The ARM Instruction Set Architecture

Dean Vidafar (Z5205847)

UNSW SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Author Note

For Any Enquiries Contact

<Z5205847@student.unsw.edu.au>

Table of Contents

[Table of Figures 3](#_Toc523828649)

[Abstract 4](#_Toc523828650)

[ARM7TDMI Core Diagram 5](#_Toc523828651)

[The ARM Instruction Set Architecture 6](#_Toc523828652)

[Conditional Execution 7](#_Toc523828653)

[Status Flags and Condition Codes 8](#_Toc523828654)

[The Thumb Instruction Set 9](#_Toc523828655)

[Design of Thumb 10](#_Toc523828656)

[Why do we use thumb? 10](#_Toc523828657)

[Compromises Made with Thumb Code 11](#_Toc523828658)

[Arithmetic and the ARM7TDMI 12](#_Toc523828659)

[Registers in the ARM7TDMI 13](#_Toc523828660)

[Cache and the ARM7TDMI 14](#_Toc523828661)

[Memory in the ARM7TDMI 14](#_Toc523828662)

[The ARM7TDMI Instruction Set 16](#_Toc523828663)

[Multi-byte Shifting on the ARM7TDMI 16](#_Toc523828664)

# Table of Figures

[Figure 1: The ARM7DMI Core Diagram (ATMEL, 2004) 5](file:///C:\Users\dvida\Documents\GitHub\2121-Assignment---ARM7TDMI\The%20ARM%20Instruction%20Set%20Architecture.docx#_Toc523828641)

[Figure 2: ARM7TDMI Architecture (Techcon, 2012) 6](file:///C:\Users\dvida\Documents\GitHub\2121-Assignment---ARM7TDMI\The%20ARM%20Instruction%20Set%20Architecture.docx#_Toc523828642)

[Figure 3: Conditional Execution in the ARM7TDMI (Varma, 2015) 7](file:///C:\Users\dvida\Documents\GitHub\2121-Assignment---ARM7TDMI\The%20ARM%20Instruction%20Set%20Architecture.docx#_Toc523828643)

[Figure 4: Flag Bits of the ARM7TDMI (Thomas, 2012) 8](file:///C:\Users\dvida\Documents\GitHub\2121-Assignment---ARM7TDMI\The%20ARM%20Instruction%20Set%20Architecture.docx#_Toc523828644)

[Figure 5: Example of how the ADD instruction is converted 9](file:///C:\Users\dvida\Documents\GitHub\2121-Assignment---ARM7TDMI\The%20ARM%20Instruction%20Set%20Architecture.docx#_Toc523828645)

[Figure 6: ARM to Thumb instruction set mapping. (Goudge and Segars, 1996) 11](file:///C:\Users\dvida\Documents\GitHub\2121-Assignment---ARM7TDMI\The%20ARM%20Instruction%20Set%20Architecture.docx#_Toc523828646)

[Figure 7: The conversion of High-Level C, to Low Level ARM. (Suh, 2015) 12](file:///C:\Users\dvida\Documents\GitHub\2121-Assignment---ARM7TDMI\The%20ARM%20Instruction%20Set%20Architecture.docx#_Toc523828647)

[Figure 8: Program status register format ATMEL (2004) 13](file:///C:\Users\dvida\Documents\GitHub\2121-Assignment---ARM7TDMI\The%20ARM%20Instruction%20Set%20Architecture.docx#_Toc523828648)

Abstract

The ARM processor is a 32-bit Reduced Instruction Set Computer (RISC), a microprocessor that recognises a limited number of instructions. An advantage of RISC microprocessors is that they can execute instructions relatively quick since the instructions are elementary. RISC chips also require fewer transistors and other components, making both RND and manufacturing more cost-efficient. This architecture is significantly different from other CISC (Complex instruction set computer) microprocessors and is primarily designed for embedded systems. In this report, the ARM Instruction Set Architecture, Memory Models, Registers, Instruction Set, and Data Types will be discussed.(Shee, 2004)

# ARM7TDMI Core Diagram

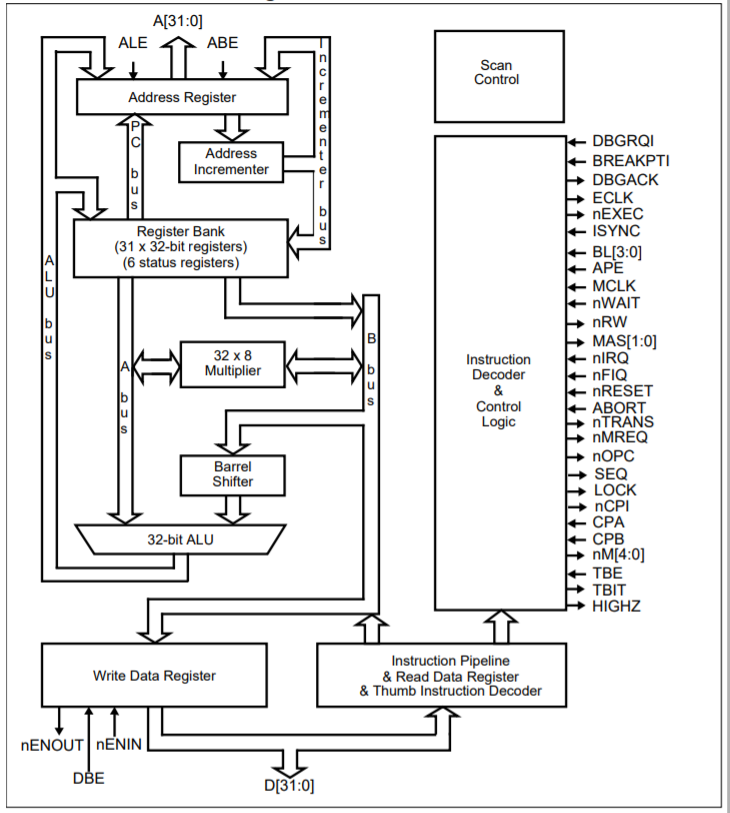


Figure 1: The ARM7DMI Core Diagram (ATMEL, 2004)

The ARM Instruction Set Architecture

The drawback of most RISC processors, including the ARM7TDMI however, is code density. In many cost-critical applications, one of the most expensive system components has always been memory. Therefore, the less memory a program occupies, the more cost effective. Since RISC processors have simple instructions, multiple instructions are needed to perform what may be done in a single CISC instruction. (Segars et al., 1995) The ARM microprocessor is based on the load/store architecture. ARM7TDMI does not offer any support for unaligned memory accesses. This microprocessor features two instruction sets, with both having mostly single clock-cycle execution.(McDermott, 2008). These instruction sets are a 16-bit Thumb instruction set and a 32-bit ARM instruction set

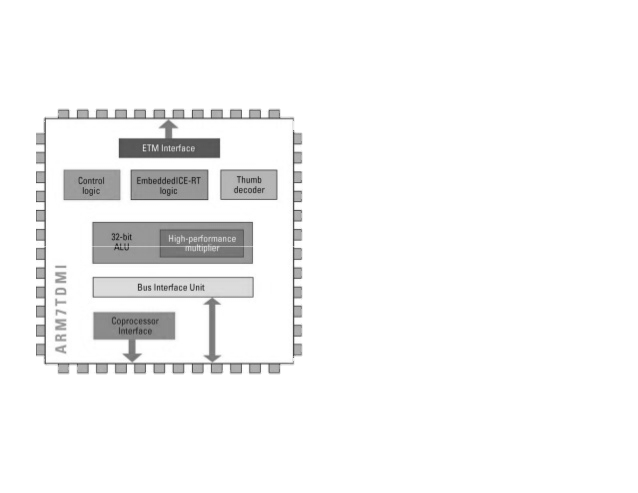


Figure 2: ARM7TDMI Architecture (Techcon, 2012)

# Conditional Execution

A beneficial and major feature of the ARM7TDMI's architecture is the ability for instructions to be executed conditionally. The condition is specified with a two-letter suffix (e.g. EQ, CC) appended to the mnemonic. This condition is then tested against the current processor flags and if it is not met, the instruction is then treated as a no-op. This feature often makes branching unnecessary, increasing code density and avoiding pipeline stalls which in turn increases execution speed.

By design the data processing instructions do not affect the condition code flags but can be made to by suffixing S. The comparison instructions CMP and TST do this implicitly. (Thomas, 2012)

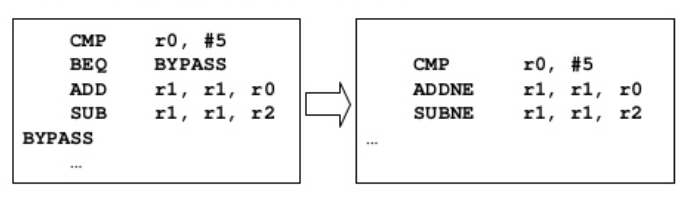


Figure 3: Conditional Execution in the ARM7TDMI (Varma, 2015)

## Status Flags and Condition Codes

When an ALU operation changes the flags:

* N — Negative
  + Set if the result of a data processing instruction was negative
* Z — Zero
  + Set if the result was zero
* C — Carry
  + Set if an addition, subtraction or compare causes a result bigger than 32bit, or is set from the output of the shifter for move and logical instructions.
* V — Overflow
  + Set if an addition, subtraction or compare produces a signed result bigger than 31 bits.

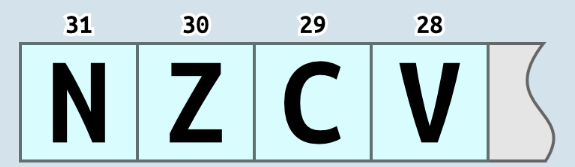


Figure 4: Flag Bits of the ARM7TDMI (Thomas, 2012)

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Suffix | Description | Flags |
| 0000 | EQ | Equal / equals zero | Z |
| 0001 | NE | Not equal | !Z |
| 0010 | CS / HS | Carry set / unsigned higher or same | C |
| 0011 | CC / LO | Carry clear / unsigned lower | !C |
| 0100 | MI | Minus / negative | N |
| 0101 | PL | Plus / positive or zero | !N |
| 0110 | VS | Overflow | V |
| 0111 | VC | No overflow | !V |
| 1000 | HI | Unsigned higher | C and !Z |
| 1001 | LS | Unsigned lower or same | !C or Z |
| 1010 | GE | Signed greater than or equal | N == V |
| 1011 | LT | Signed less than | N != V |
| 1100 | GT | Signed greater than | !Z and (N == V) |
| 1101 | LE | Signed less than or equal | Z or (N != V) |
| 1110 | AL | Always (default) | any |

Table 1: Condition Codes of the ARM7TDMI (Thomas, 2012)

# The Thumb Instruction Set

The thumb instruction set is a subset of the most commonly used 32-bit ARM instructions. Each thumb instruction is 16 bits long, with a corresponding 32-bit ARM instruction that preforms the same operation. There is excellent interoperability between ARM and Thumb states due to the thumb instructions operating with the standard ARM register configuration. 16-bit thumb instructions are transparently decompressed to full 32-bit instructions on execution without performance loss. (ATMEL, 2004) The Thumb instruction set contains a subset of 36 instruction formats taken from the standard 32-bit ARM instruction set which has then been recoded into 16-bit op-codes. The first implementation of Thumb in an ARM microprocessor is on the ARM7TDMI, where the T indicates Thumb-awareness. (Goudge and Segars, 1996)

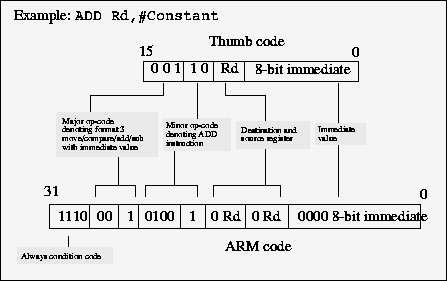


Figure 5: Example of how the ADD instruction is converted

from Thumb to ARM. (Bong-Ho, 1998)

## Design of Thumb

There were three different types of ARM instruction that were considered in creating a compressed instruction set.

1. Instructions which were shown to be the most frequently used and hence the most important
2. Instructions needed by the compiler to produce compact code
3. Instructions that had some redundancy in the fixed length op-code.

Trade-offs were made between code size, execution speed, ease of implementation and architectural elegance. Minimalistic code was the primary goal, however not at the expense of performance. (Goudge and Segars, 1996)

## Why do we use thumb?

Thumb has all the advantages of using a 32-bit core such as:

* 32-bit registers
* 32-bit shifter
* 32-bit Arithmetic Logic Unit (ALU)
* 32-bit address space
* 32-bit memory transfer

Thumb offers a long branch range, powerful arithmetic operations and a large address space. Thumb Code is typically 65% of the size of ARM code and provides 160% of the performance of ARM code when running from a 16-bit memory system. Therefore, the ARM7TDMI core is ideally suited to embedded applications where there is restricted memory bandwidth and code density as well as footprint is integral. (ATMEL, 2004) By making the ARM7TDMI core Thumb aware, the silicon area is able to be kept small and hence maintain low-power and high-MIPS/W performance. (Segars et al., 1995) The Thumb instruction set is not a replacement for the ARM instruction set. A Thumb-aware processor is able to execute both instructions sets, allowing for the designer to optimise their code at a subroutine level for either an extra fifteen percent performance with ARM or thirty percent better code size with Thumb. Code which requires the absolute maximum performance of ARM can execute in the ARM state from 32-bit wide memory and code which requires the absolute maximum code density can be run in the Thumb state from sixteen-bit or eight-bit wide memory. (Goudge and Segars, 1996)

## Compromises Made with Thumb Code

With thumb code, conditional execution is not supported since the condition code would not have left enough space for a useful set of instructions. Another compromise made was how the register bank is accessed. ARM code has free access to 16 registers at once and uses three or four operand instructions with 4-bit register specifiers. For thumb code however, in order to allow enough opcode space, there is free access to only eight registers. Instructions consist of the more conventional two or three operands and use 3- bit register specifiers. (Segars et al., 1995)

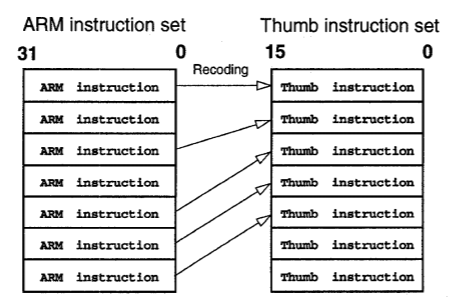


Figure 6: ARM to Thumb instruction set mapping. (Goudge and Segars, 1996)

# Arithmetic and the ARM7TDMI

The ARM7TDMI Microprocessor includes integer arithmetic operations for add, subtract and multiply, however, no support is provided for division. The microprocessor supports 32-bit by 32-bit multiplication with either a 32-bit result or 64-bit result. Arithmetic instructions can either set or preserve the conditions bits at the will of the programmer. (Diba, 2014)

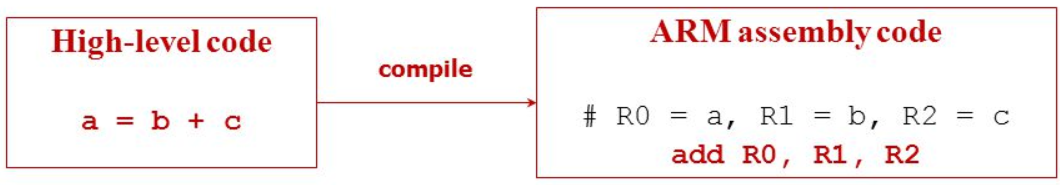


Figure 7: The conversion of High-Level C, to Low Level ARM. (Suh, 2015)

# Registers in the ARM7TDMI

The ARM7TDMI microprocessor does not contain any addressable "memory space". Instead, internally it implements several registers that can be used in various operating modes, but none of these are directly accessible, they can only be used by software running on the ARM7TDMI. (Campbell, 2017) The ARM7TDMI Microprocessor has a total of thirty-seven registers – thirty-one general-purpose 32-bit registers and six status registers - but these cannot all be seen at once. The processor state and operating mode dictate which registers are available to the programmer. In ARM state, 16 general registers and one or two status registers are visible at any one time. The ARM state register set contains 16 directly accessible registers: R0 to R15. All of these except R15 are general-purpose and may be used to hold either data or address values. In addition to these, there is a seventeenth register known as the CPSR which contains condition code flags and the current mode bits. Another special-purpose register in the ARM7TDMI is known as the link register, which holds the address to return to when a function call completes. Register fourteen is used as the subroutine Link Register (LR) in the microprocessor. In privileged operating modes, an additional register known as the Saved Program Status Register (SPSR), is accessible. This register contains condition code flags and the mode bits which were saved as a result of an exception which caused entry into the current mode. (ATMEL, 2004)

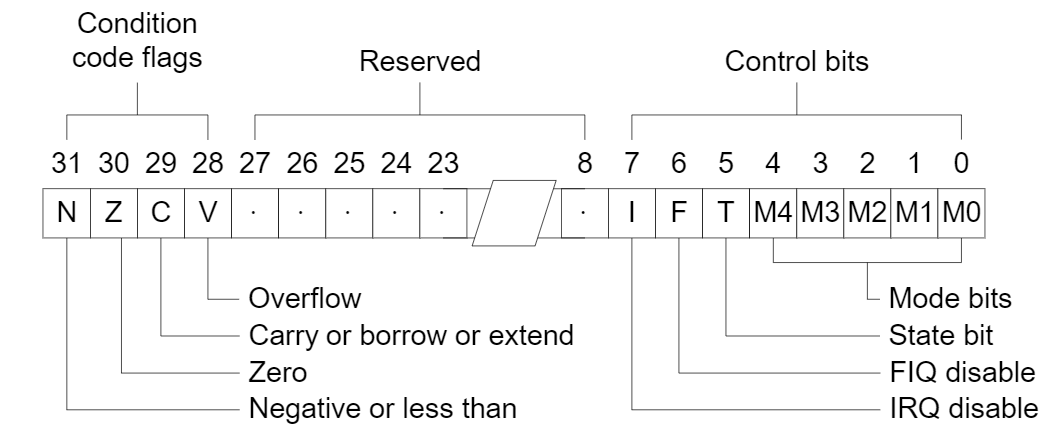


Figure 8: Program status register format ATMEL (2004)

## Control Bits

The bottom eight bits of the program status registers are known collectively as the control bits. They are the, interrupt disable bits, T bits and mode bits. These control bits change whenever an exception occurs. If the processor is operating in a privileged mode, these bits can be manipulated by software.

The I and F bits are known as the Interrupt Disable Bits. When the I bit is set, the Interrupt Request line (IRQ) interrupts are disabled. (ATMEL, 2004) IRQs are direct hardware lines over which devices can send interrupt signals to the microprocessor. (Beal, 2014) When the F bit is set, FIQ interrupts are disabled. FIQs are a higher priority interrupt request, prioritized by disabling all IRQ and FIQ handlers during request servicing meaning no other interrupts can occur during the processing of the active FIQ interrupt. (Kershaw et al., 2009)

The T bit reflects the operating state. When the T bit is set, the processor is executing in Thumb state and when the T bit is clear, the processor executing in ARM state. This operating state is reflected on the external signal TBIT. (ATMEL, 2004)

# Cache and the ARM7TDMI

The ARM7TDMI microprocessor does not have an memory management unit (Lee et al., 2004), instruction cache or data cache and is mainly used as a controller core rather than for data processing. Hence, It is of great importance to reduce memory accesses as far as possible when applying it to processing applications. (Ramkishor and Gunashree, 2001) It is known that on chip caches using static RAM consume power in the range of 25% to 45% of the total chip power. This is why its omission greatly improves power efficiency. (Panda et al., 2012) For situations where a cache is necessary, the ARM720T hard macrocell contains the ARM7TDMI core as well as a 8kb unified cache and a Memory Management Unit (MMU). (ATMEL, 2004)

# Memory in the ARM7TDMI

The ARM7TDMI microprocessor treats memory as a linear collection of bytes numbered in ascending order from zero. For example, bytes zero to three might hold the first 32 bits and bytes four to seven may hold the second. The ARM7TDMI processor is bi-endian meaning it can treat words in memory as being stored in either big or little endian, with little endian being the traditional format for ARM Processors. Little endian format on the microprocessor can be quite beneficial since numbers are calculated by the CPU starting with the least significant digits, meaning little-endian numbers are already set up for the processing order. (ATMEL, 2004)

Since the ARM7TDMI does not offer support for unaligned memory accesses, the processor features workarounds to handle both unaligned data stores and unaligned data reads. For unaligned data stores a half word store has register data bits [15:0] duplicated across the data bus and the memory controller will ignore the least significant bit of the address. For a byte store, register data bits [7:0] are duplicated on all four-byte lanes of the data bus. For unaligned data reads, a half word read, the selected half word is placed on the bottom [15:0] bits in the register and the remaining bits are cleared to zero with the memory controller ignoring the least significant bit of the address. For a byte read, the selected byte is placed on bits [7:0] in the destination register and the remaining bits of the register are filled with zeros by the core.(2008)(ARM, 2008)

## The ARM7TDMI Instruction Set

ARM code has a simple and concise operand encoding scheme, which facilitates fixed-length 32-bit instructions and fast decoding with most operations being performed on registers. (Cambridge, 2006) The ARM7TDMI features ten different addressing modes, these are:

|  |  |
| --- | --- |
| Name | ARM Example |
| Register to Register | MOV R0, R1 |
| Absolute | LDR R0, MEM |
| Literal | MOV R0, #15  ADD R1, R2, #12 |
| Indexed, Base | LDR R0, [R1] |
| Pre-Indexed, Base with Displacement | LDR R0, [R1, #4] |
| Pre-Indexed, Auto-Indexing | LDR R0, [R1, #4]! |
| Post-Indexing, Auto-Indexed | LDR R0, [R1], #4 |
| Double Register Indirect | LDR R0, [R1, R2] |
| Double Register Indirect with Scaling | LDR R0, [R1, r2, LSL #2] |
| Program Counter Relative | DR R0, [PC, #offset] |

Table 2: Summary of Arm Addressing Modes (Regina, 2012)

The ARM7TDMI Microprocessor supports some basic logical operators: AND, OR, XOR and BIC. It also supports a few stack operations such as: STM, LDM, PUSH, POP. These instructions use the Stack Pointer (SP) which by convention is Register 13.

## Multi-byte Shifting on the ARM7TDMI

The ARM7TDMI has a 32-bit barrel shifter that can simultaneously execute shift and rotation instructions with ALU operations. (Chang et al., 2003) The shifter operand is represented by the least-significant 12 bits of the instruction. It can take one of eleven forms. For immediate operands, the assembler will make substitutions of comparable instructions in order to create the desired immediate operand.(Mirzaei, 2017) For example:

CMP R0, #-1 **⟶** CMN R0, #1

BIBLIOGRAPHY

ARM 2008. What does the ARM7TDMI core read/write when using non aligned addresses? *In:* ARTICLES, A. T. S. K. (ed.).

ATMEL. 2004. ARM7TDMI Technical Reference Manual. *Infocentre ARM* [Online]. Available: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/CACBCAAE.html>.

BEAL, V. 2014. IRQ - interrupt request line. *Webopedia.*

BONG-HO, K. 1998. Introduction to Thumb. *Characteristics of ARM Processor***,** 2-10.

CAMBRIDGE, U. O. 2006. *Microprocessor Design and the ARM* [Online]. Available: <http://www-mdp.eng.cam.ac.uk/web/library/enginfo/mdp_micro/lecture2/lecture2-5-1.html> [Accessed 03/09/18].

CAMPBELL, C. 2017. *ARM7TDMI memory spaces* [Online]. Available: <https://community.arm.com/processors/f/discussions/9032/arm7tdmi-memory-spaces> [Accessed 04/09/18].

CHANG, H., LEE, W. & SUNG, W. Optimization of power consumption for an ARM7-based multimedia handheld device. Circuits and Systems, 2003. ISCAS'03. Proceedings of the 2003 International Symposium on, 2003. IEEE, V-V.

DIBA, M. 2014. ARM Arithmetic and Bitweise Instructions. *Computer Organization and Systems Programming***,** 5-20.

GOUDGE, L. & SEGARS, S. Thumb: reducing the cost of 32-bit RISC performance in portable and consumer applications. Compcon'96.'Technologies for the Information Superhighway'Digest of Papers, 1996. IEEE, 176-181.

KERSHAW, D., MCNIVEN, J. I., KEFFORD, D. L. & MANSELL, D. H. 2009. Data processing apparatus and method for controlling access to registers. Google Patents.

LEE, I., CHOI, Y., CHO, Y., JOO, Y., LIM, H., SHIM, H. G. L. H., CHANG, N. J. I. D. & COMPUTERS, T. O. 2004. Web-based energy exploration tool for embedded systems. 21**,** 572-586.

MCDERMOTT, M. 2008. The ARM Instruction Set Architecture. *The University of Texas at Austin***,** 2-2.

MIRZAEI, S. 2017. ARM7TDMI Instruction Set Reference. California State University, Northridge.

PANDA, P. R., DUTT, N. D. & NICOLAU, A. 2012. *Memory issues in embedded systems-on-chip: optimizations and exploration*, Springer Science & Business Media.

RAMKISHOR, K. & GUNASHREE, V. Real time implementation of MPEG-4 video decoder on ARM7TDMI. Intelligent Multimedia, Video and Speech Processing, 2001. Proceedings of 2001 International Symposium on, 2001. IEEE, 522-526.

REGINA, U. O. 2012. *ARM Addressing Modes* [Online]. Available: <ftp://www.cs.uregina.ca/pub/class/301/ARM-addressing/lecture.html> [Accessed 03/09/18].

SEGARS, S., CLARKE, K. & GOUDGE, L. J. I. M. 1995. Embedded control problems, Thumb, and the ARM7TDMI. 22-30.

SHEE, S. L. 2004. The ARM Architecture. Sydney: UNSW.

SUH, P. T. 2015. Arm Instructions I. *Computer Science Engineering, Korea University***,** 5.

THOMAS, D. 2012. *ARM: Introduction to ARM: Conditional Execution* [Online]. Available: <http://www.davespace.co.uk/arm/introduction-to-arm/conditional.html> [Accessed 02/09/18].

VARMA, R. R. 2015. Introduction to ARM Architecture.