EE 357 Unit 6

Compare and Branch Instructions

Branch Translation

Condition Codes (Flags)

- Processor performs tests on the result of each instruction
- 5 tests w/ results stored as 1=true/0=false in SR (Status Reg.)
  - X = eXtended Flag
    - Used for Extended Precision Arithmetic
    - Usually set the same as the C flag
  - N = Negative Flag
    - Tests if the result is negative
    - Just a copy of the MSB of result
  - Z = Zero Flag
    - Tests if the result is equal to 0
  - V = 2’s complement oVerflow Flag
    - Set if p+p=n or n+n=p
  - C = Carry Flag (Unsigned Overflow Flag)
    - Set if (Cout=1 and Add op.) or (Cout=0 and Sub. Op)

Instructions & Condition Codes

- Different instructions affect the condition codes differently
- Refer to instruction data sheets for instruction-specific effects
- General effects guidelines are as follows

	X	N	Z	V	C
MOVE	-	*	*	0	0
ADD/SUB/CMP	* or -	*	*	*	*
LSx/ASx	*	*	*	0	*
Logic	-	*	*	0	0
Branches/Jumps	-	-	-	-	-

* = Set according to traditional or instruction-specific definition
- = Unaffected (unchanged from previous value)
0 = Cleared to 0

Condition Code Examples

Instruction	D0	X	N	Z	V	C
MOVE.L #0x7fffffff,D0	7fffffff	-	0	0	0	0

- MOVE by definition leaves X unaffected and V = C = 0
- 0x7fffffff is NOT negative (N=0)
- 0x7fffffff is NOT equal to 0 (Z=0)

Condition Code Examples

Instruction	D0	X	N	Z	V	C
MOVE.L #0x7fffffff,D0	7fffffff	-	0	0	0	0
ADDI.L #0x80000001,D0	00000000	1	0	1	0	1

```
             7fffffff
           + 80000001
Result   =   00000000
```

- ADD instruc. & C_{out} = 1 so X = C = 1
- Result is NOT negative (N=0)
- Result is equal to zero (Z=1)
- P + N can't yield 2's comp. overflow

Condition Code Examples

Instruction	D0	X	N	Z	V	C
MOVE.L #0x7fffffff,D0	7fffffff	-	0	0	0	0
ADDI.L #0x80000001,D0	00000000	1	0	1	0	1
SUBI.L #0x80000000,D0	80000000	1	1	0	1	1

```
           00000000
           7fffffff
         +        1
         ----------
Result =   80000000
```

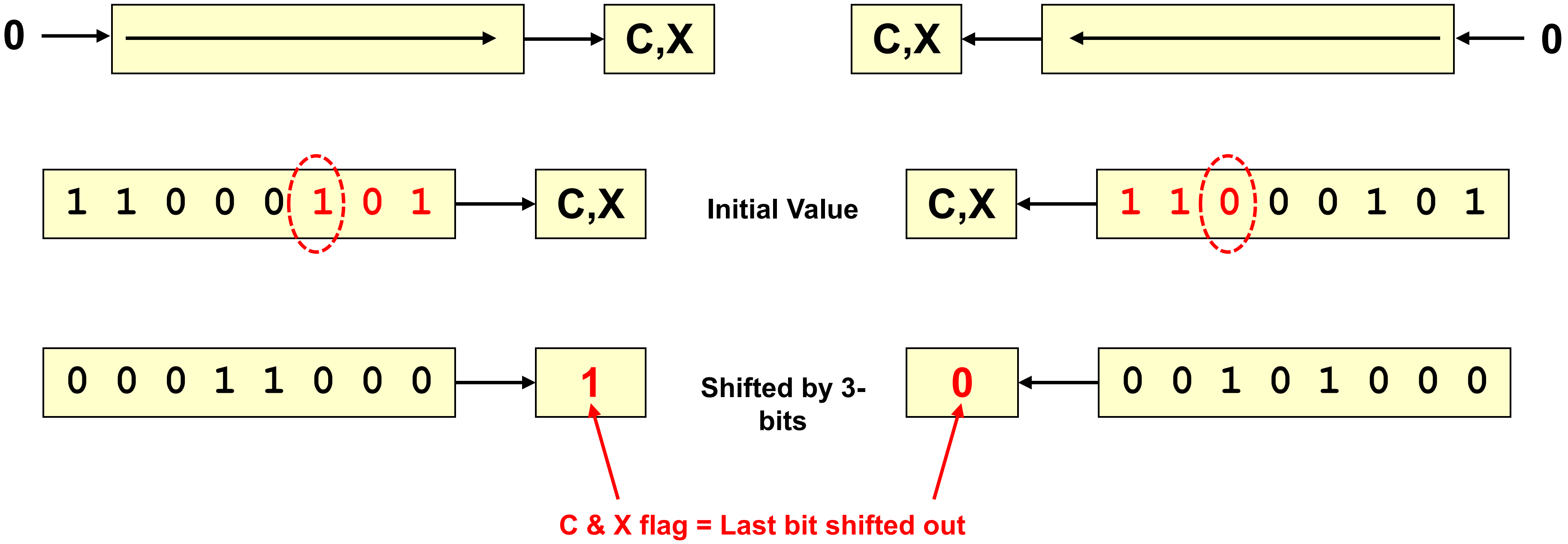
- SUB instruc. & C$_{out}$ = 0 so X = C = 1
- Result is negative (N=1)
- Result is NOT equal to zero (Z=1)
- P + P = N means 2's comp. overflow

LSx and Asx Instructions

- C *and* X flags are set with the last bit shifted out
- N and Z are set normally
- V = 0

Right Logical/Arithmetic Shift

Left Logical/Arithmetic Shift



0 → → C,X

C,X ← ← 0

1 1 0 0 0 1 0 1 → C,X

Initial Value

C,X ← 1 1 0 0 0 1 0 1

0 0 0 1 1 0 0 0 → 1

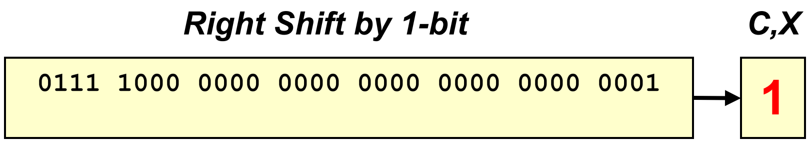
Shifted by 3-bits

0 ← 0 0 1 0 1 0 0 0

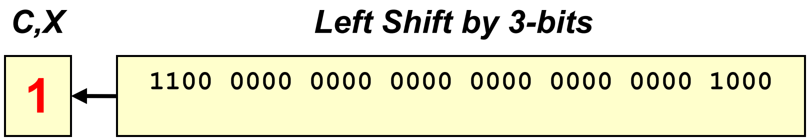
C & X flag = Last bit shifted out

LSL and LSR Instructions

	D2	X	N	Z	V	C
`MOVE.L #0xF0000003,D2`	F000 0003 1111 0000 0000 0000 0000 0000 0000 0011	-	1	0	0	0
`MOVE.L #3,D3`		-	0	0	0	0
`LSR.L #1,D2`	0111 1000 0000 0000 0000 0000 0000 0001	1	1	0	0	1
`LSL.L D3,D2`	1100 0000 0000 0000 0000 0000 0000 1000	1	0	0	0	1



D2 = 0x78000001



D2 = 0xC0000008

ASL and ASR Instructions

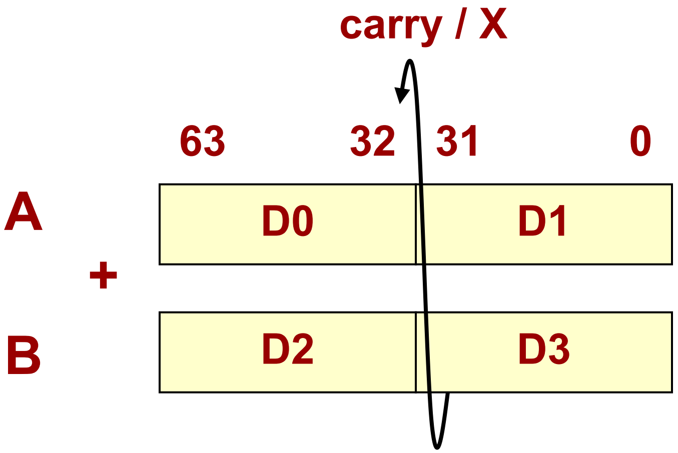
	D2	X	N	Z	V	C
MOVE.L #0x80000003,D2	8000 0003 1000 0000 0000 0000 0000 0000 0000 0011	-	1	0	0	0
MOVE.L #1,D3		-	0	0	0	0
ASR.L #5,D2	*Right Shift by 5-bits* 1111 1100 0000 0000 0000 0000 0000 0000 → *C,X* 0 **D2 = 0xFC000000**	0	1	0	0	0
ASL.L D3,D2	*Left Shift by 1-bit* *C,X* 1 ← 1111 1000 0000 0000 0000 0000 0000 0000 **D2 = 0xF8000000**	1	0	0	0	1

Extended Precision Arithmetic

- Used to add/subtract numbers greater than 32-bits
- We can only add/subtract 32-bit chunks with one instruction, but somehow need to communicate the carry or borrow between the chunks
- Carry from LS addition stored in X flag
- Use ADDX to add X flag in addition to the 2 operands

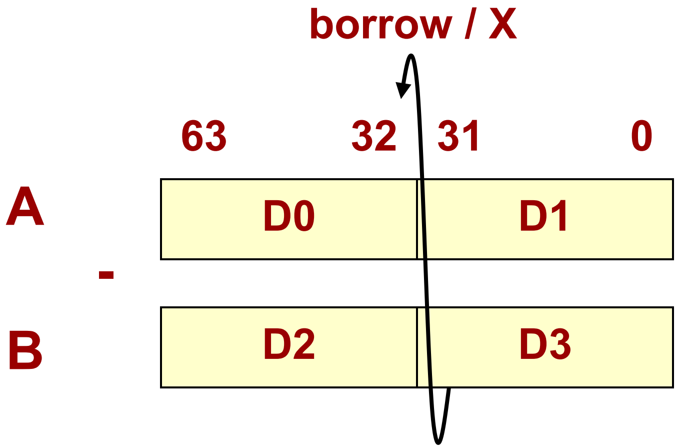
ADDX and SUBX

- ADDX.s src,dst
  - DST+SRC+X -> DST



```
// A = A + B
ADD.L   D3,D1   ;C,X set if overflow
ADDX.L D2,D0   ;Add D0+D2+X
```

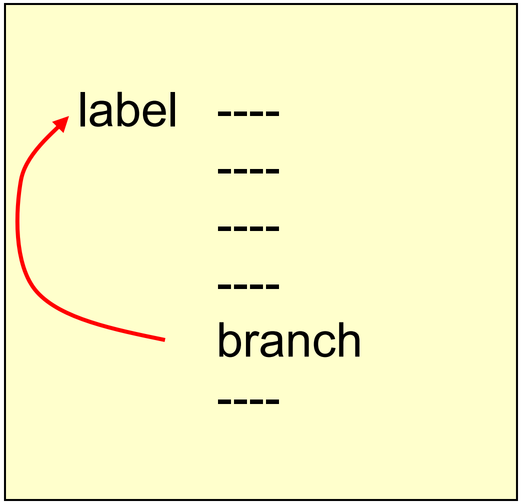
- SUBX.s src,dst
  - DST-SRC-X -> DST



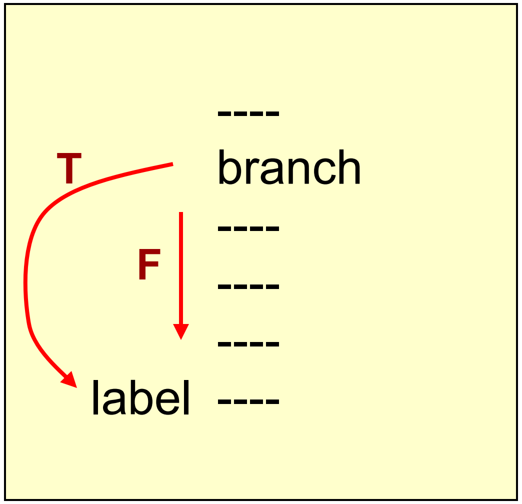
```
// A = A - B
SUB.L   D3,D1  ;C,X set if overflow
SUBX.L D2,D0  ;Sub D0–D2–X
```

Branch Instructions

- Operation: (PC) + disp. → PC
- Branches allow us to jump backward or forward in our code
  - Needed to implement loops, conditionals, etc.
  - 2 Kinds: Unconditional and Conditional



Unconditional Loops



Conditional Loops

Branch Instructions

- Unconditional Branches
  - Always branches to a new location in the code
  - Instruction: `BRA label`
- Conditional Branches
  - Branches only if a particular condition is true otherwise continues fetching instructions sequentially
  - Instruction: `Bcc label`
  - cc = {EQ, NE, LT, LE, GT, GE, LS, HI, etc.}
  - Condition determined by examining condition codes
  - Can be used with CMP and CMPI instruction
    - CMPI.L #1,D0
    - BEQ NEXT

Comparison

- Comparison of A with respect to B is performed by examining the result of A-B
  - if A=B, then A-B = 0
  - if A<B, then A-B = negative #
  - if A>B, then A-B = positive # but not 0
- Determining if the result is positive or negative requires
  - knowing what system is being used
    - signed or unsigned?
  - if overflow occurred
    - when overflow occurs N flag will be wrong =>
      Ex. if A = pos. & B = neg. we know A > B; however if we do A-B => pos – neg = pos + pos and there is overflow, we will get a negative result which would indicate A < B…WRONG!

Compare Instruction

- Format
  - CMP.L   <ea>,Dn
    - 2nd operand must be a data register
  - CMPI.L #imm,<ea>
- CMP condition code effects:
  - XNZVC = -****
- CMP instruction subtracts src2 – src1, sets the condition codes, but discards result
  - Note that to compare A w/ B the order of your operands should be CMP.L B,A

TeST Instruction

- Format
  - TST.s <ea>
- TST condition code effects:
  - XNZVC = -**00
- Can be used to check whether a number is negative/positive or zero/non-zero
- Essentially it is implicitly comparing the operand with 0

Conditional Branches

- After a CMP instruction subtracts A-B the Bcc instruction will check for the specified condition

Signed

Cond.	Test
EQ =	Z
NE ≠	Z’
LT <	N•V’ + N’ •V
LE ≤	N•V’ + N’ •V + Z
GT >	(N’•V’ + N•V) • Z’
GE ≥	(N’•V’ + N•V)

Unsigned

Cond.	Test
EQ =	Z
NE ≠	Z’
LS ≤	C + Z
HI >	C’ • Z’

Flag Check

Cond.	Test
MI	N
PL	N’
VS	V
VC	V’
CS	C
CC	C’

Branch Example

C Code

```
int A,B;
if A > B
   A = A + B
else
   A = 1
```

M68000
Assembly

```
          .data
A:         .long   ...
B:         .long   ...

           .text
         MOVE.L  A,D0
         CMP.L   B,D0
         BLE     ELSE
         ADD.L   B,D0
         BRA     NEXT
ELSE:    MOVE.L  #1,D0
NEXT:    MOVE.L  D0,A
         ----
```

Branches if A is less than or equal to B

Skips over the “else” portion

Branch Example

C Code

```
for(i=0;i < 10;i++)  (D0=i)
  j = j + i;          (D1=j)
```

M68000
Assembly

```
       .text
       CLR.L   D0
LOOP:  CMPI.L  #10,D0
       BGE     NEXT
       ADD.L   D0,D1
       ADDI.L  #1,D0
       BRA     LOOP
NEXT:  ----
```

Branches if i is not less than or equal to 10

Loops back to the comparison check

Another Branch Example

C Code

```
int dat[10];

for(i=0;i < 10;i++)  (D0=i)
   data[i] = 5;        (D1=j)
```

M68000
Assembly

```
           .data
dat:       .space   40

          .text
          MOVE.L   #10,D0
          MOVEA.L  #dat,A1
LOOP:     MOVE.L   #5,(A1)+
          SUBI.L   #1,D0
          BNE      LOOP
NEXT:     ----
```

Branches don't require a CMP instruction before it. They simply check the appropriate flags which can be set by any prior instruction

If no compare instruction prior to a conditional branch,
the branch condition is effectively
the comparison of the result of the previous instruction with 0

A Final Example

C Code

```
char A[] = "hello world";
char B[50];

// strcpy(B,A);
while(A[i] != 0){
   B[i] = A[i]; i++;
}
B[i] = 0;
```

M68000
Assembly

```
           .data
A:         .asciz   "hello world"
B:         .space   50

           .text
           MOVEA.L  #A,A0
           MOVEA.L  #B,A1
LOOP:      MOVE.B   (A0)+,(A1)+
           BNE      LOOP
NEXT:      ----
```

BNE looks at Z flag which will be set or cleared based on the value just moved to the B array

Branch Format

- Branch Instructions: (PC) + disp. → PC
- Format of short branch: Bcc.S

0110	Cond.	8-bit disp.

- Format of word branch: Bcc.W

0110	Cond.	0000 0000
16-bit disp.		

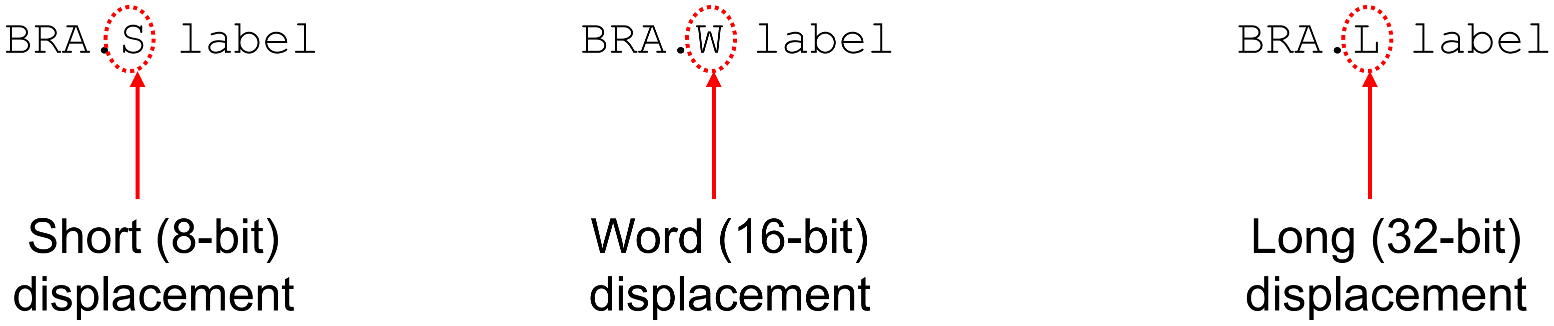
- Format of long branch: Bcc.L

0110	Cond.	1111 1111
MS- Word of 32-bit disp.		
LS- Word of 32-bit disp.		

- Displacement is the value that should be added to the PC so that it now points to the desired branch location
- Displacement is a signed value (pos. or neg.)

Specifying Displacement Size

- The programmer can force use of the different size displacements by using a size extension with the instruction
- By not providing a size specification, compiler/assembler will choose on own (recommended)



```
BRA.S label          BRA.W label          BRA.L label
```

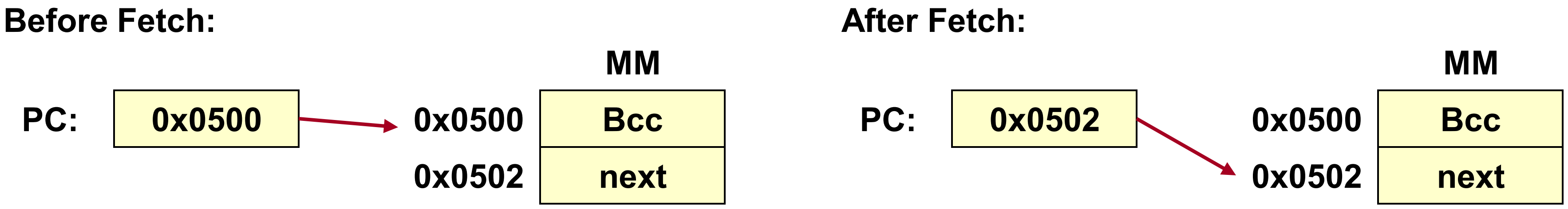
Short (8-bit) displacement

Word (16-bit) displacement

Long (32-bit) displacement

Review of Instruction Cycle

- Processor
  - Fetches instruction and increments PC by 2 to point at the next word
  - Decodes and Executes instruction
- Key: By the time the instruction starts execution, PC has been incremented by 2



Before Fetch:

After Fetch:

Branch Displacement

- To calculate displacement you must know where instructions are stored in memory

```
        .text
        MOVE.L 0x3000,D0
        CMP.L  0x3004,D0
        BLE.S  ELSE
        ADD.L  0x3004,D0
        BRA.S  NEXT
ELSE:   MOVE.W #1,D0
NEXT:   ----
        ----
```

Coldfire
Assembly

.text + 00	MOVE
.text + 02	3000
.text + 04	CMP
.text + 06	3004
.text + 08	BLE
.text + 0A	ADD
.text + 0C	3004
.text + 0E	BRA
.text + 10	MOVE
.text + 12	0001
.text + 14	----

1 word for each instruction plus additional extension words

Calculating Displacements

- Disp. = (Address of Target) – (Address of Branch + 2)
- Following slides will show displacement calculation for BLE

```
         .text
         MOVE.L 0x3000,D0
         CMP.L  0x3004,D0
         BLE.S  ELSE
         ADD.L  0x3004,D0
         BRA.S  NEXT
ELSE:    MOVE.W #1,D0
NEXT:    ----
         ----
```

.text + 00	MOVE
.text + 02	3000
.text + 04	CMP
.text + 06	3004
.text + 08	BLE
.text + 0A	ADD
.text + 0C	3004
.text + 0E	BRA
.text + 10	MOVE
.text + 12	0001
.text + 14	----

Calculating Displacements

- Disp. = (Address of Target) – (Address of Branch + 2)
- Disp. = .text + 0x10 – (.text + 0x08 + 2) = .text + 0x10 – (.text + 0A) = **0x0006**

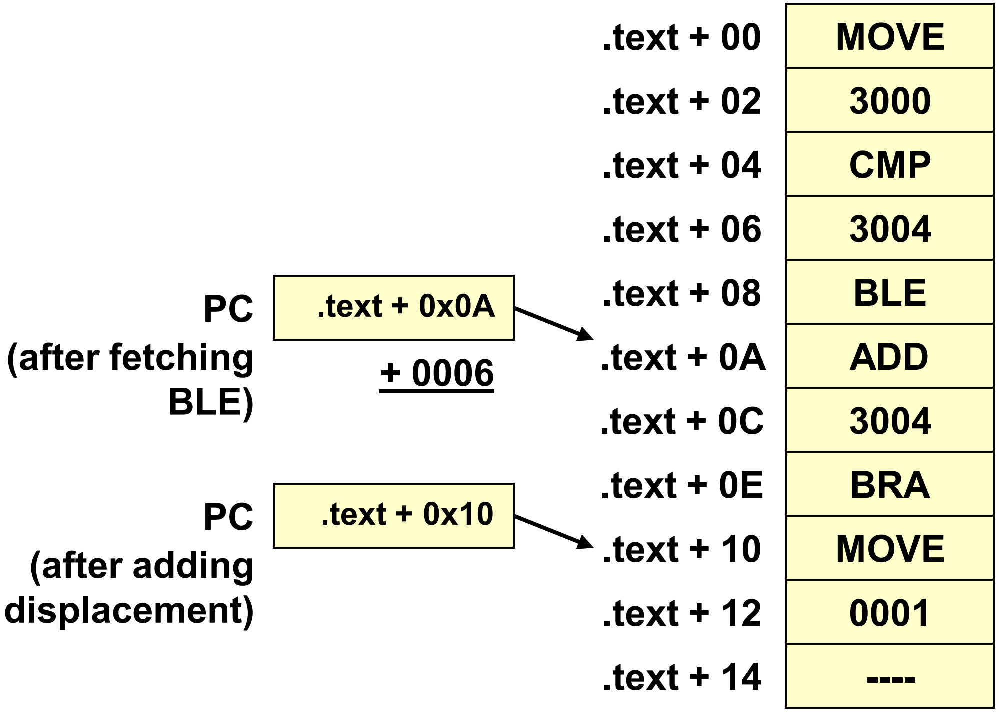
```
          .text + 00
          MOVE.L 0x3000,D0
          CMP.L  0x3004,D0
          BLE.S  ELSE
          ADD.L  0x3004,D0
          BRA.S  NEXT
ELSE:     MOVE.W #1,D0
NEXT:     ----
          ----
```

.text + 00	MOVE
.text + 02	3000
.text + 04	CMP
.text + 06	3004
.text + 08	BLE
.text + 0A	ADD
.text + 0C	3004
.text + 0E	BRA
.text + 10	MOVE
.text + 12	0001
.text + 14	----

Calculating Displacements

- If the BLE does in fact branch, it will add the displacement (0x0006) to the PC (.text + 0x0A) and thus point to the MOVE instruction (.text + 0x10)

```
          .text + 00
          MOVE.L 0x3000,D0
          CMP.L  0x3004,D0
          BLE.S  ELSE
          ADD.L  0x3004,D0
          BRA.S  NEXT
ELSE:     MOVE.W #1,D0
NEXT:     ----
          ----
```



.text + 00	MOVE
.text + 02	3000
.text + 04	CMP
.text + 06	3004
.text + 08	BLE
.text + 0A	ADD
.text + 0C	3004
.text + 0E	BRA
.text + 10	MOVE
.text + 12	0001
.text + 14	----

Calculating Displacements

- With displacement of 0x0006 it can fit in the 8-bit displacement (.S) field
- 16-bit displacement is unneeded and thus the next instruction can be stored after the 1-word BLE

0110	1111 = LE	0000 0110
next instruction		

Another Example

- Disp. = (Address of Label) – (Address of Branch + 2)
- Disp. = .text + 0x04 – (.text + 0x14 + 2)
  = .text + 0x04 – (.text + 0x16) = **0xFFEE**

```
          .text + 00
          MOVE.L #0,D0
LOOP      CMP.L  0x3004,D0
          BEQ    ELSE
          ADD.L  #5,D1
          ADD.L  #1,D0
          BRA    LOOP
NEXT      ----
          ----
          ----
```

.text + 00	**MOVE**
.text + 02	**0000**
.text + 04	**CMP**
.text + 06	**3004**
.text + 08	**BEQ**
.text + 0A	**ADD**
.text + 0C	**0000**
.text + 0E	**0005**
.text + 10	**ADD**
.text + 12	**0001**
.text + 14	**BRA**
.text + 16	**----**

Calculating Displacements

- The displacement of 0xFFEE can fit in the 8-bit displacement field

FFEE → 1111 1111 1110 1110

- Can remove leading 0’s from positive numbers and leading 1’s from negative number (need to retain at least one copy of the sign bit…can’t remove all the 1’s)

~~1111 1111 111~~0 1110 → 10 1110

0110	**0000=BRA**	1110 1110
next instruction		

More Displacements

- If a displacement is found to be 0xFF00

FF00 → 1111 1111 0000 0000

- Must leave at least one copy of the MSB, and thus this requires 9-bits and will use the 16-bit displacement field

~~1111 111~~1 0000 0000 → 1 0000 0000

<table>
<tr><td>0110</td><td><b>0000=BRA</b></td><td>0000 0000</td></tr>
<tr><td colspan="3">1111 1111 0000 0000</td></tr>
</table>

More Displacements

- If a displacement is found to be 0x00FF

00FF → 0000 0000 1111 1111

- Must leave at least one copy of the MSB, and thus this requires 9-bits and will use the 16-bit displacement field

~~0000 000~~0 1111 1111 → 0 1111 1111

<table>
<tr><td>0110</td><td><b>0000=BRA</b></td><td>0000 0000</td></tr>
<tr><td colspan="3">0000 0000 1111 1111</td></tr>
</table>