WireIt! '51

Programowanie bez pisania

Wracamy na łamach EP do pakietu projektowego WireIt!, który umożliwia przygotowywanie aplikacji na procesorze '51, bez konieczności posługiwania się jakimkolwiek językiem programowania.

Dzień z życia konstruktora

Pracowałem wtedy w jednym z licznych zakładów elektronicznych niedaleko mojego domu. Jak na poniedziałek, godzina 8.00, czułem się doskonale - jedynie doskwierała mi głośna kłótnia w pobliskiej sali: gabinecie Szefa. Ponieważ z natury jestem ciekawski, podszedłem do drzwi. Mogłem więc dowiedzieć się, o czym tak energicznie rozprawiał Dyrektor: kolega z pracowni konstrukcyjnej przekroczył znacznie plan wydatków. Otóż do małego modelu kolejki - bo tym się właśnie zajmowała Firma - zaplanował zastosowanie sterowników PLC. Ponieważ - jak się okazało - nawet „długie" LOGO! miało za małą liczbę wyjść, użył kontrolera Simatic. Oczywiście, koszt jednego systemu sterowania wzrósł tak, że żaden klient nie chciał go kupić i trzeba było sprzedać 500 dotychczas wyprodukowanych egzemplarzy po cenie niższej od kosztów produkcji.

Kiedy wyszedł - na szczęście Szef pozwolił mu zostać w Firmie, obcinając jedynie premie na kilka lat - poradziłem mu zastosować procesory ST62. Argumentowałem, że przecież wspaniale się ich używa, bo jest dla nich stworzony nawet ST-Realizer, umożliwiający wygodne programowanie. Co on na to? Zrobił kwaśną minę i cierpko mi zakomunikował, ile wydał kiedyś własnych pieniędzy na układy OTP (w sumie 39), na których testował oprogramowanie. A układów EPROM nie kupował, bo pojedynczy układ był drogi, a trzeba by było mieć ze trzy do nanoszenia małych poprawek (jeden się kasuje, drugi jest w tym czasie programowany, a trzeci służy testom). Nigdy nie zebrał też dość pieniędzy, żeby kupić emulator sprzętowy - a programowy mu nie wystarczał.

Takie cele, jak np. obsługa modelu sygnalizacji świetlnej czy kolejki, można osiągnąć stosując tańsze i mniej zaawansowane układy rodziny MCS-51, w szczególności małe i tanie AT89C2051 i podobne im kontrolery z pamięcią Flash. Cechuje je prostota budowy układów otoczenia, łatwość zbudowania małego programatora - nawet na poczekaniu - i wyjątkowo niska cena. Niestety, trudno znaleźć dla nich jakiś odpowiednik ST-Realizera.

Program WireIt!

Skoro graficzne środowisko programistyczne dla małych i tanich układów jest niedostępne, należy stworzyć je samemu. Program napisany przeze mnie, służący do tego celu, nazywa się WireIt! i obecnie dostępna jest jego jedna wersja - dla 8051 (wspomagająca także układ ADuC812 firmy Analog Devices). Można ją w każdej chwili ściągnąć ze strony http://www.wireit.cjb.net jako archiwum ZIP. Radzę również czasem przeglądać to miejsce, bo to właśnie tam pojawiać się będą nowe i ulepszone wersje programu oraz biblioteki.

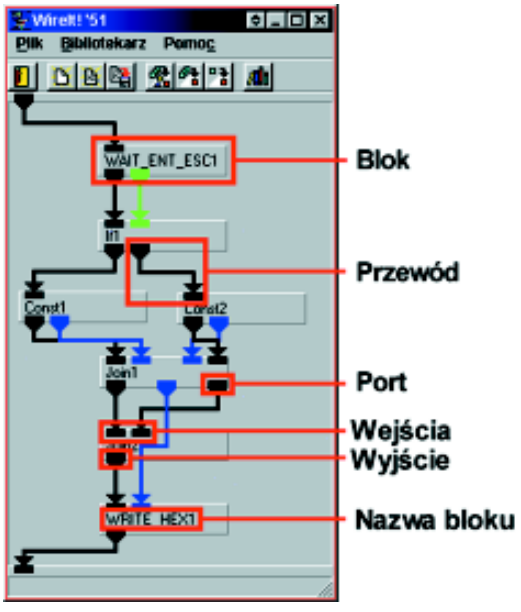
Idea programu odbiega nieco od ST-Realizera i jemu podobnych. Aby utworzyć aplikację w tych programach, trzeba narysować sieć elementów, współdziałających na zasadzie bramek logicznych. W WireIt! należy ułożyć elementy funkcjonalne (operacje arytmetyczne, komunikacja z użytkownikiem, sterowanie itp.), a następnie połączyć je przewodami. Czynność ta jest z pewnością znana każdemu elektronikowi, więc WireIt! powinien służyć nawet tym, którzy nie spotkali się przedtem z asemblerem 8051.

Podstawy pracy z programem

Aby „rozmawiać" z programem, trzeba poznać jego język. Element funkcjonalny, odpowiadający liście instrukcji asemblera, nazywa się *blokiem* (block) - **rys. 1**. Bloki mogą mieć *wejścia* i *wyjścia*, nazywane ogólnie *portami* (ports). *Porty* mogą też znajdować się na brzegu *obszaru projektowania* (designing area), wtedy służą jako element komunikacji projektu ze „światem zewnętrznym" i umożliwiają jego wykorzystanie jako pojedynczy *blok*, o czym później. Nazywamy je *portami globalnymi* (global ports). Każdy *blok*, podobnie jak *porty globalne,* powinien mieć unikalną nazwę. Jeśli pojawią się dwa *bloki* o tej samej nazwie, to asembler powinien wygenerować pewną liczbę komunikatów o błędach.

Prawie cała obsługa programu odbywa się za pomocą menu kontekstowego: można je wywołać, klikając na obszarze projektowania prawym przyciskiem myszki. Są tam operacje wstawiania i kasowania bloku, modyfikacji jego właściwości (properties), dodawania i usuwania *globalnych portów*, a także - co może się przydać nie tylko początkującym - możliwość uzyskania pomocy dotyczącej wybranego bloku.

Aby połączyć *wejście* i *wyjście* przewodem, należy przeciągnąć myszkę od jednego *portu* do drugiego (w obojętnej kolejności). Należy przy tym pamiętać, że *przewód* może łączyć tylko *wejście* i *wyjście* (i to w dodatku tego samego koloru) oraz o tym, że wszystkie *por-*



Rys. 1. Nazewnictwo elementów projektu.

Program: WireIt! 51

Przeznaczenie:
Projektowanie aplikacji sterowania dla 8051.

Cena:
0 zł (freeware).

Wymagania:
MS Windows 95, 16 MB RAM, 1,2 MB HDD.

Polecane:
MS Windows 95, 32 MB RAM, 1,2 MB HDD, ekran 800x600, 256 kolorów.

Uwaga:
System pomocy używa przeglądarki HTML (zalecany Netscape 4.x).

Źródło:
http://www.wireit.cjb.net (lub CD-EP5/2000 w katalogu \Programy\WireIt!\).

Autor:
Stanisław Skowronek.

Tab. 1.

P1.0	zielone A, zielone C	P1.4	czerwone C
P1.1	zielone B, zielone D	P1.5	czerwone D
P1.2	czerwone A	P1.6	żółte A
P1.3	czerwone B	P1.7	żółte B
P3.0	tryb pracy: 0V - normalna praca, 5V - awaria (żółte migające)		

ty muszą być połączone.

W pasku tytułowym programu znajduje się - mała ciekawostka - przycisk *RollUp* (**rys. 2**), którego kliknięcie powoduje zwinięcie okna do wysokości paska tytułowego i odkrycie Pulpitu.

Pierwszy projekt - *first.wis*

Przypuśćmy, że chcemy napisać program, odczytujący cyklicznie wartości z *portów* P1.0 i P1.1, a następnie ustawiający *port* P1.2 w stanie odpowiadającym ich logicznemu iloczynowi.

Najpierw, korzystając z menu kontekstowego umieśćmy blok odczytujący wartość z *portu* P1.0 (**rys. 3**). Po otwarciu okienka Block Properties wybierzmy bibliotekę IO, a z niej blok *InBit*. W skrócie możemy zapisać ten wybór jako *IO.InBit*. Potem musimy ustawić parametr *Bit name* na P1.0 i nacisnąć OK. Blok *InBit1* znajduje się już w naszym projekcie.

Analogicznie dodajmy kolejny blok *IO.InBit* z tym, że tym razem podamy inną nazwę bitu: P1.1. Potem możemy połączyć czarne wyjście bloku *InBit1* z czarnym wyjściem bloku *InBit2*, co będzie sygnalizowało kompilatorowi następującą kolejność wykonywania: najpierw wprowadź wartość z P1.0, potem dopiero z P1.1. W ogólności, aplikacja będzie zaczynała się w czarnym wyjściu w lewym górnym rogu obszaru projektowania, „poruszała" się wzdłuż czarnych przewodów i kończyła w lewym dolnym rogu.

Teraz wstawimy blok wykonujący najważniejszą część programu: blok *Boolean.AND*. Połączmy przewodem czarne wejście tego bloku z wyjściem *InBit2*. Jednak trzeba jeszcze wskazać kompilatorowi, na jakich wartościach ma operować dany blok. Do tego służą kolorowe przewody: zielony przekazuje pojedyncze bity, niebieski - bajty, a czerwony - słowa szesnastobitowe. Zatem jedno z zielonych wejść bloku *AND1* trzeba połączyć z wyjściem *InBit1*, a drugie - z wyjściem *InBit2*. W momencie wykonywania bloki te przypisują przewodom pewne wartości, zależne od stanu bitów, a blok *AND1* później pobiera te wartości, wykonuje na nich operację iloczynu, a potem ustawia swoje zielone wyjście zgodnie z wynikiem operacji.

Aby uzyskać żądany rezultat, tzn. aby ustawieniu uległ *port* P1.2, musimy dodać blok *IO.OutBit*. Jego czarne i zielone wejścia łączymy z odpowiednimi wyjściami *AND1*, a jako parametr *Bit name* podajemy P1.2.

Mamy już zaprojektowaną żądaną funkcję, ale program musi wiedzieć, że powinna ona być wykonywana w nieskończoność. Wstawmy blok *Standard.Label*, nadając parametrowi *Label* wartość np. "*Loop*". Wyjście w lewym górnym rogu obszaru projektowania łączymy z wejściem nowo wstawionego bloku, wyjście bloku z wejściem *InBit1*. Potem, na dole, wstawiamy blok *Standard.Jump* z parametrem *Label*, takim samym jak w bloku *Label1*, a jego wyjście łączymy z globalnym wyjściem na dole (choć w istocie program nigdy

Rys. 2. Wygląd przycisku "zwijającego" fragment okna roboczego WireIt!

nie podąży tą drogą), a wejście z wyjściem bloku *OutBit1*. Rezultat jest taki, że procesor najpierw „przechodzi" przez blok *Label1*, wykonuje nasz program, dochodzi do *Jump1*, po czym skacze do *Label1*, zamykając cykl. Program można zapisać do pliku .HEX, gotowego do zaprogramowania klikając przycisk *Build* (drugi od prawej w pasku narzędzi).

Przykłady zastosowania - *traffic.wis, light.wis*

Być może poprzedni przykład wydał się Czytelnikowi bardzo skomplikowany. Długość jego opisu jest raczej pochodną jego czysto dydaktycznej funkcji. Zauważmy, że wszystkie bloki użyte w projekcie odpowiadają praktycznie instrukcjom asemblera - stąd taki rozmiar projektu.

Teraz, aby wykazać przydatność programu do realizacji prostych systemów sterowania, zaprezentuję układ kontroli sygnalizatora świetlnego z możliwością przełączenia na „światła żółte migające". Przy okazji pokażę, jak można w WireIt! tworzyć własne bloki.

Zauważmy, że stale powtarzającym się elementem w układzie sygnalizacji świetlnej jest „zmiana świateł - pauza". Warto by było go zapisać jako jeden *blok*. Aby to uczynić, należy najpierw utworzyć *nowy projekt* (drugi przycisk od lewej), następnie ustawić *opcje kompilacji* (czwarty od prawej), wybierając format .WIO (**W**ire**I**t! **O**bject format). Wtedy można zacząć dodawać *porty globalne*. Najpierw dodajmy (w menu kontekstowym) *port wejściowy* 16-bitowy (o nazwie np. 'delay). Pojawi się czerwony *port globalny* u góry obszaru projektowania.

Wtedy dodajmy element *8-Bit.Word2Bytes*, łącząc jego wejścia z odpowiednimi portami globalnymi. Blok ten dzieli słowo na dwa bajty (starszy po lewej). Potem podłączmy do lewego niebieskiego wyjścia i czarnego wyjścia element *IO.OutPort* podając jako nazwę portu P1. Do wyjścia tego elementu i prawego wyjścia *Word2Bytes1* dołączmy blok *Thread.SecDelay* (czekający 125000 cykli * wartość wejścia). Połączmy wyjście tego elementu z globalnym wyjściem i skompilujmy projekt przyciskiem *Compile* lub *Build* (odpowiednio trzeci i drugi od prawej). Podajmy, odpowiadając na pytanie, jakąś nazwę pliku, do którego ma być wpisany utworzony kod wynikowy.

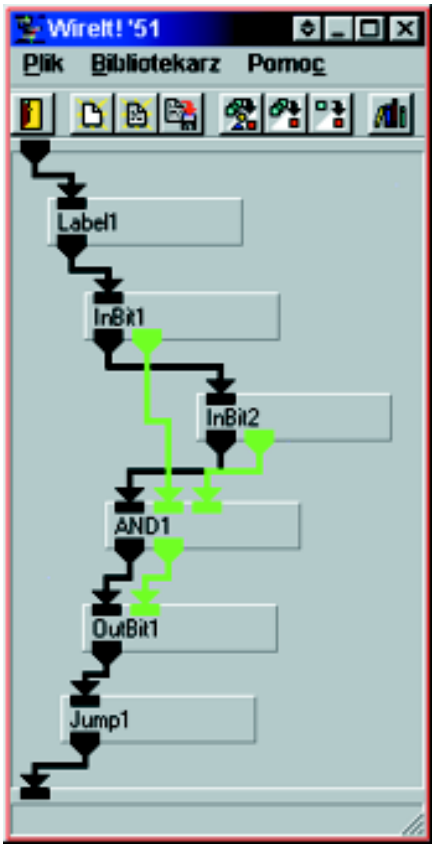
Teraz przyszedł czas, aby dodać nowy obiekt do zestawu bloków. Użyjmy do tego celu *Bibliotekarza* (Librarian), którego możemy wywołać wybierając opcję z menu lub naciskając pierwszy przycisk od prawej. Ustawmy listę bibliotek na *User* i naciśnijmy *Add*. W pojawiającym się oknie dialogowym zaznaczmy nasz plik WIO (wynik kompilacji) i wciśnijmy *Otwórz*. Zaznaczmy *Design* i - aby nadać mu bardziej odpowiednią nazwę - naciśnijmy *Rename*. W polu poniżej wpiszmy *Light* i opuśćmy *Bibliotekarza*, naciskając OK.

Jak łatwo sprawdzić, mamy nowy blok: *User.Light*. Wykorzystajmy go, tworząc program sterowania modelem sygnalizacji świetlnej. Moja propozycja połączeń portów znajduje się w **tab. 1**.

Projekt składa się z dwóch części. W związku z tym są dwa „wątki" (ciągi bloków połączonych czarnymi przewodami), jeden obsługujący normalny tryb pracy, drugi - awaryjny. Przełączenie między nimi może nastąpić tylko na początku cyklu, kiedy zostaje odczytany (znanym już nam blokiem *IO.InBit*) port P3.0. Wartość wyjściowa tego bloku podawana jest na wejście bloku *Thread.If* (podobnego do występującej w językach wysokiego poziomu konstrukcji if..then..else..). Jeśli jest ona prawdziwa (port w stanie wysokim), sterowanie przekazywane jest do prawego wyjścia, w przeciwnym razie - do lewego. Same wątki składają się z naprzemiennie ułożonych, połączonych bloków *16-Bit.Const* i *User.Light*. Wyższy bajt podany w parametrach *16-Bit.Const* oznacza wartość wysyłaną do portu P1 mikrokontrolera, niższy - czas oczekiwania po wysłaniu tej wartości (w ćwiartkach sekundy). Na końcu wątki się łączą (blok *Thread.Join*), a cały program zawarty jest w niekończącej się pętli, znanej nam z poprzedniego przykładu.

Kolejny przykład zastosowania - *codelock.wis*

Aby pokazać, jak łatwo tworzy się silne i uniwersalne oprogramowanie w WireIt!, zaprezentuję przykład drugi: zamek kodowy, którego oprogramowanie zostało w całości „narysowane" w WireIt! '51. Układ składa się z bardzo niewielu elementów, może być wykonany na małej jednostronnej płytce drukowanej - a jego możliwości wcale nie są takie małe. Ma sześciocyfrowy kod, zapisany w pamięci nieulotnej EEPROM. Szyfr może być zmieniony przez użytkownika tylko po podaniu właściwego kodu. Podanie złego kodu powoduje blokadę zamka na 10 sekund, czas otwarcia zamka po podaniu właściwego kodu wynosi 3 sekundy. Zamek sygnalizuje swój



Rys. 3. Pierwszy projekt (first.wis) w WireIt!

stan dwiema diodami świecącymi LED i ma wyjście sterujące o aktywnym stanie niskim.

Na **rys. 4** znajduje się przykładowy schemat zamka, obrazujący, jakie należy wykonać połączenia z elementami zewnętrznymi. W programie przyjęto, że częstotliwość oscylatora kwarcowego wynosi 12MHz (jedyną różnicą wynikłą z jej zmniejszenia będzie odpowiednie wydłużenie wszystkich okresów oczekiwania), dioda D1 jest czerwona, a D2 - zielona.

Jeśli komuś zależy na większej klawiaturze, może zastosować matrycę o rozmiarze 4x4, podłączając dodatkową kolumnę do P1.3 (program nie będzie wymagał zmiany). Podkreślam, że w procedurze odczytującej matrycę niektóre kombinacje wielu klawiszy mogą być odróżnione od innych (zatem można je włączyć do kodu oprócz pojedynczych naciśnięć klawiszy). Łączna liczba kombinacji w tak rozbudowanym zamku wzrośnie z 1000000 do 225^6, czyli ok. 130000 miliardów. Nie ma jednak róży bez kolców, bo kod składający się z sześciu kombinacji po kilka naciśnięć jest dość trudno zapamiętać.

Teraz o obsłudze zamka. Otóż jest jeden klawisz specjalny: nazwijmy go *Ctrl*. Jest on tym klawiszem, który jest podłączony między P1.0 a P1.4. Po wprowadzeniu sześciu cyfr (czy kombinacji) poprawnego kodu (które mogą zawierać w sobie dowolne klawisze, także *Ctrl*) należy nacisnąć dowolny klawisz w celu zatwierdzenia wyboru. Jeśli to będzie *Ctrl*, zamek będzie oczekiwał wprowadzenia kolejnych sześciu cyfr w celu zachowania ich w pamięci EEPROM jako nowego klucza, sygnalizując gotowość zapaleniem obu diod świecących. W przeciwnym przypadku zamek zapali zieloną diodę świecącą D2 i otworzy na trzy sekundy blokadę drzwi (lub też cokolwiek innego reagującego na przejście portu WY w stan niski). Jeśli natomiast zostanie wprowadzony nieprawidłowy kod, układ zablokuje się na dziesięć sekund (przy kwarcu 12MHz) i nie będzie odpowiadał na zlecenia z klawiatury. W tym czasie zapali się czerwona dioda D1.

Moja propozycja rozbudowy programu to dodanie licznika niewłaściwych kodów, umieszczonego np. w bajtach EEPROM numer 06h i 07h (bajty 00h..05h są zajmowane przez kod). Warto wykorzystać tu arytmetykę 16-bitową (biblioteka *16-Bit*). Ponadto warto by było dodać sumę kontrolną kodu (myślę o bajcie 08h). Radzę tworzyć tę sumę z operacji XOR na komórkach 00h..05h. Jeżeli ktoś zbuduje tak rozszerzony układ, proszę bardzo o przysłanie mi biblioteki i programu na adres *zamek@wireit.cjb.net*.

Kompilacja: jak to działa?

Pod graficzną powłoką aplikacji znajduje się całkiem normalny język programowania wysokiego poziomu. Przecież wszystkie pliki danych programu mają postać tekstową, zarówno biblioteki (.WIL), jak i plik opisu projektu (.WIS). Właściwie więc, jeżeli ktoś tego potrzebuje, może korzystać z programu używając jako edytora Notatnika Windows czy nawet edytora Nortona Commandera (w ten sposób powstały wszystkie biblioteki standardowe). Powstaje więc pytanie: jak WireIt! tłumaczy rysunek i plik wejściowy na gotowy kod asemblera 8051?

Bloki odpowiadają fragmentom asemblera MCS-51, które są następnie układane wzdłuż wątków, tj. najpierw cały wątek składający się z lewych czarnych wyjść i wejść, potem kolejne „pionowe“ wątki. Parametry bloku są wstawiane w miejsce tzw. *placeholderów* (dosłownie: „trzymaczy miejsca“, idea odpowiada „staczom“ w kolejkach) {BP0}, {BP1}, {WP0}, {WP1}, {SP0}. Każdy blok ma swój identyfikator, składający z jego numeru (wewnętrznie przypisywanego przez WireIt!) i numeru kompilacji (czterocyfrowej liczby tworzonej z bieżącego czasu). Ten identyfikator odpowiada *placeholderowi* {MyID}, tak jak numer kompilacji odpowiada {CompID}. Identyfikatory są konieczne przy tworzeniu etykiet w programie, aby nie nastąpił konflikt, czyli aby dwie etykiety nie miały tych samych nazw, co zresztą tłumaczy niemal kompletną nieczytelność programu skompilowanego przez WireIt!

Pozostaje jeszcze jeden problem: reprezentacja kolorowych przewodów. Każdy przewód ma swoje komórki wewnętrznej pamięci RAM kontrolera: zielone i niebieskie - po jednym bajcie, czerwone - dwa bajty. Komórki są zajmowane zawsze od RAMTOP-u, czyli - dla układu 8051 - od 127 bajtu w dół. Program przypisuje numery komórek przewodom i może czynić to na dwa sposoby (wybór możliwy w oknie *Compilation options*). Pierwszy z nich, standardowy, to sposób z optymalizacją, drugi - bez.

Optymalizacja polega na tym, że program przypisuje przewodom komórki, które na pewno są wolne podczas wykonywania fragmentu kodu zawierającego ten przewód. Algorytm sterujący tym procesem po prostu „maszeruje“ wzdłuż czarnych przewodów, zaznaczając komórki zajęte dotychczas przez pozostałe kolorowe przewody i bloki (które też mogą mieć „prywatne“ komórki, dostępne tylko podczas pracy danego bloku), a następnie przegląda trasę przewodu, któremu chcemy przypisać komórki i nadaje mu odpowiedni numer nie zajęty przez pozostałe. Należy pamiętać jednak, że jeśli projekt będzie instalował własne przerwania sprzętowe, trzeba koniecznie wybrać przypisywanie przewodów bez optymalizacji. Jest to tak istotne, bo program nie wie, kiedy zostanie wywołane przerwanie sprzętowe i może przypisać komórki wykorzystane w procedurze obsługi przerwania innym przewodom, co może mieć nieprzewidywalne następstwa.

Po przypisaniu numerów komórek przychodzi czas na tworzenie kodu. Program odczytuje z bibliotek skrawki asemblera odpowiadające poszczególnym blokom i linijka po linijce wpisuje je do programu wynikowego, zamieniając *placeholdery* na odpowiednie wartości lub numery komórek pamięci (przy czym *placeholder* {nazwa portu} lub {nazwa portu /} odpowiada niższej komórce pamięci dla danych 16-bitowych, a {nazwa portu \} - wyższej).

Ten kod składa się w przeważającej mierze z mnemoników AJMP, zajmujących po dwa takty zegara. Dlatego program przeprowadza pewną „małą“ optymalizację (którą także można wyłączyć - co radzę raz zrobić, aby zobaczyć różnicę w szybkości wykonania), polegającą na tym, że usuwane są wszystkie skoki do następnej instrukcji.

Wtedy można już zapisać gotowy kod do pliku, przy czym format pliku wyjściowego zależy od tego, czy ma być to obiekt .WIO, przeznaczony do włączenia do bibliotek, czy też samodzielny program w asemblerze. Programy w asemblerze zawsze zamykane są nieskończoną pętlą, aby zapobiec nieprzewidywalnemu zachowaniu się systemu mikroprocesorowego.

Jak wspomniałem wcześniej, w programach typu ST-Realizer projektuje się program tak, jak schemat zwykłego układu elektronicznego, a nie tak, jak schemat blokowy (WireIt!). Pozornie te dwa sposoby bardzo się od siebie różnią. Jednak można zastosować podobne algorytmy, z tym, że usunie się czarne przewody, a zamiast nich zostaną wstawione do każdego bloku procedury, działające według algorytmu:

- pobierz wartość wejść bloku,
- oblicz wartość wyjść bloku,
- jeśli wyjścia mają inną wartość niż przedtem, to wywołaj procedury obsługi wszystkich bloków połączone z tymi wyjściami.

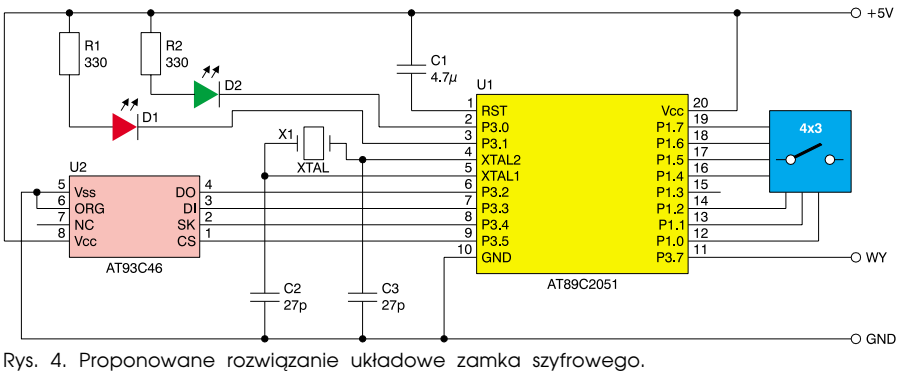
Podstawowe algorytmy, takie jak np. przypisywanie komórek pamięci, mogą zostać bez problemu przeniesione z WireIt!

Zakończenie

Jeżeli Czytelnik uznał program za przydatny, cieszę się. Jeżeli chciałby coś poprawić, może skorzystać z usług poczty elektronicznej (na przykład wybierając *Mail suggestions/Wyślij komentarz z menu Help/Pomoc*) i przesłać mi swoje uwagi na adres: *ep@wireit.cjb.net*. Gorąco zachęcam także do przysyłania swoich bibliotek - opublikuję je na stronie WireIt! Warto także odwiedzać w WWW adres mojego programu, bo WireIt! się przecież ciągle rozwija i zmienia. Pojawiać się też tam będą nowe biblioteki (w planach: biblioteka I²C o organizacji bajtowej, obsługa układów MAXIM-a, itd.) i nowe wersje programu (np. dla układów Atmel AVR z pamięcią RAM, tj. od AT90S2313 w górę).

Stanisław Skowronek

Pakiet WireIt! wraz z przykładami, w wersjach polsko- i angielskojęzycznej są dostępne w Internecie pod adresami:
http://www.wireit.cjb.net
http://www.ep.com.pl/ftp
oraz na płycie CD-EP5/2000 w katalogu \Programy\WireIt!\.



Rys. 4. Proponowane rozwiązanie układowe zamka szyfrowego.