Instytut Informatyki Politechniki Śląskiej

Zespół Mikroinformatyki i Teorii Automatów Cyfrowych

Laboratorium SMiW

Rok akademicki	Rodzaj studiów*: SSI/NSI/NSM	Numer ćwiczenia:	Grupa	Sekcja
2015/2016	**SSI**	**18**	**5**	**3**

Data i godzina planowana ćwiczenia: dd/mm/rrrr - gg:mm	**20/10/2015 - 13:15**	**Prowadzący**: OA/JP/KT/GD/BSz/GB	**JP**
Data i godzina wykonania ćwiczenia: dd/mm/rrrr - gg:mm	**20/10/2015 - 13:15**		

Sprawozdanie

Temat ćwiczenia:

Mikrokontrolery serii AVR

Skład sekcji:	1. Michał Lytek 2. Jakub Świerczek

Zadanie

Założenia:

- ATMega2560 16MHz
- do portu A podłączone przyciski zwierające do masy
- do portu B podłączone diody przez rezystor do masy (1 – świeci)
- w eeprom znajduje się tablica bajtów o nieokreślonej długości, zakończona dwoma zerami

Naszym zadaniem było napisać program, który powodował, że po naciśnięciu dowolnego przycisku na diodach z portu A pojawiał się bajt z tablicy pamięci EEPROM.

Każdorazowe naciśnięcie przycisku miało powodować wyświetlenie się kolejnego bajta z pamięci. W momencie dojścia do końca tablicy, program przestaje wypisywać jej zawartość dalej.

Zadanie to mieliśmy zrealizować w języku asemblera oraz w C.

Wykonanie

Nasz program czeka na naciśniecie dowolnego przycisku i następnie odczytuje bajt z pamięci EEPROM i wtedy wyświetla go na diodach. Następnie czeka na puszczenie przycisku (bez tego w ułamek sekundy wymrugałby całą tablicę na diody zamiast po bajcie), a potem wraca do czekania na naciśnięcie przycisku.

Jeżeli nie jest naciśnięty żaden przycisk, to program czeka na jego naciśnięcie, a na diodach wyświetla się poprzednio odczytany bajt.

Jeżeli dojdziemy do końca pamięci to pierwsze zera przepisywane są na diody aby je zgasić a następnie program się kończy.

Kody źródłowe

Asm

```
;-----------------------------------------------------------------------------
; Main program code place here
; 1. Place here code related to initialization of ports and interrupts

ldi r16,0x00
ldi r17,0xFF

out DDRA,r16 ; PORTA - jako wejsciowy
out PORTA,r17 ; PORTA - wejscia PULL-UP
out DDRB,r17 ; PORTB - jako wyjscie
out PORTB,r16 ; PORTB - wyjscie w stan niski by diody nie świeciły

;-----------------------------------------------------------------------------
; F2. Load initial values of index registers
;  Z, X, Y

ldi xl,0x00 ; nizsza czesc adresu - XL to rejest r28 (r28 i r27 to rejest X )
ldi xh,0x00 ; wyzsza czesc adresu
ldi r21,0xFF ; rejestr do odczytu wartsci eeprom - ustawianie na FF, zeby pozniej przy
czytaniu poprzednia wartosc nie byla zerem

CZEKAJ_NA_NACISNIECIE:
in r16, PINA ; wycztanie stanu portu A z przyciskami do rejestru
cpi r16, 0xFF ; porównanie bitów w porcie z 1111 1111 - żaden z przycisków nie naciśnięty
breq CZEKAJ_NA_NACISNIECIE ; równe to powrót do pętli odczytu

CZYTAJ_PAMIEC: ; przycisk wciśnięty, odczytujemy pamięc eeprom
sbic eecr, eepe ; sprawdzenie czy pamiec zajeta w sumie
rjmp CZYTAJ_PAMIEC
out eearh, xh ;adresik
out eearl, xl
sbi eecr, eere ; czytanie
mov r20, r21 ; wartosc poptrzednia do r20
in r21, eedr
cp r20, r21
brne ZWIEKSZANIE_ADRESU
cpi r21, 0x00
breq End ; koniec pamieci

ZWIEKSZANIE_ADRESU:
cpi xl, 0xFF ; sprawdzenie czy nizsze bity adresu sie przepelnily
breq INKREMENTUJ_WYZSZE ; jeśli nie - normalna inkrementacja
inc xl ; adres ++
rjmp WYPISZ_DANE
INKREMENTUJ_WYZSZE: ; gdy tak - inkrementujemy wyższą połówkę
ldi xl, 0x00 ; wyzerowanie nizszych
inc xh ; adres ++

WYPISZ_DANE:
out portb, r21 ; zawartość odczytanego bajtu z eepromu na portB

CZEKAJ_NA_PUSZCZENIE:
in r16, PINA ; wycztanie stanu portu A z przyciskami do rejestru
cpi r16, 0xFF ; porównanie bitów w porcie z 1111 1111 - żaden z przycisków nie naciśnięty
breq CZEKAJ_NA_NACISNIECIE ; równe to powrót do pętli oczekiwania na ponowne naciśnięcie
przycisku
rjmp CZEKAJ_NA_PUSZCZENIE; skok do petli czekania na puszczenie przycisku

;-----------------------------------------------------------------------------
; Program end - Ending loop
;-----------------------------------------------------------------------------
End:
rjmp END
```

C

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/eeprom.h>
#include <avr/pgmspace.h>

#define GET_FAR_ADDRESS(var) \
({ \
	uint_farptr_t tmp; \
	\
	__asm__ __volatile__( \
	\
	"ldi %A0, lo8(%1)" "\n\t" \
	"ldi %B0, hi8(%1)" "\n\t" \
	"ldi %C0, hh8(%1)" "\n\t" \
	"clr %D0" "\n\t" \
	: \
	"=d" (tmp) \
	: \
	"p" (&(var)) \
	); \
	tmp; \
})
void wypisz_dane(uint8_t dana);
uint8_t czytaj_pamiec(const int offset, uint8_t* poprzednia_wartosc, uint8_t* obecna_wartosc);
const uint8_t PROGMEM romtab[10] = {0xF0, 0x10, 0xFF, 0x11, 0x00, 0x01, 0xFA, 0x23, 0x00, 0x00};

uint8_t czytaj_pamiec(const int offset, uint8_t* poprzednia_wartosc, uint8_t* obecna_wartosc)
{
	*poprzednia_wartosc = *obecna_wartosc;
	*obecna_wartosc = eeprom_read_byte(offset); //odczyt komórki pamięci
	if(*obecna_wartosc == *poprzednia_wartosc && *obecna_wartosc == 0) //gdy mamy dwa zera w
tablicy
		return 0x00; //koniec tablicy
	return 0xFF; //poprawny odczyt

}
void wypisz_dane(uint8_t dana)
{
	PORTB = dana;
}
int main(void)
{
	int offset = 0;
	uint8_t czy_wpisac = 0x00;
	uint8_t poprzednia_wartosc = 0x00;
	uint8_t obecna_wartosc = 0xFF;
	DDRA = 0x00; // wejscie
	DDRB = 0xFF; // wyjscie
	PORTA = 0xFF; // + pullup-y
	//PINA = 0xFF;
	PORTB = 0x00; //diody nie świecą
	//PINB = 0x00;

	while(1)
	{
		if(PINA != 0xFF)
		{
			czy_wpisac = czytaj_pamiec(offset, &poprzednia_wartosc, &obecna_wartosc);
			if(czy_wpisac == 0xFF)
			{
				offset++; //poprawna wartość odczytana, przesuwamy się na kolejny bajt
tablicy
				wypisz_dane(obecna_wartosc);
			}
			else
				return 0; //wyjscie z programu - koniec działania

			while (PINA != 0xFF) {} //dopóki któryś przycisk naciśnięty, czekamy na
puszczenie go
		}
	}

	return 0;
}
```

Testowanie i Uruchamianie

Testowanie polegało na zapisaniu w Atmel Studio w trakcie debugowania zawartości pamięci EEPROM oraz stanu przycisków przed główną częścią programu. Następnie obserwowaliśmy poprawność przepływu sterowania programem w zależności od stanu pinów A, a także poprawność wypisywania danych z EEPROM na port B.

Wnioski

Ze względów praktycznych nie mogliśmy porównywać czasów wykonania poszczególnych programów, ponieważ do symulacji otoczenia (przycisków) musieliśmy podczas debuggowania samodzielnie ustawiać stan pinów A.

Jeśli chodzi o kwestię zajętości pamięci, nasz program w C zajmuje jej niemal dwukrotnie więcej – 1204 bajtów, w porównaniu do 660 bajtów wersji asm.

Dlatego jeśli dysponujemy mikroprocesorem o ograniczonej pojemności pamięci (jak np. ATtiny), możemy być zmuszeni do pisania samemu programu w asemblerze by zmieścić się z kodem o pełnej funkcjonalności projektu. Jednak w przypadku większych projektów łatwo się pogubić pisząc program w asemblerze, gdyż trzeba pamiętać o wielu rzeczach naraz i łatwo popełnić błąd. Dlatego by napisać rozbudowany program szybko i sprawnie możemy być zmuszeni do skorzystania z języka wysokiego poziomu jak np. C.