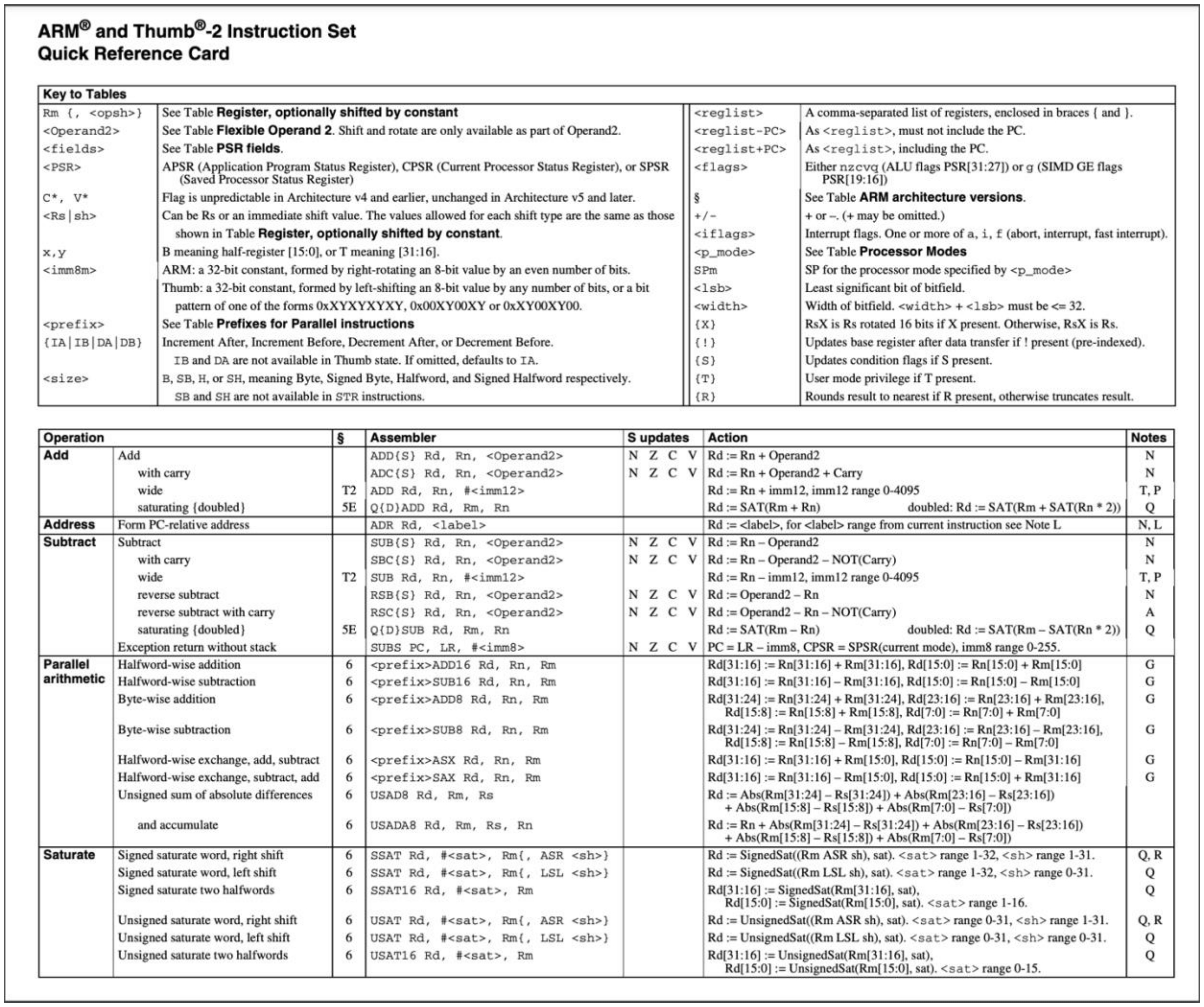
ARM Assembly Sort

Engineering 155 Lab I Report

September 21st, 2021

John Hearn



ARM® and Thumb®-2 Instruction Set
Quick Reference Card

Key to Tables

`Rm {, <opsh>}`	See Table **Register, optionally shifted by constant**	`<reglist>`	A comma-separated list of registers, enclosed in braces { and }.
`<Operand2>`	See Table **Flexible Operand 2**. Shift and rotate are only available as part of Operand2.	`<reglist-PC>`	As `<reglist>`, must not include the PC.
`<fields>`	See Table **PSR fields**.	`<reglist+PC>`	As `<reglist>`, including the PC.
`<PSR>`	APSR (Application Program Status Register), CPSR (Current Processor Status Register), or SPSR (Saved Processor Status Register)	`<flags>`	Either `nzcvq` (ALU flags PSR[31:27]) or `g` (SIMD GE flags PSR[19:16])
`C*, V*`	Flag is unpredictable in Architecture v4 and earlier, unchanged in Architecture v5 and later.	§	See Table **ARM architecture versions**.
`<Rs|sh>`	Can be Rs or an immediate shift value. The values allowed for each shift type are the same as those	`+/-`	+ or –. (+ may be omitted.)
	shown in Table **Register, optionally shifted by constant**.	`<iflags>`	Interrupt flags. One or more of `a`, `i`, `f` (abort, interrupt, fast interrupt).
`x,y`	B meaning half-register [15:0], or T meaning [31:16].	`<p_mode>`	See Table **Processor Modes**
`<imm8m>`	ARM: a 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits.	`SPm`	SP for the processor mode specified by `<p_mode>`
	Thumb: a 32-bit constant, formed by left-shifting an 8-bit value by any number of bits, or a bit	`<lsb>`	Least significant bit of bitfield.
	pattern of one of the forms 0xXYXYXYXY, 0x00XY00XY or 0xXY00XY00.	`<width>`	Width of bitfield. `<width>` + `<lsb>` must be <= 32.
`<prefix>`	See Table **Prefixes for Parallel instructions**	`{X}`	RsX is Rs rotated 16 bits if X present. Otherwise, RsX is Rs.
`{IA|IB|DA|DB}`	Increment After, Increment Before, Decrement After, or Decrement Before.	`{!}`	Updates base register after data transfer if ! present (pre-indexed).
	`IB` and `DA` are not available in Thumb state. If omitted, defaults to `IA`.	`{S}`	Updates condition flags if S present.
`<size>`	`B`, `SB`, `H`, or `SH`, meaning Byte, Signed Byte, Halfword, and Signed Halfword respectively.	`{T}`	User mode privilege if T present.
	`SB` and `SH` are not available in `STR` instructions.	`{R}`	Rounds result to nearest if R present, otherwise truncates result.

Operation		§	Assembler	S updates	Action	Notes
Add	Add		`ADD{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn + Operand2	N
	with carry		`ADC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn + Operand2 + Carry	N
	wide	T2	`ADD Rd, Rn, #<imm12>`		Rd := Rn + imm12, imm12 range 0-4095	T, P
	saturating {doubled}	5E	`Q{D}ADD Rd, Rm, Rn`		Rd := SAT(Rm + Rn) doubled: Rd := SAT(Rm + SAT(Rn * 2))	Q
Address	Form PC-relative address		`ADR Rd, <label>`		Rd := <label>, for <label> range from current instruction see Note L	N, L
Subtract	Subtract		`SUB{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn – Operand2	N
	with carry		`SBC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn – Operand2 – NOT(Carry)	N
	wide	T2	`SUB Rd, Rn, #<imm12>`		Rd := Rn – imm12, imm12 range 0-4095	T, P
	reverse subtract		`RSB{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Operand2 – Rn	N
	reverse subtract with carry		`RSC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Operand2 – Rn – NOT(Carry)	A
	saturating {doubled}	5E	`Q{D}SUB Rd, Rm, Rn`		Rd := SAT(Rm – Rn) doubled: Rd := SAT(Rm – SAT(Rn * 2))	Q
	Exception return without stack		`SUBS PC, LR, #<imm8>`	N Z C V	PC = LR – imm8, CPSR = SPSR(current mode), imm8 range 0-255.	
Parallel arithmetic	Halfword-wise addition	6	`<prefix>ADD16 Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] + Rm[31:16], Rd[15:0] := Rn[15:0] + Rm[15:0]	G
	Halfword-wise subtraction	6	`<prefix>SUB16 Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] – Rm[31:16], Rd[15:0] := Rn[15:0] – Rm[15:0]	G
	Byte-wise addition	6	`<prefix>ADD8 Rd, Rn, Rm`		Rd[31:24] := Rn[31:24] + Rm[31:24], Rd[23:16] := Rn[23:16] + Rm[23:16], Rd[15:8] := Rn[15:8] + Rm[15:8], Rd[7:0] := Rn[7:0] + Rm[7:0]	G
	Byte-wise subtraction	6	`<prefix>SUB8 Rd, Rn, Rm`		Rd[31:24] := Rn[31:24] – Rm[31:24], Rd[23:16] := Rn[23:16] – Rm[23:16], Rd[15:8] := Rn[15:8] – Rm[15:8], Rd[7:0] := Rn[7:0] – Rm[7:0]	G
	Halfword-wise exchange, add, subtract	6	`<prefix>ASX Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] + Rm[15:0], Rd[15:0] := Rn[15:0] – Rm[31:16]	G
	Halfword-wise exchange, subtract, add	6	`<prefix>SAX Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] – Rm[15:0], Rd[15:0] := Rn[15:0] + Rm[31:16]	G
	Unsigned sum of absolute differences	6	`USAD8 Rd, Rm, Rs`		Rd := Abs(Rm[31:24] – Rs[31:24]) + Abs(Rm[23:16] – Rs[23:16]) + Abs(Rm[15:8] – Rs[15:8]) + Abs(Rm[7:0] – Rs[7:0])	
	and accumulate	6	`USADA8 Rd, Rm, Rs, Rn`		Rd := Rn + Abs(Rm[31:24] – Rs[31:24]) + Abs(Rm[23:16] – Rs[23:16]) + Abs(Rm[15:8] – Rs[15:8]) + Abs(Rm[7:0] – Rs[7:0])	
Saturate	Signed saturate word, right shift	6	`SSAT Rd, #<sat>, Rm{, ASR <sh>}`		Rd := SignedSat((Rm ASR sh), sat). `<sat>` range 1-32, `<sh>` range 1-31.	Q, R
	Signed saturate word, left shift	6	`SSAT Rd, #<sat>, Rm{, LSL <sh>}`		Rd := SignedSat((Rm LSL sh), sat). `<sat>` range 1-32, `<sh>` range 0-31.	Q
	Signed saturate two halfwords	6	`SSAT16 Rd, #<sat>, Rm`		Rd[31:16] := SignedSat(Rm[31:16], sat), Rd[15:0] := SignedSat(Rm[15:0], sat). `<sat>` range 1-16.	Q
	Unsigned saturate word, right shift	6	`USAT Rd, #<sat>, Rm{, ASR <sh>}`		Rd := UnsignedSat((Rm ASR sh), sat). `<sat>` range 0-31, `<sh>` range 1-31.	Q, R
	Unsigned saturate word, left shift	6	`USAT Rd, #<sat>, Rm{, LSL <sh>}`		Rd := UnsignedSat((Rm LSL sh), sat). `<sat>` range 0-31, `<sh>` range 0-31.	Q
	Unsigned saturate two halfwords	6	`USAT16 Rd, #<sat>, Rm`		Rd[31:16] := UnsignedSat(Rm[31:16], sat), Rd[15:0] := UnsignedSat(Rm[15:0], sat). `<sat>` range 0-15.	Q

Introduction:

The goal of this lab was to write a simple assembly program which sorts an array of signed bytes in order of smallest to greatest. We were to debug using the PlatformIO debugging tool, monitoring relevant locations in memory.

Design Methodology:

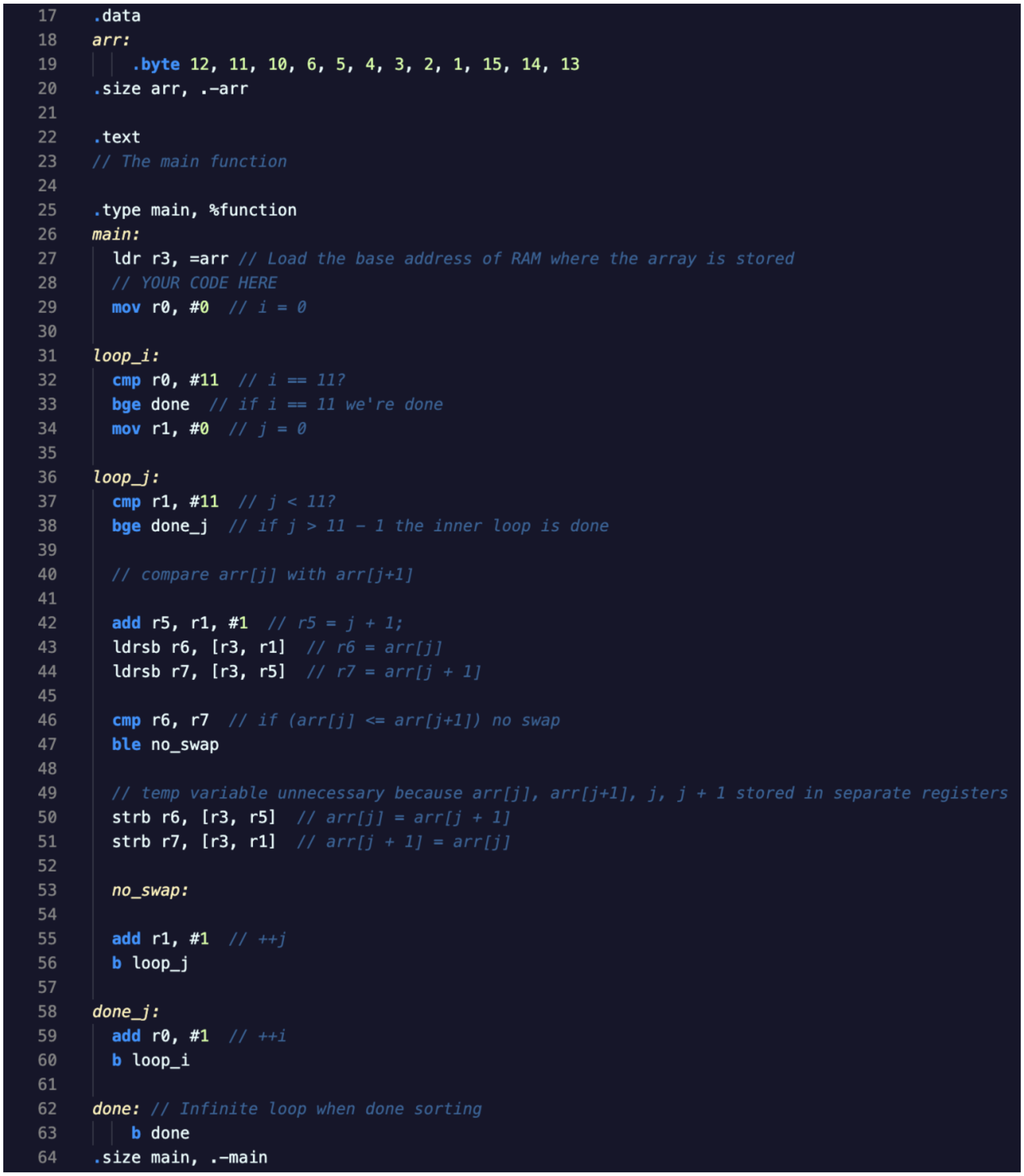
This lab only required a software component. In designing the assembly code, it was helpful to first write out the desired program in C, for which the bubble sort algorithm was chosen. Five subsections were created in writing the program: main, loop_i, loop_j, and done_j, each corresponding to an important part in the C code.

For example, main initialized the r3 register as the array being sorted and initialized the outer looping index r0 to 0, as the function would normally.

The loop_i block compares r0 with arr.size() - 1, and branches to the “done” block if r0 is greater than or equal to the previous mentioned value. Otherwise, we assign a register r1 to 0 to emulate setting the inner loop index being set to 0.

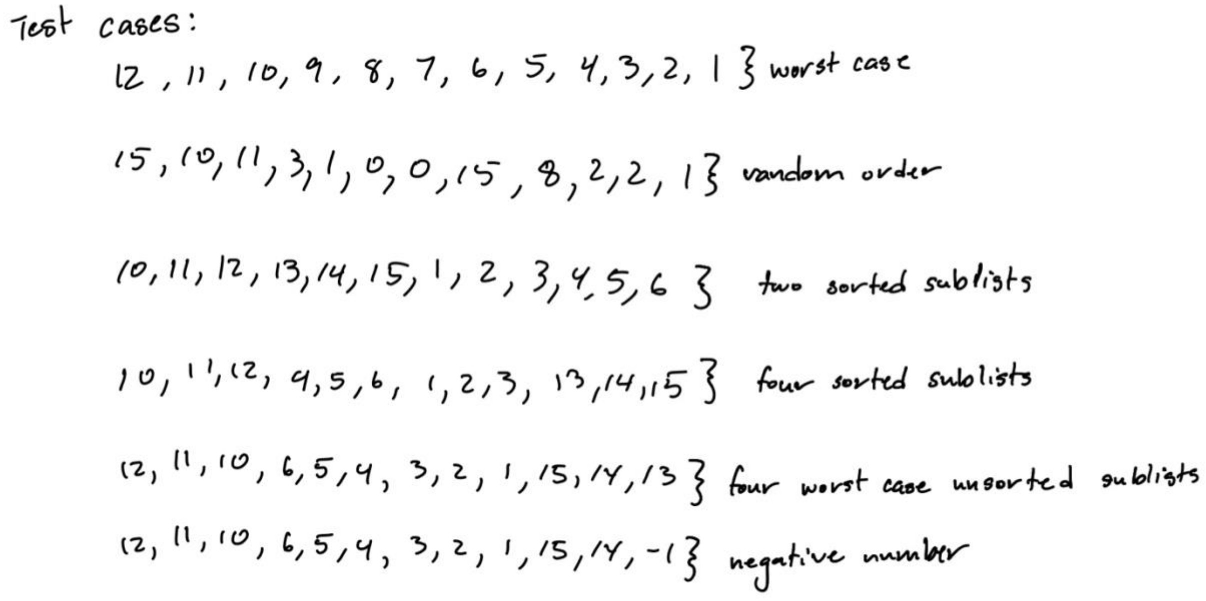
Within loop_j is where most of the heavy lifting is done. First we compare the value of r1 with arr.size() – 1 as before to let us know when we’re done with the loop. If r1 >= arr.size() – 1, we branch to the done_j block. Otherwise, we check to see if arr[j] and arr[j + 1] need to be swapped. To do this, we load the value at each index into two registers, compare the values, and store them in their appropriate places.

Technical Documentation:



```
.data
arr:
        .byte 12, 11, 10, 6, 5, 4, 3, 2, 1, 15, 14, 13
.size arr, .-arr

.text
// The main function

.type main, %function
main:
  ldr r3, =arr // Load the base address of RAM where the array is stored
  // YOUR CODE HERE
  mov r0, #0  // i = 0

loop_i:
  cmp r0, #11  // i == 11?
  bge done  // if i == 11 we're done
  mov r1, #0  // j = 0

loop_j:
  cmp r1, #11  // j < 11?
  bge done_j  // if j > 11 - 1 the inner loop is done

  // compare arr[j] with arr[j+1]

  add r5, r1, #1  // r5 = j + 1;
  ldrsb r6, [r3, r1]  // r6 = arr[j]
  ldrsb r7, [r3, r5]  // r7 = arr[j + 1]

  cmp r6, r7  // if (arr[j] <= arr[j+1]) no swap
  ble no_swap

  // temp variable unnecessary because arr[j], arr[j+1], j, j + 1 stored in separate registers
  strb r6, [r3, r5]  // arr[j] = arr[j + 1]
  strb r7, [r3, r1]  // arr[j + 1] = arr[j]

  no_swap:

  add r1, #1  // ++j
  b loop_j

done_j:
  add r0, #1  // ++i
  b loop_i

done: // Infinite loop when done sorting
    b done
.size main, .-main
```

Fig. 1: Assembly code



Test cases:

12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 } worst case

15, 10, 11, 3, 1, 0, 0, 15, 8, 2, 2, 1 } random order

10, 11, 12, 13, 14, 15, 1, 2, 3, 4, 5, 6 } two sorted sublists

10, 11, 12, 4, 5, 6, 1, 2, 3, 13, 14, 15 } four sorted sublists

12, 11, 10, 6, 5, 4, 3, 2, 1, 15, 14, 13 } four worst case unsorted sublists

12, 11, 10, 6, 5, 4, 3, 2, 1, 15, 14, -1 } negative number

Fig. 2: Test cases

Results and Discussion:

All of my test cases passed and the sorting algorithm works as expected.

If I were to redo this lab, I would familiarize myself first with the difference between loading and storing bytes versus words.

Conclusion:

I was successful in writing a sorting algorithm in assembly code and using the debugging interface within PlatformIO.

This lab took about 5 hours to complete.