Flow Control

- Flow control:
  - Altering the normal instruction execution sequence by explicit modification of the PC.
- So far, our programs have been **sequential**
- Example: Multiply by repeated addition

```
move.b d0,d1        * copy x into d1
add.b  d0,d1        * d1 holds x+x=2x
add.b  d0,d1        * d1 holds 2x+x=3x
add.b  d0,d1        * d1 holds 3x+x=4x
```


Problem:

- To multiply a number by 1000 we need a large program.
  - -> Need to specify that an instruction or a sequence of instructions should be executed many times:

```
Do 5 times
    add.b d0,d1
```

- **Solution:**
  - The **P**rogram **C**ounter (**PC**).
  - To alter the normal flow of a program we need instructions that can change the value stored in the PC:
  - We need a **PC modifying instruction**.

Classification of **PC** Modifying Instructions

- PC modifying instructions
  - Absolute/Relative
  - Unconditional/Conditional
- Absolute
  - Instruction supplies the address of the next instruction to execute
    - -> The operand value is loaded into the PC.
    - Example: **jmp**

```
1000    jmp     $1008
1006    move    d0,d1
1008    trap    #0
```

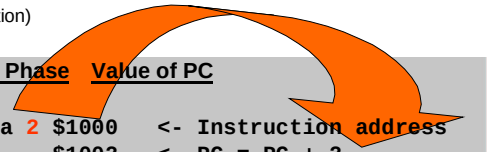

Relative

- The operand is a signed offset from the current value of the PC
  - -> Offset is added to the current PC to determine the address of the next instruction.
- Example: **bra** (branch always)

```
1000    bra     2
1002    move    d0,d1
1004    trap    #0
```

Be Careful !

- The offset or displacement (8 bit) is added to the contents of the PC after the **bra** instrction has been fetched and decoded.
  - -> PC has already been incremented by 2 bytes (to skip the bra instruction)



Execution Phase	Value of PC	
Fetch bra 2	$1000	<- Instruction address
Decode	$1002	<- PC = PC + 2
Execute	$1004	<- PC = PC + displacement
Fetch next inst.	$1004	

How do we implement backwards branches?

-> Use **negative** displacement

```
1000      move      d0,d1
1002      bra       -4          * = $fc
1004      trap      #0
```

- What does this program do?
  - It never stops.
  - This is known as an infinite loop

Adding a Negative Displacement

Earlier we added the negative displacement -4 (= $fc) to the PC to branch backwards.

```
PC before bra:      $0000 4004
Displacement:              $fc+
                    $0000 4100
```

- This is not the correct answer. Why?
  - **When adding or subtracting 2's complement numbers you must ensure that the numbers have the same modulus.**
- The displacement must be converted to a 32-bit 2's complement number.

Sign Extension

8-bit	16-bit	32-bit	
01000000	0000000001000000	00000000000000000000000001000000	+64
11111100	1111111111111100	11111111111111111111111111111100	-4

The MSB is repeated into the extra bits to the left.

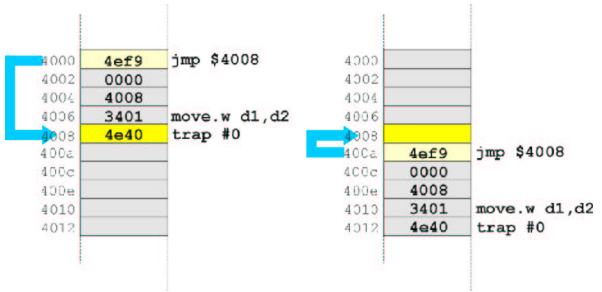
```
PC before bra:      $0000 4004
Displacement:       $ffff fffc+
                    $0000 4000
```

Sign extension is done automatically by the CPU during the execution phase of the bra instruction.

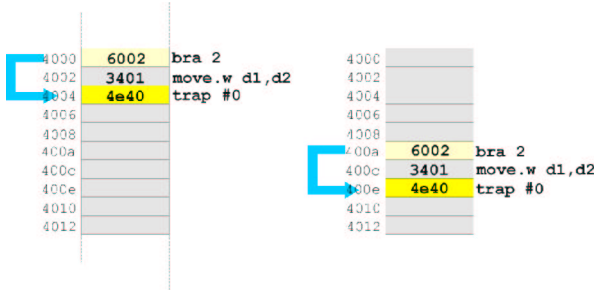
Relative Branches vs. Absolute Jumps

- Usually, **relative branches** are more desireable than **absolute jumps**.
  - Code is more compact:
    - Displacement (8-bits) vs. (32-bits) address operands.
  - Code is relocatable

Relocating Code with a **jmp** Instruction



Relocating Code with a **bra** Instruction



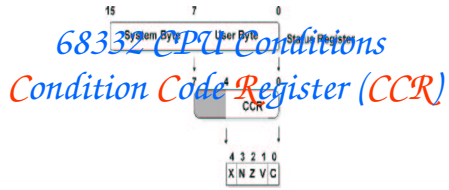
Unconditional and Conditional

- Unconditional
  - Branch/Jump is always taken
- Conditional
  - Usually we wish to execute certain instructions only when certain requirements are met.
  - Example:

```
If bank account is not empty then
      withdraw the requested amount
otherwise
      beep loudly and embarrassingly
end
```

Test a Condition

We need an instruction that can **test a condition** and **branch** if the condition was met.



68332 CPU Conditions

Condition Code Register (CCR)

We branch on the condition that a flag (XNZVC) is **set** or **cleared**.

Example

bne branches if Z=0

```
sub.w     d0,d1
bne       4
move.w    #1,d2
trap      #0
```

- If d0 and d1 contain the same word then d2 is loaded with the immediate value #1
  - -> The sequence of instructions execution depends on the input data

Pseudo-Code

To make coding more readable, it is very good practice to use pseudo-code as comments.

```
Org       $4000

sub.w     d0,d1       * compare d0,d1
bne       $04         * if equal then
move.w    #1,d2 *     d2 = 1;
trap      #0
```


All conditional branches are of form:

bxx -> where **xx** represents the condition code being checked.

Symbol	CCR flag	Branch Name
bcs	C	carry set
bcc	!C	carry clear
beq	Z	equal (to zero)
bne	!Z	not equal (to zero)
bmi	N	minus
bpl	!N	plus
bvs	V	overflow set
bvc	!V	overflow clear

Example

Write a program to multiply the word in $2000 by 5 and store the result in $2002.

- Express the solution in english:
  - Get the word at $2000
  - Add it to an accumulator 5 times
  - Store the result in $2002

Write out the Pseudo-Code

```
count = 5;
total = 0;
value = ($2000);
do {
        total = total + value;
        count = count - 1;
} while (count != 0);
($2000) = total;
```

loop counter (will loop 5 times)

accumulator

Looped Code

loop condition

Code between the braces is the code that will execute multiple times

Convert this to Assembly Language

Map *variables* to data-registers:

Count d0 Total d1 Value d2

```
move.w  #5,d0       * count=5
move.w  #0,d1       * total=0
move.w  $2000,d2    * Load value
                    * do
add.w   d2,d1       *    total=total+value
sub.w   #1,d0       *    Decrement Count
                    *
bne     -8          * while count!=0
move.w  d2,$2002    * Store Result
trap    #0          * End
```


Loop

```
move.w  #5,d0
move.w  #0,d1
move.w  $2000,d2

add.w   d2,d1
sub.w   #1,d0

bne     -8
move.w  d2,$2002
trap    #0
```

Initially

d0	d1	d2	Z
5	0	10	0
4	10	10	0
3	20	10	0
2	30	10	0
1	40	10	0
0	50	10	1

Loop Exits

Z *tested after* sub