**ARM Processor Architecture**

**Matt Lamont**

**CSCI 4591**

**Spring 2014**

**Table of Contents**

Introduction……………………………………………………………………………..………………..……………………………..3

History of the ARM Processor………………………………………..………………………………………………………….3

RISC Based Architecture………………………………………..…………………………………………………………………..4

ARM Processor Modes………………………………………..…………………………………………………………………….6

ARM Registers………………………………………..………………………………………..………………………………………..7

Current Program Status Register (CPSR) and Saved Program Status Register (SPSR)………………….9

ARM Memory Access………………………………………..……………………………………………………………………..10

ARM Instruction Set………………………………………..………………………………………..……………………………..12

ARM Barrel Shifter………………………………………..………………………………………..……………………………….14

ARM Endianess………………………………………..………………………………………………………………………………14

Thumb Instruction Set………………………………………..…………………………….……………………………………..14

Progression of the ARM Processor………………………………………..……………………………….………………..15

The Next Generation……………………………………………………………………..………………………………………..17

Sources………………………………………..………………………………………..………………………………………………..19

Index………………………………………..………………………………………..……………………………………………………20

**Introduction**

The most ubiquitous mobile processor utilized by modern mobile computing devices such as smart phones and tablets is the ARM processor. Developed by ARM Holdings and first released in April 1985, the ARM Processor has found its way into many common computing products such as Apple I-phones and Android based smartphones. As of 2010 ARM processors can be found in 95% of smartphones, 35% of televisions, and even 10% of the mobile computer architectures. The ARM’s RISC (reduced instruction set computing) based architecture makes it ideal for mobile architectures, because it lowers the amount of needed transistors, leading to reduced heat and power consumption.

**History of the ARM Processor**

Development for the very first ARM processor was initiated by a British electronics producer named “Acorn Computers”. While searching for a powerful and efficient processor to integrate into their personal computer; Acorn tested many of the top processors developed by National Semiconductor and Motorola. After none of these on the market processors met Acorn’s satisfaction, they decided on the innovative route of developing their own.

 The very first incarnation of the ARM processor was created by Sophie Wilson: an Acorn engineer, who wrote an initial instruction set based on research of the Berkley RISC project. This instruction set was merely a simulation implemented with BBC Basic programming language and run on a BBC microcomputer. However, this basic simulation was enough to convince Acorn Ceo Hermann Hauser to fund a hardware version of Wilson’s instruction set in 1983. Two years later, in April 1985 the very first ARM1 processor was fabricated by VLSI Technology. Amazingly, the ARM1 worked right off the assembly line. The ARM2: a newer variation of the processor was released just a year later and in 1987, the ARM processor found its first full time processing job in the Acorn Archimedes personal computer. Acorn even landed a Queen’s Award for Technology, for development of the ARM processor.

**Sophie Wilson: Designer of ARM Instruction Set**

Acorn’s first commercial processor, the ARM2, consisted of a 32 bit data bus, with 27 registers, and was able to utilize a 26 bit address space. One of the most incredible feats accomplished by the ARM2 processor was its significantly reduced use of transistors. As compared to the Motorola’s model 68000, which contained 68,000 transistors; the ARM2 cut this count in half with a total transistor count of 30,000. As a result, the ARM2 consumed much less power and created less internal heat than the competition, while still maintaining staggering performance. Acorn continued to add performance upgrades in subsequent models of the ARM, such as an integrated cache memory and extending the accessible address space to 32 bits.

**RISC Based Architecture**

The original ARM processor was based on the research of the Berkeley RISC project being performed by David Patterson at the University of California Berkeley and the MIPS project at Stanford University. That is why ARM is an acronym for Acorn RISC Machine. The idea behind these projects – especially the RISC project was to reduce the total number of transistors being used to decode the architecture’s instruction set. Patterson discovered that some of the most popular contemporary operating systems, such as Unix, could be compiled with only 30% of the instruction set provided by the hardware. If whole operating systems ignored nearly 70% of an architecture’s instruction set, then many of these instructions must be obsolete and therefore removable. This is where the name RISC (reduced instruction set computing) is derived from. After removing much of the decoding hardware for the unused instructions; the RISC project inserted many more registers in the empty space. The first RISC machine contained a total 78 32-bit registers and only 31 instructions, which was a stark contrast to Motorola’s current 68000 processor which contained 16 32-bit registers and 56 instructions.

However, the RISC approach to computer instruction sets forces more assembly code to be written to complete most operations. Many of the traditional instructions sets included several variations of the same command, such as a subtract command that took its two operands from registers or a separate subtract command that took operands from memory. In the RISC architecture, only a single subtract command was available: one in which both operands had to be stored in registers. As a result, an assembly programmer working with a RISC architecture, must include additional instructions to load operands from memory into available registers and then call a subtract instruction. The following code segments demonstrate the difference between programming with a RISC Architecture versus the traditional architecture:

|  |
| --- |
| **RISC Subtract Instruction**  mov register1 , mem\_location\_X  mov register2 , mem\_location\_Y  subtract register1 , register2 |

|  |
| --- |
| **Traditional Subtract Instruction**  Subtract mem\_location\_X , mem\_location\_Y |

The traditional subtract instruction encloses several micro-instructions to carry out fetching and decoding the memory locations of the operands. Consequently, all these micro-instructions must be separately manufactured into the processor hardware. This can take up a lot of space on the processor chip and is potentially wasteful if this particular instruction is almost never utilized by assembly programmers. However, a main benefit of the traditional instruction set, is that it allows increased code density when assembly programming (i.e. subtracting two numbers takes one line of code instead of three for RISC).

Because the RISC based architecture eliminates extraneous instructions and fills the empty hardware space with additional registers; it cannot achieve the same degree of code density. However, since the RISC architecture contains so many registers and forces operands to be stored in registers, the access speed of values is vastly increased over the regular memory accesses of the traditional architecture. When the first RISC machine was tested in 1982, the test program was 30% larger than the same program written for a 32-bit VAX 11/780 processor that used a traditional instruction set architecture. The RISC machine ran just as quickly as the VAX and could theoretically run much faster once improvements had been made to the faulty RISC prototype. This test faltered the long held belief that a higher code density, by using a large instruction set, provided better performance. Amazingly, the RISC architecture was able to achieve this feat, using half the total instructions of the VAX, and half of the total transistor count. It was this high performance of reduced instruction sets and fewer transistors that attracted Sophie Wilson and Acorn computers to adapt the RISC architecture into the first ARM processor design.

**ARM Processor Modes**

The ARM processor has nine different modes to operate under. These modes are divided up into privileged and unprivileged modes. The only unprivileged mode is User Mode, because it does not have access to all the instructions or registers. All other modes on the other hand are privileged and allowed to access all instructions and registers.

1. **User**

This is an unprivileged mode that most programs run under (most common mode). However, this mode does not have access to all the instructions and registers of the hardware, such as those used to process interrupts.

1. **FIQ (Fast Interrupt)**

This privileged mode is entered when a high priority or fast interrupt is encountered. Fast interrupts force the program counter and condition code registers onto the stack and do not allow the program to proceed until the interrupt is handled. The fast interrupt is really only used to handle the most critical resources.

1. **IRQ (Normal Interrupt)**

This privileged mode is entered when a normal priority interrupt is encountered. This mode is nearly the same as FIQ, except the interrupt is lower priority and the program counter is not stopped and pushed onto the stack.

1. **Supervisor**

This privileged mode is entered when upon reset or if a software interrupt is encountered. A software interrupt usually occurs from exceptions caused from program execution, which cannot be handled by the program code.

1. **Abort**

This privileged mode is entered if a memory access violation is encountered such as a pre-fetch abort or data abort exception.

1. **Undefined**

This privileged mode is entered if undefined instructions have been encountered in the program.

1. **System**

System mode is the same as user-mode, except that it is privileged. This means that the system mode can be switched over from user mode in the event that an instruction tries to write to the bits of the CPSR register.

1. **Monitor Mode**

This is a privileged mode that was introduced to support ARM’s TrustZone extension in newer ARM models. TrustZone is ARM’s hardware based security control, which creates two separate virtual processors or “worlds”. The “worlds” can operate independently of each other while still utilizing the same physical processor. This disallows a “less trusted” world from accessing the processes and information from a “more trusted” world.

1. **Hypervisor Mode**

This is a privileged hypervisor mode that was introduced into newer versions of ARM processors to support virtualization. This mode is commonly used when a guest operating system is being run virtually on within a host operating system (the OS native to the physical hardware).

Each of these processor modes have varying access to the registers and many of the registers are “banked” by the current mode.

**ARM Registers**

Modern ARM processors contain 37 32-bit registers. These include:

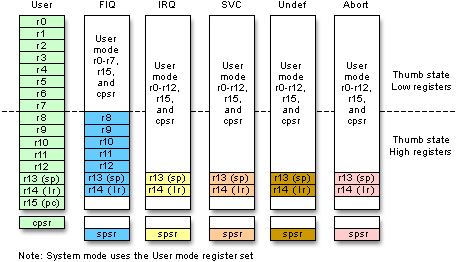
* 30 general purpose registers
* 1 dedicated program counter register
* 1 dedicated program status register
* 5 saved program status registers

These registers are “banked” by how accessible they are based on the current processor mode. The following registers can be accessed by all processor modes at any time:

* General Registers r0-r12
* Stack pointer register r13 (points to the current frame of the program stack)
* Link Register r14 (contains return address for function calls)
* Program counter register r15 (points to the current instruction of program)
* Current Program Status Register (contains status flags such as Carry, Overflow, and Interrupt flags).

The processor modes that are designated as privileged (all modes except user mode) have access to the following “banked” registers:

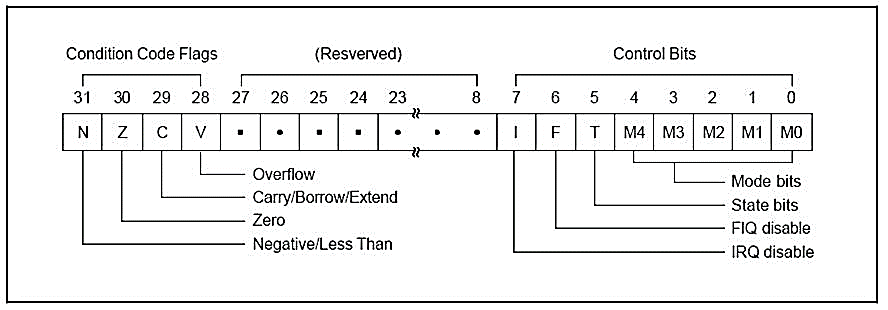
* Saved Program Status Register (when an interrupt is encountered, the contents of the current program status register “CPSR” are copied to the “SPSR” for use in the handling the interrupt).



**Available Registers to each Processor Mode**

**Current Program Status Register (CPSR) and Saved Program Status Register (SPSR)**

These 32-bit status registers contain the flags utilized by the program. During normal execution of the program the status bits of the CPSR are regularly altered and referenced. However, if an interrupt is encountered then the appropriate bit in the CPSR is set and all of the contents are copied into the SPSR register. The SPSR register is then used within the interrupt handling. The CPSR and SPSR registers are broken up like so:



**CPSR and SPSR Register Diagram**

* M (Bits 0-4) Define the current processor mode
  + 10000 = User Mode
  + 10001 = FIQ Mode
  + 10010 = IRQ Mode
  + 10011 = Supervisor Mode
  + 10111 = Abort Mode
  + 11011 = Undefined Mode
  + 11111 = System Mode
* T (Bit 5) Thumb State Bit
* F (Bit 6) FIQ disable bit
* I (Bit 7) IRQ disable bit
* A (Bit 8) imprecise data abort disable bit
* E (Bit 9) data endianess bit
* IT (Bit 10-15) if-then state bits (conditional handling)
* GE (Bits 16-19) greater than or equal bits
* DNM (Bits 20-23) do not modify bits
* J (Bit 24) Java state bit
* IT (Bit 25-26) also if-then state bits (conditional handling)
* Q (Bit 27) sticky overflow bit
* V (Bit 28) overflow bit
* C (Bit 29) carry/ borrow / extend bit
* Z (Bit 30) zero bit
* N (Bit 31) negative/less than bit

**ARM Memory Access**

The ARM runs on a reduced instruction set similar to that of a RISC machine. This also results in a “Load/Store” architecture, meaning that data can only be loaded or stored from memory into registers and the instructions do not actually access memory. This saves on the physical hardware implementation instructions at the cost of more instructions being necessary to attain a certain goal.

Most ARM Architectures only support aligned memory access. In other words, the offset of data to be written to memory can only be stored starting at an even boundary. As a result, only even address locations can be provided to ARM’s load and store instructions. Storing on even boundaries allows quicker memory access to help supplement ARM’s need for more instructions in software implementation. However, this quicker access tends to waste blocks of memory that happen to be on odd or unaligned boundaries.

A few ARM architectures provide an exception to this rule, such as some microcontrollers. Because microcontrollers tend to have smaller amounts of available memory; the ARM architecture allows access for data sizes of word or half-words (such as Boolean data types) on unaligned boundaries.

ARM architecture supports ten different addressing modes as follows:

1. **Register to Register**

This mode simply transfers data from register to another.

1. **Absolute**

Also known as direct addressing, this mode simply loads the contents of a memory location into the given register.

1. **Literal**

Also known as immediate values, this addressing mode is used to load data values directly into registers.

1. **Indexed, Base**

Also known as indirect register addressing, this mode retrieves the address of an operand from memory. This value at this address is then loaded into a register for further use.

1. **Pre-Indexed with displacement**

Also known as indirect register addressing with an offset, this mode accesses an operand from memory by adding a literal offset against an address found in a register. The value at the memory address is then loaded into a register for further use.

1. **Pre-Indexed with Auto-Indexing**

This is also known as indirect register addressing with pre-incrementing. This mode is the same as pre-indexing with displacement, except that an offset is automatically added to a base pointer before the address is accessed. After the memory access, the offset is added to the pointer again.

1. **Post-Indexing with Auto-Indexing**

Also known as indirect register addressing with a post increment, this mode is the same as the pre-indexing version, except that is does not initially add an offset to the base pointer.

1. **Double Register Indirect**

This addressing mode obtains an address to access by adding a base pointer in one register to an index in a separate register along with an offset. (Reg1 + Reg2 + Offset)

1. **Double Register Indirect with Scaling**

This mode is similar to Double Register Indirect, although it allows a scaling factor to be multiplied against the offset. This is useful for array calculations in which the size of the operand is many bytes wide.

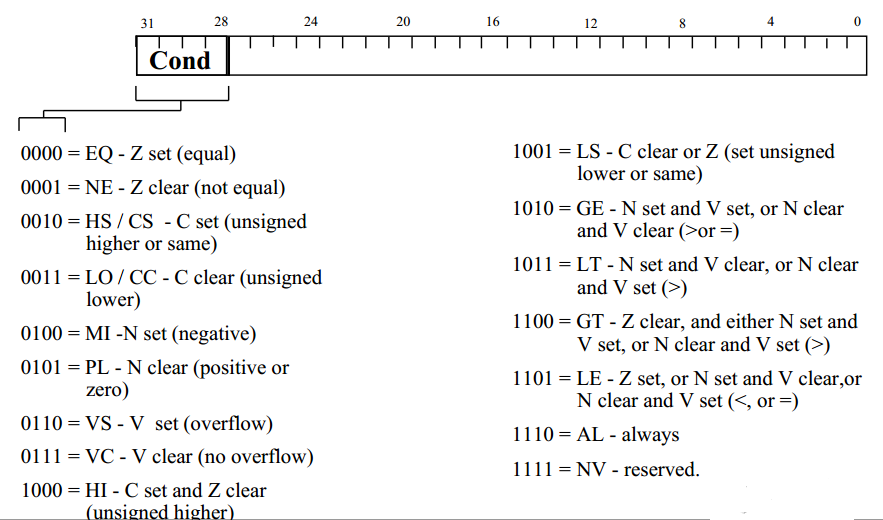
1. **Program Counter Relative**

This addressing mode adds an offset to the program counter (PC) register to access code instructions of the program.

**ARM Instruction Set**

The goal of the ARM processor instruction set is to increase speed and simplicity at the cost of code density and available instructions. To achieve this, every instruction in the traditional ARM instruction set is a fixed 32 bits wide (excluding the more the recent thumb instruction set that will be introduced later). This fixed 32 bit length, decreases much of the normal decoding time that takes place with instruction sets that contain a variety of instruction lengths. Fixed width also helps abate pipelining problems that typically accompany instruction sets that produce dense code.

All instructions in the ARM architecture contain a 4-bit field called the condition field.



The 16 possible options for the condition field, allow the instruction to be executed conditionally when reached by the program counter. For example, the instruction:

|  |
| --- |
| SUBEQ r0 , r1 , r2 |

would subtract r1 from r2 and place the result in r0 if the zero flag is set. Placing a condition field in every instruction means that branches that are normally executed in other architectures can be eliminated. As a result, the pipeline will not be stalled and there is not additional processing or subroutine calls to execute the branch. However, the ARM processor is forced to evaluate the condition field of all instructions when processing conditional execution. This evaluation only takes one cycle though, compared to three or more cycles that are typically taken up by traditional conditional branching. Because, the ARM processor evaluates the condition field of all instructions, it does not incorporate a branch predictor.

The following is a list of the most common instructions native to the ARM instruction set:

1. Arithmetic Operations
   1. ADD (reg1 + reg2)
   2. ADC (reg1 + reg2 + carry)
   3. SUB (reg1 - reg2)
   4. SBC (reg1 - reg2 + carry - 1)
   5. RSB (reg2 + reg1)
   6. RSC (reg2 + reg1 + carry - 1)
   7. MUL (reg1 \* reg2)
   8. MLA (reg1 \* reg2 + reg3)
2. Comparison Operations
   1. CMP (reg1 - reg2) update condition flags
   2. CMN (reg1 + reg2) update condition flags
   3. TST (reg1 AND reg2) update condition flags
   4. TEQ (reg1 EOR reg2) update condition flags
3. Logical Operations
   1. AND (reg1 AND reg2)
   2. EOR (reg1 EOR reg2)
   3. ORR (reg1 OR reg2)
   4. BIC (reg1 AND NOT reg2)
4. Data Movement Operations
   1. MOV (move data from reg1 to reg2)
   2. MVN (move NOT data from reg1 to reg2)
   3. LDR (load word or byte)
   4. STR (store word or byte)
   5. LDRB (load word or byte using offset)
   6. STRB (store word or byte using offset)
   7. LDM (loads multiple value from memory for program stack)
   8. STM (stores multiple registers in memory for program stack)
5. Interrupt Operations
   1. SWI (software interrupt that also switches processing modes)

There are many more variations of these instructions included in the instruction set such as loading and storing half words, division, and bit shifting.

**ARM Barrel Shifter**

The ARM also has a special hardware piece called the barrel shifter that is responsible for completing the equivalent commands of shift and rotate found in other architectures. There are no native shift instructions to the ARM instruction set and this is where the barrel shifter takes over. Use of the barrel shifter is often included when carrying out other instructions such as multiply or divide. The main positive aspect of the barrel shifter is that it can be utilized in address and arithmetic calculations without incurring an additional processing overhead, because shifts can be completed in a single cycle.

The barrel shifter can shift bits left to effectively multiply the bit pattern by powers of 2 and can also shift right to achieve the opposite of effect of dividing by power of 2. Typical bit rotations can also be achieved with the Barrel Shifter. One of the most powerful uses of the Barrel Shifter is in calculating addresses. The shifter can quickly calculate operand addresses using a specified offset. This eliminates the need for extraneous instructions and usually only requires a single cycle to complete.

**ARM Endianess**

The ARM architecture can actually support both little Endian encoding and Big Endian encoding. This is achieved simply by setting the endianess bit that is found in bit 9 of the CPSR register. For all ARM architectures, little Endian encoding is the default.

**Thumb Instruction Set**

With the release of the ARM7TDMI processor in 1994 came the new Thumb Instruction Set. When the processor entered the “T” state, it would run the specified Thumb instruction set. The instructions for this set were only 16 bits wide as opposed to the 32 bit wide traditional instructions. However, to save hardware space, most of these Thumb instructions make use of the same 32-bit traditional instructions. Because instructions are half the size; much less memory is occupied by program code. This also results in much denser code that can be written with the traditional 32-bit instructions.

The cost of reduced instruction size, is the use of implicit operands. In the original ARM instructions, two operands could be specified for many instructions including arithmetic and logical operations. However, the Thumb instructions will often imply a specific register as an operand, such as the r1 register for the ADD instruction. Unlike traditional instructions; the Thumb set only incorporates condition fields into branching operations. While, this method saves bit space in the instruction; it also misses out on the normally efficient processing of branching conditional statements. These additional instructions also mean that a separate hardware decoder must also be implemented into the processor.

The second variation of the Thumb instruction set: the Thumb-2 provided an increased number of instructions by including extra 32-bit instructions. This new set consisted of a mix of 16 and 32-bit instructions.

**Progression of the ARM Processor**

Since its inception in 1983, the ARM processor has undergone many changes and improvements. Below is a list of the versions of the ARM processor that have been created by ARM Holdings. Not included in this list are the many ARM based processors that have been developed by other parties.

1. ARM1

The very first implementation of the processor that incorporated 32-bit instructions based on a RISC architecture and supported a 26-bit address space.

1. ARM2

Multiplication and swap instructions were added to the instruction set. The hardware was upgraded to include a memory controller and provided support for graphics and input/output operations. This processor could run a peak 4 Million Instructions per Second (MIPS) at 8 MHz clock frequency.

1. ARM3

This is the first ARM processor to include a built-in 4kb memory cache. This processor could run a peak 12 MIPS at 25 MHz clock frequency.

1. ARM6

This is the first architecture to extend the 26-bit address space to 32-bits. A coprocessor bus is also added to support additional processing cores. The best version of the ARM3 could run 28 MIPS at 33 MHz.

1. ARM7

This processor introduced the Thumb 16-bit instruction set, which allowed for better code density. The peak performance of the ARM7 reached 130 MIPS at 40 MHz clock frequency.

1. ARM7TDMI

The TDMI in this processor stands for Thumb+Debug+fast Multiplier+ enhanced ICE. This model included support for code debugging, the fast multiplier which incorporated bit shifting, and the enhanced ICEbreaker that was a debugging module capable of supporting breakpoints, watch points, and other features for successful debugging.

1. ARM7EJ

This version of the processor included a new 5-stage pipeline that allowed for greater throughput of instructions and computations. The Jazelle DBX (Direct Bytecode Execution) state was also included in this family signified by the capital “J”. The Jazelle DBX is an execution state like the normal Instruction set architecture and the Thumb instruction set architecture that allowed Java byte code to be executed on the hardware. This paved the way for the adoption of ARM for mobile devices such as Android that run mostly Java programs. The “E” signifies the enhanced DSP instruction set, which provided the processor with increased digital signal processing capabilities.

1. ARM8

This family of processors allowed for double-bandwidth memory, which increased memory read/write rates by allowing transfers of data on both the leading and falling edges of the clock signal. Processors developed under this family reached 84 MIPS at 72 MHz

1. ARM9TDMI

This family was a continuation of the support for the Thumb instruction set and supported larger MMU (memory management units) and MPU (memory protection units) to increase security and data protection. Processors developed under this family reached 200 MIPS at 180 MHz

1. ARM9E

This family incorporated many instructions sets (Jazelle, Thumb, Regular, and Enhanced DSP) onto a single chip. This family also contained TCM’s (Tightly Coupled Memory) which is similar to cache, in data reading efficiency, but without the unpredictability typically associated with caching data. One of the processors created under this family: the ARM996HS; had no clock. This processor, performed asynchronous computations at the benefit of reduced power use. Processors developed under this family reached 220 MIPS at 200 MHz

1. ARM10E

This family increased total caches sizes as well as the size of the memory protection units. A 6-stage pipeline was designed to further increase the throughput of instructions. This family also introduced the VFP (Vector Floating Point) instruction set, which was effective for signal processing and media acceleration.

1. ARM11

This family saw an increase to an 8-stage pipeline as well as the addition of the SIMD (Single Instruction Multiple Data) instruction set, which was a more efficient and powerful vector floating point operations set than VFP. SIMD was utilized for the same instruction being executed across a large amount of data such as decoding an MP3 file.

The Trustzone processing mode was also included, to allow a hardware based separation of simultaneous processes. The ARM1176JZ(F)-S core under this family could perform 965 DMIPS at 772 MHz DMIPS (Dhrystone MIPS) was a new architecture independent measure of performance. It typically measured processing performance based on a benchmarking program that would be characteristic of a common workload.

1. Cortex-M

This family of chips is an adaptation of the typical ARM architecture suitable for use in microcontrollers. In order to reduce the size and resources needed to fabricate these processors; the Cortex-M family utilized the compact Thumb instruction set and generally removed processor cache and TCM’s (tightly coupled memory). Most Cortex-M processors run about 1.25 DMIPS per MHz

1. Cortex-R

This family of processors is known for their high performance due to increased clock frequencies and high error resistance. Low latency memory access units as well as memory protection units are found in all Cortex-R processors. Cortex-R processors are mainly used in embedded systems, where speed and data correctness are critical.

1. Cortex-A

This family of processors is known for power. Cortex-A processors incorporate super-scalar pipelining, SIMD instruction set for media processing, dedicated interrupt controllers, and hardware virtualization. Cortex-A processors make up nearly all of the mobile computing market and even inspire other company’s smartphone processors such as Qualcomm’s Snapdragon line. Any modern technology that is considered “smart” (smartphones, smart televisions, smart navigation, gaming consoles) probably contains a Cortex-A processor. These processors can achieve around 4 DMIPS per MHz per core.

**The Next Generation**

The newest development by ARM Holdings comes in the form of a brand new instruction set architecture. The AARCH64 is the first 64-bit instruction set created for ARM processors. A 64-bit address space is now possible with this new architecture along with user-space compatibility with older 32 bit instructions. The AARCH64 architecture also includes new hardware cryptographic instructions that include encoding, decoding, hashing, and key generation. Because of the wide adoption and development for ARM’s traditional 32-bit instruction set architecture; there is not much support for 64-bit development yet.

**Sources**

1. ARM Holdings. "ARM Information Center." *ARM Information Center*. N.p., n.d. Web. 02 Apr. 2014.
2. Multiple. "The Status Register." *- ARMwiki*. N.p., n.d. Web. 02 Apr. 2014.
3. Department of Computer Science, University of Regina. "ARM Addressing Modes."*Addressing Lab*. N.p., n.d. Web. 03 Apr. 2014.
4. "Arm strengthens Java compilers: New 16-Bit Thumb-2EE Instructions Conserve System Memory" by Tom R. Halfhill 2005.
5. "ARM\_Architecture\_Overview." University of Michigan Electrical Engineering. N.p., n.d. Web. 17 Apr. 2014.
6. "ARM Architecture Reference Manual." University of Texas Computer Science. N.p., n.d. Web. 17 Apr. 2014.
7. "Cortex-M And Classical Series ARM Architecture Comparisons." All D and R. N.p., n.d. Web. 17 Apr. 2014.
8. "ARM11 Processor Family." - ARM. N.p., n.d. Web. 17 Apr. 2014.
9. "ARM Extends Cortex Family with First Processor Optimized for FPGA." ARM Extends Cortex Family with First Processor Optimized…. N.p., n.d. Web. 17 Apr. 2014.
10. "ARM Discloses Technical Details Of The Next Version Of The ARM Architecture." ARM Discloses Technical Details Of The Next Version Of The…. N.p., n.d. Web. 17 Apr. 2014.
11. "ARM Instruction Set Architecture." University of Texas. N.p., n.d. Web. 17 Apr. 2014. <http://users.ece.utexas.edu/~valvano/EE345M/Arm\_EE382N\_4.pdf>.
12. "ARM Instruction Sets." National Central University. N.p., n.d. Web. 17 Apr. 2014. <http://www.ee.ncu.edu.tw/~jfli/computer/lecture/ch04.pdf>.

**Index**

68000 processor, 4

AARCH64, 17

Abort, 7, 10

Absolute, 11

Acorn, 3, 4, 6

ADC, 13

ADD, 13, 15

address space, 4, 15, 17

aligned, 11

AND, 13

Archimedes, 4

Arithmetic Operations, 13

ARM Holdings, 3, 15, 17, 19

ARM Registers, 2, 7

ARM1, 3, 15

ARM10E, 16

ARM11, 17, 19

ARM1176JZ(F)-S, 17

ARM2, 3, 4, 15

ARM3, 15

ARM6, 15

ARM7, 15, 16

ARM7EJ, 16

ARM7TDMI, 14, 16

ARM8, 16

ARM9E, 16

ARM9TDMI, 16

Barrel Shifter, 2, 14

BBC Basic, 3

BIC, 13

branch predictor, 13

cache, 4, 16, 17

carry/ borrow / extend bit, 10

CMN, 13

CMP, 13

Comparison Operations, 13

condition field, 12, 13

Cortex-A, 17

Cortex-M, 17, 19

Cortex-R, 17

CPSR, 2, 7, 8, 9, 14

Current Program Status Register, 2, 8, 9

data endianess bit, 10

Data Movement Operations, 13

David Patterson, 4

Dhrystone MIPS, 17

Direct Bytecode Execution, 16

do not modify bits, 10

Double Register Indirect, 12

Double Register Indirect with Scaling, 12

endianess, 10, 14

enhanced ICE, 16

EOR, 13

FIQ, 6, 10

greater than or equal bits, 10

Hermann Hauser, 3

Hypervisor Mode, 7

if-then state bits, 10

imprecise data abort disable bit, 10

Indexed, Base, 11

instruction set, 3, 4, 5, 6, 10, 12, 13, 14, 15, 16, 17

Interrupt Operations, 14

IRQ, 6, 10

Java state bit, 10

Jazelle DBX, 16

LDM, 14

LDR, 13

LDRB, 14

Link Register, 8

Literal, 11

little Endian, 14

Logical Operations, 13

memory cache, 15

memory management units, 16

microcontrollers, 11, 17

micro-instructions, 5

MLA, 13

MMU, 16

Monitor Mode, 7

Motorola, 3, 4

MOV, 13

MPU, 16

MUL, 13

Multiplier, 16

MVN, 13

National Semiconductor, 3

negative/less than bit, 10

ORR, 13

overflow bit, 10

Post-Indexing with Auto-Indexing, 11

Pre-Indexed with Auto-Indexing, 11

Pre-Indexed with displacement, 11

Processor Modes, 2, 6

Program counter, 8

Program Counter Relative, 12

Queen’s Award for Technology, 4

Register to Register, 11

RISC, 2, 3, 4, 5, 10, 15

RSB, 13

RSC, 13

Saved Program Status Register, 2, 8, 9

SBC, 13

SIMD, 17

Sophie, 3, 6

Stack pointer register, 8

sticky overflow bit, 10

STM, 14

STR, 14

STRB, 14

SUB, 13

Supervisor, 6, 10

SWI, 14

System, 7, 10, 19

TCM, 16, 17

TEQ, 13

Thumb Instruction Set, 2, 14

Thumb State Bit, 10

Thumb-2, 15

TrustZone, 7

TST, 13

Undefined, 7, 10

Unix, 4

User, 6, 10

VAX 11/780 processor, 6

VFP, 17

Wilson, 3, 6

zero bit, 10