CEG 3136 Summary Sheet

1 Data Representation

One **Byte** is defined as 8 bits. In 32 bit architecture, one **Word** is 32 bits, and therefore a half word is 16 bits, and a double word is 64 bits.

We have unsigned integers and signed integers. Unsigned is simple, signed can be represented in either signed magnitude, two complement, or ones complement.

Ex. Show -5 in 4 bits in all 3 forms

Positive 5: 0101

Signed Magnitude: 1101
Twos Complement: 1011
Ones Complement: 1010

In an adder/subtractor, we generate a carry flag (C) (carry out of most significant bit [MSB]) and an overflow flag (V). If we are doing unsigned, the carry flag signifies something is wrong (C=1 for addition, C=0 for subtraction). For signed, we ignore C, and use V. If V=1, something went wrong.

Overflow is the XOR of the carry out of the MSB and the carry in to the MSB.

We typically use two complement since it makes addition and subtraction able to use the same logic.

1.1 Strings

Strings are represented using ASCII codes. Each string is compared using its ASCII value.

| Letter | A | В | С | X | Y | Z | a | b | c | X | У | Z |
|------------|----|----|----|-------|----|----|-------|----|----|-------|----|----|
| ASCII Code | 41 | 42 | 43 | 58 | 59 | 5A | 61 | 62 | 63 | 78 | 79 | 7A |

1.2 Fixed Point

The Q Notation represents the type of fixed point using UQm.n.

| U | Unsigned. Remove if we are doing signed |
|---|---|
| Q | Means Q notation |
| m | Number of integer bits |
| n | number of fractional bits |

Ex. Approximate $-\pi$ using Q3.12

This means we have the sign bit, then 3 integer bits and 12 fractional bits.

We can represent pi as:

0011.001001000100

So after taking the two complement $-\pi$ becomes:

1100.110110111100

If it was UQ4.12 we could only represent π not $-\pi$ but it would be:

0011.001001000100

To add fixed point numbers, it is very simple. We just treat it as if the radix is non existent.

1.3 Floating Point

Floating point is similar to scientific notation in decimal. We first need to normalize it. So we make it in the form of $1.xxxxxxxx \times 2^{exp}$. We then hide the 1 as it is implied.

Using single precision, we have 1 bit for the sign, 8 for exponent, and 23 for the fractional.

To represent negative and positive exponents, we add 127 to the exponent.

Ex. Express $(36.5625)_{10}$ as a 32 bit floating point number using the IEEE standard.

The value in binary is 100100.1001 and normalized is 1.001001001×2^5 .

1. Sign: 0

2. Exponent: 5+127 = 132 = 100001003. Mantissa: 00100100100000000000000

We reserve some values for special cases:

| | Sign | Exponent | Fraction | Meaning |
|---|-------|-----------|--------------------|--------------------|
| 1 | . / 0 | 0000 0000 | 0000 0000 | 0 |
| 1 | . / 0 | 1111 1111 | 0000 0000 | + / - infinity |
| | X | 1111 1111 | any non zero value | NaN (Not a Number) |

Floating point has big dynamic range, but is less precise and more complicated than fixed point.

2 ARM Instructions

ARM has 3 main processor families;

| CORTEX-A | High performance Application processors |
|----------|---|
| CORTEX-R | Reliable Real time processors for mission critical purposes |
| CORTEX-M | Low cost, Low power Microcontroller |

ARM has a few different instruction sets. The CORTEX-M series supports the T32 instruction set which includes both space saving 16 bit instructions, and high performance 32 bit instructions.

Other architectures such as CORTEX-A support T32 and A32 instructions with some supporting A64 as well.

ARM is a RISC architecture, so it cannot directly access memory through instructions. It must first **load** from memory into registers, then **modify**, and finally **store** back into memory.

We have 16 **core registers** and some **special purpose registers** as well. Since we are 32 bit, the registers are all 32 bits wide.

| D : 4 | 0 1 0 1 | D |
|-----------|--------------------|---|
| Register | General or Special | Purpose |
| R0 | С | General Purpose |
| R1 | С | General Purpose |
| R2 | С | General Purpose |
| R3 | С | General Purpose |
| R4 | С | General Purpose |
| R5 | С | General Purpose |
| R6 | С | General Purpose |
| R7 | С | General Purpose |
| R8 | С | General Purpose |
| R9 | С | General Purpose |
| R10 | С | General Purpose |
| R11 | С | General Purpose |
| R12 | С | Intra Procedure Call Register (IR) |
| R13 | С | Stack Pointer (SP) - Often there are two: |
| | | MSP (Main) and PSP (Process) |
| R14 | С | Link Register (LR) |
| R15 | С | Program Counter (PC) |
| xPSR | S | Program Status Register |
| BASEPRI | S | Interrupt Priorities |
| PRIMASK | S | Enabling and Disabling Interrupts |
| FAULTMASK | S | Fault Handling |
| CONTROL | S | |

Often we map some hardware device to a memory to make it easier to work with. For example, we may have it set up so bit 7 of R0 is 1 if an LED is on, or 0 if the LED is off.

Assembly has 4 main classes of instructions:

- Arithmetic and Logic
- Data Movement
- Compare and Branching
- Miscellaneous

Each instruction has 4 parts:

| General From | label | mnemonic | operand(s) | comments |
|--------------|-------|----------|------------|-------------------------------------|
| Ex | | BX | LR | ; branch to LR |
| Ex | LOOP | CMP | R1, R2 | ; start of loop, compares R1 and R2 |
| Ex | | STR | R1, R2 | |
| Ex | | ADD | R3, R5, R8 | |

We also have assembly directives which are just information for the assembler such as ALIGN, EXPORT, and ENDP.

| Directive | Meaning |
|-------------|---|
| AREA | Make a new block of data or code |
| ENTRY | Declare an entry point where the program execution starts |
| ALIGN | Align data or code to a particular memory boundary |
| DCB | Allocate one or more bytes (8 bits) of data |
| DCW | Allocate one or more half-words (16 bits) of data |
| DCD | Allocate one or more words (32 bits) of data |
| SPACE | Allocate a zeroed block of memory with a particular size |
| FILL | Allocate a block of memory and fill with a given value |
| EQU | Give a symbol name to a numeric constant |
| RN | Give a symbol name to a register |
| EXPORT | Declare a symbol and make it referable by other source files |
| IMPORT | Provide a symbol defined outside the current source file |
| INCLUDE/GET | Include a separate source file within the current source file |
| PROC | Declare the start of a procedure |
| ENDP | Designate the end of a procedure |
| END | Designate the end of a source file |

2.1 Arithmetic and Logic

Here are just a few of the arithmetic instructions for T32.

| Mnemonic | Syntax | Meaning | Operation |
|----------------------|----------------|---|--------------------------------------|
| ADD | {Rd,} Rn, Op2 | Add | $Rd \leftarrow Rn + Op2$ |
| ADC | {Rd,} Rn, Op2 | Add w/ carry | $Rd \leftarrow Rn + Op2 + Carry$ |
| SUB | {Rd,} Rn, Op2 | Subtract | $Rd \leftarrow Rn - Op2$ |
| SBC | {Rd,} Rn, Op2 | Subtract w/ carry | $Rd \leftarrow Rn - Op2 + Carry - 1$ |
| RSB | {Rd,} Rn, Op2 | Reverse subtract | $Rd \leftarrow Op2 - Rn$ |
| \mathbf{MUL} | {Rd,} Rn, Rm | Multiply | $Rd \leftarrow (Rn \times Rm)[31:0]$ |
| MLA | Rd, Rn, Rm, Ra | Multiply and accumulate | |
| | | $Rd \leftarrow (Ra + (Rn \times Rm))[31:0]$ | |
| MLS | Rd, Rn, Rm, Ra | Multiply and subtract | |
| | | $Rd \leftarrow (Ra - (Rn \times Rm))[31:0]$ | |
| SDIV | {Rd,} Rn, Rm | Signed divide | $Rd \leftarrow Rn \div Rm$ |
| UDIV | {Rd,} Rn, Rm | Unsigned divide | $Rd \leftarrow Rn \div Rm$ |

There are also lots of logic ones such as AND, ORR, EOR (XOR), ORN (NOR) and so on.

There are many other instructions, the ARM T32 instruction set has a lot.

2.1.1 NZCV Flags

These flags are stored in bits 28 to 31 of the PSR.

| Flag | Meaning |
|------|---|
| N | Negative - Result is Negative |
| Z | Zero - Result is Zero |
| С | Carry - Unsigned Arithmetic out of range |
| V | Overflow - Signed Arithmetic out of range |

To update these flags, we add an S to the end of the instruction.

Ex.

Does not update NZCV flags: ADD, SUB, MUL, etc Does update NZCV flags: ADDS, SUBS, MULS, etc Always updates NZCV flags: CMP, CMN, TST, TEQ

2.1.2 Saturation

Saturation is a logical operation that deals with the case where overflow occurs.

Normally, it will wrap back around to the lowest value. However, sometimes we want to cap the highest value.

| With Saturation (4 bits) | 7+1=-8 |
|-----------------------------|--------|
| Without Saturation (4 bits) | 7+1=7 |

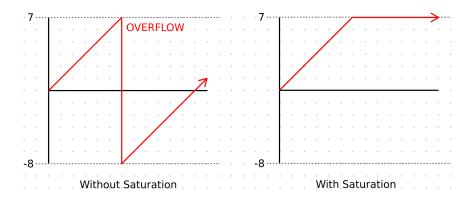


Figure 1: Saturation

2.1.3 Other Instructions

| Instruction | Description | Similar Instructions |
|---------------|------------------------------|--|
| RBIT Rd, Rn | Reverses bit order in word | REV (byte order), REV16 (For half |
| | | words), REVSH (Sign Extend) |
| SXTB {Rd,} Rm | Sign Extension (Byte) | SXTH (Half word), UXTB/UXTH |
| | | (Zero extend) |
| MOV Rd, Rx | Move from Rx to Rd | MVN (MV and NOT), MRS (From |
| | | special reg), MSR (From gen to spe- |
| | | cial) |
| LSL Rd, Rn, # | Move Rn to Rd and left shift | LSR (right logical), ASR (Right arith- |
| | | metic), ROR (rotate right) |

2.2 Memory

Memory is byte addressable, but we typically only start a 32 bit word at a multiple of 4, a 16 bit half word at a multiple of 2, and a byte at any point.

| LDRxx R0, [R1] | · · |
|----------------|--|
| STRxx R0, [R1] | Store contents of R0 into memory at R1 |

If we are storing something smaller than the memory width (byte, or halfword) we need to differentiate between signed (add S) [LDRSB, LDRSH] and unsigned (do not add S) [LDRB, LDRH].

When loading and storing, we can also address bits after the location we specify. This is useful for arrays. We have a few modes:

| Register Offset | LDR r0, [r1, r2] | Target: r1+r2 |
|------------------|-------------------|--|
| Immidiate Offset | LDR r0, [r1, #8] | Target: $r1 + 8$ |
| Pre-Index | LDR r0, [r1, #4]! | Target: r1+4, update r1 to r1+4 after read |
| PostIndex | LDR r0, [r1], #4 | Target: r1, increase r1 by 4 after read |

2.3 Endianess

Endianess means within a 32 bit word (or any multi byte data structure) do we start the LSB at the low address (little endian) or high address (big endian)?

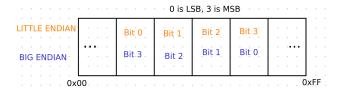


Figure 2: Endianess

2.4 Control Flow Instructions

We have 4 variations of the branch command. These will branch to either a label, or an address.

| Instruction | Operands | Description |
|-------------|----------|-------------------------------|
| В | label | Branch |
| BL | label | Branch with link |
| BLX | Rm | Branch and Exchange with link |
| BX | Rm | Branch and Exchange |

We have condition codes. These are appended to almost any instruction and they will only execute the instruction if the condition is true. It uses the status flags NZCV. These are typically performed after a CMP operation.

| Suffix | Description | Flags tested |
|--------|---------------------------|-----------------------|
| EQ | Equal | Z=1 |
| NE | Not Equal | Z=0 |
| CS/HS | Unsigned Higher or Same | C=1 |
| CC/LO | Unsigned Lower | C = 0 |
| MI | Minus (Negative) | N=1 |
| PL | Plus (Positive or Zero) | N=0 |
| VS | Overflow Set | V = 1 |
| VC | Overflow Cleared | V = 0 |
| HI | Unsigned Higher | C = 1 & Z = 0 |
| LS | Unsigned Lower or Same | $C=0 \mid Z=1$ |
| GE | Signed Greater or Equal | N = V |
| LT | Signed Less Than | $N \neq V$ |
| GT | Signed Greater Than | Z = 0 & N = V |
| LE | Signed Less than or Equal | $Z = 1 \mid N \neq V$ |
| AL | Always | None |

```
Ex. Branch to F00 if r0 is less than 0.

CMP r0, #0 ; compare r0 with 0
BLE FOO ; branch to FOO if LE (Z=1)

This is similar to some c code doing:

if (a < 0) { //assuming a is in r0
foo(); //or something else, whatever code is located at FOO
}
```

ARM assembly lets us use the IT (If Then) syntax as well.

```
Ex.

ITTE NE ; Two commands will follow with NE ANDNE r0, r0, r1 ; Then one command with the opposite ANDNE r2, r2, #1 ; of NE which is EQ MOVEQ r2, r3 ;

ITT EQ ; The IT can be ommitted from code MOVEQ ... ; and the assembler will add it.

ADDEQ ...
```

3 Subroutines

The link register LR contains the return address of the subroutine. This is copied back to the PC when the subroutine is finished.

In ARM, we store any parameters in registers R0 through R3. Any additional parameters need to be put on the stack. Also, if the parameters are larger than 32 bits, they can take up more than one register (a 128 bit parameter would take up R0, R1, R2, R3).

It returns the return value in R0.

Registers R0 through R3 can be freely changed by the subroutine, as well as R12 and R14 (LR). In other words, the calling function cannot expect them to keep the same data when the subroutine returns. The opposite is true with registers R4 to R11 where they must be preserved. If the subroutine changes anything in those registers, it must return them to the previous value before returning.

We use BL or BLX to call a subroutine.

3.1 Stack

The ARM stack uses a full descending stack. This means that the stack pointer points to the top piece of data on the stack. The stack also grows down to memory address 0 as items are pushed to it.

We have the instructions PUSH cregulist and POP cregulist.

When we push or pop multiple registers, the highest number register is pushed first, and popped last.

```
Ex. PUSH_{\square}\{r6, \_r8, \_r7\}
```

This instruction will first push r8, followed by r7 and then finally r6.

It would be equivalent to $PUSH_{\square}\{r6, _r7, _r8\}$ and to $PUSH_{\square}\{r8, _r6, _r7\}$ and so on.

```
Ex. POP_{\sqcup}\{r6, {\sqcup}r8, {\sqcup}r7\}
```

This instruction will first pop r6, followed by r7 and then finally r8.

It would be equivalent to $POP_{\sqcup}\{r6, {}_{\sqcup}r7, {}_{\sqcup}r8\}$ and to $POP_{\sqcup}\{r8, {}_{\sqcup}r6, {}_{\sqcup}r7\}$ and so on.

```
Ex.

... ; main program

BL foo

...

foo PROC

PUSH {r4} ; we are using r4 which must be preserved

...

MOV r4, #1 ; this changes r4, good thing we saved it

...

POP {r4} ; this restores r4 for the caller function
 ; goes back to caller

ENDP
```

We also need to preserve the LR on the stack if we call a subroutine from inside a subroutine since it could be overridden. Then we would have no way to return to the main program.

Often we have two stack pointers, the MSP (main) and PSP (process). This is toggled by a bit in the CONTROL register.

4 C and Assembly

When we have C code, it goes through a lot of steps to get into ARM assembly to be loaded onto the microcontroller.

```
Proprocessor 	o Compiler 	o Assembler 	o Linker 	o Loader 	o MCU(thru\ programmer) \\ 	o Debugger
```

Typically, C will set up data so every word starts at an even multiple of 4 bits address. This is for efficiency. Same idea with half words, but every 2 bits. C does this by padding extra

space. We can use the __packed keyword in c to not pad the extra space. But this can cause weird behavior.

If we want to mix C and assembly, we can do this. In assembly, we call a C function using the import keyword, and export a function to C using the export keyword. Similarly, in C we can import something from assembly using the extern keyword. This can work with data as well as functions.

4.1 Volatile Datatypes

The volatile keyword means each time we use a variable, we need to import it from memory into a register. This is useful when an external event may change memory at any point. In this case, we need to ensure that we don't use an old version of the variable.

4.2 Interrupts

An interrupt is a signal that occurs that tells the controller that it needs to stop whatever it is currently doing, save the state using the stack, and then move on to the interrupt service routine (ISR). After it then restores the stack, and goes back to the user program.

It needs to save xPSR, PC, LR, R12, R3, R2, R1, and R0. Therefore it can use any of those registers to store data. Any other registers that are used must be returned to their original state.

5 Appendix

| Registers | Arithmetic | Bitwise |
|-------------|---|--|
| RO S | ADD[S] Rd,Rn,Op2 Rd \leftarrow Rn + Op2 [N Z C V] | MVN[S] Ra,Rb Ra ← ~Rb [N Z C] |
| R1 S | ADC[S] Rd,Rn,Op2 Rd \leftarrow Rn + Op2 + C [N Z C V] | AND[S] Rd,Rn,Op2 Rd ← Rn & Op2 [N Z C] |
| R2 S | SUB[S] Rd,Rn,Op2 Rd \leftarrow Rn - Op2 [N Z C V] | ORR[S] Rd,Rn,Op2 Rd ← Rn Op2 [N Z C] |
| R3 S | SBC[S] Rd,Rn,Op2 Rd \leftarrow Rn - Op2 - C [N Z C V] | EOR[S] Rd,Rn,Op2 Rd ← Rn ^ Op2 [N Z C] |
| R4 P | RSB[S] Rd,Rn,Op2 Rd \leftarrow Op2 - Rn [N Z C V] | BIC[S] Rd,Rn,Op2 Rd ← Rn & ~Op2 [N Z C] |
| R5 P | $MUL[S] Rd,Rn,Op2 Rd \leftarrow Rn \times Op2 [N Z]$ | Compare & Test |
| R6 P | MLA[S] Rd,Rn,Rm,Ra Rd ← Ra + Rn × Rm [N Z] | CMP Rn,Op2 Rn - Op2 N Z C V |
| R7 P | MLS[S] Rd,Rn,Rm,Ra Rd ← Ra – Rn × Rm [N Z] | CMN Rn,Op2 Rn + Op2 N Z C V |
| R8 P | {S,U}MULL RdL,RdH,Rn,Rm Rd ← Rn × Rm | TST Rn,Op2 Rn & Op2 N Z C |
| R9 SB/TR | $\{S,U\}MLAL RdL,RdH,Rn,Rm Rd \leftarrow Rd + Rn \times Rm$ | TEQ Rn,Op2 Rn ^ Op2 N Z C |
| R10 P | $\{S,U\}MLSL$ RdL,RdH,Rn,Rm Rd \leftarrow Rd $-$ Rn \times Rm | Branch |
| R11 FP P | {S,U}DIV Rd,Rn,Rm Rd ← Rn ÷ Rm | B label PC ← label |
| R12 IP S | {S,U}XT{B,H} Ra,Rb Ra ← Extend(Rb) | BX Rm PC ← Rm |
| R13 SP P | $ \{S,U\}SAT \qquad Rd,\#n,Rm,Sft \qquad Rd \leftarrow Sat(Sft(Rm)) \qquad Q $ | BL label LR←PC+4; PC label |
| R14 LR S | CLZ[S] Rd,Rn Rd ← CountLeadingZeros(Rn) | BLX Rm LR \leftarrow PC + 4 ; PC \leftarrow Rm |
| R15 PC S | Shift, Rotate, Swap | TBB Rn,Rm PC ← PC+4 + [Rn+Rm] |
| xPSR S | LSL[S] Rd,Rm,Rs Rd \leftarrow Rm $<<$ Rs [N Z C] | TBH Rn,Rm,LSL #1 PC ← PC+4 + [Rn+2*Rm] |
| CONTROL | LSR[S] Rd,Rm,Rs Rd \leftarrow Rm $>>$ Rs [N Z C] | CBZ Rn,label Rn == 0: PC ← label |
| Op2 Sft* | | CBNZ Rn,label Rn != 0: PC ← label |
| #const | ROR[S] Rd,Rm,Rs Rotate right [N Z C] | Data Transfer |
| Rm | RRX[S] Rd,Rm,Rs Rd \leftarrow Rm $>> 1$ [N Z C] | ADR Rd,label Rd ← label |
| ASR #n * | REV Ra,Rb Ra $B_0B_1B_2B_3 \leftarrow Rb B_3B_2B_1B_0$ | MOV[S] Rd,Op2 Rd \leftarrow Op2 [N Z C] |
| LSL #n * | REV16 Ra,Rb Ra $B_2B_3B_0B_1 \leftarrow Rb B_3B_2B_1B_0$ | MOV[S] Rd,#imm16 Rd ← imm16 [N Z C] |
| LSR #n * | REVSH Ra,Rb Ra $B_1B_0B_3B_2 \leftarrow Rb B_3B_2B_1B_0$ | MOVT Rd,#imm16 Rd ← Rd imm16 << 16 |
| ROR #n | Bitfields | {LD,ST}R[H,B,SH,SB] Rt,Adr |
| RRX | BFC Rd,#b,#w Rd \leftarrow Rd & \sim ((2 $^{\text{w}}$ -1) $<<$ b) | {LD,ST}RD Rt,Rt2,Adr |
| Adr | BFI Rd,Rn,#b,#w Rd \leftarrow (Rd & \sim ((2 w -1) $<<$ b)) (Rn $<<$ b) | LDM[IA FD,IB ED,DA FA,DB EA] Rn[!],{Regs} |
| [Rn] | $\{S,U\}BFX Rd,Rn,\#b,\#w Rd \leftarrow (Rn >> b) \& (2^w-1)$ | STM[DB FD,DA EA,IB FA,IA EA] Rn[!],{Regs} |
| [Rn, Rm] | Condition Codes $S \ \underline{\cup}$ Miscellaneous | {PUSH,POP} {Regs} ⇔ {ST,LD}MFD SP!,{Regs} |
| [Rn,#offs] | EQ NE MI PL AL IT[T,E][T,E] cond | MRS Rd,SpecR Rd ← SpecR |
| [Rn,#offs]! | GE LT LE GT VS VC WFI Wait for interrupt | MSR SpecR,Rm SpecR ← Rm N Z C V |
| [Rn],#offs | HS LO LS HI CS CC NOP No operation | ARM T32 ISA Reference 2025.10.16 |