SEE 3223 Microprocessors

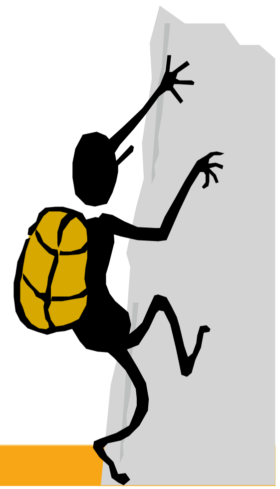
6: Flow Control

Muhammad Mun'im Ahmad Zabidi (munim@utm.my)

cc BY NC SA

Module 6: Program Control

- Contents
  - Unconditional Branch & Jump
  - Program Counter Relative Addressing
  - Compare Instructions
  - Conditional Branch Instructions
  - Implementing Basic Programming Constructs
  - Address Registers
  - Indexed Addressing



Review: Fetch - Execute Cycle

- The CPU operates in a two-phase fetch-execute mode.
  - In the first phase
    - the instruction is read from memory
    - The instruction copied into the instruction register, IR
    - The program counter is advanced to point to the next instruction
  - In the second phase
    - The instruction in IR is decoded
    - The instruction is executed
- 68000 has a variable instruction size (min 2 bytes, max 10 bytes).
  - The value in the program counter is increased by 2 in fetch phase
  - PC is increased by 0 to 8 in execute phase
- The order of instruction processing is sequential order (one by one).
- Sometimes, you want to execute instruction in a different order.
- This is done by putting a different address into the PC.

Flow Control

- Flow control - Ability to choose an instruction other than the following instruction
- Idea: just modify PC like any other register

```
ADD.L   #$30,PC             ; doesn't really work
```

- If it worked, it would powerful (and dangerous!)
- Instead, we have special instructions with limited abilities to modify the PC.

Flow Control Instructions

Instruction	Description
BRA	BRA (branch always) implements an unconditional branch, relative to the PC. The offset is expressed as an 8- or 16-bit signed integer. If the destination is outside of a 16-bit signed integer, BRA cannot be used.
JMP	JMP (jump) is similar to BRA. The only difference is that BRA uses only relative addresing, whereas JMP has more addressing modes, inluding absolute address.
Bcc	Bcc (branch on condition code) is used whenever program execution must follow one of two paths depending on a condition. The condition is specified by the mnemonic cc. The offset is expressed as an 8- or 16-bit signed integer. If the destination is outside of a 16-bit signed integer, Bcc cannot be used.
JSR **BSR** **RTS**	JSR and BSR branches to a subroutine. The PC is saved on the stack before loading the PC with the new value. RTS is used to return from the subroutine by restoring the PC from the stack.

cc	Condition	Branch Taken If
CC	Carry clear	C = 0
CS	Carry set	C = 1
NE	Not equal	Z = 0
EQ	Equal	Z = 1
PL	Plus	N = 0
MI	Minus	N = 1
VC	Overflow clear	V = 0
VS	Overflow set	V = 1
GE	Greater or equal	N'V + NV' = 0
GT	Greater than	NVZ + (NVZ)' = 1
LE	Less or equal	Z+(N'V+NV') = 1
LT	Less than	N'V + NV' = 1
HS	Higher or same	C = 0
LO	Lower	C = 1
HI	Higher	C'Z' = 1
LS	Lower or same	C + Z=1

BRA Instruction

- The BRA (for branch) instruction allows us to modify the PC by essentially adding to it or subtracting from it.
- A silly little example:

```
00001000            1          ORG      $1000
00001000  5280      2  START   ADDQ.L   #1, D0
00001002  60FC      3          BRA      START
00001004            4          END
```

- What does this code do? It infinitely loops, continually adding 1 to D0. Not very useful, but very simple.
- The machine language for BRA contains the offset $FC which says we want to subtract 4 from the PC, or add -4 (the reason it's 4 rather than 2 is that the PC starts at PC + 2):

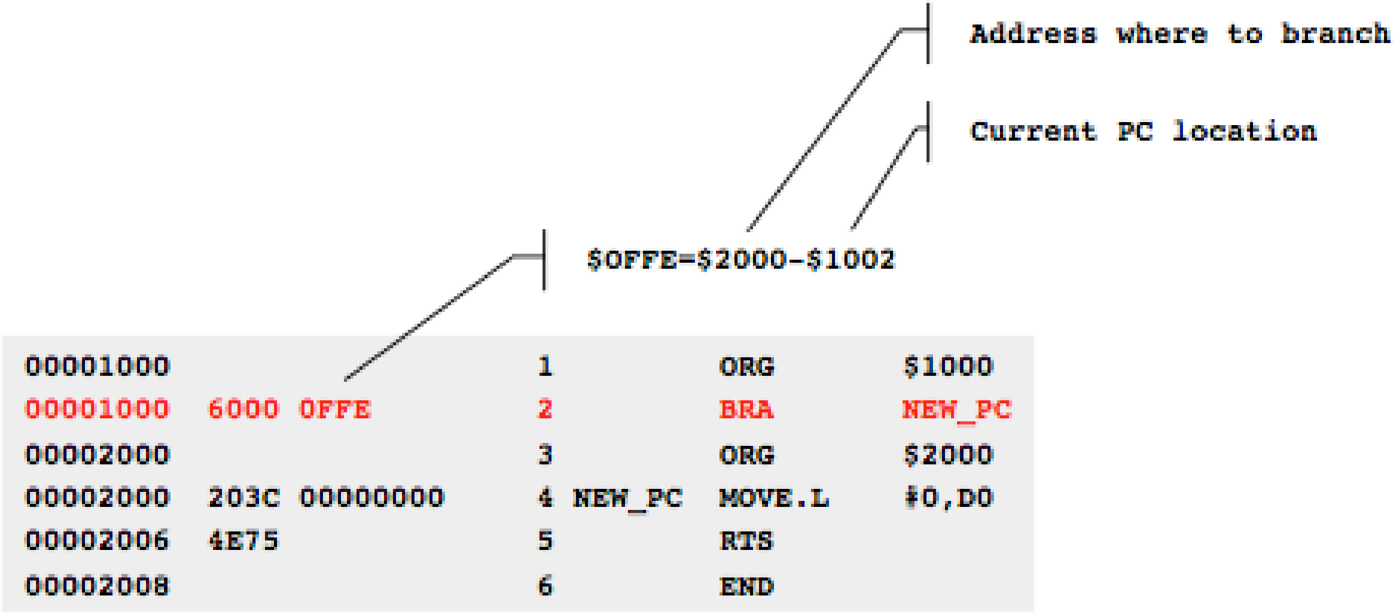
	0110	0000	1111	1100
$	6	0	F	C

Branch Offset

- Relative address is the address difference from current instruction to the instruction it branches to.
- Two versions:
  - short branch - 8 bit displacement d_8
  - long branch - 16 bit displacement d_{16}
- d_8 or d_{16} is in 2's complement.
  - d_8 allows branching from -128 to 126
  - d_{16} allows branching from -32768 to 32766
- Displacements are computed by assembler.
  - Dependent on the size of the jump
  - For forward references, assembler normally choose long branches.
  - Short branches can be forced by using BRA.S mnemonic

A Long Branch

- When the branch target cannot be reached using an 8-bit displacement, the long format is used.
- Machine code of short branch:
  - `0110 0000 PPPP PPPP`
- Machine co
  - `0110 C`
    `PPPP P`



JMP Instruction

- The JMP (for jump) instruction allows us to modify the PC in more powerful ways.
- JMP allows you to set the PC to the value of an address register and also to set it directly to a constant value.
- As an example, let's say that we wanted to jump to the location stored in A0. We can do that with:

```
4ED0        JMP     (A0)
```

- JMP loads the effective address of its operand into the PC.
- Let's look at the machine code:

```
0100 1110 11 010 000    ===> 0100 1110 1101 0000
                           $   4    E    D    0
```

- Bits 5:0 = 010 000 means address register indirect. Fairly straightforward. Note that address register indirect with displacement (and index) also work.

JMP Instruction

- Let's also assemble an example with absolute addressing.

```
4EF8 1000        JMP       $1000
```

```
0100 1110 11 111 000    ===> 0100 1110 1111 1000
                            $   4    E    F    8
                            $   1    0    0    0
```

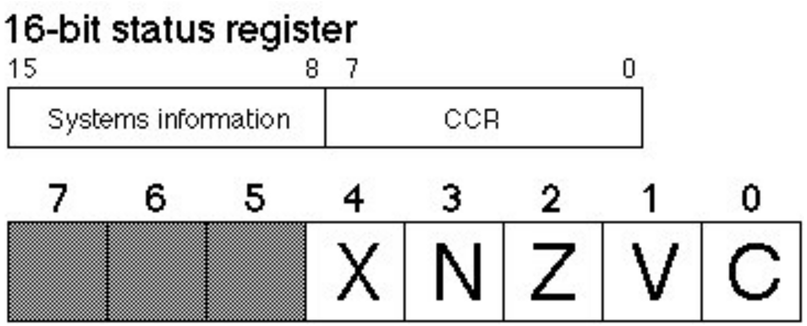
- Bits 5:0 = 111 000 means absolute short or `(xxx).W`. Again, pretty simple.
- The assembler gave absolute short because I've specified an address that was only 16 bits.
- Using absolute long or `(xxx).L` works fine too.

Why Conditional Instructions?

- While it is useful to be able to jump around in your code using BRA and JMP, they certainly don't solve all problems.
- In order to write any real program, you need to be able to branch conditionally based on the current state of the program.
- The "conditions" are stored in the Conditions Codes Register (CCR), so we will review it.
- Conditional Branch instructions examine bits in CCR and chose between two courses of action.
- CCR bits are either:
  - Updated after certain instruction have been executed, or
  - Explicitly updated (bit test, compare, or test instructions)

Review of CCR

- System Byte
  - Only modifiable is supervisor mode
  - Details in later modules
- User Byte: CCR
  - For user-level programs
  - Behavior depends on instruction



Bit	Meaning
C	Set if a carry or borrow is generated. Cleared otherwise.
V	Set if a signed overflow occurs. Cleared otherwise.
Z	Set if the result is zero. Cleared otherwise.
N	Set if the result is negative. Cleared otherwise.
X	Retains the carry bit for multi-precision arithmetic

Instructions That Modify CCR

- We have seen that most arithmetic/logic instructions modify CCR to report on the results of the ALU operation.
- Examples of how other instructions affect the condition codes.
  - MOVE: N/Z get set based on the result of the MOVE; V/C are always 0.
  - CLR: N/V/C always 0; Z always 1.
  - MOVEA: No affect on condition codes.
- One interesting instruction to look at is CMP.
  - CMP sets condition codes just like SUB, except that it doesn't actually store the result of the subtraction. Having the condition codes set allows us to compare the relative sizes of the two operands. As we will see in just a second when we look at Bcc, this is quite useful for conditional branching.
- Other instructions related to CMP are TST and BTST (later…)

Compare Instructions

- All compare instructions subtract the source operand, usually the contents of one register (or memory location) from the contents of the destination operand, usually another register (or memory location) in order to set the CCR (except the X-bit). The results of the subtraction are discarded.
- CMP or another compare instruction is usually followed immediately by a conditional branch (e.g., BEQ branch on zero, BNE branch on zero, BGT branch if greater than, BLT branch if less than, etc).

Instruction	Source Operand	Destination Operand
CMP	Any	Must be data register
CMPA	Any	Must be address register
CMPI	An immediate value	Any except address register
CMPM	Autoincrement	Autoincrement

Conditional Branch Instructions

- Identified by the mnemonic B_{cc} where "cc" represents the condition to be checked.
- General form:

```
Bcc    Address_Label
```

- If the condition is true, then control will branch to "Address_Label".
- No effect on condition codes

Conditional Branch on Single Flags

Mnemonic	Instruction	Flags
BCC	Branch on carry clear, branch on higher or same	C = 0
BCS	Branch on carry set, branch on lower	C = 1
BVC	Branch on overflow clear	V = 0
BVS	Branch on overflow set	V = 1
BNE	Branch on not equal	Z = 0
BEQ	Branch on equal	Z = 1
BPL	Branch on plus	N = 0
BMI	Branch on minus	N = 1

Understanding Branch Instructions

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

```
        CMP     D0,D1    ; when does Z=1?
        BEQ     SKIP     ; when D3 and D4 are equal!
        (something)
SKIP    (something)
```

- You can also think of B_{cc} as comparing the *result* of the last operation to zero:
  - **BNE** (branch when not equal) will be taken when **Z=0**

```
        MOVE    #5,D0
        MOVE    #1,D1
LOOP    ADD     D1,D1
        SUB     #1,D0    ; when does Z=0?
        BNE     LOOP     ; as long as D0 is not zero
```

Conditional Branches after Signed Arithmetic

Mnemonic	Instruction	Branch Taken If
BGE	Branch on greater or equal	(N = 1 and V = 1) or (N = 0 and V = 0)
BGT	Branch on greater than	(N = 1 and V = 1 and Z = 0) or (N = 0 and V = 0 and Z = 0)
BLE	Branch on less or equal	Z = 1 or (N = 1 and V = 0) or (N = 0 and V = 1)
BLT	Branch on less than	(N = 1 and V = 0) or (N = 0 and V = 1)

Conditional Branches after Unsigned Arithmetic

Mnemonic	Instruction	Branch Taken If
BHS	Branch on higher or same	C = 0
BLO	Branch on lower	C = 1
BHI	Branch on higher	C = 0 and Z = 0
BLS	Branch on lower or same	C = 1 and Z = 1

Structured Programming

- We can use B_{cc} to emulate the more structured flow control techniques present in languages like C
  - *if-then*
  - *if-then-else*
  - *while*
  - *do-while*
  - *for*

if-then

- Probably the simplest example is if. An example should suffice, as this is a straightforward concept:

```
...
if (n == 1) {
    m = 3;
}
...
```

```
          ...
          CMP     #1,N
          BNE     NotEq
          MOVE    #3,M
@NotEq    ...
```

- The most efficient way to code this is to **skip** the code block {…} if the **condition is not true**
- Remember: test for the opposite of the if condition

if-then-else

- The if-then-else construct has an alternative statement that is executed when the condition is false.

```
...
if (n == 1) {
    m = 3;
} else {
    m = 2;
}
...
```

```
        ...
        CMP      #1,N
        BNE      NotEq
        MOVE     #3,M
        BRA      Done
NotEq   MOVE.L   #2, M
Done    ...
```

- If the test in the if statement is more complex, a few more instruction might be needed.

while

- "while" isn't a whole lot more difficult than "if".

```
...
while (m > n) {
    n++;
}
...
```

```
            ...
            MOVE    M,D0
Top         CMP     N,D0
            BLE     Exit
            ADDQ    #1,N
            BRA     Top
Exit        ...
```

- One interesting note here: we did need to move M into D0. CMP can't compare two memory locations directly.
- This is an example of a "pre-test" loop. The condition is tested before going into loop.

.

do-while

- "do while" is a looping structure that doesn't compute the test before entering the loop. It runs the loop once and then computes the test.

```
...
do {
    n++;
} while (m > n);
...
```

```
        ...
        MOVE    M,D0
Top     ADD     #1,N
        CMP     N,D0
        BGT     Top
        ...
```

- Notice that the code produced by the "do while" is shorter (and faster) than the "while" loop. However, you don't get something for nothing. Often times, you really do want to do the test at the beginning of the loop. .

for

- Just to finish up all the looping structures present in C, we might as well show the “for” loop, although it’s really nothing new:

```
...
n = 0;
for (m = 1; m <= 10; m++) {
    n += m;
}
...
```

```
            ...
            CLR     N
            MOVE    #1,M
Top         CMP     #10,M
            BGT     Exit
            MOVE    M,D0
            ADD     D0,N
            ADD     #1,M
            BRA     Top
Exit        ...
```

Fixed loops

- If all you want is to repeat a loop 5 times (or any fixed count), don't use the "for" loop. In assembly it's more efficient to use a down counter.

```
...
n = 0;
for (m = 0; m <= 4; m++) {
    n += m;
}
...
```

(crossed out)

```
        ...
        CLR     N
        CLR     M
Top     CMP     #4,M
        BGE     Exit
        MOVE    M,D0
        ADD     D0,N
        ADD     #1,M
        BRA     Top
Exit    ...
```

```
        ...
        CLR     N
        MOVE    #5,D1
Top     MOVE    M,D0
        ADD     D0,N
        SUB     #1,M
        BNE     Top
Exit    ...
```

- Tip: make sure you count down to 0, and use a register for the counter!

Ex 1: Character Translation

ASCII Code

<table>
<tr><th></th><th>000</th><th>001</th><th>010</th><th>011</th><th>100</th><th>101</th><th>110</th><th>111</th></tr>
<tr><td>0000</td><td>NUL</td><td>DLC</td><td>SP</td><td>0</td><td>@</td><td>P</td><td>.</td><td>p</td></tr>
<tr><td>0001</td><td>SOH</td><td>DC1</td><td>!</td><td>1</td><td>A</td><td>Q</td><td>a</td><td>q</td></tr>
<tr><td>0010</td><td>STX</td><td>DC2</td><td>"</td><td>2</td><td>B</td><td>R</td><td>b</td><td>r</td></tr>
<tr><td>0011</td><td>ETX</td><td>DC3</td><td>#</td><td>3</td><td>C</td><td>S</td><td>c</td><td>s</td></tr>
<tr><td>0100</td><td>EOT</td><td>DC4</td><td>$</td><td>4</td><td>D</td><td>T</td><td>d</td><td>t</td></tr>
<tr><td>0101</td><td>ENQ</td><td>NAK</td><td>%</td><td>5</td><td>E</td><td>U</td><td>e</td><td>u</td></tr>
<tr><td>0110</td><td>ACK</td><td>SYN</td><td>&</td><td>6</td><td>F</td><td>V</td><td>f</td><td>v</td></tr>
<tr><td>0111</td><td>BEL</td><td>ETB</td><td>'</td><td>7</td><td>G</td><td>W</td><td>g</td><td>w</td></tr>
<tr><td>1000</td><td>BS</td><td>CAN</td><td>(</td><td>8</td><td>H</td><td>X</td><td>h</td><td>x</td></tr>
<tr><td>1001</td><td>HT</td><td>EM</td><td>)</td><td>9</td><td>I</td><td>Y</td><td>i</td><td>y</td></tr>
<tr><td>1010</td><td>LF</td><td>SUB</td><td>*</td><td>:</td><td>J</td><td>Z</td><td>j</td><td>z</td></tr>
<tr><td>1011</td><td>VT</td><td>ESC</td><td>+</td><td>;</td><td>K</td><td>[</td><td>k</td><td>{</td></tr>
<tr><td>1100</td><td>FF</td><td>FS</td><td>,</td><td><</td><td>L</td><td>\</td><td>l</td><td>|</td></tr>
<tr><td>1101</td><td>CR</td><td>GS</td><td>-</td><td>=</td><td>M</td><td>]</td><td>m</td><td>}</td></tr>
<tr><td>1110</td><td>SO</td><td>RS</td><td>.</td><td>></td><td>N</td><td>^</td><td>n</td><td>~</td></tr>
<tr><td>1111</td><td>SI</td><td>US</td><td>/</td><td>?</td><td>O</td><td>_</td><td>o</td><td>DEL</td></tr>
</table>

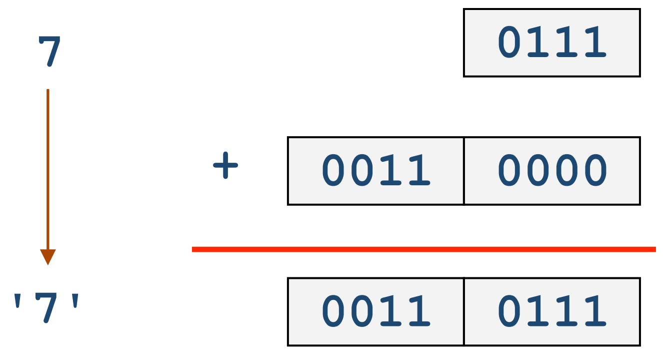
- How is a hex digit printed as a character?
- Algorithm:

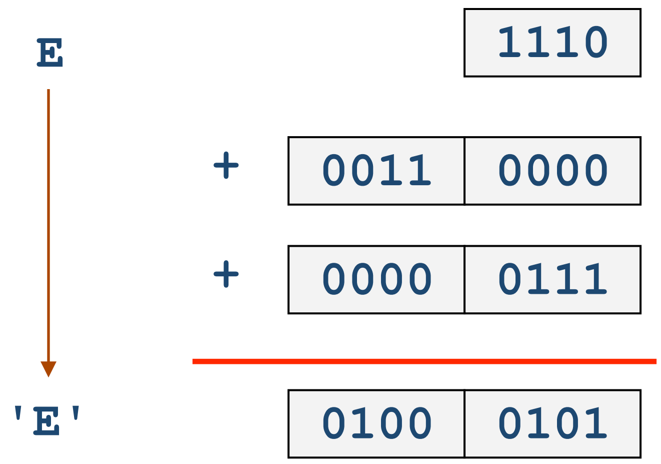
```
Char_Code = Hex_Val + 0x30;
if (Char_Code > 0x39) {
  Char_Code = Char_Code + 7
}
```

- Try verifying for ‘7’ and ‘E’

Convert a Hex. to ASCII

```
        MOVE.B    Hex_Val,D0
        ADDI.B    #$30,D0
        CMPI.B    #$39,D0
        BLS.S     EXIT
        ADDQ.B    #$07,D0
EXIT    MOVE.B    D0,Char_Code
```





Ex 2: Sum Using A Loop

- Perform the sum 1 + 2 + 3 + … + 10 by using a loop, i.e.

```
total = 0;
for (counter = 1; counter <= 10; counter++)
    total = total + counter;
```

```
        ORG     $1000
        CLR     D1              Set the total initially to 0
        MOVE.B  #1,D0           Initialize the counter to 1
Next    ADD.B   D0,D1           Add the counter to the total
        ADD.B   #1,D0           Increment the counter
        CMP.B   #11,D0          Check if loop is done
        BNE     Next            Go back for another round if not done
        STOP    #$2700          Stop execution
        END     $1000
```

Ex 3a: Bit Counting

- This version of the program uses bit operations

```
* D0 contains the byte of data whose bits we want to count
* D1 contains a bit counter which will range from 0 to 8
* D2 contains a loop counter which counts down from 8 to 0
        ORG     $1000
        MOVE.B  DATA,D0   Get the data
        CLR     D1        Clear bit counter
        MOVE    #7,D2     Set loop counter to 7
Next    BTST    D2,D0     Test the bit specified by D1
        BEQ     Zero      If the bit is 0, skip
        ADD     #1,D1     Else, increment bit counter
Zero    SUB.B   #1,D2     Decrement loop counter
        BCC     Next      Check another bit
        MOVE.B  D1,BITCT  Save bit count
        STOP    #$2700
DATA    DC.B    %10101111
BITCT   DS.B    1
        END     $1000
```

Ex 3b: Bit Counting

- This version of the program uses rotate operations

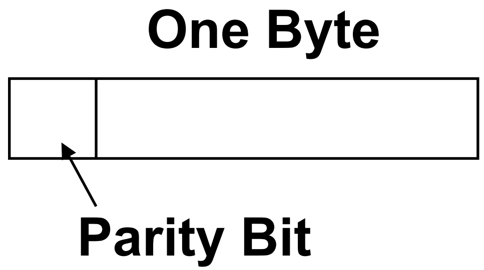
```
* D0 contains the byte of data whose bits we want to count
* D1 contains a bit counter which will range from 0 to 8
        ORG     $1000
        MOVE.B  DATA,D0   Get the data
        CLR     D1        Clear bit counter
Next    LSL.B   #1,D0     Shift whole byte left
        ADC     #0,D1     Add carry to bit counter
        TST.B   D0        If data is zero, we're done
        BNE     Next      Check another bit
        MOVE.B  D1,BITCT  Save bit count
        STOP    #$2700
DATA    DC.B    %10101111
BITCT   DS.B    1
        END     $1000
```

Ex 3b: Bit Counting

- This version of the program uses rotate operations

```
* D0 contains the byte of data whose bits we want to count
* D1 contains a bit counter which will range from 0 to 8
        ORG     $1000
        MOVE.B  DATA,D0   Get the data
        CLR     D1        Clear bit counter
Next    LSL.B   #1,D0     Shift whole byte left
        ADC     #0,D1     Add carry to bit counter
        TST.B   D0        If data is zero, we're done
        BNE     Next      Check another bit
        MOVE.B  D1,BITCT  Save bit count
        STOP    #$2700
DATA    DC.B    %10101111
BITCT   DS.B    1
        END     $1000
```

Ex 4: Setting Parity Bit of A Byte



- This version of the program uses bit operations

```
* D0 contains the byte of data whose parity bit is to be set
* D1 contains a counter which will range from 6 to 0
        ORG     $1000
        MOVE    #6,D1     Set the counter to  6
        BCLR    #7,D0     Clear the parity bit to start
Next    BTST    D1,D0     Test the bit specified by D1
        BEQ     Zero      If the bit is 1 then toggle parity bit
        BCHG    #7,D0     toggle the parity bit
Zero    SUB.B    #1,D1    Decrement the counter
        BCC     Next      Check another bit
        STOP    #$2700
        END     $1000
```

Ex 4a: Another Way to Calculate the Even Parity Bit

- If the byte is P0110100, then P=1 to make the number of bit 1 in D0 even.

```
        CLR.B   D1
        ANDI.B  #%01111111,D0
        MOVE.B  #7,D2
Next    ROR.B   #1,D0       ; [C] <- LSB of D0
        BCC     Zero
        ADDQ.B  #1,D1
Zero    SUB.B   #1,D2
        BNE     Next
        ROR.B   #1,D0
        LSR.B   #1,D1       ;Move LSB of D1 to C
        BCC     Exit
        ORI.B   #%10000000,D0
Exit    ...
```

Ex 5: Greatest Common Divisor

- Greatest Common Divisor (GCD) is the biggest number that can divide both inputs.
- Example: The GCD of 15 and 24 is 3 because both numbers can be divided evenly by 3.
- Many ways to compute but the one shown is Euclid's algorithm.

```
/* m >= n > 0 */
while( m > 0 )
  if( n > m ) {
    t = m; m = n; n = t;
  } /* swap */
  m -= n;
}
return n;
```

```
START   ORG     $1000
        MOVE    M,D0
        MOVE    N,D1
LOOP    TST     D0
        BEQ     DONE
        CMP     D1,D0
        BGT     SKIP

        EXG     D0,D1
SKIP    SUB     D1,D0
        BRA     LOOP
DONE    MOVE    D1,GCD

        STOP    #$2700
M       DC.W    24
N       DC.W    15
GCD     DS.W    1
        END     START
```

Ex 6: Leap Year Calculation

- Rules:
  - Years divisible by four are leap years, unless...
  - Years also divisible by 100 are not leap years, except...
  - Years divisible by 400 are leap years.

In-class Exercise…

```
if (year mod 4 != 0)
  {use 28 for days in February}
else if (year mod 400 == 0)
  {use 29 for days in February}
else if (year mod 100 == 0)
  {use 28 for days in February}
else
  {use 29 for days in February}
```

Pitfalls

- Example:

```
MOVE.B    #$C0,D0
CMP.B     #25,D0
BGE NEXT
```

```
MOVE.B    #$C0,D0
CMP.B     #25,D0
BHS NEXT
```

  – Will the branch be taken?

- Example:

```
MOVE.B   #$40,D0
MOVE.B   #$60,D1
ADD.B       D0,D1          [D1]=$A0
BMI      MINUS
```

  – Will the branch be taken?

UTM

Advanced Uses of JMP

- JMP can use many addressing modes.

```
CASE Test of
        CASE 1:  Action1
        CASE 2:  Action2
        CASE 3:  Action3
END:
```

```
IF TEST=0 THEN
        Action1
ELSE IF TEST=1 THEN
        Action2
ELSE IF TEST=2 THEN
        Action3
```

```
        CLR.L   D0
        LEA     JMPTAB,A0
        MOVE.B  TEST,D0
        ASL.L   #2,D0           [D0] <- [D0]*4
        MOVEA.L (A0,D0),A0
        JMP     (A0)
        …
JMPTAB  DC.L    Action1
        DC.L    Action2
        DC.L    Action3
        …
Action1 …       Code1
Action2 …       Code2
Action3 …       Code3
```

Advanced Uses of JMP

```
                                    *
                                    * Program Switch.x68
                                    *

00004000                                    ORG       $4000
00004000  00004018          TAB             DC.L      ACT1
00004004  00004020                          DC.L      ACT2

00004008  4280                              CLR.L     D0
0000400A  41F84000                          LEA       TAB,A0
0000400E  20700000                          MOVEA.L   (A0,D0),A0
00004012  4ED0                              JMP       (A0)
00004014  4E722700          EXIT            STOP      #$2700

00004018  223C00008888      ACT1            MOVE.L    #$8888,D1
0000401E  60F4                              BRA       EXIT
00004020  223C00001111      ACT2            MOVE.L    #$1111,D1
00004026  60EC                              BRA       EXIT

          00004000                          END       $4000
```