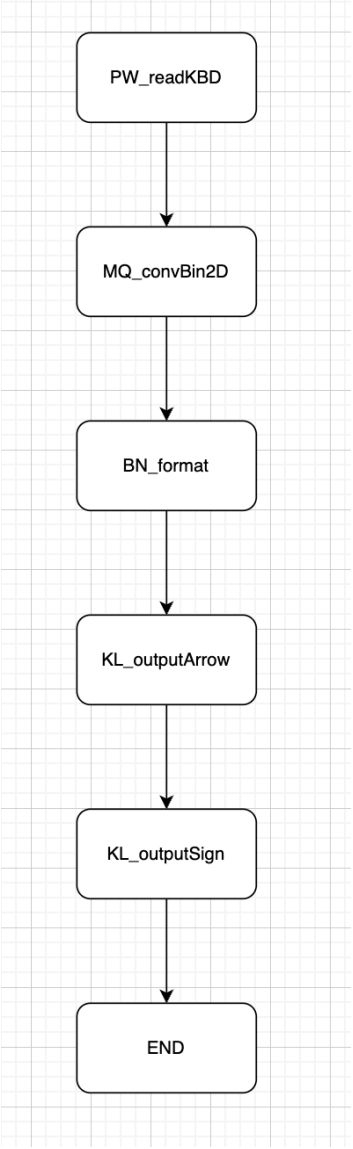


CS3A GROUP HACK: Final Report

By: Matin, Kevin, Brandon, Parker

Main:

Purpose:

Our main function serves as the entry point for the program. It handles the flow of execution by calling grouping functions to perform specific tasks:

(1) Reading and storing user input as a 16-bit binary word in R0-15.

(2) Converting the binary word to a decimal 2's complement integer.

(3) Formatting the result for output.

(4) Displaying the output with an arrow, sign, and value.

Input Handling:

The program begins with the input handling step, in which the user is given six options with which to manipulate their input (0, 1, backspace, enter, c, and q). The algorithm mainly consists of if-else statements that check for each possible input in a loop and perform the appropriate action. At the end of the step, the user's desired 16-bit binary number should be contained within the virtual registers R0 through R15, where R0 is the most significant bit, ready to be processed by Role 2.

Key Functionality & Flow

(pw_getInput)

This is the first function called by the program, with variable "ge_currentColumn" being initialized to 0 in main before its call. It begins by checking if the value of ge_currentColumn is equal to 16, in which case it would restrict access to inputting a bit (0 or 1) by jumping to (pw_non_integer_input). Otherwise, it will fall through a check for 0 (jumping to (pw_input_0) if true) and for 1 (jumping to (pw_input_1) if true). It then checks if ge_currentColumn is *not* equal to 16, in which case would restrict access to inputting an enter. Otherwise, it will fall through a check for the enter input (jumping to (pw_input_enter) if true). For the last ge_currentColumn check, ge_currentColumn is compared to 0, in which access to inputting backspace or c would be restricted if ge_currentColumn were equal to 0. Otherwise, it will fall through a check for backspace (jumping to (pw_input_backspace) if true) and for c (jumping to (pw_input_c) if true). The last input check is unrestricted and for q, which if true would simply clear the screen and jump to the end of the program. This function increments ge_currentColumn and loops after all inputs except enter and q. Before actually jumping back to the start, it goes through a conditional end loop that makes sure no key is pressed before continuing as to not fill the input with many instances of one character.

(pw_storeInput)

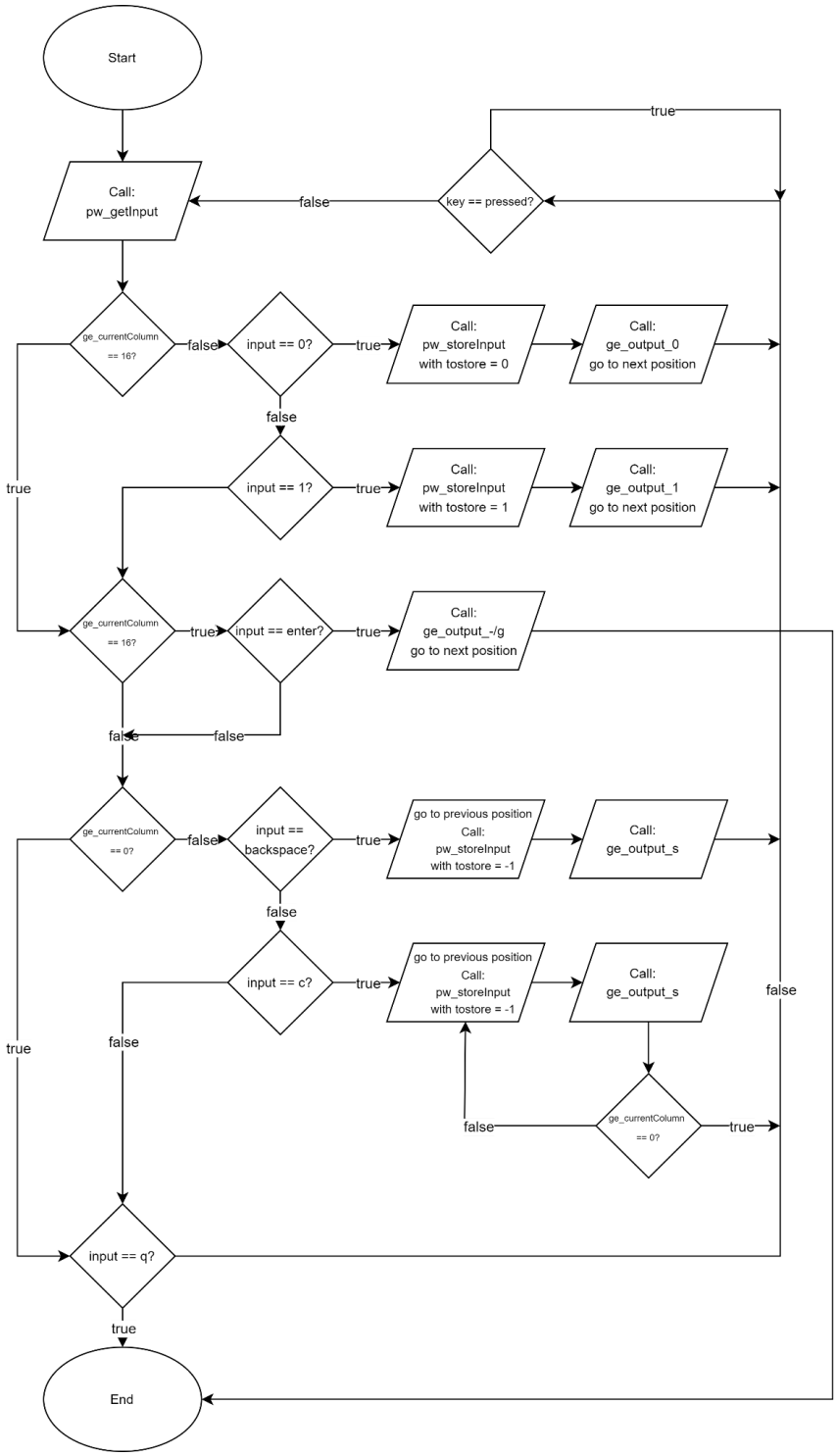
A number 0, 1, or -1 is loaded into the variable “tostore” before this function is called. The value of tostore is placed into the register (R0 - R15) corresponding to the current value of ge_currentColumn. -1 represents a cleared address, which is caused by backspacing or clearing all entries.

// Functions:
// pw_getInput
// pw_storeInput

// Variables:
// tostore
// q_after_clear (1=true, 0=false)

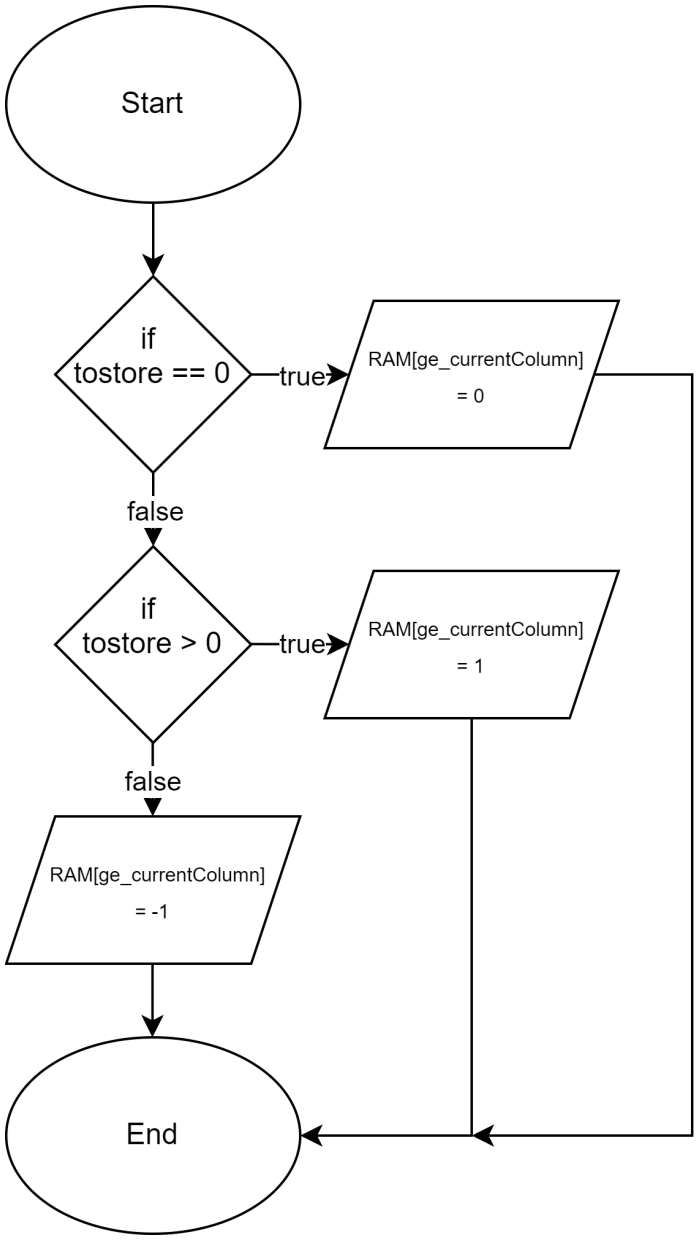
Flowchart (Input Handling):

1. Call pw_getInput.
2. Check ge_currentColumn against 16; if equal, check input against 0 and 1, otherwise skip step 3.
3. Perform appropriate actions to handle an input of 0 or 1 if either is detected.
4. Check ge_currentColumn against 16 again; if equal, check input against enter, otherwise skip step 5.
5. Perform appropriate actions to handle an enter input if it is detected.
6. Check ge_currentColumn against 0; if equal, skip step 7.
7. Perform appropriate actions to handle a backspace or a clear input if either is detected.
8. Perform appropriate action (exiting) to handle an input of q if it is detected. Otherwise, jump back to the beginning of the function and repeat the checks until a valid input is found.

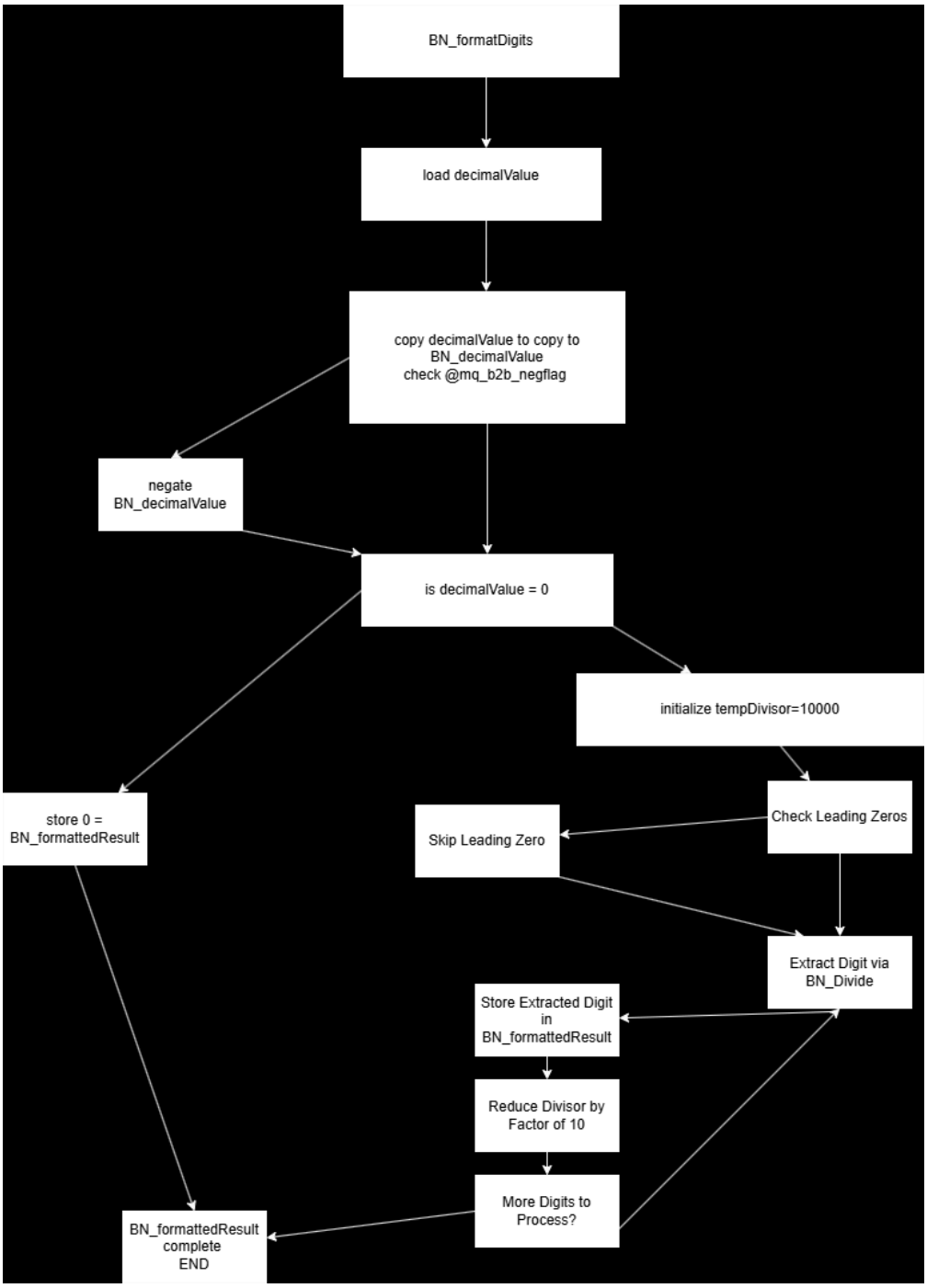


Flowchart (pw_storeInput function):

1. Check value of prestored variable tostore against 0.
2. If equal to 0, place a zero in the RAM address corresponding to the current ge_currentColumn. Then, return from the function.
3. Check value of prestored variable tostore against 0 again.
4. If greater than 0 (a 1), place a one in the RAM address corresponding to the current ge_currentColumn. Then, return from the function.
5. If the above cases don't apply, place a negative one in the RAM address corresponding to the current ge_currentColumn to represent a cleared memory address with differentiation from the binary 0 and 1. Then, return from the function.



OUTPUT_FORMATTING_BEGIN function serves as the entry point for the output formatting process. It receives the decimalValue from Role 2 and initializes the formatting sequence by directing control to the BN_formatSign function. The purpose of this function is to organize the workflow, ensuring that each step in the formatting pipeline—sign determination, digit extraction, and preparation for display—is executed in the correct order. The initialization of the process here ensures that formatting adheres to the system's input constraints and sets the foundation for producing a user-friendly output. The function begins by loading the value of @decimalValue - a variable from the previous role - and then jumps to @BN_formatDigit.



BN_formatDigits function plays a central role in converting the decimalValue provided by Role 2 into individual decimal digits. These digits are stored sequentially in the BN_formattedResult buffer for display by Role 4. The function begins with taking the variable @decimalValue and copying it to @BN_decimalValue, this is mainly to avoid any possible errors that could come from manipulating the value within. Next we take the negative flag also from Role 2 @mq_b2b_negflag and do a simple check and if it is negative we negate @BN_decimalValue as the sign will be provided later and this prevents errors in the formatting process. Next the formatting begins, this process involves progressively dividing the BN_decimalValue by decreasing powers of 10 (starting from 10,000, then 1,000, 100, etc.) to isolate and extract each digit in the correct order, from the most significant to the least significant. This function is the most complicated and so here is a step-by-step breakdown:

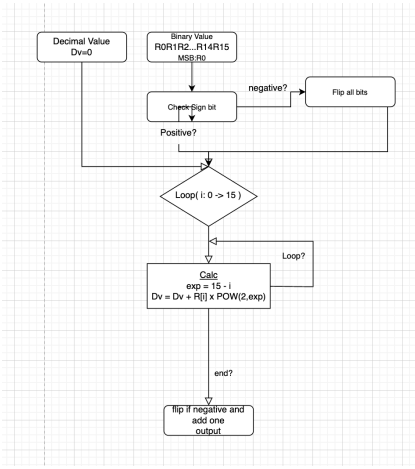
1. Initialization: The function starts after either having BN_decimalValue negated or by skipping the Negative Check, if so, there is a Handle Zero function to handle if decimal is 0. where if the by setting the tempDivisor variable to the largest power of 10 relevant to the input size (e.g., 10,000 for a 5-digit maximum). This serves as the divisor for the first iteration, targeting the most significant digit.
2. Digit Extraction: Using the BN_divide function, the decimalValue is divided by the current tempDivisor. This operation yields two key results:
    a. Quotient (tempQuotient): Represents the current digit at the position corresponding to the divisor (e.g., the ten-thousands place when dividing by 10,000).
    b. Remainder (tempRemainder): The remaining value after extracting the digit, which becomes the new decimalValue for subsequent iterations.
3. Handling Leading Zeros: The BN_hasLeadingZero flag ensures that leading zeros are skipped until a non-zero digit is encountered.
    a. If tempQuotient is 0 and BN_hasLeadingZero is still set (indicating that significant digits have not yet been processed), the current digit is skipped, and the function moves to the next iteration.
    b. Once a non-zero digit is detected, the flag is cleared (BN_hasLeadingZero = 0), and all subsequent digits, including zeros, are stored in the BN_formattedResult buffer.
4. Progressive Division: After processing the current digit, the tempDivisor is reduced by a factor of 10 (e.g., from 10,000 to 1,000) to target the next significant digit. This step ensures that the extraction process moves smoothly from the most significant to the least significant digit.
5. Iteration Until Completion: The function continues extracting digits by dividing the updated decimalValue with progressively smaller divisors until the smallest place (ones) is reached. At this point, all digits have been stored in the BN_formattedResult buffer.

The **BN_divide** function is a basic implementation of division using iterative subtraction. Its purpose is to extract a single digit from the BN_decimalValue by repeatedly subtracting the tempDivisor (the current place value, such as 10000, 1000, etc.) from BN_decimalValue until the remainder is smaller than the divisor. Here's how it works step by step:

1. Initialization: tempoQuotient set to 0. This variable will store the number of times the divisor can be subtracted from the current value, which corresponds to the digit in the current place value.
2. Iterative Subtraction: The loop begins by subtracting tempDivisor from BN_decimalValue. Each time the subtraction is successful (i.e., the result is non-negative), the quotient (tempQuotient) is incremented by 1. This process repeats until the value in BN_decimalValue becomes smaller than tempDivisor.
3. EndofSubtraction: Once the subtraction loop exits, BN_decimalValue contains the remainder for the next place value. tempQuotient contains the digit for the current place value.
4. Store the Extracted Digit: The digit (tempQuotient) is stored in BN_formattedResult, which is the buffer that collects all the digits for display. Move to the Next Place Value: After storing the digit, the program reduces tempDivisor (e.g., from 10000 to 1000) to target the next smaller place value.

After all digits have been formatted, the function jumps to Role 4's starting function,

Role4_startDisplay, signaling that the formatted data is ready for display.




```
// This file provides the functionality to convert a 16-bit signed binary number into its
// decimal representation, accounting for two's complement for negative numbers. It utilizes
// functions for binary arithmetic operations such as multiplication and exponentiation
// to perform this conversion.
//
// The conversion algorithm follows the formula:
//     decimalValue = Σ (R[i] * 2^(14-i))
// where R[i] represents the binary digits, and the exponent decrements from 14 to 0. The sign
// bit (R0) is checked and, if set, the two's complement of the number is computed to handle
// negative values.
//
// The file contains the following key components:
//
// 1. **CONVERT B2D FUNCTION**:
//    - Iteratively calculates the decimal value of the binary input by using the `MULT`
//      and `POW` functions.
//    - Computes each binary digit's contribution by multiplying the bit value by its corresponding
//      power of 2 (determined using the `POW` function).
```

```
//    - Accumulates the results in `decimalValue`.
//    - Adjusts the result for negative numbers using two's complement logic.
//
// 2. **SIGN CHECK AND CONVERSION**:
//    - After the conversion loop, checks the sign bit (R0).
//    - If the number is negative, the function flips all bits of `decimalValue`, adds 1, and
//      negates the result to complete the two's complement transformation.
//
// 3. **POW FUNCTION**:
//    - Computes `z = x^y`, where `x` is the base, and `y` is the exponent.
//    - Utilizes the `MULT` function to repeatedly multiply the base by itself in a loop, ensuring
//      modularity and code reuse.
//    - Stores the result in `POW_VAL`.
//
// 4. **MULT FUNCTION**:
//    - Computes the product of two numbers, `MULT_X` and `MULT_Y`, using repeated addition.
//    - Handles negative multipliers by checking the sign, adjusting accordingly, and restoring
//      the original values after computation.
//
// Pre-conditions:
//    - Input binary digits are stored in registers R0 to R15, where R0 is the sign bit and R1
//      to R15 hold the most significant to least significant bits.
//    - Variables for intermediate values (e.g., `POW_BASE`, `POW_EXP`, `MULT_X`, `MULT_Y`)
//      must be initialized appropriately.
//
// Post-conditions:
//    - The decimal equivalent of the input binary number is stored in `decimalValue`.
//    - For negative inputs, the result correctly reflects the two's complement interpretation.
//    - Registers and variables modified during computation are restored to their expected states.
//
// Call this program as follows:
//    - Provide binary input in registers R0 to R15.
//    - Invoke the `CONVERT B2D FUNCTION` to calculate the decimal value.
//    - Result is stored in `decimalValue` for further use or display.
//
```

(KL_outputArrow)

This function starts the drawing of an arrow onto the screen. It begins by printing the left part of the arrow: -, and then passes to the KL_outPutArrowContinue function that will finish the arrow.

It also uses *ge_currentColumn* and *ge_output_return*. The function increments *ge_currentColumn* to track the position for the output so that it is placed in the right position.

(KL_outputArrowContinue)

After the - part of the arrow is printed, this function completes the arrow by printing the greater than > sign. When combined these two functions make a -> on the screen. These functions rely on *ge_currentColumn* to handle the output and the *ge_output_return* to store the return address. The function increments *ge_currentColumn* again so that the > sign is spaced properly.

(KL_outputSign)

The purpose of this function is to display a sign in front of a number depending on whether the number is positive or negative. If the number is positive then it will display a (+) sign. If the number is negative then it will display a (-) sign. Before printing out the actual numbers this function will decide on which sign to show based on the number's most significant bit.

(KL_output_special_case)

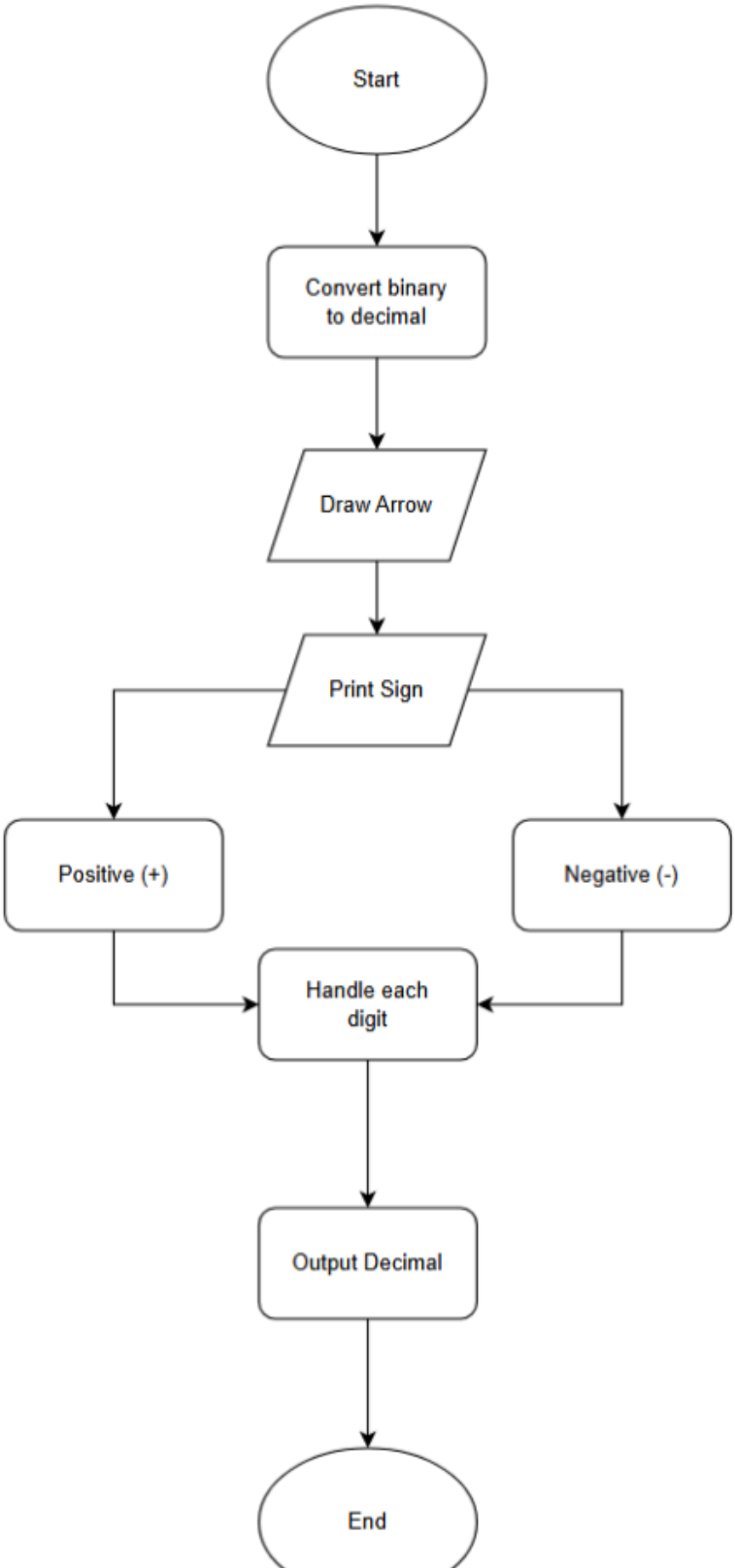
This function handles special cases when printing and makes sure that any unexpected cases are handled smoothly when producing the output. This makes sure that the program is able to handle unexpected values.

(KL_sign_return)

The *KL_sign_return* function is invoked after a special case
has been handled. This function also relies on *ge_currentColumn* and *ge_output_return* for the output and return address.

Flowchart

1. The binary number is converted into decimal.
2. An arrow symbol is drawn on the screen. This step uses *KL_outputArrow* & *KL_outputContinue*.
3. Then *KL_outputSign* checks whether the number is positive or negative. If the number is positive, then it will display a (+) sign. If it's negative, it will display a (-) sign.
4. The decimal numbers then get broken into individual digits (thousands, hundreds, tens, ones) for output.



5. The digits are then printed onto the screen.