# Laboratorium PTM

## Programowanie w środowisku Microchip Studio – pierwszy program w języku asemblera.

Celem ćwiczenia jest zaznajomienie ze środowiskiem i ilustracja najprostszych programów tym razem w języku asemblera.

Zakład Systemów Informacyjno-Pomiarowych

IETiSIP, Wydział Elektryczny, PW

Program pierwszy w języku asemblera:

Utworzenie projektu asemblerowego pisane zostało w instrukcji projektASM.pdf. Przy analizie kodów asemblerowych przydatna jest dokumentacja mikrokontrolera ATmega328P z listą rozkazów ( i przypisanymi cyklami).

Niech pierwszy program w języku asemblera dla mikrokontrolera ATmega328p ma postać taką jak na rysunku 1. Zadanie polega na tym, aby w możliwie nieskomplikowany sposób zilustrować wykorzystanie języka asemblerowego w praktycznych przykładach i jednocześnie pokazać możliwości programowania w tym języku. Zadanie pierwszego programu to cykliczna zmiana stanu (stanów) wybranych wyjść mikrokontrolera. Można to bezpośrednio wykorzystać do migania diodami, a na etapie symulacji obserwować zaprogramowane zmiany wyjść wybranych portów.

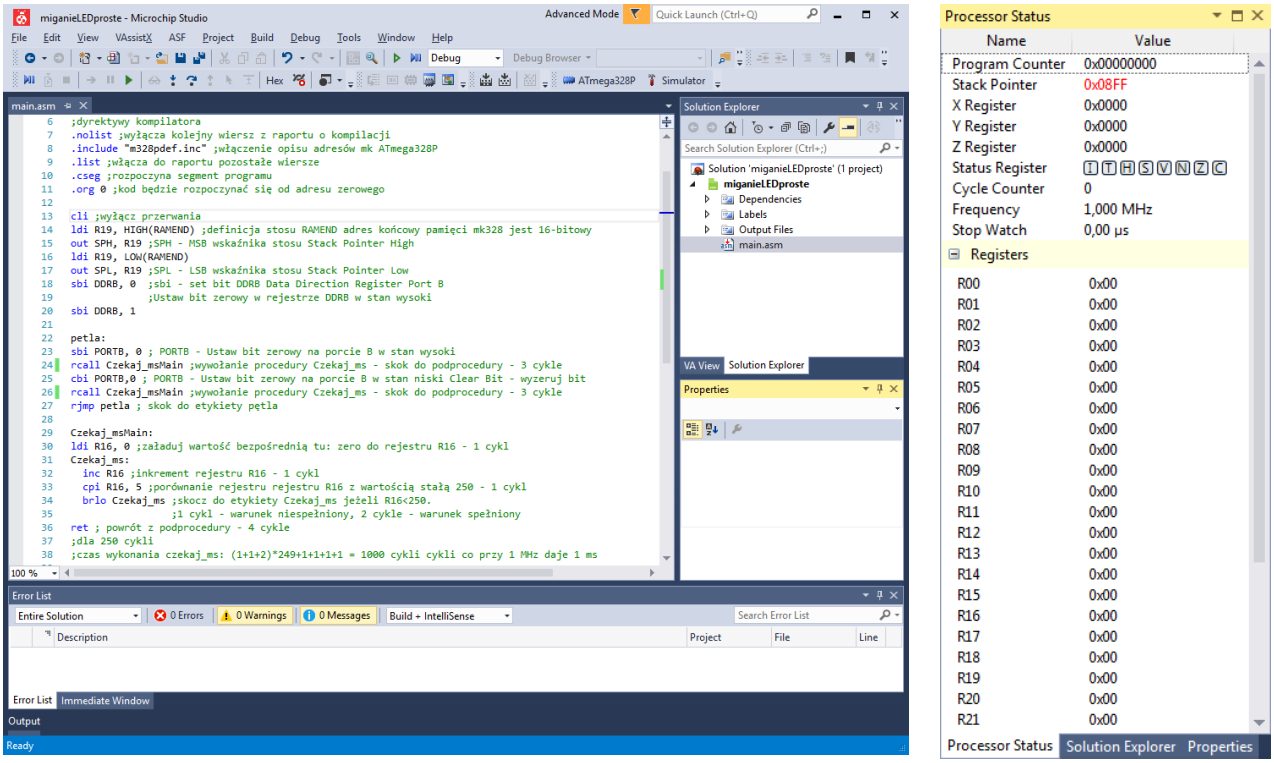
Struktura programu nie jest skomplikowana. Rozpoczyna się blokiem dyrektyw kompilatora, których charakterystyczną cechą jest początkowa kropka. Na początku znajduje się dyrektywa o wyłączeniu dalszej części kodu z raportu kompilacji. W tym przypadku chodzi o wyłączenie z raportu informacji o dodaniu opisu portów i adresów dla naszego mikrokontrolera (plik m328pdef.inc). Włączanie tego typu definicji jest typową operacją a włączany plik jest (w tym przypadku) przygotowanym przez producenta zestawem stosownych informacji. Tego typu operacja zwalnia programistę z bezwzględnej znajomości i każdorazowej kontroli zgodności tworzonego kodu ze szczegółami w zakresie wyprowadzeń i adresów programowanego mikrokontrolera, oferując zestaw stałych - literałów znakowych (PORTB, DDRB, RAMEND). Ułatwiają one pisanie programu i czynią go łatwiejszym w zrozumieniu. Jako, że raport z etapu kompilacji najbardziej interesuje Nas w zakresie naszych poczynań (własnej części programu), stąd wyłączone są elementy standardowe. Zwłaszcza, iż mogły by one „zdominować" raport jako taki. Włączony nagłówek ten zawiera informacje o adresach rejestrów i wyprowadzeń używanego mikrokontrolera. Podczas tworzenia projektu wybrany został konkretny mikrokontroler a włączony nagłówek jest tu odpowiedzialny za przypisanie właściwych danych i dodatkowo zdefiniowanie nazw dla rejestrów i wyprowadzeń konkretnego mikrokontrolera. Dzięki temu nie trzeba się szukać w dokumentacji specyficznych danych dla wybranego układu i co więcej można wykorzystywać ustandaryzowane nazwy dla poszczególnych elementów wybranego mikrokontrolera.

Zaraz po włączeniu zestawu definicji przywracane jest raportowanie procesu kompilacji (linia 10 na rysunku 1). Po niej zamieszczona jest dyrektywy `.cseg` i `.org 0` informujące o rozpoczęciu segmentu kodu, którego początek znajduje się pod adresem 0.

Następnie wyłączane są przerwania (linia 14). W tym momencie najważniejsze jest to, że nasz program nie będzie przerywany i mikrokontroler nie będzie przechodził do obsługi zdarzeń wywołujących przerwania.

Kolejne wiersze kodu (linie od 15 do 18) jest odpowiedzialny za początkowe wskazanie rejestru wskaźnika stosu. Stos jest kolejką typu LIFO wykorzystywaną w trakcie wykonywania kodu do przechowywania tymczasowych danych pomocniczych. Jako przykład można tu podać przekazywane do wywoływanej funkcji argumenty czy też adres powrotu do funkcji wywołującej . Jak widać w liniach tych znajdują się zarówno makrorozwinięcia jak i stałe. Dzięki włączonemu nagłówkowi nie trzeba pamiętać, jaka jest największa wartość adresu dla naszego mikrokontrolera i co więcej nazwa RAMEND sam się „przedstawia i wyjaśnia". Makrodefiniecje LOW i HIGH pobierają odpowiednio młodszy (LSB) oraz starszy (MSB) bajt tego adresu.

Z linii 15-18 można już wywnioskować, że umieszczany jest na końcu pamięci RAM mikrokontrolera. Skąd taka lokalizacja. W tamtym obszarze istnieje najmniejsze prawdopodobieństwo konfliktu ze zdefiniowanymi zmiennymi i danymi. I tu ujawnia się też specyfika organizacji stosu tj. „do góry nogami". Jak widać w programie początkowy wskaźnik stosu SP wskazuje na ostatnią komórkę pamięci RAM. Po umieszczeniu danych na stosie wskaźnik ten może być jedynie dekrementowany (zmniejszany) i tak się właśnie dzieje. Natomiast po pobraniu danych ze stosu jest on inkrementowany. Ze względu na to, że mikrokontroler ATmega 328P ma 2kB pamięci danych stąd adresowanie musi być 16-bitowe. Stąd ładowanie rejestru SP – wskaźnika stosu odbywa się w dwóch krokach za pomocą rejestrów 8-bitowych.




Rysunek 1. Pierwszy asemblerowy program w Microchip Studio.

Po zainicjowaniu rejestru SP w liniach 19 i 21 ustawiane są bity zerowy i pierwszy (wagi) rejestru DDRB (Data Direction Register Port B). Oznacza to, że linie te będą wyjściami.
Kolejne linie (23-28) zawierają główny kod programu. Znajdująca się w linii 23 etykieta a w 28 linii bezwarunkowy skok (powrót) do niej tworzą pętlę nieskończoną (np.: **`while(1)`** bądź **`for(;;)`** ). Treść pętli stanowi naprzemienna zmiana stanu bitu zerowego na porcie B – instrukcje **`sbi`** oraz **`cbi`**. Pomiędzy tymi instrukcjami znajdują się wywołania procedury **`Czekaj_msMain`**, która jest odpowiedzialna za wprowadzenie opóźnienia czasowego. Tym razem jednak będzie to procedura opóźniająca uproszczona i w miejsce np.: timerów użyte będą zwykłe operacje inkrementacje rejestrów (**`inc`**). W procedurze **`Czekaj_msMain`** wykorzystano rejestr ogólnego przeznaczenia **`R16`** i instrukcję skoku warunkowego **`brlo`**. Procedura ma zasadniczo postać pętli otoczonej etykietą **`Czekaj_ms`** i skokiem warunkowym (**`brlo`**) do tejże etykiety – linie 31-34 na rysunku 1. Tak więc początkowa zerowa wartość rejestru jest w pętli inkrementowana, aż do momentu kiedy zawartość **`R16`** osiągnie wartość 250. Powstaje pytanie: no tak a gdzie jest opóźnienie ? Proszę zwrócić uwagę na opisy w kodzie. Każdej instrukcji przypisano liczbę cykli procesora, jakie zajmuje jej wykonanie. Instrukcja skoku warunkowego **`brlo`** zajmuje 2 cykle przy spełnionym warunku i 1 cykl kiedy jest on niespełniony. Na tej podstawie można policzyć ile cykli pojedynczy przebieg **`Czekaj_ms`** oraz łącznie wykonanie procedury **`Czekaj_msMain`** wraz instrukcjami **`rcall`** oraz **`ret`**. Dodatkowo przyjmując wartość częstotliwości rezonatora

kwarcowego można wyznaczyć czas wykonania ww. procedury w sekundach. Rachunki można zweryfikować w trakcie wykonania krokowego programu. Szczegóły wykonania programu można poznać wybierając (przy uruchomionym programie) **Debug**->**Windows**->**Disassembly** i otrzymując w wyniku asemblerowy kod programu. Pomocne będzie w tym okno **Processor Status** (rysunek 1), a w nim pozycje **Cycle Counter**, **Stop Watch** oraz **Frequency**. W takt realizacji kolejnych instrukcji pierwsza z nich wskazuje ile cykli procesora zajął do tej wory wykonywany program, druga natomiast pokaże czas (rzeczywisty w sekundach) jaki upłynął od początku programu. Wartość ta jest wyznaczana na podstawie **Cycle Counter** oraz **Frequency**. Ten ostatni parametr można zmieniać w zależności od rezonatora jaki faktycznie jest podłączony do mikrokontrolera.

W języku asemblera również istnieje możliwość wykonywania programu w trybie krokowym. Jednakże w tym przypadku wykonywanie programu odbywa się instrukcja po instrukcji asemblerowej (NIE zaś instrukcja po instrukcji w języku C). W trakcie krokowego wykonywania programu można podglądać bieżące wartości rejestrów ogólnego przeznaczenia CPU (Rxx); rejestrów systemowych: CYCLE_COUNTER, FP, PC, SP oraz X, Y i Z oraz rejestrów wykorzystywanych przy operacjach zmiennoprzecinkowych: SREG. Wymienione rejestry dotyczą wykorzystywanego mikrokontrolera tj. ATmega328P. Stan rejestrów można również obserwować w oknie **Memory**. Należy w tym celu wybrać podgląd na **data REGISTERS**. Adresy rejestrów można odczytać z dokumentacji mikrokontrolera. Jeszcze inną możliwością jest wybranie **Debug** -> **Windows** -> **Processor Status**, rysunek 1). W tym miejscu na rysunku 3 przestawiona została oryginalna ilustracja z mapą rejestrów mikrokontrolera ATmega328p. W przypadku podglądu zmiennych programu należy przejść do **data IRAM** (Internal RAM).
Wykonanie krokowe programu należy tym razem przeprowadzić w trybie **Step Into** (F11). Tryb ten przechodzi do wykonania krokowego również i procedur. W tym przypadku chcemy przeanalizować własną procedurę, dlatego też **Step Into** (F11). Zawsze możemy „przerwać" i przejść do trybu **Step Over** (F10) – w tym przypadku musimy tylko dokończyć aktualnie analizowaną procedurę i kolejne fazy wykonania krokowego odbędą się już bez „zaglądania" do (pod)procedur.
Podczas analizy pracy krokowej należy zwrócić uwagę na zwartość rejestru SP. Został on jawnie zainicjowany wartością końcowego adresu pamięci danych (RAMEND dla ATmega 328 wynosi 0x08FF gdyż dysponuje on pamięcią danych w wielkości 2kB + 256B dla rejestrów). W programie występują dwa wywołania skoków do podprocedury **`Czekaj_msMain`** (linie 24 i 26 na rysunku 1). Z tym wiąże się zapamiętanie na stosie adresu powrotu z podprocedury do programu głównego. W przypadku z rysunku 1 będą to odpowiednio instrukcje w liniach 25 oraz 27. Wykonując analizę krokową można przeprowadzić deasemblację kodu (**Debug**->**Windows**->**Disassembly**) i sprawdzić jaki kod efektywnie jest wykonywany i czy są różnice pomiędzy kodem źródłowym a finalnie wykonywanym? Jaka może być tego przyczyna? Ponadto w oknie **Processor Status** należy podglądać wartość rejestru SP. Powinna przyjmować ona wartości 0x8FF w programie głównym i 0x8FD w podprocedurze (adres powrotu zajmuje 2 bajty stąd taka różnica). Dodatkowo po przejściu do podprocedury proszę sprawdzić zawartość samego stosu, tzn. co zawiera oraz czy jest to prawidłowy adres powrotu z podprocedury do programu głównego (pomocne będzie okno **Debug**->**Windows**->**Disassembly**)? Wykorzystanie stosu w przykładzie jest ograniczone i sprowadza się do zapamiętania adresu powrotu. W praktyce należy zapamiętać również rejestr statusu **`SREG`** oraz wszystkie rejestr używane w podprocedurze. W przykładzie byłby to rejestr R16 (oraz w wersji rozszerzonej R17, R18, R19...). Ze względu na prostotę programu stosowne instrukcje **`PUSH`** oraz **`POP`** zostały pominięte (powiedzmy optymalizacja) ale nie wolno zapominać o zasadzie.
Czasowa analiza pierwszego programu asemblerowego (wszak programujemy opóźnienie – delay) pokaże, że maksymalne opóźnienie zrealizowane w programie (użyte rejestry i zmienne są rozmiaru pojedynczego bajta) wynosi ok. 1 ms, przy założeniu częstotliwości zegara wynoszącej 1 MHz. A co zrobić, jeżeli będziemy chcieli zrealizować w analogiczny sposób większe (dłuższe) opóźnienie ? O ile możemy wydłużyć opóźnienie wprowadzane przez tę procedurę? Zapiszmy może, posługując się pseudokodem cykl **`Czekaj_ms`**. Zauważmy, że wykorzystany został rejestr ogólnego przeznaczenia tu. R16 do przechowywania wartości inkrementowanej i testowanej. W

ten sposób procedurę wykonuje się wielokrotnie (uzyskując regulowane opóźnienie czasowe). Poszczególne linie odpowiadają pojedynczym instrukcjom programu z rysunku 1. Dla mikrokontrolerów ATmega rejestry R0..R15 nie mogą być wykorzystywane w instrukcjach ze stałymi, stąd użyty jest R16 (pierwszy możliwy i dostępny).

```
Zerowanie rejestru Czekaj_ms (R16)
Czekaj_ms:
      Inkrementacja rejestru Czekaj_ms (R16)
      Sprawdzenie czy zawartość rejestru Czekaj_ms (R16) jest mniejsza niż 250
      Jeżeli zawartość jest mniejsza wtedy skok warunkowy do Czekaj_ms.
```

Wartość rejestru (1 bajt) może wynosić co najwyżej 255, więc w zasadzie mamy tu do czynienia z maksymalnym opóźnieniem. Chcąc zwiększyć opóźnienie można zastosować zagnieżdżanie procedur wg. poniższego schematu. Wykorzystane rejestry ogólnego przeznaczenia w tym przypadku to R16 i R17 (ale mogą być inne – rysunek 2). Wartości N i N2 to stałe (jednobajtowe) określające liczbę powtórzeń swoich procedur. Dobrane są tak, aby uzyskać wymaganą zadanym opóźnieniem liczbę powtórzeń odpowiednich procedur co przełoży się na liczbę cykli procesora i w efekcie czas opóźnienia. Należy pamięta, że każda pętla zagnieżdżona musi mieć przypisany osobny rejestr (np: R16, R17, R18, R19). Pamiętać należy tylko o tym, aby nie był on wykorzystywany jednocześnie do innych celów. Większe wartości opóźnień mogą wymagać dodatkowego/ dodatkowych stopni zagnieżdżeń.

```
Zerowanie rejestru Czekaj_ms2 (R17)
Czekaj_ms2:
      Zerowanie rejestru Czekaj_ms (R16)
      Czekaj_ms:
            Inkrementacja rejestru Czekaj_ms (R16)
            Sprawdzenie zawartości rejestru Czekaj_ms (R16) czy mniejsza niż N
            Jeżeli zawartość jest mniejsza - skok do Czekaj_ms
      Inkrementacja rejestru Czekaj_ms2 (R17)
      Sprawdzenie zawartości rejestru Czekaj_ms2 (R17) czy mniejsza niż N2
      Jeżeli zawartość jest mniejsza - skok do Czekaj_ms2
```

Wcięcia na powyższym schemacie są umowne, ale pokazują one przynależność do określonych podprocedur.

**Figure 7-2. AVR CPU General Purpose Working Registers**

| | 7 0 | Addr. | |
|---|---|---|---|
| | R0 | 0x00 | |
| | R1 | 0x01 | |
| | R2 | 0x02 | |
| | ... | | |
| | R13 | 0x0D | |
| General | R14 | 0x0E | |
| Purpose | R15 | 0x0F | |
| Working | R16 | 0x10 | |
| Registers | R17 | 0x11 | |
| | ... | | |
| | R26 | 0x1A | X-register Low Byte |
| | R27 | 0x1B | X-register High Byte |
| | R28 | 0x1C | Y-register Low Byte |
| | R29 | 0x1D | Y-register High Byte |
| | R30 | 0x1E | Z-register Low Byte |
| | R31 | 0x1F | Z-register High Byte |

Rysunek 3. Rejestry ogólnego przeznaczenia mikrokontrolera ATmega328P.

**Zadania do wykonania:**

Przyjmując częstotliwość zegara mikrokontrolera równą: 8MHz, 12MHz, 16MHz i 20MHz proszę napisać procedurę wprowadzającą opóźnienie: 1ms, 100ms i 1s.

**Dodatek:**
Kompilację i wygenerowanie plików binarnych można wykonać poprzez Build -> **Build Solution/Application** (F7), po ewentualnych zmianach kodu można skorzystać z opcji **Rebuild** (Ctrl+Alt+F7) a „zresetowanie" projektu można uzyskać poprzez **Clean Solution/Application**. W ćwiczeniu program można uruchamiać poprzez pasek narzędziowy i **Start Debugging** (F5) oraz **Stop Debugging** (Ctrl+Shift+F5). Identycznie można sterować wykonaniem programu wybierając odpowiednie opcje z menu **Debug**. W ćwiczeniach jednak wykorzystywać należy tryb krokowy: **Step Over** (F10). Jest to tryb pracy krokowej bez wchodzenia do funkcji/procedur (wykonywane są one w jednym kroku). Jeżeli zaistnieje potrzeba przeanalizowania działania funkcji (szczególnie samodzielnie napisanej) wtedy można użyć trybu **Step Into** (F11). Możliwość wyjścia z procedury da w tym przypadku **Step out** (Shift+F11). Powyższe opcje są standardowe dla środowisk programistycznych. Po uruchomieniu pierwszego programu (F10) okno środowiska będzie wyglądać jak na rysunku 1 (z prawej strony). Wciskając kolejno F10 program wykonywał będzie kolejne instrukcje. Żółta strzałka wskazuje aktualną instrukcję do wykonania (wiąże się to z zawartością rejestru **PC – Program Counter**). W tym samym obszarze, w którym znajduje się strzałka można ustawiać pułapki (breakpoints). Wystarczy użyć lewego klawisza myszy. Pułapki są użyteczne, kiedy zaistnieje potrzeba dokładnej analizy pewnego fragmentu kodu (przy założeniu, że program jest złożony niż omawiany obecnie) z pominięciem pozostałej części.
Okno **Autos** (na dole po lewej stronie) zawiera zmienne zdefiniowane w programie wraz z informacją o położeniu (lokalizacji, adresie) zmiennej w pamięci danych. Architektura mikrokontrolerów ATmega jest architekturą Harwardzką z podziałem na pamięć danych i pamięć programu. W oknie **Memory** po prawej stronie można podglądać zawartość pamięci. Przy wyborze **IRAM** (Internal RAM) można podglądać aktualną zawartość pamięci danych. Znając adresy poszczególnych zmiennych w programie (okienko **Autos** i zmienne: a, b, c) można podglądać zmiany tychże zmiennych (w takt wykonywania programu - F10).