EECE 218
Microcontrollers

Multiple precision math and multiplication
Some other math instructions

- **ABA**: Add B to A: \( A \leftarrow A + B \)
- **ABX**: Add B to X: \( X \leftarrow X + B \)
  - B sign-extended
  - Also: ABY
- **ADDA, ADDB**: Adds to A (B) the operand
  - May generate carry (in the C bit)
- **ADCA, ADCB**: Adds to A (B) the operand + C
- **ADDD**: Add to D: \( D \leftarrow D + \text{operand}(16) \)
- **SUBA**: Subtract from A: \( A \leftarrow A - \text{operand} \)
  - Also: SUBB
  - May generate borrow (in the C bit)
- **SBA**: Subtract B from A: \( A \leftarrow A - B \)
- **SBCA, SBCB**: Subtracts from A (B) the operand + borrow
- **SUBD**: Subtract from D: \( D \leftarrow D - \text{operand}(16) \)
Multiple precision addition

\[
\begin{array}{c}
\text{H} & \text{L} \\
\text{H} & \text{L} \\
\hline
\text{V1} & \text{V2} \\
\text{V1H+V2H+C} & \text{V1L+V2L} \\
\end{array}
\]

; Data
V1 DS.B 2
V2 DS.B 2
SUM DS.B 2

Principles:
- Operands must have the same number of bytes
- Work from least significant byte towards most significant byte
- Utilize carry

→ Can be extended to arbitrary length

; Code
MADD
LDAA V1+1
ADDA V2+1
STAA SUM+1
LDAA V1
ADCA V2
STAA SUM
RTS
Multiplication using add and shift

- How do we multiply?

\[ 24 \times 17 \]

\[
\begin{array}{c}
24 \\
168 \\
+24 \\
408
\end{array}
\]

Note: shift left!
Multiplication

- In binary

24 = 11000 : Multiplicand
17 = 10001 : Multiplier

11000
00000
00000
00000
11000
1100011000 = 198

Algorithm:
1. Consider each bit of multiplier
2. If bit is one, add shifted multiplicand to partial result.
Multiplication

**Notes:**

- 8 x 8 bits yields 16 bit result
- Addition and left shift must be done on 16 bits!
Multiplication

; 8x8 bit unsigned multiplication
; A: multiplicand; B: multiplier
; Data
ANSRHI   DS.B 1 ; Answer high byte
ANSRLO   DS.B 1 ; low byte
DPACHI   DS.B 1 ; Double precision ‘acc’
DPACLO   DS.B 1
### Multiplication

<table>
<thead>
<tr>
<th>MULT</th>
<th>CLR ANSRHI ; Clear ANS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR ANSRLO</td>
<td></td>
</tr>
<tr>
<td>CLR DPACHI</td>
<td></td>
</tr>
<tr>
<td>STA A DPACLO ; Set up DPAC</td>
<td></td>
</tr>
<tr>
<td>LOOP LSRB ; Check M'er bit</td>
<td></td>
</tr>
<tr>
<td>BCC DTEST ; Branch if bit = 0</td>
<td></td>
</tr>
<tr>
<td>PSHB ; Save B</td>
<td></td>
</tr>
<tr>
<td>LDD ANSRHI ; Add DPAC to ANSR</td>
<td></td>
</tr>
<tr>
<td>ADDDPACHI</td>
<td></td>
</tr>
<tr>
<td>STD ANSRHI</td>
<td></td>
</tr>
<tr>
<td>PULB ; Restore B</td>
<td></td>
</tr>
<tr>
<td>DTEST TSTB ; Is B=0?</td>
<td></td>
</tr>
<tr>
<td>BEQ DONE ; End if yes</td>
<td></td>
</tr>
<tr>
<td>ASL DPACLO ; DP left shift</td>
<td></td>
</tr>
<tr>
<td>ROL DPACHI</td>
<td></td>
</tr>
<tr>
<td>BRA LOOP ; Do it again</td>
<td></td>
</tr>
<tr>
<td>DONE RTS ; End of subroutine</td>
<td></td>
</tr>
</tbody>
</table>

One hardware instruction:

\[
\text{MUL} : D \leftarrow A \times B
\]

8x8 bit unsigned multiplication
Passing parameters to subroutines

- Subroutines: reusable code
  - Like mathematical formulas…
  - How to supply different values to subroutines?

- Parameter passing: mechanism to provide data for and get results from subroutines

- Three techniques:
  - In registers - Fast, but only few registers!
  - In defined memory variables
    - Somewhat slower, lots of space
    - Non-reusable if the same subroutine is ‘active’ multiple times (e.g. recursive call)
      - Each invocation of the subroutine needs its own copy of the parameters
Passing parameters to subroutines

- Passing parameters on the stack

**Caller:**
- PSHA ;V1
- PSHB ;V2
- JSR Callee
- PULA ;Cleanup!
- PULB

**Callee:**
- TSX
- LDAA 2,X A has V2
- LDAB 3,X B has V1
- ....
- RTS

Note:
- Caller must clean up!
- If callee calls itself, it pushes parameters, calls/cleans up.

Stack upon entering S/R:
## Multiple precision multiplication

<table>
<thead>
<tr>
<th>8 bits (digits by digit)</th>
<th>16 bits (byte by byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b</td>
<td>ab cd</td>
</tr>
<tr>
<td>c d</td>
<td>ef gh</td>
</tr>
<tr>
<td>---</td>
<td></td>
</tr>
<tr>
<td>(b*d)</td>
<td>(cd*gh)</td>
</tr>
<tr>
<td>(b*c) 0</td>
<td>(cd*ef) 00</td>
</tr>
<tr>
<td>(a*d) 0</td>
<td>(ab*gh) 00</td>
</tr>
<tr>
<td>(c*a) 0 0</td>
<td>(ab*ef) 00 00</td>
</tr>
<tr>
<td>4bit shifted</td>
<td>8bit shifted</td>
</tr>
<tr>
<td>8bit shifted</td>
<td>16bit shifted</td>
</tr>
<tr>
<td>(sum)</td>
<td>(sum)</td>
</tr>
</tbody>
</table>
16 bit multiply

; Data – MP and MC are loaded with values
MPHI   DS.B 1  ; Multiplier H
MPLO   DS.B 1  ; Multiplier L
MCHI   DS.B 1  ; Multiplicand H
MCLO   DS.B 1  ; Multiplicand L
ANS3   DS.B 1  ; Answer MSB
ANS2   DS.B 1  ;
ANS1   DS.B 1  ;
ANS0   DS.B 1  ; Answer LSB
16 bit multiply

MPY16
LDD #0 ; Clear ANSR MSB 2 bytes
STD ANS3
LDAA MPLO ; Calculate cd*gh
LDAB MCLO
MUL
STD ANS1
LDAA MPLO
LDAB MCHI
MUL ; Calculate cd*ef
ADDD ANS2 ; Add partial result CARRY?? No!
STD ANS2
LDAA MPHIL
LDAB MCLO
MUL ; Calculate ab*gh
ADDD ANS2 ; Add partial result  CARRY?? Maybe!
STD ANS2
ROL ANS3 ; Roll Carry into ANS3
LDAA MPHIL
LDAB MCHI
MUL
ADDD ANS3
STD ANS3
RTS
16bit multiply

- Carry on first ‘cross-term’
  Worst case: \( FF \times FF = FE01 \)
  \[ \begin{align*}
  &+ 00FF \\
  \hline
  &FF00 \leftarrow \text{No carry!}
  \end{align*} \]

- Carry on second ‘cross-term’
  Worst case: \( FF \times FF = FE01 \)
  \[ \begin{align*}
  &+ FF00 \\
  \hline
  \text{(sum)} \leftarrow \text{Carry!}
  \end{align*} \]
More instructions

- EMUL : 16x16 bit unsigned multiply
  \[ Y : D \leftarrow Y \times D \quad \text{--- Result is in 32bit register } Y:D \]

- EMULS : 16x16 bit signed multiply
  \[ Y : D \leftarrow Y \times D \quad \text{--- Result is in 32bit register } Y:D \]
Division instructions

- **IDIV:** unsigned integer divide
  D divided by X, quotient in X, remainder in D
  If division by 0 → X = $FFFF$ and Carry set

- **IDIVS:** signed integer divide (see IDIV)

- **EDIV:** 32 by 16 bit unsigned divide
  Y:D divided by X, quotient in Y, remainder D

- **EDIVS:** 32 by 16 bit signed divide
  Y:D divided by X, quotient in Y, remainder D

- **FDIV:** 16 by 16 fractional divide
  D divided by X, quotient in X, remainder in D
  Assumes D < X, result is BINARY FRACTION (bits correspond to negative powers of 2, i.e. ½, ¼, 1/8, etc.)