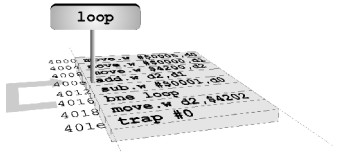
Labels

- Calculating each of the 2's complement displacements in a typical program is very **tiresome** and **error-prone**.
  - **-> The assembler should work out the offset**
  - **The Assenbler must therefore be told where you want to branch/jump to:**
    - -> Use **Lables**

A Label is a Symbolic Representation of an Address or Value.

```
4000  move.w #$0005,d0
4004  move.w #$0000,d1
4008  move.w $4200,d2
400e  add.w d2,d1
4012  sub.w #$0001,d0
4016  bne $f8
4018  move.w d2,$4202
401e  rts
```

loop = $400e



New Code

```
        move.w#5,d0             * count = 5
        move.w#0,d1             * total = 0
        move.w$2000,d2          * Load value
loop                            * do
        add.w       d2,d1       *
        sub.w       #1,d0       *
                                *
        bne         loop        * while count != 0
        move.wd1,$2002          * store result
        trap        #0          * end
```


More Labels

- We are free to choose the name of each label
  - e.g. **LOOP, PETER…**
- In theory there is no limit to the number of labels used.

Label Format Rules

- Must appear in the first character position of the line
- First character must be a letter (A-Z, a-z) or a period (“.”)
- Remaining characters must be letters (A-Z, a-z), digits (0-9), dollar signs (“$”), periods (“.”) or underscores (“_”).
- Symbols will be converted to uppercase before processing -> **LOOP** = **loop**
- But only the first 8 characters are used
  - -> **loopthree** = **loopthre**

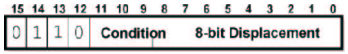
The Assembler and Labels

The assembler determines the value of the **loop** symbol before determining the machine code for the **bne loop** instruction.

- Branch Range
  - Branch range = - 128 . . . + 127 bytes
  - Often need larger displacements
    - -> Use **long version** of branch instruction that uses a 16-bit displacement -> range = - 32768 . . . + 32767 bytes

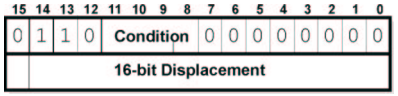
```
beq.s           10    * short form
beq.l           10    * long form
```


8-bit Branch Instruction Format



- Condition = 4-bit code represention the branch condition
  - 0100 -> BCC (Carry Clear) **C**
  - 0111 -> BEQ (Equal) **Z**
  - 1011 -> BMI **N**
  - etc . . .
- Displacement = 8-bit 2’s complement

If displacement is zero, then the **long form** is assumed



- 16-bit version of the branch instruction occupies 2 words of program memory
- 16-bit displacement is sign-extended to 32-bits before adding it to the PC.

16-bit Displacement

Displacement is still calculated using address of branch instruction + 2 bytes.

Loops

- Loops take two forms:
  - **WHILE:**
    - **Body executed 0 + times**
  - **DO-WHILE:**
    - **Body executed 1 + times**
    - Some-times known as repeat-until loop

More Loops

- The 68332 does not provide loop instructions/constructs
  - -> We must implement loops using flow control
- Good structure leads to readable bug-free code
  - We create assembly templates for writing loops

WHILE Loop General Form

Initialise
WHILE condition **DO**
body of while loop
update condition
END WHILE

Template:

```
      Initialise
while
      branch on opposite condition to endwhile
      body of while loop
      update condition
      branch always to while
endwhile
      rest of program
```

Example: Multiply 2 Numbers (in $2000 and $2002)

Pseudo-code

```
value=($2000);
count=($2002);
total=0;
while(count != 0)
{
      total=total+value;
      count=count-1;
}
($2004)=total;
```


Assembly Language Implementation

```
        Initialise
while
        branch on opposite condition to endwhile
        body of while loop
        update condition
        branch always to while
endwhile
        rest of program
```

```
        move.w   #$0,d1       * total=0
        move.w   $2000,d2     * load value
        move.w   $2002,d0     * load count
while   beq      endwhile     * while count!=0
        add.w    d2,d1        *   total+=value
        sub.w    #1,d0        *   count--
        bra      while
endwhile                      * endwhile
        move.w   d1,$2004     * store result
        trap     #0
```