CS271
Winter 2015

ARMv7: A Small-Scale Architecture

Isaac Stallcup

The ARMv7 architecture is one in a line of processor architectures designed by Advanced RISC Machines. The language itself is designed to be simple and power-efficient, making it useful for optimization and small-scale applications. ARMv7 is also a Reduced Instruction Set Computer architecture, which means as compared to CISC architectures like Intel x86, ARM uses more instructions to complete a task but completes the task faster and is easier to optimize.

ARMv7 is divided into three subsets: ARMv7-A, ARMv7-R and ARMv7-M. The A in ARMv7-A stands for Application Profile; ARMv7-A uses virtual memory in its operations and is therefore ideal for applications. ARMv7-R does not use virtual memory management and so is more tied by physical memory limitations. ARMv7-M is a subset of ARMv7-R, and uses a different instruction set and physical memory.

ARMv7-A uses a flat memory model with 2^{32} 8-bit bytes available for use; Intel x86 architectures have a larger addressing limit and therefore more addresses available. ARM also supports several different data types; 8-bit bytes, 16-bit halfwords, 32-bit words and 64-bit doublewords. In x86 architecture, bytes are still 8 bits but words are 16 bits and doublewords are 32 bits. Quad words replace doublewords as the 64-bit operand. In ARM, the minimum addressable unit is 8 bits, or 1 byte.

ARMv7 has 17 registers to Intel x86's 16; 12 of the ARMv7 registers are "general purpose" as opposed to eight of the x86 registers. These registers are named R0-R12. In addition, these twelve registers are truly general purpose, while those in x86 are often used for other purposes (EBP, ESP, etc). There are similarities between the two however, as there are three reserved registers in ARM that mimic special registers in x86, plus an additional register, the CPSR or current program status register. This register is analogous to the flags register in x86 architecture, however with fewer flags included.

These special registers are the SP, LR, and PC, usually denoted R13-15 respectively. The first three of these registers, SP or the stack pointer, mimics x86's ESP and points to the current location of the stack in the program. LR, the link register, holds a return address when certain subroutines are run, and PC (the program counter) holds the location of the program on the stack.

When the program performs instructions, each instruction's size is added to the PC to progress the program along its memory space. PC begins at the location 0x00000000 and can be modified by branching and linked branching, topics that will be discussed later. Each instruction follows the syntax

```
INSTRUCTION{Condition}<flag> Rn, Rm, {Roptional}
```

The condition space is used for an instruction's special conditions; each instruction has the potential to have many different conditions, and some (like loading and saving to the stack) can even determine fundamental aspects of the program such as if the stack operates up or down. This feature is not included in x86 architecture and is unique to ARM.

Another unique aspect of instructions in ARM is the flag space in the instruction template given above. Instructions can be written to execute only if a certain flag condition is met. For example

```
MOVVS r0, r1
```

will only execute if the overflow flag has been set. This feature is not in x86 assembly, and promotes much greater code density and optimization. Entire sections of x86 assembly can be removed with this feature, and it allows an instruction to check flags and execute in the same cycle speeding code up greatly. Additionally, the flags are only updated from an arithmetic operation if an S is affixed to the end of the operation. This allows for more control than in x86 architecture; you can set a flag earlier in a program and have successive mathematical operations not tamper with this flag.

Finally, in the instruction template given above, and `Rn` and `Rm` are registers. The Roptional space, also for registers, is used for some operations, most notably addition, subtraction, and multiplication where it allows the storage of the result of the mathematical operation in a register. An interesting quirk of ARM is the lack of a division operator; division requires the inclusion of a separate instruction subset for floating opint numbers. The addition operator, as well as subtraction and multiplication operators, have this three-input syntax; they add the second and third inputs and store the result in the first register given. This is demonstrated in Example Program 1.

Example Program 1

```
GLOBAL Reset_Handler
AREA Reset, CODE, READONLY
EXPORT __Vectors
```

```
var DCD 0x00

__Vectors   DCD Reset_Handler

      ENTRY

Reset_Handler

      LDR r1, = 5 ;r1 = 0x00000005
      LDR r2, = 4 ;r1 = 0x00000004

      ADD r3, r2, r1 ;r3 = r2 + r1 = 0x00000009

END
```

Another useful feature of the Roperator space comes from a set of instructions related to barrel shifting; this allows for five shifts to be applied to the Roperator through suffixes to an operation in ARM as such:

```
MOV r1, r1, LSL #1
```

Where LSL #1 is interpreted as the C << instruction, effectively multiplying the contents of r1 by 2. This is another feature of ARM not present in x86.

An operation that is shared between Intel x86 and ARM is the comparison operation; like in x86, it does not produce a result but instead sets the value of certain flags in the CPSR register Loops use these comparison operations, as well as branch commands that serve as `jmp` instructions in x86 architecture. However some branch statements hold the return address in the LR register, making it easy to return from a jump to the point from which you jumped. An example of this branching behavior is given in example program 2:

Example Program 2

```
            GLOBAL Reset_Handler
            AREA Reset, CODE, READONLY
            EXPORT __Vectors

      sum DCD 0x00

      __Vectors   DCD Reset_Handler

            ENTRY

      Reset_Handler

            LDR r1, = 5 ;r1 = 0x00000005
            LDR r2, = 4 ;r2 = 0x00000004

            BL func1    ;branch with link to func1
                              ;r14 = 0xFFFFFFFF
                              ;r15 = 0x00000010
                  ;branch returns here in instruction path, with
                  ;r14 still 0x00000015 but r15=0x00000014, address
                  ;of add instruction below
            ADD r4, r2, r1    ;r4 = r2 + r1 = 0x00000009
            ADD r4, r3, r4    ;r4 = r3 + r4 = 0x0000000E

            B theend    ;break to the end

      func1
                  ;after branch, r14=0x00000015, r15 = 0x0000001C
            MOV r3, r1  ;r3 = 0x00000005

            BX r14              ;before instruction r15=0x0000001E, sets PC
                                ;to value pushed to r14 effectively returning
                                ;program to the place from where it branched

      theend

            END                 ;end program
```

Some processors running the ARM assembly have a hardware Floating Point Unit, integrated into the chip as in x86. This FPU, called the VFP, is a coprocessor that along with specialized code allows the use of floating point numbers in ARMv7. The VFP supports single and double precision IEEE.754 floating point values, and contains sixteen 64-bit or 32 32-bit registers for manipulating floating point values. Intel x86 contains eight 80-bit registers for using floating point values, and therefore the VFP is more prone to rounding errors. The VFP instructions also contain syntax for dividing numbers, something the default ARM syntax lacks.

ARMv7 is an architecture designed first and foremost for small-scale applications and optimization; there are many different aspects of the language that give it an advantage over Intel x86 architecture in this venue. It is an RISC language, which means even if more lines of code are required to complete a task it completes it faster, it uses less power and generates less heat. It also contains flag-modified instructions and built-in barrel shifting, as well as a three-operand instruction set that can eliminate lines of code that would otherwise be cumbersome in x86.

In all, ARMv7 is a versatile language built to be used in more than just home- and super-computing applications. Its focus on small-scale, efficient code has made it the most widely used assembly in the world and in the future it will continue to grow as everyday items we use become more and more wired with technology.

Example Program 1 x86 Assembly:

```
SECTION .text

      global main

main:

            push ebp
            mov ebp, esp

            mov eax, 5
            mov ebx, 4

            add   eax, ebx
            mov ecx, eax

            mov esp, ebp
            pop ebp

            mov eax, 0
            ret
```

Example Program 2 x86 Assembly:

```
SECTION .text

      global main

main:

            push ebp
            mov ebp, esp

            mov eax, 5
            mov ebx, 4

            jmp .func1

.post_f:
            add   eax, ebx
            mov   edx, eax
            add   edx, ecx

            jmp .theend

.func1:
            mov ecx, eax

            jmp .post_f

.theend:

            mov esp, ebp
            pop ebp

            mov eax, 0
            ret
```

Works Cited:

M. Levy, “The history of ARM architecture: from inception to IPO,” Convergence Promotions, CA, Special Report, 2005. Available: http://reds.heig-vd.ch/share/cours/reco/documents/thehistoryofthearmarchitecture.pdf

Staff, “ARM’s way”, Electronics Weekly.com, Apr. 1998. Available: http://www.electronicsweekly.com/news/archived/resources/arms-way-1998-04/

ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition, ARM Ltd., Cambridge, England, 2008; pp. 1-1147 Available: http://www.club.cc.cmu.edu/~mjrosenb/ARM%20v7%20Architecture%20Reference%20Manual.pdf

(2002). *Virtual Memory* Online Dictionary, FOLDOC Free On-Line Dictionary Of Computing. Available: http://foldoc.org/virtual+memory

Thomas, D., *Introduction to ARM*, davespace.co.uk. Mar., 2012. Available: http://www.davespace.co.uk/

ARM Infocenter. ARM Ltd., 2010. Available: http://infocenter.arm.com/help/index.jsp