world!”, as shown below in Figure 6.3

A terminal window titled 'shuvamnandi — shuvam001@Ubuntu30: ~/FYP/riscv — ssh shuvam001@172.21.149.207 — 141x24'. The terminal shows a series of commands and their outputs. The user sets an alias 'STOP' for the directory ~/home/shuvam001/FYP/riscv. They then navigate to this directory and run 'echo \$RISCV', which outputs the RISC-V architecture path. Next, they run 'echo \$PATH', which outputs a long list of system and user paths. Finally, they compile a C program 'hello.c' using 'riscv64-unknown-elf-gcc -o hello hello.c' and run it with 'spike pk hello', which outputs 'Hello world!'.

```
shuvam001@Ubuntu30:~$ echo $STOP
/home/shuvam001/FYP/riscv
shuvam001@Ubuntu30:~$ cd $STOP
shuvam001@Ubuntu30:~/FYP/riscv$ echo $RISCV
/home/shuvam001/FYP/riscv/riscv
shuvam001@Ubuntu30:~/FYP/riscv$ echo $PATH
/usr/share/centrifuge/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/home/shuvam001/FYP/riscv/riscv/bin:/opt/riscv32i/bin
shuvam001@Ubuntu30:~/FYP/riscv$ riscv64-unknown-elf-gcc -o hello hello.c
shuvam001@Ubuntu30:~/FYP/riscv$ spike pk hello
Hello world!
shuvam001@Ubuntu30:~/FYP/riscv$
```

Figure 6.3: Hello World program execution on Spike

6.3 PicoRV32

To understand how RISC-V is implemented in real, a processor implementation PicoRV32 was studied and analyzed. PicoRV32 is a processor core implementing the RV32IMC instruction set. It can be configured as RV32E, RV32I, RV32M, RV32C, or RV32IMC, and optionally contains a built-in interrupt controller. It is available free and is open sourced under the ISC license [22].

6.3.1 About

PicoRV32 is a small sized processor which can operate at high frequencies (250-450 MHz on 7-Series Xilinx FPGAs). The core exists in two variants, `picorv32` and `picorv32_axi`. The former providing a simple native memory interface, easy to use in simple environments, while the latter providing an AXI-4 Lite Master interface that can be easily integrated with existing systems already using the AXI standard. It provides a selectable native memory interface, optional IRQ support and optional co-processor interface. This CPU core is designed to be added as an auxiliary processor in FPGA designs and ASICs. It could be easily integrated with most existing designs without exceeding clock domains.

6.3.2 Setup Required

PicoRV32 is based on the 32-bit version of RISC-V ISA, hence requires a setup of the complete toolchain targeting a pure RV32I CPU.

6.3.2.1 Toolchain

The following steps will download the toolchain required for programs to be compiled on PicoRV32 processor:

1. To build GCC, several other Ubuntu packages like `flex`, `bison`, `autotools`, `libmpc`, `libmpfr`, and `libgmp` are required. Obtain the required Ubuntu packages required for installation with this command:
`sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc`
2. Create a new directory `/opt/riscv32i` to store the toolchain once installed:
`sudo mkdir /opt/riscv32i`
3. Change permissions of the directory created in the previous step:
`sudo chown $USER /opt/riscv32i`
4. Obtain the toolchain from the `riscv-tools` GitHub repository of RISC-V foundation and store it in the directory `riscv-gnu-toolchain-rv32i`:
`git clone https://github.com/riscv/riscv-gnu-toolchain riscv-gnu-toolchain-rv32i`
5. Change into this directory:
`cd riscv-gnu-toolchain-rv32i`
6. Checkout the specific commit version:
`git checkout 914224e`
7. Update submodules within the repository:
`git submodule update --init --recursive`
8. Create a new directory `build`:
`mkdir build`
9. Change into this directory:
`cd build`

10. Run the `configure` script, which specifies the instruction set architecture as RV32I, and the directory to install the toolchain as `/opt/riscv32i`:
`../configure --with-arch=rv32i --prefix=/opt/riscv32i`
11. Run the `make` script to build the toolchain:
`make -j$(nproc)`

6.3.2.2 PicoRV32 Source Code

Now that the pure RV32I toolchain is downloaded, the CPU implementation can be downloaded and tested.

1. Clone PicoRV32 repository from GitHub:
`git clone https://github.com/cliffordwolf/picorv32.git`
2. Run the `make` script to build the toolchain:
`cd ~/picorv32`
3. Run the `make` script in the PicoRV32 source directory to download RV32IMC toolchains to be able to run all tests:
`make -j$(nproc) build-riscv32imc-tools - RV32IMC`

6.3.2.3 Icarus Verilog

In order to test the working of the downloaded toolchain, the test bench provided by PicoRV32 must be run. This requires a software called *Icarus Verilog*, which needs to be installed. Icarus Verilog is a Verilog simulation and synthesis tool. It operates as a compiler, compiling source code written in Verilog (IEEE-1364) into some target format. For batch simulation, the compiler can generate an intermediate form called vvp assembly. This intermediate form is executed by the vvp command. For synthesis, the compiler generates netlists in the desired format [23]. The following steps will download and install Icarus Verilog:

1. Clone the GitHub repository:
`git clone https://github.com/steveicarus/iverilog.git`
2. Change into the `iverilog` directory:
`cd iverilog`
3. Update to the latest code in master branch:
`git pull origin master`

4. Install `autoconf` and `gperf` packages installed for the configuration script to work (skip if step already done):
`sudo apt-get install autoconf gperf`
5. Run `autoconf.sh` script:
`sh autoconf.sh`
6. Run `configure` script:
`./configure`
7. Run `make` command with sudo access:
`sudo make`
8. Run `make install`:
`make install`

6.3.3 Execution of Simple C Programs

After the installation of all tools required for using PicoRV32 was completed, the testbench provided was run by executing the `make` command. The following output was obtained:

```
Cycle counter ..... 281843
Instruction counter .... 63691
CPI: 4.42
DONE
Number of fast external IRQs counted: 35
Number of slow external IRQs counted: 4
Number of timer IRQs counted: 22
TRAP after 322543 clock cycles
ALL TESTS PASSED.
```

Figure 6.4: Successful execution of PicoRV32 testbench

This demonstrates that the installation was done successfully. The output is produced by different C files which are executed on the PicoRV32 CPU. The functions written in the C files are called in an assembly file `start.S`, located in `picorv32/firmware` directory. Along with C function calls, the

assembly file also contains various tests in assembly files which get executed upon running make command.

Upon further analysis of the compilation and execution flow, it was observed that the Icarus Verilog compiler compiles the *testbench.v* and *picorv32.v* files and generates an intermediate **vvp** assembly file, *testbench.vvp*. This is followed by the compilation of the *start.S* assembly file, the C codes and test assembly files into their binaries using the **riscv32-unknown-elf-gcc** command. This is succeeded by the generation of an executable file *firmware.elf*, which is translated into a *firmware.bin* file by **riscv32-unknown-elf-objcopy** utility. In order to run the machine code on the processor, a script *make-hex.py* in the firmware directory was used to convert the *firmware.elf* file into *firmware.hex* file. The *firmware.hex* file contains instructions as defined by the RISC-V specification in the 32-bit format (represented as 8 digit hexadecimal numbers), which are passed to PicoRV32. The **vvp** command of Icarus Verilog runs the previously generated *testbench.vvp* file and executes the machine code on PicoRV32.

Observing the testbench output provided deep insights into how the execution maps from the high-level language C code to the low-level machine code. To understand how the processor executes these instructions, the Verilog file *picorv32.v* was studied and understood. The instructions are loaded from the *firmware.hex* file in the testbench, which is stored in a 1024-bit wide register.

The above code snippet shows the loading of the instructions into a 32-bit wide, 16384 lines long memory in an **axi4_memory** instance. The instance is declared as follows:

This memory is the source of instructions for the processor, which lacks a default instruction memory. As mentioned in 5.3.1, PicoRV32 is available in two variants. In this case, the **axi4_memory** is interfaced to the **picorv32_axi** version of PicoRV32 core, using the **picorv32_axi_adapter**.

6.4 Implementation of RISC-V Processor

After understanding the working of PicoRV32 from a high-level perspective, the next step is to design and implement a small processor based on the RISC-V 32-bit base integer instruction set. PicoRV32 has several features, like custom instructions for IRQ handling, which make the processor complicated, heavier and relatively slower. On an average, PicoRV32 was observed to have a Cycles Per Instruction (CPI) value of 4.8. The execution of some specific instructions increased the CPI.

6.4.1 Objective

The objective is to develop a smaller and more efficient processor and analyze whether it could perform better than PicoRV32. The initial implementation intends to be capable of performing only a small set of instructions and compare its performance with PicoRV32. A fully functional processor implementing RISC-V ISA meeting the desired requirements (as mentioned in Section 6.4.2) must be developed before it could be further extended.

6.4.2 Design

6.4.2.1 Requirements

The design of any RISC based processor follows the load store architecture. It would be a 32-bit variant of the base integer instruction set, meaning that the registers width would be 32 bits. To start off with, the following instructions need to be present in our RISC-V implementation:

- Arithmetic: ADD, ADDI, MUL
- Load/Store: LW, SW
- Jump: JAL
- Branch: BNE

Among the above instructions, ADD, MUL, fall into the category of R-type instructions; ADDI and LW fall into the I-type instruction category, SW is in the S-type instruction category; JAL is in the UJ-type instruction category; BNE uses the SB-type instruction format.

Harvard architecture is selected over von Neumann architecture, as data and instruction memory separation is preferred. Two separate memory systems can perform better than having a single, unified memory for both data and instructions. The various components required by the processor implementation would be the control unit, registers, Arithmetic and Logic Unit (ALU) and instruction memory.

6.4.2.2 Components

The processor would be built using the required components as stated above in 6.4.2.1. This section discusses the functional requirements desired in the components.

1. Control Unit

The Control Unit determines the execution of operations on the processor. It instructs the processor's components like memory, ALU and input and output devices to respond to the program instructions. It can be perceived as the brain inside the CPU, making it the brain inside the brain of the computer.

It consists of two registers, namely the Program Counter (PC) register and the Instruction Register (IR). PC points to the address of the instruction to be executed, while IR contains the instruction being currently interpreted. Instructions are fetched from the instruction memory and decoded in the Control Unit. The data required for the instruction to execute is extracted from the registers or data memory, and passed on to the ALU to perform the operation specified by the instruction opcode.

The design of the control unit needs to be done according to RISC-V instruction specifications. As seen in Figure 6.5, the instruction contains additional information about the instruction in bits func7 and func3 which distinguish several instructions; opcodes can be the same for various instructions, as shown below.

2. Registers

Under the RISC-V specification, the base integer instruction set has a set of 32 registers, each 32-bit wide (x1-x31, x0 hardwired to constant

0000000	rs2	rs1	000	rd	0110011	ADD
imm[11:0]		rs1	000	rd	0010011	ADDI
0000001	rs2	rs1	000	rd	0110011	MUL
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[20:10:1 11:19:12]				rd	1101111	JAL
imm[12:10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE

Figure 6.5: Formats for RISC-V instructions to be implemented

0). The Control Unit decodes instructions and extracts the register address on which the ALU needs to perform specified instructions. Data can be transferred between memory and registers using the load and store instructions. Thus, in a load store architecture, register memory is significant and should be wisely used in order to reduce overheads of memory accesses. It can be declared as shown in the following code snippet:

3. Memory

Memory stores instructions and data. Being a Harvard Architecture processor, there would be a clear separation of instructions and data. A little endian memory system would be followed. The instruction memory stores the instructions generated by the makehex.py script, converting the firmware.bin file to firmware.hex format. The width and length of the instruction memory are 32 bits and 16384 lines, respectively.

4. Arithmetic and Logic Unit

The Arithmetic and Logic Unit of a CPU performs arithmetic and logical operations on operands specified by the instructions. It is the fundamental unit of the CPU of any computing device. After performing an operation, the result is stored back into the register file at the address specified by the instruction.

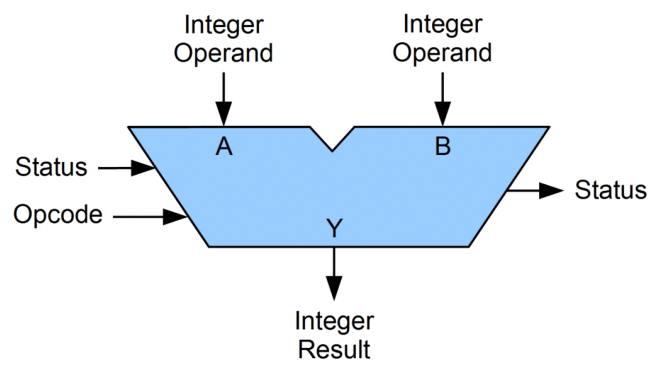


Figure 6.6: Symbolic representation of an ALU

Chapter 7

Conclusions and Future Work

This chapter concludes and summarizes this report. Furthermore, in this chapter we discuss future research directions in detail.

7.1 Conclusions

7.2 Future work

Appendix A

OpenCL

Appendix B

RISC-V

Bibliography

- [1] Wikipedia. OpenCL.
- [2] Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, Jeongdo Son, and Satoshi Miki. The OpenCL Programming Book. <https://www.fixstars.com/en/openc1/book/OpenCLProgrammingBook/what-is-openc1/>, 2012.
- [3] AJ Guillon. Openc1 1.2: High level overview, 2013.
- [4] Khronos OpenCL Working Group. *The OpenCL Specification*. 2012.
- [5] Yann LeCun. LeNet-5, Convolutional Neural Networks.
- [6] Stanford. Convolutional Neural Networks.
- [7] Ujjwal Karn. An intuitive explanation of Convolutional Neural Networks. <https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets>, 2016.
- [8] Wikipedia. MNIST Database.
- [9] Ernst Kussui and Tatiana Baidyk. Improved method of handwritten digit recognition tested on mnist database. 22, 2004.
- [10] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST Database.
- [11] Gopalkrishna Hegde. Lenet5 openc1 application. <https://github.com/gplhegde/papaa-openc1/tree/master/openc1-src/mnist>, 2012.
- [12] XDA Developers. Open source risc-v core designs, why google cares and why they matter. <https://www.xda-developers.com/risc-v-cores-and-why-they-matter>, January 2016.

- [13] Krste Asanović and David A. Patterson. Instruction sets should be free: The case for risc-v. <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html>, August 2014.
- [14] RISC-V Foundation. RISC-V: The Free and Open RISC Instruction Set Architecture.
- [15] ANSI/IEEE Std 754-2008. IEEE Standard for Floating-Point Arithmetic. <http://ieeexplore.ieee.org/document/4610935/>, August 2008.
- [16] Andrew Waterman, Yunsup Lee, and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1. <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.pdf>, May 2016.
- [17] Albert Ou. RISC-V Software Tools Bootcamp. , 2015.
- [18] RISC-V Foundation. Software Tools.
- [19] RISC-V Foundation. RISC-V Proxy Kernel and Boot Loader.
- [20] Andrew Waterman and Yunsup Lee. RISC-V ISA Simulator. <https://riscv.org/software-tools/risc-v-isa-simulator/>, June 2011.
- [21] RISC-V Foundation. RISC-V Opcodes.
- [22] Clifford Wolf. PicoRV32 - A Size-Optimized RISC-V CPU .
- [23] Stephen Williams. Icarus Verilog. <http://iverilog.icarus.com/>.