GAZI UNIVERSITY
FACULTY OF ENGINEERING
DEPARTMENT OF
ELECTRICAL AND ELECTRONICS
ENGINEERING

Experiment #6

Ahmet Emin Karakaya
191110046

The concept of a physical keypad and ways to get rid of it.

Physical keypads are input devices commonly used in various electronic devices. When a key is pressed, a phenomenon called mechanical bouncing can occur, leading to undesired multiple key activations. This issue presents a common challenge in achieving reliable operation of keypad input systems.

Debouncing is a method used to prevent or reduce the effect of mechanical bouncing in keypads. Mechanical bouncing refers to the rapid opening and closing of contacts when a key is pressed. This can lead to the electronic circuit incorrectly detecting multiple key activations.

Debounce Methods:

1. **Software Debouncing:** In this method, signal changes occurring when a key is pressed and released are detected by software, and time delays are employed to filter out undesired bounces. A simple approach is to ignore any other key activations for a certain period of time after a key is pressed.

2. **Hardware Debouncing:** This method aims to prevent keypad bouncing using physical components. Integrated circuits or capacitors can be employed in the keypad circuitry to suppress or dampen the bouncing effect when a key is pressed.

3. **Mechanical Design Improvements:** The physical design of keypads can be optimized to reduce the effect of mechanical bouncing. This may involve using more durable contacts or employing springs that prevent bouncing. Mechanical design improvements aim to address the debounce issue at its source.

In conclusion, debounce is a significant concern in physical keypads, and addressing this issue is crucial for achieving reliable and accurate key input.

Code review and explanation

lab6.s file

In Figure 1, a waiting time has been defined for the buzzer to sound at certain intervals. different durations are defined. The reason for this is that when the code enters the interrupt, it is requested to ring with different waiting times while inside the different normal code. In addition, delay_address was defined to save these defined values to an address.

```
50
51  high_frequency_value EQU 0x4C0000    ; high
52  low_frequency_value  EQU 0x0C0000    ; low_
53  delay_address  EQU 0x20000400        ; Dela
54
```

Figure 1

Figure 2 shows the part that will run when the code interrupts. If I talk generally, the buzzer will ring and extinguish the buzzer according to the low_frequency_value value. At first, that waiting time is recorded in the R2 register and writing is done to the delay_address address. Buzzer is on bit 6 of port A. That address is reached and it activates the buzzer by making the relevant bit 1. Then it waits for the low_frequency_value value and continues from where it left off. Finally, to deactivate the buzzer, the relevant bit is cleared and the buzzer is deactivated. After waiting for a certain time, the flag is cleared for the next interrupts.

```
56      AREA INTRPT, CODE
57          ALIGN
58          ;EXTERN checkblink
59
60          EXPORT ISRD
61  ISRD    PROC
62
63          NOP
64
65          LDR R2, =low_frequency_value    ; Low_frequency_value wri
66          LDR R3, =delay_address          ; Delay_address written t
67          STR R2, [R3]                    ; Low_frequency_value is
68
69          LDR R1, =gpioAbase
70          ADD R1, #0x100                  ; R1 stores address of da
71          LDR R0, [R1]
72
73          ORR R0, R0, #0x01
74          STR R0, [R1]                    ; Makes Buzzer's state lo
75          BL delay_isrd
76
77          LDR R0, [R1]
78
79          BIC R0, R0, #0x01
80          STR R0, [R1]                    ; Makes Buzzer's state lo
81          BL delay_isrd
82
83  delay_isrd
84          LDR R4, [R3]            ; The value at address R3, which
85
86  wait_isrd
87          SUB R4, R4, #1          ; Delay time reduced by 1
88          CMP R4, #0              ; Comparing whether the delay tim
89          BNE wait_isrd           ; If not 0 it went back to the be
90          BX   LR
```

Figure 2

In Figure 3, flag clearing is required for the next interrupt situations. Here, the flag has been cleared. After the base address of the D port is taken, the offset value of the GPIO Interrupt Clear register is added and the value there is written to the R1 register. Then, the relevant bit is set to 1 and the flag is cleared.

```
91
92          ;clear flag for further interrupts
93          LDR R0,=gpioDbase
94          ADD R0,R0, #gpioIcr
95          LDR R1,[R0]
96          ORR R1,R1,#1
97          STR R1,[R0]
98
99          BX LR
100         ENDP
101     ALIGN
102
```

Figure 3

Figure 4 shows the main part of our code. At first, he went to the intializegpio branch with the BL directive and did some operations there. Then the code continued where it left off. Here, he actually did what he did when he entered the interrupt, but with a slight difference. The time between the buzzer being active and deactivated is different. Same process outside.

```
__main

        NOP
        BL initializegpio


high_frequency_ringing
                LDR R2, =high_frequency_value  ; High_frequency_value writt
                LDR R3, =delay_address         ; Delay_address written to R
                STR R2, [R3]                   ; High_frequency_value is wr

                LDR R1, =gpioAbase
                ADD R1, #0x100                 ; R1 stores address of data
                LDR R0, [R1]

                ORR R0, R0, #0x01
                STR R0, [R1]                   ; Makes Buzzer's state logic
                BL delay

                LDR R0, [R1]

                BIC R0, R0, #0x01
                STR R0, [R1]                   ; Makes Buzzer's state logic
                BL delay

                B high_frequency_ringing       ;The code went back to the h

delay
                LDR R4, [R3]                   ; The value at address R3, w

wait
                SUB R4, R4, #1                 ; Delay time reduced by 1
                CMP R4, #0                     ; Comparing whether the dela
                BNE wait                       ; If not 0 it went back to t
                BX   LR
```

Figure 4

init.s file

In Figure 5, A,d and NVIC base addresses are defined.

```
; GPIO base addresses of port A,D and NVIC
gpioDbase    EQU      0x4005B000 ; inp
gpioAbase    EQU      0x40058000 ; out
nvicBase     EQU      0xE000E000 ; nvic

```

Figure 5

In Figure 6, we need to activate NVIC for port D. Therefore, the offset value of the relevant register is defined.

```
52  rcgcgpio    EQU     0x400FE608
53
54  nvicEN0     EQU     0x100   ; offset value of NVICEN0 register
55
56
```

Figure 6

In Figure 7, the initialize gpio operation in the previous examples was performed.

```
61  initializegpio
62                      ; enable clck for ports A and D
63                      LDR R1, =rcgcgpio
64                      LDR R0, [R1]
65                      ORR R0, R0, #0x09
66                      STR R0, [R1]
67                      NOP
68                      NOP
69                      NOP
70
71                      ; out port A
72                      LDR R1, =gpioAbase
73                      ADD R1, R1, #gpioDir
74                      LDR R0, [R1]
75                      ORR R0, R0, #0x3F
76                      STR R0, [R1]
77
78                      ; afsel
79                      LDR R1, =gpioDbase
80                      ADD R1, R1, #gpioAfsel
81                      LDR R0, [R1]
82                      BIC R0, #0xFF
83                      STR R0, [R1]
84
85                      LDR R1, =gpioAbase
86                      ADD R1, R1, #gpioAfsel
87                      LDR R0, [R1]
88                      BIC R0, #0xFF
89                      STR R0, [R1]
```

Figure 7

In Figure 8, these processes continued, but some differences were made. The interrupt mask register is set for the first pin of port D. So, interrupts that are generated by the corresponding pin to be sent to the interrupt controller on the combined interrupt signal. Then the GPIO Interrupt Event register is set. So, configured to detect rising edges or high levels based on the corresponding bit value in the GPIO Interrupt Sense (GPIOIS) register. Finally, NVIC was activated for port D.

```
 91                          ; den 1 for ports
 92                          LDR R1, =gpioDbase
 93                          ADD R1, R1, #gpioDen
 94                          LDR R0,[R1]
 95                          ORR R0, R0, #0xFF
 96                          STR R0, [R1]
 97
 98                          LDR R1, =gpioAbase
 99                          ADD R1, R1, #gpioDen
100                          LDR R0,[R1]
101                          ORR R0, R0, #0xFF
102                          STR R0, [R1]
103
104                          ;interrupt mask for portD first pin
105                          LDR R1,=gpioDbase
106                          ADD R1, R1, #gpioIm
107                          LDR R0,[R1]
108                          ORR R0, R0, #0x01
109                          STR R0, [R1]
110
111                          LDR R1,=gpioDbase
112                          ADD R1, R1, #gpioIev
113                          LDR R0,[R1]
114                          ORR R0, R0, #0x01
115                          STR R0, [R1]
116
117                          ;NVIC for portD
118                          LDR R1, =nvicBase
119                          ADD R1, R1, #nvicEN0
120                          LDR R0, [R1]
121                          ORR R0, R0, #0x8 ; for port D
122                          STR R0, [R1]
123
```

Figure 8

startup_TM4C129.s file

Here, the GPIOD_Handler part is seen. In this way, we manage to go to the ISRD area when the interrupt comes.

```
279  GPIOD_Handler    PROC
280                   EXPORT  GPIOD_Handler [WEAK]
281                   IMPORT ISRD
282                   B  ISRD
283                   ENDP
284
```

Figure 9

Simulation outputs

We learned that the interrupt part is provided as hardware by asking our teacher. Therefore, we cannot observe that part in the simulation. But since we can observe the running part of the code in the simulation, you can find them. In addition, the waiting time was set to 3 to be able to observe it in the simulation.

In Figure 10, the code entered the high_frequency_ringing branch and activated the buzzer after performing the initial assignments.

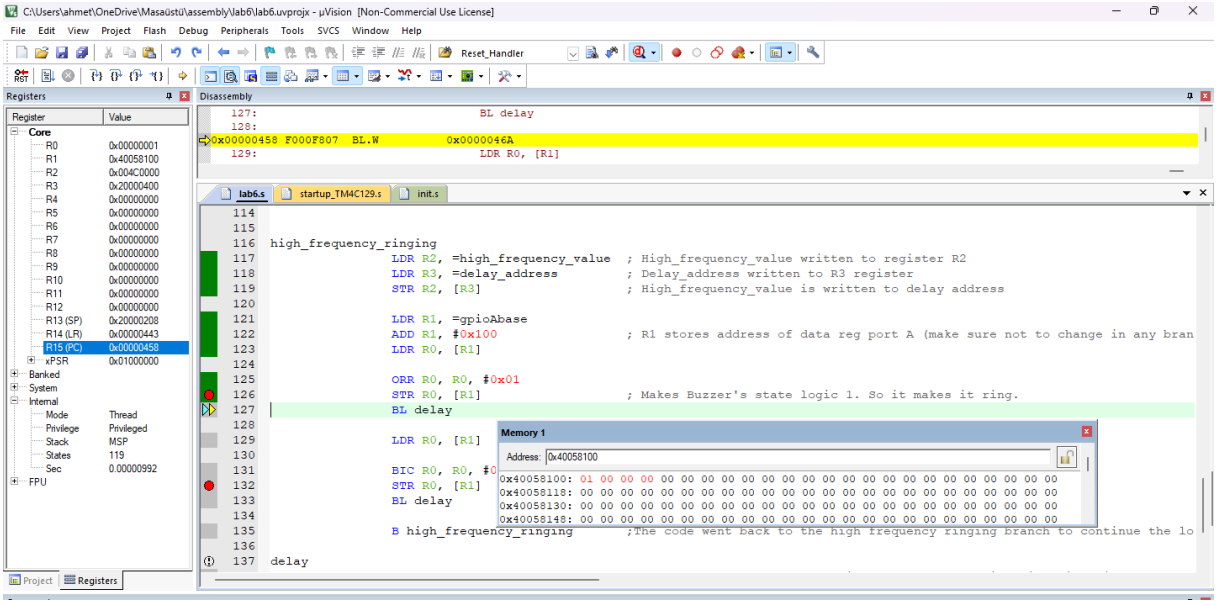

Figure 10

In Figure 11, the code went to the delay branch and wrote the corresponding wait time to the R4 register.

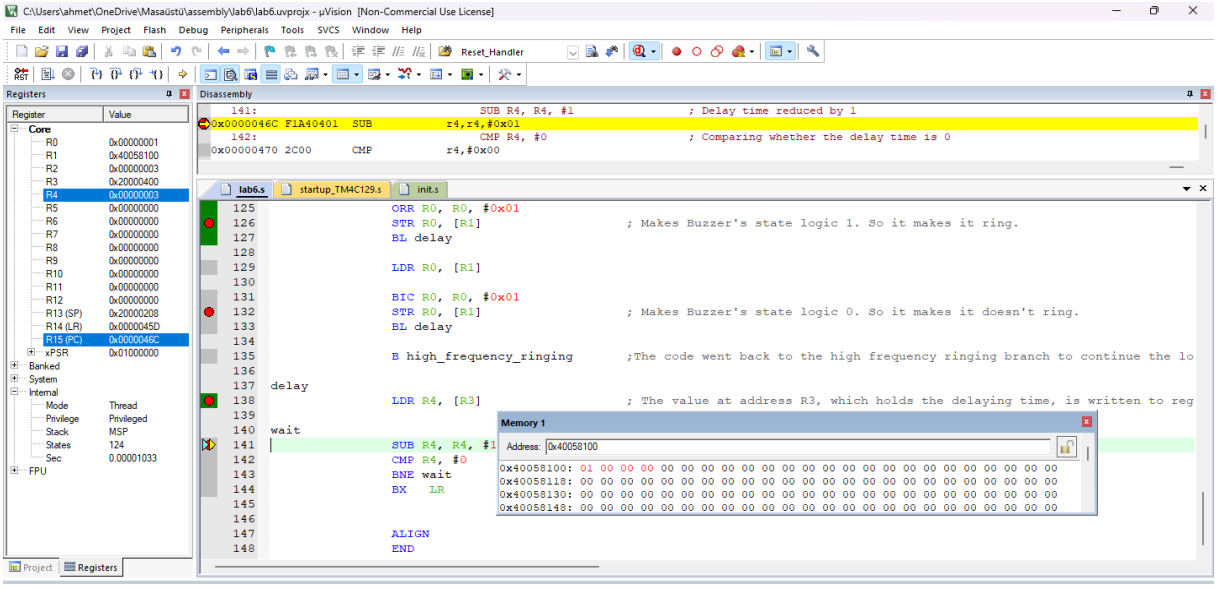

Figure 11

In figure 12, the code reduced the wait time by one and checked for 0. If it's not 0, it's back in the wait branch.

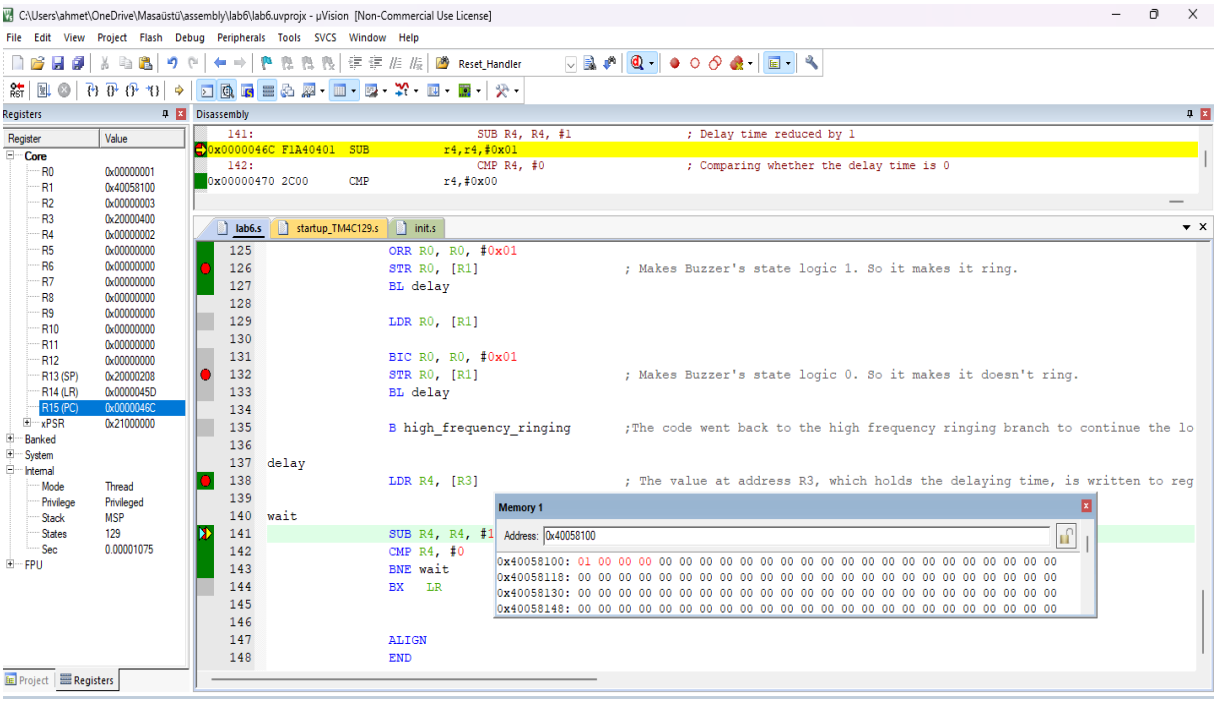

Figure 12

In Figure 13, the high_frequency_value became 1.

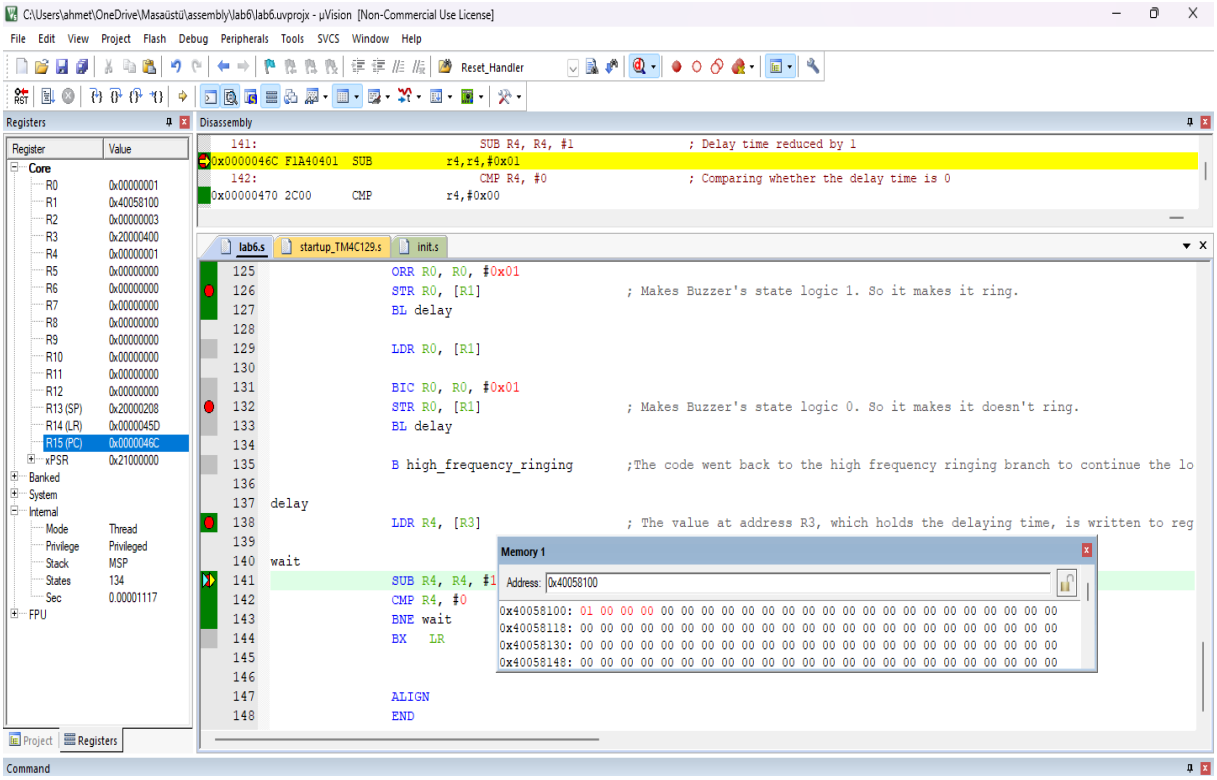

Figure 13

In Figure 14, the high_frequency_value became 0. The code went where it left off and deactivated the buzzer.

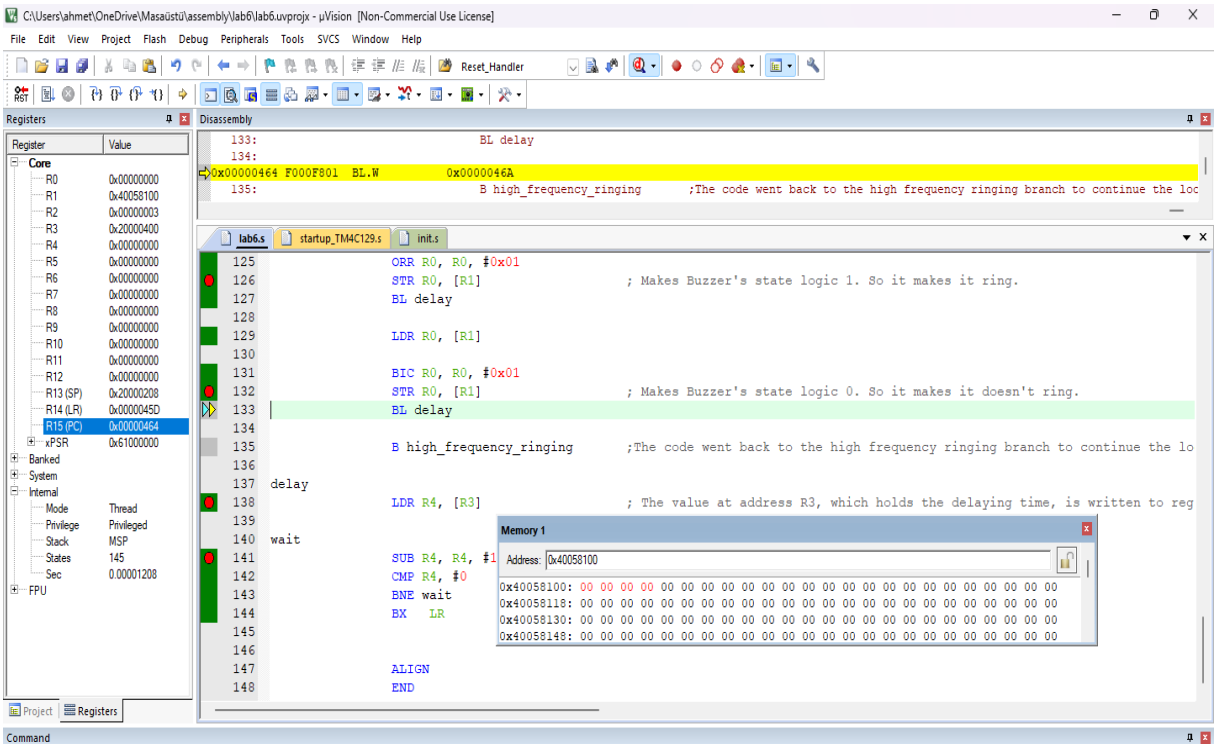

Figure 14

In Figure 15, the code went to the delay branch. Writes the high_frequency_value to the R4 register.

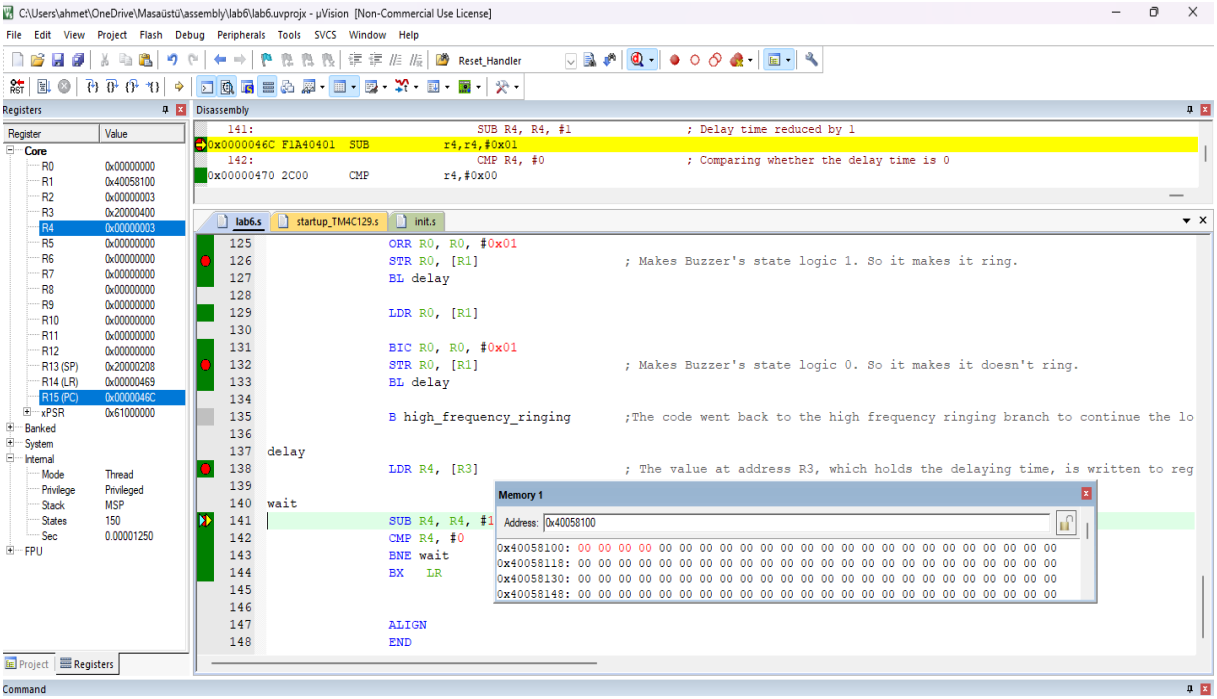

Figure 15

In Figure 16, the value in register R4 is 2.

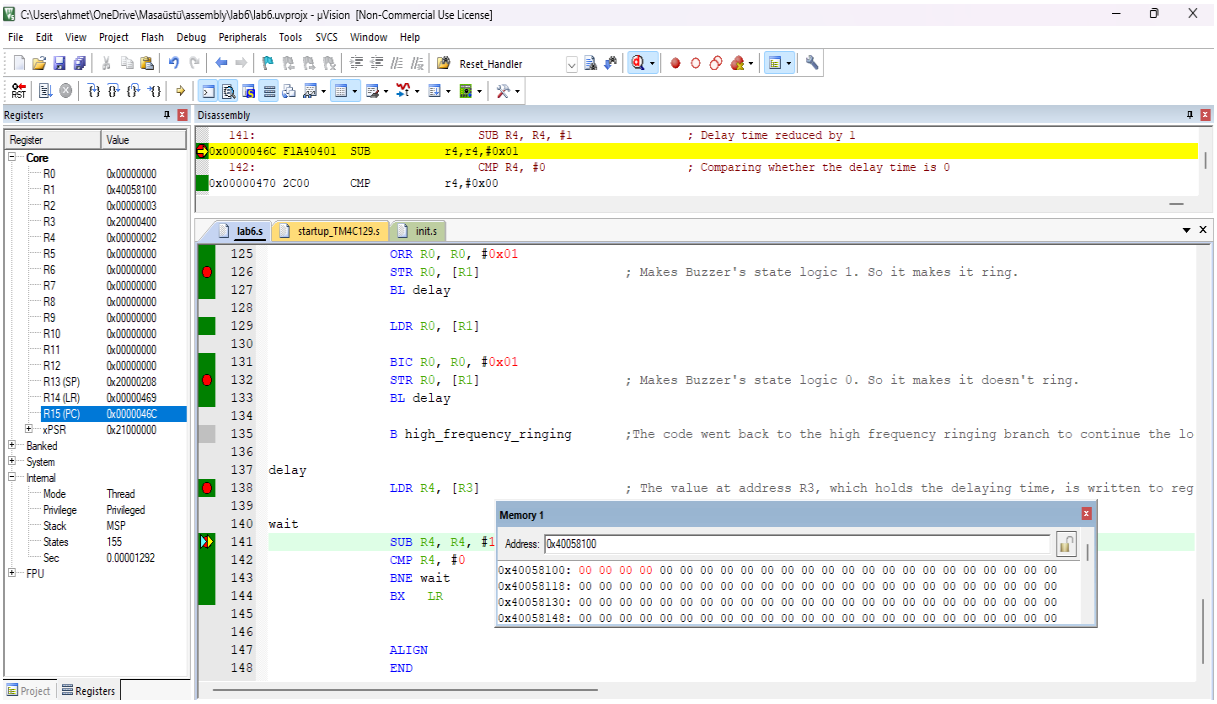

Figure 16

In Figure 17, the value in register R4 is 2.

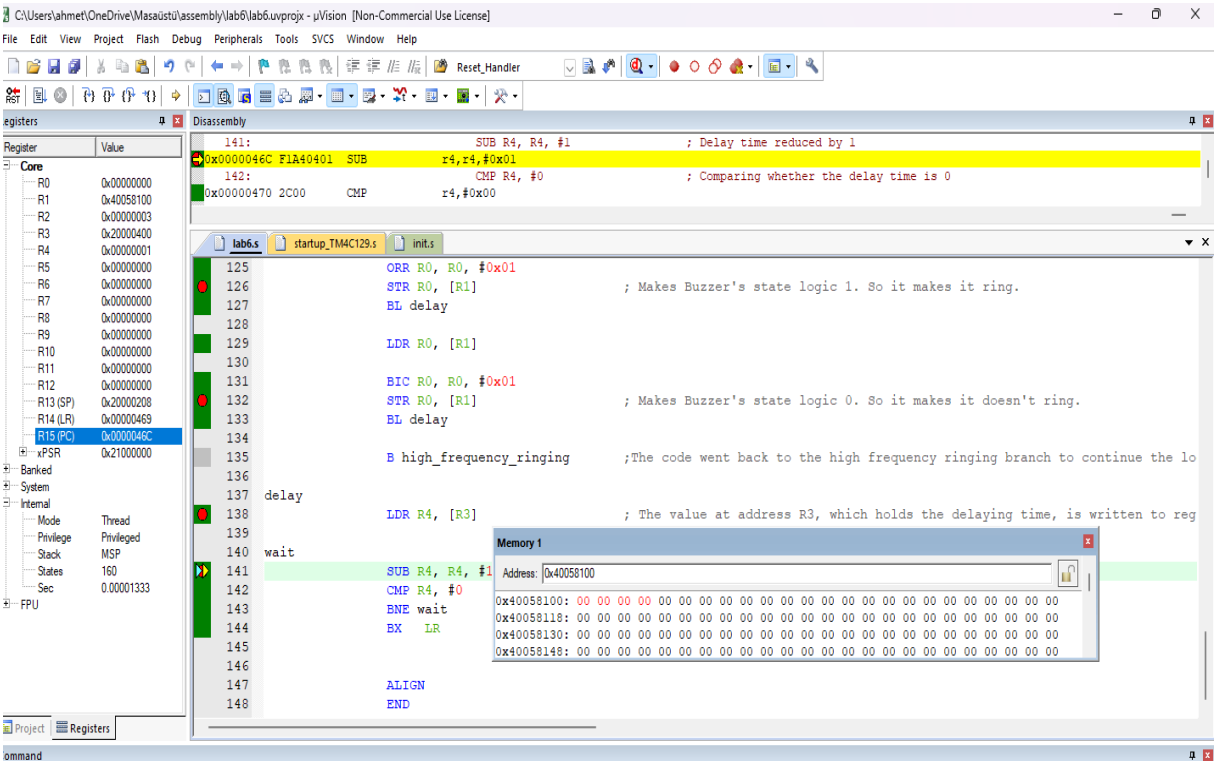

Figure 17

In Figure 18, the code reverted back to the high_frequency_ringing branch and did the same things.

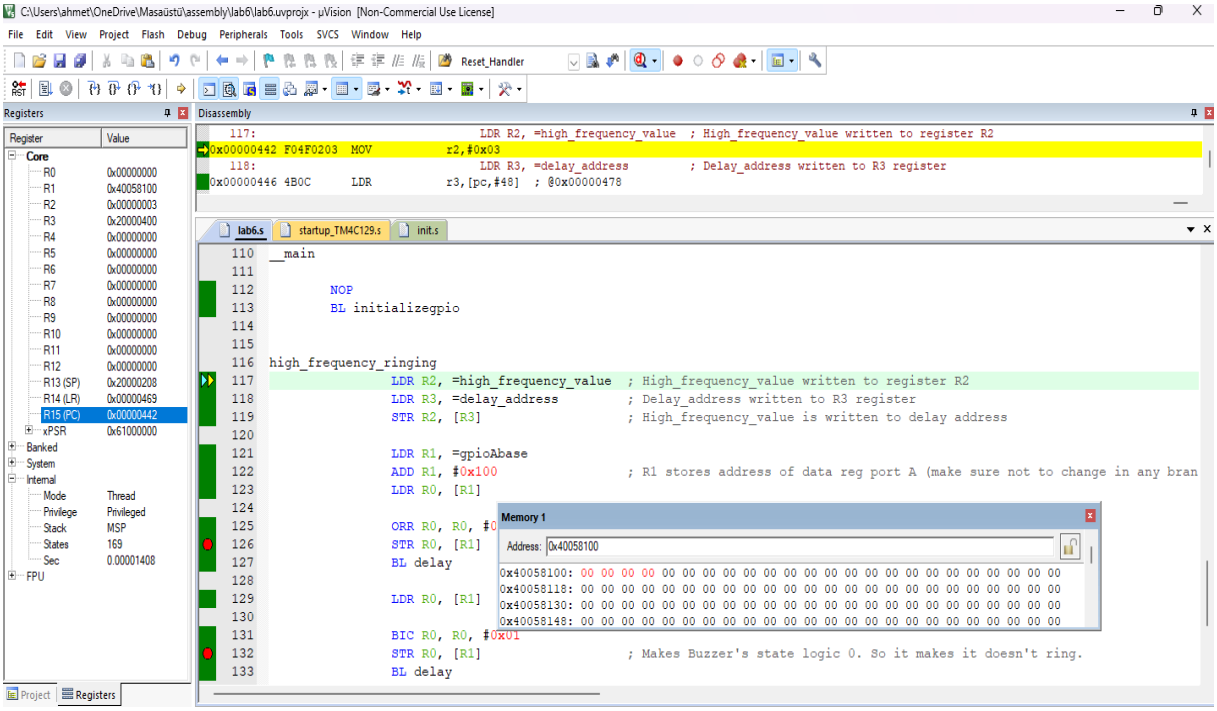

Figure 18