Programming in Assembly

Data Structures, Addressing Modes, and Flow Control Basics

Autoincrement/decrement modes

- In our generic CPU:

  `MOVE   (R1)+,R2`

  – Always increments R1 by 2 (next legal address)
- In the 68000, the increment/decrement depends on the operand size
  – Suppose A0 = $00002000
  – MOVE.B (A0)+,D0 → A0 = $2001
  – MOVE.W (A0)+,D0 → A0 = $2002
  – MOVE.L (A0)+,D0 → A0 = $2004

A few more assembly instructions…

- **`LEA`** – Load Effective Address
  – Loads the *address of* the operand, instead of the value.

```
        MOVE.W  ARRAY,A0
        LEA     ARRAY,A0
        ORG     $400500
ARRAY   DC.W    12,15,30,5
        END
```

A0 ← 12

A0 ← $2000

Another LEA example

```
        LEA     ARRAY,A0    ; A0 ¬ $2000
        MOVE.W  4(A0),D0    ; D0 ¬ 8
        LEA     4(A0),A1    ; A1 ¬ $2004
        ORG     $2000
ARRAY   DC.W    12,4,8
```

$2000	12
$2002	4
$2004	8

Some more details about EXT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	***7***	***6***	***5***	***4***	***3***	***2***	***1***	***0***

- **`EXT.W`** extends a *byte* to a *word*
  - Bit 7 is copied into bits 8-15.
- **`EXT.L`** extends a *word* into a *long word* (or double word)
  - Bit 15 is copied into bits 16-31
- Both **`EXT`** instructions set CCR bits Z and N

Programming Conventions

```
; This example demonstrates some techniques we will use for all our programming
; assignments…
        ORG     $1000        ; 1) Use an explicit ORG statement
        CLR.W   D3           ; 2) Code comes first, then data
        MOVEA.L  #NUMS,A1
        MOVE.W  LEN,D2       ; NO SPACES BETWEEN OPERANDS
LOOP    ADD.W   (A1)+,D3
        SUBQ.W   #1,D2       ; 3) We end every program by TRAPing back
        BNE     LOOP         ;  into the simulator.  This is done by placing
        MOVE.W  #EXIT,D7     ;    #228 into register D7, and executing the
        TRAP    #14          ;    instruction TRAP #14.
;
EXIT    EQU     228          ; Use EQU statements to make code more readable
;
LEN     DC.W    5            ; Data comes after the code and should
NUMS    DC.W    123,-56,453,-1045,765   ; be well-commented!
        End                  ; The END statement comes after the data.
```


Setting the CCR bits.

- The low-order 4 bits of the status register (**`SR`**) are the condition code bits: **`NZVC`**
- The last column of table C.4 shows which condition codes are affected by every instruction
  - N = was the result negative?
  - Z = was the result zero?
  - V = was there an overflow?
  - C = was there a carry-out?

Branch Instructions

- The *mnemonics* for the branch instructions assume that you are following a SUB or a **`CMP`** instruction:
  - **`BEQ`** (branch when equal) **`Z=1`**

    ```
    SUB.W  D3,D4
    BEQ    LOOP
    ```
  - when does Z=1?
    - When [D3] and [D4] are equal!
- <u>Remember</u> that CMP and SUB compute *[dest] – [src]*

Another way to think about Bcc

- You can also think of B*cc* as comparing the *result* of the last operation to zero:

```
       MOVE.W   #-12,D0    ; D0 is a counter, starting
       LEA      ARRAY,A0   ; … at the value -12
LOOP   ADD.W    (A0)+,D1
       ADDQ.W   #1,D0      ;Add 1 to the counter
       BLT      LOOP       ;Loop while result < 0
ARRAY  DC.W     12,4,8
```


Branches and Overflow

- In the 68000 the V bit is set on *2's complement overflow* for the operand size (B, W, L)
  - – BGE (branch when greater or equal)
    - Branch when $N \oplus V = 0$
  - – Example: SUB.B D1, D2 (DEST – SRC)
    - N=0 when D2 ≥ D1
    - What if [D1] = 1, and [D2] = –128?
    - **Can we represent –129 in an 8-bit byte in 2's complement?** (10000000 + 11111111)
    - The result is 127 (positive), N=0, V=1
    - We don't branch, which is good since –128 < 1 !

Operation sizes and Overflow

- In the 68000, the V-bit is set when there is a 2's complement overflow ***for the size of operand specified in the instruction!***
- In other words, suppose D0 = $00000063

  `ADD.B  #$60,D0` sets V=1 and N=1

  `ADD.W  #$60,D0` sets V=0 and N=0
- Same thing goes for the carry bit
  - – If a byte operation would produce a carry into bit 8, the C bit is set, and bit 8 retains its old value.

Implementing C-like flow control:

- **`for(i=0; i<5; i++) {…}`**

```
        CLR.B     D0
LOOP    …
        ADDQ.B    #1,D0
        CMPI.B    #5,D0
        BLT       LOOP
```

This is easy to read, and necessary if you want to use the value of i.

`D0 - #5`

However, you have to get the immediate value #5 from memory repeatedly. There is a more efficient way to loop 5 times…

Fixed loops

- Using a ***down counter.*** A more efficient way to loop 5 times:

```
        MOVEI.B   #5,D0
LOOP    do something…
        SUBQ.B    #1,D0
        BNE       LOOP   ; BRA if Z=0
        move on…
```


While loops

- while (j < 5) {…} *Test at beginning.*
- condition: (j < 5) opposite: (j ≥ 5)

```
        MOVE.W    j,D0    ;get j from memory
LOOP    CMPI.W    #5,D0
        BGE       NEXT    ; exit loop if
        …                 ; condition false
        ADDQ.W    #1,D0
        BRA       LOOP
NEXT    …
j       DC.W      2
```


Other ways to use branch

- You don't have to follow the mnemonics
- The best thing to do is to look at the branch condition in Table C.6
- EXAMPLE: **for(j=-5; j!=0; j++){…}**

```
        MOVE.B    #-5,D0
LOOP    BEQ       DONE
        do something…
        ADDQ.B    #1,D0
        BRA       LOOP
DONE    move on…
```

We'll use D0 for j

BRA if Z=1

BRA doesn't affect the CCR

Conditionals

- The most efficient way to code this is to *skip* the code {…} if the condition is *not* true.

```
if (x == 5){.}
```

```
        MOVE.W    x,D2
        CMPI.W    #5,D2
        BNE  SKIP
        ; {
        ; ...
        ; }
SKIP    ......
```

If…then…else conditionals

```
if (y > 3)
{
  true-code
}
else
{
  false-code
};
```

```
     MOVE.W   Y,D0
     CMPI.W   #3,D0
     BLE      ELSE
     true-code
     BRA      NEXT
ELSE false-code
NEXT next-instruction
```

Again we test for the opposite of the ***if*** condition, and skip the true-code if necessary. At the end of the true-code, we use a BRA to avoid **also** executing the false code.

Putting it together: Summing an array

```
        ORG      $1000
        CLR.W    D3              ; The sum will go into D3
        MOVEA.L  #NUMS,A1        ; A1 -> current array element
        MOVE.W   LEN,D2          ; D2 = number of array elements remaining
LOOP    ADD.W    (A1)+,D3        ; Add the next element
        SUBQ.W   #1,D2           ; Now there is one less remaining
        BNE      LOOP            ; Continue if D2 != 0
        MOVE.B   #EXIT,D7        ;
        TRAP     #14             ; Exit back to the simulator
;
EXIT    EQU      228
;
         ORG     $2000
LEN     DC.W     5                       ; LEN = Size of the array
NUMS    DC.W     123,-56,453,-1045,765   ; NUMS = the array
        End
```


Introduction to the stack

- A7 is a special address register, called the *stack pointer*.
- When programming assembly, we can use SP as an alias for A7. **`MOVEA.L #$3000,SP`**
- In the simulator, it is also called US (*user stack* pointer)
- There is also a supervisor stack pointer, but we won't worry about it yet.
- Since our program usually starts at a low memory address and grows downward, we start the stack at a high memory address and work backwards.

The Stack

- We push values onto the stack using *predecrement mode*

```
MOVE.W  D2,-(SP)
MOVE.W  D3,-(SP)
```

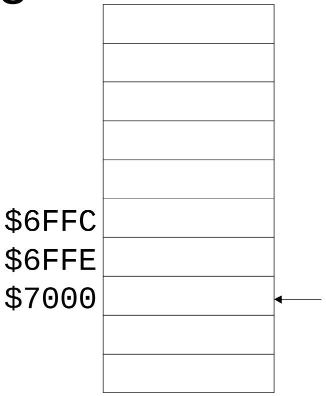
- We pop values from the stack using *postincrement mode*

```
MOVE.W  (SP)+, D3
MOVE.W  (SP)+, D2
```

- Some instructions affect the stack directly:
  – Look at JSR

Pushing and popping the stack

```
main   MOVE.W #12,-(A7)
       MOVE.W #20,-(A7)
       MOVE.W #30,-(A7)
       …
       MOVE.W (A7)+,D0
       MOVE.W (A7)+,D1
       MOVE.W (A7)+,D2
```



PUSH(12)	*Last-in, first-out*
PUSH(20)	*(LIFO)*
PUSH(30)	
POP	= 30
POP	= 20
POP	= 12

What is the stack used for?

- Temporary storage of variables
- Temporary storage of program addresses
- Communication with *subroutines*
  - Push variables on stack
  - Jump to subroutine
  - Clean stack
  - Return

Did you know?

- You can give C a “hint” about which variables to keep in registers?

```
register int counter;
int i, j;
counter = 0;
for (i=0; i<100; i++) {
  for (j=0; j<100; j++) {
      counter += 3;
  }
}
```


Example (post increment)

What does this program do?

```
TABLE_1     EQU       $002000
TABEL_2     EQU       $003000
N           EQU       $30
            :
            LEA       TABLE_1,A0
            LEA       TABLE_2,A1
            MOVE.B    #N,D0
NEXT        CMPM.B    (A0)+,(A1)+
            BNE       FAIL
            SUB.B     #1,D0
            BNE       NEXT
SUCCESS     :
            :
FAIL        :
```

Example (multiple precision subtration)

Two unsigned binary numbers each with 128 bits (16 bytes) and stored in memory starting at locations **Num1**, **Num2. Num1** is subtracted from **Num2** together with the result to be stored in memory starting at **Num2**

```
       ORG     $0400400

MPADD  MOVE.W  #3,D0          Four long words to be added
       ANDI    #$EF,CCR       Clear X-bit  in CCR
       LEA     Num1,A0 A0     points at start of source
       ADDA    #16,A0         A0 points to end of source + 1
       LEA     Num2,A1 A1     points at start of destination
       ADDA    #16,A1         A1 points to end of destination + 1
LOOP   SUBX.L  -(A0),-(A1)    Subtract pair of long words with borrow
       DBRA    D0,LOOP        Repeat until 4 long words are subtracted
       RTS

Num1      DS.L         4
Num2      DS.L         4
```


Example (multiple precision addition)

Two unsigned binary numbers each with 128 bits (16 bytes) and stored in memory starting at locations **Num1**, **Num2** are to be added together with the result to be stored in memory starting at Num2

```
       ORG     $0400400

MPADD  MOVE.W  #3,D0          Four long words to be added
       ANDI    #$EF,CCR       Clear X-bit  in CCR
       LEA     Num1,A0 A0     points at start of source
       ADDA    #16,A0         A0 points to end of source + 1
       LEA     Num2,A1 A1     points at start of destination
       ADDA    #16,A1         A1 points to end of destination + 1
LOOP   ADDX.L  -(A0),-(A1)    add pair of long words with borrow
       DBRA    D0,LOOP        Repeat until 4 long words are subtracted
       RTS

Num1      DS.L         4
Num2      DS.L         4
```