Program Control Flow

CIS*2030
Lab Number 4

Name: Zachery Abbas

Mark: __________/214

Overview

Ultimately, what makes computers useful is their ability to choose between different courses of action. Without the ability to make decisions, computers would be limited to executing simple programs consisting only of straight-line code. In this lab, we will explore some of the instructions in the 68000's ISA that control program flow. These instructions are used to control the order in which other instructions are executed, which, in turn, is the key to implementing higher level decision making constructs, like if-statements, switch-statements, while-loops, do-while loops, etc.

Objectives

Upon completion of this lab you will:

- Understand how computers make decisions,
- Understand the difference between conditional and unconditional branch instructions,
- Understand how compare instructions affect the various flags in the CCR,
- Understand which types of compare and branch instructions to use when working with signed or unsigned data, and,
- Understand how to construct high-level decision-making constructs from a control-flow graph.

Preparation

Prior to starting the lab, you should review your course notes, and perform the following reading assignments from your textbook (if you have not already done so):

- Sections 0.1, 0.2, and 0.3 (number systems, un/signed numbers, ASCII)

Introduction

As explained in class, computers are really designed to execute straight-line code. The program counter (PC) holds the address of the *next* instruction to be executed. The instruction cycle then fetches the instruction pointed to by the PC, decodes the instruction, increments the PC by the size of the instruction (so that the PC now points to the next instruction in memory), and finally executes the fetched instruction. The instruction cycle then repeats, and in this way instructions are executed in sequence; that is, one after another.

The key to executing instructions out of order (i.e., not sequentially) is to change the address in the PC, so that the PC no longer points to the *next instruction in memory*, but to the location of some other (target) instruction. Instructions that have the ability to alter the address in the PC are referred to as *branch* or *jump* instructions, and can be either *conditional* or *unconditional*. Conditional branch instructions first perform a test to see if a particular condition is true. If the condition is true, the branch is taken (i.e., the PC is updated with the target address of the branch). Otherwise, the branch is not taken (i.e., the PC is not updated, and so the next instruction to execute is the next one in sequence immediately following the branch). In the case of an unconditional branch instruction, or a jump instruction, the contents of the PC are always changed, and so the branch always happens.

In practice, both branch and jump instructions must contain a target address, which is the new address that is to be placed into the PC if the branch happens. In some situations, like loops, the target address may be close to the location of the current instruction. In cases like this, it is not necessary to store the full 32-bit address as part of the instruction. Rather, a smaller 16-bit, or even 8-bit, offset can be stored in the instruction that represents the difference in bytes from the current instruction to the target instruction. This offset can be treated as a signed value, allowing for both forward (positive) and backward (negative) branches. This is known as *PC-relative* addressing, and is used by branch all instructions, regardless of whether or not they are conditional or unconditional. In cases where the target is farther away than can be reached with an 8-bit or 16-bit offset, the entire 32-bit address can be stored as part of the instruction. This is known as *absolute* addressing, and is used by jump instructions.

As illustrated in Fig. 1, at the assembly-language level, decision-making is a two-step process. The first step involves storing the results of operations on program data in the flags in the condition-code register (CCR). The second step involves using a conditional branch instruction to test these flags to see if a particular condition is met. If the condition is met, the branch happens. Otherwise, the branch is not taken, and the instruction following the branch instruction is the next to be fetched and executed. This simple decision-making mechanism allows one course of action to be associated with the branch being true, and another with the branch being false.

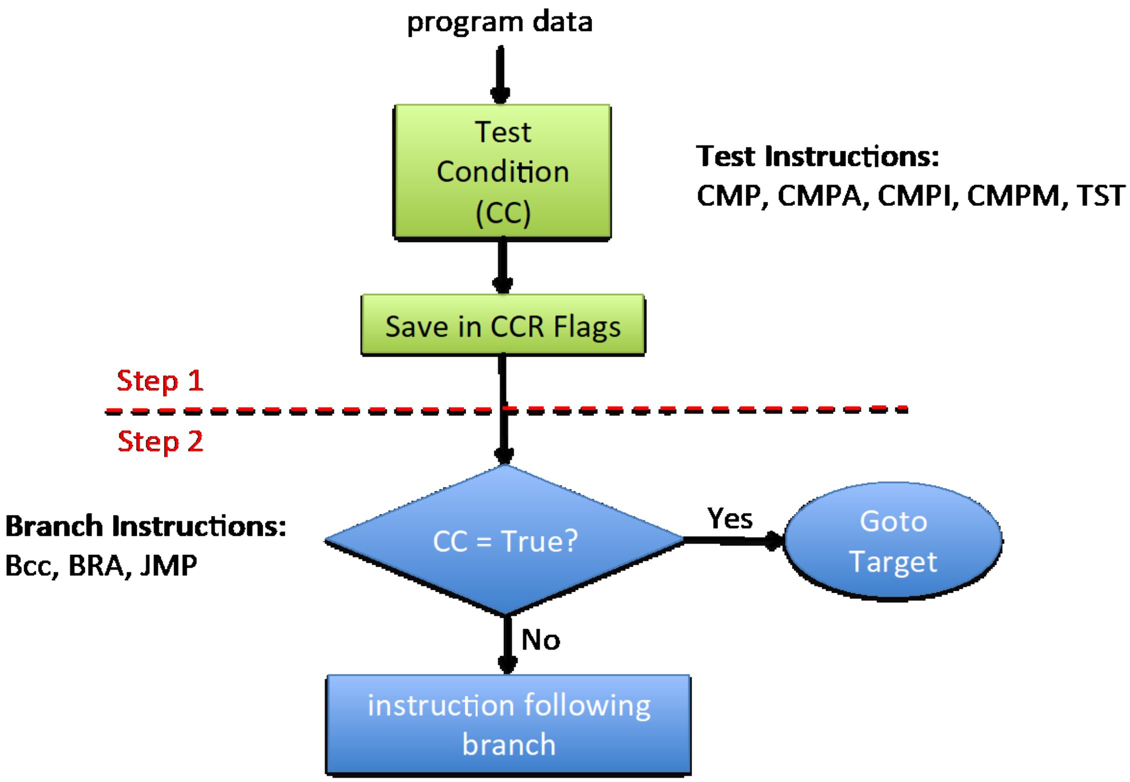


Figure 1: Decision making at the assembler level.

Part 1: Compare and Test Instructions

In practice, *any* instruction that updates the state of the flags in the CCR can be used in the first step described in Fig. 1. However, specific compare and test instructions are available for the purpose of testing individual data items or pairs of data items. These include: CMP, CMPA, CMPI, CMPM, and TST. Compare instructions are used to compare the relative magnitude of two values, and only affect the flags in the CCR. The comparison is accomplished by subtracting the contents of source operand from the destination operand without affecting either operand. The result of the subtraction is then used to set or clear the flags in the CCR, which can then be examined using different conditional branch instructions. Multiple compare instructions (with different address modes) are available for use in different situations. For example, CMPA is used when comparing pointers, CMPI is used to make a comparison with an immediate (constant) value, CMPM is used to compare two memory-resident values, and the CMP is used to compare data with the contents of a data register. The TST instruction operates similarly to the compare instructions, but has a single operand that is compared with *zero*. TST instructions are useful in situations where it is necessary to determine whether a value is zero/non-zero or positive/negative.

Before proceeding, review the various **compare instructions on pages 83, 293-296 of your textbook**, and the **TST instruction on pages 85 and 362**.

Questions

1. What is the state of the zero (Z) flag in the CCR after execution of the following instruction:

```
CMP.W D0,D1
```

Assume that both `D0` and `D1` contain the hexadecimal value 0x00003215. Show your work by doing the calculation by hand. **[2 points]**

```
0000 3215 = 11001000010101
0000 3215 = 11001000010101
            ______________
             0000000000000
z=1
n=0
v=0
c=0
```

2. What is the state of the zero (Z), Negative (N), Overflow (V) and Carry (C) flags in the CCR after execution of the following instruction:

```
CMPI.B #5,(A0)
```

Assume that `A0` contains the address 0x00009000 and memory location 0x00009000 contains the hexadecimal value 0x04. Show your work by doing the calculation by hand. **[4 points]**

```
(100)
(-101)
---------
001

(0x01)
         16

c=1 n=1 z=0 v=0
```

3. What is the state of the zero (Z), Negative (N), Overflow (V) and Carry (C) flags in the CCR after execution of the following instruction:

```
TST.L D0
```

Assume that `D0` contains the hexadecimal value 0xFFFFFFFF. Show your work by doing the calculation by hand. **[4 points]**

```
11111111111
00000000000
-------------------
11111111111

Z=0
N=1
V=0
C=0
```

You should use Easy68K to verify that the answers that your answers for questions 1-3 are correct.

Part 2: Branch Instruction Encodings

As discussed in class, the 68000 supports fourteen different conditional branch instructions, and one unconditional branch instruction, called BRA. The conditional branch instructions have the format "Bcc Label", where *cc* is one of 14 conditional tests (**see Table 3-14 of your textbook**) and *Label* is the memory address of the instruction to branch to should the condition being tested (cc) evaluates to true. The unconditional branch instruction has the format "BRA Label", where, once again, *Label* is the location to branch to.

With regards to branch instruction encodings, both conditional and unconditional branch instructions employ a form of "PC-relative addressing," where either an 8-bit signed offset or a 16- bit signed offset is added to the current address in the PC to form the address represented by the branch Label. An 8-bit signed value allows for a branch +126 bytes forward in memory or -128 bytes backward in memory, while a 16-bit signed value extends the previous range to +32766 to - 32768. As illustrated in Fig. 2., branch instructions that use an 8-bit offset encode the offset in the instruction's operation word, while branch instructions that use a 16-bit offset encode the offset in a single extension word following the operation word.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
								8-bit offset							
16-bit offset															

Figure 2: Branch instruction format.

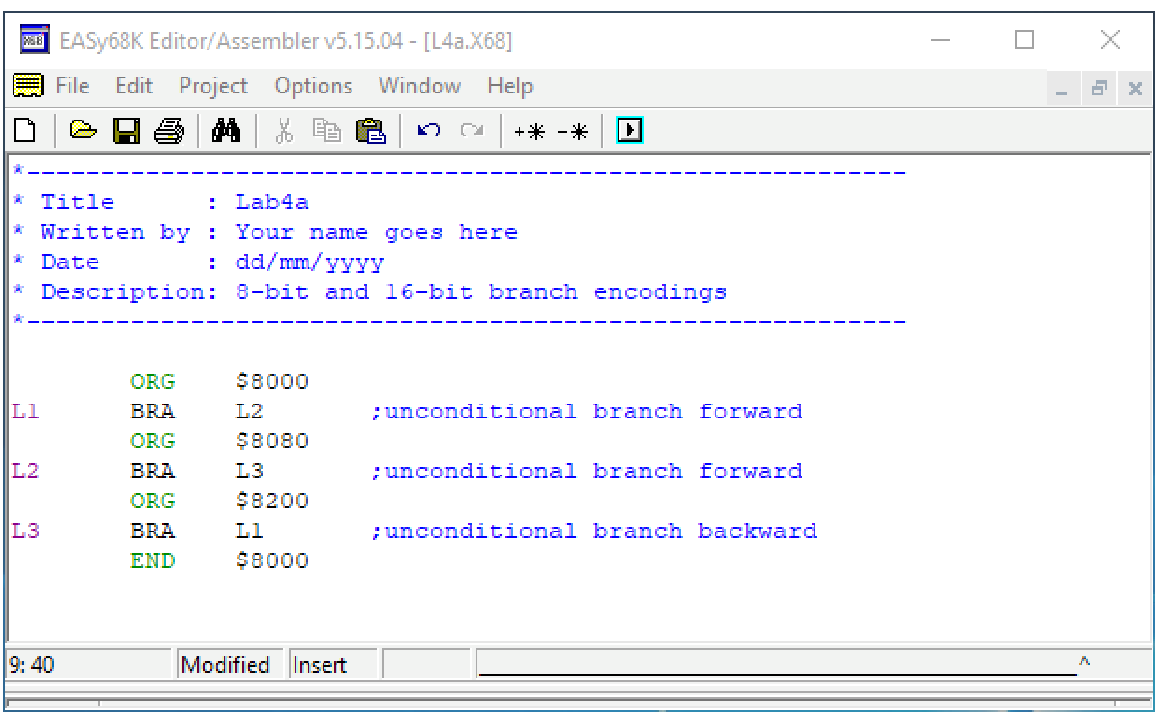
Before proceeding, **read Sec. 2.4.14 of your textbook to review how PC-relative addressing** works, and how the assembler computes the 8-bit (or 16-bit) offset to be added to the PC to form the effective address of the branch location.

Step 1

Download the sample program called **Lab4a.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab4a.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)



Step 3

What memory address is associated with each of the three labels `L1`, `L2` and `L3`? Print the (32-bit hexadecimal) memory address in the table below. **[3 points]**

Label	Address (32-bit)
L1	00008000
L2	00008080
L3	00008200

Step 4

Calculate the offset of the `BRA L2` instruction by hand, showing your work. Will the offset be stored as an 8-bit or 16-bit signed value? Explain. **[3 points]**

pc+offset

007E

the offset will be started as 16-bit value because the low byte is 0 of extension word.

Step 5

Calculate the offset of the `BRA L3` instruction by hand, showing your work. Will the offset be stored as an 8-bit or 16-bit signed value? Explain. **[3 points]**

pc+offset

17E

the offset will be 16-bit because the value because the low byte is 0 of extension word.

Step 6

Calculate the offset of the `BRA L1` instruction by hand, showing your work. Will the offset be stored as an 8-bit or 16-bit signed value? Explain. **[3 points]**

pc+offset the offset will be 16-bit because the value because the low byte is 0 of extension word.

Two important benefits arising from the use of PC-relative addressing are (1) the resulting code is able run no matter where it is located in memory, and (2) encoding the offset in the operation word, or at most one extension word, is efficient. However, one limitation associated with PC-relative addressing is the maximum branching address is only 32K bytes on either side of the current instruction. In cases where it is necessary to branch farther away, a JMP instruction can be used. The JMP instruction is functionally equivalent to the BRA instruction, but places a full 32-bit address into the PC. Thus, a JMP instruction can be used to reach *any* location in the processor's address space.

Before proceeding, **read Sec. 3.2.7 and page 310 of your textbook** to review how the JMP instruction works. Notice the different address modes that can be used with a JMP instruction.

Part 3: Conditional Branch Instructions that examine a single bit in the CCR

As discussed in class, the different conditional branch instructions are organized into three distinct groups. One group contains those that enable the programmer to make decisions based on an examination of a single bit in the CCR. These branch instructions are summarized in below:

Instruction	Effect
`BCC Label`	if (C=0) PC ← Label
`BCS Label`	if (C=1) PC ← Label
`BVC Label`	if (V=0) PC ← Label
`BVS Label`	if (V=1) PC ← Label
`BPL Label`	if (N=0) PC ← Label
`BMI Label`	if (N=1) PC ← Label
`BNE Label`	if (Z=0) PC ← Label
`BEQ Label`	if (Z=1) PC ← Label

In the previous table, PC 🡐 Label refers to PC-relative addressing, and `Label` refers to the following:

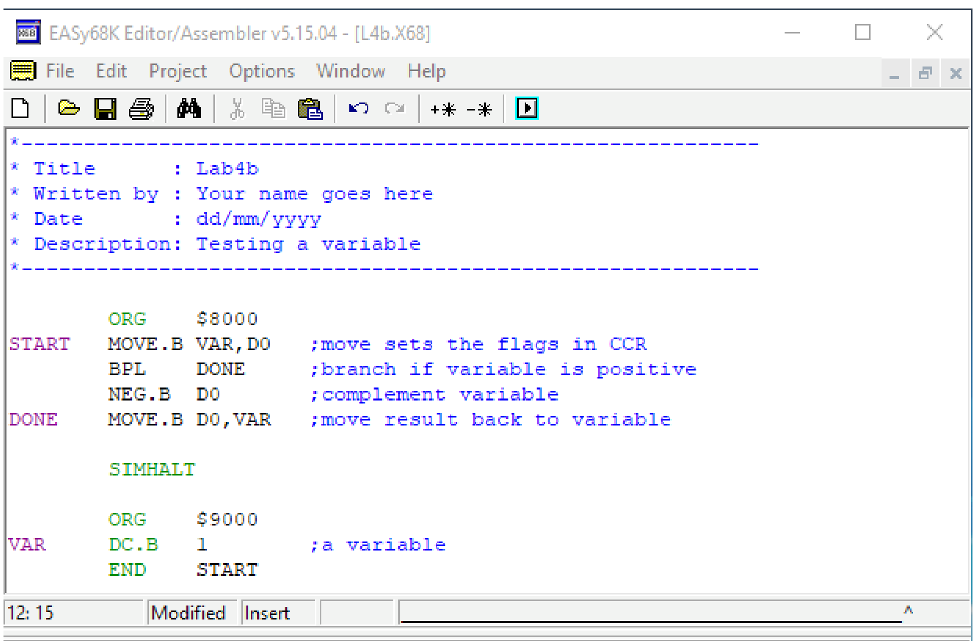
- An explicit numeric address or a symbolic name (i.e., label in the assembly-language program) for a numeric address in the original source code.
- Either an 8-bit or 16-bit offset encoded into the actual machine instruction, depending on the relative location of the target to the branch instruction.

Step 1

Download the sample program called **Lab4b.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab4b.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)



Note: You can read about the NEG instruction on **pages 84 and 330 of your textbook**.

Step 3

Read through the previous source code. In a single sentence, explain the high-level purpose of the program. Don't explain what each instruction does! **[2 points]**

Gets the absolute value

Step 4

What 8-bit (hexadecimal) value do you expect for `VAR` after executing this code? **[1 point]**

Variable	Expected Value
`VAR`	01

Step 5

Assemble the program, and then fill out the "Before Run" section of the following table. **[1 point]**

Variable	Before Run	After Run
`VAR`	00000001	00000001

Now, run the program, and then complete the "After Run" section of the previous table. Compare the expected value for with the actual value you obtained after running the program.

Step 6

Use the trace facility to answer the following question. What is the value in the CCR immediately before and after the `MOVE.B VAR,D0` instruction executes? **[4 points]**

Instruction	Before				After			
	N	Z	V	C	N	Z	V	C
`MOVE.B VAR,D0`	0	0	0	0	0	0	0	0

Why are the individual flags in the CCR set or cleared the way that they are after the `MOVE.B VAR,D0` instruction executes? Be precise. **[4 points]**

they are like that because 01 is not zero it is a carry it is not an overflow so all 0s are given to CCR

Step 7

Use the trace facility to answer the following question. What is the 32-bit (hexadecimal) value in the PC immediately before and after the `BPL DONE` instruction executes? **[1 point]**

Instruction	PC Before	PC After
`BPL DONE`	00008006	0000800C

Clearly the previous branch is taken, and control is passed to the instruction at location `DONE`. Explain why the branch was taken. You should make reference to the appropriate flag(s) in the CCR in your answer. **[2 points]**

the branch was taken there because the code branches because not a negiative number (N=0) when iwhy the code went straight to done and didn't go through theyre neg.b

Step 8

Modify the branch condition in the source code presented in step 1 by replacing the test for a positive number with a test for a *negative* number. Save the new program in a file called **Lab4c.X68**.

Step 9

What 8-bit (hexadecimal) value do you expect for `VAR` after executing this code? **[1 point]**

Variable	Expected Value
`VAR`	FF

Step 10

Assemble the program, and then fill out the "Before Run" section of the following table. **[1 point]**

Variable	Before Run	After Run
`VAR`	0000 00FF	0000 00FF

Now, run the program, and then complete the "After Run" section of the previous table. Compare the expected value for with the actual value you obtained after running the program.

Step 11

Use the trace facility to answer the following question. What is the 32-bit (hexadecimal) value in the PC immediately before and after the `BMI DONE` instruction executes? **[1 point]**

Instruction	PC Before	PC After
`BMI DONE`	00008004	00008008

Clearly the previous branch is not taken, and the next instruction to execute following the branch is `NEG.B D0`. Explain why the branch was not taken. You should make reference to the appropriate flag(s) in the CCR in your answer. **[2 points]**

the pervious branch is taken because now the program branches at 1 which is why n=1

Part 4: Signed and Unsigned Conditional Branch Instructions

The remaining conditional branch instructions are organized into two groups: those that allow for a comparison of two signed numbers, and those that allow two unsigned values to be compared. These branch instructions are summarized in below:

Type	Instruction	Effect
Signed Branches	`BGE Label`	if (A>=B) PC 🡐 Label
	`BGT Label`	if (A>B) PC 🡐 Label
	`BLE Label`	if (A<=B) PC 🡐 Label
	`BLT Label`	if (A<B) PC 🡐 Label
Unsigned Branches	`BHS Label`	if (A>=B) PC 🡐 Label
	`BHI Label`	if (A>B) PC 🡐 Label
	`BLS Label`	if (A<=B) PC 🡐 Label
	`BLO Label`	if (A<B) PC 🡐 Label

In the previous table, it is assumed that the flags in the CCR have first been set as a result of subtracting value B from value A (e.g., by using one of the compare instructions introduced in Part 1).

PC 🡐 Label refers to PC-relative addressing, and `Label` refers to the following:

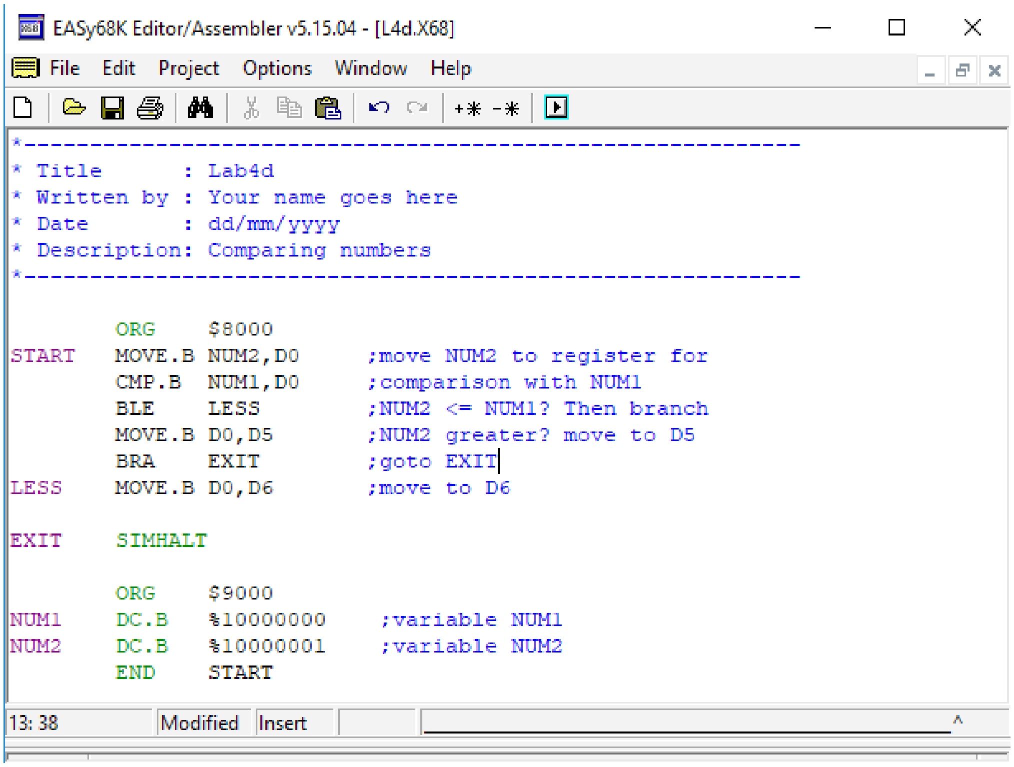
- An explicit numeric address or a symbolic name (i.e., label in the assembly-language program) for a numeric address in the original source code.
- Either an 8-bit or 16-bit offset encoded into the actual machine instruction, depending on the relative location of the target to the branch instruction.

Step 1

Download the sample program called **Lab4d.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab4d.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)



Step 3

Read through the previous source code. In 2-3 sentences, explain the high-level purpose of the program. Don't explain what each instruction does! Get to the core of the apple! **[2 points]**

there are two numbers that are compared if the second # is greter then the first one it is too the address

Step 4

Are the variables `NUM1` and `NUM2` signed or unsigned? How do you know? Hint: Examine the conditional branch instruction. **[2 points]**

It is for signed # because they use BLE which is for signed #

Step 5

What 32-bit (hexadecimal) values do you expect registers `D5` and `D6` to have after executing this code? **[1 point]**

Registers	Expected Value
D5	00000081
D6	00000000

Step 6

Assemble the program, and then fill out the "Before Run" section of the following table. **[2 points]**

Registers	Before Run	After Run
D5	00000000	00000081
D6	00000000	00000000

Now, run the program, and then complete the "After Run" section of the previous table. Compare the expected value for with the actual value you obtained after running the program.

Step 7

Use the trace facility to answer the following question. What is the value in the CCR immediately before and after the `CMP.B NUM1,D0` instruction executes? **[4 points]**

Instruction	Before				After			
	N	Z	V	C	N	Z	V	C
CMP.B NUM1,D0	1	0	0	0	0	0	0	0

Why are the individual flags in the CCR set or cleared the way that they are after the `CMP.B NUM1,D0` instruction executes? Be precise. **[4 points]**

it is dne like that because when cmp the two number the machine is company the consective values so n=1 but now it is n=0 the rest are zero

Step 8

Use the trace facility to answer the following question. What is the 32-bit (hexadecimal) value in the PC immediately before and after the `BLE LESS` instruction executes? **[1 point]**

Instruction	PC Before	PC After
`BLE LESS`	0000800C	00008010

Clearly the previous branch is not taken, and control is passed to the instruction immediately following the `BLE LESS` instruction. Explain why the branch was not taken. You should make reference to the appropriate flag(s) in the CCR in your answer. **[2 points]**

The branch was not taken because it itakes me signed if which is why to ccr is 000 because the cmp.b makes them so # 0000

Step 9

Modify the branch condition in the source code presented in step 1, so that the program works correctly under the assumption that the variables `NUM1` and `NUM2` are unsigned values. Save the new program in a file called **Lab4e.X68**.

Step 10

What 32-bit (hexadecimal) values do you expect registers `D5` and `D6` to have after executing this code? **[1 point]**

Registers	Expected Value
`D5`	00000081
`D6`	00000000

Step 11

Assemble the program, and then fill out the "Before Run" section of the following table. **[2 points]**

Registers	Before Run	After Run
`D5`	00000000	00000081
`D6`	00000000	00000000

Now, run the program, and then complete the "After Run" section of the previous table. Compare the expected value for with the actual value you obtained after running the program.

Step 12

Use the trace facility to answer the following question. What is the 32-bit (hexadecimal) value in the PC immediately before and after the `BLS LESS` instruction executes? **[1 point]**

Instruction	PC Before	PC After
`BLS LESS`	0000800C	00008010

Clearly the previous branch is not taken, and control is passed to the instruction immediately following the `BLS LESS` instruction. Explain why the branch was not taken. You should make reference to the appropriate flag(s) in the CCR in your answer. **[2 points]**

nounsigned the ccr is negitave which means first thatcomparing regular numbers that it is not taken.

Part 5: A Conditional Program

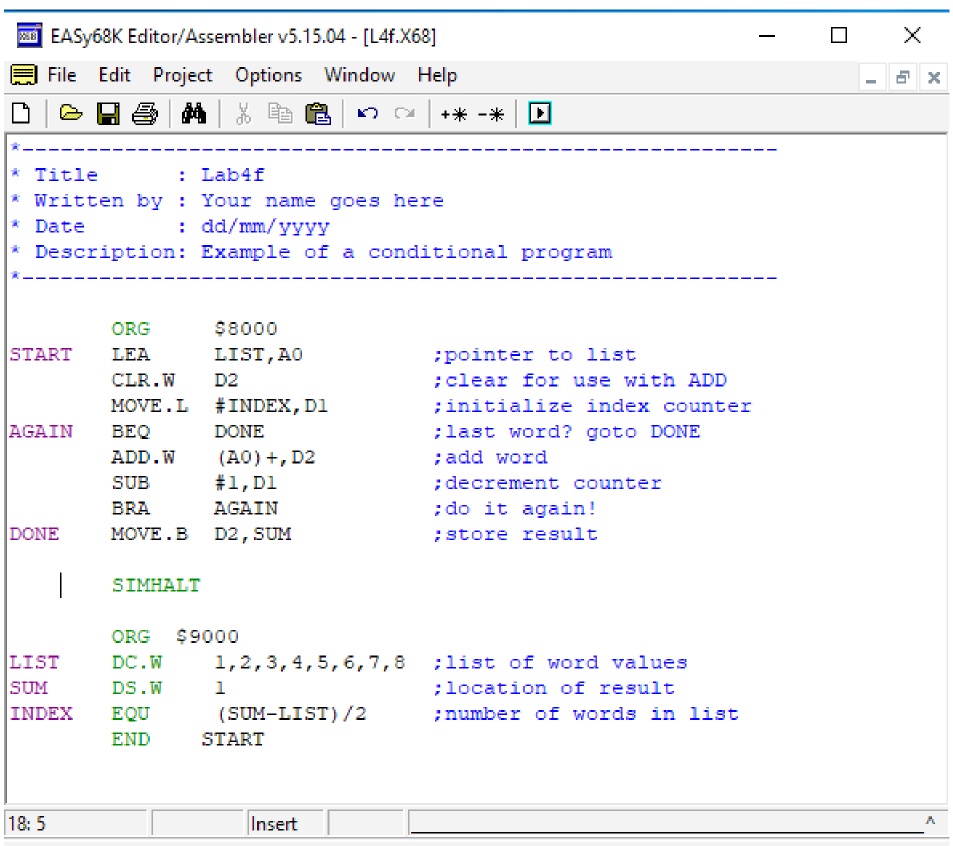
Conditional instructions enable programs to solve problems by choosing between different courses of action during the execution of the program code. Before proceeding, we will look at an example of a slightly more complex program than we have encountered in the previous examples.

Step 1

Download the sample program called **Lab4f.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab4f.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)



Questions

4. What is the purpose of the program? Be brief? **[2 points]**

the propgram goes through a list of eords then adds the #s in the list and will loop till done then stores it

5. What is the 32-bit hexadecimal final value in address register A0 once the program terminates? **[1 point]**

0000 9010

6. What is the final 16-bit value in memory location 0x00009010 once the program terminates? **[1 point]**

24FF

7. How many times is the conditional branch `BEQ DONE` found to be true and, therefore, taken? **[1 point]**

Once

8. How many times is the unconditional branch `BRA AGAIN` instruction executed? **[1 point]**

8

Step 3

Modify the previous program so that it now works with a list of 32-bit long words rather than a list of 16-bit words. Save the new program in a file called **Lab4g.X68**. Make sure to simulate the program to ensure that it works correctly.

Part 6: If-else Construct

In the remainder of the lab, we continue our discussion of instructions that can be used to affect program flow by considering how these instructions are used to implement fundamental decision-making and looping constructs in high-level languages. Before proceeding, make sure that you review your lecture notes, especially those notes that describe how to construct control-flow graphs for different high-level constructs.

As discussed in class and illustrated in Fig. 3, the *if-else* statement can be represented using a control-flow graph (CFG) with four blocks. The condition block contains the condition to be checked, code block 1 contains the code to execute if the condition is true, code block 2 contains the code to execute if the condition is false (i.e., the else), and the exit block is where control is passed to once the if-else statement completes executing. In principle, the four blocks can be in any, arbitrary order in memory. However, in Fig. 2, the assumption is that the condition block appears first in memory, followed by the two code blocks, and, finally, the exit block.

Recall that all of the conditional branch instructions only branch on a *true* condition. However, in the case of the *if-else* statement, it is clear that we do not want to branch (but simply fall through into code block 1) if the condition associated with the *if* is true. Similarly, if the condition associated with the *if* is false, we want to branch over code block 1 in memory and go to code block 2. In practice, this behaviour requires branching on the *reverse* condition. For example, if the condition in the high-level code is "==", the reverse condition is "!=". Or, if the conditions is "<", the reverse condition is ">=".

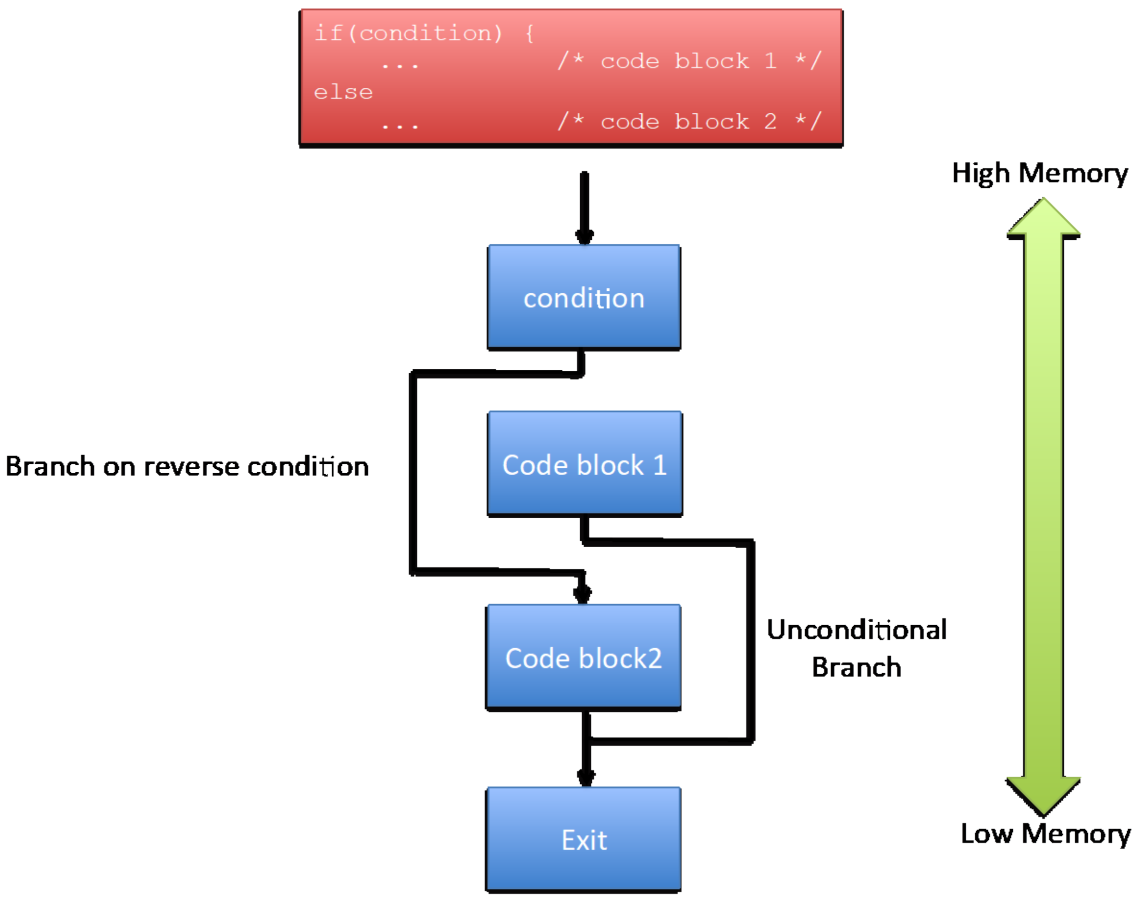


Figure 3: Control-Flow Graph for *if-else* statement.

Step 1

Create an assembly-language program called **Lab4h.X68**. The program should meet the following specifications:

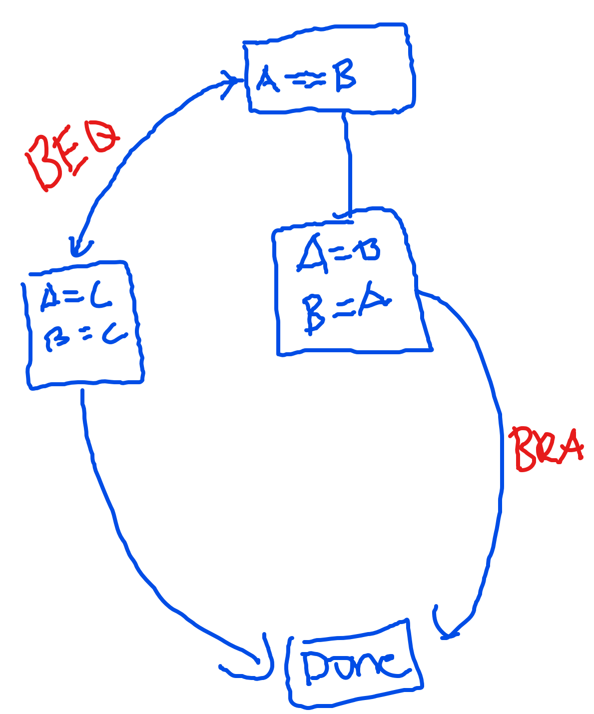
- Reserve space in memory for three 8-bit variables with the names `A`, `B`, and `C`. The initial values of `A`, `B`, and `C` should be the first, second and third digit of your student identification number, respectively.
- The program will implement the following snippet of high-level code.
- The local variable `temp` can be implemented using any of the registers `D0` through `D8`.

```
if(A == B) {

    A = C;
    B = C;
}
else {

    temp = A;
    A = B;
    B = temp;
}
```

Before you start to code, draw a complete CFG (below) for the if-else statement **following the same procedure in class**. Make sure to show the contents of each block. Also, label the various arcs with the specific name of the 68000-branch instruction that will be used to implement the transition indicated by the arc. **[10 points]**



Step 2

What 32-bit (hexadecimal) values do you expect `A` and `B` to have after executing this code? **[1 point]**

Variables	Expected Value
A	00000000
B	00000001

Step 3

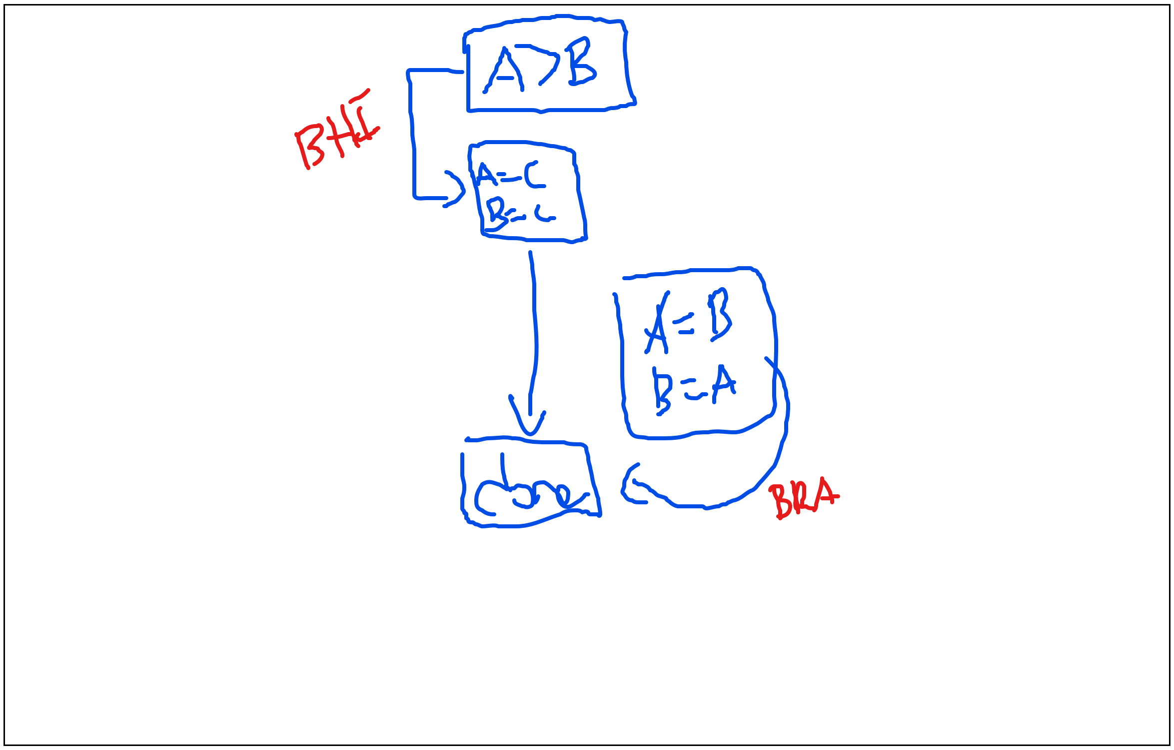
Assemble the program, and then fill out the "Before Run" section of the following table. **[2 points]**

Variables	Before Run	After Run
A	00000001	00000000
B	00000000	00000001

Now, run the program, and then complete the "After Run" section of the previous table. Compare the expected values with the actual values you obtained after running the program. **[10 points]**

Step 4

In the high-level code given in step 1, replace the equality (==) test by greater than (>), and implement a program called **Lab4i.X68** that implements the modified high-level code. (You should assume that variables `A`, `B`, and `C` are unsigned.) Begin your implementation by re-drawing the CFG for the if-else construct below, remembering to identify the appropriate branch instructions associated with the arcs. **[10 points]**



Step 5

What 32-bit (hexadecimal) value do you expect `B` to have after executing this code? **[1 point]**

Variable	Expected Value
B	00000000

Step 6

Assemble the program, and then fill out the "Before Run" section of the following table. **[2 points]**

Variables	Before Run	After Run
A	00000001	00000001
B	00000000	00000000

Now, run the program, and then complete the "After Run" section of the previous table. Compare the expected values with the actual values you obtained after running the program. **[10 points]**

Part 7: Short-Circuit Evaluation

As discussed in class, many high-level languages, like C, employ short-circuit evaluation to improve the speed and robustness of code. Short-circuit evaluation applies to Boolean expressions used in conditions, and results in the termination of the evaluation as soon as the result is known.

Step 1

Create an assembly-language program called **Lab4j.X68**. The program should meet the following specifications:

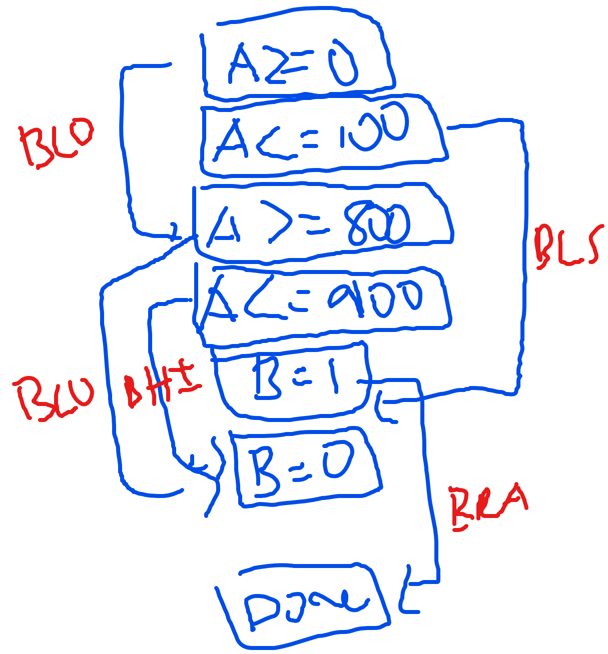
- Reserve space in memory for two 16-bit (unsigned) variables with the names `A` and `B`, respectively. The initial value of `A` should be the first, second and third digit of your student identification number concatenated together to form a 3-digit decimal number. Variable `B` is uninitialized.
- The program will implement the following snippet of high-level code.

```
if((A >= 0 && A <=100) || (A >= 800 && A <= 900)) {

    B = 1;
}
else {

    B = 0;
}
```

Before you start to code, draw a complete CFG (below) for the if-else statement **following the same procedure in class**. Use short-circuit evaluation in your implementation. Make sure to show the contents of each block. Also, label the various arcs with the specific name of the 68000-branch instructions that will be used to implement the transition indicated by the arc. **[20 points]**



Step 2

What 32-bit (hexadecimal) values do you expect registers `A` and `B` to have after executing this code? **[1 point]**

Variable	Expected Value
`B`	00000000

Step 3

Assemble the program, and then fill out the "Before Run" section of the following table. **[2 points]**

Variables	Before Run	After Run
A	00000000	00000065
B	00000000	00000000

Now, run the program, and then complete the "After Run" section of the previous table. Compare the expected values with the actual values you obtained after running the program. **[20 points]**

Part 8: While-Statement and For-Statement

As explained in class and shown below, in C, the *for*-statement and *while*-statement are equivalent looping constructs.

```
for(i=start; i<end; i++) {

    /* perform loop body */

}
```

```
i = start;
while(i<end) {

       ...
       i++;
}
```

The only difference between the two loop constructs is that the for-statement gathers together all of the constituent components of the while-statement into one convenient location for the programmer. However, during compilation, the for-statement is turned back into a while-statement. This means that both high-level looping constructs result in the same assembly-language code being generated by the compiler.

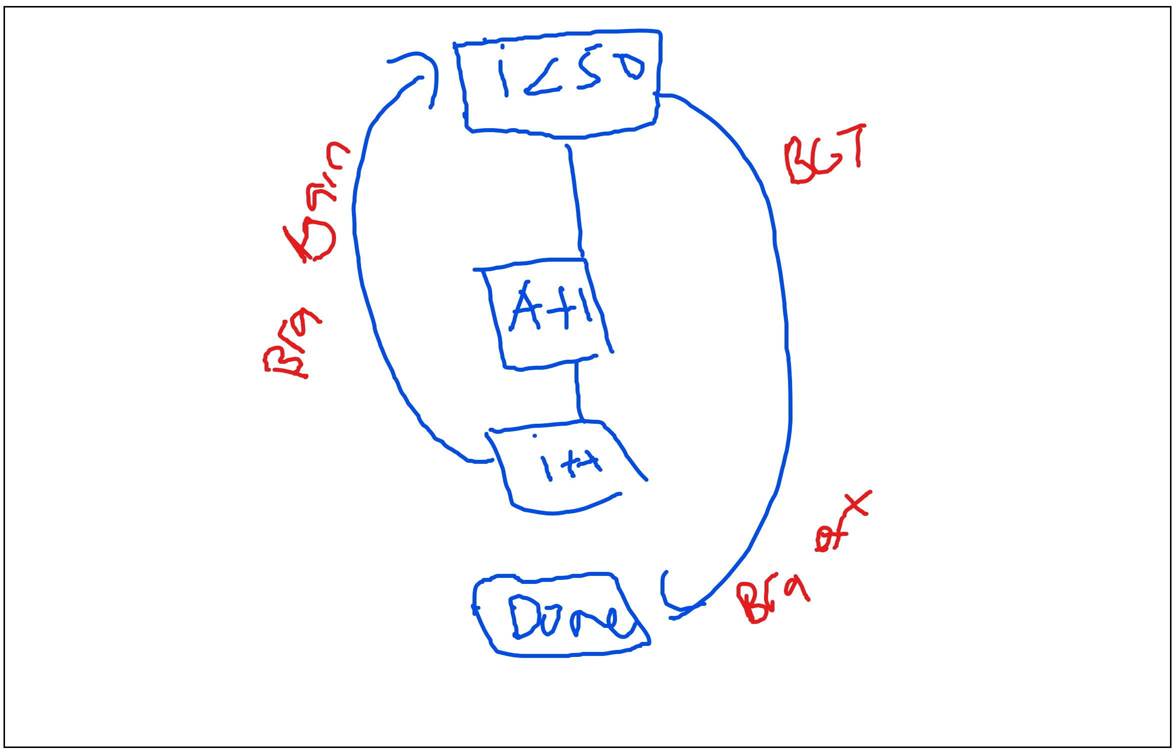
Step 1

Create an assembly-language program called **Lab4k.X68**. The program should meet the following specifications:

- Reserve space in memory for a 16-bit variable with the name `A`. The initial value of `A` should be the first digit your student identification number.
- The program will implement the following snippet of high-level code.

```
for(i=A; i<50; i++) {
     A = A + 1;
}
```

Before you start to code, draw a complete CFG (below) for the for-statement **following the same procedure in class**. Make sure to show the contents of each block. Also, label the various arcs with the specific name of the 68000-branch instruction that will be used to implement the transition indicated by the arc. **[10 points]**



Step 2

What 32-bit (hexadecimal) values do you expect `A` and `B` to have after executing this code? **[1 point]**

Variable	Expected Value
A	00000032

Step 3

Assemble the program, and then fill out the "Before Run" section of the following table. **[2 points]**

Variable	Before Run	After Run
`A`	0000001	00000032

Now, run the program, and then complete the "After Run" section of the previous table. Compare the expected values with the actual values you obtained after running the program. **[10 points]**

Part 9: Loops and Arrays

Your final task is to write a program to count the number of even and odd digits that appear in your student identification number.

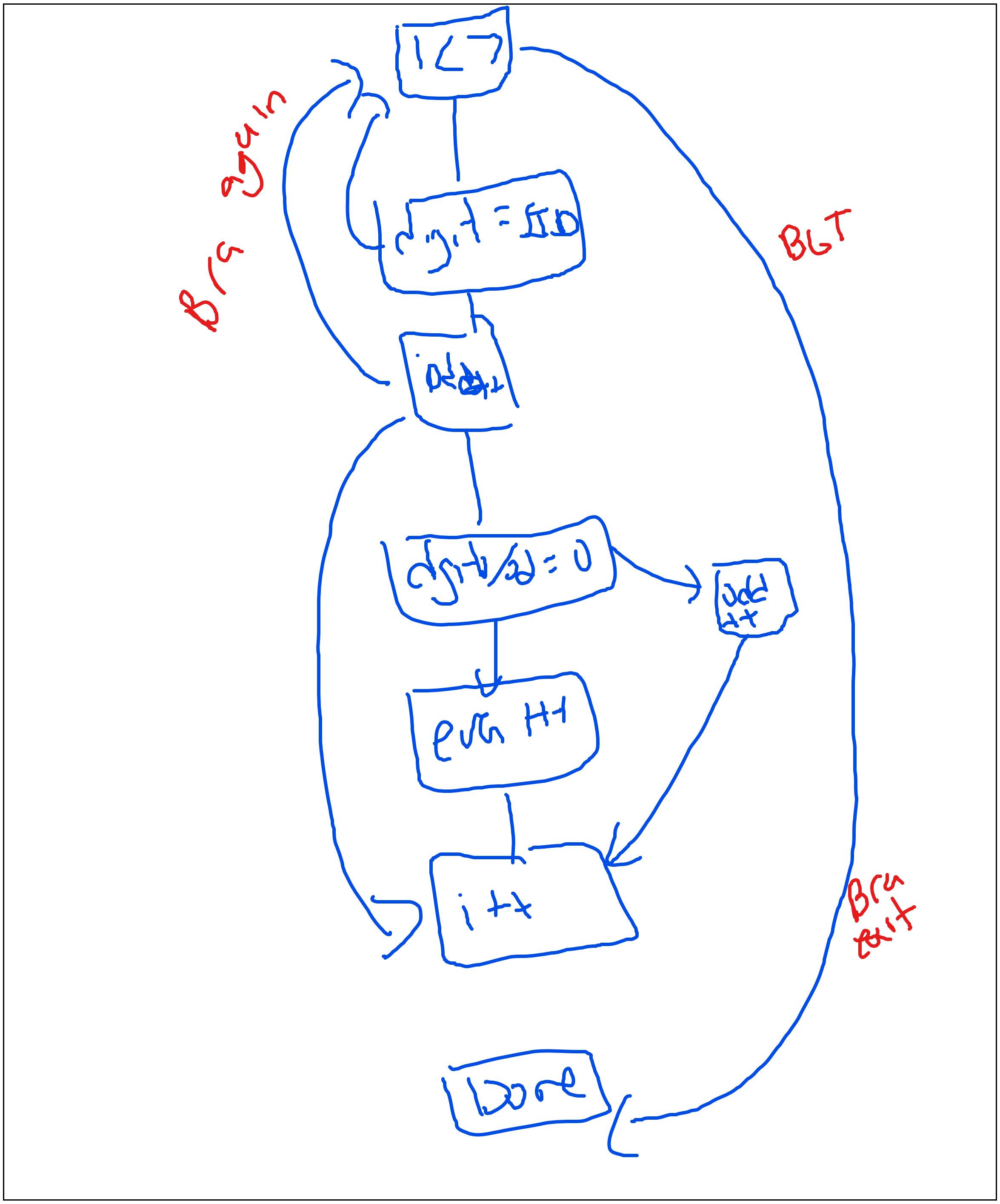
Step 1

Create an assembly-language program called **Lab4l.X68**. The program should meet the following specifications:

- Reserve space in memory for an array of seven 16-bit words, and initialize the array so that the array contains the 7 digits that comprise your student identification number. You should called this array, MY_SID.
- The program will implement the following snippet of high-level code.
- `even`, `odd` and `digit` can be implemented using any register `D0-D7`.
- You should use the *indirect-addressing with index and offset* addressing mode to access individual elements of the array.

```
MY_SID[] = {1,2,3,4,5,6,7};     /* Your SID */
even=0;
odd=0;
for(i=0; i<7; i++) {
     digit = MY_SID[i];
     if(digit % 2 == 0)
        even++;
     else
        odd++;
}
```

Before you start to code, draw a complete CFG (below) for the for-statement **following the same procedure in class**. Make sure to show the contents of each block. Also, label the various arcs with the specific name of the 68000-branch instruction that will be used to implement the transition indicated by the arc. **[10 points]**



Verify the program runs correctly. **[20 points]**