ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1993

Seria: INFORMATYKA z. 22 Nr kol. 1195

Dariusz ZONENBERG
Instytut Informatyki Teoretycznej i Stosowanej
Polskiej Akademii Nauk

EMULATOR UKŁADOWY – OBSERWACJA UZALEŻNIEŃ CZASOWYCH OPROGRAMOWANIA SYSTEMU MIKROPROCESOROWEGO

Streszczenie. W opracowaniu przedstawiono metody uruchamiania systemów mikroprocesorowych pod kątem rejestrowanych za pomocą emulatora układowego uzależnień czasowych. Dokonany został przegląd metod zliczania czasu w emulatorze oraz metod rejestracji zdarzeń w uruchamianym systemie. Następnie za pomocą umownego języka relacji, wzorowanego na notacji Pascala, dokonano przeglądu podstawowych filtrów zdarzeń na magistrali oraz ich semantyki.

IN-CIRCUIT EMULATOR – OBSERVATION OF TIMING DEPENDENCIES OF MICROPROCESSOR SYSTEM SOFTWARE

Summary. The article presents some debugging techniques of microprocessor system software using timing characteristics obtained with an ICE (In-Circuit Emulator). An overview of time measurement and event registration methods employed by the ICE is also presented. The paper ends in an overview of basic event filters and their semantics, using an example *Pascal*-based language of relations.

EMULATEUR MATERIEL – OBSERVATION DES INTERDEPENDENCES TEMPORELS DANS LOGICIEL DE SYSTEME MICROPROCESSEUR

Resumé. L'article présente les méthodes de mettre au point les systemès microprocesseur compte tenu les interdépendences temporels enregistrées par un emulateur matériel. Nous examinons les méthodes du comptage du temps par emulateur aussi que les méthodes d'enregistrement des événements dans les systèmes étant mis au point. Dans la suite un langage des rélations ayant notation pareille au *Pascal*, est utilisé pour examiner les principaux filtres des événements et leur sémantique.

W trakcie konstruowania i uruchamiania systemu mikroprocesorowego projektant posługuje się różnymi środkami do wykrywania i analizy błędów popełnionych zarówno w fazie konstruowania i montażu układów elektronicznych, jak i projektowania oprogramowania. Obszerna klasa błędów zarówno sprzętowych, jak i programowych dotyczy niewłaściwych czasów realizacji określonych funkcji zaprojektowanego systemu. Poniższe opracowanie jest próbą uporządkowania zagadnień związanych z analizą czasów operacji w systemie mikroprocesorowym w trakcie jego obserwacji za pomocą jednego z bardziej efektywnych środków uruchomieniowych, jakim jest emulator układowy mikroprocesorów.

1. Uzależnienia czasowe mikroprocesora

Z każdą funkcją realizowaną przez mikroprocesor związany jest moment jej rozpoczęcia oraz czas jej wykonania. Mikroprocesor jest taktowany zegarem o określonym cyklu, który dla poszczególnych procesorów jest podstawą do wyliczenia dokładnych lub minimalnych (jeśli dostęp do pamięci i portów jest realizowany na podstawie detekcji sygnału WAIT) czasów pobrania i wykonania rozkazu mikroprocesora (instrukcji). Jest on również podstawą do analogicznych szacowań czasu dostępów do zasobów zewnętrznych procesora, jak pamięci i portów. W przypadku dostępu do zasobów zewnętrznych mikroprocesor łączy swoje poszczególne cykle w dłuższe sekwencje, zwane cyklami maszynowymi lub cyklami magistrali, w których realizuje dostęp do określonego zasobu. Cykle maszynowe z kolei składają się na realizację instrukcji mikroprocesora, działającego wg modelu von Neumana, czyli realizującego kolejno:

- pobranie rozkazu (w cyklach pobrania kodu),
- dekodowanie rozkazu (w cyklach procesora),
- realizację rozkazu (w cyklach wykonawczych procesora oraz cyklach wykonawczych magistrali, jeśli rozkaz dotyczył dostępu do zasobów zewnętrznych procesora).

Podstawowym zasobem logicznym, do którego procesor ma dostęp, jest pamięć kodu programu, zawierająca rozkazy dla procesora. Czas wykonania jednej instrukcji wynika z czasu potrzebnego na jej pobranie oraz czasu potrzebnego na jej zdekodowanie i realizację. Zatem czas wykonania programu może być sumą czasów pobrania, dekodowania i realizacji poszczególnych jego rozkazów. Jeśli procesor może wykonywać pobranie, dekodowanie i wykonanie rozkazów równolegle, np. na skutek wewnętrznego kolejkowania wyników poszczególych faz całej realizacji rozkazu, to zakładając przykładowo, że są trzy fazy realizacji rozkazu o następujących czasach trwania:

T_p — czas potrzebny na pobranie rozkazu,

T_d — czas potrzebny na dekodowanie rozkazu,

T_w — czas potrzebny na wykonanie rozkazu

oraz oznaczając całkowity czas wykonania przez T_c, możemy go oszacować jak poniżej:

$$max(T_p, T_d, T_w) \leq T_c \leq (T_p + T_d + T_w)$$

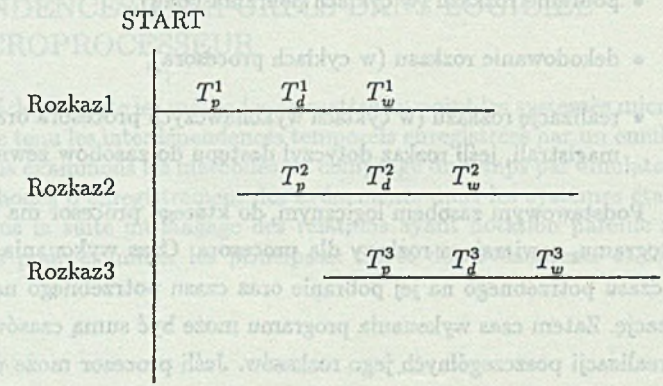
Przykładowo, w procesorze 80386 istnieje wewnętrzne kolejkowanie rozkazów już pobranych oraz rozkazów już zdekodowanych, zatem mikroprocesor może wykonywać równocześnie 3 operacje:

a) pobranie kodu lub danych z magistrali,

b) pobranie do dekodowania rozkazu z kolejki rozkazów i jego zdekodowanie,

c) pobranie rozkazu z kolejki rozkazów zdekodowanych oraz jego wykonanie.

W przypadku realizacji trzech kolejnych rozkazów będzie to miało następującą postać czasową:



zatem dla ciągłej pracy mikroprocesora, gdy kolejne rozkazy są bez przerwy pobierane, dekodowane i wykonywane, szacowany średni czas realizacji dowolnego z nich nie będzie mniejszy od trwania jego najdłuższej fazy realizacji, ale nie przekroczy sumarycznej długości trwania jego wszystkich faz. Podkreślenie, że szacowany czas jest średni, wynika z faktu, że czas realizacji każdego rozkazu jest zależny od fazy T_w rozkazu poprzedniego i T_p rozkazu następnego. Można ocenić, ile te czasy średnio wynoszą, natomiast ich dokładne szacowanie jest utrudnione poprzez występowanie wielu czynników, jak operacje czyszczenia kolejek wewnętrznych procesora, przepełnianie tych kolejek, czy też nie w pełni udokumentowana architektura wnętrza mikroprocesora, które często uniemożliwiają przeprowadzenie dokładnej analizy zachowań czasowych procesora.

Jeśli możliwa jest realizacja rozkazu w większej ilości równoległych faz o czasach $T_1, T_2, .., T_n$, to powyższa nierówność powinna być uzupełniona o n elementów:

$$max(T_1, T_2, \ldots, T_n) \leq T_c \leq (T_1 + T_2 + \ldots + T_n)$$

Dla mikroprocesorów realizujących równolegle kilka rozkazów w oszacowaniu należy uwzględnić fakt współbieżnej realizacji K rozkazów, każdy w n współbieżnych fazach. Zatem będzie ono coraz trudniejsze i coraz mniej dokładne, tak że tylko fizyczny pomiar czasu realizacji programu pozwoli wyznaczyć pożądaną wartość rzeczywistego czasu wykonania danych procedur.

Dla oprogramowania kontrolującego określone otoczenie fizyczne znaczący będzie zarówno czas realizacji poszczególnych funkcji, jak i czas reakcji na wydarzenia. Zasoby zewnętrzne procesora, do których procesor ma dostęp w cyklach wykonawczych, to pamięć danych i porty. Istotny dla nich jest zarówno czas trwania dostępu do danego zasobu, jak i moment, w którym dany dostęp miał miejsce. Równie ważna bywa częstość dostępów do zasobu lub kolejność dostępów.

Uogólniając, można powiedzieć, że w trakcie pracy mikroprocesora na jego magistrali pojawiają się cykle maszynowe, związane z pobieraniem kodu do wykonania i jego realizacją. Są to głównie cykle odczytu kodu i dostępu do danych, ale mogą pojawić się i inne, związane np. z pracą kanału DMA, obsługą przerwań, komunikacją z koprocesorami lub innymi procesorami w architekturze wieloprocesorowej. Każdy z zaobserwowanych cykli trwa przez określony czas, a sumaryczna długość ich trwania, odstępy między nimi oraz wartości z nimi związane, jak adres, dana i bity statusu, stanowią specyficzny ślad sposobu wykonania programu. Niezależnie od możliwości analizy semantycznej wartości pojawiających się na magistrali, opierając się na ich pokrewieństwie (tzn. przynależności adresów, danych lub typów cykli do pewnego zbioru wartości, określających jednoznacznie wystąpienie wybranego zdarzenia), można wyciągnąć daleko idące wnioski dotyczące czasu wykonania określonych funkcji, częstości zdarzeń itp. Przy użyciu notacji języka *Pascal*, wg założeń przyjętych w pozycji [4], ogólna postać stanu magistrali, będącego podstawą do analizy, może być przedstawiona za pomocą rekordu o poniższej strukturze:

```
type Bus = record
          Address   : integer;
                    (* Adres wystawiony w danym cyklu *)
          Data      : integer;
                    (* Dana wystawiana w danym cyklu *)
          Status    : array [0..N-1] of Bit;
                    (* N-bitowy status cyklu *)
          Time      : integer;
                    (* Moment wystawienia cyklu *)
end;

var BusState    : Bus;
```

Zakłada się, że w strukturze zawarto pełną, rozwiniętą do rozmiaru maszynowej reprezentacji standardowego typu integer informację o danym cyklu, a `Address`, `Data` i `Time` mają swoje reprezentacje maszynowe w podzbiorze reprezentacji standardowego typu `integer`. Struktura `Bus` jest więc złożeniem statusu magistrali i zanotowanego momentu jego wystąpienia, co można przedstawić w następujący sposób:

```
Bus ::= <stan-magistrali> <czas-rejestracji-stanu>
```

W trakcie konstrukcji sprzętu i oprogramowania systemów mikroprocesorowych, oprócz typowych błędów natury logicznej, wynikających ze złych połączeń lub rozwiązań

elektroniki, albo niepoprawnej struktury lub realizacji oprogramowania, napotykamy błędy związane z nieprawidłowymi czasami pracy systemu. W trakcie konstrukcji i uruchamiania sprzętu elektronik zwraca jeszcze uwagę na kwestie związane z czasami reakcji układów, niezależnie od uruchamianej logiki. Na etapie konstrukcji oprogramowania uwaga programisty skupia się już głównie na logice programu — wynika to z następujących przyczyn:

a) systemy operacyjne, w których środowisku jest tworzone i uruchamiane oprogramowanie, często zupełnie izolują programistę od parametrów czasowych poprzez wirtualizację komputera — programista pisząc i uruchamiając oprogramowanie orientuje się w dynamice i szybkości reakcji konsoli, natomiast gospodarka procesorem, leżąc zwykle w gestii systemu operacyjnego, jest niewidoczna i trudna do oszacowania przez użytkownika systemu,

b) typowe jest matematyczne wykształcenie inżynierów programistów, co wiąże się u nich z brakiem nawyków analizy czasowej — z kolei osoby o wykształceniu nieinformatycznym (elektronicy lub użytkownicy końcowi) mogą mieć odpowiednie nawyki, ale brak im przygotowania do prowadzenia całości tego typu projektów,

c) struktura oprogramowania jest najczęściej sekwencyjna.

W przypadku projektowania systemów mikroprocesorowych, pracujących w czasie rzeczywistym, konstruktor oprogramowania napotyka zupełnie inną sytuację:

a) docelowy system komputerowy posiada wyspecyfikowane czasy reakcji,

b) algorytmy obsługują sytuacje o wyspecyfikowanych parametrach czasowych (kiedy występują, jak często, jak długo trwają pewne istotne zdarzenia),

c) logika oprogramowania jest najczęściej współbieżna (wielozadaniowa).

Wielozadaniowość tego typu systemów jest związana z dużą ilością zdarzeń w otoczeniu. Algorytm jednobieżny, odpytujący wszystkie urządzenia fizyczne w każdej sytuacji jest najczęściej zbyt skomplikowany, aby można było mówić o jego niezawodności oraz łatwości jego przyszłej pielęgnacji. O wiele prostsza jest metoda dekompozycji problemu na szereg prostych, łatwych do uruchomienia i pielęgnacji zadań przy użyciu mechanizmów monitora wielozadaniowego lub całego systemu operacyjnego czasu rzeczywistego. Niestety, zarówno kwestia pracy w czasie rzeczywistym, jak i wielozadaniowość mogą być źródłem nowych kategorii błędów. Mogą one wynikać z niespełnienia takich założonych wymogów systemu, jak:

a) założony czas reakcji oprogramowania na zdarzenia (realizacja procedury w określonym czasie po wystąpieniu zdarzenia),

b) założony czas obliczeń (zakończenie realizacji procedury w określonym czasie),

c) kolejność obliczeń (kończenie różnych zadań równoległych w pożądanej kolejności),

d) inne szczegółowe kwestie związane z obsługą zgłoszeń zewnętrznych (np. obsługa periodycznych zdarzeń, kanały DMA oraz zakładane ich czasy pracy itp.)

Należy zauważyć, że powyższe kwestie dotyczą zarówno systemów wielozadaniowych, jak i wieloprocesorowych. Poprawnie napisane oprogramowanie, poprzez użycie odpowiednich elementów jądra systemu wielozadaniowego, sprosta tym wymogom, ale potencjalne powstałe błędy oprogramowania wymagają szczególnych metod ich wykrywania. Tradycyjne podejście, z użyciem typowych właściwości debuggera (np. wg [8]), nie dostarczy niezbędnych informacji o przyczynie błędów. Środki bardziej przydatne do tego celu zostaną zaprezentowne w następnych rozdziałach.

2. Sposób wyznaczania czasu w emulowanym prototypie

Wyznaczanie czasu dla bieżącej chwili w emulowanym systemie prototypowym polega na określeniu ilości równomiernych odcinków czasu, jakie upłynęły od arbitralnie przyjętego momentu przed rozpoczęciem emulacji prototypu do momentu pomiaru. Zliczanie odcinków czasu najprościej realizować za pomocą układu zliczającego impulsy generowane przez równomiernie występujące zdarzenia. Wyjściowa zawartość licznika winna być równolegle zapisywana w pole `BusState.Time`. Narzucają się dwa podstawowe źródła zdarzeń, na podstawie których można by generować zliczane impulsy:

- kolejne cykle magistrali,
- zegar czasu rzeczywistego w emulatorze.

Szczegółowe ich wykorzystanie może być następujące:

1) **Kolejne cykle magistrali.** Dla pewnej grupy mikroprocesorów czas każdego cyklu jest stały, zależny od częstotliwości kwarcu oscylatora mikroprocesora. Jednak większość procesorów dopuszcza możliwość oczekiwania na gotowość pamięci lub portów przez różny czas, będący wielokrotnością taktu zegara (oscylatora), zależny od

statusu wystawianego przez pamięć lub port. Powoduje to, że zliczona ilość cykli daje tylko orientacyjny czas wykonania programu, gdyż poszczególne, zliczone cykle mogły mieć różną długość. Metoda ta daje ścisły pomiar czasu tylko w nielicznych przypadkach (głównie rodzina procesorów 8051).

2) **Zegar czasu rzeczywistego w emulatorze.** Okres jego taktu może być oparty na okresie taktu podstawowego oscylatora mikroprocesora. Pozwoli to dokładnie wyznaczać zarówno czasy realizacji poszczególnych funkcji oprogramowania, jak i odstępy między interesującymi operatora zdarzeniami. Typowy cykl magistrali znanych procesorów jest zwykle realizowany w czasie trwającym od 3 do 6 taktów oscylatora (zależnie od procesora i operacji), nie licząc taktów WAIT wydłużających go (jeśli dany mikroprocesor dopuszcza możliwość wstawiania w cykl magistrali taktów oczekiwania). Do pomiaru długości cykli okres zegara powinien być więc co najwyżej tak długi jak okres oscylatora, natomiast do pomiaru czasów realizacji całych sekwencji rozkazów można dopuścić wolniejszy zegar, pamiętając jednak, że okres jego drgań jest wartością określającą dokładność pomiaru czasu.

Do pomiaru czasu w uruchamianym prototypie, jak to zostało podkreślone, niezbędny jest zorientowany na tę funkcję układ, zapewniający kolejno:

a) generację impulsów odmierzających upływ czasu,

b) zliczanie impulsów,

c) zapis wartości bieżących czasu do BusState.Time.

Metoda 1) wymaga zliczania kolejnych cykli magistrali, a metoda 2) realizacji zegara czasu rzeczywistego w konstrukcji emulatora lub zliczania kolejnych taktów oscylatora procesora.

3. Sposoby zapamiętywania zdarzeń pojawiających się na magistrali

Wartości sygnałów na magistrali, związane z poszczególnymi momentami czasowymi, muszą zostać w określony sposób zapamiętane, aby mogły ulec późniejszej obróbce przez program sterujący emulacją. Nasuwają się dwie podstawowe metody rejestracji stanów magistrali:

- pamięć śladu operacji magistrali,
- układy szybkich specjalizowanych rejestrów.

Szczegółowe ich wykorzystanie może być następujące:

1) **Pamięć śladu operacji magistrali.** Za [4] można założyć, że poszczególne cykle są sprzętowo zapamiętywane sekwencyjnie lub według określonego klucza w tablicy stowarzyszonej z rekordem Bus zdefiniowanej następująco:

```
Trace_memory   : array [0..K-1] of Bus;
```

gdzie K oznacza maksymalną ilość pamiętanych cykli. Pole `Trace_memory[i].Time` jest zapisywane automatycznie, wraz z resztą informacji pobieranej z magistrali, wprost z licznika upływu czasu. Zapis selektywny pamięci śladu jest możliwy poprzez użycie, typowych dla większości konstrukcji emulatorów, układów komparacji zawartości magistrali z zadanym warunkiem. Wynik porównania jest traktowany jako kwalifikator zapisu do `Trace_memory` poszczególnych cykli. Zaletą tego rozwiązania jest jego zgodność z typową konstrukcją emulatora. Angażuje ono jednak układy, które operator może zechcieć używać do innych celów, a sam dostęp do tablicy śladu i jej przetworzenie mogą w niektórych sytuacjach być zbyt czasochłonne (np. ze względu na jej duże rozmiary, konieczność jej rozpakowania itp.).

2) **Układy szybkich specjalizowanych rejestrów** zintegrowanych z użytkowaną pamięcią emulowaną lub magistralą, działających jako pamięć asocjacyjna, tzn. rejestry zawierające pole robocze, pamiętające daną zawartość magistrali wg określonego kryterium oraz pole klucza, wskazujące kryterium zapisu. Ogólna struktura rejestru powinna być następująca:

```
type ARregister = record
     State : Bus;          (* Zawartosc magistrali wraz z
                              czasem jej wystapienia      *)
     Key   : Cond;         (* Zadany warunek zapisu pola
                              State                       *)
end;
```

gdzie typ danej `Cond` został zadeklarowany następująco:

```
type Cond =     record
                    (* Szczegolna zawartosc deklaracji warunku
                       bedaca najczesciej odbiciem struktury
                       sprzetowej modulu testujacego.        *)
                end;
```

Cechy konstrukcyjne rejestrów powinny zezwalać na ich odczyt przez oprogramowanie emulatora bez wstrzymywania programu prototypu, zatem powinny być to bardzo szybkie układy, co przy ich małych rozmiarach nie powinno nadmiernie zwiększać kosztu całego emulatora (tzn. zakładana mała ilość rejestrów specjalizowanych implikuje małą ilość użytych drogich i szybkich elementów). Celem takiej konstrukcji powinno być sygnalizowanie określonych momentów czasowych, wg zadanego klucza, bez angażowania innych uniwersalnych, ale wolniejszych modułów emulatora, jak pamięć śladu i układy komparacji. Kosztem takiego rozwiązania są dodatkowe szybkie i często bardzo drogie moduły sprzętowe, jednak wybór specjalizowanego układu często ułatwi automatyzację pomiarów upływu czasu w emulowanym systemie prototypowym.

Oba proponowane rozwiązania powinny posiadać następujące cechy:

a) możliwość zapamiętania wybranych cykli, wraz z czasem ich wystąpienia, bez wstrzymywania pracy procesora,

b) możliwość deklaracji selektywnego zapisu cykli do układów pamiętających.

Abstrahując od konkretnej realizacji sprzętowej załóżmy, że w tak zdefiniowanym funkcjonalnie środowisku sprzętowym możliwa jest deklaracja selektywnej rejestracji stanu magistrali, wraz z czasem jego wystąpienia, do układów pamiętających poprzez następująco zdefniowaną procedurę:

```
procedure SetTrace(Condition: Cond);
```

gdzie typ danej Cond jest ogólnym zdefniowanym już powyżej rekordem zawierającym warunek dla komparacji zawartości magistrali z zadanym warunkiem, zdefiniowanym obecnie w celu zwiększenia czytelności deklaracji jako łańcucha znaków, opisującego relację za pomocą umownych zmiennych STATUS, DATA, ADDRESS (wg [5]), przykładowo:

```
"ADDRESS > 500",
"DATA=$80 and ADDRESS<$30 and STATUS=IFE".
```

Ścisła syntaktyka warunku wymaga oczywiście formalnego zapisu, semantyka będzie ograniczana cechami konstrukcyjnymi emulatora, a w ogólnym przypadku bardziej użyteczny może się okazać zapis warunku jako pól rekordu zgodnego ze strukturą układów komparujących niż jako łańcucha znaków (np. tabela do wypełnienia w trybie interaktywnym).

Od momentu użycia powyższej procedury realizowana jest rejestracja zadeklarowanych wydarzeń.

Załóżmy następnie, że zdefiniowano funkcję pobierającą zarejestrowane wydarzenia, wraz z czasem ich wystąpienia, z układów śledzących. Postać nagłówka funkcji może być następująca:

```
function GetTrace : integer;
```

Niechaj funkcja poprzez swą nazwę przekazuje ilość pobranych cykli. Dla uproszczenia załóżmy, że zarejestrowane zdarzenia są przepisywane w jedno ustalone miejsce, a mianowicie tablicę K–elementową zdefiniowaną w następujący sposób:

```
var Timings : array [0..K-1] of Bus;
```

Po przyjęciu powyższych definicji przykładowa sekwencja programowa zbierająca, przetwarzająca i wizualizująca zarejestrowane wydarzenia może mieć następującą postać.

```
var  i: integer;

Condition := (* zalezna od przyjetej definicji typu Cond
                deklaracja cykli, ktorych wystapienie winno
                podlegac detekcji *)

SetTrace(Condition);

(* Zależne od konstrukcji emulatora oraz jego języka zleceń
testowanie, czy zostały już zarejestrowane odpowiednie
zdarzenia. Wedlug definicji zawartych w pozycji [4] sekwencja
ta mogłaby mieć następującą postać:

        var C1 : Cond;

C1 := (* deklaracja kryterium zakończenia rejestracji *)

        Set_condition(C1);

        Put_CPU(HBR);

*)
```

```
i := GetTrace;

DisplayTime(Timings, Mode, i);
```

Niezależnie od założeń konieczna wydaje się również możliwość współpracy procedur z mapą symboli generowaną przez języki programowe. Jej użycie zapewni możliwość deklaracji kwalifikatorów rejestracji zdarzeń raczej poprzez podawanie nazw logicznych adresów w programie (nazw danych i procedur) niż ich fizycznych lokacji. Wiele kompilatorów dostarcza również informacji o związkach między generowanym kodem maszynowym a programem źródłowym, pozwala to uruchamiać program obserwując przebieg sterowania zarówno na plikach źródłowych, jak i na kodzie maszynowym.

Sama definicja wizualizacji zarejestrowanych danych powinna mieć następującą postać:

```
procedure DisplayTime( Timings : array [0..K-1] of Bus,
                       Mode    : SpecifiedModes
                       Count   : integer              );
```

gdzie parametr `Mode` jest zmienną typu wyliczeniowego `SpecifiedModes` określającą interesujący użytkownika sposób i przedmiot wizualizacji, a `Count` określa ilość zarejestrowanych cykli. Możliwe sposoby wizualizacji zarejestrowanych zdarzeń będą przedmiotem rozważań zawartych w punkcie 7.

4. Analiza czasu realizacji kodu programu

Uzależnienia czasowe, związane z pobieraniem i realizacją kodu mogą być częstym źródłem błędów lub wad projektu, jeśli ich wartości są istotnie różne od założonych. Śledzenie poszczególnych momentów realizacji kodu programu polega na deklaracji kwalifikatora rejestracji ich wystąpienia w postaci:

```
Condition :=  "ADDRESS = <adres-w-pamięci-kodu-programu> and
                 STATUS = <cykl-dostępu-do-kodu>"
```

lub w ogólnym przypadku

```
Condition :=  "ADDRESS R <adres-1> or ADDRESS R <adres-2>
                or ... ADDRESS R <adres-N> ) and
                STATUS = <cykl-dostępu-do-kodu>"
```

gdzie R może być dowolną relacją typu =, >, <, >= lub <=.

Istnieje wiele różnych szczegółowych problemów, związanych z pomiarami i doborem czasów realizacji kodu oprogramowania, jednak najistotniejszymi wydają się poniższe:

1) **Pomiar czasu wykonania procedury.** Realizuje się go poprzez rejestrację momentów rozpoczęcia i zakończenia wykonania procedury. Dotyczy on głównie reakcji systemu czasu rzeczywistego na określone wydarzenia w otoczeniu. Typowym przykładem błędu związanego ze zbyt długim czasem wykonania procedury jest sytuacja obsługi wystąpienia zdarzenia periodycznego, gdy czas trwania procedury przekracza długość odstępu pomiędzy kolejnymi zdarzeniami wywołującymi wykonanie procedury. Zilustrują tę sytuację dwa poniższe przykłady:

Przykład 1. W trakcie uruchamiania nowych procedur obsługi kanałów RS232 w jądrze systemu XENIX-86, po wstępnej optymalizacji kodu procedur sterujących, zauważono zwolnienie pracy systemu przy długich raportach na terminal dołączony przez RS232. Komunikacja była prowadzona z szybkością 9600 Bd (ok. 1 ms na znak). Należało w tej sytuacji rozstrzygnąć następujące kwestie:

a) które procedury obsługi są realizowane zbyt długo; jak zminimalizować czasochłonność; co trzeba jeszcze zoptymalizować lub wręcz napisać w języku asemblera,

b) jak wygląda zajętość procesora 8088 (procesor systemu uruchamianego) przez procedury obsługi transmisji przy oscylatorze kwarcowym procesora o częstotliwości 12 MHz (system prototypowy),

c) jak zmieni się sytuacja przy oscylatorze 18 MHz (system docelowy).

Procesor 8088 posiada wewnątrz dwa niezależne moduły funkcjonalne: jeden obsługuje magistralę, drugi realizuje rozkazy wewnątrz procesora. Powoduje to, że zmiana częstotliwości kwarcu wpływa nieliniowo na zmianę szybkości realizacji programu (np. wzrastająca częstotliwość przyspiesza realizację kodu, ale jego pobieranie nie uległo przyspieszeniu, bo magistrala jest ta sama). Zatem tylko pomiar czasów realizacji procedur od strony magistrali może dać jednoznaczną odpowiedź.

Przykład 2. W środowisku MS-DOS, przy użyciu monitora czasu rzeczywistego wzorowanego na architekturze systemu XINU, zrealizowano program sterujący emulatorem RTDS/2-PC. Jego struktura była wielozadaniowa, celem pobierania pamięci śladu bez wstrzymywania pracy konsoli (współbieżna transmisja śladu z emulatora) oraz umożliwienia równoległej transmisji programu do pamięci prototypu w czasie realizacji innych zleceń sterujących emulatorem. Napotkano następującą błędną sytuację, w której aktywne były dwa procesy:

A. proces o wyższym priorytecie, realizujący komunikację z klawiaturą, a w danej chwili wykonujący sekwencję rozkazów z pliku zleceń,

B. proces transmitujący plik do pamięci prototypu.

Proces A sprawdzał w pętli, czy proces B zakończył transmisję. Przez nieuwagę programisty została skasowana instrukcja usypiania na pewien czas procesu A w czasie tego typu pracy, aby procesy o niższych priorytetach mogły realizować współbieżnie swoje zadania. Równocześnie dokonano poprawki protokołu transmisji celem jej przyspieszenia. W trakcie uruchomienia powyższego testu proces B został zagłodzony czasowo, cały czas aktywny był proces A. Szybkość transmisji spadła do zera, najprostsze rozwiązanie to zmierzyć w tej sytuacji, ile czasu zużywają uruchamiane procedury, ale brak odpowiednich narzędzi spowodował, że uwaga programistów skupiła się na logice protokołu.

Innego rodzaju błędy mogą wynikać z niewłaściwego czasu pracy poszczególnych modułów oprogramowania, prowadząc do braku synchronizacji pracy procesów lub nieoczekiwanych spowolnień pracy systemu.

Przykład 3. W sytuacji opisywanej w przykładzie 1 początkowe wersje oprogramowania RS232 realizowały obsługę przerwania w ok. 1200 cyklach magistrali (pomiar dokonywany emulatorem RTDS/2-PC), co przy kwarcu 12 MHz procesora powodowało całkowite wstrzymanie pracy głównej konsoli. Po przyspieszeniu procedur transmisji system zawiesił się, zmiana prędkości jednego z procesów na lini "producent-konsument" spowodowała ujawnienie się dotychczas nie wykrytych błędów synchronizacji procesów, powodując upadek systemu.

Wykrycie tego typu błędów kieruje uwagę programisty na odcinki programu, gdzie niezbędna jest optymalizacja kodu źródłowego, wprowadzenie dodatkowych elementów synchronizacji lub wręcz użycie innego algorytmu.

2) **Pomiar odstępów pomiędzy wejściami do procedury.** Realizuje się go poprzez rejestrację wszystkich momentów rozpoczęcia wykonania procedury. Jest on często związany ze wspomnianymi problemami wynikającymi z konieczności dopasowania czasu wykonania procedury do częstości jej wywoływania.

Przykład 4. Po zakupieniu systemu operacyjnego czasu rzeczywistego przystępujemy do projektowania struktury oprogramowania aplikacyjnego. W dokumentacji systemu podano czasy realizacji poszczególnych elementarnych operacji. Użytkujemy dodatkowe pakiety programowe (oprogramowanie sieciowe, system plików, bazy danych), które z założenia operują w środowisku wirtualnym, więc nie posiadają specyfikacji czasowych. Dosyć dobrze przybliża tę sytuację użytkowanie systemu QNX (wg [9]). Jak oszacować przepustowość planowanej instalacji? Jaki będzie czas reakcji jej poszczególnych komponentów? Warto od tej chwili rejestrować momenty dostępu do wyspecyfikowanych czasowo elementów systemu oraz wiedzieć, jak często będą wywoływane, aby zebrać informacje o elementach użytkowanego oprogramowania, które nie są wystarczająco opisane na nasze potrzeby.

Częstość wywoływania danej procedury może również być wskaźnikiem stanu, w jakim znajduje się uruchamiany system czasu rzeczywistego.

Przykład 5. W wielu systemach wielozadaniowych istnieje proces tracenia czasu (przykładowo w systemie XENIX jest to procedura `idle()` jądra systemu). Pomiar, kiedy i jak często ten proces jest realizowany, pozwala czasem rozstrzygnąć, jaka operacja jest w danej chwili realizowana, np.:

- proces tracenia czasu nieaktywny — długie obliczenia lub przeciążony procesor,
- periodyczne, krótkie realizacje procesu tracenia czasu — najprawdopodobniej bardzo szybka transmisja danych w łączu RS232, być może kanał DMA (zależnie od zmierzonego okresu powrotu do realizacji tracenia czasu),

- periodyczne, długie realizacje procesu tracenia czasu z krótką przerwą o częstotliwości zegara czasu rzeczywistego — oczekiwanie, zakłócane tylko tykaniem zegara,
- inne charakterystyczne sekwencje, wynikające z właściwości urządzeń fizycznych i architektury systemu — współpraca z urządzeniem lub kontrolerem określonej operacji.

Oparcie się na właściwościach urządzeń fizycznych jest naturalne i uzasadnione dla systemów mikroprocesorowych z założenia współpracujących z tego typu otoczeniem.

3) Ocena częstości wywołań określonych funkcji oprogramowania.

Realizuje się ją na podstawie zarejestrowanych momentów wywołań różnych procedur, których realizacja określa stan oprogramowania, według arbitralnych kryteriów przyjętych przez programistę. Może ona być związana z sygnalizowaną uprzednio oceną stanu, w jakim znajduje się oprogramowanie czasu rzeczywistego. Przy zaawansowanej znajomości cech oprogramowania (tzn. znajomości, jakie procedury w danym stanie systemu są typowo realizowane oraz jak często, kiedy i w jakiej kolejności następuje ich wywoływanie) programista jest w stanie określić charakterystyczne stany systemu, sygnalizowane zarejestrowanymi czasami i według takiego kryterium oceniać poprawność realizacji kodu bez wstrzymywania pracy programu.

5. Analiza przepływu danych w systemie prototypowym

Badanie uzależnień czasowych, związanych ze zmianą danych lub ich przepływem, daje zwykle wiele przesłanek do wynajdywania błędów powstałych zarówno na etapie projektowania oprogramowania, jak i na etapie założeń, dotyczących nadchodzenia samych danych. Śledzenie poszczególnych momentów dostępów do danych programu polega na deklaracji kwalifikatora rejestracji ich wystąpienia w postaci:

```
Condition :=  "ADDRESS = <adres-w-pamięci-danych-programu> and
                    STATUS = <cykl-dostępu-do-danych>"
```

lub w ogólnym przypadku

```
Condition :=  "ADDRESS R <adres-1> or ADDRESS R <adres-2>
                or ... ADDRESS R <adres-N> ) and
              STATUS = <cykl-dostępu-do-danych>"
```

gdzie R może być dowolną relacją typu =, >, <, >= lub <=. Deklaracja może być również rozszerzona o wyszczególnienia typu cyklu (zapisu lub odczytu danych, zapisu, lub odczytu z portu itp.) oraz o szczegółowe deklaracje dopuszczalnych wartości danych kwalifikujących rejestrację:

```
Condition :=  "ADDRESS R <adres-1> or ADDRESS R <adres-2>
                or ... ADDRESS R <adres-N> ) and
              DATA R <dana-1> or DATA R <dana-2>
                or ... DATA R <dana-N> ) and
              STATUS = <typ-1> or STATUS = <typ-2>
                or ... STATUS = <typ-N>)
```

Typowe analizy przepływu danych mogą być następujące:

1) Pomiar częstości dostępu do danych. Realizuje się go poprzez rejestrację momentów dostępów do danych programu. Wyniki uzyskane z takiego pomiaru świadczą o szybkości realizacji programu, a równoczesne ustalenie, kiedy i jaka procedura realizowała dostęp do danych, często ułatwia usuwanie błędów związanych z synchronizacją lub kolizjami w systemach wielozadaniowych.

   Przykład 6. W opisywanym w przykładzie 2 środowisku, w trakcie uruchamiania jego modułów komunikacyjnych, napotkano sytuację zaprzestania transmisji po wysłaniu ok. 2 - 3 kilobajtów danych. Nasuwały się następujące hipotezy:

   a) odbiorca danych został zatkany dużą ilością danych (był to procesor dużo wolniejszy od nadającego) — błąd logiczny protokołu,

   b) protokół ma błędy, powodujące upadek przy przekłamaniu transmisji (przekłamanie mogło być spowodowane nałożeniem danych u wolnego odbiorcy lub błędem parzystości, związanym z jego dużym prawdopodobieństwem przy tej ilości danych),

c) nadawca lub odbiorca w czasie transmisji wykonuje błędną operację, związaną z wielozadaniowym środowiskiem,

d) inne.

Obieg danych w obu systemach był realizowany opierając się na puli buforów kontrolowanej przez monitor wielozadaniowy. W rzeczywistości jeden z procesów nadawcy danych nie zwracał zużytych buforów do puli i w końcu zawieszał się w oczekiwaniu na wolny bufor. Śledzenie czasów dostępu do buforów nadawcy i odbiorcy mogłoby szybko zweryfikować powyższe wstępne hipotezy. W użytym emulatorze nie było mechanizmów pomiaru czasu, natomiast mechanizm rejestracji 16-cyklowych kadrów zdarzeń (tzn. cykl operacji śledzonej plus 15 cykli występujących wokół zadeklarowanego zdarzenia, wg [4]) pozwolił zidentyfikować źródło błędu (proces gubiący bufory). W każdej innej sytuacji tylko narzędzie posiadające możliwość rejestracji zarówno kadrów zdarzeń, jak i czasów ich wystąpienia pozwoliłoby od razu zweryfikować większość możliwych przyczyn błędu.

2) **Pomiar prędkości nadchodzenia danych i ich przepływu.** Realizuje się go poprzez rejestrację momentów dostępu do portów lub danych dynamicznych oprogramowania, zatem często wymaga szczegółowej wiedzy o architekturze systemu lub sposobach dynamicznej gospodarki pamięcią realizowanych przez użyte oprogramowanie narzędziowe (systemy operacyjne czasu rzeczywistego, języki programowania itp.). Analiza zapisu i odczytu danych z buforów cyklicznych oraz analiza odczytów i zapisów do portów prototypu może dać wiele przesłanek do usuwania błędów powstałych na styku "producent-konsument danych", kiedy dane mogą nie być chronione mechanizmami monitora wielozadaniowego lub samego mikroprocesora.

Przykład 7. W systemie na procesorze 8051 funkcjonują 3 procesy:

A. Odbiorca danych z łącza RS232 do bufora cyklicznego B1 (256 bajtów).

B. Odbiorca danych z bufora B1. Dane są interpretowane i przekazywane do bufora B2. Gdy informacja jest skompletowana, zawartość bufora B2 jest przetwarzana i przesyłana do bufora B3.

C. Proces wizualizujący dane z bufora B3. Jest on aktywizowany cyklicznie co 2ms.

Najwyższy priorytet ma proces A, a najniższy B. Po rozszerzeniu algorytmu procesu C o dodatkowe możliwości wizualizacji transmisja szeregowa

zaczęła się załamywać. Maksymalny komunikat nie mógł przekroczyć ok. 300 bajtów. Po wielu testach okazało się, że proces B został zagłodzony przez proces C (tzn. miał za mało czasu na realizację swych funkcji), co spowodowało błąd przepełnienia bufora cyklicznego. Posiadanie odpowiedniego narzędzia i prosty pomiar momentów dostępu do bufora B1 (nawet zwykłe ich zliczenie) wyjaśniłoby tę sytuację od razu.

Błędy związane z różnicą szybkości pracy producenta danych i ich odbiorcy mogą być źródłem zaskakujących, trudnych do oceny zachowań uruchamianego oprogramowania.

3) **Ocena częstości dostępu do danych.** Realizuje się ją na podstawie zarejestrowanych momentów dostępów do danych, do których dostęp określa stan oprogramowanie, według arbitralnych kryteriów przyjętych przez programistę. Podobnie jak wspomniana w rozdziale 4. ocena częstości wywołań funkcji, ocena częstości dostępu do danych może dostarczać charakterystycznego śladu stanu oprogramowania uruchamianego systemu mikroprocesorowego, rozumianego jako pewien typowy stan uzależnień czasowych systemu.

6. Analiza stanów magistrali

Niezależnie od cykli dostępu do kodu, danych i portów większość mikroprocesorów realizuje jeszcze inne cykle magistrali, związane ze współpracą z kontrolerami rozszerzającymi możliwości funkcjonalne procesora. Śledzenie poszczególnych cykli magistrali polega na deklaracji kwalifikatora rejestracji ich wystąpienia w postaci:

```
Condition :=   "STATUS = <wybrany-typ-cyklu-magistrali>"
```

lub w ogólnym przypadku

```
Condition :=   "STATUS = <typ-1> or STATUS = <typ-2>
                    or ... STATUS = <typ-N>
```

Rejestracja i analiza cykli pobrania kodu i dostępu do danych została skomentowana w poprzednich rozdziałach. Poniżej zostaną opisane ważniejsze aspekty analizy innych typów cykli:

1) **Cykl obsługi przerwań.** Jego detekcja jest realizowana w celu dopasowania czasów wykonania procedur do częstości przychodzenia przerwań lub odwrotnie — doboru maksymalnej częstości napływu przerwań do aktualnych parametrów czasowych oprogramowania. W typowej sytuacji programista ocenia maksymalny czas wykonania obsługi przerwania, a następnie za pomocą analizatora lub oscyloskopu dokonuje pomiarów przebiegów czasowych magistrali. Analogiczne pomiary można wykonać od strony procesora, a przy odpowiednim doborze taktu zegara czasu rzeczywistego w emulatorze mogą one być równie dokładne.

   Przykład 8. W trakcie generacji procedur sterujących transmisją z dysku twardego w systemie XENIX-86 powstał problem doboru przeplotu sektorów na posiadanym oryginalnym dysku, tzn. ustalenia, gdzie powinien być sektor o następnym numerze, aby był pod głowicą czytająco/piszącą w momencie, gdy rozpoczyna się operacja dostępu do niego. Pomiaru dokonywano za pomocą oscyloskopu, mierzone były odstępy pomiędzy przerwaniami sygnalizującymi koniec transmisji jednego sektora. Znając prędkość obrotową dysku i ilość sektorów na ścieżce, można było oszacować, czy nastąpił dodatkowy obrót dysku (za gęsty przeplot), czy też nie (przeplot dobry lub za szeroki, wtedy również można by ocenić, o ile za szeroki). Podobnych pomiarów można było dokonać od strony magistrali bez potrzeby rozszerzania stanowiska o dodatkowe urządzenia, gdyby używany emulator miał mechanizm rejestracji czasów zdarzeń. Rejestracja obsługi przerwania od kontrolera dysku pozwoliłaby rejestrować czasy transmisji poszczególnych sektorów oraz dokonać oceny przeplotu.

2) **Cykl stanu HALT.** Część mikroprocesorów posiada specyficzny rozkaz HALT, wstrzymujący pracę procesora. Realizację tego rozkazu można zakończyć tylko za pomocą przerwania lub resetu procesora. Wiele monitorów wielozadaniowych ma za pomocą tego rozkazu zrealizowany proces tracenia czasu. Analiza magistrali pod kątem realizacji tego stanu dostarcza, w takim przypadku, danych o zajętości systemu.

   Przykład 9. Na początku lat 80 pracownicy ZSAK PAN w Gliwicach uruchamiali system mikroprocesorowy oparty na mikroprocesorze Z80

oraz autorskim monitorze wielozadaniowym. Proces tracenia czasu był zrealizowany za pomocą rozkazu HALT. Diagnoza stanu systemu polegała na "naocznej" ocenie jasności świecenia i częstotliwości mrugania diody luminescencyjnej monitorującej stan sygnału HALT.

Możliwa jest oczywiście ocena charakterystycznego stanu systemu za pomocą tego stanu magistrali, jeśli tylko jest on wykorzystywany w oprogramowaniu.

3) **Stan HOLD magistrali.** W niektórych mikroprocesorach można spotkać się z opisem tego stanu jako PSV (passive) lub innymi. Dotyczy on sytuacji, gdy procesor odcina się od magistrali w celu udostępnienia jej innym kontrolerom. Może to być na przykład związane z transmisją DMA. Znajomość architektury systemu pozwoli wyciągnąć określone wnioski z faktu pojawienia się tego cyklu, jego czasu i częstości występowania oraz ewentualnych korelacji powodujących nieoczekiwane efekty w realizacji funkcji systemu. W systemach wieloprocesorowych lub posiadających koprocesory można również napotkać stany magistrali powodowane przez rozkazy mikroprocesora oznaczane przez INTELa jako LOCK (wyłączość dostępu do danych) oraz WAIT (oczekiwanie na koprocesor). Znaczenie ich analizy jest podobne jak dla sygnału HOLD.

Przykład 10. IBM PC/XT realizuje odświeżanie pamięci dynamicznych za pomocą kanału DMA. Można by dokonać pomiaru, jaki procent czasu jest tracony na odświeżanie i czy obliczenia są w ten sposób spowalniane, biorąc pod uwagę kolejkowanie rozkazów do wykonania wewnątrz procesora 8088. Analogiczna sytuacja może dotyczyć systemów wieloprocesorowych, w jakim stopniu kolizje na magistrali spowalniają obliczenia. Przykłady te mają na celu uświadomienie, jakich pomiarów można i należy dokonywać przy tworzeniu własnych rozwiązań.

4) **Stan WAIT magistrali.** Jeśli współpraca z pamięcią i portami procesora oparta jest na testowaniu aktywności sygnału WAIT, wystawianego przez dany zasób zewnętrzny, pożyteczna może okazać się operacja pomiaru tego czasu od strony mikroprocesora. W ten sposób można uzyskać wiele wniosków dotyczących wpływu prędkości używanych pamięci i portów na prędkość realizacji algorytmów oraz dostarczyć przesłanek do modyfikacji sprzętu lub oprogramowania prototypu.

Przykład 11. W systemie wejścia/wyjścia mikrokomputera IBM PC/AT rezydującym w pamięci stałej (ROM BIOS) zauważamy po każdym do-

stępie do portu sztuczną operację zerowania kolejki wewnętrznej procesora. Ma ona na celu opóźnienie następnego dostępu do portu, aby wyeliminować potencjalne przekłamania danych, gdyż ilość taktów WAIT jest dla portów stała, ale i tak często za mała dla współpracy szybkiego procesora 80286 z kontrolerami starszej generacji zastosowanymi w AT. Zapewniało to uniwersalność konstrukcji i równoczesną jej niezawodność, ale w przypadku konstruowania specjalizowanej jednostki ważne może się okazać każde przyspieszenie jej pracy i wtedy trzeba zacząć od pomiarów i doboru odpowiednio szybkich kontrolerów.

W przypadku występowania błędów sprzętowych istotny do ich wykrycia może być pomiar długości trwania specyficznych cykli. Dla tego typu pomiarów niezbędne wydaje się zwiększanie częstotliwości zegara czasu rzeczywistego w emulatorze, aby wyniki pomiarów były uzyskiwane z pożądaną dokładnością.

7. Sposoby wizualizacji uzależnień czasowych

Poniżej zostaną opisane potencjalne możliwości realizacji procedury `DisplayTime`. Ogólne założenia dla tej procedury powinny być takie, aby zależnie od rodzaju pomiaru sposób wizualizacji był zgodny z typowymi oczekiwaniami operatora (np. użycie emulatora jako analizatora logicznego magistrali implikuje wizualizację w postaci przebiegów czasowych poszczególnych jednobitowych sygnałów logicznych). Mimo to nie zawsze ograniczone, wyłącznie standardowe możliwości są korzystne, zważywszy na różne intencje, jakich można by od użytkownika oczekiwać. Typowe metody mogą być następujące:

1) **Wyświetlanie zawartości tablicy** `Timings`. Możliwość ta musi być pozostawiona dla zaawansowanych prac, kiedy mimo że wyświetlenie wszystkich danych utrudnia często ich interpretację, to jednak jest jedyną szansą sformułowania jakiejkolwiek przesłanki o przyczynie błędu.

2) **Symboliczna interpretacja tablicy** `Timings` , na podstawie tablic symboli użytych w oprogramowaniu i plików źródłowych. Wyświetlanie symbolicznej informacji o realizacji oprogramowania, łącznie z informacją z plików źródłowych uzupełnioną o czasy realizacji funkcji, czasy osiągania określonych fragmentów kodu, czasy pomiędzy dostępami do danych itp.

3) **Statystyki dostępu do kodu i danych.** Graficzna lub tabelaryczna prezentacja statystyk dostępów do wybranych elementów kodu, lub danych. Mogą one być podstawą do określania charakterystycznego śladu stanu oprogramowania.

4) **Prezentacje graficzne dostępów do danych i realizacji fragmentów kodu** jako przebiegów czasowych. Znaczenie tego sposobu również dotyczy określania charakterystycznego śladu stanu oprogramowania.

5) **Prezentacje graficzne przebiegów czasowych określonych sygnałów magistrali.** Mają one na celu pełne umożliwienie wykorzystania emulatora układowego jako analizatora logicznego magistrali uruchamianego prototypu.

8. Podsumowanie

Po przeprowadzeniu powyższych rozważań otrzymujemy zestaw procedur i funkcji, których sprzętowo-programowa implementacja pozwoli na rejestrację i prezentację informacji o uzależnieniach czasowych uruchamianego prototypu. Powinny one być zaimplementowane w środowisku emulatora układowego lub analizatora magistrali mikroprocesora, zaprojektowanego zgodnie z uprzednio wyspecyfikowanymi wymogami, pozwalającymi na zachowanie trybu pracy prototypu w czasie rzeczywistym. Po zestawieniu funkcje te, zadeklarowane w gramatyce języka *Pascal*, prezentują się następująco:

a) Ustawianie warunku rejestracji zdarzeń i start rejestracji:

```
procedure SetTrace(Condition: Cond);
```

b) Pobranie zarejestrowanych wyników w ustalone miejsce:

```
function GetTrace : integer;
```

c) Prezentacja zarejestrowanych danych w określonych trybach:

```
procedure DisplayTime( Timings : array [0..K-1] of Bus,
                       Mode    : SpecifiedModes
                       Count   : integer
```

Operują one na następujących stowarzyszonych z nimi strukturami danych:

a) Struktura zawierająca logiczny warunek rejestracji zdarzeń:

```
type Cond = record
(* Szczególna zawartość deklaracji warunku będąca
najczęściej
odbiciem struktury sprzętowej modułu testującego. *)
end;

var Condition : Cond;
```

b) Tablica zawierająca zarejestrowane zdarzenia:

```
type Bus = record
     Address   : integer;
               (* Adres wystawiony w danym cyklu *)
     Data      : integer;
               (* Dana wystawiana w danym cyklu *)
     Status    : array [0..N-1] of Bit;
               (* N-bitowy status cyklu *)
     Time      : integer;
               (* Moment wystawienia cyklu *)
end;

var Timings : array [0..K-1] of Bus;
```

Współpraca zaproponowanych procedur z emulatorem i emulowanym prototypem będzie miała następującą postać:

System sterujący		Emulator układowy		System Prototypowy
S1:Inicjacja emulatora	→ E1:	Inicjacja elektroniki	→ P1:	Reset procesora
	E2:	Start zegara czasu rzeczywistego		
	E3:	OCZEKIWANIE	→ P2:	Obsługa kanału komunikacyjnego emulatora
S2:Inicjacja logiczna Condition				
S3:SetTrace	→ E4:	Ustawienie układów komparujących rejestracji zdarzeń		
	E5:	Start programu systemu prototypowego	→ P3:	Realizacja własnego programu, którego zachowania będą rejestrowane
S4:Oczekiwanie na wyniki: if skompletowana rejestracja then następny paragraf	E6:	Rejestracja zdarzeń w prototypie		
S5:GetTrace	→ E7:	cobegin Przesył danych; Rejestracja kolejnych zdarzeń; coend		
	E8:	goto E6		
S6:DisplayTime				
S7:goto S4				

Należy tu zwrócić uwagę na funkcje realizowane w paragrafie E7, gdzie tylko pełna równoległość ich realizacji zapewni nieprzerwane zachowywanie trybu rejestracji danych w czasie rzeczywistym. Równie istotna dla zachowania tego warunku jest możliwość buforowania informacji rejestrowanych w paragrafie E7 oraz wielkość buforów, gdy w paragrafie S6 realizowana jest prezentacja danych, uniemożliwiająca bieżący odbiór zarejestrowanych danych od układów rejestrujących.

Pierwszy powyższy wymóg odnosi się do prawdziwości zarejestrowanych czasów wystąpienia określonych zdarzeń, natomiast spełnienie drugiego zapewni ciągłość rejestrowanych danych oraz ich aktualność.

9. Uwagi końcowe

Przedstawione powyżej środki i sposoby analizy czasowej uruchamianego prototypu, za pomocą emulatora układowego, mają w zasadzie charakter poglądowy; są zaledwie przeglądem potencjalnych możliwości istniejących już konstrukcji. W wielu realizacjach emulatorów były one bardziej lub mniej rozwijane, zależnie od oczekiwań i sugestii użytkowników oraz linii rozwoju konstrukcji przyjętej przez producenta. W wielu konstrukcjach emulatorów napotykamy silnie rozwiniętą prezentację symboliczną, związaną z faktem, że większość producentów emulatorów dostarcza własne kompilatory i opiera się na własnej informacji symbolicznej. Graficzna prezentacja jest również często spotykana, ale chociaż wizualnie jest bardzo efektowna, to kwestia jej efektywnego wykorzystania jest otwarta. Prezentacja wielu zjawisk zachodzących w czasie jest czytelniejsza w postaci graficznej, jednak przy dużej ilości danych powstają istotne problemy w jej wyświetlaniu. Dodatkowym problemem może być stworzenie prostego w użyciu języka, umożliwiającego deklarację wyświetlanych graficznie danych. Mimo to prezentacja graficzna wyników należy do bardziej komunikatywnych środków wizualizacji, co powoduje jej częste, lepiej lub gorzej zrealizowane, stosowanie.

W wielu przypadkach analiza uzależnień czasowych jest właściwie możliwa tylko "post mortem", tzn. po wstrzymaniu emulowanego prototypu, co wiąże się z przerwaniem pracy w czasie rzeczywistym już po zaobserwowanych wydarzeniach. Nie zawsze jest to dopuszczalne, jeśli użytkownik nie chce przerywać pracy emulowanego systemu, jednak wiele emulatorów oferuje taki właśnie tryb analizy śladu. W innych konstrukcjach wstrzymanie może nie nastąpić, ale na czas odczytu i analizy zarejestrowanego śladu wstrzymywana jest rejestracja następnych zawartości magistrali. Istnieje zatem zagrożenie utraty ciągłości odczytów interesujących stanów.

Najbardziej użyteczne dla prezentowanego w opracowaniu sposobu użytkowania emulatora rozwiązanie — rejestracja i wizualizacja na bieżąco, bez utraty danych i warunków pracy w czasie rzeczywistym, może i tak spowodować przepełnienie buforów oraz w konsekwencji utratę danych. Ponadto to ostatnie rozwiązanie stwarza wiele problemów natury konstrukcyjnej — układy śledzące pracują z bardzo dużą szybkością, często na granicy swych parametrów katalogowych, a dodatkowe ich przyspieszanie, w celu realizacji przeplotu zapisów stanu magistrali z odczytami ich od strony emulatora, zawsze będzie związane z dużymi kosztami sprzętu, co w wielu przypadkach uczyni to przedsięwzięcie nieopłacalne lub nawet nierealizowalne technicznie.

Wielokrotnie sygnalizowana możliwość analizy charakterystycznych śladów stanu oprogramowania powinna, ogólnie biorąc, oznaczać możliwość wykrywania podobieństw w uzależnieniach czasowych emulowanego prototypu do zadanych. "Zadany" wzorzec oznacza przebiegi wzorcowe albo przebiegi, których użytkownik się "nauczył" w trakcie uruchamiania oprogramowania. Można założyć, że problem doboru kryterium "podobieństwa" charakterystycznego śladu do wzorca oraz problem znajomości charakterystycznych śladów typowych sytuacji w przypadku ich analizy "na oko" (tylko przez człowieka) jest kwestią doświadczenia i intuicji projektanta. Przypadek automatycznej analizy i wykrywania podobieństw oraz automatycznej nauki cech danego stanu wydaje się już problemem złożonym i jego rozwijanie wykracza już poza założone ramy tego opracowania.

Wiele emulatorów posiada w swej warstwie programowej zaimplementowany, pełny funkcjonalnie język zleceń, pozwalający pisać programy, automatyzujące realizację złożonych sekwencji zleceń emulatora. Ogólnie omówiona procedura `DisplayTime` może być oczywiście jednym z elementów takiego języka. Wydaje się, że kwestia doboru składni tego zlecenia lub rodziny zleceń, realizujących wyświetlanie w środowisku danego języka, mogłaby być tematem odrębnym, również wykraczającym poza przyjęte ramy prezentowanego opracowania.

Niezależnie od pojawiających się, w trakcie uruchamiania nowej konstrukcji, wymogów pomiaru czasu w emulowanym prototypie, część współczesnych emulatorów nie posiada tej możliwości. Orientują się one raczej na maksymalnie efektywne uruchamianie kodu programu w trybie debuggingu programowego oraz wizualizację danych, zatem dla pewnej grupy istniejących konstrukcji emulatorów układowych powyższe rozważania są bezprzedmiotowe.

LITERATURA

[1] Intel — "ICE-86A/ICE-88A MICROSYSTEM IN-CIRCUIT EMULATOR OPERATING INSTRUCTION FOR ISIS-II USERS", Intel 1982.

[2] Microtec — "MICE-II Users Guide", Microtec 1986.

[3] Intel — "ICE-196 PC User Guide", Intel 1987.

[4] Dariusz Zonenberg — "Emulator układowy mikroprocesorów jako narzędzie uruchomieniowe w ujęciu programistycznym.", Wydawnictwo SIGMA-NOT, mies. INFORMATYKA nr 2, Warszawa 1992.

[5] Dariusz Zonenberg — "Emulator układowy — obserwacja danych dla analizy zachowań systemów czasu rzeczywistego." Seminaria IITiS PAN, Gliwice listopad 1990.

[6] MERA-Elzab — "RTDS/2-PC. Dokumentacja użytkowa.", MERA-Elzab, Zabrze 1989.

[7] Raport IITiS PAN — "Stanowisko wspomagające analizę systemów mikrokomputerowych, bazujących na procesorze 80286", IITiS PAN, Gliwice, wrzesień 1990.

[8] Edward Yourdon — "Projektowanie systemów o działaniu bezpośrednim.", Wydawnictwa Naukowo-Techniczne, Warszawa 1976.

[9] Quantum Software Systems — "QNX — Operating System", Quantum Software Systems, Ltd. 1988.

Recenzent: Prof. dr hab. inż. Andrzej Grzywak

Wpłynęło do Redakcji 28 listopada 1991 r.

Abstract

The article pointed out the necessity of time registration while debugging current microprocessor system software using an ICE. Next, basic possibilities of time measurement and event filtering and registration in a debugging system were presented. Using the *Pascal* language, the basic functions of an ICE, connected with events registration,

were defined. A *Pascal*-based bus event filter declaration language was also informally proposed. Next paragraphs contain an overview of basic bus event filters and their semantics. The paper ends in a discussion of basic possibilities of the presentation of the registered events that would connect registered time stamps with event semantics. Conclusions point out the ICE functional features necessary for the application area discussed. The problem of automatic recognition of software signature (characteristic timings of bus events connected with software execution) and the problem of defining a language for registered events presentation were indicated as still open.