

C# .NET Aplikacja symulująca pracę mikrokontrolera

1. Wstęp

Celem zadania 5 napisanie aplikacji stanowiącej model programowego symulatora mikroprocesora. Wykorzystując dowolnie wybrany język programowania należy napisać aplikację symulującą pracę mikroprocesora. Z założenia, program powinien mieć przyjazny interfejs użytkownika graficzny lub znakowy do uznania przez autorów.

Do realizacji posłużyliśmy się programem Microsoft *Visual Studio, Windows Forms App (.NET Framework)* w języku C#, co umożliwia na wykorzystanie techniki graficznego tworzenia interfejsu użytkownika.

2. Założenia

Założyliśmy, że model naszego procesora będzie następujący:

- 1) Procesor posiada cztery 16-bitowe rejestry ogólnego przeznaczenia oznaczone jako AX, BX, CX i DX;
- 2) Każdy z rejestrów powinien być traktowany jako para 8-bitowych rejestrów o oznaczeniach NH dla części starszej i NL dla części młodszej gdzie N oznacza A albo B albo C albo D;
- 3) Lista rozkazów naszego procesora obejmuje trzy rozkazy MOV – przesłania, ADD – dodawania i SUB – odejmowania;
- 4) Procesor umożliwia realizację dwóch trybów adresowania: trybu rejestrowego oraz trybu natychmiastowego;
- 5) Program umożliwia pisanie krótkich programów z użyciem dostępnych rozkazów i trybów adresowania;
- 6) Program umożliwia realizację napisanych programów w trybie całościowego wykonania i w trybie pracy krokowej.
- 7) W trybie pracy krokowej należy zapewnić śledzenie kolejności wykonywanych instrukcji. Zalecenie to należy zrealizować stosując numerację linii kodu programu oraz odwołania do numeru aktualnie wykonywanej instrukcji.
- 8) Program symulatora umożliwia zapisanie do pliku a następnie ponowne wczytanie napisanego wcześniej programu w celu jego dalszej edycji i ponownego uruchomienia.

3. Postęp pracy

3.1. Wykonanie interfejsu graficznego

Na początku zbudowaliśmy prosty interfejs graficzny. Do tego posłużyliśmy się rozszerzeniem pliku .cs który pozwala na tworzenie aplikacji w sposób graficzny. Dodaliśmy przyciski oraz label'i do wyświetlania potrzebnych danych:

- Reset – przycisk resetujący program.
- Command – wybranie instrukcji: mov, add, sub.
- Register 1 – wybranie pierwszego rejestru: ax, bx, cx, dx.

- Register 2 v wybranie drugiego rejestru: ax, bx, cx, dx.
- Number – pole do wpisania liczby do rejestru nieprzekraczającej zakres [0; 2¹⁶-1].
- Confirm – zatwierdzenie linijki kodu, przejście do pisania linijki następnej.
- Holistic mode – praca w trybie całościowym.
- Step mode – praca w trybie krokowym.
- ax, bx, cx, dx – wyświetlanie zawartości rejestrów.
- Edytuj linijkę – miejsce na wpisanie linijki do edytowania.
- Upload code to file – zapisywanie napisanego programu do pliku .txt.
- Download code from file – pobranie kodu z komputera i wyświetlanie programu w polu tekstowym.

Na danym etapie plik *Form1.cs [Design]* wyglądał następująco:

Rys.1 - Zbudowanie interfejsu graficznego

3.2. Podstawowe funkcje kodu

Aby program działał według założeń, stworzyliśmy podstawowe funkcje, których zadaniem jest:

1. Reset – resetowanie programu.
2. Confirm – weryfikacja poprawności wpisanej linijki oraz jej zatwierdzenie.
3. Holistic mode – numeryczne obliczania oraz wyświetlenie wyniku końcowego na rejestrach.
4. Step mode – numeryczne obliczania oraz wyświetlanie wyników pośrednich zależących od aktualnie wykonywanej instrukcji.
5. ToFile i FromFile – zapisywanie kodu do pliku .txt i jego wczytywanie do programu.

Funkcja *reset()* resetuje program, nadając wszystkim zmiennym wartość początkową oraz zerując zawartość rejestrów i pole do wpisywania instrukcji.

Funkcja *confirm_Click()* sprawdza, co wpisano do aktualnej linijki (za pomocą flag *instr*, *pierwszy_r*, *drugi_r*, *liczba*) i zapisuje te dane w liście rozkazów: *ins.Add(rozkaz.Text)*, listę rejestru 1: *reg1.Add(register1.Text)*, listę rejestru 2: *reg2.Add(register2.Text)*, listę liczby:

`licz3.Add(number.Text)`. Oprócz tego wynik aktualnej instrukcji jest na bieżąco obliczany, aby zapobiec przekroczeniu zakresu danych możliwych do wpisania rejestrów 16-bitowych i poinformować o tym użytkownika: `if(przekroczono_zakres == true)`.

W taki sposób zatwierdzana/ lub nie jest każda linijka kodu, wpisane dane są zapisywane do list, aby w przyszłości wykonać obliczenia numeryczne i wyświetlić wynik w postaci liczb binarnych na rejestrach.

```
private void confirm_Click(object sender, EventArgs e)
{
    if (nowa_linija == true && instr == true && pierwszy_r == true && (drugi_r == true || liczba == true))
    {
        if (edytowanie == true) ...
        if (edytowanie == false)
        {
            ins.Add(rozkaz.Text);
            reg1.Add(register1.Text);

            if (drugi_r == true)
            {
                reg2.Add(register2.Text);
                licz3.Add(0);
                r2.Add(true);
                li.Add(false);
            }
            else if (liczba == true)
            {
                licz3.Add(convert_number);
                reg2.Add("0");
                r2.Add(false);
                li.Add(true);
            }
        }

        if (przesun == true) ...
        if (dodaj == true) ...
        if (odejmij == true) ...

        if (przekroczono_zakr == false) ...
        else ...
    }
}
```

Rys.2 - Funkcja `confirm_Click()`

Po wpisaniu kilku linijek kodu można uzyskać wynik, naciskając na przycisk *Holistic/ Step mode*, w zależności od tego jaki tryb wyświetlania wyników chcemy uzyskać.

```

private void holistic_mode_Click(object sender, EventArgs e)
{
    h_mode = true;
    try
    {
        uwagi.Text = "Wykonano tryb HOLISTIC MODE";
        int wynik;
        wykonaj_obliczenia();

        for (int i = 0; i < numeracja_l - 1; i++)
        {
            wynik = Convert.ToInt32(this.step_wynik[i]);
            switch (step_komenda[i])
            {
                case "1":
                    wpisz_do_rejestru(ah, al, wynik);
                    break;
                case "2":
                    wpisz_do_rejestru(bh, bl, wynik);
                    break;
                case "3":
                    wpisz_do_rejestru(ch, cl, wynik);
                    break;
                case "4":
                    wpisz_do_rejestru(dh, dl, wynik);
                    break;
                default:
                    break;
            }
        }
    }
    catch (FormatException)
    {
        uwagi.Text = "Nie wprowadzono żadnej instrukcji!";
        instrukcje.Text = instrukcje.Text.Substring(0, 8);
    }
    h_mode = false;
}

private void step_mode_Click(object sender, EventArgs e)
{
    s_mode = true;
    try
    {
        uwagi.Text = "Wykonywano tryb STEP MODE. " +
            "Aby kontynuować program, zakończ tryb";

        if (step == 0)
        {
            wpisz_do_rejestru(ah, al, 0);
            wpisz_do_rejestru(bh, bl, 0);
            wpisz_do_rejestru(ch, cl, 0);
            wpisz_do_rejestru(dh, dl, 0);
            wykonaj_obliczenia();
        }

        if (step < numeracja_l-1)
        {
            aktualna_lin.Text = (step+1).ToString();
            int wynik = Convert.ToInt32(this.step_wynik[step]);

            switch (step_komenda[step])
            {
                case "...": // blokowanie możliwości naciśnięcia innych przycisków
                    step++;
                    break;
                case "...": // wyzwalanie przycisków
                    break;
            }
        }
    }
    catch (FormatException)
    {
        uwagi.Text = "Niewprowadzono żadnej instrukcji!";
        instrukcje.Text = instrukcje.Text.Substring(0, 8);
    }
    s_mode = false;
}

```

Rys.3 - Porównanie działania funkcji `step_mode_Click()` i `holistic_mode_Click()`

Funkcja `holistic_mode_Click()` działa w taki sposób, że najpierw wykonuje obliczenia na liczbach zebranych w wyżej wymienionych listach (`wykonaj_obliczenia()`), po czym w pętli `for` wpisuje do rejestrów wyniki uzyskane w każdym kroku (`wpisz_do_rejestru()`). Działanie to jest na tyle szybkie, że użytkownik widzi jedynie wynik ostateczny wyświetlony na rejestrach.

Funkcja `step_mode_Click()` działa nieco w inny sposób - nie ma w niej pętli `for`, dlatego funkcja wykonuje się tylko jeden raz. Ale zatem jest licznik, który zlicza wynik którego kroku aktualnie powinien być wyświetlony na rejestrach. Ilość naciśnieć na przycisk `Step mode` jest równoważna ilości wykonanych kroków (wykonanych linijek kodu). Dla bezpieczeństwa podczas pracy krokowej uniemożliwiona żadna edycja kodu. Dopiero po zakończeniu tego trybu pracy można dalej pisać kod.

Obydwie funkcje zawierają zabezpieczenia w postaci `try/catch` gdyby użytkownik chciał wywołać `step/olistic mode` bez wprowadzenia żadnego kodu do aplikacji.

Warte uwagi są funkcje `wpisz_do_rejestru()` i `wykonaj_obliczenia()`.

Pierwsza funkcja otrzymuje jej przekazany wyniki `wyn` i przekształca daną liczbę do postaci binarnej `binary_num` w taki sposób, żeby uzyskać liczbę 16-bitową. W dalszej części wynik jest podzielony na dwie równe części po 8 bitów. Starsza część jest wpisywana do rejestru `h`, a młodsza do rejestru `l`.

```

private void wpisz_do_rejestru(Label h, Label l, int wyn)
{
    binary_num = Convert.ToString(wyn, 2);
    binary_num = binary_num.PadLeft(16, '0');

    int chunkSize = 8;
    int stringLength = binary_num.Length;
    for (int i = 0; i < stringLength; i += chunkSize) // petla wykonuje sie 2 razy
    {
        if (i + chunkSize > stringLength) chunkSize = stringLength - i;
        {
            if (i == 0 & (h_mode == true || s_mode == true))
            { h.Text = (binary_num.Substring(i, chunkSize)); }

            if (i == chunkSize & (h_mode == true || s_mode == true))
            { l.Text = (binary_num.Substring(i, chunkSize)); }
        }
    }
}

```

Rys.4 - Funkcja wpisz_do_rejestru()

Kolejna funkcja *wykonaj_obliczenia()*, można powiedzieć, jest najistotniejsza w całym programie, ponieważ właśnie ona dokonuje obliczeń numerycznych napisanego przez użytkownika kodu. Na początku resetowano listy *step_komenda* i *step_wynik*, aby do nich wpisywać wyniki każdej z linii kodu. W pętli for co krok patrzy się na zawartość szeregu listy *ins[i]*. W zależności od tego, jaka instrukcja wystąpiła w kroku *i*, wykonywane jest dodawanie, odejmowanie lub przesunięcie rejestrów lub liczby i rejestru.

Przykładowo została przedstawiona funkcja *wykonaj_przesunięcie()*. Do niej przekazana jest zawartość rejestrów (listy *r1* i *r2*) oraz liczby (*l3*) w chwili *i*. W zależności od tego do jakiego rejestru z *ax*, *bx*, *cx* i *dx* chcemy przesunąć liczbę/zawartość innego rejestru, wynik działania (w tym przypadku przesunięcia) zapisujemy do listy *step_wynik*, a rejestr, do jakiego zamieszczony został wynik do listy *step_komenda*.

W podobny sposób działają funkcje *wykonaj_dodawanie()* i *wykonaj_odejmowanie()*.

```

private void wykonaj_obliczenia()
{
    step_komenda.Clear();
    step_wynik.Clear();
    for (int i = 0; i < numeracja_1 - 1; i++)
    {
        switch (ins[i])
        {
            case "mov":
                przesun = true;
                break;
            case "add":
                dodaj = true;
                break;
            case "sub":
                odejmij = true;
                break;
            default:
                break;
        }

        if (przesun == true)
        {
            wykonaj_przesuniecie(r2[i], li[i],
                                reg1[i], reg2[i], liczb3[i]);
            przesun = false;
        }

        if (dodaj == true) ...
        if (odejmij == true) ...
    }
}

private void wykonaj_przesuniecie(bool rej2,
    bool liczb, string r1, string r2, int l3)
{
    if (rej2 == true)
    {
        switch (r2) ...
    }
    if (liczb == true) ...
    switch (r1)
    {
        case "ax":
            axx = r1_to_r2;
            step_wynik.Add(axx.ToString());
            step_komenda.Add("1");
            break;
        case "bx":
            bxx = r1_to_r2;
            step_wynik.Add(bxx.ToString());
            step_komenda.Add("2");
            break;
        case "cx":
            cxx = r1_to_r2;
            step_wynik.Add(cxx.ToString());
            step_komenda.Add("3");
            break;
        case "dx":
            dxx = r1_to_r2;
            step_wynik.Add(dxx.ToString());
            step_komenda.Add("4");
            break;
        default:
            break;
    }
    przesun = false;
}

```

Rys.5 - Funkcje `wykonaj_obliczenia()` i `wykonaj_przesuniecie()`

Ostatnie najważniejsze funkcje to `ToFile_Click()` oraz `FromFile_Click()`.

Funkcja `ToFile_Click()`, obsługująca kliknięcie guzika zapisu, odpowiedzialna jest za zapisanie tekstu naszego assemblerowego programu do pliku tekstowego. Jest ona jedną z prostszych funkcji w projekcie. Pobiera nazwę pliku docelowego (bez rozszerzenia) i sprawdza, czy takowy plik już istnieje za pomocą komendy `if (!File.Exists(FileNameBox.Text + ".txt"))`. Jeżeli takowy plik nie istnieje, Tworzy go i wykorzystując obiekt klasy `StreamWriter` wpisuje do niego tekst z textboxu zawierającego nasz kod. W przeciwnym razie wykonuje te same czynności tylko bez utworzenia pliku.

Funkcja `FromFile_Click()`, obsługująca kliknięcie guzika odczytu jest już trochę bardziej skomplikowana. Wykorzystuje ona stworzony na etapie projektowania graficznego obiekt typu `openFileDialogue`. W momencie, gdy plik zostanie wybrany przyciskiem zatwierdzającym w oknie dialogowym, tworzony jest nowy obiekt `sr` klasy `StreamReader` i powiązany z wybranym przez okno dialogowe plikiem. Następnie kolejne linijki tekstu w tym pliku są zapisywane do tablicy stringów. Po wpisaniu linijek do tablicy wchodzimy do pętli `while(true)`, w której, korzystając z `try`, przechodzimy po kolejnych elementach tablicy, zliczając je i wpisując ich zawartość do textboxu z kodem. Aktualizujemy przy tym również pole numeracji linijek. W momencie, kiedy wyjdziemy poza tablicę, dojdzie do błędu. W ten sposób dowiadujemy się, że cały plik został przejrany. Dlatego też wykorzystywaliśmy metodę `try`, a zakończenie procesu (wyjście z pętli) wykonaliśmy za pomocą `catch() {break;}`.

```

private void ToFile_Click(object sender, EventArgs e)
{
    if (!File.Exists(fileNameBox.Text + ".txt")) // If file does not exists
    {
        File.Create(fileNameBox.Text + ".txt").Close(); // Create file
        using (StreamWriter sw = File.AppendText(fileNameBox.Text + ".txt"))
        { sw.WriteLine(instrukcje.Text); } // Write text to .txt file
    }
    else // If file already exists
    {
        // File.WriteAllText("FILENAME.txt", String.Empty); // Clear file
        using (StreamWriter sw = File.AppendText(fileNameBox.Text + ".txt"))
        { sw.WriteLine(instrukcje.Text); } // Write text to .txt file
    }
}

```

Rys.6 - Funkcja ToFile_Click()

```

private void FromFile_Click(object sender, EventArgs e)
{
    if (openFileDialog1.ShowDialog() == System.Windows.Forms.DialogResult.OK)
    {
        reset();
        System.IO.StreamReader sr = new
        System.IO.StreamReader(openFileDialog1.FileName);
        //string lines = sr.ReadToEnd();
        string[] line = sr.ReadToEnd().Split('\n');
        int number_of_lines = 0;
        instrukcje.Text = null;
        numeracja_linii.Text = null;
        numeracja_l = 0;

        while (true)
        {
            try
            {
                string a = line[number_of_lines];
                numeracja_l++;
                number_of_lines++;
                instrukcje.Text += a;
                instrukcje.Text += "\n";
                linijka += a.Count()+1;
                numeracja_linii.Text = numeracja_linii.Text + numeracja_l.ToString() + newLine;
            }
            catch (System.Exception eas)
            {
                break;
            }
        }
    }
}

```

Rys.7 - Funkcja FromFile_Click()

The screenshot shows a software window titled "Form1" with a light gray background. At the top, there are window control buttons (minimize, maximize, close). The main interface is divided into several sections:

- Command List:** A table with columns "Reset", "Command", "Register 1", "Register2", and "Number". It contains five rows of assembly-like commands:

Reset	Command	Register 1	Register2	Number
<input type="button" value="OK"/>	sub	ax		1
1	mov ax, 1			
2	mov bx, 2			
3	add ax, 4			
4	add cx, 5			
5	sub ax, 1			
6				
- Registers:** A section titled "Registers" showing four registers: ax, bx, cx, and dx. Each has two 8-bit value boxes.

Register	Value 1	Value 2
ax	00000000	00000100
bx	00000000	00000010
cx	00000000	00010100
dx	00000000	00000000
- Execution Controls:**
 - Buttons for "Confirm", "Holistic mode", and "Step mode". The "Step mode" button is highlighted with a blue border.
 - A status bar at the bottom left says: "Aktualnie wykonywana instrukcja w trybie STEP MODE: 5".
 - A warning message: "Uwagi: Wykonywano tryb STEP MODE. Aby kontynuować program, zakończ tryb".
 - A checkbox labeled "Edytuj linijkę:" followed by an "OK" button.
 - Buttons for "Upload code to file" and "Download code from file".

Rys.8 - Ostateczne działanie programu - tryb pracy krokowej dla napisanego kodu

4. Wyniki pracy

Wyniki pracy można uznać za pomyślne, gdyż aplikacja dobrze pełni swoją funkcję symulacji mikroprocesora. Aplikacja umożliwia użytkownikowi symulację pracy mp, przy czym uwzględnione są wszystkie założenia projektu. Procesor posiada cztery rejestry 16-bitowy, wynik obliczeń zapisywany jest w części starszej h i części młodszej l ośmiobitowych rejestrów. Możliwa jest realizacja trybu adresowania rejestrowego i natychmiastowego. Lista rozkazów obejmuje wszystkie wymagane rozkazy. Realizacja napisanych programów może być przeprowadzona w trybie całościowego wykonania lub w trybie pracy krokowej, przy czym w drugim trybie możliwe jest śledzenie aktualnie wykonywanej instrukcji. Możliwy jest zapis napisanego programu do pliku .txt oraz jego odczyt w celu dalszej edycji i ponownego uruchomienia.

Można uznać, że zaletą tej aplikacji jest jej prostota budowy oraz łatwość obsługi, natomiast wadą jest skomplikowany kod realizujący dany projekt. W kodzie uwzględniono dużo wyjątków, żeby program działał poprawnie i nie pokazywał błędów, co znacznie wydłużyło kod. Ponadto, przy aktualnej strukturze kodu, mała edycja programu wymaga dużej pracy i skomplikowanych rozwiązań, żeby wszystko dało się połączyć w całość.

Najwięcej trudności doświadczyliśmy podczas pisania funkcji, dokonującej obliczeń numerycznych oraz wyświetlającej te wynik w dwóch różnych trybach. Mieliśmy kilka pomysłów na ten temat, jednak wybraliśmy najbardziej optymalny (którego realizacja jest opisana w sprawozdaniu), żeby przy modyfikacji programu nie pojawiło się większych problemów z edycją samego kodu. Niestety, nie udało się nam zrealizować możliwości edycji poszczególnych linijek ze względu na brak czasu i skomplikowany kod. Mimo to, przy dłuższej pracy nad tym nie powinno być problematyczne ukończenie możliwości edycji jednej linijki. W dodatku nie mamy możliwości edycji i uruchomienia kodu po jego wpisywaniu do programu. Jest tak dlatego, że potrzebowalibyśmy zmodyfikować całą strukturę programu na potrzebę realizacji tego punktu, co przy braku czasu i skomplikowanych rozwiązaniach jest problematyczne.