|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| logowydzialu | Instytut Informatyki Politechniki Śląskiej  Zespół Mikroinformatyki i Teorii Automatów Cyfrowych  **Laboratorium SMiW** | | logoii | |
| **Rok akademicki** | **Rodzaj studiów\*: SSI/NSI/NSM** | **Numer ćwiczenia:** | **Grupa** | **Sekcja** |
| **2015/2016** | **SSI** | **7** | **4** | **2** |
| **Data i godzina planowana ćwiczenia:**  dd/mm/rrrr - gg:mm | **20/10/2015 8:30** | **Prowadzący**:  JP/KT/GD/GB | **JP** | |
| **Data i godzina wykonania ćwiczenia:**  dd/mm/rrrr - gg:mm | **20/10/2015 8:30** |
| ***Sprawozdanie*** | | | | |
| **Temat ćwiczenia:**  Mikrokontrolery z serii AVR | | | | |
| **Skład sekcji:** | 1. Dominika Błasiak 2. Aleksandra Chronowska | | | |

# Treść zadania

Do portu A mikrokontrolera AVR Atmega256 o zegarze taktowanym z częstotliwością 4 MHz podłączono 8 przycisków podpiętych do masy.

Do portu B podłączono 8 diod świecących przez tranzystor do zasilania.

W dowolnym miejscu w pamięci programu umieszczono tablice - "tab\_ROM" o nieznanej długości i zakończonej słowem 0 x FF oraz tablice o znanej długości "tab\_RAM".

Program powinien poprawnie działać w przedziale 0-1024 bajtów.

Program ma wyświetlać zawartość tablicy” tab\_ROM” za pomocą diod po naciśnięciu pojedynczego przycisku.

Założenie: Po kliknięciu dowolnego przycisku na diodach zostaje wyświetlony pierwszy bajt tablicy. Wczytanie i wyświetlenie kolejnej wartości tablicy odbywa się po puszczeniu danego przycisku i ponownym kliknięciu dowolnie wybranego przycisku (kliknięcie przycisku -> wyświetlenie na diodach -> puszczenie przycisku -> kliknięcie przycisku -> wyświetlenie na diodach -> puszczenie przycisku itd. dopóki program nie napotka 0xFF). Program kończy działanie, gdy napotka 0xFF.

# Kod programu – język wysokiego poziomu C

|  |
| --- |
| 1. #include <avr/io.h> 2. #include <avr/pgmspace.h> 3. #define ROM\_SIZE 5 4. #define *F\_CPU* 4000000UL //clock = 4MHz 5. //tablica umieszczona gdzieś w pamięci ROM 6. const char ROMTAB[ROM\_SIZE] PROGMEM= {1,2,3,4,0xFF}; 7. //inicjalizacja portów 8. void initPorts() 9. { 10. DDRA = 0x00; //port a jako wejscie przez rezystor podciagajacy (podłączamy do niego klawisze) 11. PORTA = 0xFF; //PORTA podpinamy do masy 12. DDRB = 0xFF; //port b jako wyjscie (podłączamy do niego diody) 13. PORTB = 0xFF;//zapewniamy napiecie wysokie, aby diody nie swiecily 14. } 15. void printROMTAB() 16. { 17. int i=0; 18. //wczytaj pierwszy element z tablicy 19. unsigned char dataBit = pgm\_read\_byte(&(ROMTAB)); 20. while(1) 21. { 22. //czeka na wciśnięcie klawisza 23. while(PINA == 0xFF); 24. //zapalenie wybranej diody odbywa sie przez wystawienie 0 na odpowiednim pinie portu B 25. PORTB = (dataBit) ^ 0xFF; //0 xor 1 = 1 ; 1 xor 1 = 0; 26. // jeśli wczytany element jest równy 0xFF wtedy kończy się wykonywanie pętli 27. if(dataBit == 0xFF) 28. break; 29. i++; 30. //wczytanie kolejnego elementu 31. dataBit = pgm\_read\_byte(&(ROMTAB[i])); 32. //czekanie na puszczenie klawisza 33. while(PINA != 0xFF); 34. } 35. } 36. int main(void) 37. { 38. initPorts(); 39. printROMTAB(); 40. } |

# Kod programu – język niskiego poziomu ASM

|  |
| --- |
| 1. .nolist 2. .include "m2560def.inc" 3. .list 4. .ESEG ; EEPROM memory segment 5. .DSEG ; SRAM memory.segment 6. .ORG 0x100; may be omitted this is default value 7. .CSEG ; CODE Program memory. Remember that it is "word" address space 8. .org 0x0000 9. jmp RESET ; Reset Handler 10. RESET: 11. ldi r20, 0xFF ; Maska. 12. ldi r16, byte3(ROMTAB<<1) ; Wskazanie odpowiedniego bloku pamięci (byte3() - trzeci bajt). 13. out RAMPZ, r16 ; Wpisanie adresu do rejestru RAMPZ. 14. ; RAMPZ (RAM Page Z Select Register) Rejestr ten określa numer strony pamięci programu. 15. ldi ZH, high(ROMTAB<<1) ; Załadowanie starszej części adresu do rejestru Z. 16. ldi ZL, low(ROMTAB<<1) ; Załadowanie młodszej części adresu do rejestru Z. 17. ; Wskaźnik Z potrafi zaadresować jedynie 64K pamięci. 19. cli ; Wyłączenie obsługi przerwań. 21. ; Inicjalizacja portów. 22. ; Port A 23. ldi r16, 0x00 ; 24. out DDRA,r16 ; Ustawienie portu A jako wejście. 25. ldi r16,0xFF ; 26. out PORTA,r16 ; Wejścia typu PULL-UP. 27. ; Port B 28. ldi r16, 0xFF ; 29. out DDRB, r16 ; Ustawienie portu B jako wyjście. 30. ldi r16, 0xFF ; 31. out PORTB, r16 ; Wygaszenie diod (port B w stanie wysokim). 32. ; Koniec inicjalizacji portów. 34. sei ; Włączenie obsługi przerwań. 35. ; Oczekiwanie na wciśnięty przycisk (dowolny). 36. ; r16 - aktualny stan portu B. 37. Czekanie\_na\_przycisk: 38. in r16, PINA ; Wczytanie stanu pinów portu A do rejestru r16. 39. cpi r16, 0xFF ; Sprawdzamy czy wciśnięty został jakiś przycisk. 40. breq Czekanie\_na\_przycisk ; Gdy nie, czekamy na wciśnięty przycisk. 42. elpm R17, z 43. ; Odczyt kolejnych wartości z ROMTAB. 44. ; r16 - odczytana wartość tablicy. 45. Odczyt: 46. elpm r16, z+ ; Odczyt wartości oraz inkrementacja Z. 47. ; Wykorzystanie adresowania RAMPZ, ZH ,ZL wymaga użycia instrukcji 48. elpm. 49. cpi r16, 0xFF ; Sprawdź, czy wczytany element to FFh. 50. breq Wyswietl\_ostatni\_bajt ; Gdy tak, skocz do end (wyświetli zawartośc ostatniego bajta i zakończy program) 51. eor r16, r20 52. out PORTB, r16 ; Gdy nie, wyświetl wartość na diodach. 53. Czekanie\_na\_odklikniecie: 54. in r16, PINA 55. cpi r16,0xFF 56. brne Czekanie\_na\_odklikniecie 57. rjmp Czekanie\_na\_przycisk 58. Wyswietl\_ostatni\_bajt: ;wyświetlenie zawartości ostatniego bajta na diodach, czyli 0xFF 59. eor r16, r20 60. out PORTB, r16 ; Wyświetlenie 0xFF na diodach 61. ; Zakończenie programu. 62. End: 63. rjmp END 64. ; .org 0x8000 -> "wyrównanie słowowe" (WORD-aligned) = 0x10000 -> "wyrównanie bajtowe" (BYTE-aligned). 65. .org 0x8000 66. ROMTAB: .db 0x01, 0x02, 0x03, 0x04, 0x05, 0x00, 0x06, 0xFF 67. .EXIT |

# Porównanie czasowe i pamięciowe kodu

Porównywanie czasów inicjalizacji:

Porównanie zajętości pamięci:

# Wnioski

Laboratorium miało na celu zapoznanie z programowaniem mikrokontrolerów z rodziny AVR. Na zajęciach naszym zadaniem było napisanie kodu programu w środowisku Atmel Studio zarówno w języku wysokiego poziomu – C jak i w języku asemblerowym.

Niestety w naszym przypadku zmierzenie czasu wykonania programu było bardzo trudne. Czas zależał głównie od tego jak szybko użytkownik kliknie bądź odkliknie wybrany przycisk, dlatego nasze pomiary nie były wiarygodne. Mimo wszystko podczas testowania można było zauważyć, że czas wykonywania poleceń w asemblerze jest znacznie krótszy niż w C.

Język asemblerowy wykazał się również optymalniejszy w przypadku wykorzystywania pamięci.

Można stwierdzić, że jakość kodu napisanego językiem niskopoziomowym wymaga obszerniejszego zakresu umiejętności i większego wysiłku od programisty.

Jednak program w języku wysokiego poziomu jest znacznie czytelniejszy, łatwiejszy do zaimplementowania. Dodatkową zaletą są również, możliwe do wykorzystania, gotowe biblioteki.