ECSE 324 - Lab 2 Report - Marc Bretones - 260973002

Section 1 - Calculator.s:

Approach:

The calculator algorithm employs a modular approach to perform arithmetic operations such as addition, subtraction, and multiplication while handling negative numbers and overflow. I broke down the algorithm into subroutines to accomplish specific tasks, like polling push buttons, clearing the HEX display, writing values to the HEX display, and executing arithmetic operations. The main functions of the calculator are:

Polling Push Buttons: I implemented the PB_polling_ASM subroutine (lines 12-25) to continuously monitor the state of the push buttons by reading the edge capture register & resetting the value. Depending on which button is pressed, the subroutine branches to the corresponding arithmetic operation subroutine (multiplication, subtraction, or addition). I used conditional branching (CMP and BEQ) to manage the transitions between these operations.

Reading Slider Switches: In the read_slider_switches_ASM subroutine (lines 63-77), I read the values from the slider switches and store them in registers R6 and R7. These values serve as operands for the arithmetic operations.

Clearing HEX Displays: I designed the HEX_clear_ASM subroutine (lines 79-95) to clear the HEX display by writing zeros to each segment. This subroutine is called when the first push button is pressed.

Performing Arithmetic Operations:

Multiplication: In the multiplication subroutine (lines 26-33), I call read_slider_switches_ASM to fetch the operands, multiply them using the MUL instruction, store the result in register R7, and write the output to the HEX display with the HEX_write_ASM subroutine.

Subtraction: The subtraction subroutine (lines 40-53) calls read_slider_switches_ASM to obtain the operands, compares them, and performs subtraction accordingly. If the result is negative, I set the flag R10 to indicate a negative number. I store the result in register R7 and write it to the HEX display using the HEX_write_ASM subroutine.

Addition: For the addition subroutine (lines 54-62), I call read_slider_switches_ASM to get the operands, add them with the ADD instruction, store the result in register R7, and write the output to the HEX display using the HEX_write_ASM subroutine.

Handling Negative Numbers: When performing subtraction, if the result is negative, I set the flag R10 to indicate a negative number (lines 49-50). In the HEX_write_ASM subroutine (lines 193-198), I check the flag R10 and display a minus sign in the HEX5 display if the result is negative.

Writing Values to HEX Displays: The HEX_write_ASM subroutine (lines 143-213) takes the result of an arithmetic operation from register R7 and writes the output to the HEX displays. I update the displays sequentially based on the value of R8 (lines 146-161) and convert the numerical value to the corresponding HEX segment pattern using a series of conditional branches (lines 166-190). I call the HEX_flood_ASM subroutine (lines 103-111) to turn on the appropriate segments of the selected HEX displays.

Challenges:

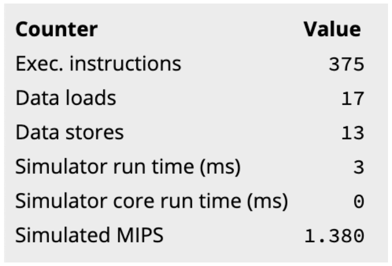
The main challenge I faced was efficiently handling negative numbers in +/- operations. To address this, I introduced a flag (R10) to indicate a negative result and displayed a minus sign on the HEX5 display. To handle situations where the addition or subtraction results in a negative number, I designed the addition and subtraction subroutines to call each other when dealing with negative numbers. Specifically I resolved this issue through dynamic linkage, by having the addition subroutine call subtraction_core when a negative number is involved, and the subtraction subroutine call addition_core when the result is negative.

Another main challenge was in displaying 'F' on the HEX displays. As one can only show a single digit to represent each number, I had to develop a method to handle multi-digit numbers. I addressed this by breaking down the result into individual digits and displaying them on the corresponding HEX displays (R8 is used to keep track of the current HEX display to be filled). Ensuring that the HEX display updates were accurate and synchronized with the arithmetic operations required testing and fine-tuning of the HEX_write_ASM subroutine. This involved comparing register values (R9) with the expected digit values and loading the appropriate constants to represent the digits on the HEX displays.

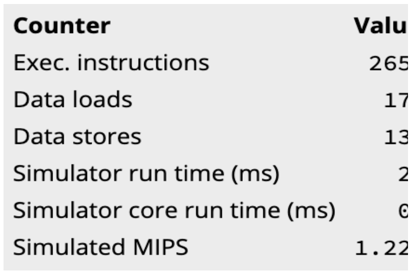
Shortcomings, Possible Improvements Performance Analysis:

Several shortcomings hinder the code's performance. One such shortcoming is the inefficient handling of segment value loading through a series of conditional branches. To improve this, a lookup table can be employed,

significantly reducing the number of comparisons and branches. Additionally, the absence of error handling and input validation could lead to unexpected behavior, which can be rectified by incorporating these features to ensure the code's reliability. More importantly, the current polling method employed in the algorithm is inefficient as it repeatedly checks for input changes in a busy loop, consuming valuable CPU cycles; instead, utilizing interrupts can significantly improve efficiency.
Figure 1 below demonstrates that for a single arithmetic iteration, and displaying the result on the HEX displays, the algorithm executes 375 instructions. This figure highlights potential inefficiencies in the current implementation, addressed by figure 2, reducing the same operation to 265 instructions by using a lookup table instead of iterating through all possible digits everytime the program must display one. By addressing such shortcomings and implementing improvements, I managed to substantially reduce the executed instructions, leading to optimized code.



Counter	Value
Exec. instructions	375
Data loads	17
Data stores	13
Simulator run time (ms)	3
Simulator core run time (ms)	0
Simulated MIPS	1.380



Counter	Valu
Exec. instructions	265
Data loads	17
Data stores	13
Simulator run time (ms)	2
Simulator core run time (ms)	0
Simulated MIPS	1.22

Figures 1 and 2: Original and Improved counter values for traversal of one arithmetic operation.

<u>Testing:</u>

<u>Methodology:</u> In order to ensure that each subroutine functioned as desired, I created unit tests for each of them. The latter were used for individual subroutines exclusively. With regards to testing the entire system, once I made sure individual subroutines work correctly, I performed integration tests to test the overall functionality of the calculator. This helped me identify issues that arose when different subroutines interact (ie. edge cases).

<u>Unit Testing:</u>

HEX_clear_ASM: (1) Test if all the HEX displays are turned off when this subroutine is called. (2) Test if the values of R12 and R10 are reset to zero.
HEX_write_ASM: (1) Test if the HEX displays show the correct hexadecimal value for all values from 0 to 15. (2) Test if the '-' sign is displayed correctly when the value of R10 is set to 1.
HEX_flood_ASM: (1) Test if the HEX displays are turned on correctly based on the value of R0.
read_slider_switches_ASM: (1) Test if the switches are read correctly for various combinations of switch values. (2) Test if the R6 and R7 registers are set correctly according to the switch values.
addition, subtraction, and multiplication subroutines: (1) Test each subroutine with a variety of input values, including boundary values (e.g., adding 0xFFFF and 1 in the addition subroutine to check for Overflow).

<u>Integration Testing:</u> Instead of a methodology, I opted for a list in integration testing:
- Test the calculator with various combinations of operations and input values.
- Test if the correct operation is performed based on the pushbutton input.
- Test if the calculator performs operations in the correct order (e.g., if addition is followed by subtraction, the correct result should be displayed).

Section 2 - Wackamole_polling.s:

<u>Approach:</u>

The Whack-a-Mole game has been implemented using a combination of subroutines to handle button presses and slider switches, display the mole on the HEX displays, and manage the timer.
<u>_start routine:</u> The main routine (Lines 34-41) initializes the timer count (R7 register) and sets up the timer using ARM_TIM_config_ASM subroutine. It then proceeds to call the mole and poll_timer subroutines, which are responsible for displaying the mole and handling button presses and timer events, respectively.
<u>Mole Display:</u> The mole subroutine (Lines 43-69) uses the ARM private timer counter value (loaded from PT_COUNTER_ADDR) to determine which of the four HEX displays to show the mole on. It masks the counter value with 0x3 (AND R6, R6, #3) to obtain an index between 0 and 3. Depending on the index, the MOLE pattern

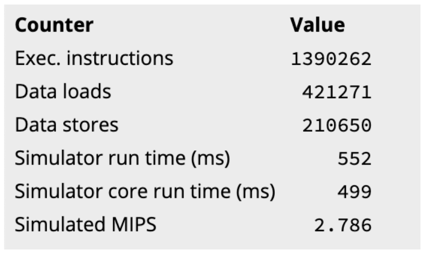
(0x5C) is stored in one of the four HEX displays using conditional store instructions (STREQB). The mole's position is stored in the R9 register for future reference.
Button and Slider Switch Polling: The poll_timer subroutine (Lines 71-196) monitors the button press and slider switch states. The button press events (start_game, stop_game, reset_game) are handled using conditional branches based on the values of the edge-capture register (PB_EDGE_CAPTURE_ADDR). The slider switches are read using the read_slider_switches_ASM subroutine, which checks the switch positions and branches to the switch_input subroutine if any switch is active. The switch_input subroutine verifies if the switch position corresponds to the current mole position (R9) and increments the hit counter (R8) if there's a match.
Timer Management: The timer (line 198-258), implemented with ARM_TIM_read_INT_ASM, ARM_TIM_clear_INT_ASM, and update_timer subroutines. The ARM_TIM_read_INT_ASM subroutine reads the timer's interrupt status register to check if a timer interrupt has occurred. If so, the mole subroutine is called again to move the mole. The ARM_TIM_clear_INT_ASM subroutine clears the timer interrupt, and the update_timer subroutine decrements the timer count (R7) and updates the HEX displays with remaining time using HEX_write_ASM subroutine.

Challenges:

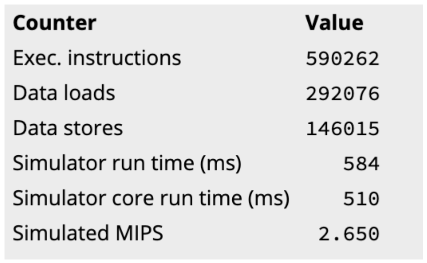
The most important problem I faced in this whack-a-mole implementation was randomizing the mole's position on the HEX displays. To address this, I used the value stored at PT_COUNTER_ADDR (R6) and performed a bitwise AND operation to obtain a seemingly random position between 0 and 3 (based on the last 2 digits of the private timer). While this approach provides some degree of randomness, it is not truly random and can be improved by using a more sophisticated randomization algorithm, yet is sufficient for a mere whack-a-mole game. Another challenge was handling simultaneous button inputs, such as starting, stopping, and resetting the game. My previous implementation only relied on polling the button edge capture register (PB_EDGE_CAPTURE_ADDR) to determine which button was pressed. This approach led to missed button inputs if multiple buttons were pressed simultaneously, so I added specific conditions to prevent an appearance if multiple buttons were activated.

Shortcomings, Possible Improvements Performance Analysis

My code employs a series of conditional branches for loading segment values, which is inefficient. This can be improved by using a lookup table to store segment values, significantly reducing the number of comparisons and branches.
The code uses a busy loop for checking input changes, consuming valuable CPU cycles. Utilizing interrupts instead of the polling method would greatly improve the efficiency of the algorithm.
By addressing these shortcomings and implementing the suggested improvements, the performance of the code can be significantly optimized. For instance, the original implementation required 1,390,262 instructions executed for a runtime of 500ms. By employing a lookup table instead of iterating through all possible digits as well as having all the polling done in only one subroutine, the same operation was reduced to 590,262 instructions over a similar time period (figures 3 & 4). This optimization results in a significant reduction in the number of executed instructions from.



Counter	Value
Exec. instructions	1390262
Data loads	421271
Data stores	210650
Simulator run time (ms)	552
Simulator core run time (ms)	499
Simulated MIPS	2.786



Counter	Value
Exec. instructions	590262
Data loads	292076
Data stores	146015
Simulator run time (ms)	584
Simulator core run time (ms)	510
Simulated MIPS	2.650

Figures 3 and 4: Original and Improved counter values for running 500ms of Whack-A-Mole.

Testing:

Methodology: To ensure the proper functioning of the Whack-a-Mole game, we employed a combination of unit tests and integration tests. Unit tests were created for each subroutine to verify their individual behavior, while integration tests were designed to ensure the overall functionality of the game, particularly focusing on edge cases and the interaction between subroutines.
Unit Testing:
mole: Test if the mole appears in the correct HEX display, based on the value of R6. Also, test if the mole reappears in the correct position after disappearing.
HEX_clear_ASM: Test if all the HEX displays are cleared when this subroutine is called.

poll_timer: Test if the timer is polled correctly and branches to the appropriate subroutines based on the timer's status and button presses.
read_slider_switches_ASM: Test if the switches are read correctly for various combinations of switch values and if R10 is set correctly according to the switch values.
switch_input and mole_disappear: Test if the mole disappears when the correct switch is selected and if R8 is incremented accordingly. Also, test if the mole doesn't disappear when the incorrect switch is selected.
update_timer: Test if the timer count is decremented correctly and displayed on the HEX displays.
start_game, stop_game, reset_game: Test if the game starts, stops, and resets correctly based on the corresponding button presses.
ARM_TIM_config_ASM, ARM_TIM_read_INT_ASM, ARM_TIM_clear_INT_ASM: Test if the timer configurations, reading, and clearing of interrupts work as expected.
Integration Testing: Instead of a methodology, I opted for a list in integration testing:

- Test the Whack-a-Mole game with various combinations of button presses, slider switch values, and timer countdowns to ensure proper functionality.
- Test if the game starts, stops, and resets correctly based on the corresponding button presses and if the mole disappears and reappears in the correct positions.
- Test if the timer countdown updates correctly on the HEX displays and if the game behaves as expected when the timer reaches zero.
- Test if the scoring system (R8) increments correctly when the correct switch is selected and if it remains unchanged when the incorrect switch is selected.

Section 3 - Wackamole_interrupts.s:

To avoid repetition, as sections 2 & 3 are grandly similar, I will only cover the differences with the latter section, please note most subroutines apart from the polling ones are relevant to this section. With regards to the testing, the tests are exactly the same but test interrupt_handler instead of its polling counterparts. For this reason: for testing, challenges and approach, most of the information will be found in the section above.

Approach:

Button and Slider Switch Polling: Button and Slider Switch Interrupts: The interrupt_handler subroutine (Lines 82-152) manages the button press and slider switch states using interrupts instead of polling. The button press events (start_game, stop_game, reset_game) are handled by configuring the interrupts based on the edge-capture register (PB_EDGE_CAPTURE_ADDR) and attaching the corresponding interrupt service routines (ISRs) for each event.
The slider switches are managed using the read_slider_switches_ASM subroutine, which is called by an ISR when a switch change interrupt is detected. This subroutine checks the switch positions and calls the switch_input subroutine if any switch is active. The switch_input subroutine verifies if the switch position corresponds to the current mole position (R9) and increments the hit counter (R8) if there's a match.

Challenges:

A challenge was designing and implementing the interrupt service routines (ISRs) for each button event and the slider switch change event. Ensuring that the ISRs were efficient and executed quickly was crucial to prevent the system from missing any input events or becoming unresponsive. Finally, integrating the read_slider_switches_ASM and switch_input subroutines with the interrupt-driven approach took some time. I had to modify these routines to work with ISRs instead of polling, and I needed to test and debug the new implementation thoroughly to ensure proper game functionality and responsiveness.

Shortcomings, Possible Improvements Performance Analysis:

A possible improvement is optimizing the interrupt service routines (ISRs) further to reduce latency and improve system responsiveness. This could be achieved by minimizing the execution time of the ISRs and using interrupt priorities to ensure that more critical tasks are not delayed by lower-priority events.
Lastly, a performance analysis could be performed to identify any bottlenecks or areas where the game's performance could be enhanced. However, my code only works line by line as I had trouble integrating the timer with the interrupt implementation of the game; as such, I was unable to show the Counter values with executed instructions over time to attempt improving my implementation.