Microcomputers I – CE 320

Mohammad Ghamari, Ph.D.
Electrical and Computer Engineering
Kettering University

Lecture11: Advanced Arithmetic Instructions

Announcement

- Lecture 9 and 10 are already uploaded on BB.
- You are going to have your third quiz on Friday, Feb 19.

Topics and tentative schedule

No.	Topic	Tentative schedule	Assigned students
1	SCI	Week 7, M	Ryan Easter, Will Kaspari, and Joseph Saval
2	SPI	Week 7, W	Stephen White, Ethan Knott, Josiah Jaster
3	I^2C	Week 7, F	Roshan Cheriyan, Chase Philport, Brendan Kinell
4	ADC	Week 8, M	**Humzah Hafeez,** Timo Budiono, Henry Bensted
5	PWM	Week 8, W	Brent , Kasril George
6	CAN	Week 8, F	**Jack Lindner,** Mason Nordstrom, Jordyn Marlow
7	Input Capture & Output Compare	Week 9, M	Kuwar Nagpal, Colin Quinn, Guy Compton
8		Week 9, W	
9		Week 9, F	

Today’s Topics

- Binary-Coded-Decimal (BCD) Addition
- Use of basic multiplication and division instructions.
- Use of shift and rotate instructions

Binary-Coded-Decimal

- Although computers work internally with binary numbers, the input and output equipment generally uses decimal numbers
  - Since most logic circuits only accepts two-valued signals, the decimal numbers needs to be coded in terms of binary signals.

In binary code: 2,538 is represented as 1001 1110 1010
In BCD: 2538 is represented as 0010 0101 0011 1000

2 5

- This representation is called binary-coded-decimal (BCD)
  - E.g.: 2538 ➡0010 0101 0011 1000 bcd
  - **Advantage** of BCD encoding method is the simplicity of input/output conversion.
  - **Disadvantage** is the complexity of arithmetic processing, it must be converted into and out of binary for processing!

- The HCS12 microcontroller uses binary numbers
  - Numbers may be enter as a BCD, but they will be treated as binary numbers by the normal arithmetic commands.

Binary-Coded-Decimal

- So how are BCD values processed?
  - **Option 1**
    - Translate BCD to binary on input
    - Operate in binary
    - Translate binary to BCD before output (add a correction factor)
      - A special instruction supports this … DAA
  - **Option 2**
    - Adjust the result of BCD arithmetic after every operation
      - Determining and adding correction factors based on results
    - The DAA instruction can help
    - Dedicated “BCD representation” memory locations
      (all numbers in memory do not have the same “value”)

Binary-Coded-Decimal Addition

- Addition of \$25 with \$31 produces a result of \$56 according to the rules of binary addition
  - This is also a correct BCD answer, 25 + 31 = 56

- Problem occurs when addition of two BCD digits generates a **sum greater than 9**, because the sum is incorrect in the decimal number system (needs to be between 0 and 9)
  - Ex: \$18 + \$47 = \$5F according to rules of binary addition and a result of 18 + 47 = 65 according to the rules of decimal addition

- BCD sum correction factors
  - Adding \$6 to <u>every sum digit greater than 9 </u>(A-F), or
  - Adding \$6 to every sum digit that had a carry of 1 to the next higher digit (half carry and carry bits in the CCR)

Binary-Coded-Decimal Addition

Example1:

$$\begin{array}{r} \$18 \\ +\ \$47 \\ \hline \$5F \end{array}$$

Not correct since 18+47=65 in decimal addition

-In the above addition,

8+7 resulted in a half carry

Applying adjustments:

$$\begin{array}{r} \$5F \\ +\quad \mathbf{6} \\ \hline \$65 \end{array}$$

Example2:

$$\begin{array}{r} \$98 \\ +\ \$98 \\ \hline \$130 \end{array}$$

Not correct since 98+98=196 in decimal addition

-In the above addition,

8+8 resulted in a half carry and

9+9 resulted in a carry

Applying adjustments:

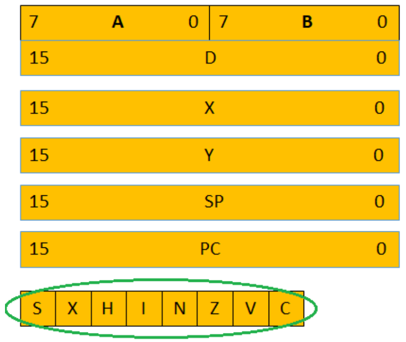
$$\begin{array}{r} \$130 \\ +\quad \mathbf{66} \\ \hline \$196 \end{array}$$

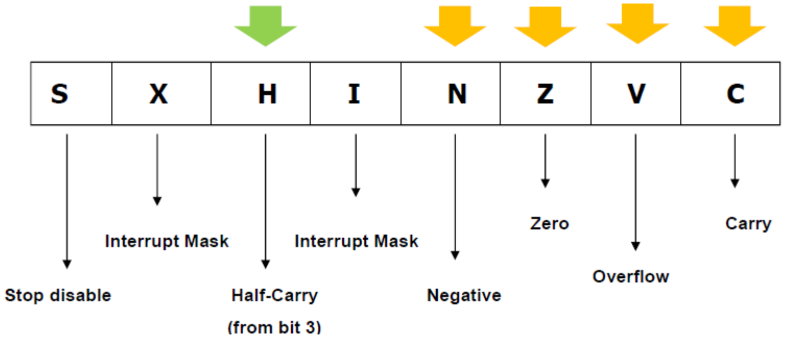
Binary-Coded-Decimal Addition

- The fifth bit of the condition code register is the **half-carry**, or **H flag**.

- A carry from the lower nibble to the higher nibble during the addition operation is a **half-carry**.
  - A half-carry of **one** during addition **sets the H flag to one**.
  - A half-carry of **zero** during addition **clears it to zero**.

- If there is a **carry** from the high nibble during addition, the **C flag** is set to one, which indicates that the high nibble is incorrect.
  - A $6 must be added to the high nibble to adjust it to the correct BCD sum.





Binary-Coded-Decimal Addition

- How to detect the illegal BCD sum following a BCD addition?

- The HCS12 provides a **decimal adjust accumulator A** instruction (**DAA)**, which takes care of all these detailed detection and correction operations.
  - There is no necessity of writing program to detect illegal BCD sum following a BCD addition.
  - The **DAA** instruction monitors the sums of BCD additions and the C and H flags and automatically adds $6 to any nibble that requires it.

- Rules for using DAA instruction:
  1. The **DAA** instruction can only be used for BCD addition. It does not work for subtraction or hex arithmetic.
  2. The **DAA** instruction must be used immediately after one of the three instructions that leave their sum in accumulator A. (These three instructions are **ADDA**, **ADCA**, **ABA**.)
  3. The numbers added must be legal BCD numbers to begin with.

BCD Example Code

- How to write an instruction sequence to add the BCD numbers stored at memory locations $800 and $801, and store the sum at $810?

Solution:

```
ldaa    $800        ; load the first BCD number in A
adda    $801        ; perform addition
daa                 ; decimal adjust the sum in A
staa    $810        ; save the sum
```

Multiplication

- Three different multiplication instructions
  - MUL
    - **Unsigned** 8 by 8 multiplication
    - D (A:B) ← A * B
  - EMUL
    - **Unsigned** 16 by 16 multiplication
    - Y:D ← D * Y
  - EMULS
    - Signed 16 by 16 multiplication
    - Y:D ← D * Y
- Note:
  - Register Y is being used for multiplication.

- Multiplies the **8-bit unsigned integer** in accumulator A by the **8-bit unsigned integer** in accumulator B to obtain a **16-bit unsigned result** in double accumulator D.
- The **upper byte** of the product is in accumulator A whereas the **lower byte** of the product is in B.

- Multiplies the **16-bit unsigned integers** stored in accumulator D and index register Y and leaves the product in these two registers.
- The **upper 16 bits** of the product are in Y whereas the **lower 16 bits** are in D.

- Multiplies the **16-bit signed integers** stored in accumulator D and index register Y and leaves the product in these two registers.
- The **upper 16 bits** of the product are in Y whereas the **lower 16 bits** are in D.

Multiplication

Example

- How to write an instruction sequence to **multiply** the contents of index register X and double accumulator D, and store the product at memory locations $800~$803.

```
sty   $810     ; save Y in a temporary location
tfr   x,y      ; transfer the contents of X to Y
emul           ; perform the multiplication
sty   $800     ; save the upper 16 bits of the product
std   $802     ; save the lower 16 bits of the product
ldy   $810     ; restore the value of Y
```

- There is no instruction to multiply the contents of double accumulator D and index register X.
- However, we can transfer the contents of index register X to index register Y and execute the EMUL instruction.
- If index register Y holds useful information, then we need to save it before the data transfer.

Division

- Five different division instructions
  - IDIV
    - **Unsigned** 16 by 16 integer division
    - X ← (quotient) (D) / (X)
    - D ← (remainder) (D) / (X)
  - IDIVS
    - **Signed** 16 by 16 integer division
    - X ← (quotient) (D) / (X)
    - D ← (remainder) (D) / (X)
  - FDIV
    - Like **IDIV**, but 16 by 16 **fractional** division.
    - Expect dividend to be smaller than divisor.
    - X ← (quotient) (D) / (X)
    - D ← (remainder) (D) / (X)

Divides an **unsigned 16-bit** dividend in double accumulator D by the **unsigned 16-bit** divisor in index register X, producing an **unsigned 16-bit quotient** in X, and an **unsigned 16-bit remainder** in D.

Divides the **signed 16-bit** dividend in double accumulator D by the **signed 16-bit** divisor in index register X, producing **a signed 16-bit quotient** in X, and a **signed 16-bit remainder** in D.

Divides an **unsigned 16-bit** dividend in double accumulator D by an **unsigned 16-bit** divisor in index register X, producing an **unsigned 16-bit quotient** in X and an **unsigned 16-bit remainder** in D.

Division- continued

- EDIV
  - **Unsigned** 32 by 16 integer division
  - Y ← (quotient) (Y:D) / (X)
  - D ← (remainder) (Y:D) / (X)

- Performs an **unsigned 32-bit** by **16-bit division**.
- The dividend is the register pair Y and D with Y as the upper 16-bit of the dividend.

- EDIVS
  - **Signed** 32 by 16 integer division
  - Y ← (quotient) (Y:D) / (X)
  - D ← (remainder) (Y:D) / (X)

Performs a **signed 32-bit** by **16-bit** division using the same operands as the EDIV instruction does.

- Note: Register X is being used for division while Y for multiplication.

Division

Example

- Write an instruction sequence to **divide** the signed 16-bit number stored at memory locations \$805~\$806 by the 16-bit unsigned number stored at memory locations \$820~\$821, and store the quotient and remainder at \$900~\$901 and \$902~\$903, respectively.

```
ldd     $805        ; place the dividend in D
ldx     $820        ; place the divisor in X
idivs               ; perform the signed division
stx     $900        ; save the quotient
std     $902        ; save the remainder
```

- Before we can perform the division, we need to place the dividend and divisor in D and X, respectively.

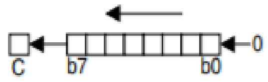
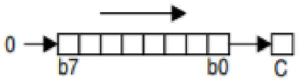
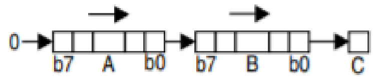
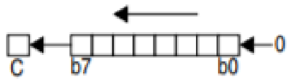
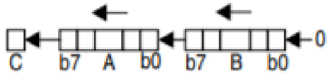
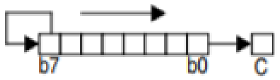
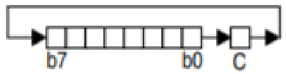
Shift and Rotate Instructions

- Shift and rotate instructions are useful for **bit field manipulation**.

- Shift or Rotate instruction shifts or rotates the operand by one bit.

- The HCS12 has shift instructions that can operate on accumulators **A**, **B**, and **D**, or on a **memory location**.

Shift and Rotate Table

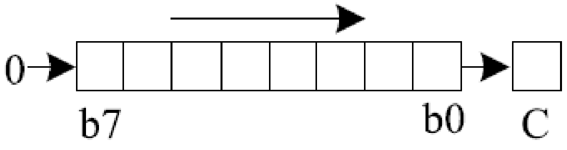
Mnemonic	Function	Operation
Logical Shifts		
LSL LSLA LSLB	Logic shift left memory Logic shift left A Logic shift left B	
LSLD	Logic shift left D	
LSR LSRA LSRB	Logic shift right memory Logic shift right A Logic shift right B	
LSRD	Logic shift right D	
Arithmetic Shifts		
ASL ASLA ASLB	Arithmetic shift left memory Arithmetic shift left A Arithmetic shift left B	
ASLD	Arithmetic shift left D	
ASR ASRA ASRB	Arithmetic shift right memory Arithmetic shift right A Arithmetic shift right B	
Rotates		
ROL ROLA ROLB	Rotate left memory through carry Rotate left A through carry Rotate left B through carry	
ROR RORA RORB	Rotate right memory through carry Rotate right A through carry Rotate right B through carry	

Shifts

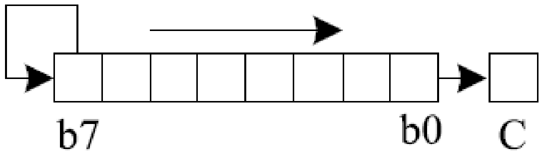
Logical and Arithmetic Right Shift

NOTE: The V = N⊕C for all of these, but the only real use is in the shift **lefts.**

- LSRx
  - **Logical** Shift **Right** for memory, A, B, or D
  - Bit shifted out into C CCR bit and **0 shifted in**.
  - Affects all **four** CCR bits
  - Unsigned divide by 2



- ASRx
  - **Arithmetic** Shift **Right** for memory, A, B, or D
  - Bit shifted out into C CCR bit and **sign bit replicated**.
  - Affects all **four** CCR bits
  - Unsigned divide by 2

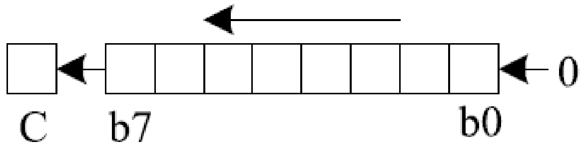


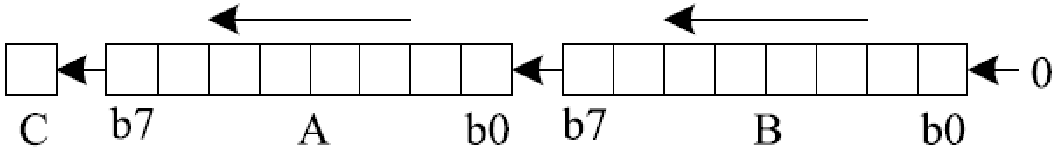
Shifts

Logical and Arithmetic Left Shift

NOTE: The V = N⊕C for all of these, but the only real use is in the shift **lefts.**

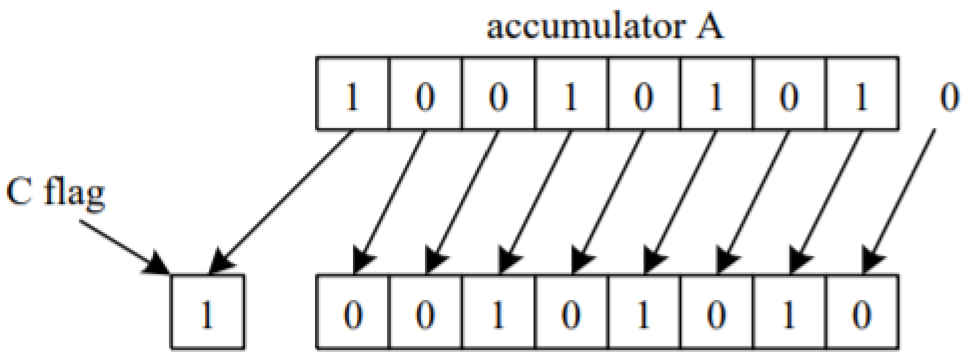
- ASLx, LSLx
  - **Arithmetic and Logical** Shift **Left** for memory, A, B, or D
  - Bit shifted out into C CCR bit and **0 shifted in.**
  - Affects all four CCR bits
  - Unsigned/signed divide by 2
  - Implemented with the **same opcodes**, no physical difference





Arithmetic Left Shift
Example

- Let’s consider what are the values of accumulator A and the C flag after executing the **ASLA** (Arithmetic Shift Left Accumulator A) instruction? Assume that originally A contains $95 and the C flag is 1.



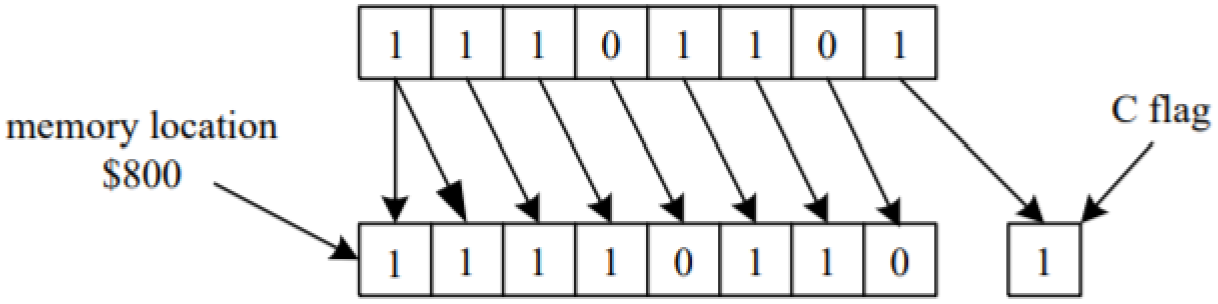
Operation of ASLA Instruction

Original value	New value
[A] = 10010101 C = 1	[A] = 00101010 C = 1

Execution Result of ASLA Instruction

Arithmetic <u>Right</u> Shift
Example

- What are the new values of the memory location at $800 and the C flag after executing the instruction **ASR** (Arithmetic Shift Right) $800? Assume that the memory location $800 originally contains the value of $ED and the C flag is 0.



Operation of ASR $800 Instruction

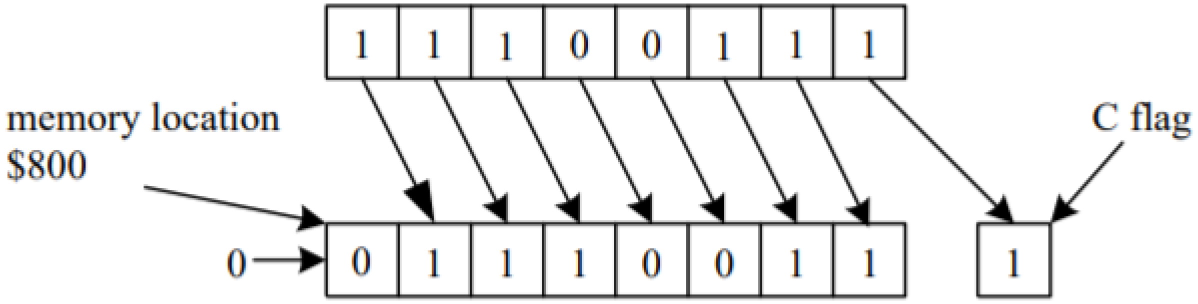
Original value	New value
[$800] = 11101101 C = 0	[$800] = 11110110 C = 1

Execution Result of ASR $800 Instruction

Logical <u>Right</u> Shift

Example

- What are the new values of the memory location at $800 and the C flag after executing the instruction LSR $800? Assume the memory location $800 originally contains $E7 and the C flag is 1.



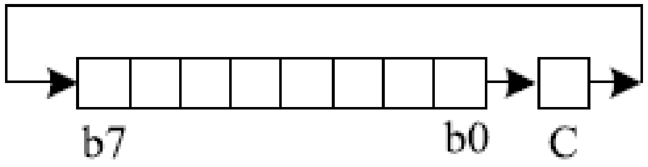
Operation of LSR $800 Instruction

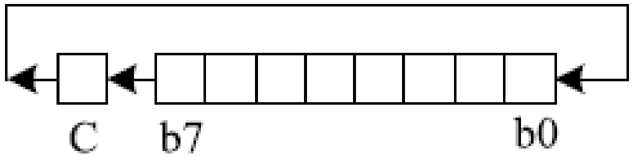
Original value	New value
[$800] = 11100111 C = 1	[$800] = 01110011 C = 1

Execution Result of LSR $800 Instruction

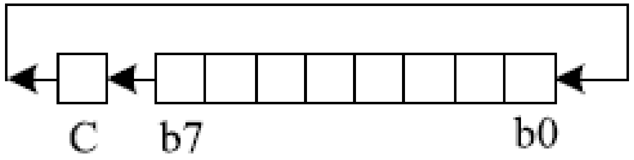
Rotates

- RORx, ROLx
  - **Rotate Right/Left** for memory, A, or B.
  - C CCR bit shifted in, bit shifted out goes to C CCR Bit.
  - Affects all four CCR bits.
- Rotates are used to extend shift operations to multi-precision numbers.

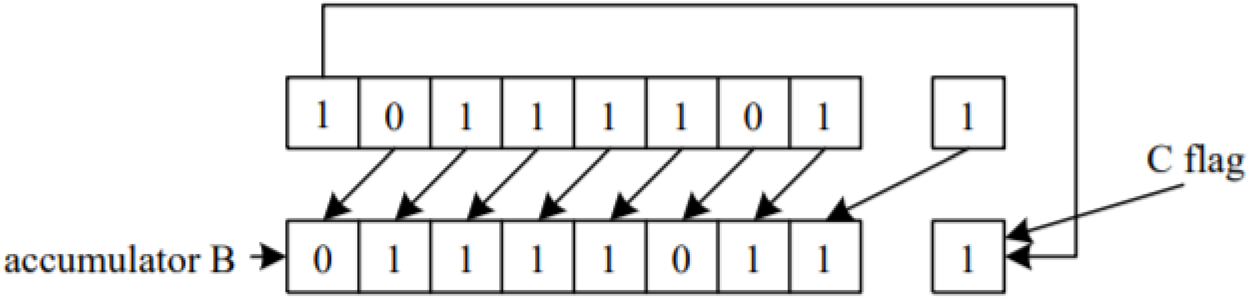




Rotate Left
Example



- Compute the new values of accumulator B and the C flag after executing the instruction ROLB (Rotate B Left through Carry). Assume the original value of B is $BD and C flag is 1.



Operation of ROLB Instruction

Original value	New value
[B] = 10111101 C = 1	[B] = 01111011 C = 1

Execution Result of ROLB Instruction

Examples

NOTE: The V = N⊕C for all of these, but the only real use is in the shift **lefts.**

C1 = 1100 0001

Code	A	N	Z	V	C	A (unsigned)	A (signed)
LDAA #$C1	C1	1	0	0	-	193	-63
ASLA	82	1	0	0	1	130	-126
ASLA	04	0	0	1	1	4	4

Code	A	N	Z	V	C	A (unsigned)	A (signed)
LDAA #$C1	C1	1	0	0	-	193	-63
ASRA	E0	1	0	0	1	224	-32
ASRA	F0	1	0	0	0	240	-16

Code	A	N	Z	V	C	A (unsigned)	A (signed)
LDAA #$C1	C1	1	0	0	-	193	-63
LSRA	60	0	0	1	1	96	96
LSRA	30	0	0	0	0	48	48

Homework Example1

Exponential Filter

- An **exponential filter** is often used to condition an incoming input signal.

$$X_{ave}(t) = \alpha \times X(t) + (1 - \alpha) \times X_{ave}(t-1)$$

- *X(t)* is an input signal at time *t*.
- The weight factor α must be between 0 and 1.
- α determines how quickly the filtered value will respond to a change in input.
- Let's write a program
    - $\alpha = 5/9$
    - 8 bit input is supplied in address \$0000
    - 8 bit output is written to \$0001

Example1 – source code

Exponential Filter

$$X_{ave}(t) = 5/9 \times X(t) + (1-5/9) \times X_{ave}(t-1)$$

$$X_{ave}(t) = (5 \times X(t))/9 + ((9-5) \times X_{ave}(t-1))/9$$

$$X_{ave}(t) = \boxed{\alpha \times X(t)} + \boxed{(1-\alpha) \times X_{ave}(t-1)}$$

```
alpn        EQU         5
alpd        EQU         9

            ORG         $0000
Xin         DS.B        1
Xave        DS.B        1

            ORG         $2000
Loop:
            ; alpha * X (t)
            LDAA        Xin         ; A <-- (Xin)
            LDAB        #alpn       ; B <-- 5
            MUL                     ; D <-- (A) * (B) = (Xin) * 5
            LDX         #alpd       ; X <-- 9
            IDIV                    ; X <-- Q((D)/(X)),    D <-- R((D)/(X))
            TFR         X,Y         ; Y <-- (Xin * 5) / 9
            ; (1 - alpha) * Xave (t)
            ; (1 - 5 / 9) = (9 - 5) / 9
            LDAA        Xave
            LDAB        #(alpd-alpn)          ; (9-5) = 4
            MUL                     ; D <-- Xave * 4
            LDX         #alpd       ; X <-- 9
            IDIV                    ; X <-- (Xave * 4) / 9
            TFR         X,B         ; The result of the second term in the
equation
            TFR         Y,A         ; The result of the first term in the
equation
            ABA                     ; A <-- (A) + (B)
            STAA        Xave        ; Save the average value. Xave <-- (A)
            BRA         Loop
            SWI
```

- The program uses register Y to save the first term.
- The second term is saved to register X and transferred to B
- Transfer register Y to A to add those two terms. (ABA)

Example1 – listing file

```
             =00000005         alpn    EQU     5
             =00000009         alpd    EQU     9

             =00000000                 ORG     $0000
        0000 +0001             Xin     DS.B    1
        0001 +0001             Xave    DS.B    1

             =00002000                 ORG     $2000
        2000                   Loop:
                                       ; alpha * X (t)
        2000 96 00                     LDAA    Xin      ; A <-- (Xin)
        2002 C6 05                     LDAB    #alpn    ; B <-- 5
        2004 12                        MUL              ; D <-- (A) * (B) = (Xin) * 5
        2005 CE 0009                   LDX     #alpd    ; X <-- 9
        2008 1810                      IDIV             ; X <-- Q( (D)/(X) ),    D <-- R( (D)/(X) )
        200A B7 56                     TFR     X,Y      ; Y <-- (Xin * 5) / 9
                                       ; (1 - alpha) * Xave (t)
                                       ; (1 - 5 / 9) = (9 - 5) / 9
        200C 96 01                     LDAA    Xave
        200E C6 04                     LDAB    #(alpd-alpn)    ; (9-5) = 4
        2010 12                        MUL              ; D <-- Xave * 4
        2011 CE 0009                   LDX     #alpd    ; X <-- 9
        2014 1810                      IDIV             ; X <-- (Xave * 4) / 9
        2016 B7 51                     TFR     X,B      ; The result of the second term in the equation
        2018 B7 60                     TFR     Y,A      ; The result of the first term in the equation
        201A 1806                      ABA              ; A <-- (A) + (B)
        201C 5A 01                     STAA    Xave     ; Save the average value. Xave <-- (A)
        201E 20 E0                     BRA     Loop
        2020 3F                        SWI
```

Homework Example2

Shift and Rotate

- Shift the **<u>signed</u>** three-byte number in addresses $1000 - $1002 right two bits.

```
bits    EQU             2
num             EQU             $1000

                ORG             $2000
                LDAB    #bits
                LDX             num
again:
                ASR             0, X
                ROR             1, X
                ROR             2, X
                ; DBNE abdxys, rel9
                ;; Decrement Counter and Branch if not 0
                ; (cntr) – 1 → (cntr)
                ; if cntr not 0 then Branch
                DBNE    B, again
                SWI
```

Questions?

Wrap-up

What we’ve learned

- Binary-Coded-Decimal (BCD) Addition
- **Multiplication** and **division** instructions.
- Use of **shift** and **rotate** instructions

What to Come

- Boolean logic instructions