# Control Flow

- The default operation of the processor is to execute a straight-line sequence of instructions read from successive memory locations
  - This results in two problems
    1. Programs run until they reach the “end” of memory
    2. Programs can’t accomplish anything “useful”

- Solution
  - Allow programs to occasionally depart from the usual sequence by transferring control (conditionally or unconditionally) to another part of the memory

# Control Flow and Branch Instructions

- Branch instructions modify the flow of control and cause the program to continue execution at the target address specified by the branch

- Two types:
  1. Unconditional branch
     - Like "goto" statement in C
       - `goto L3;`
     - always forces a jump to the instruction at the target address (often specified using a label)

  2. Conditional branch
     - Like "if-statement" in C
       - `if(condition is TRUE) goto L3;`
     - test condition and only branch to the target address (label) if condition is TRUE

# Branch Always (Unconditionally)

## BRA Branch Always

Syntax: `BRA <target address>`

| Name | Displacement | Machine Language | Operation Performed |
|---|---|---|---|
| BRA.S | 8-bit (XX) | 60XX | PC ← PC + displacement |
| BRA.L | 16-bit (XXXX) | 6000 XXXX | |

# How the Assembler Computes the Displacement

Displacement = Branch Destination – Program Counter

```
1   00002000                                ORG      $2000
2   00002000   601E                         BRA.S    AHEAD
3   00002002   4E71                          NOP
4   00002020                                ORG      $2020
5   00002020   4E71                 AHEAD   NOP
6   00002022   60FE                 HERE    BRA.S    HERE
7   00002024   4E71                          NOP
```

# Jump (Unconditionally)

**JMP** **Jump**

Syntax: `JMP <ea>`

Operation: PC ← destination address

| Dn | An | (An) | (An)+ | -(An) | d(An) | d(An,Xn) | ABS.W | ABS.L | Imm | d(PC) |
|---|---|---|---|---|---|---|---|---|---|---|
|  |  | ✓ |  |  | ✓ | ✓ | ✓ | ✓ |  | ✓ |

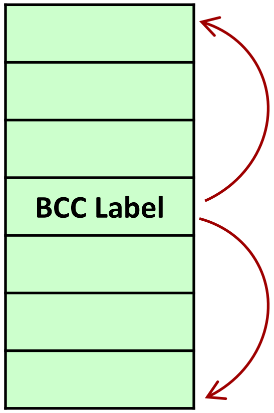
Slides are IP. Do not share, copy or post on website(s).

# Conditional Branch Instructions

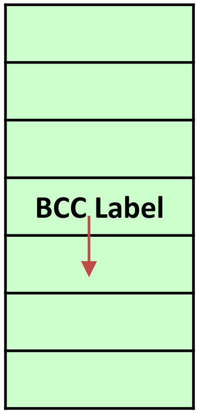
**Bcc Branch on condition cc**

Syntax:
```
Bcc <label>
Bcc <literal>
```

Operation: if (cc = TRUE)
PC ← PC + displacement



cc = TRUE



cc = FALSE

| Mnemonic | Flags |
|---|---|
| **BEQ** | Z=1 |
| **BNE** | Z=0 |
| **BMI** | N=1 |
| **BPL** | N=0 |
| **BCS** | C=1 |
| **BCC** | C=0 |
| **BVS** | V=1 |
| **BVC** | V=0 |
| **BGE** | Z=V |
| **BGT** | Z=0 and N=V |
| **BLE** | Z=1 or (N≠V) |
| **BLT** | N≠V |
| **BHI** | C=0 and Z=0 |
| **BLS** | C=1 and Z=1 |

# If-then Conditional Statement

- Consider the following C code

```
if (a == b) goto Same;
```

- Assume that the variables are bytes (i.e., chars) and **a** is contained in D1 and **b** is contained in D0

- Solution

```
sub.b d0,d1           ;Set or Clear Z flag
beq   Same;           ;goto Same if Z = 1
```

# Instructions for Comparing Two Values

## CMP Compare

Syntax: `CMP <ea>,<ea>`

Operation: CCR ← (destination – source)

| Instruction | Source Operand | Destination Operand |
|---|---|---|
| **CMP** | Any address mode | Must be a data register |
| **CMPA** | Any address mode | Must be an address register |
| **CMPI** | An immediate value | Any address mode except address register indirect and immediate |
| **CMPM** | Compares one memory location with another. The only address mode permitted is address register indirect with post-incrementing | |

# If-then Solution

- Consider the following C code

```
if (a == b) goto Same;
```

- Assume that the variables are bytes (i.e., chars) and **a** is contained in D1 and **b** is contained in D0

- Solution

```
cmp.b d0,d1       ;compare values, discard result
beq   Same        ;goto Same if Z = 1
```

# Conditional Branch Instructions Depending on a Single CCR Flag

| Mnemonic | Condition | Flags |
|---|---|---|
| **BEQ** | equal | Z=1 |
| **BNE** | not equal | Z=0 |
| **BMI** | negative | N=1 |
| **BPL** | positive or zero | N=0 |
| **BCS** | carry set | C=1 |
| **BCC** | carry clear | C=0 |
| **BVS** | overflow set | V=1 |
| **BVC** | overflow clear | V=0 |

Check to see if previous result is zero/non-zero

# Conditional Branch Instructions Depending on a Single CCR Flag

| Mnemonic | Condition | Flags |
|---|---|---|
| **BEQ** | equal | Z=1 |
| **BNE** | not equal | Z=0 |
| **BMI** | negative | N=1 |
| **BPL** | positive or zero | N=0 |
| **BCS** | carry set | C=1 |
| **BCC** | carry clear | C=0 |
| **BVS** | overflow set | V=1 |
| **BVC** | overflow clear | V=0 |

Only use to check value of most-significant bit

# Conditional Branch Instructions Depending on a Single CCR Flag

| Mnemonic | Condition | Flags |
|---|---|---|
| **BEQ** | equal | Z=1 |
| **BNE** | not equal | Z=0 |
| **BMI** | negative | N=1 |
| **BPL** | positive or zero | N=0 |
| **BCS** | carry set | C=1 |
| **BCC** | carry clear | C=0 |
| **BVS** | overflow set | V=1 |
| **BVC** | overflow clear | V=0 |

Check for unsigned Overflow or borrow

# Conditional Branch Instructions Depending on a Single CCR Flag

| Mnemonic | Condition | Flags |
|---|---|---|
| **BEQ** | equal | Z=1 |
| **BNE** | not equal | Z=0 |
| **BMI** | negative | N=1 |
| **BPL** | positive or zero | N=0 |
| **BCS** | carry set | C=1 |
| **BCC** | carry clear | C=0 |
| **BVS** | overflow set | V=1 |
| **BVC** | overflow clear | V=0 |

Check for signed overflow

# Be Careful when using BMI

- Consider the following code

```
add.b d0,d1
bmi   $9000   ; branch taken if N=1 (sum is negative)
```

| Before | After |
|---|---|
| D0 = $64_{10}$ | D0=$64_{10}$ |
| D1 = $96_{10}$ | D1 = $160_{10}$ |
| CCR:  X=0 N=0 Z=0 V=0 C=0 | CCR:  X=0 **N=1** Z=0 **V=1** C=0 |

# Conditional Branch Instructions for Signed Numbers

| mnemonic | cmp value1, value2 | flags |
|---|---|---|
| **BGE** | value2 ≥ value1 | N=V |
| **BGT** | value2 > value1 | Z=0 and N=V |
| **BLE** | value2≤ value1 | Z=1 or (N≠V) |
| **BLT** | value2 < value1 | N≠V |

- V-bit ensures that signed branches are correctly executed even when the compare operation produces overflow

# Including Signed Overflow into Branches

- Consider the following code

```
cmpi.b #-8,d0
bge    $9000  ; take branch if N=V (d0 >= -8)
```

| Before | After |
|---|---|
| D0 = $127_{10}$ | D0 = $127_{10}$ |
| Calculation: ($127_{10}$ - $(-8)_{10}$) = $135_{10}$ | |
| CCR:  X=0 N=0 Z=0 V=0 C=0 | CCR:  X=0 **N=1** Z=0 **V=1** C=0 |

# Conditional Branch Instructions for Unsigned Data

| mnemonic | cmp value1, value2 | flags |
|---|---|---|
| **BHS,BCC** | value2 ≥ value1 | C=0 |
| **BHI** | value2 > value1 | C=0 and Z=0 |
| **BLS** | value2≤ value1 | C=1 and Z=1 |
| **BLO,BCS** | value2 < value1 | C=1 |

# Signed and unsigned Branches

- Unsigned Comparison

```
cmp.b d0,d1
bhs   $9000
```

- Signed Comparison

```
cmp.b d0,d1
bge   $9000
```

| | Unsigned | Signed |
|---|---|---|
| D0<br>D1 | 00000010 (2)<br>11111111 (255) | 00000010 (2)<br>11111111 (-1) |
| Result | 11111101 (253) | 11111101 (-3) |
| CCR | X=0, N=1, Z=0, V=0, **C=0** | X=0, **N=1**, Z=0, **V=0**, C=0 |
| Branch | C=0 | N=V |

# Simple “if” Statement

## C code

```
int a,b;
   .
   .
   .
if (a == 1)
    b=3;
```

## Assembly Language

```
          org      $8000
          cmpi.l   #1,a        condition
          bne      exit
          move.l   #3,b        code
exit      move.b   #9,d0       exit
          trap #15

          org       $9000
a         ds.l     1
b         ds.l     1
```

# Simple “if-else” Statement

C code

```
int a,b;
.
.
if (a == 1)
    b=3;
else
    b=5;
```

Assembly Language

```
         org       $8000
         cmpi.l    #1,a       condition
         bne       else
         move.l    #3,b       code
         bra       exit
else     move.l    #5,b       code
exit     move.b    #9,d0      exit
         trap      #15
         org        $9000
a        ds.l       1
b        ds.l       1
```

# Complex Condition

C code

```
int a,b;
.
.
if(a>1 && a<10)
    b=3;
else
    b=5;
```

With ANDs and ORs C uses short-circuit evaluation, in which it stops evaluating the condition as soon as it finds that it must be true or false no matter what the rest of the evaluation would give.

# Complex Condition

## C code

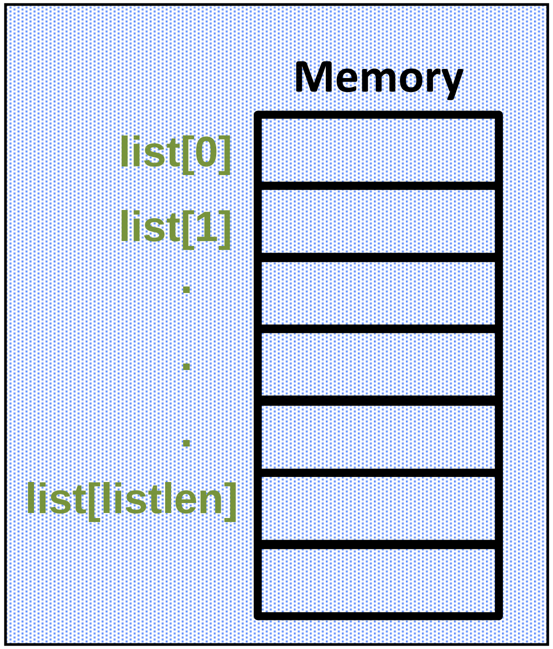
```
int a,b;
.
.
if(a>1 && a<10)
    b=3;
else
    b=5;
```

## Assembly Language

```
        org       $8000
        cmpi.l    #1,a        condition
        ble       else
        cmpi.l    #10,a       condition
        bge       else
        moveq     #3,b        code
        bra       exit
else    moveq     #5,b        code
exit    ...                   exit
        org       $9000
a       ds.l      1
b       ds.l      1
```

# Short-Circuit Evaluation

- Short circuit evaluation helps to
  - Improve code performance
  - Avoid runtime errors



- Consider the following code

```
index = 0;
while((index < listlen) && (list[index] != item))
  index++;
```

# Switch Statements

```
typedef enum {ADD, MULT, MINUS, DIV,
   MOD, BAD}  op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

**Implementation Options**

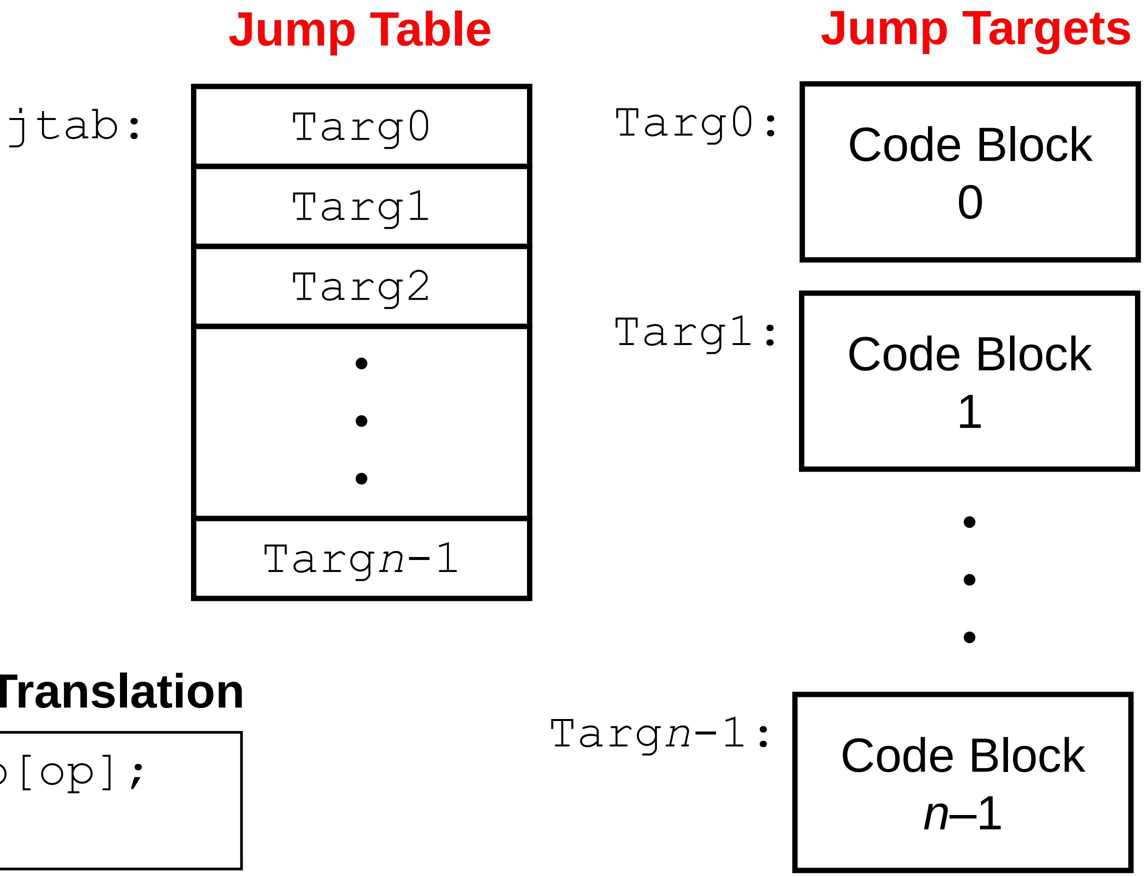
1. Series of conditional branches
   - Good if few cases
   - Slow if many
2. Jump Table
   - Lookup branch target
     - Use 68000 **jmp** instruction to unconditionally jump to address stored in a register
   - Avoids conditional branches
   - Possible when cases are small integer constants

- C compiler e.g., gcc
  - Picks one based on case structure

# Jump Table Structure

### Switch Form

```
switch(op) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```



### Jump Table

jtab:

| |
|---|
| Targ0 |
| Targ1 |
| Targ2 |
| •<br>•<br>• |
| Targ*n*-1 |

### Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

•
•
•

Targ*n*-1: Code Block *n*–1

### Approximation Translation

```
target = JTab[op];
goto *target;
```

# Switch Statement Example

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;
char unparse_symbol(op_type op){
  switch (op) {
    • • •
  }
}
```

**Enumerated Values**

| | |
|---|---|
| ADD | 0 |
| MULT | 1 |
| MINUS | 2 |
| DIV | 3 |
| MOD | 4 |
| BAD | 5 |

- Assume op = D0, JumpTable = A1

```
cmpi.l    #0,d0           ;if op < 0
bls       exit            ;goto exit
cmpi.l     #6,d0          ;if op >= 6
bhs       exit            ;goto exit
mulu      #4,d0           ;compute displacement
movea.l    (a1,d0.l),a2   ;a2 = JumpTable[op]
jmp       (a2)            ;jump to JumpTable[op]
```

# Jump Table

```
JumpTable
  .dc.l L0   ;Op = 0
  .dc.l L1   ;Op = 1
  .dc.l L2   ;Op = 2
  .dc.l L3   ;Op = 3
  .dc.l L4   ;Op = 4
  .dc.l L5   ;Op = 5
```

Targets

```
L0   move.b   #’+’,d7
     jmp      exit

L1   move.b   #’*’,d7
     jmp      exit

L2   move.b   #’-’,d7
     jmp      exit

L3   move.b   #’/’,d7
     jmp      exit

L4   move.b   #’%’,d7
     jmp      exit

L5   move.b   #’?’,d7

exit    ;end of switch
```

# Simple “while” Statement

## C code

```
int a,b;
.
.
while (a < b)
    a++;
```

## Assembly Language

```
        org       $8000
        move.l    a,d0
top     cmp.l     b,d0        condition
        bge       exit
        addq      #1,d0
        bra       top         code
exit    ...                   exit
        org        $9000
a       ds.l      1
b       ds.l      1
```

# “do-while” Statement

## C code

```
int a,b;
.
.
do {
        a++;
while (a < b);
```

## Assembly Language

```
          org       $8000
          move.l    a,d0
top       addq      #1,d0        code
          cmp.l     b,d0
          blt       top          condition
exit       . . .                 exit
          org        $9000
a         ds.l       1
b         ds.l       1
```

# While loop – again!

- Consider the following C code

```
while (a[i] == k)
      i = i + j;
```

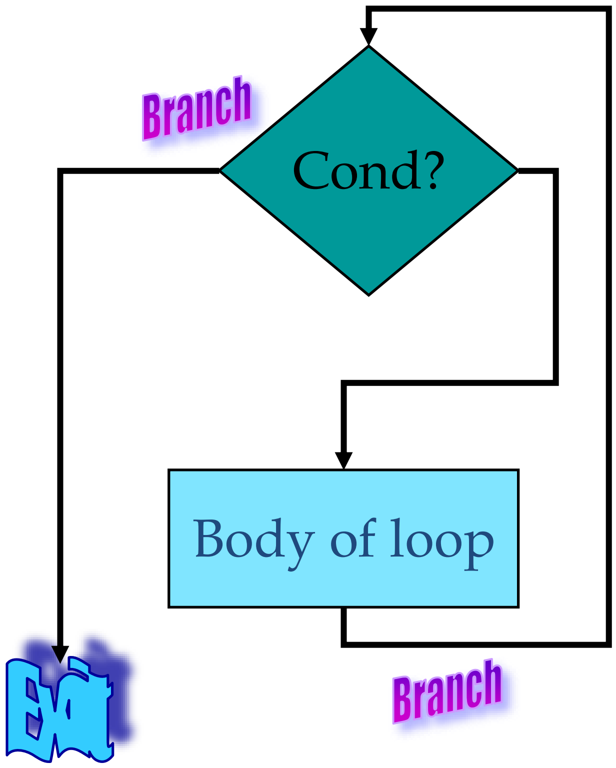
- Assume `i`(D0), `j`(D1), `k`(D2) and `a`(A0)

```
loop    cmp.b   (a0,d0.l),d2       ;a[i] == k?
        bne      exit               ;no
        add.l    d1,d0              ;i = i + j
        bra      loop               ;do it again
exit    ...
```

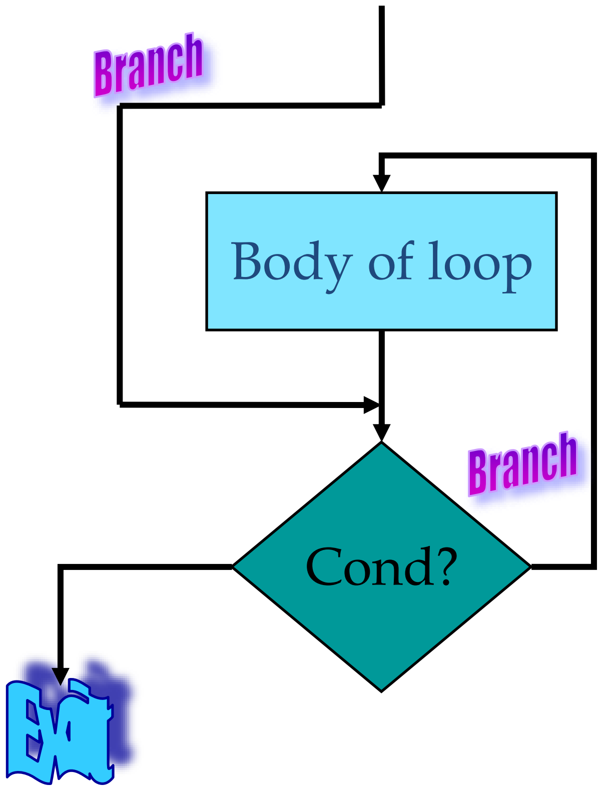
Slides are IP. Do not share, copy or
post on website(s).

# Improving While-Loop Efficiency

- Code uses two branches/iteration:



- Better structure:



# Improved Loop Solution

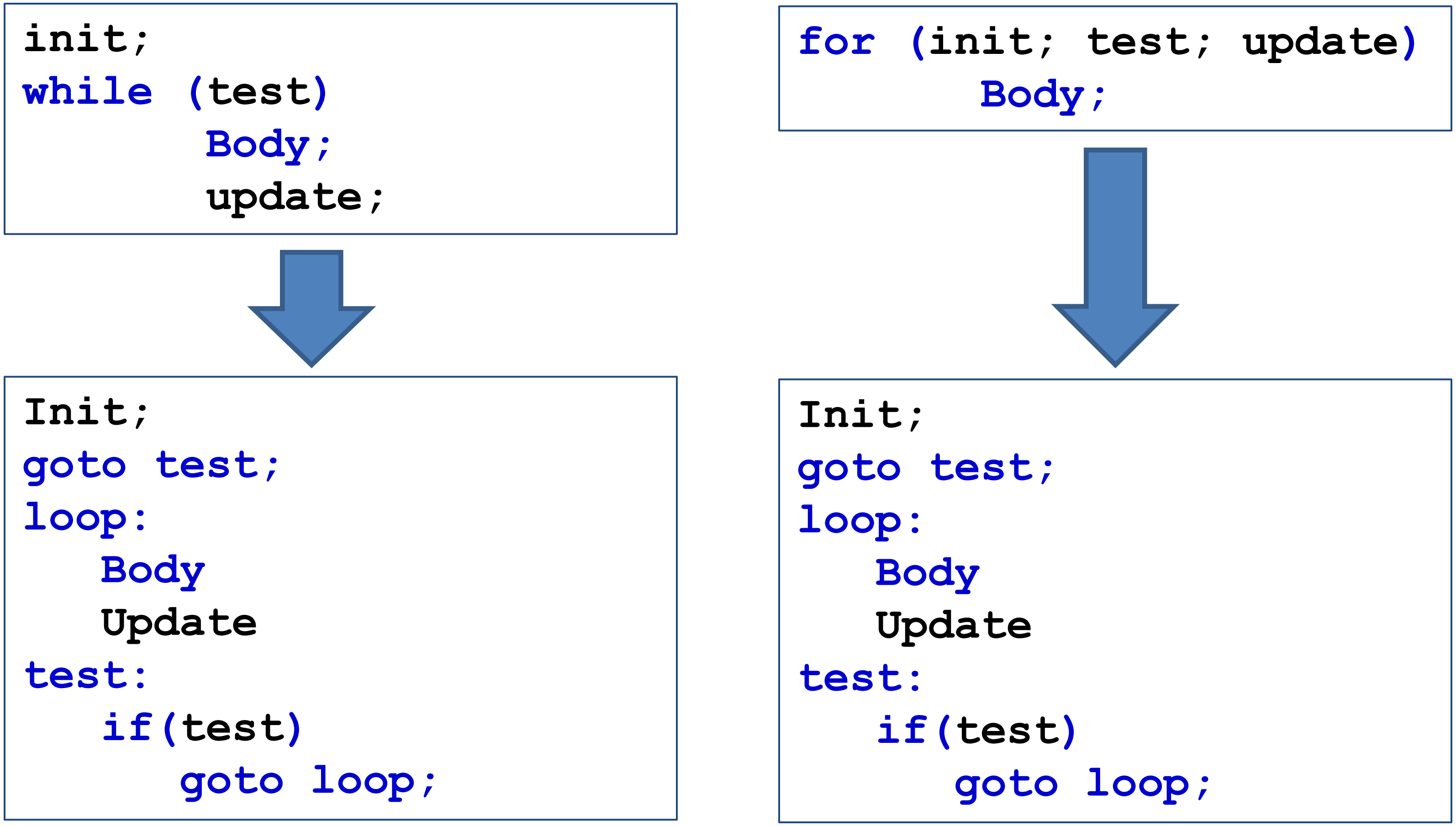
- Remove extra branch from loop body

```
        bra     cond
loop    add.l   d1,d0             ;i = i + j
cond    cmp.b   (a0,d0.l),D2      ;a[i] == k?
        beq     loop              ;do it again
exit
```

- Reduced loop from 4 to 3 instructions
  - Even small improvements are important if loop executes many times
- Question
  - How do you implement "for" loops?

# For Statement

- Similar to while loop



```
init;
while (test)
       Body;
       update;
```

```
for (init; test; update)
       Body;
```

```
Init;
goto test;
loop:
   Body
   Update
test:
   if(test)
      goto loop;
```

```
Init;
goto test;
loop:
   Body
   Update
test:
   if(test)
      goto loop;
```

# Assembly Language

## Original Loop

```
for (i=0; a[i]==k; i=i+j)
```

## Optimized Loop

```
i=0 ;
  goto test ;
loop:
  i=i+j;
test:
  if (a[i]==k) goto loop ;
```

- Assembly Language

```
        move.l  #0,d0
        bra     test
loop    add.l   d1,d0
test    cmp.b   (a0,d0.l),d2
        beq     loop
exit
```

# Summary

- Decision making is a two-step process
  - Compare two values (update flags in CCR)
  - Perform conditional branch (based on flags in CCR)
- Signed branches must be used with signed data, unsigned branches with unsigned data
- BMI and BPL should only be used to check MSB of result
- Signed branches are guaranteed never to fail as a result of the comparison
- C uses short-circuit evaluation to implement complex conditions
  - Improve performance
  - Avoid runtime errors
- Compiler decides how a switch statement is to be implemented
  - Series of if-statements versus jump table
- Do-loop is fastest loop, but while loop can be optimized
- No final difference between while-loop and for-loop