ARM Version 7 Architecture Design: Comparisons to Intel x86

Corwin Perren

3/13/14

Oregon State University

Computer Science 271

Introduction

ARM based processors are some of the most common processors in use today. Their small form factor makes them a pefect fit for use in embedded applications that require more than a low powered microcontroller while still being space conscious. This, combined with their comparatively low power consumption in relation to Intel or AMD, has helped forefront them for use in devices such as cell phones, tablets, medical devices, point-of-sale systems, and many others. As such a common and popular processor type, one would expect the processor's architecture to be unique, powerful, and easy to use. This paper will cover some of the features of the ARM Version 7 architecture and how these features stack up against its x86 counterparts.

ARM History

The first ARM processor was released in 1985 by Acorn as a response to unmet needs of the company for a low latency processor that didn't use the overly slow CISC instruction set. When it was released, it outperformed the Intel 80286 processor that came out at roughly the same time while having significantly fewer transistors and costing signficantly less to manufacture [Ryzhyk, 2006]. This processer used what was later called the verion one architecture. By 1990, Acorn paired with Apple to produce a RISC processor for their Netwon PDA (using architecture version three), and spun off a seperate company for this project under the name ARM (Advanced RISC Machines), and focussed the company to the embedded market. As 2000 approached, highly advanced features began appearing on their version 5 architecture such as built in digital signal processing capabilities [Levy, 2005]. Multi-processor enhancements were added in 2001, along with virtualization options. Now on ARMv7, released in 2006, floating point support was enhanced and has been placed in uncountable numbers of devices. Over the years, their focus and tuning of the RISC architecture and miniaturization have led to their growing success.

Instruction Set Design

CPU Design

The first thing to note about the ARMv7 architecture is that it uses RISC(Reduced Instruction Set Computing), compared to the CISC (Complex Instruction Set Computer) design that Intel x86 uses. This means that for those who are used to writing code for intel processors, you will most likely find youself having to write more for ARM to acheive the same output. This is not necessarily a bad thing , and in fact has the potential to be faster, as will be covered later as well as giving the programmer finer control over how computations are done. A noticeable change that makes light of this would be the appending of an "S" character onto an instruction if you want it to set the cpu flags based on the outcome for ARMv7. In x86, this happens automatically (even if you'd prefer it not to), but in this case the programmer would be able to choose whether or not they would actually want to set them.

Endianness

ARMv7 by default is a bi-endian architecture meaning that either the chip manufacturer or board designer has control over whether the endianness of the processor will be little of big. Some manufacturers will leave a pin available so the end designer can specify it based on the specific needs of a design. Others come preconfigured with the endianness set interally. Processors using this architecture also have built in instructions to reverse the endianness of data so it can easily communicate with other processors or devices using alternate endian types with ease. Though ARMv7 is bi-endian, the default and most commonly used endianness is little-endian [Limited, 2010].

Registers and Data Types

This architecture uses 32 bit registers and can support data types down from an 8 bit byte up to a 64 bit integer stored between two registers. There are a total of 16 32-bit registers [Clements, 2014], which coming from intel's six (if you're lucky) is a welcome change. These registers are accessed using r0-r16 and support the same kinds of operations and uses that x86 does with the exception that the registers normally do not have as many pre-defined uses that have to be adhered to. You are able to define all the registers used for inputs, outputs, divisors, multipliers, shifters, etc.. for each instruction used, which when combined with the overall larger number of registers means that coding can be much nicer and easier than x86. For those ARMv7 processors that include the optional FPU, there would then be another 16 64-bit registers avaiable just for floating point operations, which again is very nice compared to Intel's eight.

Addressing and Addressing Modes

All version 7 ARM processors use 32 bit addressing, just the same as Intel processors [Limited, 2010]. The addressing modes are also similar. ARMv7 has the ability to address data using direct and indirect adressing, as well as loading immediates, with some slight variations compared to x86. Rather than having a single command to move data into and out of registers and memory, they was x86 does, ARMv7 has not only a different mov operator for different lengths of data, but it actually has seperate ones for moving data into and out of those registers and memory spaces. To move data into registers, you have to use the instruction "LDR" and to store what's currently in the register elsewhere, "STR". Those two instructions have letter modifications to declare the data size being moved, so "LDRB" would translate to "load a byte into the register". The same goes for the store instruction. Immediates are also slightly different to use as they require either a # or = sumbol in front of the number depending on which instruction is used. The "MOV" instruction does exist in ARMv7, however, it's actually part of the instrucion extension set called Thumb2 and will only allow you to load bytes into registers [Limited, 2010]. In this case, you would use the pound symbol to preface the immediate. Otherwise, LDR or STR should be used, depending on where you want it to end up, and the immediate should be prefaced with the equals sign. Otherwise, the differences between indirect/direct addressing in ARMv7 vs x86 are minimal. "LDR r0, r2" would load the contents of register two into register zero. "LDR r0, [r2]" would load the value stored in the memory location pointed to by the address stored in r2 into r0. To use scaling with indirect adressing, you have to use comma's rather than a plus and minus signs. Therefore, "mov eax, [ebx-edx]" would instead be "LDR r0, [r1,-r2]" [Clements, 2014].

Common Mathematical Instructions

Most common arithmatic and bitwise instructions available on x86 processors are available on ARMv7 including, but not limited to ADD, SUB, MUL, AND, OR (ORR), and SHL (ROL) and all are as easy to use as their x86 counterparts or even easier considering the free range you're allowed with ARMv7 register usage as mentioned before. There is one common one that is missing however, and that is the division instruction. Only a select few versions actually contain a hardware division unit, and those select few will have the instruction but it is not the norm [Limited, 2010]. In order to do division on the rest, you have to create your own division procedure using the other avaible instructions. This is completely do-able of course, but could potentially be challanging for those without a firm handle on bitwise operations. One of the most useful features of the mathematical instructions of this architecture is that they have built in conditional capabilities. So, for example, if in pseudocode you wanted to increment a variable by 10 if it was under 100, that'd require one compare, one jump, and an add instruction in x86 assmebly. To do this in ARMv7 assembly it would take two lines and no jumps. A simple "CMP r0, #100" with "ADDLO r0, #10" solves the issue very succintly and efficiently.

Jumping and Looping Constructs

Jumping in ARMv7 uses a different instruction name (Bxx) with fewer conditional branching options, but behaves exactly the same as x86. To do an unconditional branch, all ones needs to do it use the instruction "B" and then the label name [Clements, 2014]. To perform a conditional jump, "CMP" can be used or an "S" placed in the instruction you would like the create cpu flags for before using a conditional such as "BNE",

which stands for "break if not equal". Since the "S" flag must be appended manually, there is a potential for major code malfunction over a missed letter. Therefore, programming in assembly for ARMv7 requires a bit more detail and bookkeeping than IA32. Looping must be done using status flags on an instruction or from a compare, paired with a branch. There is no built in looping construct or count register, so the programmer has free reign to choose these as they see fit.

Procedures

The way procedures are done vaguely resembles x86, though without all the nice wrappers and macros and a few modifications. Calling a procedure is as simple as using the branch instruction "BL" (branch to subroutine) and a standard label name [Clements, 2014]. When "BL" is used, the return address is stored in a dedicated link register (lr), as opposed to getting pushed onto the stack, and when the programmer is ready to return to that address they must pass the link register back to the program counter by using "MOV PC, lr". Arguments can still be passed to the subroutines using registers, since there are actually enough for it not to pose as much of a problem, but arguments can also still be passed using the stack if needed.

The Stack

The ARMv7 stack grows downwards in memory and is used for basically the same purposes as its x86 counterpart and is stored in r13 [Clements, 2014]. Pushing and popping the stack is done using the normal store and load commands that are used with all other registers and memory spaces. Since there is no dedicated push and pop instruction to use, a programmer using the stack in ARM has to manually keep track of incrementing and decrementing the stack pointer at all times, as the x86 instruction took care of that for us. It's uses are still identical to that of x86; storing subroutine return addresses, providing temporary storage, and passing parameters to subroutines.

Notable Features

NEON SIMD (MPE)

These architecture extension allow interfacing with the on-chip media processing device which allows for hardware audio and video processing and decoding without the need for an off-die co-processor [Limited, 2010]. This device can also be used as a general purpose digital signal processor when not used for it's optimized media applications.

Thumb2 Extensions

Thumb2 extensions are requiered extensions for ARMv7 assembly, and add the ability to minimize code for size by using a combination of 16 and 32-bit instructions [Limited, 2010]. This can result in a 40% decrease in program size while only minimally decreasing efficiency [Ryzhyk, 2006]. This also enables the sofware-floating point linkage to maintain reverse compatiblity with older generation processors without an FPU.

Performance Notes

RISC vs CISC

With RISC laying the foundation for this artchitecture, there is the possiblity for code written to be more efficient than for CISC counterparts due to shorter pipelines and reduced cycles per instruction. However, it is also very easy for a programmer to make code that would be one instruction in CISC take significantly longer on this RISC architecture if they do not have good optimization skills or enough time to optimize their code throughly.

Number of Registers

The large number of usable registers can greatly improve the processing time for code executed in the ARMv7 architecture. Rather than having to constantly use the stack for passing parameters for procedures, or doing a significant amount of memory manipulation to save register data as is the case with x86, the programmer can instead use registers for as much of their coding as possible. Since fewer memory accesses will be made, the overall execution time will be significantly reduced.

Enhanced Math Instructions

The fact that most math operations can have every operand, input, and destination explicitly specified can greatly reduce the execution time that would normally be required on x86 for setting up specific registers with their required values and purposes. When this is then paired with single instruction conditional math statements, operations that would require many lines of code for x86 assembly can be optimized down only a fraction of that for ARM, meaning fewer clock cycles for overall execution.

Sample Programs

Factorial

This program loops an array containing five elements and multiplies them together essentially calculating the factorial of the largest number with the current array. It's very easy to see how ARMv7 is more flexible in the registers it is alllowed to use for this operation, and does so in fewer lines of code even without writing the value to the screen.

Intel x86

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
TITLE Factorial                                                     (main.asm)
;Author: Corwin Perren
;Date: 3/13/2014
;Description:
;       This program runs through an array and multiplies each value until all
;       array elements have been iterated through
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

INCLUDE Irvine32.inc

.data
        Array DWORD 1,2,3,4,5  ;Array with five values
.code

main PROC
        cld  ;;;;Set direction forward
        mov esi, OFFSET Array  ;;;;Point esi to offset for array
        mov ecx, LENGTHOF Array  ;;;;Set loop counter to length of array
        mov ebx, 1  ;;;;Set storage reg to 1 so initial multiplication works
MultLoop:
        lodsd  ;;;;Load current array value into eax
        mul ebx  ;;;;Multiply eax by ebx
        mov ebx, eax  ;;;;Store result in ebx
        loop MultLoop  ;;;;Loop until array empty
        mov eax, ebx  ;;;;Move final value into eax for printing
        call WriteDec  ;;;;Print to screen
```

```
        call Crlf ;;;;Print line
        exit ;;;;Exit program
main ENDP
END main
```

ARMv7

```
        AREA Factorial, CODE, READONLY
        ENTRY
Start
        ADR  r0, Array ;Point to address of array
        MOV  r1, #5 ;Set up a loop counter for array
        MOV  r2, #1 ;Set first val to one so math works
MultLoop
        LDR  r3,[r0] ;Put current array value into r3
        MUL r2,r3,r2 ;Multiply array val by stored product
        ADD  r0,r0,#4 ;Move forward by one array element
        SUBS  r1,r1,#1 ;Decrease loop counter
        BNE MultLoop ;repeat until all elements added

        ;Final value in r2 (0x78 == 120 == 5!)

Array DCD 1,2,3,4,5  ;Array to hold values
        END
```

Constrain using procedures and a stack argument

This program takes a value and passes it through a procedure which constrains it to a specific range. The parameters are passed using the stack to show the differences and the alternatve forms of procedures are also shown.

Intel x86

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
TITLE Constrain                                              (main.asm)
;Author: Corwin Perren
;Date: 3/13/2014
;Description:
;       This program runs passes an argument via stack to a constain function
;       based off of hobby R/C pulse times (1000-2000 microseconds)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

INCLUDE Irvine32.inc

.data
        ParamToConstrain DWORD 2500 ;;;;Value to constrain
        UpperVal DWORD 2000 ;;;;Upper limit for constraint
        LowerVal DWORD 1000     ;;;;Lower limit for constraint
.code

Constrain PROC
        mov ebp, esp ;;;;Move stack pointer into ebp
        mov eax, [ebp+4] ;;;;Move the param to constrain into eax
        cmp eax, [ebp+8] ;;;;Compare param to constrain to upper val
        jg IsGreater ;;;;If it's greater, jump to isgreater
```

```
        cmp eax, [ebp+12] ;;;;If it's not greater check if it's lower than lower val
        jg NoChange ;;;;If it's not lower, jump to end of procedure
        mov eax, [ebp+12] ;;;;If lower, move lower constrain value into eax
        mov [ebp+4], eax ;;;;Replace value to constrain with eax
        jmp NoChange ;;;;Just to end of procedure
IsGreater:
        mov eax, [ebp+8] ;;;;If greater, move upper val into eax
        mov [ebp+4], eax ;;;;Replace value to constrain with eax
NoChange:
        ret ;;;;Return from procedure
Constrain ENDP

main PROC
        push LowerVal ;;;;Push lower constraint to the stack
        push UpperVal ;;;;Push upper constraint to the stack
        push ParamToConstrain ;;;;Push value onto the stack
        call Constrain ;;;;Constrain this value
        pop eax ;;;;Pop constrained value off the stack
        add esp, 8 ;;;;Clean up stack
        call WriteDec ;;;;Write value to screen
        call Crlf ;;;;Prints a new line
        exit ;;;;Exits the program
main ENDP
END main
```

ARMv7

```
        AREA ConstrainProg, CODE, READWRITE
        ENTRY
Start
        ADR sp, Base ;Point to the base of the stack
        LDR r0, LowerVal ;Load the lower value into r0
        LDR r1, UpperVal ;Load the upper value in r1
        LDR r2, ParamToConstrain ;Load the value to constrain into r2
        STR r0,[sp,#-4] ;Load r0 onto the stack pointer
        STR r1,[sp,#-4] ;Load r1 onto the stack pointer
        STR r2,[sp,#-4] ;Load r2 onto the stack pointer
        BL Constrain ;Call constrain procedure
        LDR r2, [sp] ;Load constrained value back into r2
        ;r2 should now contain 2000
        ADD sp, sp, #8 ;Fix the stack pointer
Constrain
        STR LR, [sp, #-4] ;Store return address
        LDR r5, [sp, #12] ;Retrieve lower
        LDR r4, [sp, #8] ;Retrieve upper
        LDR r3, [sp, #4] ;Retrieve val to constrain
        CMP r3, r4 ;Check if higher than max
        LDRHI r3, [sp, #8] ;If so, load max
        CMP r3, r5 ;Check if lower than min
        LDRLO r3, [sp, #12] ;If so, load min
        STR r3, [sp, #4] ;Overwrite constrained val
        LDR PC, [sp], #4 ;Return from procedure
ParamToConstrain DCD 2200 ;Value to constrain
UpperVal DCD 2000  ;Value to constrain
LowerVal DCD 1000  ;Value to constrain
```

```
Base DCD 0xAAAAAAAA ;Used as a marker in memory
        END
```

References

A. Clements. *Computer Organization and Architecture*. Cenage Learning, 2014.

M. Levy. The history of the arm architecture: From inception to ipo. *ARM IQ*, 4(1), 2005.

A. Limited. *ARMv7-M Architecture Reference Manual*. ARM Limited, 2010.

L. Ryzhyk. The arm architecture. *Chicago University, IL, U.S.A.*, 2006.