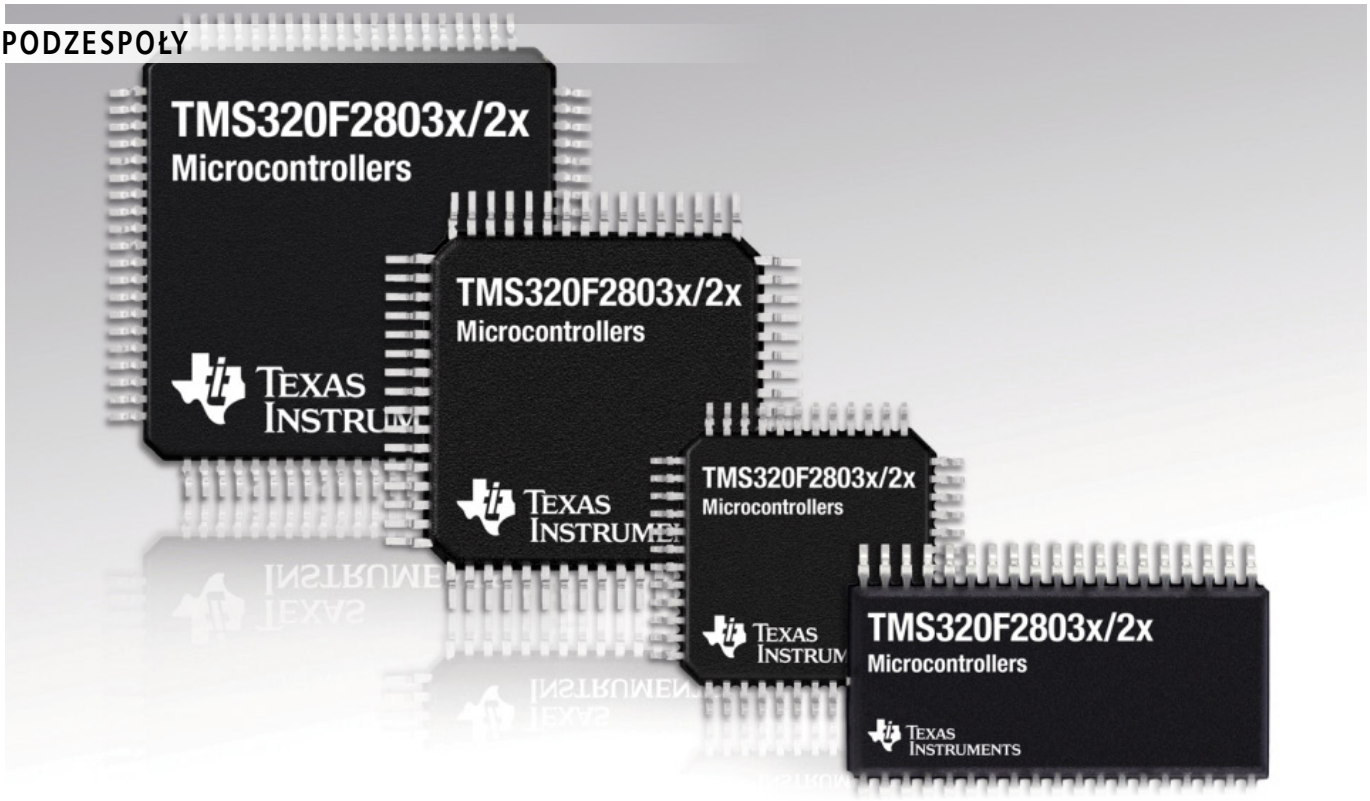


Mikrokontroler Piccolo z rdzeniem C28x

Rdzeń C28x jest używany m.in. w mikrokontrolerach czasu rzeczywistego należących do rodziny Piccolo. Warto zapoznać się bliżej z architekturą C28x, ponieważ znając szczegóły budowy rdzenia i stosując, tam gdzie jest to uzasadnione, kod pisany w asemblerze, można znacznie zwiększyć wydajność algorytmów, przy jednoczesnym zmniejszeniu rozmiarów kodu wynikowego.

Do celów niniejszego artykułu wykorzystano zestaw startowy controlSTICK, w którym producent, Texas Instruments, zamontował mikrokontroler TMS320F28027. Wspomniany zestaw startowy był wielokrotnie przedstawiany na łamach EP. Zamieszczone w artykule informacje dotyczą aplikacji tworzonych w środowisku Code Composer Studio 4, jednak nic nie stoi na przeszkodzie, aby uruchomić je także w starszej wersji tego środowiska.

Rdzeń C28x

Rdzeń C28x jest efektywną 32-bitową jednostką stałoprzecinkową typu RISC o zmodyfikowanej architekturze harwardzkiej. Schemat blokowy rdzenia zamieszczono na **rysunku 1**. Architekturę C28x zoptymalizowano pod kątem wydajnego wykonywania algorytmów cyfrowego przetwarzania sygnałów. Z tego powodu konstruktor otrzymuje do dyspozycji sprzętowy układ mnożenia oraz wsparcie dla bufora kołowego. Istotne jest również zwielokrotnienie magistral, których w sumie jest sześć: trzy adresowe i trzy danych. Magistrala adresowa dla pamięci programu jest 22-bitowa, co pozwala na zaadresowanie 4M słów (słowo to 16 bitów) przestrzeni dla programu. Przestrzeń adresowa dla danych wynosi 4G słów z racji 32-bitowej magistrali danych.

Rejestry

Mapę rejestrów rdzenia C28x przedstawiono na **rysunku 2**. Rdzeń C28x ma osiem 32-bitowych rejestrów ogólnego przeznaczenia. Modyfikacja lub odczyt części rejestrów (m.in. do akumulatora ACC) nie musi oznaczać operacji na wartościach 32-bitowych. Przykładowo rejestry ACC, P, XT umożliwiają bezpośredni dostęp do starszego lub młodszego słowa. W przypadku akumulatora odwołanie do młodszego słowa zapewnia alias AL, natomiast do starszego AH. Akumulator jest rejestrem przeznaczenia dla większości operacji, z wyjątkiem tych, które operują bezpośrednio na rejestrach lub na pamięci.

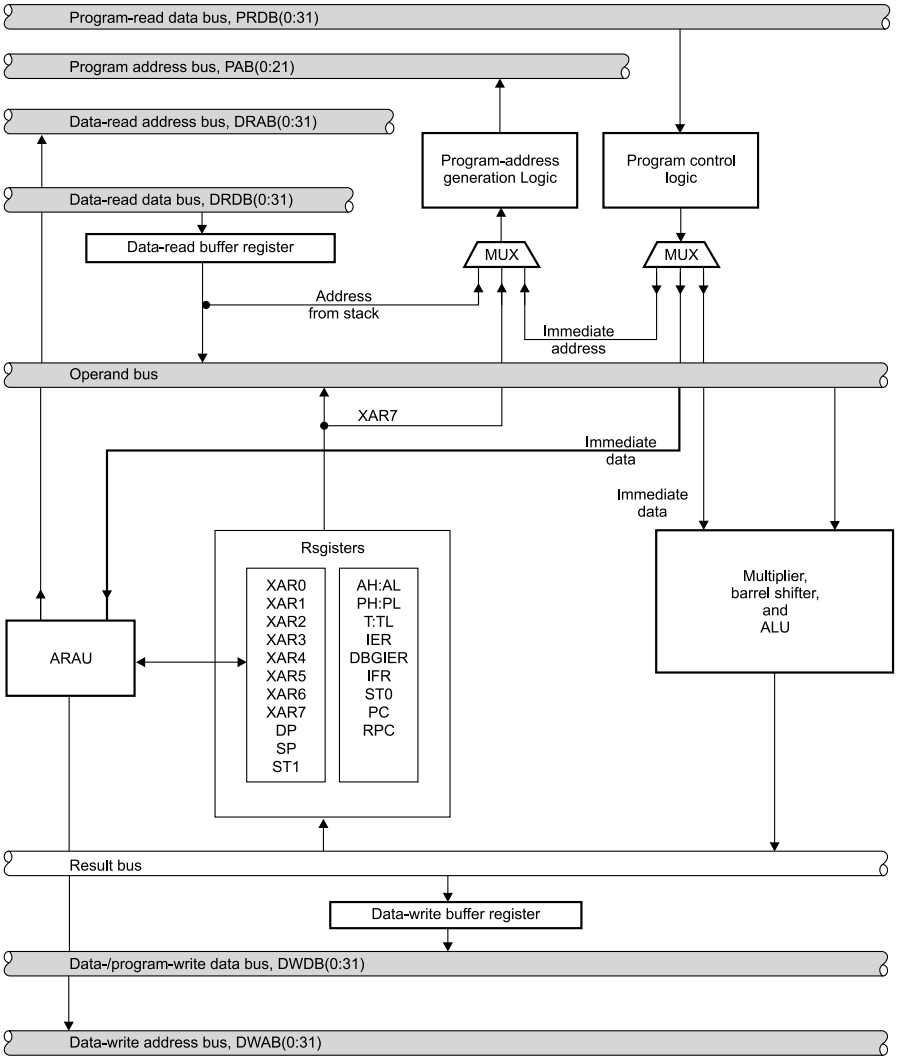
Rejestr XT jest rejestrem mnożnych, natomiast rejestr P zawiera produkt mnożenia. PC to oczywiście licznik rozkazów, który wskazuje na aktualnie wykonywaną instrukcję. Adres powrotu jest przechowywany w rejestrze RPC (*Return Program Counter*). Pozostałe rejestry to: rejestry statusu ST0 i ST1 oraz rejestry służące do zarządzania przerwaniami.

Tryby adresowania

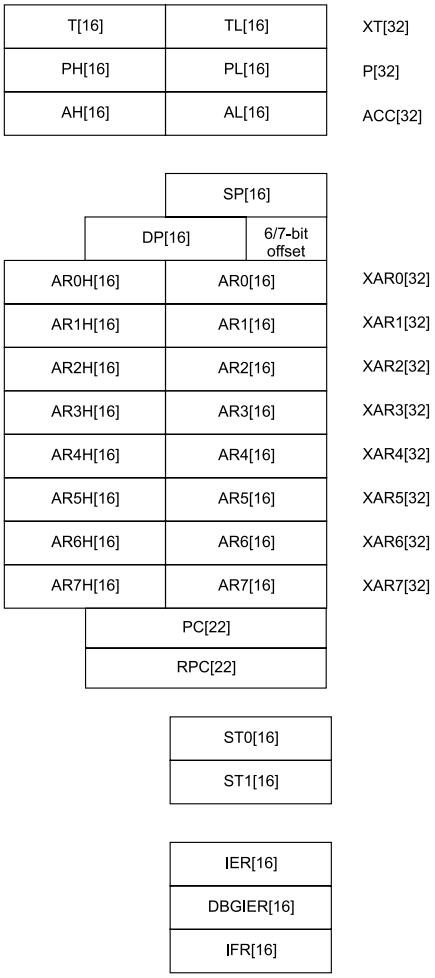
Podobnie jak większość mikroprocesorów, architektura C28x wyróżnia cztery tryby adresowania:

- bezpośrednie,

Dotychczas w EP na temat mikrokontrolerów czasu rzeczywistego TMS320F28027 ukazały się artykuły:
EP 08/2009 – „Mocarny maluch. Piccolo najmniejszy DSP z Texasa";
EP 09/2009 – „DSP w praktyce. Konfiguracja przetwornika A/C w TMS320F28027";
EP 10/2009 – „DSP w praktyce. Generator PWM i interfejs SCI";
EP 01/2010 – „Zestaw DSP controlSTICK w praktyce. Eksperymentalna platforma DSP";
EP 03/2010 – „Zestaw DSP controlSTICK w praktyce. Szybkie przekształcenie Fouriera".



Rys. 1. Schemat blokowy rdzenia C28x [Źródło: TI]



Rys. 2. Mapa rejestrów rdzenia C28x [Źródło: TI]

- stosu,
- pośrednie,
- rejestrowe.

Do adresowania bezpośredniego służy 16-bitowy rejestr DP (*Data Page Pointer*), którego stosowanie może wpływać na zwiększenie wydajności przy operowaniu stałymi strukturami danych (np. tablicami globalnymi). Adresowanie rejestrowe nie wymaga komentarza, pozostałe dwa tryby omówiono poniżej.

Operacje na stosie

Domyślnie po zerowaniu rdzenia wierzchołek stosu znajduje się pod adresem 0x0400. Adres kolejnego pustego elementu stosu jest przechowywany w rejestrze SP (*Stack Pointer*). Stos jest zwiększany w kierunku rosnących adresów. Każdy element stosu jest długości słowa (16 bitów), zatem odłożenie na stos wartości 32-bitowej powoduje zwiększenie wskaźnika stosu o dwa elementy.

Najprostszym przypadkiem wykorzystania stosu jest używanie instrukcji POP i PUSH w odniesieniu do wartości 16-bitowych. Przykładowo odłożenie na stos zawartości młodszego słowa akumulatora ACC może odbywać się przez wywołanie instrukcji PUSH AL. Wskaźnik stosu zostanie automatycznie zaktualizowany (zwiększony o 1). Jeśli odłożona zostanie zawartość całego akumulatora, to wskaźnik stosu ulegnie automatycznemu zwiększeniu o 2. Analogicznie ma się sprawa operacji zdejmowania ze stosu.

Stosując instrukcje PUSH i POP, programista nie musi martwić się o poprawną zawartość wskaźnika stosu SP. Istnieje jednak możliwość całkowicie ręcznego sterowania zawartością stosu i jego wskaźnikiem. Identyczną w efektach jak wyżej przedstawiona instrukcja PUSH AL jest poniższa linia kodu:
MOV *SP++, AL.

Pod adres będący aktualnym stanem wskaźnika stosu jest zapisywana 16-bitowa zawartość rejestru AL, a następnie rejestr SP jest postinkrementowany. W wyniku tych operacji następuje ręczne ustawianie wskaźnika na kolejny pusty element. Zdejmowanie ze stosu wartości 16-bitowej bez użycia instrukcji POP można uzyskać za pomocą instrukcji:
MOV AL, *--SP

Należy zwrócić uwagę na predekrementację rejestru wskaźnika stosu – rejestr SP wskazuje zawsze kolejny pusty element, więc aby zdjąć element, trzeba w pierwszej kolejności zmniejszyć adres wskaźnika stosu.

Oprócz skomplikowanych możliwości manipulowania zawartością rejestru wskaźnika stosu SP można także odczytywać i zapisywać zawartość stosu bez uprzedniego zdejmowania elementów znajdujących się wyżej. Operacje odczytu elementu znajdującego się o sześć adresów w głąb stosu realizuje poniższy kod:
MOV ACC, *-SP[6]

Pojedynczy znak '-' oznacza, że instrukcja odwołuje się pod adres w pamięci mniejszy od aktualnie wskazywanego przez rejestr SP o taką wartość, jaka jest podana w nawiasach.

Adresowanie pośrednie

Adresowanie pośrednie można potraktować jako odmianę niskopoziomowych wskaźników. W rejestrach adresowania pośredniego zapisywane są adresy w pamięci, na których mają być przeprowadzane operacje. Wszystkie osiem rejestrów XARn (gdzie n jest od 0 do 7) rdzenia C28x mogą być użyte w roli wskaźników do pamięci. Przykład wykorzystania rejestru XAR4 w adresowaniu pośrednim przedstawia fragment kodu w asemblerze zamieszczony na **listingu 1**. Jest to kod asemblera wywoływany

List. 1. Adresowanie pośrednie za pomocą rejestru XAR4

```
        .def _func

        .global  __func
_func:
        MOV      AL, #9
        RPT      AL
        || MOV   *XAR4++, #0

        LRETR
```

jako funkcja z poziomu Języka C, a jego zadaniem jest zerowanie 10 kolejnych komórek pamięci, począwszy od adresu zawartego w rejestrze XAR4. Aby wspomniana funkcja była widoczna w pliku *.c, musi być w nim zadeklarowana jako zewnętrzna (extern), po to, żeby kompilator wiedział, iż jej definicja znajduje się w innym pliku.

W tym miejscu komentarza może wymagać konstrukcja kodu w asemblerze i mogący budzić zdziwienie znak '||'. Instrukcja RPT powoduje, że następująca po niej linia kodu będzie wykonywana tyle razy, ile wynosi wartość operandu RPT. Dwie pionowe kreski ('||') mają za zadanie dodatkowo podkreślić, że dana linia kodu będzie wykonywana wiele razy. Warto tutaj wspomnieć, że kod następujący po rozkazie RPT będzie się powtarzał n + 1 razy.

Program w języku C przedstawiono na **listingu 2**. Tworzone są dwie tablice: tab1 i tab2, pierwsza jest globalna, a druga lokalna, zdefiniowana w funkcji main(). Tablica globalna tab1 zostanie umieszczona w przestrzeni pamięci danych, druga tablica tab2 jest obiektem lokalnym, czyli będzie utworzona na stosie. Każda z tablic jest inicjowana, by zachowanie mikrokontrolera było powtarzalne i jednoznaczne. Zaprezentowana deklaracja funkcji asemblerowej sprawia, że adres pierwszego elementu przesyłanego obiektu będzie znajdował się we wspomnianym już wyżej rejestrze XAR4.

Jako pierwsza do funkcji func() przekazywana jest tablica globalna tab1. Podczas debugowania programu łatwo można sprawdzić adres początku tablicy. Jeśli pułapka zostanie ustawiona na początku kodu funkcji w asemblerze, to po wstrzymaniu pracy MCU w tym

List. 2. Program w języku C wywołujący funkcję napisaną w asemblerze

```
extern void func(char*);

char tab1[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

void main(void)
{
    char tab2[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};

    DeviceInit();        // Inicjalizacja MCU
#ifdef FLASH
    MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);
    InitFlash();         // Inicjalizacja pamieci FLASH
#endif

    func(tab1);
    func(tab2);
}
```

⊞ XAR0	0x00000484
⊞ XAR1	0x0000FFFF
⊞ XAR2	0x00000000
⊞ XAR3	0x00000000
⊟ XAR4	0x00008900
AR4H	0x0
AR4	0x8900
⊞ XAR5	0x0000891C
⊞ XAR6	0x0000000A
⊞ XAR7	0x003D7C80

Rys. 3. Zawartość rejestru XAR4 po przesłaniu argumentu globalnego

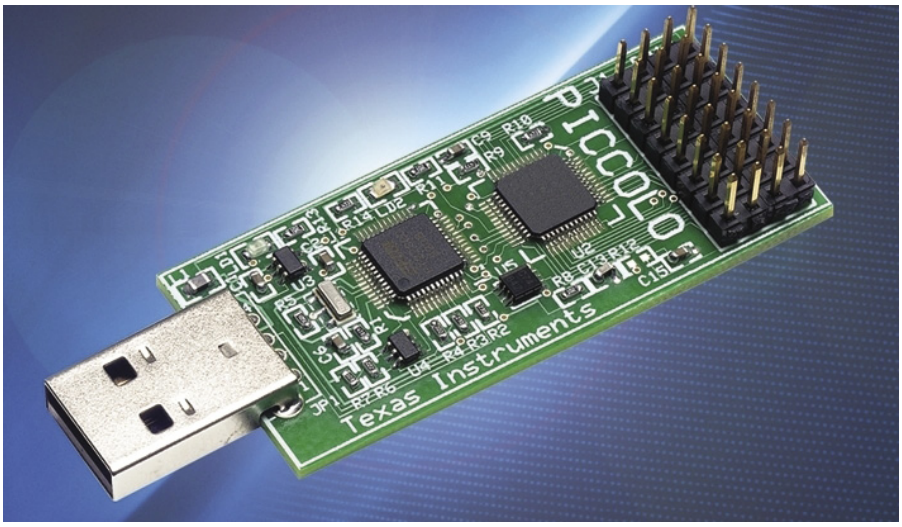
⊞ XAR0	0x00000484
⊞ XAR1	0x0000FFFF
⊞ XAR2	0x00000000
⊞ XAR3	0x00000000
⊟ XAR4	0x00000484
AR4H	0x0
AR4	0x484
⊞ XAR5	0x0000891C
⊞ XAR6	0x0000000A
⊞ XAR7	0x003D7C80

Rys. 4. Zawartość rejestru XAR4 po wysłaniu argumentu lokalnego

miejscu rejestr XAR4 powinien zawierać adres początku tablicy, w przedstawianym przypadku jest to 0x8900, co potwierdza **rysunek 3**.

W drugim wywołaniu funkcja func() ma przekazywany w argumencie adres pierwszego elementu tablicy tab2. Jest to tablica lokalna, zatem powinna być umieszczona na stosie, co udowadnia **rysunek 4** przedstawiający m.in. zawartość rejestru XAR4.

Podobnie jak było to w przypadku operacji na stosie, tutaj również istnieje możliwość odwoływania się w jednej instrukcji do komórek w pamięci innych, aniżeli aktualnie wskazywana przez adres w rejestrze adresowania pośredniego. Załóżmy, że mamy tablicę 20 elementów. W rejestrze XAR4 znajduje się adres początku tablicy, jednak wymagany jest zapis zawartości akumulatora do dziewiątego elementu tablicy. W takim przypadku nie ma potrzeby zwiększania zawartości rejestru adresowania pośredniego. Wystarczy podać przesunięcie w stosunku do aktualnie przechowywanego adresu. Opisane wyżej zadania realizuje jedna linia kodu w asemblerze: MOVL *+XAR4[9], ACC.



Potok rozkazów/przetwarzanie potokowe

Przetwarzanie potokowe pozwala na zwiększenie efektywności wykonywania programu, choć należy zdawać sobie sprawę, że może być ono również źródłem niepoprawnego działania aplikacji. Potok rozkazów rdzenia C28x jest podzielony na osiem faz, których celem jest wykonanie pięciu podstawowych operacji:

- pobranie rozkazu z pamięci,
- dekodowanie instrukcji,
- odczytanie danych z pamięci lub rejestrów,
- wykonanie instrukcji,
- zapisanie wyników do pamięci lub rejestrów.

Dodatkowa fragmentacja pięciu powyższych etapów ma na celu zwiększenie efektywności pracy CPU. Wszystkie osiem faz pokrótce omówiono poniżej.

Fetch 1 – W tej fazie rdzeń adresuje pamięć programu poprzez 22-bitową magistralę.

Fetch 2 – Następuje odczyt rozkazu przez 32-bitową magistralę danych programu oraz załadowanie instrukcji do kolejki.

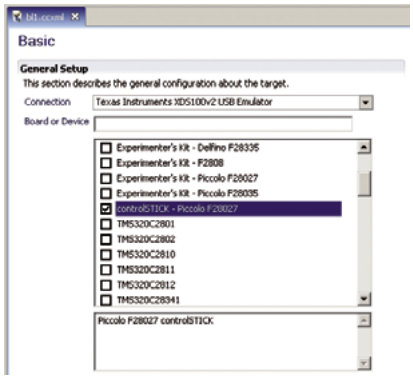
Decode 1 – Instrukcje dla rdzenia C28x mogą być zarówno 32-jak i 16-bitowe, ponadto wyrównywanie może być do adresów parzystych lub nieparzystych. Zadaniem fazy Decode 1 jest określenie rozmiaru instrukcji oraz jej wyrównywania.

Decode 2 – Rozkaz jest ładowany z kolejki rozkazów do rejestru rozkazów. Dekodowanie jest zakończone, więc na tym etapie są już wygenerowane, w zależności od wymagań zdekodowanej instrukcji m.in. adres odczytu lub zapisu z/do pamięci.

Read 1 – Jeśli dane mają być odczytane z pamięci, to odpowiedni adres jest wystawiany na magistralę adresową.

Nowa wersja Code Composer Studio

Na stronach internetowych Texas Instruments dostępna jest nowa edycja pakietu Code Composer Studio w wersji 4. Środowisko zbudowano na bazie zyskującego coraz większą popularność Eclipse. Nowe oprogramowanie współpracuje z zestawem startowym controlSTICK, jednak niezbędna jest, podobnie jak miało to miejsce w przypadku poprzedniej wersji Code Composera, instalacja emulatora XDS100. Następnie, podczas tworzenia nowego (edytowania istniejącego) projektu dla zestawu controlSTICK, należy wybrać opcję konfiguracji sprzętu (menu *Target/New target configuration*) zaznaczoną na rysunku poniżej.



Rys. Konfiguracja pakietu Code Composer Studio v.4 dla zestawu controlSTICK

Read 2 – Podczas tej fazy, jeśli jest taka potrzeba, następuje odczytanie danych z pamięci.

Execute – Nazwa fazy jest jednoznaczna, następuje tutaj wykonanie rozkazu.

Write – Jeśli produkt wykonania instrukcji wymaga zapisania do pamięci, następuje to w tej ostatniej fazie.

Cały potok rozkazów jest ponadto podzielony na dwie grupy. Pierwsza obejmuje etapy F1–D1, a druga D2–W. Podział taki umożliwia m.in. zwiększenie efektywności pracy programu z obsługą przerwań. Gdy CPU zarejestruje wystąpienie w systemie przerwania, wtedy porzucane są jedynie instrukcje będące w fazach F1–D1. Wszystkie instrukcje, które znajdują się w fazach D2–W, zostają wykonane. Działa to również w drugą stronę. Jeśli z jakiegoś powodu zablokowane zostanie wykonywanie instrukcji D2–W, to pobieranie rozkazów w pierwszych fazach nie zostanie wstrzymane.

Równoległe przetwarzanie potokowe narażone jest na zakłócenia zwane hazardami. Rdzeń C28x ma sprzętową ochronę przed dwoma rodzajami hazardów: konfliktem rejestrów i konfliktem zapisu-odczytu. Unikanie wymienionych błędów polega na automatycznym dodawaniu pustych cykli do przetwarzania potokowego. Zrozumienie mechanizmów powstawania hazardów pozwala na takie pisanie kodu, które pozwoli ich uniknąć, a co za tym idzie pozwoli zwiększyć efektywność wykonywania programu, ponieważ nie będą dodawane zbędne cykle.

Błąd zapisu-odczytu może wystąpić, gdy dwie następujące po sobie instrukcje będą operować na tej samej komórce pamięci. Pierwsza instrukcja zapisuje wartość pod adres X, a następna czyta zawartość tej komórki pamięci. Z powodu równoległego przetwarzania potokowego może się zdarzyć tak, że druga instrukcja odczytująca będzie próbowała odczytać wartość, która nie została jeszcze zapisana przez poprzednią (pierwszą) instrukcję. Aby uchronić przed takimi sytuacjami, dodawane są sprzętowo puste cykle pomiędzy budzące wątpliwości instrukcje. Minimalizacja liczby pustych cykli jest możliwa poprzez dodawanie, jeśli to możliwe, pomiędzy konfliktowe instrukcje innych.

Konflikt rejestrów polega na próbie odczytu zawartości rejestru wtedy, gdy poprzednia instrukcja zapisywała jakąś wartość to tego rejestru. Również, podobnie jak miało to miejsce w przypadku błędu zapisu-odczytu, wzrost wydajności wykonywania programu można osiągnąć przez umieszczanie innych instrukcji pomiędzy te konfliktowe.

Tryby debugowania

Mikrokontrolery czasu rzeczywistego z rdzeniem C28x zostały wyposażone w sprzętowe układy wspomagające debugowanie pracy układu w systemie. Standardowo proces odpluskwiania odbywa się przez interfejs JTAG. Dostępne są następujące tryby kontroli wykonywania programu:

- *stop mode*,
- *real-time mode*.

Pierwszy tryb, *stop mode*, jest typowym sposobem debugowania w systemie. W tym trybie rdzeń może znajdować się w trzech stanach: ciągłym (*Run state*), wykonywania pojedynczej instrukcji (*Single-istruction state*) oraz w stanie zatrzymania (*Debug-halt state*). Podgląd i ingerencja w rejestry procesora są możliwe jedynie w ostatnim wymienionym stanie zatrzymania (*Debug-halt state*). W dwóch pozostałych zawartość okna debuggera nie jest aktualizowana.

Praca rdzenia w trybie *stop mode*, gdy znajduje się on w stanie zatrzymania (*Debug-halt state*), jest zawieszana „na sztywno". Oznacza to, że nie ma możliwości obsługi przerwania. Na pierwszy rzut oka może się to wydawać mało istotne, jednak często układy mikroprocesorowe sterują elementami wykonawczymi, których działanie nie powinno być zatrzymywane nawet podczas debugowania. Z myślą o takich przypadkach rdzenie C28x mogą pracować w trybie *Real-time mode*. Najistotniejszą różnicą w stosunku do trybu *stop mode* jest możliwość obsługi krytycznych przerwań w stanie zatrzymania, dzięki czemu np. praca sterowanych silników nie musi ulegać przerwaniu.

Krzysztof Paprocki, EP
krzysztof.paprocki@ep.com.pl