Week 8

Disassembly Window (of ARM Development) → takes your code line-by-line, converts it into assembly, turns it into machine language code, and then converts it back into assembly

Execution of ARM program: Click on Start/Stop Debug Session

Assembly Language Directives:

- AREA → to name a region of code or data
- ENTRY → execution starting point
- END → physical end of program
- name EQU value → equate a name to a value (will not make any memory allocations)
- {label} DCD v.expr, v.expr → set up on or more 32-bit constants in memory (v.expr must be a constant expression)(must start at a memory location that is a multiple of 4)
- {label} DCW v.expr, v.expr → set up one or more 16-bit constant memory (a half-word) (must start at a memory location that is even)
- {label} DCB v.expr, v.expr → set up one or more 8-bit constant memory (can start anywhere)
- {label} SPACE size expr → reserves a zeroed block of memory (can start anywhere)
- ALIGN → ensures that next data items is correctly aligned on 32-bit boundaries ie. to start at a multiple of 4 address-location
    - Explicit aligns location counter

Location Counter vs Program Counter (EXAM QUESTION!!)

- Location-Counter = variable inside the assembler to keep track of memory-locations; only available during assembling a program
- Program-Counter = register to keep track of the next instruction to be executed; only available at run time

Double Quotations used to define a string.
Strings must be used with DCB.

Pseudo Instructions

- Doesn't have a direct machine language equivalent
- ADR r0, label→ address to register
    - Loads the 32-bit address and stores it in a register
- Uses program counter relative addressing to implement it
- Register indirect relative addressing ([r0]) allows us to specify the location of an operand with respect to a register value

ARM's Data-Processing Instructions

Addition: ADD (ADDS), ADC (ADCS)

- Adds two 32-bit values

- ADC = add with carry, adds two 32-bit bit values together with the carry bit

Subtraction: SUB (SUBS), RSB (RSBS)

- RSB = reverse subtraction; when you have RSB r1, r2, r3, you are subtracting r2 from r3 and storing it in r1, as opposed to SUB r1, r2, r3, where you are subtracting r3 from r2 and storing it in r1
- This can be used when you're using constants instead of registers; you can only use a constant in the last operand spot

Negation: NEG (NEGS)

- Negation is to subtract a number from 0
- NEG r1, r2; negates second operand and stores it in first operand
- RSB is used to implement NEG
    - NEG r1, r2 = RSB r1, r2, #0
- To store the result of the negation in the same register, we can write NEG r2, r2
    - This is equivalent to RSB r2, r2, #0 or RSB r2, #0

Move: MOV (MOVS)

- Copes the value of the second operand into the first operand
- MOV r0, r1 → copies content of r1 into r0
- MVN = move not (logical NOT), meaning the processor flips each zero to one and each one to zeroN r0, r1

Multiply: MUL, MLA

- MUL Rd, Rm, Rs → multiples second operand to third operand and stores result in first operand
- Can't have any constants in MUL; only registers
- Stores the lower-order 32 bits of the 640-bit product; the result is truncated
- In MUL, the same register can't be used to specify both the destination Rd and the operand Rm
- MLA = multiply and accumulate
    - Same properties to MUL except for the operation syntax
    - Performs a multiplication and adds the product to a running total
    - MLA Rd, Rm, Rs, Rn; [Rd] = [Rm] x [Rs] + [Rn]

Bitwise Logical Operations

- AND
- ORR
- EOR (exclusive or)

Keil Assembler Prefixes

- 2_ before a constant means binary (after the #)
- 8_ means octal
- 0x or & means hexadecimal
- If you want to write a decimal number, don't precede constant with anything

BIC (bit clear instruction)

- Uses AND with the first operand and the complement of the second operand

Explicit Comparisons

- CMP r1, r2 → instruction will evaluate r1 - r2 without storing the result but set the flags
- TEQ (test equivalent instruction)
    - Determines whether two operands are equivalent or not; result is discarded; updates flags except overflow and carry flag
- TST (test instruction)
    - Compares two operands by ANDing them together and updating flags except for overflow and carry flag
- CMN (compare negative instruction)
    - 2s complement the second operand and subtract from first operand
    - However, this is equivalent to adding r1 and r2
    - Set flags, evaluate

Shift Operations (move bits one or more places left or right)

ARM supports both static and dynamic shifts.
Static = number of shift spaces determined when the code is written
Dynamic = number of shift spaces determined when the code is executed ie. at run time.

- Logical Shifts
    - Insert a 0 in the vacated position
    - Bit shifted out is copied to the carry bit
    - ARM implemented both LSL (logical shift left) and LSR (logical shift right)
- Arithmetic Shifts
    - Replicate the sign-bit in the vacant position
    - Bit shifted out is copied to the carry bit
    - ARM only implemented ASR (arithmetic shift right)
    - ASL (arithmetic shift left) is the same as LSL
- Circular Shifts
    - Look at it as circle; whatever bits you lose at one end are added to the other end
    - No bits lost during rotation
    - Bit shifted out is copied into the carry bit
    - ARM only implemented ROR (rotate right)
    - ROL (rotate left) is identical to a 32 - n ROR
- Circular Shifts through Carry
    - Include the carry bit in the shift path
    - Exactly like circular shifts but the carry bit is within the circle (carry is shifted into register)
    - ARM only implemented RRX (rotate right through carry)

Note: other shift operations that are not implemented by ARM have to be synthesized by the programmer

Single Quotation (‘A’) → used for a single character only; can be used with DCB, DCW, or DCD
Double Quotations (“ABC”) → used for a string; MUST be used with DCB

Equal Signs

- Can have different meaning when placed in different locations
- At opcode column, = means DCB
    - XYZ = 0x41; = is being used as DCB
- In LDR, the equals sign before the constant means the LDR is a pseudo instruction

Ampersand Sign “&”

- Can have different meaning when placed in different locations
- At opcode column, means DCD
    - AAA & 0x123456
- When coming before a constant, the & means a hexadecimal value
    - & = 0x in this case

Percent Sign “%”

- At opcode column, means SPACE
    - BBB % 0x40

Hash “#”

- The # in ARM assembly means literal or immediate addressing mode
- Illegal to use # with DCD, DCW, or DCB