Branching

Textbook 5.1 - 5.8

Overview

A program consists of 4 basic types of statements

1. Simple Statements
2. Decisions
3. Loops
4. Invocation of sub-programs

Basic Idea of Branching

- Remember: Instruction Execution has three phases
  - Fetch, Decode, and Execution
  - In Fetch phase, the address of the next instruction in the PC
- During the Execution phase if an instruction change the PC address
  - the next instruction will be at the newly changed address
  - changing PC allows for change to flow-of-control
  - changing of PC called branching
- Most assembly language do **NOT** allow direct access to the PC

Types of Branching

- Assembly Languages provide indirect means by which to alter PC
    1. Unconditional Branches
        - Branch always
    2. Conditional Branches
        - Branch sometimes
            - Based on Status Register's (SR) Condition Code Register
            - Condition Code Register is the low order byte of the SR

Unconditional Branching

- There are two unconditional branching instructions:
  - jmp <label>
  - bra <label>
- They are rarely used alone
  - Combined with other instructions

Unconditional Branching

Example for **jmp**:

```
spin:       jmp     spin    ; infinite loop!
```

Example for **bra**:

```
            move.b  #'a',d0     ; output 'a' forever!
forever:jsr     write_char
            bra     forever
```

Unconditional Branching

```
            .
            .
            nop
            nop
            bra     next_code    ; jumping over data?
variable:   ds.w42              ; BAD STYLE!
next_code:  nop
            nop
            .
            .
```

Jump vs. Branch

- jmp =   PC address
- bra =   PC PC + displacement
- Discuss:
  - which is more “powerful”?
  - which is faster?

Condition Codes

The user byte of SR contains 5 condition codes (CC's) in bits 4-0:

X N Z V C

- data transfer instructions affect the N & Z bits
- arithmetic & logic instructions affect most/all CC bits
- program sequencing & control instructions affect nothing
- Note: the carry bit is also called the "borrow" bit. For subtraction, the C bit is set when a borrow into the most significant bit is required.

Condition Codes

For now ignore X as it follows C, usually!

When using the reference card the condition codes column shows how each bit is set. Each of the 5 CC bits may be:

- `left unaffected` `-`
- `left undefined` `U`
- `set always (= 1)` `1`
- `cleared always (= 0)` `0`
- `set/cleared conditionally` `*`

Note: The condition code table in the reference card shows how the * is calculated

Testing and Comparing

The 68000 provides instructions to set the condition codes:

```
tst.x   <ea>        tests a single value <ea> - 0:  XNZVC = -**00
cmp.x   <ea>,Dn         tests result of dst - src:      XNZVC = -****
```

Note that the 68000 also provides:

```
cmpa.x  <ea>,An         compare against address register
cmpi.x  #<data>,<ea>compare immediate
cmpm.x  <ea>,<ea>       compare memory
btst.x  <ea>,<ea>       test an individual bit
                        (notice only b or l for btst)
```

Conditional Branching

- The Bcc instruction – branch based on the condition codes (cc)
- While there is actually a bcc instruction it is only one of many.
- They all work by:
    - if the specified condition is true the branch is performed,
    - i.e. the PC is altered; if the specified condition is true
        - otherwise the specified condition is false
        - and no action is taken
            - i.e. sequential processing continues.

Conditional Branching

- Zero Branches - based on Z bit
  - beq equal
  - bne not equal

Conditional Branching

Unsigned Branches - based on unsigned comparison involves C, sometimes Z

- `bhi` high
- `blo (= bcs)` low or carry set
- `bhs (= bcc)` high same or carry clear
- `bls` low same
- `bcs, bcc` carry set or clear

Conditional Branching

Signed Branches - based on signed comparison involve N and V, sometimes Z

- `bgt` greater than
- `blt` less than
- `bge` greater than or equal to
- `ble` less than or equal to
- `bmi, bpl` minus or plus based on N bit
- `bvs, bvc` overflow set or clear

Conditional Branching - Examples

```
cmpi.w #$F00,my_val ; perform a test
blt wow ; branch if my_val < $F00
```

The exact CC settings required for the branch to be taken are specified in the Conditional Tests table, but there is an easier and more intuitive way to determine this.

Conditional Branching - Examples

Typically these two lines are read as:

if dst condition src

then branch to label

{else continue sequential processing }

```
tst.b d0         ; perform a test
beq handle_zero  ; branch if d0 is zero
                 ; if d0 == 0
```

Conditional Branching - Examples

- ```
  move.w value,d0
  tst.w d0          ; not necessary - REDUNDANT!!
  bmi my_label      ; branch if d0 < 0
  ```
- ```
  add.l #$FFFFFFF0,d5
  bcs handle_error
  ```
- ```
  mulu.w d4,d7
  bne prod_not_zero
  ```
  ```

Conditional Branching - Examples

- ```
  move.b #1,d0
  cmp.b #-1,d0      ; compute d0 - (-1) and set CCs
  bgt do_it         ; branch if d0 “greater than” -1
  ```
- ```
  move.b #1,d0
  cmp.b #-1,d0
  bhi do_it2        ; branch if d0 “higher than” $FF
  ```

IF/THEN

General form in high level language:

```
if <test> then
    <if-body>
```

In high level languages if the <test> evaluates to true the then is executed; otherwise the then is skipped.

IF/THEN

```
if (x > 0) then
    printf("...");


          tst.w x
          bhi then
then:     ???
```

Problem: - how do you get the desired pattern of execution??

IF/THEN

General form in high level language:

```
if <test> then
    <if-body>
```

This requires reversing the branching condition as specified in the high level language code.

General Translation

```
        <test>
        bcc after        ; branch to after if the test is false
        <if-body>
after:  ...
```

IF/THEN

```
if (x > 0) then
    printf("...");


        tst.w x
        bls after
then:   ???
after:
```

IF/THEN/ELSE

General form:

```
if <test> then
    <if-body>
else
    <else-body>
```

With an if-then-else you have the choice of reversing the branching condition

OR

leaving the condition unchanged and reversing the order of the then and else

IF/THEN/ELSE

General Translation 1:

```
<test>
bcc else ; branch if test is false
    <if-body>
    bra after
else: <else-body>
after: ...
```

General Translation 2:

```
<test>
bcc else ; branch if test is true
    <else-body>
    bra after
else: <if-body>
after: ...
```

Nested IF/THEN/ELSE

General form of nesting in the if-body:

```
if <test1> then
    if <test2> then
        <if-body2>
    else
        <else-body2>
else
    <else-body1>
```

Nested IF/THEN/ELSE

General translation:

```
            <test1>
            bcc else1           ; branch if false
then1:      <test2>
            bcc else2           ; branch if false
then2:          <if-body2>
            bra end_if2
else2:          <else-body2>
end_if2:    bra end_if1
else1:          <else-body1>
end_if1:    ...             ; after
```

Nested IF/THEN/ELSE

General translation:

```
            <test1>
            bcc else1           ; branch if false
then1:      <test2>
            bcc else2           ; branch if false
then2:          <if-body2>
            bra end_if2
else2:          <else-body2>
end_if2:    bra end_if1
else1:          <else-body1>
end_if1:    ...             ; after
```

Nested IF/THEN/ELSE

```
then2:          <if-body2>
            bra end_if2
else2:          <else-body2>
end_if2:    bra end_if1
else1:          <else-body1>
end_if1:    ...             ; after
```

Notice the excessive branching to unconditional branch – the branch at the end of then2 transfers control to an unconditional branch.

Nested IF/THEN/ELSE

Unless there is code following the then-else blocks this can be simplified and optimized in the following fashion

```
then2:          <if-body2>
            bra after
else2:          <else-body2>
end_if2:    bra after
else1:          <else-body1>
after:      ...
```

IF/THEN/ELSE IF/ELSE

General form of nesting in the else-body (“else if” statements):

```
if <test1> then
    <if-body1>
else if <test2> then
    <if-body2>
else
    <if-body3>
…
```

IF/THEN/ELSE IF/ELSE

Optimized solution:

```
        <test1>     ; case 1?
        bcc else1       ; branch if not
        <if-body1>      ; handle case 1
        bra after
else1:  <test2>     ; case 2?
        bcc else2       ; branch if not
        <if-body2>      ; handle case 2
        bra after
else2:  <if-body3>      ; default case
after: …
```

Loops

What is a loop?

- a branch (if statement) at the top
- an unconditional branch at the bottom
  - branches back to the top

```
loop:   <test>
        bcc after
        …
        bra loop
after:
```

Loops

Examples:

- while loop
- for loop
- repeat – until loop (do while)

Loops

As a convenience, the 68000 provides a “decrement and branch” instruction. This instruction can be used to implement “down counting” for loops.

```
;if cc is false: decrement Dn, then branch if Dn ≠ -1

    dbcc Dn,<label>
```

Note:

- for db**cc**, Dn’s size is always word.
- dbra is equivalent to dbf, i.e. an unconditional decrement and branch.
- the condition allows for an early stopping case.

Compound Conditionals

Compound conditionals involve the logical operators AND and OR.

```
if (x > 0 && y <= 10) then
    ...
```

You could implement this by:

- creating a Boolean type
- creating a variable or assigning a register for each term
- evaluate each term and set the corresponding variable/register
- evaluate the logical expression by applying the logical operations to the appropriate Boolean values.
- test the resulting Boolean value and branch accordingly

Compound Conditionals

This seems like an excessive amount of effort.

Alternatively a complex logical expression using:

- ANDs can be converted into nested IF-THEN-ELSE.
- ORs can be simplified by applying DeMorgan's Law (to make an expression of ANDs) and then convert into nested IF-THEN-ELSE

Note: DeMorgan's Law:

- A or B = !(!A and !B)

Compound Conditionals - Short Circuits

- For languages that use short-circuit evaluation a direct translation is possible
- Remember that short-circuit evaluation means that if the final result of a logical expression can be determined without evaluating the entire expression then this is done.
  - For an AND if the LHS is false then the expression will be false
  - For an OR if the LHS is true then the expression will be true

Compound Conditionals - Short Circuits

So for the following logical expression:

```
if (x > 0 && y <= 10) then
    ...
```

- If x > 0 is false then the expression will be false and there is no need to evaluate the second term, but can directly go to the else or end
- If x > 0 is true then y <= 0 must be evaluated to determine whether or not to go to the else or end

Compound Conditionals - Short Circuits

```
if (x > 0 && y <= 10) then
    ...
```

Translates to: ; assuming x is an unsigned word and y is a signed longword

```
if:     tst.w x
        bls after           ; branch if x <= 0
        cmpi.l #10,y
        bgt after           ; branch if y > 10
then:   ...                 ; only get here if both of the
                            ; above conditions are true
after:
```

Compound Conditionals - Short Circuits

```
if (x > 0 || y <= 10) then
    ...
```

Translates to: ; assuming x is an unsigned word and y is a signed longword

```
if:     tst.w x
        bhi then        ; first condition is true
        cmpi.l #10,y    ; only get here if 1st condition fails
        bgt after       ; normal fail branch
then:   ...         ; get here if either condition
                        ; is true
after:
```