Cyfrowa archeologia - mikrokontroler 80C42

Prolog (gadu-gadu)

PG (7-05-2008 14:58)
"(...) to bardzo interesujący przykład. Dla starszych Czytelników prawdopodobnie kwestia sentymentalna poruszająca serce, a dla najmłodszych zapewne muzeum. Wiec nie ma odpowiedzi jednoznacznej. Z punktu widzenia EdW to interesująca ciekawostka - pokazanie jak to się robiło dawniej, jakie były możliwości i problemy."

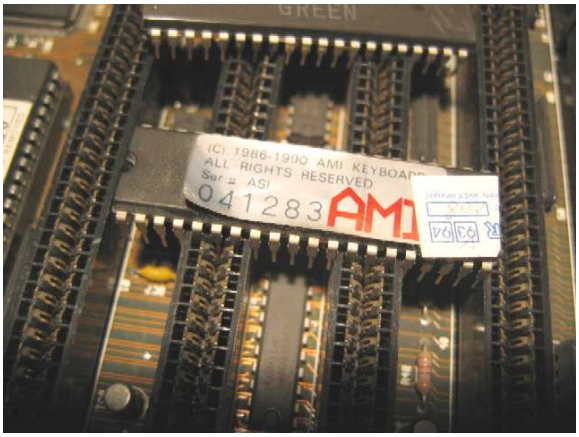
Inspiracja (Forum Elportalu)

Forum Elportal.pl stało się dla mnie ostatnio doskonałym źródłem inspiracji do własnych poszukiwań. Często pojawiają się dyskusje, udział w których owocuje wieloma pomysłami
i w jakiś sposób zmusza do aktywności w dziedzinie elektroniki.
Poniższy tekst inspirowany był wątkiem z Forum założonym przez Adama Kulpińskiego (*Kulpina*) o tytule "*Recycling, czyli wykorzystanie elementów z odzysku...* " **[1]**.
Pierwotnie sprawa dotyczyła pamięci EPROM, potem temat zszedł na stare mikrokontrolery i ogólnie na wykorzystanie zabytkowych, przestarzałych już komputerów.
I właśnie te stare mikrokontrolery szczególnie nie dawały mi spokoju. Zabytkowe płyty główne PC, które aktualnie coraz częściej lądują na złomowisku, są doskonałym źródłem różnego rodzaju elementów - począwszy od drobnicy typu złącza, dławiki, kwarce itp., a skończywszy na pamięciach czy innych specjalizowanych układach.
Niektóre z nich są tak mocno związane z budową płyty, że ich wykorzystanie poza dedykowanym układem jest praktycznie niemożliwe. Inne, pomimo że w elektronice płyty
mają swoją ściśle określoną rolę, dają się "oswoić" na płytce stykowej i dostarczyć wielu emocji na zasadzie: zadziała mi czy nie? Przykładem takich właśnie układów są kontrolery
interfejsu klawiatury. W starych płytach głównych (XT/AT/386) są to układy scalone, które można znaleźć w okolicach klawiaturowego złącza DIN. Jeżeli delikatnie pozbędziemy się okrywającej kostkę nalepki, jest bardzo duże prawdopodobieństwo, że okaże się ona się mikrokontrolerem jednoukładowym 8041 lub 8042 firmy Intel lub innej, która miała licencję na produkcję tego typu procesora (w tym tekście występuje układ firmy NEC). Mikrokontroler ma już zaszyte firmowe oprogramowanie (właśnie do obsługi klawiatury) i zaraz po wyjęciu z płyty głównej jest średnio użyteczny. Ale przecież możemy zmusić ten układ, aby wykonywał nasze, a nie wbudowane oprogramowanie. Kwestia tylko - jak to zrobić i ile to będzie wymagało wysiłku.
Oczywistym jest, że aby taka zabawa zakończyła się sukcesem musimy mieć dostęp do dokumentacji kontrolera i narzędzi programistycznych, czyli jakiegokolwiek assemblera
akceptującego mnemoniki rodziny 8041/42. Z narzędziami problemu wielkiego nie miałam,
ponieważ już dawno temu zlokalizowałam w Sieci bardzo sprytny translator assemblera SB-Assembler **[2]**. Rozumie on mnemoniki bardzo wielu procesorów i kontrolerów, także 8041/42. Z dokumentacją był znacznie większy problem.
Znalazłam przy pomocy Google bardzo interesującą stronkę: "*Devster Specialties*", na której pokazano przykłady układów z wykorzystaniem kontrolera 8042.
Dodatkowo dowiedziałam się, że dalszych wiadomości o tych kostkach należy szukać w Google także pod hasłami UPI-41 oraz UPI-42. UPI to od angielskiej nazwy ***U****niversal* ***P****eripheral* ***I****nterface* - ponieważ tak był nazywany i taką rolę pełnił w płytach PC.
Więc teraz, krok po kroku - przedstawię jak próbowałam nawiązać choćby najcieńszą

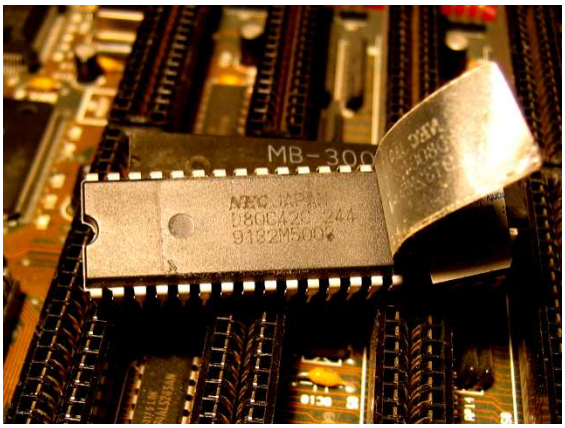
nić porozumienia ze świeżo wyciągniętym ze starej płyty '386 kontrolerem 80C42.

Nalepka AMI (a pod nią...)

Kontroler, jak wspominałam wcześniej, siedział sobie cicho pośród innych elementów płyty głównej, zakryty dla niepoznaki firmową naklejką American Megatrends Inc. (**fotografia 1**). Nalepki mają to do siebie, że zawsze ciekawi co jest pod spodem. Po oderwaniu owej zobaczyłam coś, co wreszcie wyglądało znajomo: układ 80C42 firmy NEC (**fotografia 2**).



fot.1



fot.2

Niestety, dwie pozostałe płyty, które też pozwoliłam sobie "zbadać" na podobnej zasadzie, miały jako UPI nie mikrokontrolery z rodziny 8041/42 ale specjalizowane kostki. No trudno, dobrze że jest choć jeden do zabaw i testów, a zaraz okaże się co on jest wart.

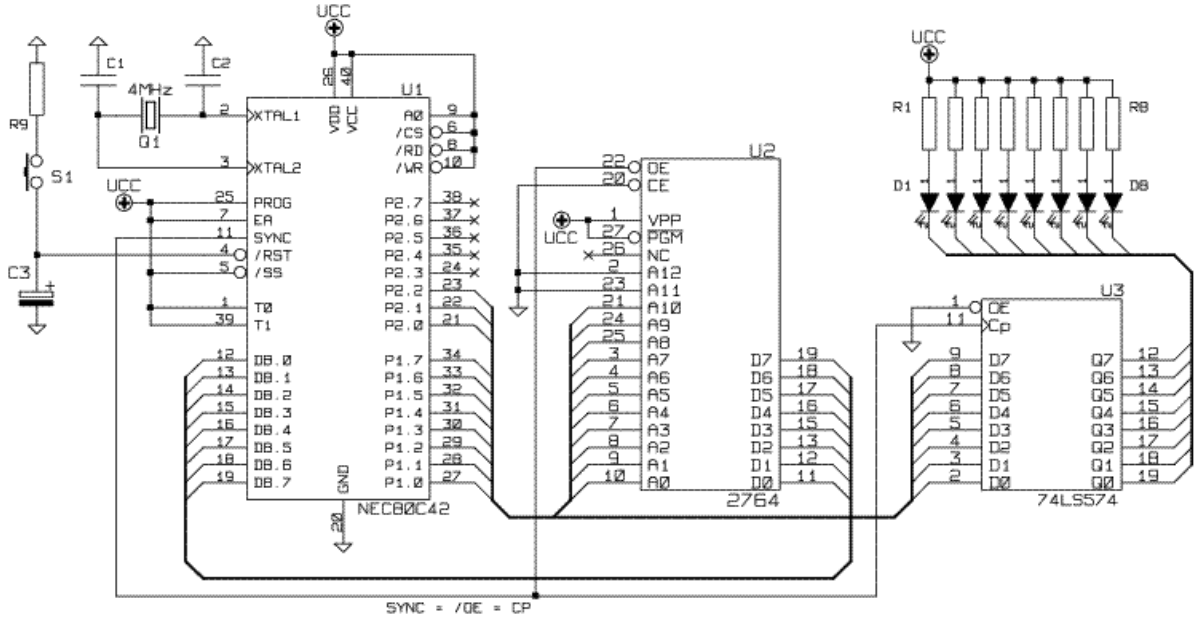
Klonowanie (przecież ktoś już to zrobił)

Na stronce "*Devster Specialties*" znajduje się podstrona *"8042 and 8041 Microcontrollers"* **[3]**, która okazuje się cenna z kilku powodów.
Po pierwsze, jest tam link do dokumentacji kostki, po drugie trzy przykładowe programy i co ważne - zdjęcia działających, próbnych układów. To niejako dowód rzeczowy, że wykorzystanie odzyskanych z płyt PC kontrolerów jest możliwe i że dalsze prace nie będą stratą czasu. Na początek postanowiłam zbudować prosty układzik do sterowania diodami LED, identyczny z zaprezentowanym w punkcie 6 wspomnianej strony. Kod źródłowy udostępnionego programiku testowego doprowadziłam do postaci akceptowalnej przez SB-Assembler (**listing 1**) i po kompilacji powstał maleńki, gotowy do wykorzystania plik **leds1.bin**. Pierwotnie planowałam rozpocząć pracę z emulatorem EPROM, jest to po prostu wygodniejsze od ciągłego przeprogramowywania pamięci, ale ten pomysł zarzuciłam. Powstało ryzyko, że emulator "nie dogada się" z testowym układem i całość nie zadziała. Testowy programik został więc zapisany do pamięci EEPROM typu 28C64, a ta została podłączona do reszty układu. No i pierwszy sukces, który nie ukrywam dodał mi skrzydeł - całość zadziałała! Pora późna już była, więc tego dnia został tylko nagrany pierwszy, mały filmik oraz powstało kilka zdjęć - ten materiał można obejrzeć w serwisie YouTube **[4-a]**. Skoro układ zadziałał tak jak oczekiwałam, podłączyłam do kontrolera emulator pamięci EPROM i załadowałam do jego pamięci rzeczony plik leds1.bin. No i ponownie sukces - układ uparcie pracował dalej.

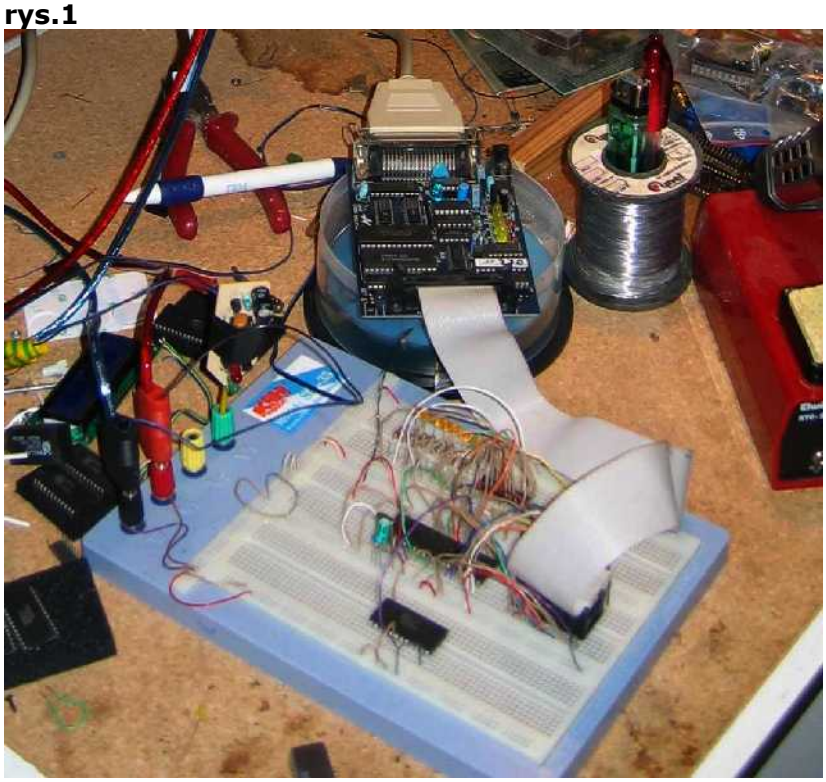
Aby wyjaśnić dlaczego to sukces, pozwolę sobie na małą dygresję.
Jak wiadomo, korzystanie z emulatora pamięci EPROM w przypadku układów mikroprocesorowych z zewnętrzną pamięcią programu niesamowicie poprawia komfort pracy. Jedno polecenie z konsoli tekstowej Windows i już mamy nowy wsad w pamięci emulatora, a przy okazji automatycznie zresetowany mikrokontroler. Z drugiej strony, zawsze istnieje ryzyko, że emulator akurat w tym szczególnym układzie nie będzie

w stanie poprawnie emulować pamięci stałej, stąd też ten etap pośredni z prawdziwą kością pamięci EEPROM. Zwyczajnie, chciałam sobie oszczędzić ewentualnego zniechęcenia. I to podejście także polecam - jeżeli powielamy jakiś układ, co do którego mamy wątpliwości czy nie bazuje na jakichś trikach z sygnałami sterującymi pamięcią stałą (/OE, /CS, itp.) - kopię lepiej uruchomić używając rzeczywistej kostki EPROM lub EEPROM. Dopiero jak układ testowy poprawnie zadziała, spróbować na emulatorze.

Wracamy do naszego kontrolera. Schemat ideowy układu (niemal identyczny z tym na stronie **[3]**) przedstawia **rysunek 1**, a układ zrealizowany na płytce stykowej (zamiast pamięci EEPROM podłączony jest już emulator) - **fotografia 3**.



rys.1

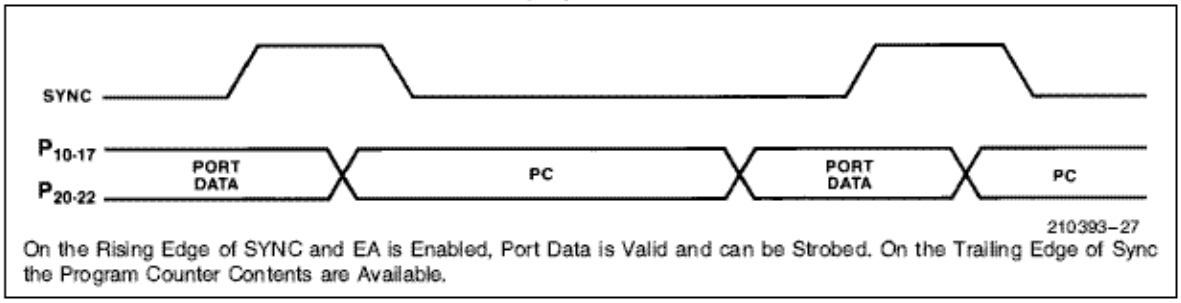


fot.3

I oczywiście kilka zdań wyjaśnienia co do czego służy.
U1 - to jak łatwo zgadnąć kontroler NEC 80C42 pracujący w architekturze z zewnętrzną pamięcią programu. U2 - pamięć stała EPROM/EEPROM - tu układ DE28C64 firmy SEEQ, kostka o pojemności 8kB. Oczywiście, jako U2 możemy zastosować pamięć EPROM typu 2764, tyle tylko, że jakakolwiek modyfikacja programu będzie wymagała kasowania kostki promieniami UV. U3 - to ośmiobitowy rejestr typu D (74LS574), który w chwilach gdy nie jest adresowana zewnętrzna pamięć stała pozwala na przepisanie zawartości portu P1 na własne wyjścia. Do nich (w celu wizualizacji) podłączone jest przez rezystory R1...R8 osiem diodek świecących (D1...D8). Procesor taktowany jest generatorem, którego podstawę stanowi zewnętrzny kwarc Q1 o częstotliwości 4MHz (akurat taki był pod ręką). Aby procesor pobierał kody rozkazów z zewnętrznej pamięci wyprowadzenie EA musi znajdować się w wysokim stanie logicznym.
Elementy R9, C3, S1 to układ zerowania procesora (elementy w sumie nadmiarowe, ponieważ zerowanie zapewnia także emulator w chwili ładowania danych).

Popatrzmy teraz na **rysunek 2** (cytat z dokumentacji procesora), to przebiegi czasowe w chwilach gdy następuje dostęp do zewnętrznej pamięci programu.

PORT TIMING DURING EXTERNAL ACCESS (EA)



On the Rising Edge of SYNC and EA is Enabled, Port Data is Valid and can be Strobed. On the Trailing Edge of Sync the Program Counter Contents are Available.

rys. 2

Cała sztuka w tym układzie polega na tym, że narastające zbocze generowanego przez kontroler sygnał SYNC może posłużyć jako sygnał strobujący zewnętrzny rejestr (lub inny układ peryferyjny). Narastające zbocze SYNC to sygnał, że na porcie dostępne są stabilne dane wystawione instrukcją **outl Pp,A**. Opadające zbocze SYNC oznacza, że na porcie P1 oraz na młodszych bitach portu P2 wystawiana jest zawartość rejestru PC (Program Counter), czyli licznika rozkazów. I właśnie tak udostępniona liczba binarna służy do adresowania zewnętrznej pamięci programu, a SYNC pełni rolę sygnału zezwalającego pamięci stałej na wystawienie kolejnej wartości na swoją szynę danych. Stąd bezpośrednie połączenie SYNCH z wyprowadzeniem /OE (Output Enable) pamięci EEPROM.

A teraz programik testowy **leds1.asm**, którego kod źródłowy przedstawia **listing 1**:

```
    ; wskazujemy assemblerowi z jakim procesorem będzie miał do czynienia
    .CR     8041
    ; plik wynikowy zostanie zapisany jako leds1.bin
    .TF   leds1.bin,BIN
    ; listing programu wyląduje w pliku leds1.lst
    .LF   leds1.lst
    ; globalnie blokujemy przerwania
    dis    i
    ; profilaktycznie blokujemy też przerwania od wewnętrznego timer-a
    dis    tcnti
loop:
    ; do akumulatora wstawiamy binarną liczbę 0111.1111,
    ; co zapali siódmą diodkę
```

```
    mov    a,#$7F        ; LED 7 ON, REST OFF
    ; zawartość akumulatora przepisujemy do portu P1
    outl    p1,a
    ; wywołujemy procedurę realizująca opóźnienie
    call    delay
    ; to samo jeszcze sześć razy, zmienia się tylko wzorek bitowy na P1
    mov    a,#$BF        ; LED 6 ON, REST OFF
    outl    p1,a
    call    delay
    mov    a,#$DF        ; LED 5 ON, REST OFF
    outl    p1,a
    call    delay
    mov    a,#$EF        ; LED 4 ON, REST OFF
    outl    p1,a
    call    delay
    mov    a,#$F7        ; LED 3 ON, REST OFF
    outl    p1,a
    call    delay
    mov    a,#$FB        ; LED 2 ON, REST OFF
    outl    p1,a
    call    delay
    mov    a,#$FD        ; LED 1 ON, REST OFF
    outl    p1,a
    call    delay
    mov    a,#$FE        ; LED 0 ON, REST OFF
    outl    p1,a
    call    delay
    mov    a,#$FD        ; LED 1 ON, REST OFF
    outl    p1,a
    call    delay
    mov    a,#$FB        ; LED 2 ON, REST OFF
    outl    p1,a
    call    delay
    mov    a,#$F7        ; LED 3 ON, REST OFF
    outl    p1,a
    call    delay
    mov    a,#$EF        ; LED 4 ON, REST OFF
    outl    p1,a
    call    delay
    mov    a,#$DF        ; LED 5 ON, REST OFF
    outl    p1,a
    call    delay
    mov    a,#$BF        ; LED 6 ON, REST OFF
    outl    p1,a
    call    delay
    jmp    loop
    ; poniżej procedura opóźniająca....
delay:
    ; zewnętrzny licznik w rejestrze R1, inicjowany wartością 255
    mov    r1,#255
delay_2:
    ; wewnętrzny licznik - w rejestrze R0, wartość odpowiednio mniejsza
```

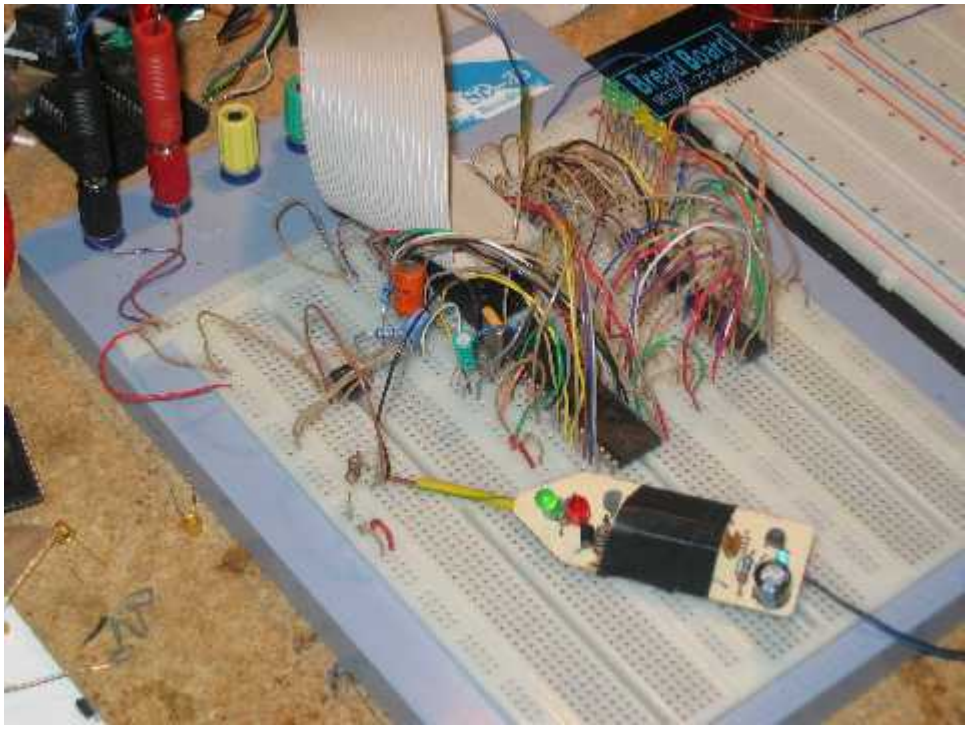
```
    mov    r0,#63
delay_1:
    ; nie rób nic dwa razy
    nop
    nop
    ; zmniejszamy licznik wewnętrzny (R0), aż dojdzie do zera
    djnz    r0,delay_1
    ; zmniejszamy licznik zewnętrzny (R1), aż także dojdzie do zera
    djnz    r1,delay_2
    ; tu licznik R1 jest wyzerowany - powrót z procedury
    ret
```

lst.1

Powyższy programik nie jest zbytnio elegancki, ale za to jest bardzo klarowny i od razu widać co program będzie robił. I chwała dla Autora strony *"Devster Specialties"*, że go napisał i udostępnił w sieci...

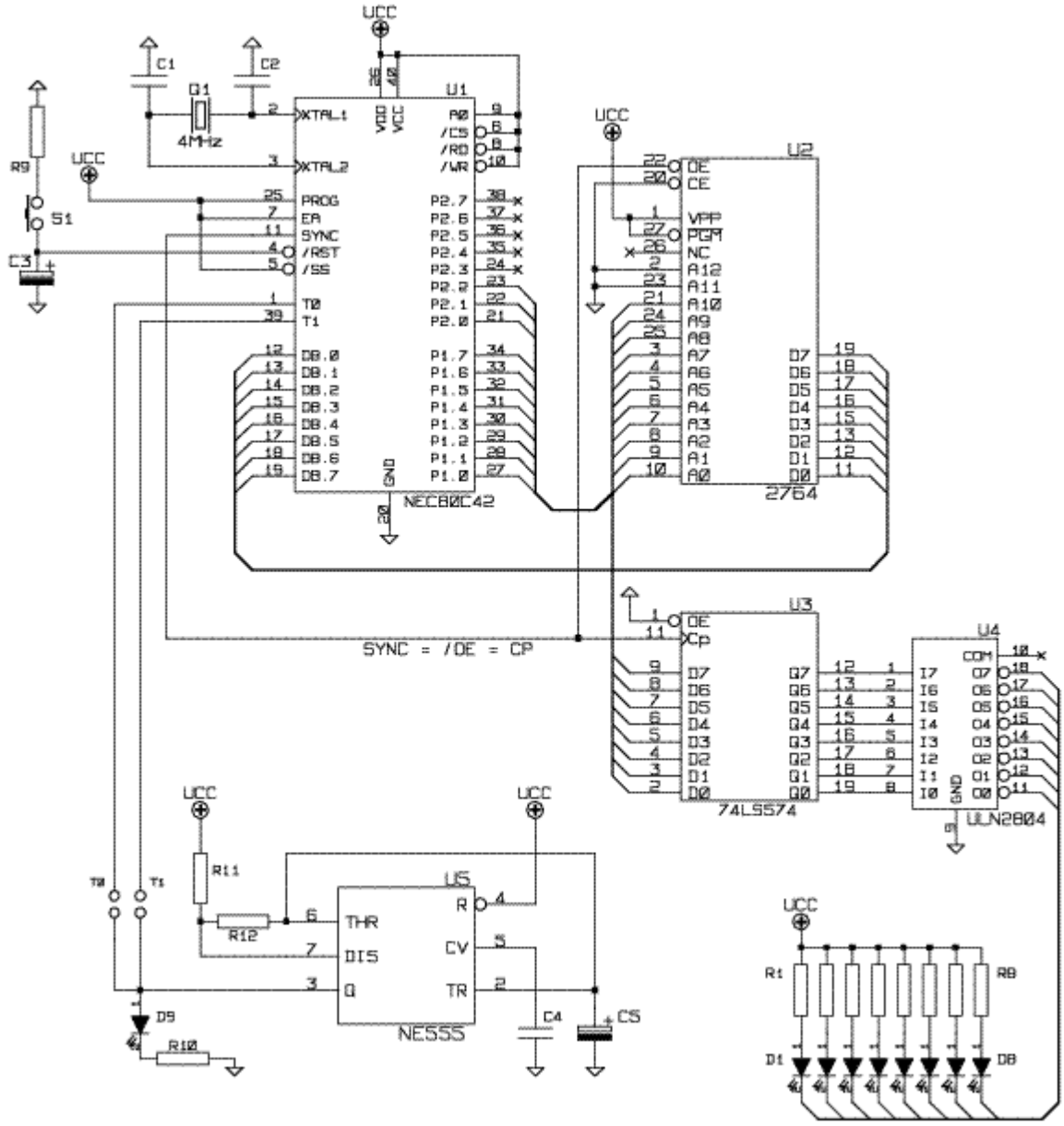
Inwencja twórcza (własne eksperymenty)

Pierwszy krok, to mała reorganizacja układu na płytce stykowej - **fotografia 4**.



fot.4

Pierwszy układ, który zbudowałam, zajmował zbyt wiele miejsca, a jak wiadomo - apetyt rośnie w miarę jedzenia i potrzebowałam więcej przestrzeni na jego dalszą rozbudowę. Przy okazji przebudowy zamieniłam sposób sterowania LED oraz dodałam elementy pomocne przy dalszych eksperymentach z kontrolerem.
Nowy schemat przedstawia **rysunek 3**.



rys.3

Listing 2 czyli program **leds2.asm** to już moja własna wizja migadełka LED, tym razem wzorki do wyświetlenia znajdują się w bloku stałych, na końcu programu.
Dane do wyświetlenia pobierane są sekwencyjnie i może być ich dość sporo, maksymalnie
do adresu 0FFh. Tyle właśnie wynosi rozmiar jednej strony pamięci stałej i taka jest pojemność rejestru adresującego tę pamięć. Jak widać, dane z portu P1 podawane są w logice dodatniej, czyli LED świeci gdy na danym bicie portu jest wysoki stan logiczny. Zabieg ten był niezbędny, ponieważ tym razem diodki podłączone są do rejestru D (U3) nie bezpośrednio, ale przez bufor ULN2804 (U4). Oczywiście, zamiast diod LED możemy teraz podłączyć osiem małych żaróweczek (ale tak, aby nie przeciążyć układu ULN2804).

```
    .CR     8041
    .TF   leds2.bin,BIN
    .LF   leds2.lst
start:
    ; do A adres tabelki i zabezpiecz w R2
    mov    A,#pattern_begin
```

```
    mov    R2,A
    ; do R3 długość wzorca
    mov    R3,#pattern_end-pattern_begin
loop:
    mov    A,R2        ; pobierz adres elementu
    movp   A,@A        ; pobierz element w/g adresu
    outl   P1,A        ; wystaw na port
    inc    R2          ; R2++ czyli nowy element
    djnz   R3,loop     ; powtarzaj dopóki R3 != 0
    jmp    start       ; zainicjuj zmienne ponownie
    ;
delay:
    mov    R1,#255
delay_2:
    mov    R0,#63
delay_1:
    nop
    nop
    djnz   R0,delay_1
    djnz   R1,delay_2
    ret
    ;
    ; dane do wzorka dla LED, maksymalnie do adresu 0FFh
pattern_begin:
    .db %0000.0001
    .db %0000.0010
    .db %0000.0100
    .db %0000.1000
    .db %0001.0000
    .db %0010.0000
    .db %0100.0000
    .db %1000.0000
    .db %0100.0000
    .db %0010.0000
    .db %0001.0000
    .db %0000.1000
    .db %0000.0100
    .db %0000.0010
    .db %0000.0001
    .db %0000.0011
    .db %0000.0111
    .db %0000.1111
    .db %0001.1111
    .db %0011.1111
    .db %0111.1111
    .db %1111.1111
    .db %1111.1110
    .db %1111.1100
    .db %1111.1000
    .db %1111.0000
    .db %1110.0000
    .db %1100.0000
```

```
    .db %1000.0000
    .db %1111.0000
    .db %0000.1111
    .db %1111.0000
    .db %0000.1111
pattern_end:
```

Film pokazujący działanie programu leds2.asm znajduje się w **[4-b]**.

Wspomniane wcześniej dodatkowe elementy na schemacie - to generator przebiegu prostokątnego zbudowany na bazie dobrze znanego układu NE555 (U5). Elementy generatora (C5,R11,R12) zostały dobrane tak, aby częstotliwość wyjściowa była bardzo mała i wynosiła około 0.5...1Hz. Do sygnalizacji wysokiego stanu logicznego na wyjściu generatora służy niebieska dioda świecąca - na schemacie D9. Do czego posłuży ten generator? Już wyjaśniam, a zacznę od cytatu z dokumentacji UPI-42 **[9]** (**rysunek 4**).

Symbol	DIP Pin No.	PLCC Pin No.	QFP Pin No.	Type	Name and Function
TEST 0, TEST 1	1 39	2 43	18 16	I	**TEST INPUTS:** Input pins which can be directly tested using conditional branch instructions. **FREQUENCY REFERENCE:** TEST 1 (T_1) functions as the event timer input (under software control). TEST 0 (T_0) is a multi-function pin used during PROM programming and ROM/EPROM verification, during Sync Mode to reset the instruction state to S1 and synchronize the internal clock to PH1.

Mnemonic	Description	Bytes	Cycles
BRANCH			
JT0 addr	Jump on T0 = 1	2	2
JNT0 addr	Jump on T0 = 0	2	2
JT1 addr	Jump on T1 = 1	2	2
JNT1 addr	Jump on T1 = 0	2	2

rys.4

Z cytowanych tabelek wynika jasno, że stan wejść T0 i T1 może być sprawdzony programowo przy pomocy specjalnych rozkazów procesora. Dodatkowo, podany na wejście T1 sygnał prostokątny może posłużyć do taktowania wbudowanego w procesor ośmiobitowego licznika. I właśnie dlatego dodałam do układu mały generatorek,
aby mieć źródło sygnału testowego, a jego niewielka częstotliwość ułatwia obserwacje i nagrywanie dokumentalnego filmu.
Najpierw zajmiemy się sterowanymi zewnętrznie skokami warunkowymi.
Instrukcja **JT0** to polecenie skoku, jeżeli w chwili jej wykonywania na wejściu T0 będzie wysoki stan logiczny. Analogicznie **JNT0** - skok zostanie wykonany, gdy wejście będzie w niskim stanie logicznym. Dla wejścia T1 jest identycznie. Dalej, dwa banalne programiki
w formie ilustracji jak obsługiwać wejścia T0 oraz T1.
Oczekiwanie działanie programu **test0.asm** (**listing 3**) to zapalanie się diodek D1...D8 wraz z niebieską diodą D9 sygnalizującą wysoki stan logiczny na wejściu T0.
Program **test1.asm** (**listing 4**) - diody będą świecić się, gdy niebieski LED jest zgaszony,
czyli gdy wejście T1 jest w niskim stanie logicznym.

```
    .CR     8041
    .TF   test0.bin,BIN
    .LF   test0.lst
    ;
```

```
    ; początkowo zgaś wszystkie ledy
    mov    A,#00h
loop:
    outl    P1,A
    jt0    set_leds_on   ; skok gdy T0 == H
    mov    A,#00h        ; zgaś
    jmp    loop 
    ;
set_leds_on:
    mov    A,#0FFh       ; zapal
    jmp    loop
    ;
    ;
```

lst.3

```
    .CR     8041
    .TF   test1.bin,BIN
    .LF   test1.lst
    ;
    ; początkowo zgaś wszystkie ledy
    mov    A,#00h
loop:
    outl    P1,A
    jnt1    set_leds_on  ; skok gdy T1 == L
    mov    A,#00h        ; zgaś
    jmp    loop
    ;
set_leds_on:
    mov    A,#0FFh        ; zapal
    jmp    loop
    ;
```

lst.4

Działanie obu programów prezentują dwa krótkie filmy **[4-c]** oraz **[4-d]**.

A teraz zabierzemy się za wbudowany w kontroler licznik/timer.
Pojemność licznika to osiem bitów (zakres 00h..0FFh) i może on pracować w dwóch trybach. Jeżeli źródłem impulsów do zliczania jest podany z zewnątrz (na wejście T1) sygnał prostokątny - jest to tryb *counter*, czyli zwyczajny licznik zdarzeń zewnętrznych. Jeżeli układ taktowany jest sygnałem zegarowym procesora (ale podzielonym wewnętrznie przez wartość 15 a następnie przez 32 /o czym dalej/), mówimy o trybie pracy *timer*, w którym może on posłużyć do odmierzania ustalonych wcześniej odcinków czasu. Dobrze jest też pamiętać, że bez względu na sposób taktowania układ zawsze zlicza w górę. Zliczanie następuje począwszy od wartości wpisanej rozkazem **mov T,A** do maksymalnej wartości 0FFh. Potem następuje automatyczne wyzerowanie licznika (0FFh zmienia się na 00h), w tym też momencie może nastąpić zgłoszenie przerwania maskowalnego - skok pod adres **7h**.

Tu chyba powinnam jedną sprawę wyjaśnić. Dokumentacja UPI-42 nie wspomina *explicite* o dzielącym przez 32 preskalerze, oraz o dodatkowym podziale zegara przez 15, dokonywanym przez wewnętrzne układy procesora.

Nie wspomina także, że wektor obsługi przerwania licznikowego jest dokładnie pod adresem 7h. Wspomina tylko, że jest...gdzieś.
Tu właśnie widać uroki wykorzystania takich staroświeckich kostek, do których dokumentacja jest bardzo skąpa, często niekompletna i pozbawiona detali krytycznych dla procesu pisania oprogramowania. Testy z licznikiem oraz przerwaniami w 80C42 wykonałam bazując także na dokumentacji (o niebo lepszej!) kontrolera 8048 **[5,6,7,8]**, traktując sprawę następująco: skoro model '48 niejako wywodzi się z '42 to przynajmniej
część podstawowych bloków procesora powinna działać podobnie.
Oczywiście, trzeba to sprawdzić eksperymentalnie, ale mając na uwadze ryzyko błędnego wnioskowania wynikające z niewiedzy o innych zjawiskach, które nie są udokumentowane, a za naszymi plecami wpływają na pracę kontrolera.

Uruchomienie licznika w trybie *counter* następuje po wykonaniu rozkazu **strt CNT**, bieżącą zawartość licznika możemy w dowolnym momencie odczytać (pobrać do akumulatora) rozkazem **mov A,T**.

Listing 5 czyli program **counter1.asm** to prezentacja działania licznika w trybie *counter*.

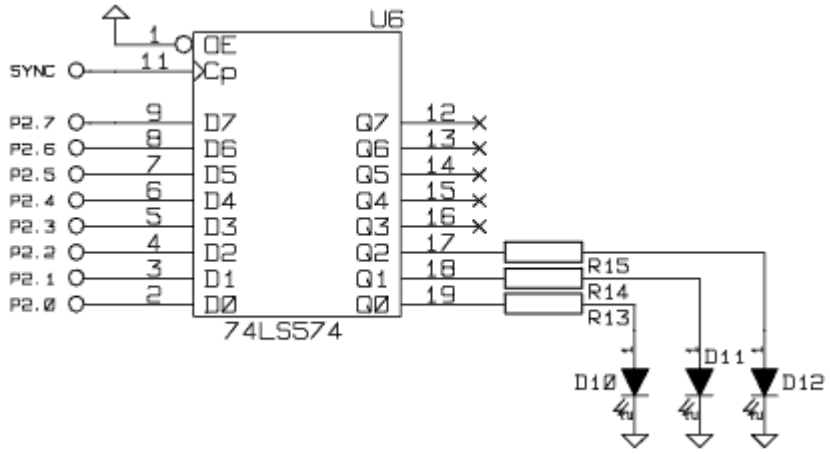
```
    .CR    8041
    .TF    counter1.bin,BIN
    .LF    counter1.lst
    ;
    mov    A,#00h
    mov    T,A    ; wyzeruj licznik
    strt   CNT    ; i wystartuj...
loop:
    mov    A,T    ; oczytaj licznik
    outl   P1,A   ; pokaż na porcie
    jmp    loop   ; i tak w kółko
    ;
```

lst.5

Po uruchomieniu licznika program wykonuje nieskończoną pętlę, w której pracowicie przepisuje pobraną z licznika wartość na port P1. Ponieważ na porcie P1 mamy zapięty rejestr z diodami sygnalizacyjnymi, na filmie **[4-f]** bardzo ładnie widać jak pracuje zewnętrznie taktowany licznik, oczywiście jego aktualną zawartość widzimy w postaci binarnej. Przy okazji zauważamy, że licznik reaguje na opadające zbocze sygnału zegarowego (zmiana stanu T1 z H na L) - liczba binarna prezentowana na diodach D1...D8 zmienia się w momencie gdy dioda D9 gaśnie.

Następna próba (przerwania) będzie wymagała delikatnej modyfikacji układu, po prostu potrzeba dodatkowych diodek sygnalizacyjnych. Przy okazji dowiemy się czy na porcie P2 można zapiąć zewnętrzny rejestr buforujący dane, analogicznie jak miało to miejsce w przypadku portu P1.

Nowego, kompletnego schematu już nie załączam, niech wystarczy tylko ten dodatkowy fragment widoczny na **rysunku 5**.



rys.5

Oczywiście, najpierw mały test czy port P2 może być wykorzystany podobnie jak P1 - to **listing 6** programu **p2latch.asm**.

```
    .CR    8041
    .TF    p2latch.bin,BIN
    .LF    p2latch.lst
    mov    A,#00h
loop:
    outl   P2,A        ; port P2 (!!!)
    jnt1   set_leds_on
    mov    A,#00h
    jmp    loop
set_leds_on:
    mov    A,#0FFh
    jmp    loop
    ;
```

lst.6

Szczerze mówiąc, program nie robi nic mądrego - ot, po prostu miga dodatkowymi diodami D10...D12 (film **[4-e]**). Ale to właśnie pokazuje, że możemy do kontrolera dołączyć jeszcze jeden ośmiobitowy rejestr typu D, zyskując w ten sposób całe szesnaście bitów wyjściowych na własne potrzeby. A to już całkiem sporo.

Wracamy do przerwań. W momencie przepełnienia licznika (*overflow*) czyli przejścia z 0FFh na 00h i gdy odblokowane są przerwania od licznika (rozkazam **en TCNTI**) procesor wykona skok pod sztywno ustalony adres 7h. Pod tym właśnie adresem powinna znajdować się procedura obsługi przerwania (*interrupt handler*) lub do niej bezwarunkowy skok, jeżeli adres 7h z jakichś względów nam sie nie podoba. Procedurę obsługi przerwania kończymy zawsze rozkazem **retr** (***ret****urn and* ***r****estore status*), co poza zwróceniem sterowania do programu głównego odtwarza też rejestr statusu procesora. Jak będzie działał program testowy? Bardzo prosto: aby nie czekać zbyt długo do licznika wpiszemy jakąś wartość bliską 0FFh (powiedzmy, że 0F8h), odblokujemy przerwania od licznika, wystartujemy licznik i...poczekamy chwilkę. Gdy licznik osiągnie maksymalną wartość (będzie to widać na diodach D1...D8), kolejne opadające zbocze na wejściu T1 spowoduje jego przepełnienie i wygenerowanie przerwania. Procedura obsługi przerwania zapali dodatkowe (te na porcie P2) diody świecące a to będzie dowodem, że cały proces zadziałał tak, jak tego oczekiwaliśmy. **Listing 7** to właśnie nasz programik do testowania przerwań - **cntrint.asm**,

a jego działanie można zobaczyć na krótkim filmie **[4-g]**.

```
    .CR     8041
    .TF   cntrint.bin,BIN
    .LF   cntrint.lst
    ;
    .no    0h
    jmp    main
    ;
    .no    7h
    jmp    tc_interrupt_service
    ;
    .no    20h
main:
    ; dodatkowe ledy wyłączone
    mov    A,#00h
    outl   P2,A
    ;
    ; zainicjuj licznik
    mov    A,#0F8h
    mov    T,A
    en     TCNTI
    strt   CNT
    ;
    ; pokazuj stan licznika
loop:
    mov    A,T
    outl   P1,A
    jmp    loop
    ;
tc_interrupt_service:
    mov    A,#0FFh
    outl   P2,A
    retr
    ;
```

lst.7

Jak wspominałam, w trybie *timer* licznik taktowany jest sygnałem, powstałym z podziału wewnętrznego sygnału zegarowego przez wartość 32. Sygnał ten pobierany jest ze specjalnego bloku procesora odpowiedzialnego za synchronizację pracy wszystkich jego elementów, blok ten wprowadza dodatkowy podział przez 15 względem częstotliwości zewnętrznego sygnału zegarowego (oscylatora kwarcowego).
Dlaczego podział akurat przez 15? Po szczegóły (lojalnie uprzedzam - trudne) odsyłam do pozycji "*Grokking the MCS-48 System*" **[5]**, w której znajduje się opis liczników cykli i stanów maszynowych rdzenia procesora (*cycle counter*, *state counter*).
A my przyjmiemy aksjomatycznie, że do taktowania timera służy sygnał o częstotliwości 480 (32*15) razy mniejszej niż ma dołączony zewnętrznie kwarc.
I to nam z powodzeniem wystarczy do dalszych rozważań.

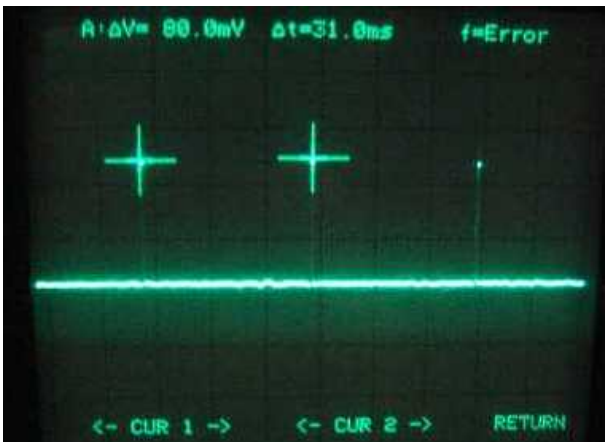
Dość banalny przykład wykorzystania timera do generowania impulsów prostokątnych to kolejny programik o nazwie **timerint.asm** czyli **listing 8**.

```
    .CR     8041
```

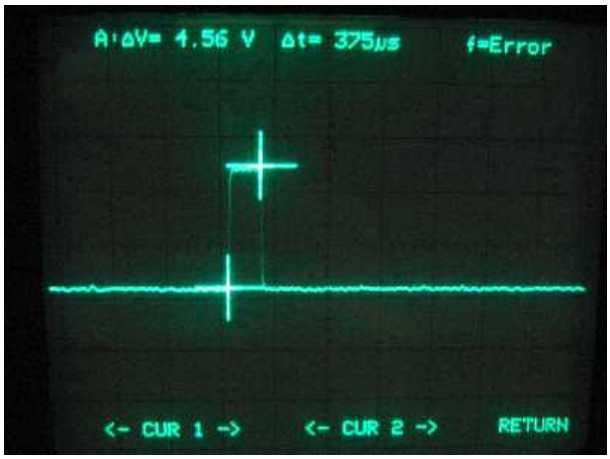
```
    .TF   timerint.bin,BIN
    .LF   timerint.lst
    ;
    .no    0h
    jmp    main
    ;
    .no    7h
    jmp    tc_interrupt_service
    ;
    .no    20h
main:
    ;
    ; dodatkowe ledy wyłączone
    mov    A,#00h
    outl   P2,A
    ;
    ; zezwolenie na start timera i przerwania
    en     TCNTI
    strt   T
    ;
loop:
    jmp    loop
    ;
tc_interrupt_service:
    mov    A,#0FFh
    outl   P2,A
    mov    R7,#32
tc_imp:
    nop
    djnz   R7,tc_imp
    clr    A
    outl   P2,A
    retr
    ;
```

lst.8

Tym razem zamiast filmu - **fotografia 5**, na której widzimy krótkie impulsy (szpilki) generowane przez program timerint.asm co około 31 milisekund. Zastanówmy się chwilkę,
dlaczego tyle? Wiemy, że licznik ma pojemność 0FFh, czyli przepełni się po 256 impulsach wejściowych, generując przerwanie maskowalne. Wiemy też, że impulsy owe są podawane z preskalera (podział przez 32), zasilanego sygnałem o piętnaście razy mniejszej częstotliwości niż ma zewnętrzny kwarc. Czyli całość wprowadzi podział przez 122880 (32*15*256) względem częstotliwości kwarca Q1.
Skoro ta wynosi 4MHz (okres 250ns), to proste mnożenie: 250ns * 122880 da nam wartość 0.03072 sekundy czyli liczbę pokazaną przez oscyloskop. Drobna niedokładność wynika z tego, że kursory oscyloskopu ustawiałam ręcznie, dla tak krótkich impulsów automatycznie rozmieszczenie kursorów dawało dość niecodzienne efekty...
Teraz weźmy pod lupę (dosłownie!) jedną z widocznych na **fotografii 5** szpilek. Rozciągnięty na osi czasu impuls (funkcją *zoom*) i z automatycznie rozmieszczonymi kursorami pomiarowymi przedstawia **fotografia 6**. Jak widać, impuls trwa 375us, a my ponownie spróbujemy określić skąd akurat taka wartość.



fot. 5



fot.6

Przeanalizujmy teraz fragment procedury obsługi przerwania odpowiedzialny za generowanie wspomnianego impulsu, ale pod kątem czasu wykonania kolejnych instrukcji.

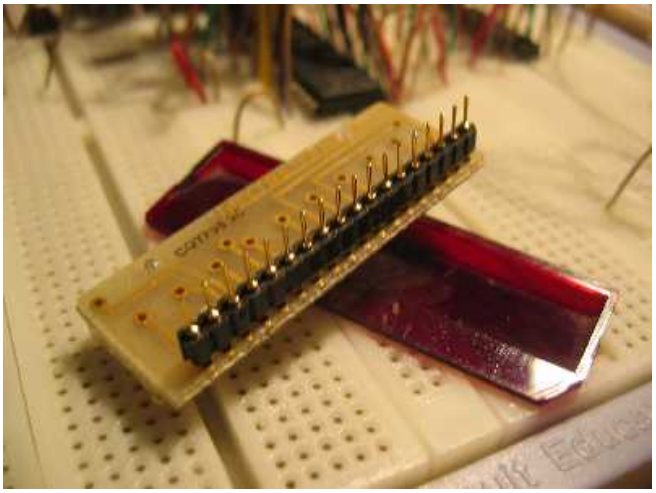
```
tc_interrupt_service:
    mov     A,#0FFh     ; tego nie liczymy
    outl    P2,A        ; tego też nie!
    mov     R7,#32      ; 2 cykle czyli 2*(15*250ns) daje 7.5us
tc_imp:
    nop                 ; 1 cykl czyli 15*250ns (3.75us)
    djnz    R7,tc_imp   ; 2 cykle czyli 2*(15*250ns) daje 7.5us
    ; powyższe dwie instrukcje wykonają się 32 razy
    ; trwając sumarycznie 360us
    clr     A           ; 1 cykl czyli 15*250ns (3.75us)
    outl    P2,A        ; 2 cykle czyli 2*(15*250ns) daje 7.5us
    retr
```

Wnętrze petli (nop + djnz) wykonane 32 razy trwa 360us, dodając czas wykonania rozkazów przed i po otrzymamy wartość około 379us. Ponownie widać różnicę pomiędzy wskazaniem przyrządu a obliczeniami choć jej przyczyna jest teraz nieco innej natury. Oczywiście - oscyloskop ustawił kursory najlepiej jak umiał, ale to nie znaczy, że bezbłędnie. Po drugie, należałoby teraz zastanowić się nad samym rozkazem **outl**, generującym fizycznie dostępną informację (zmianę stanu logicznego na wyprowadzeniu procesora). Rozkaz ma dwa bajty (kod rozkazu i jego parametr) i trwa dwa cykle. Powstaje pytanie - w którym momencie od chwili rozpoczęcia jego przetwarzania zostanie zmieniony stan wyjścia. Rozkazy takie w procedurce generującej impuls mamy dwa (zmiana z L na H, potem z H na L) ale myślę, że można w obliczeniach uwzględnić tylko ten drugi (generujący stan L). Dlaczego? Pobranie z pamięci i zdekodowanie rozkazu zajmuje więcej czasu niż wykonanie zmiany na porcie w ostatnim cyklu maszynowym, a nas tak naprawdę interesuje czas od chwili wystawienia stanu H. Drugi rozkaz outl musimy uwzględnić w obliczeniach, ponieważ w czasie jego przetwarzania na wyjściu portu ciągle mamy jedynkę logiczną i trwa pomiar czasu. Może ktoś teraz się zadziwił - po co w ogóle wnikać w takie szczegóły, w nano i mikrosekundy? Odpowiedź jest prosta - często zdarza się, że piszemy oprogramowanie, które wymaga zachowania bardzo precyzyjnych zależności czasowych w generowanych sygnałach. Niech za przykład posłuży tu programowa obsługa magistrali 1-Wire. I w tym momencie po prostu musimy umieć choć zgrubnie oszacować ile dany fragment kodu będzie się wykonywał. Można oczywiście "stroić" taka procedurę obserwując przebieg na oscyloskopie, ale jakieś wstępne wartości dobrze jest przyjąć bazując właśnie na podobnych do przedstawionych powyżej wyliczankach.

Po tej dawce "zaawansowanej matematyki" - coś dla relaksu.

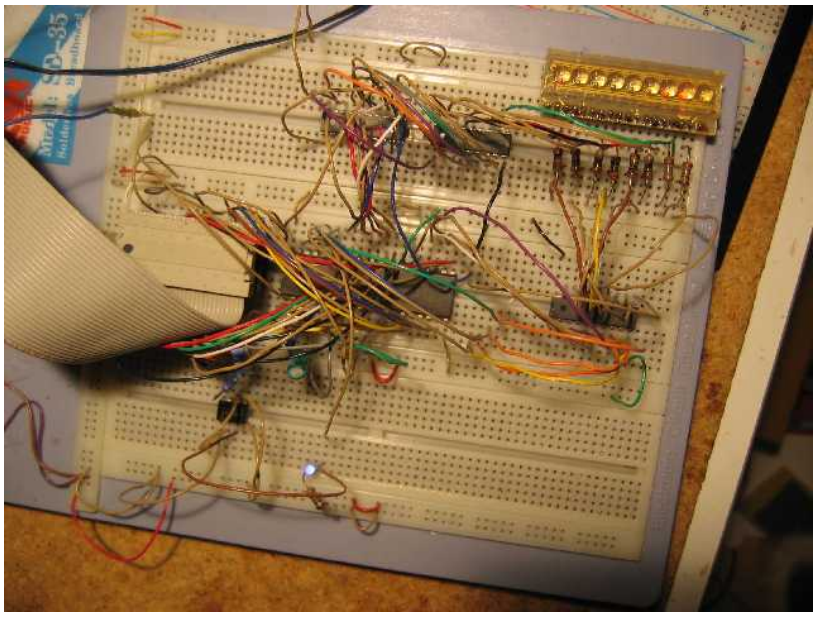
Skoro wiemy jak posługiwać się timerem, umiemy skorzystać z przerwań, które on generuje i mamy dołączony do portu P2 dodatkowy rejestr wyjściowy to może... spróbujmy zastosować to wszystko do multipleksowanego sterowania wyświetlaczem siedmiosegmentowym.

fot. 7

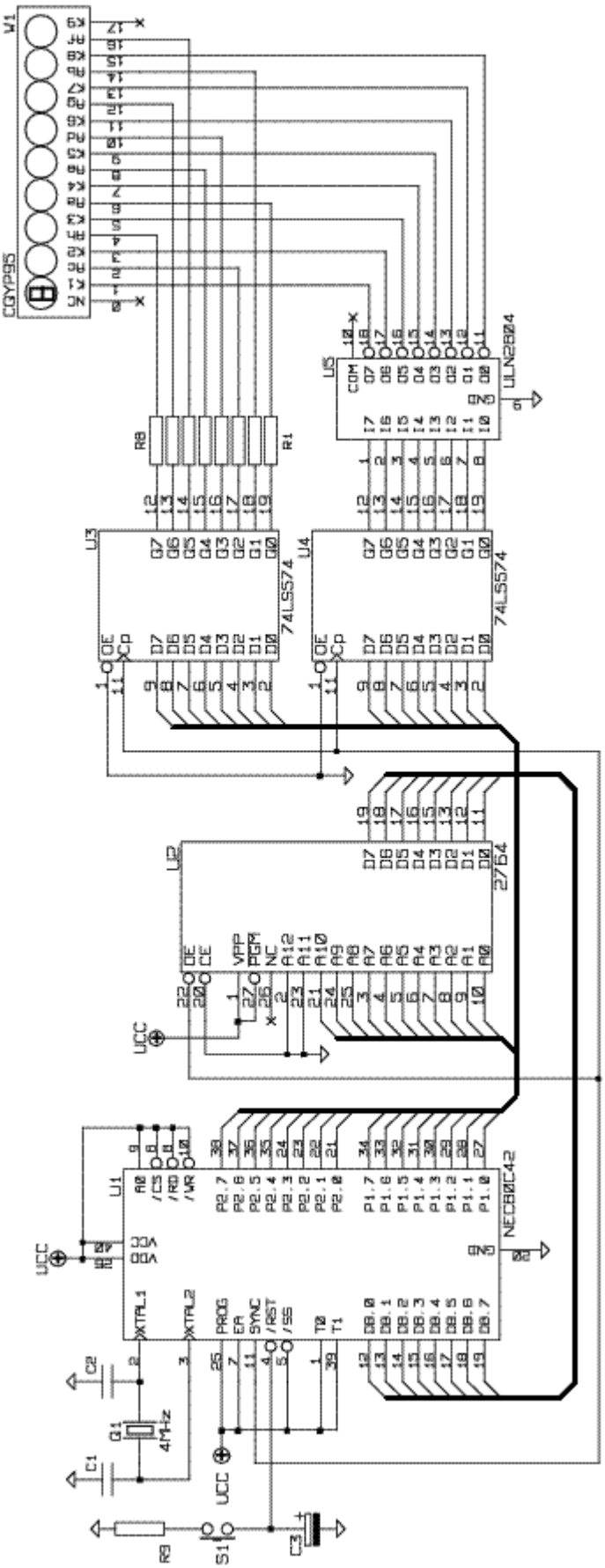


fot. 8

Nowy, nigdy jeszcze nie używany wyświetlacz przedstawia **fotografia 7**, na **fotografii 8** ma już dolutowane złącze igłowe aby ładnie dał się połączyć z płytką stykową. Ten wyświetlacz to polski produkt o nazwie **CQYP95** - dziewięć pól odczytowych, każde po siedem segmentów plus kropka dziesiętna, diody LED w układzie wspólna katoda. Element ten (podobnie jak kontroler 80C42) pochodzi z "poprzedniej epoki" i dlatego chyba dobrze pasuje do tego tekstu.
Układ sterowania wyświetlaczem zrealizowany na płytce stykowej przestawia **fotografia 9**, na **rysunku 6** znajduje się jego schemat ideowy.



fot. 9


W1
COY95
R8
R1
U5
COM
ULN2804
U3
U4
74LS574
74LS574
U2
2764
VPP
PGM
NC
A12
A11
A10
A9
A8
A7
A6
A5
A4
A3
A2
A1
A0
OE
CE
UCC
U1
/CS
/RD
/WR
P2.7
P2.6
P2.5
P2.4
P2.3
P2.2
P2.1
P2.0
P1.7
P1.6
P1.5
P1.4
P1.3
P1.2
P1.1
P1.0
NEC80C42
VDD
XTAL1
XTAL2
PROG
EA
SYNC
/RST
/SS
T0
T1
DB.0
DB.1
DB.2
DB.3
DB.4
DB.5
DB.6
DB.7
C2
Q1
4MHz
C1
R9
S1
C3

rys.6

Program obsługujący wyświetlacz nazywa się **cqyp95.asm** i widzimy go na **listingu 9**.

```
        .CR     8041
        .TF   cqyp95.bin,BIN
        .LF   cqyp95.lst
        ;
display_buffer    .eq    32
        ;
        ;       kgfedcba
H_def   .eq    %01110110
E_def   .eq    %01111001
L_def   .eq    %00111000
O_def   .eq    %00111111
d_def   .eq    %01011110
W_def   .eq    %00111110
T_def   .eq    %00110001
N_def   .eq    %00110111
A_def   .eq    %01110111
S_def   .eq    %01101101
Z_def   .eq    %01011011
        ;
        .no        0h
        jmp    main
        ;
        .no     7h
        jmp    tc_interrupt_service
        ;
        .no    20h
main:
        sel    RB1        ; dobierz sie do drugiego banku R
        mov    R0,#0h     ; wyzeruj aktualna pozycje wyswietlacza
        sel    RB0        ; ustaw z powrotem bank zerowy
        ;
        ; zainicjuj licznik
        en    TCNTI
        strt    T
        ;
loop:
        mov     A,#message1
        call    set_text
        call    delay
        ;
        mov     A,#message2
        call    set_text
        call    delay
        ;
        mov     A,#message3
        call    set_text
        call    delay
        jmp    loop
        ;
        ; R7' - ochrona akumulatora
```

```
        ; R0' - aktualna cyfra
tc_interrupt_service:
        ; pracuj z drugim bankiem rejestrów
        sel    RB1
        ;zabezpiecz aku w R7' skoro nie ma stosu :(
        mov    R7,A
        ; przeładuj licznik (szybkość multipleksu!)
        stop     TCNT
        mov    A,#0FFh-30
        mov    T,A
        strt    T
        ;
        ; numer aktualnej cyfry - P1
        mov    A,R0            ; wez aktualną pozycję
        inc    A            ; pozycja++
        anl    A,#07h             ; modulo 7
        mov    R0,A            ; zapisz zmianę
        add    A,#digits_sel_table    ; digits_sel_table + pozycja
        movp    A,@A            ; aktywne katody w/g pozycji
        outl    P1,A            ; wystaw na port
        ;
        ;dana do wyswietlenia - P2
        mov    A,R0            ; aktualna pozycja
        add    A,#display_buffer    ; display_buffer + pozycja
        mov    R1,A            ; ustaw wskaźniek (R1)
        mov    A,@R1            ; weź daną w/g wskaźnika
        outl    P2,A            ; i na port
        mov    A,R7            ; odtwórz aku.
        retr                ; i to wsyztsko
        ;
        ;
set_text:
        ; we: A - adres komuniaktu, ma mieć 8 znaków
        mov    R0,#display_buffer
        mov    R1,#8
set_text_1:
        mov    R2,A
        movp    A,@A
        mov    @R0,A
        inc     R0
        mov    A,R2
        inc     A
        djnz    R1,set_text_1
        ret
        ;
delay:
        mov    R1,#0FFh
delay_2:
        mov    R0,#0FFh
delay_1:
        nop
        nop
```

```
        nop
        nop
        nop
        nop
        djnz    R0,delay_1
        djnz    R1,delay_2
        ret
        ;
digits_sel_table:
        .db    %0000.0001
        .db    %0000.0010
        .db    %0000.0100
        .db    %0000.1000
        .db    %0001.0000
        .db    %0010.0000
        .db    %0100.0000
        .db    %1000.0000
        ;
message1:
    .db    H_def,E_def,L_def,L_def,O_def,0,0,0
message2:
    .db    0,0,0,E_def,d_def,W_def,0,0
message3:
    .db    N_def,A_def,T_def,A_def,S_def,Z_def,A_def,0
```

lst.9

Szczegółową analizę jak ten programik działa pozostawiam dociekliwym i nie będą nią zajmowała cennego miejsca. Chcę natomiast zwrócić uwagę na kilka "uroków" kontrolera 8042, które zepsuły mi humor podczas pisania programu do obsługi CQYP95.

Po pierwsze - brak dostępnego dla programisty stosu.
Stos, owszem jest - ale obsługiwany niejawnie przez rozkazy **call** / **ret** oraz przez wywołanie przerwania i z niego powrót rozkazem **retr**. O rozkazach, znanych choćby z '51, typu *push*, *pop*....należy szybciutko zapomnieć. A jak wiadomo bardzo często potrzebujemy swoistej ochrony stanu rejestrów roboczych, szczególnie gdy program korzysta z przerwań. One wykonują się w tle i jeżeli obsługa przerwania zmodyfikuje nam zawartość rejestrów to program główny pójdzie w przysłowiowe maliny w dowolnie wybranym przez siebie momencie. A stos jest idealnym rozwiązaniem do chwilowego przechowywania nawet sporej ilości danych...o ile w ogóle jest.
Na szczęście 8042 (podobnie jak 8048) posiada alternatywny bank ośmiu rejestrów roboczych, wyboru banku dokonuje sie rozkazami **sel RB0** oraz **sel RB1**.
To jest niesamowite ułatwienie, ponieważ możemy napisać program tak, że obsługa przerwań korzysta z drugiego banku, pierwszy (podstawowy) zostanie na potrzeby programu głównego. Dodatkowo, powrót z przerwania rozkazem **retr** automatycznie ustawia jako aktywny ten bank, który był w użyciu w momencie zgłoszenia przerwania i skoku do procedury jego obsługi. Na naszej głowie pozostaje więc tylko ochrona akumulatora, a to jest dość łatwe - wymaga jednak poświęcenia jednego rejestru roboczego lub komórki pamięci wewnętrznej, do wyboru.

Druga, dość niemiła cecha 8042 to sposób dostępu do stałych zapisanych w pamięci programu czyli rozkaz **movp A,@A**. Rozkaz ten pobiera do akumulatora zawartość komórki pamięci programu o adresie wskazanym poprzednią (z chwili wykonania) zawartością tego rejestru. Akumulator jest ośmiobitowy, łatwo więc zgadnąć, że możemy rozkazem **movp** operować w zakresie adresów 00h...0FFh, a to nie jest

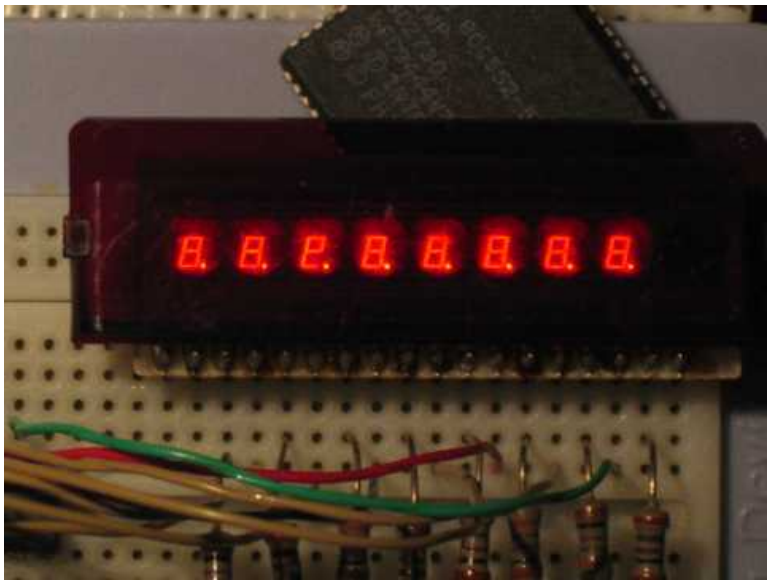
zbyt dużo i trzeba czasem dobrze pogłówkować, jak w pamięci umieścić stałe i tablice, tak aby program mógł w ogóle z nich skorzystać.

Trzecia ciekawosta - brak rozkazu odejmowania (czyli *sub* w jakiejkolwiek postaci). No nie ma i już! Oczywiście, możemy odejmować dodając wartość ujemną na przykład takim jak poniżej ciągiem rozkazów:

```
cpl A        ; zaneguj akumulator
add A,Rx     ; dodaj drugi argument z rejestru R0...R7
cpl A        ; ponownie zaneguj
```

Może tak jest nawet bardziej interesująco, pytanie tylko - kiedy stracimy cierpliwość...

Po tej dawce narzekań - wracamy do programu cqyp95.asm.
Program uruchamiałam etapami, najpierw wybór katod, potem dobór optymalnej częstotliwości multipleksowania, wreszcie - sekwencyjne wyświetlanie trzech napisów na wyświetlaczu. Filmik z działającym CQYP95 jest w **[4-h]**, a **fotografia 10** przedstawia
jeden z pierwszych etapów pracy: test wyświetlacza - czyli zapalenie wszystkich segmentów na wszystkich polach odczytowych.



fot. 10

Tu, z przykrością stwierdzam, display spłatał mi psikusa - jeden z segmentow trzeciej od lewej strony cyfry (segment C) - nie zaświecił. Przez dłuższą chwilę łudziłam się, że może to jakiś błąd w programie, ale niestety nie. Wystawienie samych jedynek na port P2 musiało wysterować wszystkie segmenty, skoro w całości świeciła poprzednia i następna cyfra - ta niekompletna okazała się jednak feralna. Szkoda trochę, ponieważ wyświetlacz wygląda naprawdę niezwykle, a cyferki pomimo że małe, są dość dobrze widoczne. Historię o CQYP95 zakończę trzema niewielkimi zdjęciami (**fotografie 11a**, **11b**, **11c**), na których widać statyczne napisy generowane przez program cqyp95.asm.

fot. 11a

fot. 11b

fot. 11c

Epilog (pomalutku kończymy)

No, jak ktoś dotarł do tego miejsca z lekturą - naprawdę gratuluję wytrwałości!
Artykuł ten powinnam zakończyć jakimś spektakularnym, powalającym na kolana projektem z wykorzystaniem mikrokontrolera 80C42...ale tego nie zrobię.
Szczerze mówiąc, brakuje mi pomysłu na coś rzeczywiście ciekawego, a kolejny zegarek czy mikroprocesorowe lampki choinkowe? To chyba byłoby nudne, więc na koniec tylko kilka zdań podsumowania.

Powyższy tekst to wyraźny sygnał, że przy odrobinie wysiłku stare, pozyskane ze złomu komputerowego mikrokontrolery można sensownie zagospodarować.
Pamiętajmy, że mikrokontroler mamy praktycznie za darmo (płyta PC przecież i tak idzie do kosza), procesor wprawdzie nie poraża mocą obliczeniową, ale za to jest dość wdzięczny do programowania (pomimo, że lista rozkazów powoduje pewien niedosyt).

Do tych, którzy zdecydują się pójść tą właśnie drogą - kilka słów, niestety w formie kubełka zimnej wody na głowę. Ponieważ we wszystkich zaprezentowanych układach procesor pracuje z zewnętrzną pamięcią programu, musimy mieć możliwość programowania pamięci EPROM/EEPROM lub posiadać emulator tychże pamięci, szczególnie typu 2764/2864. Bez tego po prostu nic nie zrobimy!
Jeżeli chodzi o programator - ja do pracy korzystam z LabTool-48, ale to jest sprzęt profesjonalny, wiec niejako poza konkursem. Polecam za to zainteresować się programatorem Willem, który wspomniane wcześniej kostki potrafi programować, a można go z powodzeniem wykonać w warunkach domowych.
Co do emulatorów - proszę nie posądzać mnie o kryptoreklamę, ale z tanich i skutecznych emulatorów EPROM ja osobiście polecam zestaw AVT270 (emulacja układów 2716...27512). Nie jest to droga rzecz, a posiadanie takiego emulatora otwiera zupełnie nowe możliwości. Ważne jest też to, że nie wymaga on specjalnego oprogramowania, ładowanie danych wykonuje się znanym z DOS poleceniem **copy** wywołanym z konsoli tekstowej Windows.
Niektórzy mawiają: "jeżeli czegoś nie ma w Google, to znaczy że nie istnieje".
A właśnie na kilku stronach internetowych wskazanych przez Google znalazłam wzmiankę,
że Intel opublikował (w latach siedemdziesiątych) podręcznik użytkownika do tej rodziny kontrolerów. Tytuł pozycji brzmi: "*MCS-48 and UPI-41 Assembly Language Manual*".
Niestety, publikacja ta jest niedostępna w formie elektronicznej, ale może przypadkiem znajduje się w zasobach uczelnianej biblioteki - polecam to sprawdzić.
I to chyba wszystko...
Kończąc, ewentualnym naśladowcom życzę przede wszystkim cierpliwości i wiary w swoje siły. I nie mówię, że będzie łatwo - ale za to na pewno będzie ciekawie...

Natasza Biecek
bienata@wp.pl

Referencje

[1] "*Recycling, czyli wykorzystanie elementów z odzysku...*", Forum Elportal.pl

http://elportal.pl/forum/viewtopic.php?t=6937

[2] SB-Assembler, San Bergmans
http://www.sbprojects.com/sbasm/sbasm.htm

[3] *"8042 and 8041 Microcontrollers"* na "*Devster Specialties*", Joseph Norman
http://devster.monkeeh.com/z80/upi42/

[4] Lista "*Digital Archaeology*" kanału *"bienata*", serwis YouTube, Natasza Biecek
a) "*NEC8042 - leds1.asm*" http://youtube.com/watch?v=jWKLWcrW2ks
b) "*NEC8042 - leds2.asm*" http://youtube.com/watch?v=D0HLLQjh4ig
c) "*NEC8042 - test0.asm*" http://youtube.com/watch?v=_AbYe91MRXE
d) "*NEC8042 - test1.asm*" http://youtube.com/watch?v=MEXbfzb9yKU
e) "*NEC8042 - p2latch.asm*" http://youtube.com/watch?v=1qizMmSgEvA
f) "*NEC8042 - counter1.asm*" http://youtube.com/watch?v=DxqkHbBmXP4
g) "*NEC8042 - cntrint.asm*" http://youtube.com/watch?v=2TVAp9fhnj4
h) "*NEC8042 - cqyp95.asm*" http://youtube.com/watch?v=cGjFaHIK2QQ

[5] "*Grokking the MCS-48 System*", Arnim Läuger
http://home.mnet-online.de/al/mcs-48/mcs-48.pdf

[6] "*Mikrokontrolery jednoukładowe rodziny MCS-48*", Andrzej Rydzewski

[7] "*8048*", Coprolite.com
http://www.coprolite.com/8048.html

[8] "*Mikrokomputer jednoukładowy 8048*", R-MIK
http://r-mik.eu/katalog8048/index.htm

[9] "*UPI-C42/UPI-L42 UNIVERSAL PERIPHERAL INTERFACE*", Intel
http://datasheets.chipdb.org/Intel/UPI-42/29041403.PDF

[10] archiwum z oprogramowaniem i materiałami dodatkowymi do artykułu
http://elportal.pl/tasza/8042/nec80c42.zip