Michał Dramiński, 176093
Mateusz Kolimaga , 175663

Sprawozdanie – Zadanie 1 z laboratoriów z przedmiotu Architektura Systemów Komputerowych

Opis zadania i założenia szczegółowe

Głównym celem zadania było stworzenie modelu programowego symulatora mikroprocesora z przyjaznym interfejsem graficznym użytkownika. Do spełnienia zadania, postanowiliśmy skorzystać z języka Python oraz biblioteki PyQt5. Program napisany jest w całości w języku angielskim, co ułatwia czytelność kodu.

Ważną cechą opisywanego oprogramowania jest fakt, że każdy element znajdujący się w interfejsie graficznym użytkownika jest zarazem obiektem w programie. Również z tego powodu praktycznie cały kod jest umieszczony w jednej klasie i podzielony na metody. Metody oddzielone są od siebie pojedynczą pustą linią. Wynika to z faktu, że kluczowym elementem składni języka Python, jest brak obojętności na znaki białe. Metody rzadko pobierają argumenty, gdyż prawie każda zmienna w programie jest atrybutem klasy. Jeśli natomiast argument jest podawany, to tylko wtedy gdy wartość ta jest tymczasowa i byłaby zbędna jako atrybut klasy, lub w sytuacji gdy inne rozwiązanie pogorszyłoby czytelność kodu.

Model posiada wszystkie funkcjonalności wymagane w poleceniu. Szczegółowe założenia dotyczące działania elementów interfejsu opiszę w następnym rozdziale, w punktach.

Opis przyjętych założeń programowych

Poniższe punkty odpowiadają tym porozmieszczanym na zdjęciu podglądowym znajdującym się na samym dole tego rozdziału.

1. 16-bitowe rejestry ogólnego przeznaczenia AX,BX,CX,DX, składające się z części starszej (H) oraz młodszej (L) – dane zawarte w wspomnianych rejestrach są widoczne po lewej stronie okna programu. Dostępna jest reprezentacja binarna oraz dziesiętna. Zawartość rejestrów można zmieniać nie tylko za pomocą napisanego programu, ale również bezpośrednio poprzez kliknięcie w dany bit. Jest

to bardzo komfortowa funkcjonalność, szczególnie przydatna podczas debugowania programu pisanego za pomocą interfejsu.

Uściślając, wspomniany bit jest tak naprawdę cyfrą na przycisku. Wszystkie przyciski natomiast to obiekty przechowywane jako elementy listy, dostępne pod słownikowym (struktura danych w Pythonie) kluczem odpowiadającym danemu rejestrowi. Można to zauważyć na poniższym zdjęciu programu:

```
self.buttons = {"AH": [], "AL": [], "BH": [], "BL": [], "CH": [], "CL": [], "DH": [], "DL": []}
```

Listy są na wycinku puste, lecz tuż pod spodem tworzone są one w pętli. Następnie przypisywane są im funkcje, które mają zostać wykonane po kliknięciu w nie (przykładowy przycisk poniżej):

```
self.buttons["AH"][0].clicked.connect(
    lambda _: self.buttons["AH"][0].setText("1") if self.buttons["AH"][0].text(_) == "0" else self.buttons["AH"][0].setText("0"))
self.buttons["AH"][0].clicked.connect(lambda: self.updateDecimal(128, self.dec["AH"],int(self.buttons["AH"][0].text())))
```

Jak widać powyżej, gdy przycisk zostanie wciśnięty jego wartość (napis) zmienia się na przeciwną, Co za tym idzie, zmieniana jest całkowita wartość przechowywana w rejestrze oraz ta widoczna w formie dziesiętnej poprzez okno tekstowe. Okna tekstowe są zapisywane w strukturze identycznej jak w przypadku przycisków.
Po stylu odwołania, można wywnioskować, że ten szczególny przycisk jest związany z najstarszym bitem rejestru AH.

Poza tym można tutaj zauważyć pewną konwencję łączenia danej metody z przyciskiem. Wykorzystywane jest połączenie atrybutu clicked i funkcji connect (są to funkcje dostępne w bibliotece PyQt5). W argumencie tej drugiej funkcji podaje się metodę, która ma być powiązana z kliknięciem. W powyższym przypadku jest to metoda obsługująca pole tekstowe z liczbą w formie dziesiętnej. Pozostałe elementy (przyciski, pola tekstowe itd.) na panelu korzystają z tego samego schematu łączenia obiektu z docelową funkcją. Z tego powodu skupię się na omawianiu metod.

Zawartość poszczególnych rejestrów można łatwo zresetować z wykorzystaniem przycisku „Reset”. Oto metoda związany z tym przyciskiem:

```
def restart(self,index):
    for i in range(8):
        self.buttons[index][i].setText("0")
    self.dec[index].setText("0")
```

Jak widać zerowane są wszystkie napisy na przyciskach oraz pole tekstowe wyświetlające wartość dziesiętną

2. Lista rozkazów – dostępna lista rozkazów jest dostępna w dziale „Instruction”. Wybór rozkazu zrealizowany jest w intuicyjny sposób, dzięki rozwijanym listom. Po lewej stronie możemy wybrać interesującą nas komendę, następnie miejsce docelowe i na końcu źródło, z którego mamy pobierać daną. Może być to dana zawarta w rejestrze lub bezpośrednio podana liczba w postaci dziesiętnej. Dzięki temu zrealizowana jest część zadania związana z dwoma trybami adresowania – rejestrowym oraz natychmiastowym. Gdy cała instrukcja będzie gotowa, możemy dodać ją do programu (miejsce tekstowe poniżej). Przy czym instrukcje mogą być zaimplementowane np. w poniższy sposób:

```
def MOV(self, dst, src):
    if src.isnumeric():
        out = self.getBinary(int(src))
        self.operation(dst,out)
    elif (dst[1] != 'X' and src[1] != 'X') or dst[1] == src[1]:
        out = []
        if dst[1] != 'X':
            out = [0 for _ in range(8)]
            for button in self.buttons[src]:
                out.append(int(button.text()))
        else:
            for button in self.buttons[src[0] + "H"]:
                out.append(int(button.text()))
            for button in self.buttons[src[0] + "L"]:
                out.append(int(button.text()))

        self.operation(dst,out)
    else:
        self.Msg.setText("MOV " + dst + "," + src + " - Error: sizes of registers are not equal")
        self.handleError()
```

Metoda pobiera jako argumenty miejsce docelowe oraz źródło danych. Jeśli źródło jest w postaci dziesiętnej, wartość zamieniamy na binarną za pomocą metody getBinary(). Następnie wykonywana jest operacja, to znaczy każdy przycisk zamienia swoją wartość na pasującą do odpowiadającego bitu w słowie maszynowym (lub bajtu, ponieważ

miejsce docelowe może być rejestrem NH/NL) uzyskanym wcześniej. Inna sytuacja ma miejsce, gdy następuje adresowanie rejestrowe. Wtedy rozmiary rejestrów muszą się zgadzać, a słowo maszynowe (lub bajt) jest układane z wykorzystaniem aktualnych wartości w rejestrze źródłowym (w formie szeregu przycisków).

Ten sam program możemy następnie wykonać w trybie całościowego wykonania lub pracy krokowej, używając przycisku „Execute" lub „Step exec.". Przy czym praca krokowa, będzie wykonywana z częstotliwością 1s na komendę. Oto metoda związana z wykonywaniem pracy krokowej:

```
def execStep(self):
    if self.instr_list_copy and self.step_mode:
        curr_instr = self.instr_list_copy.pop(0)
        self.step(curr_instr)

    elif self.instr_list_:
        self.step_mode = True
        self.Msg.setText("Step mode is ON")
        self.instr_list_copy = copy.deepcopy(self.instr_list)
        self.text_length = [x for x in range(1, 100)]
        self.textEdit.setText("")
        curr_instr = self.instr_list_copy.pop(0)
        self.step(curr_instr)
```

Do wykonywania trybu krokowego, potrzebna jest kopia listy instrukcji tworzonej w trakcie budowania programu. W czasie pracy krokowej kolejne wartości są konsekwentnie zdejmowanie (instrukcja pop()) z wspomnianej kopii i wyświetlane w oknie programu. W funkcji step natomiast wykonywana jest aktualna komenda:

```
def step(self, curr_instr):
    dot = ", "
    try:
        if curr_instr[2].isnumeric(): dot += "#"
    except AttributeError:
        pass
    num = self.text_length.pop(0)
    instr = curr_instr[0]
    dest = curr_instr[1]
    src = curr_instr[2]

    self.printProgramLine(num,instr,dest,dot,src)

    if curr_instr[0] == 'MOV':
        self.MOV(curr_instr[1], curr_instr[2])
    elif curr_instr[0] == 'ADD':
        self.ADD(curr_instr[1], curr_instr[2])
    elif curr_instr[0] == 'SUB':
        self.SUB(curr_instr[1], curr_instr[2])
    elif curr_instr[0] == 'PUSH':
        self.PUSH(curr_instr[2])
    elif curr_instr[0] == 'POP':
        self.POP(curr_instr[1])
    elif curr_instr[0][0:3] == 'INT':
        self.interrupt(curr_instr[0][3:5])

    if not self.instr_list_copy:
        self.turnOffStepMode()
    elif self.step_mode:
        self.timer.start(self.delay)
```

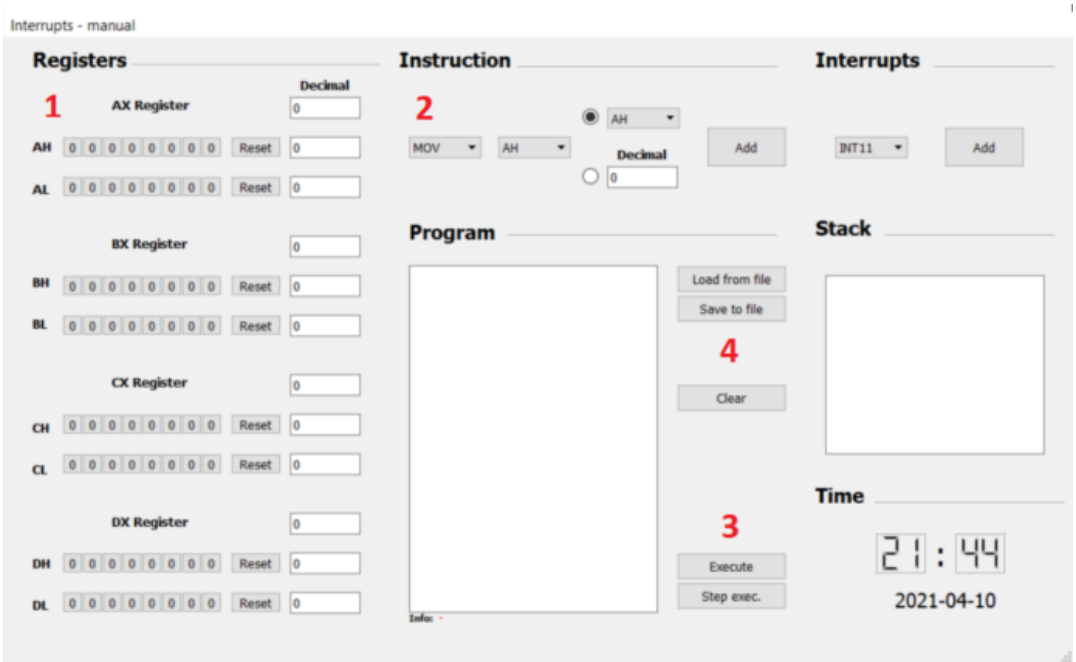
Instrukcja jest wypisywana w polu tekstowym programu a następnie, w zależności od tego jaka to instrukcja jest wykonywana poszczególna metoda. Jeśli kopia listy się skończy to wywoływana jest funkcja kończąca tryb krokowy, jeśli nie to odczekiwana jest 1s i pobierana jest następna instrukcja.

3. Śledzenie wykonywanych instrukcji w trybie pracy krokowej – linie kodu są ponumerowane. W wspomnianym trybie łatwo można śledzić program, gdyż co każdą sekundę pojawia się linia kodu, która została właśnie wykonana. Gdyby jednak w kodzie pojawił się błąd, wyświetlany jest odpowiedni do sytuacji komentarz pod polem tekstowym a sam program jest zatrzymywany i możemy go wyczyścić za pomocą przycisku „Clear” lub zrestartować za pomocą przycisku „Restart”. Poniżej umieszczam wycinek programu, związany z realizacją czyszczenia programu:

```
def clearProgram(self):
    self.Msg.setText("-")
    self.instr_list = []
    self.text_length = [x for x in range(1, 100)]
    self.instr_list_copy = []
    self.turnOffStepMode()
```

Na początku czyszczona jest wiadomość poniżej programu, zawierająca błąd, następnie czyszczona jest lista instrukcji. Restartowana jest lista zawierająca numerację wierszy, oraz czyszczona jest kopia listy instrukcji. Taka kopia była potrzebna w przypadku używania trybu krokowego. Wyłączony jest również tryb pracy krokowej, jeśli był włączony.

4. Wczytywanie oraz zapisywanie do pliku – te opcje możliwe są do zrealizowania za pomocą przycisków znajdujących się po prawej stronie w dziale „Program”. Działanie jest intuicyjne, format zapisywanych/wczytywanych plików to „.txt”.



Rys.1 Podglądowe zdjęcie programu, z zamieszczoną numeracją.
Zdjęcie przedstawia również elementy programu, które są związane z następnym zadaniem laboratoryjnym.

Uzyskane wyniki oraz wady i zalety programy

Program w całości realizuje wszystkie zadane mu funkcje, stąd wyniki są zadowalające. Do zalet możemy zaliczyć:

- Wygodny i uporządkowany interfejs
- Kompleksowa możliwość zmiany zawartości rejestrów
- Prezentacja binarna i dziesiętna zawartości rejestrów
- Kontrola i komunikacja błędów programu
- Każdy element interfejsu jest obiektem, łatwa czytelność kodu:
- Łatwa rozszerzalność programu i łatwa kontrola jego zawartości
- Adaptacja interfejsu w zależności od wyboru instrukcji przez użytkownika
- Program nie zawiesza się i działa płynnie
- Wieloplatformowość

Jeśli chodzi o wady to można odczuwać brak kolejnej formy reprezentacji wartości – heksadecymalnej. Kolejną wadą może być brak innych typów akceptowanych plików, w których zapisujemy program (jest jedynie możliwość zapisu do txt). Może przeszkadzać również brak możliwości zmiany wartości rejestru przez zmianę zawartości pola tekstowego z formą dziesiętną.