ARMv7 vs x86 Computer Architecture

Prepared by

Tyson Fairhurst
Allen Chan

Oregon State University
Winter 2020
CS 271 Computer Architecture and Assembly Language

Oregon State University

Table of Contents

History of the ARM Architecture

The ARM architecture began in 1978 at Acorn Computers based in the UK. Looking to create a low cost and simple to program processor that is easy to manufacture, Acorn Computers designed and built the ARM1, originally called the Acorn RISC Machine. This processor was based on 32-bit architecture and due to its simplistic design focus, the ARM1 favored efficiency over more complex machine code. This enabled low power consumption for executing machine code, creating an efficient CPU that anyone could implement in their systems [1]. Eventually, Acorn Computers became Advanced RISC Machines or ARM.

Today ARM continues to develop high-end CPUs implementing the ARM architecture and is used in billions of devices as low power, efficient CPUs that are perfect for battery-powered devices. One of the largest and most well known modern companies that utilize the ARM architecture is Apple, in which most of their current devices are based on [2]. The ARM architecture is perfect for many types of devices and applications, from laptops and smartphones to cybersecurity and machine learning. ARM is used in all sorts of devices. Small devices such as smart home enabled speakers do not need full-sized processors in order to function, but instead can use small ARM-based microcontrollers to save on energy consumption and cost of manufacturing [3].

Two of the most widely used computer architectures today are x86 and ARM. While both are very capable, there are a few key differences that separate the use cases for each. X86 is better for desktop-grade CPUs built by AMD and Intel, that are not limited by space and energy constraints. X86 based chips are also expensive and complex to manufacture, compared to ARM CPUs [4-6]. Between the two architectures, their many differences and similarities will be compared and discussed throughout this report.

RISC vs CISC

The fundamental difference between the x86 and ARM computer architectures is the instruction set that each is based on. X86 is based on Complex Instruction Set Computing or CISC. CISC is an instruction set founded on the idea of using complex instructions or code to complete multiple actions at once. This means a single line of machine code could contain multiple instructions and take multiple clock cycles to be fully executed [7]. This, in turn, increases the time and complexity to execute a single line of machine code in CISC based x86 CPUs.

For example, in order to multiply two numbers together, you first need to fetch both numbers from memory and store them in registers. This takes a single line of code for each fetch. Then you need to multiply the two registers together and store the result, which requires a single line of machine code. In total it takes three lines of code, the fetch for both numbers and multiplication, but each of the three lines of code contains multiple instructions within them in order for the execution to be complete. For instance, the fetch retrieves the variable from memory and stores the value in the specified register, all in a single line of machine code. This is what makes up CISC, the ability to execute multiple instructions from a single line of machine code.

ARM CPUs are based on RISC or Reduced Instruction Set Computing. RISC was created as a way to have the software and hardware match in complexity, meaning that each instruction would do one task and take a single clock cycle to execute. Unlike x86, this meant each instruction needed to have a fixed size in memory to maintain single clock cycle execution. Since each instruction is fixed in size, every instruction can be fetched from memory and executed much faster than x86, but each line of machine code can only complete one task.

For example, if you were to add one to a number on a CPU using CISC architecture, it would take a single line of code: inc EAX, which will add 1 to the data stored in the EAX register. In RISC you would have to write three lines of code in order for the same action to be completed. First, the fetch where the CPU will find and return the address of the variable you want to add one to. Then, you need to store the number 1 temporarily in another register. Finally, for the final line of machine code, you need to add the two values and specify where that result will be stored. [8]

The instruction set is one of the main differences between x86 and ARM. The RISC instruction set allows an ARM CPU to perform fast execution of fixed size instructions, where an x86 CPU has to find the size of the next instruction before it fetches it from memory. This helps ARM CPUs achieve a low cost of production and energy efficiency that an x86 CPU could not achieve. RISC helps simplify ARM CPUs in both size and complexity, making ARM CPUs a perfect solution for small embedded applications.

Coprocessors and Pipeline

To make up for additional instructions during programming, ARM CPUs employ a key feature called pipelining. Each ARM CPU has a different style of pipeline with different types of stages depending on the application. A single ARM CPU could have anywhere from 3 stages to 14 stages along the pipeline. The pipeline is essential in giving ARM CPUs an advantage in efficiency over a CISC x86 based CPU. Each stage along the pipeline is responsible for a single

task for each instruction to be executed. In a common three-stage pipeline, the first stage fetches the instruction from memory, the second stage decodes the instruction and the third stage executes the instruction and writes back to memory. As each stage completes a task, the stage will move on to the next instruction. So once the fetch has been completed for a specific instruction, the fetch stage will perform the fetch for the next instruction. For example, in three clock cycles, the pipeline will have fully completed one instruction, fetched and decoded another, and fetched the third instruction from memory (Figure 1). This method of executing instructions is very fast and efficient, again helping ARM CPUs achieve low power consumption [9].

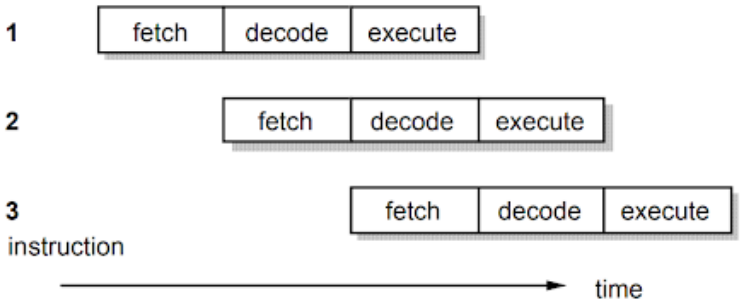

Figure 1. Example of a three stage pipeline.[9]

In addition to the pipeline, the ARM architecture also uses coprocessors to help speed up complex instructions. Coprocessors are attached to the main CPU and can be accessed using machine code similar to how a register would be accessed. Coprocessors have their own pipeline but without the fetch step. This is because when an instruction is fetched and decoded in the main core of the CPU and the CPU determines that it is an instruction for a coprocessor, it will send the fetched instruction directly to the necessary coprocessor to be decoded and executed [10].

One example of a coprocessor is the Vector Floating Point Coprocessor (VFP). The VFP is used to handle floating-point arithmetic. When wanting to perform arithmetic on floating-point values, instead of using the floating-point registers in the main CPU, you can send the values to the VFP to speed up the calculations and efficiency of executing the instructions, since the VFP is optimized to perform these instructions faster [11].

Modes and Execution Levels

The ARM architecture supports seven processor modes. Modes can be changed through software, external interruptions, or exception processing.

Arm Processor Modes

- User (usr)
- Fast interrupt (fiq)
- Interrupt (irq)
- Supervisor (svc)
- Abort (abt)
- System (sys)
- Undefined (und)

Most applications programs execute in user mode. While executing in user mode, the program being executed is unable to access protected system resources or change mode without causing an exception to occur [12]. This allows the operating system to control and use system resources. This mode is used to protect the user from changing the system that can possibly break the system.

Another type of mode is known as privileged modes. Privilege modes have full access to the system and can alter settings and files freely. They can change to any mode without restrictions such as **FIQ, IRQ, Supervisor, Abort, and Undefined**. These modes are entered when a specific exception occurs. Each of the exception modes have registers to avoid corrupting the system.

Figure 2 shows how an ARM communicates with other parts of the architecture. It is crucial to understand how this every part communicates with each other for the ARM architecture to work properly.

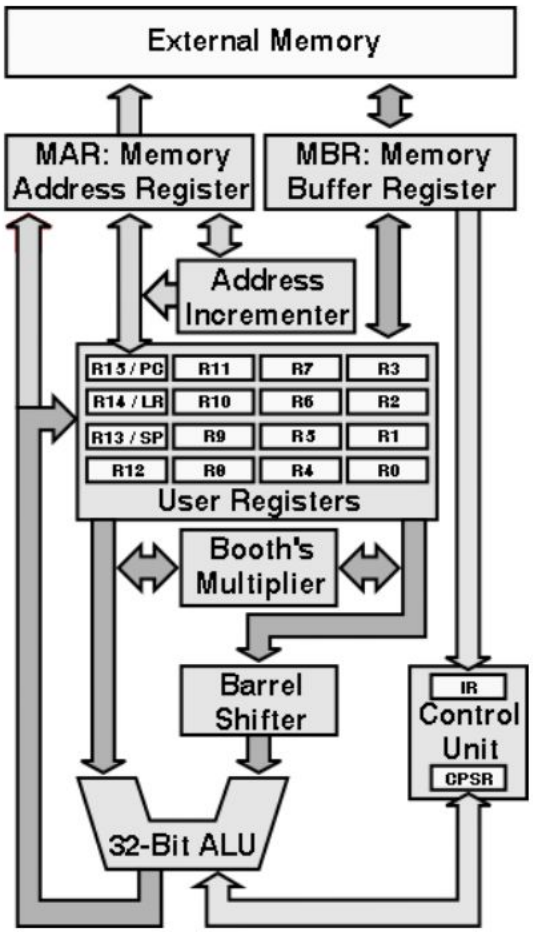

Figure 2. ARM Block Diagram[12]

The x86 architecture has four main modes that it can run on. It can run on 16-bit real mode, 16-bit protected mode, 32-bit protected mode, and 64-bit long mode. The main difference between a real and protected mode is in how they handle segments. In real mode, the segments directly address memory as a 16-byte page whereas protected mode segments are indexed into a table that contains the physical base and size of segment [13].

Both 16-bit and 32-bit operating modes allow the use of 16-bit and 32-bit registers through instruction prefixes that set the address size to 16-bit or 32-bit. When the operating mode is set to default, the operation size is flagged with a prefix. This operation mode and the address sizes affect the size of operands. For example, a 32-bit operation size instruction with an immediate operand will have a 32-bit value instruction.

In 64-bit long mode, it is obsolete about instructions. It is the only mode where 64-bit registers are available to use. 64-bit registers are not accessible from 16-bit or 32-bit. Due to

wide use of relocatable libraries, long mode is able to add the addressing mode: RIP-relative [13].

Registers

ARMv7 supports 13 general-purpose registers, and three special type registers the stack pointer (SP), link register (LR), and the program counter (PC). The general-purpose registers are defined as r0 to r13. All of which are 32-bits in size. These general-purpose registers are used to store values, addresses, and any other sort of data that needs to be stored throughout a program. In addition to holding values, the general-purpose registers can also be used to perform arithmetic between variables and other registers.

In ARMv7, the stack pointer is a pointer that contains the address of the current stack being referenced. It is similar to the ESP general-purpose register in x86 architecture and is useful in performing operations that involve the system stack. The link register holds the return link, which holds the return address of a subroutine. The program counter is a register that holds the address of the next instruction to be executed. This is similar to the EIP pointer in x86 architecture [14-15].

X86 is similar in its use of registers. X86 has 8 general-purpose registers, six-segment registers, a processor status flags register (EFLAGS), and an instruction pointer. The 8 general-purpose registers are 32-bits in size and include EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP. Similar to the ARM architecture these registers are used for arithmetic and data storage throughout a program. The six-segment registers are 16-bits in size and are used for storing program instructions, pointers to segment descriptor tables, variables, as well as, local procedure variables and procedure parameters. The EFLAGS register contains control and status flags to help control the operations the CPU needs to take on a specific instruction. Lastly, the instruction pointer is the EIP general-purpose register, and it contains the address in memory of the next instruction to be executed by the CPU [16].

Between the ARMv7 and x86 architectures, they both employ a similar use of general-purpose, and special registers to run machine code. ARMv7 has more general-purpose registers, but they are limited in functionality due to running RISC. X86 may have less general-purpose registers, but each register can help supplement more complex instructions than those on an ARM CPU. The way each architecture implements these registers allows them to take full advantage of the instruction sets that run each of them, enabling some of the key features and differences between ARMv7 and x86.

Memory Model

A memory model is a way of organizing and defining how memory processes. It provides a structure and rules for following when changing how addresses are accessed and used on a system. The memory model provides attributes that can be applied to an address that defines rules with the memory access ordering. The ARM architecture has four main models.

Memory Models:

1. Application Level Memory Model
2. Common Memory System Architecture Features
3. Virtual Memory System Architecture (VMSA)
4. Protected Memory System Architecture (PMSA)

Application Level Memory Model: In ARM architecture, it uses a single, flat address space of 2^{32} 8-bit bytes[19 now 18]. In the address space, byte addresses are treated as unsigned numbers from the range of 0 to 2^{32} - 1. Address space is divided into 2^{30} 32-bit words and 2^{31} 16-bit halfwords. In 2^{30} 32-bit words, each address is divisible by 4 and the last two bits of the address are 0b00. In 2^{31} 16-bit words, the address is divisible by two and the last bit of the address is 0 [17].

Common Memory System Architecture Features: The ARM architecture supports different implementation choices for the memory system microarchitecture and memory hierarchy based on the requirements of the system being implemented. In this context, memory system architecture describes a design space in which an implementation is made.

Virtual Memory System Architecture (VMSA): In VMSA, a memory Management Unit (MMU) controls the address translation, access permissions, and memory attribute determination and checks for memory accesses made by the processor. The MMU is controlled by system control registers that can disable the MMU if it is asked to.

Protected Memory System Architecture (PMSA): PMSA is a Memory Protection Unit (MPU) that provides simple memory protection compared to MMU based on VMSA. The simplification applies to both hardware and the software. PMSA processor is identified by the presence of MPU type registers.

Addressing Modes

The ARM architecture uses data values that must be loaded into CPU registers before arithmetic or logic operations can be performed. After instructions are loaded from memory or store values in memory, they can't be modified. Addressing mode is the way a CPU combines memory address for a load by adding a positive or negative offset to a value in base register [18]. The ARM architecture has three addressing modes:

Immediate - The offset is an unsigned integer that is stored as part of the instruction. The integer can be added or subtracted from the value currently stored in the base register.

Register - The offset is an unsigned integer stored in a register instead of the pc. It can also be added or subtracted from the value stored in the base register.

Scaled Register - The offset is an unsigned integer that is in a register. It is shifted by an immediate amount before it can be added or subtracted from the value in the base register.

These addressing modes can affect value in the base register in three ways:

Offset - Value in base register is not changed

Pre-indexed - The offset is combined with the value currently stored in the base register. The base register is now updated with a new address before it is used to access memory.

Post-indexed - The value stored in the base register alone is used to access memory. Then the offset is combined with the base register. Then the base register is updated with the new address after accessing new memory.

In the x86 architecture, assembly language instructions require operands to be processed. An operand address provides the location of where the data must be processed and stored. However, some instructions don't require an operand whereas others can require one, two, or three. The x86 architecture has three basic modes of addressing:

1. Register
2. Immediate
3. Memory

In **register addressing**, a register contains an operand. The register may be the first, second, or both depending on the given instruction. An example in figure 3, shows when the register is in the first, second, or both registers.

```
MOV DX, TAX_RATE   ; Register in first operand
MOV COUNT, CX      ; Register in second operand
MOV EAX, EBX       ; Both the operands are in registers
```

Figure 3. Register addressing [19]

In **immediate addressing**, the operand has a constant value. When an instruction with two operands using the immediate addressing, the first operand is a register or a memory location. The second operand is an immediate constant while the first operand defines the length of the data. Figure 4 shows an example of immediate addressing.

```
BYTE_VALUE  DB  150    ; A byte value is defined
WORD_VALUE  DW  300    ; A word value is defined
ADD  BYTE_VALUE, 65    ; An immediate operand 65 is added
MOV  AX, 45H           ; Immediate constant 45H is transferred to AX
```

Figure 4. Immediate addressing [19]

In **memory addressing**, there are three different types. There are direct, direct-offset, and indirect memory addressing. These three types of memory addressing are important when accessing registers and memory.

Direct memory addressing is when the offset value is specified directly as part of the instruction. The assembler will offset the value and store the offset value of all variables in the program. There are two operands in memory addressing. One operand refers to a memory location and the other operand references a register. Adding, subtracting, division, and multiplying are all examples of direct memory addressing as in figure 5.

```
ADD     BYTE_VALUE, DL  ; Adds the register in the memory location
MOV     BX, WORD_VALUE  ; Operand from the memory is added to register
```

Figure 5. Direct Memory Addressing [19]

In **direct-offset addressing,** it uses arithmetic operators to modify an address. For example, you can use *MOV* to get an element in an array at a specific index. As in figure 6, it gives an example of accessing data from tables in the memory into registers.

```
BYTE_TABLE DB  14, 15, 22, 45      ; Tables of bytes
WORD_TABLE DW  134, 345, 564, 123  ; Tables of words

MOV CL, BYTE_TABLE[2]   ; Gets the 3rd element of the BYTE_TABLE
MOV CL, BYTE_TABLE + 2  ; Gets the 3rd element of the BYTE_TABLE
MOV CX, WORD_TABLE[3]   ; Gets the 4th element of the WORD_TABLE
MOV CX, WORD_TABLE + 3  ; Gets the 4th element of the WORD_TABLE
```

Figure 6. Direct-Offset Addressing [19]

Indirect memory addressing is when it utilizes the computer's ability of segment offset addressing [18]. Base registers such as EBX, EBP, and index registers are coded inside square brackets to reference memory. For example, figure 7 shows how to access different elements of a variable.

```
MY_TABLE TIMES 10 DW 0  ; Allocates 10 words (2 bytes) each initialized to 0
MOV EBX, [MY_TABLE]     ; Effective Address of MY_TABLE in EBX
MOV [EBX], 110          ; MY_TABLE[0] = 110
ADD EBX, 2              ; EBX = EBX +2
MOV [EBX], 123          ; MY_TABLE[1] = 123
```

Figure 7. Indirect Memory Addressing [19]

Code Example One

This code example is a simple loop in ARM assembly language. This program starts by storing the number 5 into a variable called num. Next, the data stored in num is moved to the register r0. Then, the number 1, a constant is stored in the r4 register. These steps together set up the variable num and registers for the loop structure to iterate over.

The first line of code in the loop compares the data stored in r0, which in this case is currently 5 to the number 1. Since it is greater than or equal to 0, the next line will take effect and multiply the data stored in r0 and r4 and store the result in r4. For the first iteration, this is 5*1 and now r4 becomes 5. Next, 1 is subtracted from the data stored in r0, and since 5 is still greater than or equal to 0, the loop will again iterate. This loop will run until r0 is less than 1. The following table shows the changes for every iteration.

Results

Num	Register r0	Register r4	Loop iteration
5	5	1	1
5	4	5	2
5	3	20	3
5	2	60	4
5	1	120	5
5	0	120	6

As shown in the table above, the loop will iterate 6 times, with each iteration multiplying the data stored in r0 by the data stored in r4 and replacing the data stored in r4 with the result. The end result for register r4 is 120, and register r0 is 0.

Code Example Two

In code example 2, it is a loop that stores numbers to an array and adds the previous two numbers. This code prints fibonacci numbers in an array by loading addresses to register. The number “1” gets stored in memory to “termOne” and “termTwo”, number “10” to “nmofTrm”. The number “40” reserves a zeroed block of memory to fill an array called “fibArr”.

This program starts by allocating “termOne” to r0, “termTwo” to r1, and “=fibArr” to r2. Then the address of r2 gets stored to the task register of r0, and r1. After that “=fibArr” is loaded into r3 which makes it possible to now add “#8” to r3. Now the loop starts to add the previous 2 numbers and store it to the register. Then #2 gets compared to r0 and if the branch is not equal to the value in r0. This loop will continue to run and fill the array until #2 is equal to r0.

Conclusion

Both ARM and x86 computer architectures are essential in driving the CPUs that empower everything that we do. Without these computer architectures, computing power would not be at the level it is today. The ARM computer architecture is excellent at fast, low power consumption computing that makes it perfect for small devices, and electronics that don't need the complex computing power of an x86 CPU. Whereas x86, on the other hand, is widely used in desktop, and laptop CPUs to deliver the highest performance possible.

While the use cases for each architecture are different, improvements in ARM development is steadily improving the types of devices that ARM is capable of powering. This year ARM-based CPUs are returning to computers. Manufacturers such as Lenovo, Microsoft, and Samsung are adopting ARM-based CPUs for sleek, portable 2-in-1 laptops and notebooks that deliver fast performance, and all-day battery life, while still being able to meet the performance demands of a similar x86 CPU powered device [20]. The advantages that ARM enables in portable devices are plentiful and will only continue to improve and empower billions of consumer electronics each year.

References

[1] “History of the Development of the ARM Chip at Acorn.” [Online]. Available: https://www.cs.umd.edu/~meesh/cmsc411/website/proj01/arm/history.html. [Accessed: 01-Mar-2020].

[2] D. E. Dilger, “ARM to A4: How Apple changed the climate in mobile silicon,” *AppleInsider*. [Online]. Available: https://appleinsider.com/articles/19/11/06/arm-to-a4-how-apple-changed-the-climate-in-mobile-silicon. [Accessed: 01-Mar-2020].

[3] J. V. Vijay and B. Bansode, “ARM Processor Architecture Evolution and Applications,” *International Journal of Science, Engineering and Technology Research (IJSETR)*, vol. 4, no. 10, pp. 3385–3387, Oct. 2015.

[4] *AMD, “AMD64 Architecture Programmer’s Manual Volume 1: Application Programming.” Advanced Micro Devices, Dec-2017.* Available: https://www.amd.com/system/files/TechDocs/24592.pdf [Accessed: 01-Mar-2020].

[5] S. Rodgers and R. A. Uhlig, “Intel's X86: Approaching 40 and Still Going Strong,” *Intel Newsroom*, 08-Jun-2017. [Online]. Available: https://newsroom.intel.com/editorials/x86-approaching-40-still-going-strong/#gs.y1gm6r. [Accessed: 01-Mar-2020].

[6] R. Mitchell, “Understanding the Differences Between ARM and x86 Processing Cores - News,” *All About Circuits*, 05-May-2017. [Online]. Available: https://www.allaboutcircuits.com/news/understanding-the-differences-between-arm-and-x86-cores/. [Accessed: 01-Mar-2020].

[7] C. Chen, G. Novick, and K. Shimano, *RISC vs. CISC*. [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/index.html. [Accessed: 01-Mar-2020].

[8] ARM, *ARM® Developer Suite Assembler Guide*, 1.2 ed. ARM, 2001. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dui0068b/DUI0068.pdf. [Accessed: 01-Mar-2020].

[9] A. Milenković, “ARM Organization and Implementation.”[Online]. Available:https://myweb.ntut.edu.tw/~tylee/Courses/91_2/SOC_Archtecture_Course/SOCA_5_ARM_Organization%20and%20Implementation.pdf [Accessed 01-Mar-2020].

[10] ARM, “Chapter 11 Coprocessor Interface,” in *ARM1176JZ-S™ Technical Reference Manual*, 2009, pp. 11–1-11–4.[Accessed 01-Mar-2020].

[11] ARM, “Chapter 6 Vector Floating-point Programming,” in *ARM® Developer Suite Assembler Guide*, 1.2 ed., ARM, 2001, pp. 6–1-6–5.[Accessed 01-Mar-2020].

[12] P. Knaggs and S. Welsh, “RM Assembly Language Programming,” 22-Dec-2003. [Online]. Available: http://arantxa.ii.uam.es/~gdrivera/sed/docs/ARMBook.pdf. [Accessed: 02-Mar-2020].

[13] 21.2. Execution Modes and Extensions. [Online]. Available: https://www.tortall.net/projects/yasm/manual/html/arch-x86-modes.html. [Accessed: 02-Mar-2020].

[14] ARM, “Chapter A2 Application Level Programmers’ Model," in *ARM® v7-M Architecture Reference Manual*, 2010, pp. A2–37-A2–46.

[15] ARM, “Chapter 3 Overview of the ARM Architecture,” in *ARM® Compiler Version 4.1 Using the Assembler*, 2011, pp. 3–17.[Accessed 01-Mar-2020].

[16] K. R. Irvine, “Chapter 2 x86 Processor Architecture,” in *Assembly language for x86 processors*, Upper Saddle River, NJ: Pearson, 2011, pp. 37–39.[Accessed 01-Mar-2020].

[17] “ARM® Architecture Reference Manual.” [Online]. Available: https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf. [Accessed: 05-Mar-2020].

[18] R. G. Plantz, “Introduction to Computer Organization: ARM Assembly Language Using the Raspberry Pi,” Addressing Modes. [Online]. Available: https://bob.cs.sonoma.edu/IntroCompOrg-RPi/sec-addr-mode.html. [Accessed: 02-Mar-2020].

[19] “Assembly - Addressing Modes,” Tutorialspoint. [Online]. Available: https://www.tutorialspoint.com/assembly_programming/assembly_addressing_modes.htm. [Accessed: 02-Mar-2020].

[20] R. Varma, “Third Generation of Windows on Arm Laptops,” *Third Generation of Windows on Arm Laptops - Processors blog - Processors - Arm Community*, 04-Feb-2020. [Online]. Available: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/third-generation-of-windows-on-arm. [Accessed: 07-Mar-2020].

Oregon State
University