UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
W LUBLINIE
Wydział Matematyki, Fizyki i Informatyki

Kierunek: **informatyka**

Bartosz Choma
nr albumu: 303839

Implementacja maszyny URM na Arduino

Unlimited Register Machine implementation on Arduino

Praca licencjacka
napisana w Katedrze Cyberbezpieczeństwa
Instytutu Informatyki UMCS
pod kierunkiem **dra hab. Jarosława Byliny**

Lublin 2023

Spis treści

Wstęp **5**

1 Opis działania maszyny URM **7**

1.1 Podstawowe zagadnienia 7

1.2 Przykłady programów 8

2 Tworzenie układu implementującego maszynę URM **11**

2.1 Projektowanie klawiatury 11

2.2 Projektowanie wyświetlaczy 14

2.3 Projektowanie przełączników 15

3 Tworzenie oprogramowania **21**

3.1 Podstawowe metody 21

3.2 Działanie kodu 24

3.3 Informacje dodatkowe 28

3.3.1 Obsługa klawiatury 28

3.3.2 Obsługa wyświetlaczy 30

3.3.3 Interpretacja wartości z dzielnika napięcia 31

4 Podsumowanie **33**

Spis tabel **35**

Spis rysunków **38**

Bibliografia **39**

4:9506755390

Wstęp

Teoria Obliczalności jest to dział teorii obliczeń zajmujący się badaniem, jakie problemy są rozwiązywalne przy użyciu komputerów teoretycznych. Przykładem takiego komputera jest Maszyna URM (Unlimited Register Machine).

Koncepcja URM opiera się na 4 podstawowych poleceniach, które wykonują obliczenia na nieskończonej liczbie rejestrów. Dowodząc, że dane działanie jest URM-obliczalne należy rozbić poszczególne kroki na podstawowe komendy, które z definicji maszyny URM są URM-obliczalne [1, 2].

Celem pracy jest przeniesienie konceptu komputera teoretycznego na świat rzeczywisty w celu automatyzacji wykonywania obliczeń oraz fizyczne przedstawienie obliczeń maszyny URM w czasie wykonywania programu.

Fizyczna interpretacja niesie za sobą niewiele problemów, które są dosyć kluczowe. Mianowicie sprzęt fizyczny nie jest w stanie przedstawić obliczeń na nieskończonej ilości rejestrów, tym samym wartości w samych rejestrach są ograniczone o typ danych. W ostateczności ograniczenie pamięciowe nie będzie miało większego znaczenia, z racji na to, że podstawowe programy (np. dodawanie) nie potrzebują dużej ilości rejestrów, a wyniki obliczeń nie koniecznie muszą zwracać dużych wyników.

Uznając powyższe zasady ograniczenia sprzętowego, można dojść do różnych sposobów przechowywania danych. Przykładowo, można wykorzystać układ 256KB pamięci RAM, gdzie każdy bajt będzie działał jako 8-bitowy rejestr. Dla uproszczenia wykonywania obliczeń będziemy wykorzystywali Arduino Uno, tym samym jako rejestry będą mi służyć elementy tablicy w programie.

W pierwszym rozdziale opisujemy podstawowe założenia i działanie maszyny URM oraz przykłady programów URM-obliczalnych. Następny rozdział opisuje układ, dzięki którego pomocy, będzie umożliwiało wprowadzenie danych do mikrokontrolera, oraz ich wyświetlenie. Na końcu, omówimy działanie programu wykonującego obliczenia, emulujące działanie maszyny URM.

5:5505331476

6:4307379206

Rozdział 1

Opis działania maszyny URM

Niniejszy rozdział jest oparty na [3].

1.1 Podstawowe zagadnienia

Rejestry URM są oznaczane jako R_n, zaczynając od $n = 1$. Zawartość rejestrów oznaczamy r_n, podając odpowiedni indeks rejestru.

Zawartość rejestrów może być zmieniania za pomocą odpowiedzi na instrukcje oznaczane I. Skończona lista instrukcji tworzy program ($P = I_1, I_2, I_3...$).

Podstawowe polecenia maszyny URM przedstawia tabela 1.1.

Nazwa	Instrukcja	Opis
Zero	Z(n)	Zamienia rejestr r_n na 0
Następnik	S(n)	Dodaje 1 do r_n ($r_n := r_n + 1$)
Transfer (kopiowanie)	T(m,n)	Zastępuje r_n przez r_m ($r_n := r_m$)
Skok	J(m, b, q)	Jeżeli $r_m = r_n$ skocz do instrukcji q, w przeciwnym razie przejdź do kolejnej instrukcji

Tabela 1.1: Podstawowe instrukcje maszyny URM.

Co więcej, po ustaleniu n instrukcji, po wyjściu poza program (przejście z instrukcji I_n na I_{n+1}) powoduje zatrzymanie programu, czy też wykonanie polecenia STOP.

Jak widać z tabeli 1.1, URM w odróżnieniu od Maszyny Turinga, nie porusza się po rejestrach jeden po drugim, tylko od razu wykonuje działania na komórkach o

7:7007261140

wyznaczonym indeksie, nie zmieniając wartości w innych rejestrach niżę te, podane w argumentach.

Maszynę oznacza się jako M, razem z początkowym ułożeniem rejestrów. Przykładowo dla $M(0,3,5)$ rejestry pokazane są w tabeli 1.2.

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	...
0	3	5	0	0	0	0	0	0	0

Tabela 1.2: Układ początkowy rejestrów maszyny $M(0,3,5)$.

1.2 Przykłady programów

Przykład 1.1. *Udowodnij, że $f(x,y) = x + y$ jest URM-obliczalne.*

Program będzie przebiegał w następujący sposób. Dla maszyny $M(x,y)$, wyznaczamy miejsce dla wartości pomocniczej (r_3), następnie w instrukcji I_1 sprawdzamy, czy nasza wartość pomocnicza jest równa wartości y (r_2). Jeżeli wartości są równe, kończymy program. W przeciwnym razie zwiększamy wartość x (r_1) oraz wartość pomocniczą o 1. Następnie wykonujemy bezwarunkowy skok do początku programu.

Taki plan działania ukazuje konfigurację podczas obliczeń pokazaną w tabeli 1.3, gdzie x i y są z góry ustalone jako argumenty, a k będzie się zmieniać wraz z działaniem programu. Rejestr R_1 służy do przechowania odpowiedzi na koniec działania programu.

R_1	R_2	R_3	R_4	...
$x+k$	y	k	0	0

Tabela 1.3: Początkowa konfiguracja rejestrów w programie wykonującym dodawanie.

Program zwracający wynik oddawania przedstawia tabela 1.4.

Z racji że powyższy program składa się wyłącznie z poleceń URM-obliczalnych (podstawowe polecenia są URM-obliczalne z definicji), $f(x,y) = x + y$ jest URM-obliczalne.

Przykład 1.2. *Udowodnij że $f(x) = \lfloor \frac{x}{4} \rfloor$ jest URM-obliczalne.*

Działanie programu będzie wyglądało w następujący sposób. Dla maszyny $M(x)$ wyznaczane są 2 miejsca na wartości pomocnicze n oraz m (r_2 oraz r_3). Cały program będzie polegał na zwiększaniu m, porównywaniu x oraz m, gdzie n jest zwiększane o

8:8911963142

n	I_n	Opis
1	J(3, 2, 5)	Jeżeli $r_3 = r_2$, przejdź do instrukcji I_5 (koniec programu)
2	S(1)	Dodaj 1 do r_1
3	S(3)	Dodaj 1 do r_3
4	J(1, 1, 1)	Bezwarunkowy skok do I_1

Tabela 1.4: Program wykonujący dodawanie.

1, co 4-te zwiększenie liczby m ($\lfloor m/4 \rfloor = n$). Wynikiem będzie liczba n, która zostanie przeniesiona na R_1.

Konfiguracja rejestrów jest przedstawiona w tabeli 1.5, a przebieg programu jest przedstawiony w tabeli 1.6.

R_1	R_2	R_3	R_4	...
$\lfloor \frac{x}{4} \rfloor$	n	m	0	0

Tabela 1.5: Układ rejestrów maszyny obliczającej $\lfloor \frac{x}{4} \rfloor$.

n	I_n	Opis
1	J(1, 3, 11)	Jeżeli $r_1 = r_3$, przejdź do instrukcji I_{11} (znaleziono wynik)
2	S(3)	Dodaj 1 do r_3
3	J(1, 3, 11)	
4	S(3)	
5	J(1, 3, 11)	
6	S(3)	
7	J(1, 3, 11)	
8	S(3)	
9	S(2)	Dodaj 1 do r_2
10	J(1, 1, 1)	Bezwarunkowy skok do I_1
11	T(2, 1)	Zapisanie wyniku w R_1

Tabela 1.6: Program obliczający $\lfloor \frac{x}{4} \rfloor$.

Z racji że powyższy program składa się wyłącznie z poleceń URM-obliczalnych, $f(x) = \lfloor \frac{x}{4} \rfloor$ także jest URM-obliczalne.

9:9691766508

Rozdział 2

Tworzenie układu implementującego maszynę URM

Jak zostało wspomniane we wstępie, dane będą przechowywane oraz obliczane za pomocą Arduino Uno. Koncepcja układu oraz programu jest następująca:

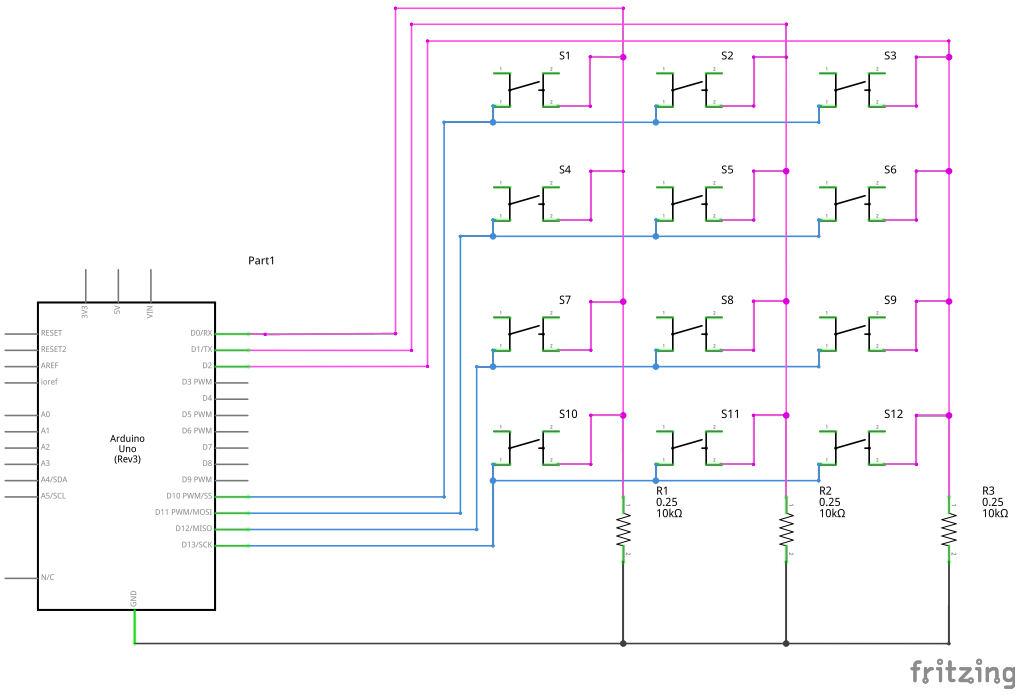
1. Stan rejestrów będzie przechowywany w tablicy typu `unsigned char[256]`.
2. Polecenia będą przechowywane w tablicy obiektów posiadające pola `command` oraz `args[]`.
3. Układ będzie posiadał tryby (taki jak: Setup, Input, Compile, Reset) ustawiane za pomocą przełączników.
4. Dane będą wprowadzane za pomocą klawiatury numerycznej z wartościami od 0 do 9, oraz przycisków ACCEPT i DECLIN.
5. Wprowadzane dane, oraz obliczenia będą wyświetlane na 4 wyświetlaczach 8-segmentowych.

2.1 Projektowanie klawiatury

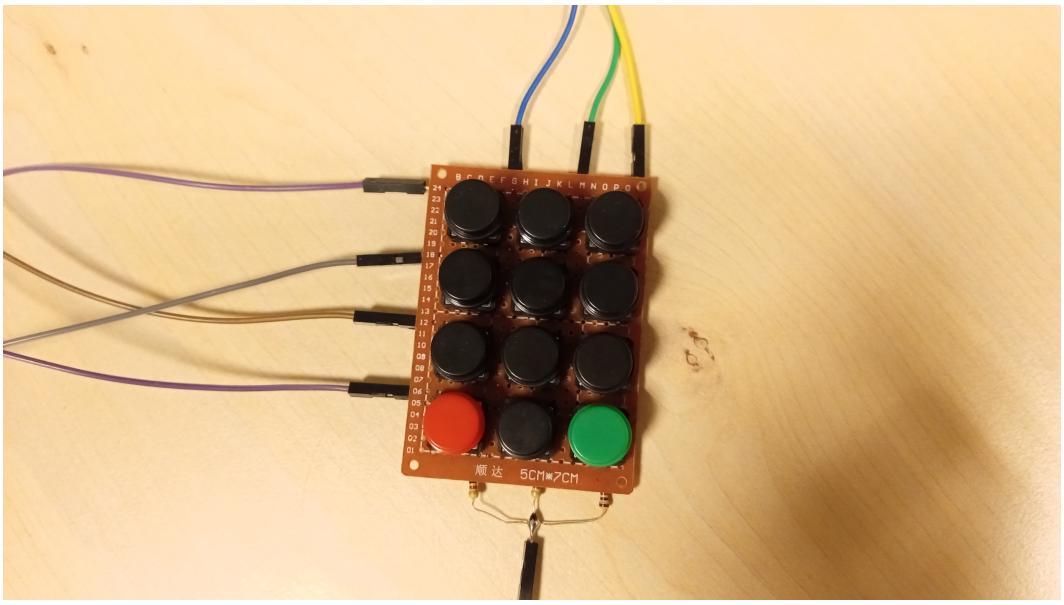
Grafika 2.1 pokazuje podstawowy koncept klawiatury. Rezystory pełnią funckję „ściągacza“ sygnału do wartości 0. Niebieskie kable to output z Arduino, wysyłany kolejno do każdego z rzędu przycisków. Różowe połączenia to jest input odbierany przez mikrokontroler.

Działanie klawiatury polega na wysłaniu sygnału na jeden z rzędów klawiatury oraz sczytywaniu kolumny. Wiedząc to, który rząd sczytujemy oraz która kolumna posiada na sobie napięcie 5V, wiadomo dokładnie który z przycisków został naciśnięty.

11:8905906889



Rysunek 2.1: Schemat przedstawiający podstawową koncepcję klawiatury.



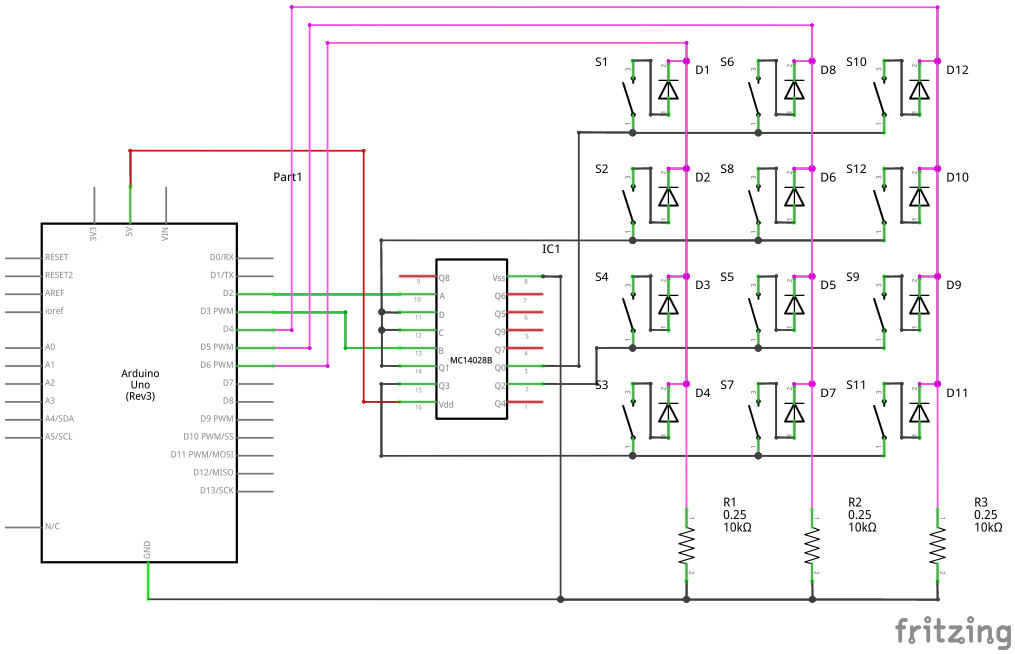
Rysunek 2.2: Zdjęcie przedstawiające układ z rysunku 2.1.

Po wykonaniu testowego układu (2.2) zauważyliśmy, że jest on niepełny. Mianowicie naciskając 2 przyciski w jednej kolumnie, Arduino nie odczytuje żadnej wartości. Jest to spowodowane tym, że tworzone jest zwarcie, między pinem wysyłającym sygnał na jeden rząd klawiatury oraz pinu bez sygnału na drugim rzędzie. Można temu zapobiec używając diod zaporowych 1N4007.

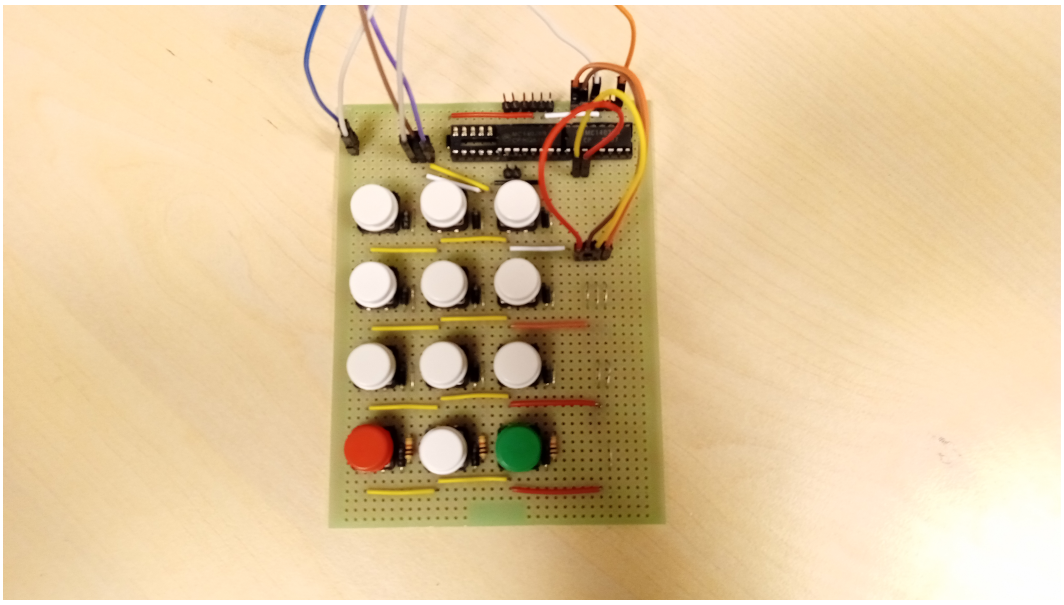
12:8579091372

Dodatkowo, klawiatura potrzebuje dużej ilości pinów. Z racji, że jeszcze potrzebujemy obsłużyć wyświetlacze 8-segmentowe, przyda się jak największa redukcja pinów. Do tego posłużymy się dekoderem MC14028 [6].

Dekoder sprawi że zamiast używać 4 pinów wyjściowych, będą potrzebne tylko 2 bity, które dekoder zamieni na 4 oddzielne ścieżki sygnału. Układ przedstawią rysunki 2.3 oraz 2.4.



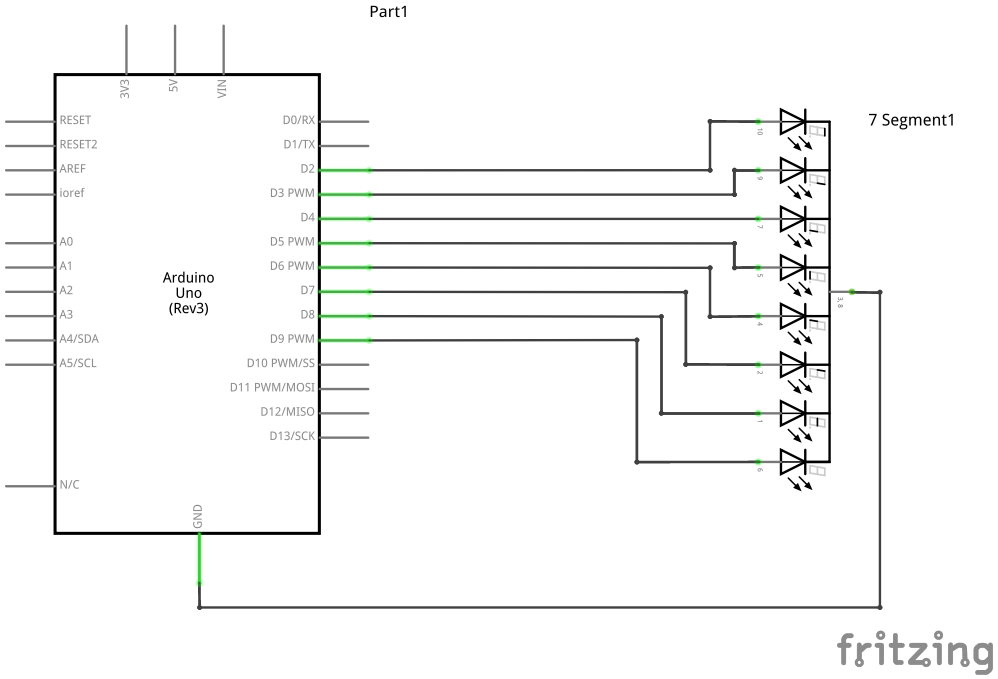
Rysunek 2.3: Schemat poprawnie zaprojektowanej klawiatury.



Rysunek 2.4: Zdjęcie przedstawiające układ opisany na rysunku 2.3.

2.2 Projektowanie wyświetlaczy

Wyświetlacz 8-segmentowy jest bardzo wymagający pod względem ilości pinów. Mianowicie, każdy z elementów wyświetlacza (7 diod na wyświetlenie cyfry oraz 1 dioda na kropkę) wymaga osobnego pasma sygnału, połączone do wspólnego uziemienia, jak pokazuje rysunek 2.5.

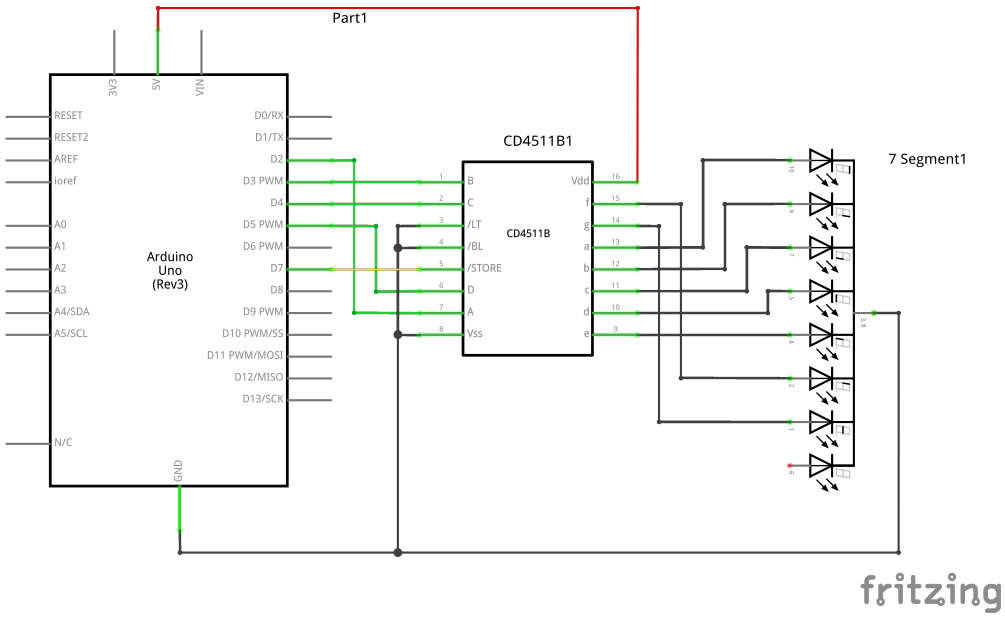


Rysunek 2.5: Schemat przedstawiający poprawne podłączenie wyświetlacza 8-segmentowego.

Z pomocą przychodzi dekoder 7-segmentowy (użyty został układ CD411B [4]). Działa on podobnie jak dekoder MC14028, wymieniony wyżej. Po podaniu wartości binarnej dopasowywuje swoje wyjścia tak, aby wyświetlić daną wartość na wyświetlaczu w formie dziesiętnej. Wykoszystanie tego układu pokazuje rysunek 2.6.

Tym samym zmniejsza ilość potrzebnych pinów z 7 do 4 (dodatkowo potrzebny jest pin do wejścia `/STORE`, o czym będzie wspomniane później).

Docelowo, chcemy użyć 4 wyświetlacze, co dalej sprawia problem ilości pinów potrzebnej do ich obsługi. Pomocne do tego jest wejście na układzie CD4511, który odpowiada za przechowywanie danej wartości do wyświetlenia (blokuje dane wejściowe w taki sposób, aby zapamiętać i przetrzymać wcześniej podaną wartość). Jest to wejście o nazwie `/STORE`. Jak wskazuje znak `/`, jest to wejście sterowane wartością 0. Inaczej mówiąc: przy podaniu wartości 0, wartość wejściowa jest przekazywana na wyświetlacz, a przy wartości 1, wzór na wyświetlaczu nie zmienia się razem ze zmianą wartości



Rysunek 2.6: Schemat przedstawiający użycie układu CD4511B.

wejściowych.

Taka mechanika pozwala na współdzielenie wartości wejściowej do dekoderów 7-segmentowych, za pomocą wyboru odpowiedniego wyświetlacza, do którego potrzebny jest zapis danych, tym samym redukując ilość danych wejściowych z 16, na 4 (dla wartości do wyświetlenia) + 4 (pasmo do wybrania wyświetlacza, które można zastąpić dekoderem tak jak w przypadku klawiatury).

Z racji, że dekoder MC14028 wysyła pojedyncza pasmo z wartością 1, potrzebuję zanegować te pasma, aby mieć pojedyncze pasmo z wartością 0, oraz resztę z wartością 1. Do tego posłużymy się układem negującym sygnał - CD4049 [5].

Pełne układy są przedstawione na rysunku 2.7 oraz 2.8.

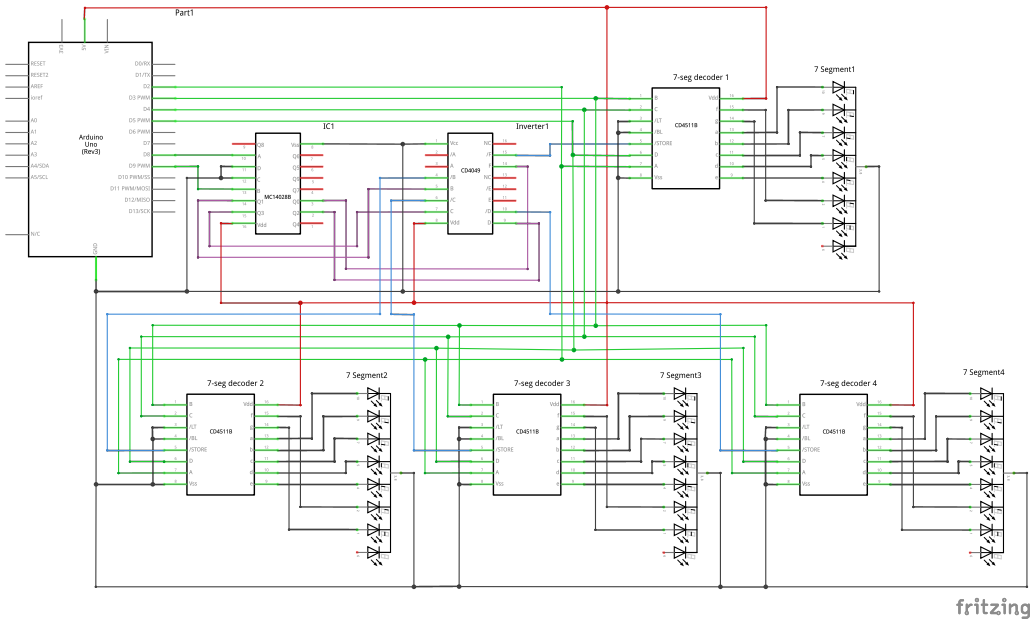
2.3 Projektowanie przełączników

Ostatnim elementem układu są przełączniki. Idealny do naszego planu będzie dzielnik napięcia z rezystorów.

Układ będzie posiadał 5 przełączników. Naciśnięcie któregokolwiek z przycisków stworzy unikalną wartość napięcia. Wartość z dzielnika napięcia będzie odczytywania za pomocą analogowego portu Arduino.

Dobieranie rezystorów do wykonania układu wymaga wykonaniu paru obliczeń. Napięcie wyjściowe (Uwy) jest równe: $U_{wy} = (U_{we}/R + R1) * R1$, gdzie:

U_{we} — napięcie wejściowe,



Rysunek 2.7: Pełny schemat przedstawiający sterowanie 4-roma wyświetlaczami.

R_1 — rezystancja między uziemieniem a napięciem wyjściowym,

R — rezystancja między napięciem wejściowym a wyjściowym.

5 przełączników będzie posiadało 6 unikalnych wartości odpowiadających kolejno za:

1. None
2. Run MODE
3. Select MODE:
   (a) NONE
   (b) Input Program
   (c) Normal Compile
   (d) Delay Compile
   (e) Setup Machine (ustawienie rejestrów)
4. Set Delay [200] (dla MODE == 3)
5. DISPLAY TEST
6. RESET (ustawienie programu i rejestrów na 0)

Wstępne założenie do wykonania obliczeń:

$U_{we} = 5V$

$R_1 = 4,7kR$

Każdy z przycisków zwiększa rezystancję na układzie o 1kR.

Korzystając ze wzoru, zaokrąglone wartości wynoszą:

$U_1 = 4.12V$

$U_2 = 3.5V$

$U_3 = 3.05V$

$U_4 = 2.7V$

$U_5 = 2.42V$

Analogowe wejście Arduino przyjmuje wartości od 0 do 1023, które odpowiada zakresowi od 0V do napięcia wyjściowego Arduino. Uznając, że wartość maksymalna odpowiada 5V, jak zostało wyżej wspomniane, można przemapować wartość napięcia na wartości na Arduino.

Wartości dla napięcia U_n, które odczyta Arduino:

$V_1 = 843$

$V_2 = 718$

$V_3 = 624$

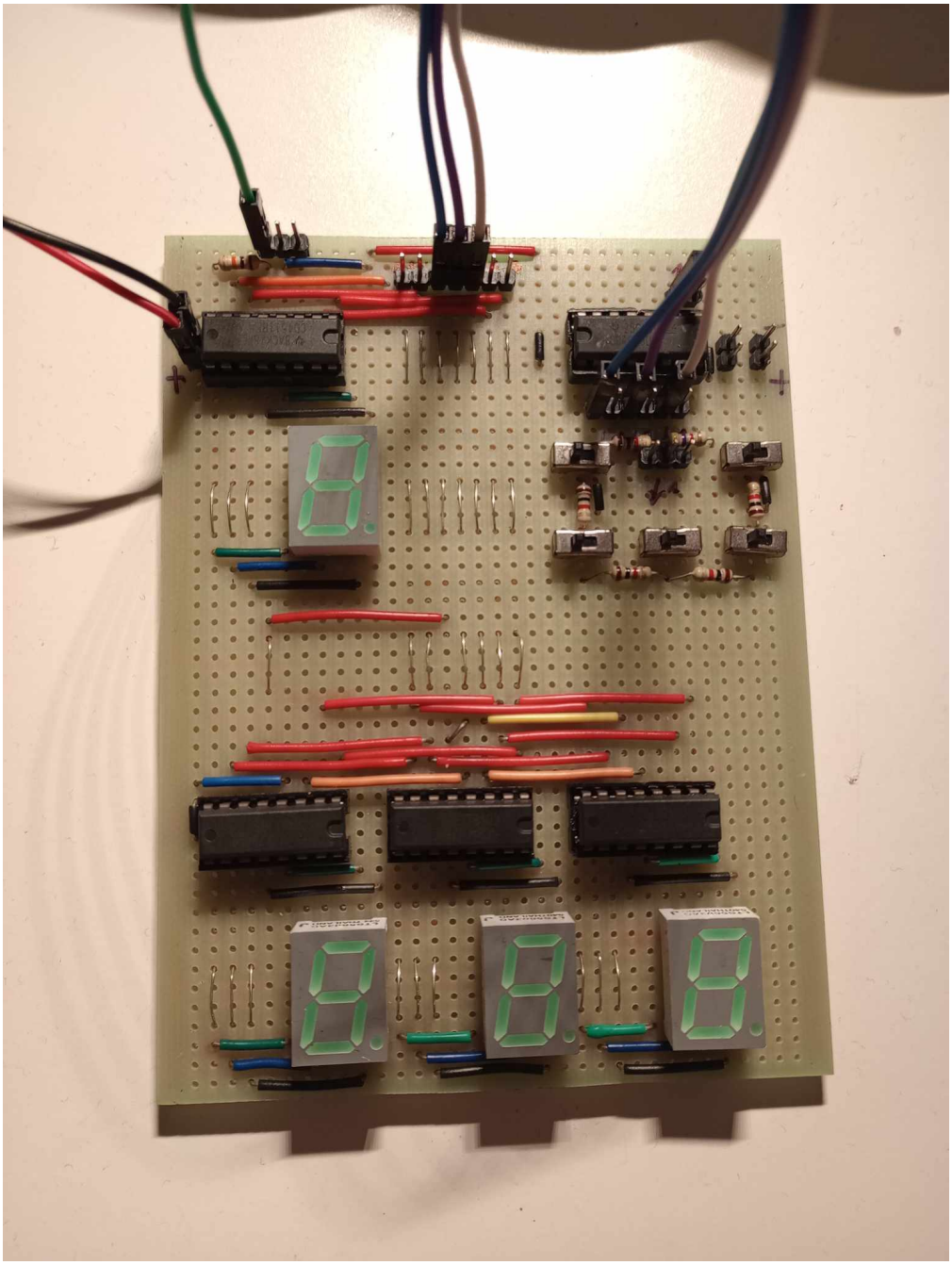
$V_4 = 552$

$V_5 = 495$

Rzeczywiste wartości nie będą tak dokładne, ale z teoretycznej perspektywy kodu, te wartości są na tyle odległe od siebie, że błędy pomiarowe, nie będą miały dużego wpływu na rozróżnienie, który z przełączników został wybrany.

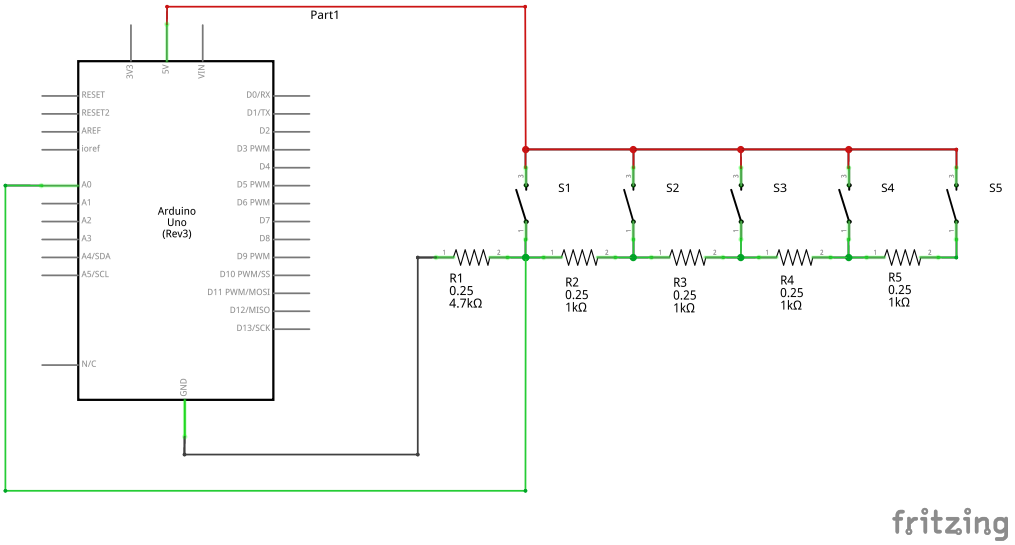
Schemat układu przedstawiającego działanie dzielnika napięcia przedstawia rysunek 2.9.

17:1269097238



Rysunek 2.8: Zdjęcie układu z wyświetlaczami.

18:1140819531



Rysunek 2.9: Układ przedstawiający układ dzielnika napięcia z przełącznikami

Rozdział 3

Tworzenie oprogramowania

Tworzenie programów na Arduino w większości wygląda podobnie. Należy odpowiednio ustawić porty wejścia/wyjścia, a następnie w odpowiedni sposób je obsłużyć. Dopiero to, co się będzie działo z danymi z portów, praktycznie odróżnia jeden program od drugiego, więc w tym dziale skupimy się głównie na implementacji działania maszyny URM, oraz ważniejszych aspektach w obsłudze wyświetlaczy, czy też klawiatury.

3.1 Podstawowe metody

Całość kodu oparta jest na klasie o nazwie `URM`, dla sprostowania sposobu działania metod, należy dodać, że ta klasa posiada pola opisane na schemacie 3.1.

```
// Default URM memory setup
std::vector<int> default_data;
// Data used in calculations
std::vector<int> data;
// Set of commands
std::vector<command> commands;
// Set of programs
std::vector<custom_command> custom_commands;

int current_command = 0;
bool SHOW_ANWSER_ONLY;
bool SHOW_INDEX;
```

Rysunek 3.1: Pola prywatne głównej klasy `URM`.

21:5473992955

Podstawowe polecenia maszyny URM z tabeli 1.1 są przedstawione w formie kodu, na rysunkach 3.2, 3.3, 3.4, 3.5.

```
void S(int index){
    if (index < 0) return;
    if (index >= data.size())
        append_data(index);

    data[index]++;
}
```

Rysunek 3.2: Następnik (Succesor) — zwiększa wartość w danej komórce.

```
void Z(int index){
    if (index < 0) return;
    if (index >= data.size())
        append_data(index);

    data[index] = 0;
}
```

Rysunek 3.3: Wyzerowanie (Zero) — zeruje wartość w danej komórce.

```
void T(int index_from, int index_to){
    if (index_from < 0) return;
    if (index_to < 0) return;

    if (index_from >= data.size() || index_to >= data.size())
        append_data(std::max(index_from, index_to));

    data[index_to] = data[index_from];
}
```

Rysunek 3.4: Transfer — kopiuje wartość z jednej komórki do drógiej.

```
void J(int v1, int v2, int index){
    if (v1 < 0) return;
    if (v2 < 0) return;
    if (index < 0) return;
    if (v1 >= data.size()) return;
    if (v2 >= data.size()) return;

    if (data[v1] == data[v2])
        current_command = index - 1; // while-loop in run()
   increments current_command by 1, bo it needs to be bypassed
}
```

Rysunek 3.5: Skok (Jump) — jeżeli wartości z 2 komórek są takie same, przechodzi do odpowiedniego polecenia.

Do ułatwienia pisania kodu, utworzona został typ wyliczeniowy `command_type` oraz struktura `command` (rys. 3.6), trzymająca w sobie informacje o poszczególnym poleceniu do wykonania, takie jak typ oraz odpowiednie argumenty.

```
enum command_type{
    Succesor,
    Zero,
    Transfer,
    Jump
};

struct command{
    int* variables;
    command_type type;

    command(){};

    command(command_type type, int v1, int v2=0, int v3=0){
        this->type = type;
        if (type == Jump){ [...] }
        if (type == Transfer){ [...] }
        [...]
    }
};
```

Rysunek 3.6: Struktura przechowująca informacje o poleceniu.

Dodatkowo, w kodzie programu jest struktura `custom_command` (rys. 3.7), która będzie dokładniej opisana w następnej sekcji.

```
struct custom_command{
    std::vector<command> commands;
    std::string name;

    custom_command(std::string name, std::vector<command> commands){
        this->name = name;
        this->commands = commands;
    }
};
```

Rysunek 3.7: Struktura pomocnicza do zapisywania programów.

3.2 Działanie kodu

Dla lepszego zrozumienia działania kodu, przekazywane wartości będą pokazane w postaci statycznie ustawionych zmiennych w formacie języka C++.

Inicjalizacja oraz podstawowe wartości dla programu są przedstawione na rysunku 3.8.

```
int data[2] = {2, 5};

command commands[4] = { // addition
        command(Jump, 1, 2, 4),
        command(Succesor, 0),
        command(Succesor, 2),
        command(Jump, 0, 0, 0)
};

URM urm(false, false);

urm.setup(data, 2);

for (int i=0; i<4; i++) {
    urm.add_command(commands[i]);
}
```

Rysunek 3.8: Przykładowe przygotowanie obiektu klasy URM do wykonania obliczeń. Na przykładzie widać program wykonujący działanie $2 + 5$, na podstawie 1.4.

W konstruktorze ustawiane są kolejno flagi `SHOW_ANWSER_ONLY` oraz `SHOW_INDEX`. Metoda `setup()` ustawia tablicę `data` o długości 2 jako wartość domyślną rejestrów maszyny URM, jak zostało to pokazane w tabeli 1.3.

Metoda `add_command()` przyjmuje jako argument obiekt typu `command` oraz dodaje

dane polecenie do listy w obiekcie.

```
void run(){

    if(commands.empty()) return;

    current_command = 0;
    data = default_data;

    print_data();

    while(current_command < commands.size()) {

        switch (commands[current_command].type) {
            case Succesor:
                S(*commands[current_command].variables);
                break;
            case Zero:
                Z(*commands[current_command].variables);
                break;
            case Translate:
                T(commands[current_command].variables[0], commands[current_command].variables[1]);
                break;
            case Jump:
                J(commands[current_command].variables[0], commands[current_command].variables[1],
                  commands[current_command].variables[2]);
                break;
        }
        current_command++;

        if (!SHOW_ANWSER_ONLY) print_data();

    }
```

Rysunek 3.9: Główna metoda wykonująca polecenia na rejestrach.

Jak widać z rysunków 3.9 oraz 3.10, metoda `run()` jest miejscem gdzie wykonywane są polecenia na kopii domyślnego ustawienia rejestrów. Metoda `print_data()` wypisuje zawartość zmiennej data w odpowiednim formacie. Pętla `while` jest odpowiedzialna za przechodzenie poprzez poszczególne polecania z listy, za pomocą zmiennej `current_command` (z racji że wykorzystywane jest polecenie skoku, nie można przechodzić liniowo przez listę poleceń. Potrzebna jest do tego zmienna wyznaczająca

```
    std::cout << "Anwser: " << data[0] << std::endl;

    std::cout << "Add program as custom funciton? (y/n): ";
    std::string anwser;
    std::cin >> anwser;

    if (anwser == "y"){
        std::cout << "Name a function : ";
        std::cin >> anwser;
        add_custom_auto(anwser);
    }

}
```

Rysunek 3.10: Rys. 3.9, ciąg dalszy.

aktualny indeks). W momencie, kiedy program wyjdzie poza listę poleceń, pętla zatrzymuje się, a następnie jest wypisywana odpowiedź z pierwszego rejestru.

Na samym końcu metody (rys. 3.10), program pyta o zapisanie aktualnego programu, po zaakceptowaniu pytania oraz zapisaniu nazwy programu, następuje stworzenie obiektu struktury `custom_command`, który jako pole, posiada aktualną listę poleceń, którą można wywołać za pomocą nazwy, w przyszłych programach.

Przykład wykorzystania `custom_command` jest przedstawiony na rysunku 3.11.

```
int data2[] = {10, 5, 0, 0, 0, 0};

urm.setup(data2, 6);

command commands2 [5] = {
        command(Translate, 1, 3),       //0
        command(Translate, 0, 1),       //1
        command(Zero, 0),               //2
        command(Jump, 3, 4, 11),        //3
        command(Zero, 2)                //4
};

for (int i=0; i<5; i++)
    urm.add_command(commands2[i]);

urm.add_command("add");                  //5, 6, 7, 8
urm.add_command(command(Succesor, 4));   //9
urm.add_command(command(Jump, 0, 0, 3)); //10
```

Rysunek 3.11: Ustawienie programu maszyny URM do wykonywania mnożenia 10 oraz 5, za pomocą wcześniej zapisanego programu `add` z rysunku 3.8 (w komentarzach, są podpisane indeksy poleceń).

3.3 Informacje dodatkowe

Jak było wspomniane na początku rozdziału, większość kodu tworzonego na Arduino, to obsługa portów. W ramach tej pracy, nie pasowałoby umieszczać całego kodu z mikrokontrolera, z racji, że większość kodu nie odnosi się do tematu. Jednak jest parę ważnych aspektów w obsłudze układów wymienionych w rozdziale 2. Wszystko wynika z tego, że układy fizyczne nie są idealne, a czas wykonywania akcji nie jest zerowy.

3.3.1 Obsługa klawiatury

Ważnym aspektem w tworzeniu programu do obsługi tego typu klawiatury (rys. 2.3) jest odpowiednie zadecydowanie, kiedy uznać przycisk za poprawnie wciśnięty. Najprościej jest uznać, że przyciśnięty przycisk w kodzie, następuje w momencie, gdy wartość przycisku zmienia się z 0 na 1. Oto przykład z wykorzystaniem 2 zmiennych, `new_value` oraz `old_value` (rys. 3.12).

```
new_value = digitalRead(BUTTON);

if (new_value == 1){
    if (old_value == 0){
        handle_button_press();
    }
}

old_value = new_value;
```

Rysunek 3.12: Najprostszy i podstawowy sposób weryfikacji naciśniętego przycisku.

O ile jest to proste rozwiązanie, z którego pomocą uzyskamy oczekiwany wynik dla większości przypadków, lepiej się zabezpieczyć przed anomaliami wartości sczytywanych z układu, możemy wykorzystać większą historię naciśniętych przycisków.

```
//Check for button value from last 3 loops
bool checkValue(){
    if (value[0])
        if(value[0]==value[1] && value[1]==value[2] && value[2]!=
   value[3]){
            //Serial.println("Button " + name +" pressed!");
            return true;
        }
    //Serial.println("No change at " + name);
    return false;
}

//Update values and return button value
void update(bool new_value){
    value[3] = value[2];
    value[2] = value[1];
    value[1] = value[0];
    value[0] = new_value;
}
```

Rysunek 3.13: Przykład obsługi przycisku z historią 3 poprzednich stanów.

Rozwiązanie na rysunku 3.13 sprawia, że przycisk musi być wciśnięty przez przynajmniej 3 cykle, gdzie 3 cykle temu, nie był wciśnięty. Zakładając zastosowanie `delay(100)` w kodzie, zatrzymujący kod, na 100 ms oznacza, że przycisk musi być wciśnięty przez przynajmniej 0.3 sekundy, co rozwiązuje sytuacje nieprzewidzianych, chwilowych wzrostów napięcia na porcie, czy też przypadkowego naciśnięcia przycisku.

29:5862573192

3.3.2 Obsługa wyświetlaczy

Obsługiwanie wyświetlaczy, także ma swoje ważne aspekty. Mianowicie, po przesłaniu wartości do wyświetlenia na jednym wyświetlaczu, przed zmianą wartości, którą chcemy wybrać dla kolejnego wyświetlacza, należy wyłączyć jeden wyświetlacz, zmienić przesyłane dane dla kolejnego wyświetlacza, dopiero po tym włączenie odpowiedniego wyświetlacza. Jeżeli zapomnimy o tych krokach, nastąpi zjawisko tak zwanych „duchów“, gdzie wyświetlacze próbują wyświetlić parę wartości na raz. Dodatkowo należy pamiętać o symbolicznym opóźnieniu, aby się upewnić, że układ scalony zdążył zareagować na zmianę danych (rys. 3.14).

```
void handle_display(int display, int value){

    select_display(4); // Select non-existing display
    delay(5);

    set_display_value(value); // Set correct output value
    delay(5);

    select_display(display); // Select correct display
    delay(5);
}
```

Rysunek 3.14: Część kodu, odpowiadająca za poprawny wybór wyświetlacza i ustalenia wartości do wyświetlenia.

3.3.3 Interpretacja wartości z dzielnika napięcia

Obliczenia z sekcji 2.3 w teorii się zgadzają, lecz w praktyce wartości okazały się mniejsze. Mianowicie, zakres wartości przełączników wynosi około 450-765 (gdzie przewidywany zakres był od 495 do 843) oraz wartości oscylują się w przedziale 15 jednostek. Tym samym, bez żadnego włączonego przełącznika, mikrokontroler odczytuje wartości od 0 do 2. Na szczęście, różnica między wartościami, jak wynikało z obliczeń, jest na tyle duża, żeby z łatwością odróżnić wartości jednego przełącznika od reszty.

Jednym ze źródeł problemu jest fakt, że mikrokontroler, który został użyty do wykonania projektu, posiada wartość referencyjną (ta sama wartość, co jest na pinie 5V) równą 5.15V, zamiast teoretycznych 5V. Dodatkowo można uznać, że kolejnym powodem mniejszych wartości są spadki napięcia, spowodowane działaniem reszty układu. Jest do dobry przykład, gdzie w układach elektrycznych, teoria może się różnić od praktyki, z racji na specyfikację urządzeń, czy też sytuacje, których się nie przewidzi.

31:8302371552

Rozdział 4

Podsumowanie

W pierwszym rozdziale omówiliśmy podstawowe informacje na temat maszyny URM, podstawowe komendy, z których składa się program oraz przedstawiliśmy przykłady programów URM-obliczalnych. Następnie zostały przedstawione schematy układów autorskiej koncepcji — klawiatura służąca do przekazania danych do mikrokontrolera, układ składający się z dekoderów oraz wyświetlaczy do przedstawienia danych w programie na Arduino, oraz przełączniki tworzące dzielnik napięcia do sczytywania odpowiedniej wartości analogowej, w celu wybrania odpowiedniego trybu działania programu. Cały działający model maszyny URM wraz z oprogramowaniem został samodzielnie wykonany w ramach niniejszej pracy.

Projekt ma predyspozycje do dalszego rozwoju. Budowa maszyny URM, składająca się z rejestrów przypomina odnoszenie się do pamięci RAM. Podstawowe polecenia odnoszą się do odpowiednich indeksów rejestrów, więc wybranie odpowiedniej komórki z pamięci RAM nie powinno być problematyczne. Idąc tym tokiem myślenia, mikrokontroler jest zbyteczny, a wszystkie działania maszyny URM można uzyskać za pomocą dedykowanego układu.

33:3711995003

Spis tabel

1.1 Podstawowe instrukcje maszyny URM. 7
1.2 Układ początkowy rejestrów maszyny $M(0, 3, 5)$. 8
1.3 Początkowa konfiguracja rejestrów w programie wykonującym dodawanie. 8
1.4 Program wykonujący dodawanie. 9
1.5 Układ rejestrów maszyny obliczającej $\lfloor \frac{x}{4} \rfloor$. 9
1.6 Program obliczający $\lfloor \frac{x}{4} \rfloor$. 9

35:3959350780

Spis rysunków

2.1 Schemat przedstawiający podstawową koncepcję klawiatury. 12
2.2 Zdjęcie przedstawiające układ z rysunku 2.1. 12
2.3 Schemat poprawnie zaprojektowanej klawiatury. 13
2.4 Zdjęcie przedstawiające układ opisany na rysunku 2.3. 13
2.5 Schemat przedstawiający poprawne podłączenie wyświetlacza 8-segmentowego. 14
2.6 Schemat przedstawiający użycie układu CD4511B. 15
2.7 Pełny schemat przedstawiający sterowanie 4-roma wyświetlaczami. . . . 16
2.8 Zdjęcie układu z wyświetlaczami. 18
2.9 Układ przedstawiający układ dzielnika napięcia z przełącznikami 19

3.1 Pola prywatne głównej klasy `URM`. 21
3.2 Następnik (Succesor) — zwiększa wartość w danej komórce. 22
3.3 Wyzerowanie (Zero) — zeruje wartość w danej komórce. 22
3.4 Transfer — kopiuje wartość z jednej komórki do drógiej. 22
3.5 Skok (Jump) — jeżeli wartości z 2 komórek są takie same, przechodzi do odpowiedniego polecenia. 23
3.6 Struktura przechowująca informacje o poleceniu. 23
3.7 Struktura pomocnicza do zapisywania programów. 24
3.8 Przykładowe przygotowanie obiektu klasy URM do wykonania obliczeń. Na przykładzie widać program wykonujący działanie 2+5, na podstawie 1.4. 24
3.9 Główna metoda wykonująca polecenia na rejestrach. 25
3.10 Rys. 3.9, ciąg dalszy. 26
3.11 Ustawienie programu maszyny URM do wykonywania mnożenia 10 oraz 5, za pomocą wcześniej zapisanego programu `add` z rysunku 3.8 (w komentarzach, są podpisane indeksy poleceń). 27
3.12 Najprostszy i podstawowy sposób weryfikacji naciśniętego przycisku. . . 28
3.13 Przykład obsługi przycisku z historią 3 poprzednich stanów. 29

37:1843106964

3.14 Część kodu, odpowiadająca za poprawny wybór wyświetlacza i ustalenia wartości do wyświetlenia. 30

38:9172923253

Bibliografia

Artykuły i książki

[1] John C. Shepherdson i H.E. Sturgis. “Computability of Recursive Functions”. W: (1963), s. 217–255.

[2] N.J. Cutland. *Computability: An Introduction to Recursive Function Theory*. 1980.

[3] Krzysztof Borzęcki. *Wstęp do Teorii Obliczalności - Materiały z zajęć*. 2021.

Odnosniki w sieci

[4] Texas Intruments. *Dekoder BCD-to-7-seg CD4511B*. URL: `https://pdf1.alldatasheet.pl/datasheet-pdf/view/953770/TI1/CD4511B.html`.

[5] Texas Intruments. *Inderter CD4049*. URL: `https://pdf1.alldatasheet.pl/datasheet-pdf/view/26882/TI/CD4049.html`.

[6] ON Semiconductor. *Dekoder BCD-To-Decimal MC14028*. URL: `https://pdf1.alldatasheet.pl/datasheet-pdf/view/3524/MOTOROLA/MC14028B.html`.

39:1007506739