Obsługa rdzenia Cortex-M4F w systemie ISIXRTOS

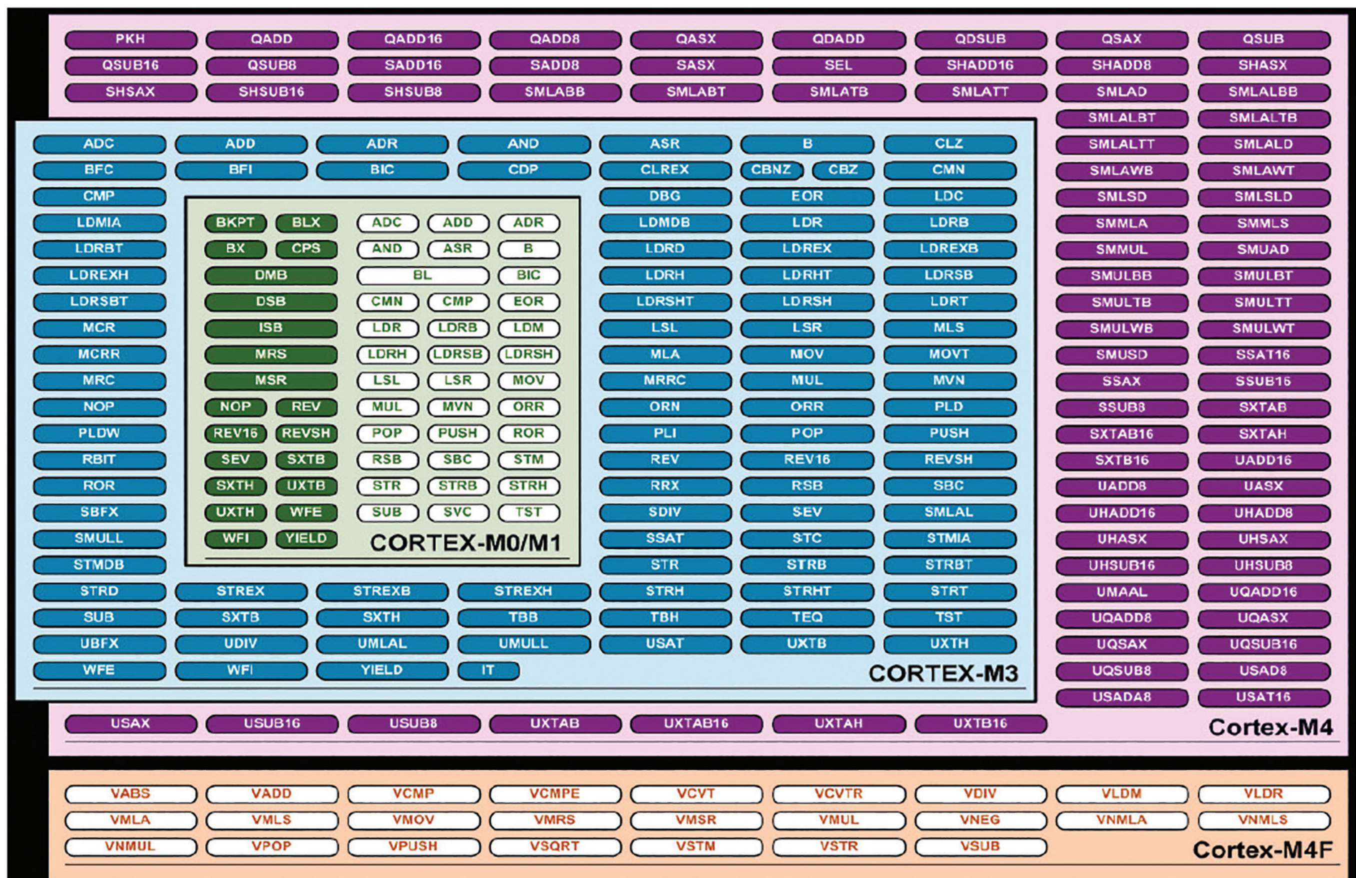
Jeśli zestaw instrukcji i moc obliczeniowa mikrokontrolera z rdzeniem Cortex-M3 jest niewystarczająca, pomocny może okazać się bliźniaczy mikrokontroler z rdzeniem Cortex-M4F, który może wykonywać dodatkowe instrukcje, pomocnych w obliczeniach DSP oraz ma jednostkę FPU pomocną przy wykonywaniu obliczeń na liczbach IEEE754 o pojedynczej precyzji. Celem niniejszego artykułu jest omówienie różnic pomiędzy rdzeniami, pokazanie doboru optymalnych nastaw kompilatora GCC oraz pokazanie różnic w implementacji przełączania zadań wynikające z obecności FPU.

Mikrokontrolery z rdzeniem ARM na dobre zadomowiły się w naszych konstrukcjach i stopniowo wypierają układy 8-bitowe. Do najbardziej popularnej rodziny zaliczają się obecnie układy z rdzeniem Cortex-M3. Charakteryzuje się on dobrym stosunkiem możliwości do ceny i pozwala na wykonywanie podstawowych instrukcji wspomagających algorytmy z pogranicza DSP. Możemy do nich zaliczyć takie operacje, jak: *REV* (odwracanie kolejności bitów przydatne przy FFT), *SMLAL* (mnożenie 32-bitowe połączone z dodawaniem 64-bitowym, przydatne przy realizacji filtrów FIR*)*, *SSAT* (lub USAT – instrukcja obcięcia z nasyceniem, przydatna przy wszelkiego rodzaju obliczeniach multimedialnych) itd. Jeśli zestaw instrukcji i moc obliczeniowa mikrokontrolerów z rdzeniem Cortex-M3 są niewystarczające, to może okazać się pomocny bliźniaczy mikrokontroler z rdzeniem Cortex-M4F, który pozwala na wykonywanie dodatkowych instrukcji, pomocnych w obliczeniach DSP oraz ma jednostkę zmiennoprzecinkową (FPU) umożliwiającą wykonywanie obliczeń na liczbach IEEE754 o pojedynczej precyzji. Pomimo iż oba rdzenie mają wspólny zestaw instrukcji, istnieją różnice wymagające odrębnego traktowania obu rdzeni, zarówno przez kompilator, jak i system RTOS.

Dodatkowy zestaw instrukcji oraz koprocesor zmiennoprzecinkowy

Rdzeń Cortex-M4 potrafi wykonywać wszystkie instrukcje rdzenia Cortex-M3, a dodatkowo ma zestaw instrukcji wspomagających przetwarzanie sygnałów i równoległe operacje na danych sygnałowych (**rysunek 1**).

Do najważniejszych instrukcji umożliwiających zwiększenie wydajności podczas realizacji zadań związanych z przetwarzaniem sygnałów jest możliwość powiązania operacji arytmetycznych z nasyceniem. Wykonując typowe obliczenia arytmetyczne, gdy wynik operacji przekroczy maksymalną



Rysunek 1. Zestaw instrukcji rdzenia Cortex-M4F

zawartość rejestru, to ulega on przewinięciu. Jeżeli np. do rejestru R0 zawierającego wartość *R0=0xFFFFFFFF* dodamy 1, to w wyniku wykonanej operacji wpisana zostanie do niego wartość 0. Jeśli zawartość rejestru reprezentuje wartość próbki sygnału, intuicyjnie zdajemy sobie sprawę, że w tym wypadku nie jest to zachowanie naturalne. W rzeczywistym układzie analogowym, jeśli wartość sygnału przekroczy zakres dynamiczny, sygnał ulega obcięciu, a nie odwróceniu. Dla rdzenia *Cortex-M3*, aby otrzymać taki rezultat, należy połączyć operacje arytmetyczne z instrukcją **SSAT** lub **USAT**, która realizuje operację obcięcia z nasyceniem. Rdzeń *Cortex-M4* zawiera dodatkowe instrukcje pozwalające na uprawnienie wykonywania podstawowych operacji na sygnałach, takich jak dodawanie (**QADD**) oraz odejmowanie (**QSUB**) dzięki połączeniu ich z nasyceniem w jednej instrukcji.

Często mamy również do czynienia z sytuacją, gdy próbki są reprezentowana przez liczby 8- lub 16 bitowe, chociażby przy próbkowaniu sygnału 16-bitowym przetwornikiem A/C. Wtedy wykorzystując odrębne instrukcje **QADD8/QADD16** czy **QSUB8/QSUB16** możemy wykonać dodawanie lub odejmowanie z nasyceniem dwóch liczb 16-bitowych lub czterech liczb 8-bitowych, równocześnie w jednej instrukcji! W podobny sposób możemy również wykonywać równoległe operacje dodawania **SADD8/SADD16** lub odejmowania **SSUB8/SSUB16** bez nasycenia.

Poza usprawnieniem operacji związanych z obliczeniami stałoprzecinkowymi, jedną z najistotniejszych cech nowego rdzenia jest wbudowany koprocesor zmiennoprzecinkowy umożliwiający wykonanie operacji na liczbach *IEEE754* o pojedynczej precyzji. Pozwala to na znaczące przyśpieszenie wykonywania operacji zmiennoprzecinkowych w porównaniu do implementacji programowej realizowanej przez kompilator. Koprocesor ma 32 rejestry *S0-S31* i umożliwia wykonywanie podstawowych operacji takich jak:

- mnożenie,
- dzielenie,
- dodawanie,
- odejmowanie,
- konwersja liczby zmiennoprzecinkowej na stałoprzecinkową,
- konwersja liczby stałoprzecinkowej na zmiennoprzecinkową,
- pierwiastek kwadratowy.

Wykorzystanie koprocesora przy pracy bez systemu operacyjnego wymaga jedynie wykonania podstawowej operacji włączenia koprocesora. Poza samą konfiguracją startową oraz ustawieniem opcji kompilatora, nie ma potrzeby wykonywania żadnych zmian w programie. Sytuacja komplikuje się przy użyciu systemu operacyjnego, ponieważ podczas przełączania zadań jest wymagane zachowanie kontekstu rejestrów koprocesora. Fragment kodu systemu odpowiedzialny za przełączanie kontekstu musi ulec zmianie i jest rożny dla obu rdzeni. Duża liczba rejestrów koprocesora jest z jednej strony zaletą, ponieważ pozwala na zmniejszenia liczby cykli dostępu do pamięci, jednak z drugiej strony znacząco zwiększa czas przyjęcia przerwania czy przełączenia zadania, ponieważ wymaga zapisania dodatkowych danych na stosie. O sposobie poradzenia sobie z tym problemem dowiemy się w dalszej części artykułu.

Ustawienia kompilatora GCC na przykładzie systemu ISIX i mikrokontrolera rodziny STM32F4

Zestaw instrukcji Cortex-M4F stanowi rozszerzenie instrukcji Cortex-M3. Jeśli koprocesor numeryczny jest wyłączony, wówczas ramka stosu jest identyczna jak dla rdzenia M3, a układ może bezpośrednio wykonywać kod dla niego przeznaczony i jest z nim w pełni kompatybilny. Tak skompilowany kod będzie działał poprawnie, jednak nie będziemy mogli w pełni skorzystać z nowych możliwości układu – kod będzie zawierał jedynie podzbiór instrukcji, a operacje zmiennoprzecinkowe będą realizowane w sposób programowy. Aby w pełni skorzystać z możliwości rdzenia, konieczne jest wsparcie ze strony kompilatora. W przypadku użycia GCC powinien być on skonfigurowany w trybie *mutilib* z włączoną obsługą wsparcia dla rdzenia Cortex-M4F. Należy zwrócić na to szczególną uwagę, ponieważ starsze lub inaczej skonfigurowane wersje kompilatorów mogą mieć tę opcję niedostępną lub wyłączoną. W systemie ISIX zaleca się korzystanie z GCC przygotowanego przez autora, którą można pobrać ze strony autora projektu, zarówno dla systemu Windows (*http://goo.gl/ZBpqp8*), jak i dla systemu ARCHLinux (*http://goo.gl/9tfPw8*).

Oprócz odpowiedniego kompilatora jest również wymagane przekazanie odpowiednich flag konfiguracyjnych. Dla systemu budowania gnu make, flagi te należy przekazać za pomocą zmiennej *CFLAGS* oraz *CXXFLAGS*. Do najważniejszej opcji należy flaga: **-mcpu=cortex-m4**, która określa typ procesora. Przekazanie tej flagi spowoduje, że kompilator – w razie możliwości – będzie mógł użyć instrukcji rozszerzonych generując bardziej zoptymalizowany kod.

Kolejne istotne optymalizacje związane są z koprocesorem:

- **mfloat-abi=hard -mfpu=fpv4-sp-d16** – Opcje te nakazują kompilatorowi użycie koprocesora zmiennoprzecinkowego oraz określają jego rodzaj, w tym przypadku jest to koprocesor o pojedynczej precyzji.
- **ffast-math** – opcja ta włącza szereg flag optymalizujących przyspieszających działanie operacji zmiennoprzecinkowych. Między innymi polega to na wyłączeniu wykonania dodatkowych sprawdzeń, czy np. zmienna nie zawiera liczby typu **NAN** lub **INF**, wyłączane jest również ustawianie zmiennej **errno** jako rezultatu wykonania operacji arytmetycznych. Ustawienia te pozwalają na przekierowanie większości operacji zmiennoprzecinkowych bezpośrednio do koprocesora, bez konieczności wywołania funkcji bibliotecznych, co prowadzi do znacznego przyśpieszenia działania, jednak sprawia, że kompilator nie zachowuje się zgodnie ze standardem *IEEE/ISO.*
- **fsingle-precision-constant** – domyślnie wszystkie stałe zmiennoprzecinkowe (zgodnie ze standardem) traktowane są jako podwójnej precyzji dopóki explicite nie zostaną zdefiniowane z suffixem f. Ustawienie tej flagi spowoduje, że domyślnie liczby zmiennoprzecinkowe bez określonego suffixu będą domyślnie zdefiniowane jako pojedynczej precyzji.

Przedostatnia opcja jest związana z generowaniem kodu przez kompilator i sprawia, że wygenerowany kod może być znacznie szybszy, jednak kosztem kompatybilności z normą ISO. Ostatnia flaga pozwala na uniknięcie przypadkowego zdefiniowania liczby podwójnej precyzji. Ma to o tyle istotne znaczenie, że zintegrowany koprocesor nie obsługuje operacji na liczbach podwójnej precyzji, więc każda operacja zmiennoprzecinkowa na liczbach typu **double** powoduje przekierowanie do biblioteki emulującej obliczenia zmiennoprzecinkowe, która będzie wykonana przez jednostkę centralną z pominięciem koprocesora.

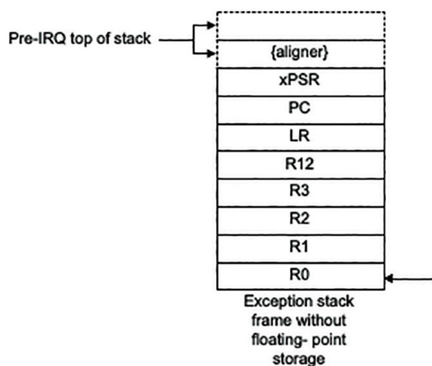
W systemie ISIX za wsparcie aplikacji dla mikrokontrolerów STM32 jest odpowiedzialna biblioteka **libstm32**, która zawiera kod specyficzny dla STM32 wraz ze skryptami służącymi do budowania aplikacji oraz całego systemu. Wszystkie wspomniane wcześniej flagi dobierane są w sposób automatyczny przez odpowiednio przygotowany plik *make* i użytkownik nie musi się martwić o wpisywanie wszystkich opcji. Aby skompilować kod na platformę Cortex-M4 wystarczy zdefiniować odpowiedni typ procesora STM32 w pliku *Makefile* w zmiennej **MCU_MAJOR_TYPE = f4**. Wtedy wszystkie flagi zostaną dobrane automatyczne. Biblioteka może być również używana zupełnie niezależnie od systemu operacyjnego i może istnieć jako samodzielny byt, niezależny od systemu ISIX. Wybranie mikrokontrolera STM32 z wbudowanym koprocesorem spowoduje również automatyczne włączenie koprocesora numerycznego w pliku startowym *crt0.c:*

```
//Enable FPU if present
#if defined(STM32MCU_MAJOR_TYPE_
F4)     && (__FPU_PRESENT == 1)
&& (__FPU_USED == 1)
SCB->CPACR |= ((3UL << 10*2)|(3UL
<< 11*2));  //set CP10 and CP11
Full Access
#endif
```

Rdzeń Cortex-M4F a przełączanie zadań w systemie ISIXRTOS

Obecność koprocesora zmiennoprzecinkowego przy zmianie kontekstu wymusza – oprócz zapisania stanu jednostki centralnej – zapisanie zawartości jego rejestrów. Przy pracy mikrokontrolera bez systemu operacyjnego taka sytuacja ma miejsce jedynie podczas wykonywania kodu obsługi przerwania, gdy jednak mamy do czynienia z systemem operacyjnym, zmiana kontekstu następuje również w momencie przełączania zadań. Z uwagi na to że liczba rejestrów do zapisania jest duża, zapisanie stanu jednostki zmiennoprzecinkowej powoduje zwiększenie rozmiaru stosu, zwiększa czas obsługi przerwania, a w przypadku systemów operacyjnych zwiększa również czas potrzebny do przełączania zadań. Z tych właśnie powodów rdzeń Cortex-M4 zawiera mechanizmy pozwalające na zapisywanie stanu koprocesora tylko i wyłącznie wtedy, gdy jest to konieczne. W związku z tym istnieją dwie ramki stosu, które są tworzone przez rdzeń w momencie wystąpienia przerwania. Jeśli bieżący kontekst nie zawiera operacji zmiennoprzecinkowych, ramka ta jest identyczna, jak w przypadku rdzenia M3 i wygląda jak na **rysunku 2**. Gdy kontekst zawiera operacje zmiennoprzecinkowe ramka zawiera dodatkowo automatycznie zapisane młodsze 16 rejestrów koprocesora S0-S15, co pokazano na **rysunku 3**, natomiast pozostałe rejestry *S16-S31* powinny zostać zapisane przez kompilator w razie potrzeby za pomocą instrukcji **vpush** a następnie przywrócone za pomocą instrukcji **vpop**.

Ponieważ istnieją dwa formaty ramki stosu musi istnieć sposób na to, aby rozróżnić rodzaj aktualnego kontekstu, co odbywa się poprzez sprawdzenie 4-go bitu kodu **EXC_RETURN** znajdującego się w rejestrze LR w momencie przyjęcia przerwania. Jeśli w momencie zgłoszenia przerwania kontekst zmiennoprzecinkowy był aktywny w procesie głównym, wówczas ten bit jest wyzerowany. Jeśli kontekst zmiennoprzecinkowy jest nieaktywny, wówczas ten bit jest ustawiony, a kod EXC_RETURN wygląda identycznie, jak dla rdzenia M3.

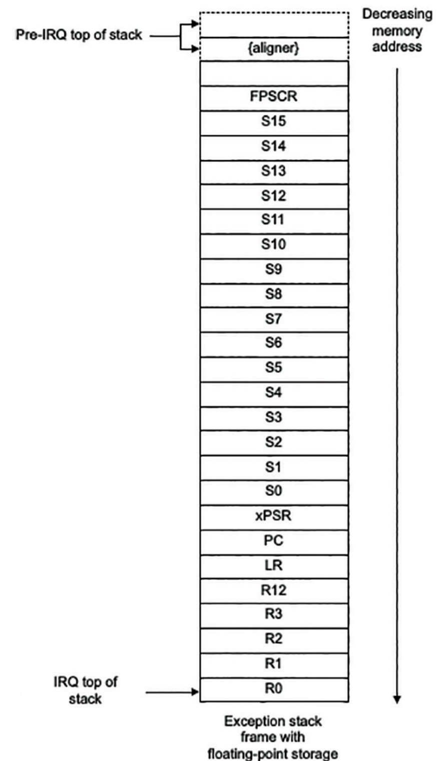


Rysunek 2. Ramka stosu Cortex-M3 i Cortex-M4 bez operacji zmiennoprzecinkowych

Pozostaje jeszcze kwestia, w jaki sposób jest określane czy kontekst powinien zawierać ramkę stosu operacji zmiennoprzecinkowych, czy też nie, co również może przyczynić się do dalszej optymalizacji programu. Dla rdzenia M4 konieczność zapisania stanu koprocesora jest wykrywana automatycznie. Odbywa się to w taki sposób, że w momencie napotkania pierwszej operacji zmiennoprzecinkowej jest ustawiany bit FPCA w rejestrze CONTROL. Oznacza to, że dany kontekst wykorzystuje operacje zmiennoprzecinkowe. Jeśli procesor zmieni kontekst poprzez wystąpienie przerwania, bit FPCA po zakończeniu zapisywania ramki stosu jest zerowany. Jeśli podczas wykonania procedury obsługi przerwania zostanie wykryta instrukcja zmiennoprzecinkowa, wówczas zamiast bitu FPCA w rejestrze CONTROL zostanie wyzerowany bit **LSPACT** (domyślnie ustawiony) w rejestrze **FPCCR**. Bazując na zawartości kodu **EXC_RETURN** oraz stanie bitu **LSPACT**, rdzeń może dokonać szeregu optymalizacji podczas zapisywania oraz odtwarzania ramki stosu nazywanych *„Lazy stacking”*. Możemy tutaj wyróżnić trzy sytuacje:

1. Jeśli kontekst główny nie wykonuje operacji zmiennoprzecinkowych oraz procedura obsługi przerwania również nie korzysta z tych operacji, ramka stosu jest identyczna jak dla rdzenia M3.
2. Jeśli kontekst główny wykonuje operację zmiennoprzecinkowe, a procedura obsługi przerwania ich nie używa (*LSPACT=1, EXC_RETURN[4]=0*), wówczas następuje optymalizacja polegająca na tym, że na stosie zostanie zarezerwowana pamięć na ramkę stosu według rys. 2, ale stan rejestrów S0-S16 nie zostanie fizycznie zapisany, co pozwala na skrócenie czas przyjęcia przerwania.
3. Jeśli kontekst główny wykonuje operacje zmiennoprzecinkowe oraz procedura obsługi przerwania również używa koprocesora (*LSPACT=0 EXC_RETURN[4]=0*), wówczas jest wykonywany pełny proces rezerwacji stosu oraz zapisania stanu rejestrów S0-S16.

Dzięki temu mechanizmowi istnieje możliwość znacznego skrócenia czasu do przyjęcia przerwania w zależności od okoliczności. Istnieje również możliwość wyłączenia tego mechanizmu poprzez zapis odpowiedniej konfiguracji w rejestrze **FPCCR.**



Rysunek 3. Ramka stosu Cortex-M4 z młodszymi 16 rejestrami FPU

Do dyspozycji mamy albo możliwość całkowitego wyłączenia zapisywania stanu koprocesora, albo zapisywania go zawsze po wystąpieniu przerwania, niezależnie od okoliczności. Niemniej jednak, nie są to rozwiązania optymalne, a domyślna konfiguracja jest jak najbardziej sensowna, dlatego w dalszej części rozważań będzie ona pominięta. Jeśli pracujemy bez systemu operacyjnego, obecność koprocesora nie wymaga specjalnego traktowania i zmian w oprogramowaniu. Sytuacja wygląda inaczej dla projektanta systemu podczas przenoszenia z platformy M3 na M4F i wymaga uwzględnienia dodatkowej ramki koprocesora oraz mechanizmu *„Lazy Stacking”*. Dla użytkownika systemu zastosowanie rdzenia M4F jest transparentne.

W systemie *ISIXRTOS* wystarczy w pliku *Makefile* zdefiniować typ procesora na **MCU_MAJOR_TYPE = f4** , co spowoduje ustawienie odpowiednich flag kompilatora, oraz wybranie kodu przełączania kontekstu uwzględniającego ramkę stosu rejestrów zmiennoprzecinkowych. Implementacja przełączania zadań w *ISIX-ie* uwzględnia pełne zapisywanie stanu koprocesora i została przygotowana w taki sposób, aby obliczenia zmiennoprzecinkowe mogły być wykonywane zarówno przez procedury obsługi przerwań oraz przez dowolne zadania systemu. Działanie mechanizmu zapisywania stanu koprocesora w systemie ISIX działa w taki sposób, że w kontekście danego zadania dodatkowo jest zapisywany kod **EXEC_RETURN** zawierający informację o tym czy w danym zadaniu aktualnie wykorzystywany jest koprocesor. Odpowiedni

kod **EXEC_RETURN** jest ustawiany w momencie, gdy zadanie wykonuje operacje zmiennoprzecinkowe za pomocą mechanizmu *„Lazy Stacking”*. Procedura obsługi przełączania zadań realizowana przez wektor *svc_pend*, która sprawdza zapamiętany kod **EXEC_RETURN** i w przypadku wykrycia ramki rozszerzonej dodatkowo zapamiętuje i odtwarza zawartość rejestrów *s16-s32*, które nie są zapamiętywane automatyczne. Obsługa kontekstów przerwań działa w sposób automatyczny i w razie potrzeby zapamiętuje lub odtwarza zawartość rejestrów koprocesora tak jak to zostało opisane na początku rozdziału.

Przykład praktyczny – weryfikacja poprawności działania

Aby zweryfikować poprawność działania przełączania kontekstu zmiennoprzecinkowego i zademonstrować działanie systemu dla *Cortex-M4F*, przygotowano prostą aplikację. Jako platformę testową wykorzystano zestaw uruchomieniowy *ZL41ARM_F4*. Do komunikacji z użytkownikiem użyto portu szeregowego, którego linię danych **TX** stanowi **PD5**. Aby móc zaobserwować rezultat działania aplikacji, należy port szeregowy dołączyć za pomocą konwertera napięć do komputera i uruchomić program terminalowy.

Działanie aplikacji testowej polega na:

- Utworzeniu 6 zadań o takim samym priorytecie, w których po uruchomieniu danego zadania do wszystkich rejestrów koprocesora (s0-s31) zapisywana jest unikalna wartość. Po wpisaniu wartości następuje w pętli nieskończonej cyklicznie sprawdzanie zawartości rejestrów koprocesora czy zawierają takie same wartości, które zostały wpisane na początku. Jeśli zawartość rejestrów koprocesora zostanie zamieniona, wówczas zadanie jest kończone z komunikatem o błędzie. Ponieważ wszystkie zadania cały czas są aktywne i mają ten sam priorytet, następuje cykliczne przełączanie zadań. Jeśli kontekst zmiennoprzecinkowy byłby niepoprawnie zachowany, wówczas kod odpowiedzialny za sprawdzenie zawartości rejestrów spowoduje zwrócenie błędu i przerwanie działania aplikacji.
- Utworzeniu jednego zadania, które wykonuje proste aplikacje arytmetyczne i wyświetla wynik na konsoli szeregowej co 1 sekundę.
- Utworzeniu cyklicznego przerwania od układu czasowo licznikowego T3, w którego procedurze obsługi jest wywoływana funkcja wypełniająca zawartość rejestrów koprocesora zadaną wartością. Równocześnie w procedurze obsługi przerwania jest wywoływane przerwanie zagnieżdżone, które powoduje przerwanie wykonania aktualnej procedury. Dodatkowe przerwanie zagnieżdżone zapisuje wszystkie rejestry koprocesor nową wartością i kończy działanie. Po zakończeniu obsługi zagnieżdżonego przerwania następuje powrót to pierwotnego przerwania, w którym jest sprawdzana zawartość rejestrów koprocesora z poprzednio wpisaną wartością. Jeśli wartość jest różna od pierwotnie wpisanej oznacza to, że kontekst koprocesora został źle zachowany. Równocześnie wykonanie kodu przerwań od układu czasowo licznikowego jest testem zachowania kontekstu rejestrów koprocesora dla zadań/wątków systemu.

Wykonanie podstawowych testów pozwala upewnić się odnośnie do poprawnego działania przełączania kontekstu koprocesora numerycznego. Główny kod aplikacji testowej jest zawarty w pliku *fpuctxtest.cpp*, natomiast kod aplikacji odpowiedzialnej za sprawdzenie zawartości oraz wypełnianie rejestrów zmiennoprzecinkowych został napisany w assemblerze i znajduje się w pliku *fputest.S*

Aby skompilować przykład należy przejść do katalogu **stm32f4/fpuctx** a następnie wydać polecenie *make*. Po skompilowaniu należy zaprogramować zestaw *ZL41ARM_F4* za pomocą interfejsu JTAG poprzez wydanie polecenia *make program*.

Po zaprogramowaniu układu program testowy rozpocznie działanie a na konsoli terminalowej powinna być wyświetlana wartość zmiennej float zwiększanej cyklicznie w pętli nieskończonej jednego zadania. W tym samym czasie 6 zadań testowych oraz wspomniane przerwanie od układu czasowo licznikowego również działają i weryfikują poprawność przełączania kontekstu. Celem emulowania błędu kontekstu koprocesora możemy włączyć skomentowaną linię **FPU->FPCCR &= ~((1<<31U)|(1<<30));** znajdującą się w pliku *fpuctxtest.cpp*, a następnie skompilować przykład i zaprogramować mikrokontroler. Wspomniana linia powoduje wyłączenie zapisywania kontekstu koprocesora. Po zaprogramowaniu skutkować to powinno wystąpieniem natychmiastowego komunikatu o błędzie stanu rejestrów koprocesora w poszczególnych zdaniach.

Zakończenie

Nowoczesny rdzeń Cortex-M4F pozwala na wykonywanie zaawansowanych operacji, które jeszcze do niedawna zarezerwowane były dla mikrokontrolerów sygnałowych oraz bardziej zaawansowanych mikroprocesorów. Dzięki wsparciu systemu operacyjnego oraz kompilatora możemy w łatwy sposób przygotować aplikacje, które wymagają stosunkowo dużej mocy obliczeniowej.

Lucjan Bryndza, EP

REKLAMA