Branching

6 M68000 (4) end

(well, call, Ret not yet)

M68000 instructions - continued (after Addr. modes)

- So far, we have seen M68000 instructions for:

  (1) moving (copying) data, swapping, exchange.
  (2) arithmetic (+, −, *, /, negate)
  (3) logic (AND, OR, EOR, NOT)

With this arsenal of instructions (and addr. modes), we can do all assignment statements:

<variable> := <expression>

eg: A = B + C;
A[i] := B[i] + C[i]; etc.

- Now we turn our attention to other programming constructs and their implementation in assembler programming:

  if - statement
  while statement
  for - statement

  subroutine call & return.
  Recursion.

- Consider the ~~static~~ different constructs:

```
if (x ≤ 10)          while (x ≥ 10)          for (x = 0; x ≤ 10; x++)
  { .... }             { ...                   { ...
else
  { .... }             }                       }
```

All constructs calls for:

- Comparing different values and decide on some action to perform.

Therefore, all computer must have some instruction that allows it to compare 2 values.

→ One single (simple) ~~assembler~~ assembler construct is used to mimic the behaviour or all flow of controls. → the conditional branch (or jump).

Conditional branching in assembler is perform in 2 parts:

(1) Set up conditions
(2) branch on given conditions.

} discussed next.

Compare & conditional Branch

(1) Compare instruction:

CMP :

Compares a value in Dn against another value

Syntax: CMP[.s] <ea>, Dn

Effect: Set condition flags (N, Z, V, C) according to the comparison between Dn and <ea>

(Note: value in Dn & <ea> are unaffected).

eg:

CMP.L #0, D0

Compares long operand in D0 against the value 0.

- The use of CMP instruction is always in conjunction with a <u>conditional branch</u> instruction.

- The conditional branch instruction:

  There are a total of 14 conditional branch instructions, all have a mnemonic code of the form:

  Bcc <label>

  Where cc = the condition code (a mnemonic encoding for the condition)

  <label> = a symbolic name for the address location to branch to when the condition is satisfied.

  Effect of the conditional branch instruction:

  (1) if the condition specified by Bcc is satisfied, execution will be transfered to memory location given by <label> (load PC with the value <label>)

  (2) Otherwise (condition not satisfied), execution continues with the instruction following the conditional branch instruction.

- <label> must lie within -2^{15} & $(2^{15}-1)$ bytes from the current program location.
  - If <label> is outside this range, use: Jcc <label>

Bacon Lettuce & Tomato ....

Example:

BLT = branch less than
One of the 14 conditional branch instructions

How to use it:

Compares D0 against 0

```
CMP.L  #0, D0
BLT    LABEL1
```

(test D0 < 0).

instructions ~~executed when~~ skipped over when D0 < 0, but executed when D0 ≥ 0 (D0 not less than 0).

LABEL1: instruction executed when D0 < 0

Conclusion:

~~BLT~~

The construct:

```
CMP.L  #0, D0     ~~sets~~ compare flags
BLT    LABEL1
```

will make the branch to LABEL1 if

$$D0 < 0$$

(Other branches are similar).

- Other conditional branch instructions: (most common. less common at end!).

BLT = branch less than (signed)

BLE = branch less than or equal (signed)

BGT = branch greater than (signed)

BGE = branch greater than or equal (signed)

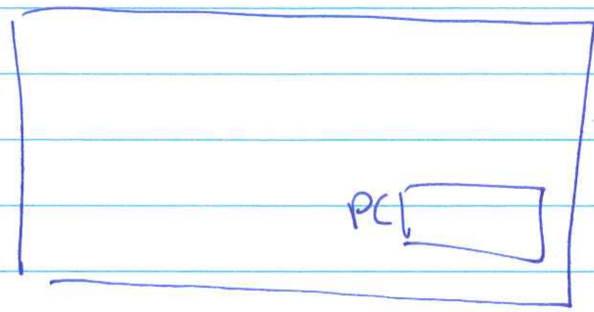
BEQ = branch equal

BNE = branch not equal.

- Unconditional branch (always branch)

BRA

Q: Knowing the CPU structure:



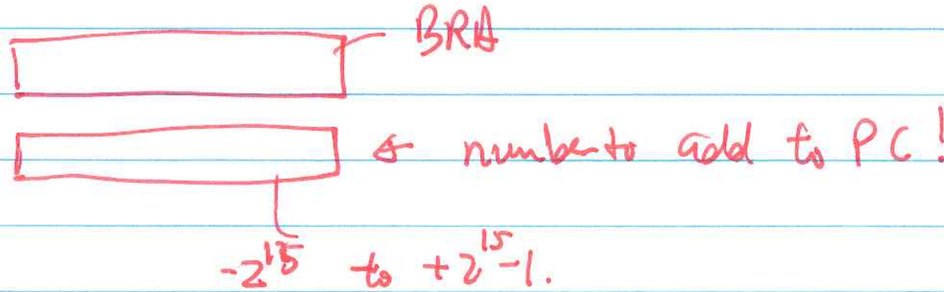
and knowing the effect of the BRA instruction:

BRA <LABEL> eg BRA 2000.

~~how do you think that BRA LABEL~~;

what actions does the CPU take/perform when it executes the BRA <label> instruction?
(BRA 2000)

BRA 2000 is encoded as:



Assembler constructs for If-statements

- There are 2 types of If-statement:
    - If - then
    - If - then - Else

- If-then:

```
if (condition)
  { statement1;
    statement2;
       :
  }
```

"ThenPart"

How is it executed?

(1) Evaluate condition.

(2) If true, execute statements.

(3) If false, skip.

The assembler construct to "mimic" the if-then statement behavior is as follows:

"Write assembler code to evaluate condition"

"Write branch ~~condition~~ instruction that skips over all instructions for the statements when condition is <u>FALSE</u>"

"Write assembler code to do all statements in the Then Part".

Example 1:

C's construct:

if.s

```
int x;

if (x < 0)
    x = -x;
```

Assembler construct:

"Test x against Ø"
"Branch to label if x ≧ Ø"

"negate x"

Label:

Code:

```
    move.l   x, DØ    } test x against Ø
    cmp.l    #Ø, DØ   }

    BGE      Label

    neg.l    X

Label:
```

Example: swap a and b if a < b.

```
if (a < b)
  { "swap values in variables a & b" }.
```

In C:
```
if (a<b)
  { help=a;
    a = b;
    b = help;
  }
```

Assembler construct:

"Test a against b"
"Branch over ThenPart if $a \geq b$" (to label)

```
help=a;      swap a & b.
a=b;
b=help;
```

label:

Code:

```
move.l  a, D0          D0 = a
cmp.l   b, D0

bge     label          if (a >= b)

move.l  b, a           (I already have a
move.l  D0, b           saved in D0 !!!)

Label:
```

Alternatively

"Test b against a"
"branch over ThenPart if b < a" (to label)

"Swap a & b"

Label:

Code:

```
    move.l  b, D0          D0 = b
    cmp.l   a, D0

    blt     label          if b < a, jump

    move.l  a, b
    move.l  D0, a

label:
```

- If-then-else construct:

```
if (condition)
  { statement1
    statement2
       :
  }                    "Then Part"
else
  { statement1
    statement2
       :
  }                    "Else Part"
```

How is it executed?

(1) Evaluate condition

(2) If true → execute Then part and skip over else part

(3) If false → Skip over Then Part and execute else part.

The assembler construct to "mimic" the if-then-else behavior is :

"write assembler code to evaluate the condition"

"Write branch instruction that take you to the else part if condition is false".

"write assembler codes to do all statements in the ThenPart"

"write branch instruction that take you over the else part"

ElseLabel: "write assembler code to do all statements in the ElsePart"

Done:

Example 2:

```
int a, b, max

if (a ≥ b)
    max = a;
else
    max = b;
```

if-else.s

Assembler construct:

"Test a against b"

"branch to ElseLabel if a < b"

"max = a"
"branch over else part to done"

ElseLabel: "max = b"

done:

```
        move.l   a, DØ
        cmp.l    b, DØ

        blt      ElseLabel

        move.l   a, max
        bra      Done

ElseLabel:  move.l   b, max

Done:       next instruction...
```

Alternatively:

"Test b against a"

"branch to ElseLabel if b > a"

"max = a"
"branch over Else part to Done"

ElseLabel: "max = b"

Done :

Code:

```
          move.l  b, D0
          cmp.l   a, D0

          bgt     ElseLabel

          move.l  a, max
          bra     Done

ElseLabel: move.l b, max

Done:      nextinstruction...
```

Example 3: compound condition

Write a prog. segment that:
"increase number x if it is divisible by 2 or 3, otherwise increase it by 2".

In C:

```
if (x%2 == 0 || x%3 == 0)
    x = x+1;
else
    x = x+2;
```

if-or.S

```
if (x<=a || x>=b)
    x = x+1
else
    x = x-1
```

The assembler program works like this:

"Compute x%2"
"Test against 0"

"Branch to ThenPart if x%2 == 0"

"Compute x%3"
"Test against 0"

"Branch to ElsePart if x%3 ≠ 0"

[Then part (x = x+1)
branch to done

ElseLabel: [Else Part (X = x+2)]

Done :

Code:

```
        move.l   X, D0        ]
        divs     #2, D0       ]  Compute X / 2
        swap     D0           ]
        cmp.w    #0, D0       ]  Test against 0
        beq      ThenPart

        move.l   X, D0
        divs     #3, D0
        swap     D0
        cmp.w    #0, D0
        bne      elsepart

ThenPart:  addq.l   #1, X
           bra      Done

ElsePart:  addq.l   #2, X

Done :
```

Example 4: Compound condition

"Increase x by 1 if $1 \le x \le 10$ otherwise decrease x by 1"

In C:
```
if (x >= 1 && x <= 10)
    x = x + 1;
else
    x = x - 1;
```

if-and.s

a b

if (a <= x && x <= b)
x = x + 1;
else x = x - 1;

Assembler construct:

"Test x against 1"
"Branch to elsepart if $x < 1$"

"Test x against 10"
"Branch to elsepart if $x > 10$"

[Then part: x = x + 1
"Branch over else part"

Elselabel: [Else part: x = x - 1

Done :

code:

```
          move.l   x, dØ
          cmp.l    #1, DØ

          blt      ElseLabel

          cmp.l    #10, DØ
          bgt      ElseLabel

          addq.l   #1, X

          bra      Done

ElseLabel: subq.l  #1, X

Done:      next instruction...
```