Model i implementacja dwurdzeniowego sterownika programowalnego opartego na maszynie wirtualnej

Marcin Hubacz, Jan Sadolewski, Bartosz Trybus
Politechniki Rzeszowska, Wydział Elektrotechniki i Informatyki, Katedra Informatyki i Automatyki, ul. Wincentego Pola 2, 35-021 Rzeszów

Streszczenie: Przedstawiono koncepcję, model semantyczny oraz prototypową implementację dwurdzeniowego sterownika programowalnego. Koncepcja budowy sterownika obejmuje równoległe przetwarzanie dwóch programów wykonawczych za pośrednictwem maszyny wirtualnej, wykorzystując wspólny obszar pamięci zmiennych globalnych. Zaprezentowany model przedstawia formalny opis wykonywania przenaszalnych programów binarnych powstałych na podstawie języków normy IEC 61131-3 w środowisku programistycznym CPDev. Przedstawiona architektura opisuje działanie maszyny wirtualnej za pomocą abstrakcyjnych obiektów algebraicznych. Rozwiązanie zaimplementowane zostało w języku C/C++ na dwurdzeniowej platformie mikrokontrolerowej.

Słowa kluczowe: PLC, IEC 61131-3, model formalny, maszyna wirtualna

1. Wprowadzenie

Rosnąca złożoność systemów automatyki przemysłowej stawia wyższe wymagania co do mocy obliczeniowej jednostek sterujących. Nowoczesne aplikacje podnoszą oczekiwania w zakresie pracy w czasie rzeczywistym, bezpieczeństwa i niezawodności. Jednocześnie ograniczenia technologiczne układów scalonych uniemożliwiają poprawę wydajności przez wyłącznie zwiększanie częstotliwości taktowania jednostek obliczeniowych. Obecnie rozwiązaniem tego problemu mogą być jednostki wielordzeniowe. Można zauważyć, że ewolucja systemów wbudowanych zmierza w tym kierunku, czego przykładem są komercyjne rozwiązania, np. seria Embedded PC firmy Beckhoff [13], która stosuje dwurdzeniowy procesor ARM Cortex-A53 oraz PLCnext firmy Phoenix Contact [14] z dwurdzeniowym procesorem ARM Cortex-A9.

Według badań [6] architektura ARM (ang. *Advanced RISC Machine*) znajduje zastosowanie w aż 70 % systemów wbudowanych. Jej najważniejszymi zaletami są wysoka wydajność, stosunkowo niskie zużycie energii oraz korzystna cena jednostkowa. Popularność ta wynika również z elastycznego sposobu licencjonowania tej architektury. Rodzina procesorów Cortex, oparta na wspomnianej architekturze składa się z trzech głównych serii: aplikacyjnej (A), mikrokontrolerowej (M) oraz czasu rzeczywistego i bezpieczeństwa (R). Warto zaznaczyć, że rodzina M nie zawiera jednostki MMU (ang. *Memory Management Unit*), w przeciwieństwie do rdzeni A i R. MMU jest wymagany do implementacji zaawansowanych systemów operacyjnych.

Producenci układów opartych na architekturze ARM wyposażają je często w kilka rdzeni procesorowych. Możliwe jest wówczas bezstratne podzielenie zadań w zależności od zapotrzebowania. W niniejszej pracy cecha ta jest wykorzystywana do wykonywania dwóch projektów sterujących jednocześnie, co odpowiada zastosowaniu dwóch niezależnych sterowników PLC (PLC+PLC). Innym przykładem może być połączenie sterowania (PLC) z interfejsem człowiek-maszyna (HMI), które można określić w skrócie rozwiązaniem PLC+HMI. Dodatkowy podział może objąć również zadania dodatkowe, jak zarządzanie magistralą obiektową, połączenie z chmurą, serwer OPC UA i wiele innych. Takie połączenie można określić połączeniem PLC+EXT TASK. Inne rozwiązanie znane jest pod nazwą sterowników redundantnych lub zabezpieczeniowych, zwyczajowo wyposażonych w dwa procesory wykonujące dwa programy: główny i rezerwowy.

Norma IEC 61131-3 [15] standaryzująca automatykę przemysłową definiuje pięć języków programowania. Do wyboru są tekstowe IL i ST, graficzne LD i FBD oraz mieszany SFC. Jednostki organizacyjne oprogramowania (POU) zdefiniowane w normie składają się z programów, bloków funkcyjnych i funkcji. Zmienne dzieli się na lokalne, deklarowane wewnątrz jednostek i globalne, dostępne w ramach całego projektu, służące także do komunikacji z otoczeniem. Do przykładowych środowisk implementujących standard IEC 61131-3 zaliczyć można STEP7 [2], CODESYS [16] czy LogicLab [1]. Budowę takiego narzędzia można zasadniczo podzielić na trzy główne komponenty: środowisko IDE, kompilator i środowisko uruchomieniowe (ang. *runtime*). W edytorach języków normy tworzony jest system sterowania, który następnie kompilator przekształca na wykonywalny kod binarny. Ten kod jest przenoszony do środowiska uruchomieniowego, dedykowanego dla danego sterownika. Kod jest wykonywany w czasie rzeczywistym, w sposób cykliczny lub w reakcji na określone zdarzenie.

Autor korespondujący:
Marcin Hubacz, m.hubacz@prz.edu.pl

Artykuł recenzowany
nadesłany 03.12.2023 r., przyjęty do druku 21.08.2024 r.

Środowisko programistyczno-uruchomieniowe CPDev zostało zaprojektowane zgodnie z wymogami obowiązującej normy automatyki IEC 61131-3. Podstawą działania sterownika jest koncepcja maszyny wirtualnej [12], której zadaniem jest przetwarzanie binarnego kodu pośredniego generowanego przez kompilator. Takie podejście zyskało na znaczeniu ze względu na powszechne wykorzystanie Javy [17] i .NET [18, 19]. Rozwiązania oparte na maszynach wirtualnych mają kilka istotnych zalet, takich jak: a) niezależność programu źródłowego i kodu pośredniego od platform docelowych, b) jeden kompilator, c) wykonywanie programu w chronionym środowisku. Jednak wady to: a) wolniejsze wykonywanie kodu pośredniego oraz b) konieczność przygotowania środowiska uruchomieniowego dla konkretnej platformy docelowej.

W środowisku CPDev kompilator przekształca kod ST (inne języki są wcześniej konwertowane do ST) do kodu pośredniego VMASM, który jest następnie przetwarzany przez zaimplementowaną na danej platformie docelowej maszynę wirtualną opracowaną w języku C [8, 12]. Początkowo rozważano małe i średnie sterowniki. Jednak w ostatnim czasie pojawiła się potrzeba, motywowana bardziej rozbudowanymi aplikacjami, rozszerzenia kompilatora i maszyny wirtualnej CPDev o wsparcie dla dwurdzeniowych platform wykonawczych. W związku z tym opracowano formalny model semantyczny dwurdzeniowego sterownika z dwoma pracującymi maszynami wirtualnymi i nowymi instrukcjami do wymiany zmiennych między rdzeniami. Następnie przystąpiono do implementacji prototypu w języku C na wybranej platformie sprzętowej. Przygotowany model formalizuje niezależne działanie maszyn wirtualnych na dwóch jednostkach wykonawczych. Każda z nich przetwarza swój kod pośredni, na co składa się dekodowanie instrukcji, ich operandów oraz niskopoziomowych operacji. W opisie formalnym jest stosowana semantyka denotacyjna [3, 11] odpowiednia dla języków programowania [7, 10]. Zastosowana została adekwatna notacja wykorzystująca wyrażenia lambda [3, 20].

2. Koncepcja rozwiązania

Koncepcję dwurdzeniowego sterownika programowalnego można postrzegać jako równoważną dwóm niezależnym sterownikom programowalnym połączonym pewnym łączem komunikacyjnym. W ramach opracowanego rozszerzenia wykorzystano obszar pamięci współdzielonej dostępny w układach wielordzeniowych.

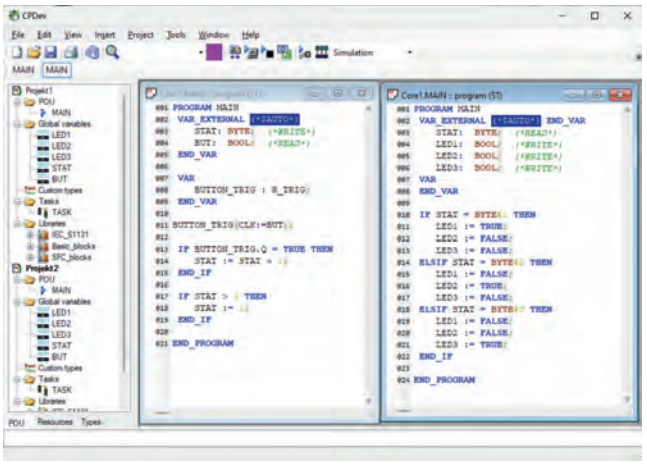
Głównym założeniem modyfikacji jest zachowanie w możliwie niezmienionej formie schematu działania maszyny wirtualnej w każdym z rdzeni. W związku z tym utrzymywana jest niezależność pracy i zasobów instancji maszyny, ze szczególnym uwzględnieniem obszarów pamięci dla zmiennych. Realizacja kanału wymiany informacji wymagała jednak podziału pamięci na lokalną dla każdego z rdzeni i współdzieloną. Pierwsza została zachowana zgodnie z pierwotną koncepcją, natomiast druga stanowi niezależne rozszerzenie.

Na rysunku 1 przedstawiono ogólną architekturę dwurdzeniowego sterownika PLC, gdzie Projekt 1 i Projekt 2 reprezentują oprogramowanie sterujące wykonywane przez rdzenie. Maszyna wirtualna została rozszerzona o dodatkowe instrukcje i mechanizmy zarządzające pamięcią w porównaniu do wersji jednordzeniowej. Zależność między współpracującymi projektami opiera się na wymianie zmiennych globalnych przez pamięć współdzieloną, do której dostęp mają oba rdzenie układu.

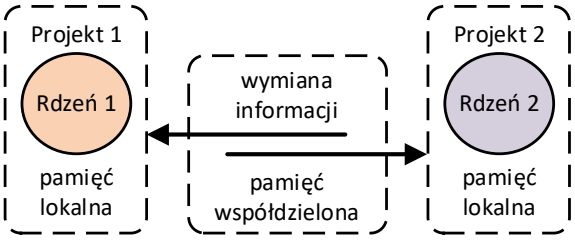
Aby utworzyć łącze komunikacyjne w oparciu o pamięć współdzieloną, konieczna jest synchronizacja współdzielonych zmiennych globalnych między dwoma projektami. Ze względów bezpieczeństwa i spójności dostęp do zmiennych musi być zabezpieczony przed odczytem podczas aktualizacji. Praktyką programistyczną jest aktualizowanie zmiennej wyłącznie z jednego projektu. W związku z tym wprowadzono dodatkowe reguły kontrolujące dostęp do zmiennej globalnej.

W projekcie zawarta jest pełna lista zmiennych globalnych. Specjalne atrybuty zmiennych oznaczają typ dostępu do zmiennej z odpowiedniego rdzenia. I tak, element listy może być modyfikowany (WRITE = 1, READ = 0), odczytywany (WRITE = 0, READ = 1) lub nieużywany (WRITE = 0, READ = 0). Zastosowanie przygotowanych reguł umożliwia bezpieczny dostęp i edycję zmiennych w projektach współpracujących. Środowisko CPDev umożliwia jednoczesne rozwijanie i edycję obu projektów, co przedstawione zostało na rysunku 2.

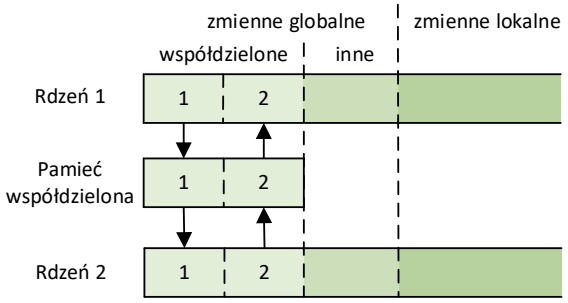
Podczas wykonywania cyklu sterowania przez sterownik PLC stosowany jest tryb odczyt-wykonanie-zapis w odniesieniu do zmiennych globalnych. Taki układ skutkuje przeniesieniem zmiennych wejściowych do lokalnych kopii pamięci, gdzie są używane. Po zakończeniu cyklu sterowania, zmienne wyjściowe przekazywane są do globalnych kopii. W praktyce oznacza to, że wszystkie wartości będące wejściami są przenoszone do wewnętrznych lokalnych kopii. Aby zaimplementować takie zachowanie, dwurdzeniowy sterownik musi zostać rozszerzony



Rys. 2. Projekty współpracujące Projekt1 i Projekt2 dla dwurdzeniowego sterownika w środowisku CPDev
Fig. 2. Cooperating projects Project1 and Project2 for a dual-core controller in the CPDev environment



Rys. 1. Ogólna architektura dwurdzeniowego sterownika PLC
Fig. 1. General architecture of a dual-core PLC



Rys. 3. Organizacja pamięci dwurdzeniowego sterownika PLC
Fig. 3. Memory organization of a dual-core PLC controller

o operacje kopiowania z pamięci współdzielonej na początku cyklu i kopiowania do niej na końcu. Na rysunku 3 przedstawiono przykład operacji odczytu i zapisu zmiennych globalnych do pamięci współdzielonej przez oba rdzenie.

3. Dziedziny semantyczne

Modele semantyczne umożliwiają formalny opis języków programowania [9, 10]. Tutaj to podejście zostało zastosowane do modelowania działania maszyny wirtualnej dla systemów sterowania. Na opis maszyny składają się różne dziedziny semantyczne, które definiują stan, operacje dotyczące pamięci, interpretery wartości odnoszące się do typów danych normy IEC, operatory o ograniczonym zasięgu czy uniwersalne funkcje semantyczne wywołujące konkretne instrukcje maszyny wirtualnej.

Model obejmuje typy danych charakteryzujące wartości przetwarzane przez maszynę wirtualną. Domena *BasicTypes* jest unią czterech zbiorów bajtów reprezentujących rozmiary podstawowych typów danych (np. BOOL, BYTE – 1B; INT, WORD – 2B; DWORD, REAL, ADDRESS – 4B; LWORD, LREAL – 8B).

$$BasicTypes = Byte1 + Bytes2 + Bytes4 + Bytes8$$

Dziedzina *Memory* jest funkcją mapującą *Address* (*Bytes*4, czyli adres 32-bitowy) do *Byte*1. Maszyna wirtualna używa dwóch obszarów pamięci: *CodeMemory* zawierający kod programu do wykonania oraz *DataMemory*, gdzie przechowywane są wartości zmiennych. Dziedzina *Stack* reprezentuje sekwencję wartości (oznaczoną jako $*$ – symbol domknięcia Kleenego) z dziedziny *Address*, przy czym *CodeStack* i *DataStack* służą jako aliasy dla konkretnych typów stosów. Stosy *CodeStack* i *DataStack* są używane przy skokach do procedur (np. bloków funkcjonalnych). *CodeReg* i *DataReg* reprezentują rejestry bazowe dla kodu i danych, zaś rejestr *Flags* przechowuje różne flagi statusowe.

$$\begin{gathered} Address = Bytes4 \\ Memory = Address \rightarrow Byte1 \\ CodeMemory = Memory \\ DataMemory = Memory \\ Stack = Address* \\ CodeStack = Stack \\ DataStack = Stack \\ CodeReg = Address \\ DataReg = Address \\ Flags = Bytes2 \end{gathered}$$

Fundamentalnym celem wykonania programu jest zmiana aktualnego stanu na nowy. Bieżący stan maszyny wirtualnej jest iloczynem kartezjańskim dziedzin pamięci stanów, rejestrów, stosów i flag itd.

$$\begin{gathered} State = CodeMemory \times DataMemory \times CodeStack \times \\ DataStack \times CodeReg \times DataReg \times Flags \end{gathered}$$

Dziedzinę oznaczoną jako *State* można także opisać jako zbiór krotek:

$$(cm,\ dm,\ cs,\ ds,\ cr,\ dr,\ flg)$$

gdzie każdy element odpowiada wartości w odpowiadającej mu dziedzinie.

Funkcje przedstawione dalej modelują niskopoziomowe operacje związane z pamięcią, stosem i flagami.

– Pobieranie z pamięci (odczyt) danych o określonym rozmiarze

$$\begin{gathered} G1BM = (Address \times Memory) \rightarrow Byte1 \\ G2BM = (Address \times Memory) \rightarrow Bytes2 \\ G4BM = (Address \times Memory) \rightarrow Bytes4 \\ G8BM = (Address \times Memory) \rightarrow Bytes8 \end{gathered}$$

– Pobieranie adresu z pamięci

$$GetAddress = (Address \times Memory) \rightarrow Address$$

Funkcja zwraca wartość przechowywaną pod danym adresem w pamięci, który jest innym adresem (adresacja pośrednia). Maszyna wirtualna nie ma rejestru akumulatora i działa bezpośrednio na adresach, stąd funkcja *GetAddress* jest istotna dla modelu.

– Aktualizacja pamięci (zapis) danych o określonym rozmiarze

$$\begin{gathered} U1BM = (Address \times Memory \times Byte1) \rightarrow Memory \\ U2BM = (Address \times Memory \times Bytes2) \rightarrow Memory \\ U4BM = (Address \times Memory \times Bytes4) \rightarrow Memory \\ U8BM = (Address \times Memory \times Bytes8) \rightarrow Memory \end{gathered}$$

– Przenoszenie pamięci (kopiowanie)

$$\begin{gathered} MemMove = \\ (Address \times Memory \times Address \times Memory \times Byte1) \\ \rightarrow Memory \end{gathered}$$

gdzie *Address* reprezentuje źródło i cel kopiowania pamięci, a *Byte1* zawiera liczbę kopiowanych bajtów danych (0–255).

– Funkcje stosu

$$\begin{gathered} Push = (Stack \times Address) \rightarrow Stack \\ Pop = Stack \rightarrow (Address \times Stack) \end{gathered}$$

Funkcje wykonują operacje stosu potrzebne w podprogramach. Należy zauważyć, że funkcja *Pop* zwraca parę, tj. adres i nową zawartość stosu.

Interpretatory wartości realizują następujące przykładowe interpretacje fragmentów pamięci na dane określonych typów.

$$\begin{gathered} BoolOf = Byte1 \rightarrow BOOL \\ FromBool = BOOL \rightarrow Byte1 \\ IntOf = Bytes2 \rightarrow INT \\ FromInt = INT \rightarrow Bytes2 \\ DIntOf = Bytes4 \rightarrow DINT \\ FromDInt = DINT \rightarrow Bytes4 \\ LIntOf = Bytes8 \rightarrow LINT \\ FromLInt = LINT \rightarrow Bytes8 \end{gathered}$$

4. Model maszyny wirtualnej

Maszyna wirtualna pobiera kody poszczególnych instrukcji wraz z operandami z pamięci kodu (*CodeMemory*). Kod każdej z instrukcji składa się z dwóch składowych, oznaczających grupę, do której należy instrukcja (*ig*) oraz typ jej określonego działania (*it*). Do przedstawienia koncepcji grupy i typu, po którym następuje wykonanie określonej instrukcji, w modelu zdefiniowano uniwersalną funkcję obejmującą wszystkie instrukcje:

$$\mathcal{U}[[any\_instruction]] = State \rightarrow State$$

Wewnętrznie, po zdekodowaniu *ig* i *it*, jest wywoływana określona funkcja maszyny wirtualnej.

$$\mathcal{C}[[instruction]] = State \rightarrow State$$

Na rysunku 4 przedstawiono schemat blokowy algorytmu dekodowania instrukcji maszyny wirtualnej. Najpierw pobierane są jej składowe *ig* i *it*. Następnie grupa *ig* jest porównywana z określonymi wartościami. Wartość $ig = 05$ oznacza grupę instrukcji negacji bitowej (NOT), zaś $ig = 12$ oznacza porównania (EQ). W przypadku obu grup składowa *it* wybiera typ danych. Stąd dla $it = 10$ negacja dotyczy typu BOOL, a $it = 11$ typ BYTE. Dla instrukcji EQ $it = 00$ oznacza typ BOOL, a $it = 01$ typ SINT.

Dekodowanie instrukcji można formalnie wyrazić denotacyjnym równaniem semantycznym pokazanym na listingu 1. Zgodnie z [3] lub [20], wyrażenie λ ma postać $\lambda s.body$, gdzie s oznacza aktualny stan, a *body* określa wartość zwracaną przez funkcję. *Body* składa się z sekwencji operacji, z których pierwsza dzieli bieżący stan s na krotkę złożoną z komponentów modelu. Następnie wykonywane są operacje dekowania wartości identyfikatorów *ig* oraz *it*, aktualizacji rejestru kodu do cr_2 i przez dopasowanie wywołanie odpowiednich funkcji $\mathcal{C}$. Otrzymany wynik z $\mathcal{C}$ definiuje nowy stan s_1 zwrócony przez funkcję $\mathcal{U}$.

Modelowanie instrukcji maszyny za pomocą równań denotacyjnych przedstawić można ogólnie jako $\mathcal{C}[[x]] = \lambda s.body$, gdzie zastępuje się deskryptorem konkretnej instrukcji. Pierwszą operacją w *body* jest podzielenie przez unifikacje aktualnego stanu s na komponenty.

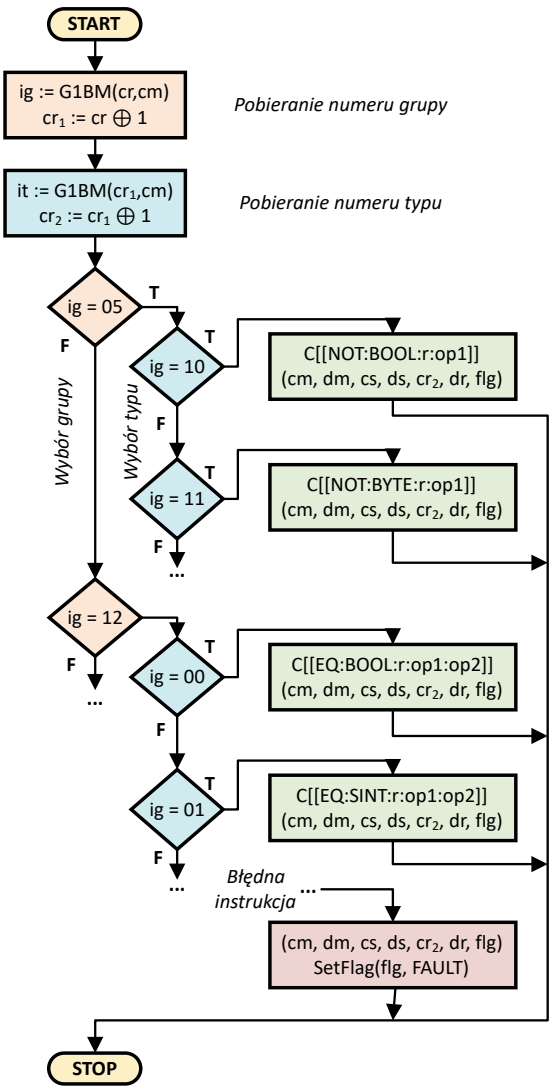
$$(cm,\ dm,\ cs,\ ds,\ cr,\ dr,\ flg) := s$$

Uzyskanie wartości zmiennej, np. logicznej w pamięci danych *dm* uzyskuje się za pomocą:

$$BoolOf(G1BM(operandaddr,\ dm))$$

$\mathcal{U}[[any\_instruction]] = \lambda s.$
$(cm,\ dm,\ cs,\ ds,\ cr,\ dr,\ flg) := s$
$ig := G1BM(cr,\ cm)$
$cr_1 := cr \oplus 1$
$it := G1BM(cr_1,\ cm)$
$cr_2 := cr_1 \oplus 1$
$s_1 :=$ ***match*** ig ***with***
$\quad |05 \rightarrow$ ***match*** it ***with***
$\quad\quad |10 \rightarrow \mathcal{C}[[NOT\text{: }BOOL\text{: }r\text{: }op1]]$
$\quad\quad\quad (cm,\ dm,\ cs,\ ds,\ cr_2,\ dr,\ flg)$
$\quad\quad |11 \rightarrow \mathcal{C}[[NOT\text{: }BOOL\text{: }r\text{: }op1]]$
$\quad\quad\quad (cm,\ dm,\ cs,\ ds,\ cr_2,\ dr,\ flg)$
$\quad\quad |\ \ldots$ ***end***
$\quad |\ \ldots$ ***end***
s_1

Listing 1. Równanie semantyczne funkcji dekodującej
Listing 1. Semantic equation of the decoding function



Rys. 4. Algorytm dekodowania instrukcji maszyny wirtualnej
Fig. 4. Instruction decoding algorithm of the virtual machine

W przypadku, gdy instrukcja ma operand, rejestr kodu *cr* jest zwiększany, wskazując na następną komórkę pamięci:

$$cr_1 := cr \oplus AddressSize$$

Zdefiniowanie nowego stanu s_1 w postaci krotki jest ostatnią operacją w *body*, podczas której kropki zastępowane są nowymi wartościami pamięci danych (jeżeli aktualizowane), stosami itp.

$$s_1 := (cm,\ \ldots)$$

Semantyka funkcji NOT przedstawiona na listingu 2 neguje wartość przechowywaną w *op*1. Adresy *raddr*, *op*1*addr* wyznaczane są, jak wskazano poprzednio, po czym następuje wartość logiczna *bv*. Za pomocą funkcji *U*1*BM* aktualizowana jest wówczas wartość *raddr* w pamięci danych *dm*. Wartość ta jest określana na podstawie *bv* przez *FromBool* i dopasowywana, a *um* oznacza nowy stan pamięci danych.

$\mathcal{C}[[NOT\text{: }r\text{: }op1]] = \lambda s.$
$\quad (cm,\ dm,\ cs,\ ds,\ cr,\ dr,\ flg) := s$
$r := GetAddress(cr,\ cm)$
$raddr := dr \oplus r$
$cr_1 := cr \oplus AddressSize$
$op1 := GetAddress(cr_1,\ cm)$
$op1addr := dr \oplus op1$
$cr_2 := cr_1 \oplus AddressSize$
$bv := BoolOf(G1BM(op1addr,\ dm))$
$um := U1BM\ (raddr,\ dm,\ FromBool($***match*** bv ***with***
$\quad |05 \rightarrow$ ***match*** it ***with***
$\quad\quad |$***true*** $\rightarrow$ ***false***
$\quad\quad |$***false*** $\rightarrow$ ***true end***$))$
$s_1 := (cm,\ dm,\ cs,\ ds,\ cr_2,\ dr,\ flg)$
s_1

Listing 2. Równanie semantyczne funkcji NOT
Listing 2. Semantic equation of the function NOT

5. Model dla dwóch rdzeni

Ze względu na konieczność wymiany zmiennych globalnych przez rdzenie sterownika model uwzględnia pamięć współdzieloną:

$$SharedMemory = Memory$$

Stan systemu uwzględniający obie maszyny wirtualne pracujące na dwóch rdzeniach jest określony następująco:

$$CoreState = CodeMemory \times DataMemory \times CodeStack \times DataStack \times CodeReg \times Flags$$

$$State = CoreState \times CoreState \times SharedMemory \times SharedFlags$$

Stan systemu dwurdzeniowego uwzględnia dodatkowo flagi statusowe pamięci dzielonej *SharedFlags*. Oznaczając obie maszyny za pomocą indeksów A i B stan ten można również zapisać jako:

$$((cm_A, dm_A, cs_A, ds_A, cr_A, dr_A, flg_A), (cm_B, dm_B, cs_B, ds_B, cr_B, dr_B, flg_B))$$

Jak wspomniano, synchronizacja zmiennych globalnych odbywa się poprzez kopiowanie ich wartości z pamięci rdzenia do pamięci globalnej i odwrotnie. Służą do tego procedury `DM_TO_SH` (*Data Memory To Shared Memory*) i `SH_TO_DM` (*Shared Memory To Data Memory*). Poniżej przedstawiono równanie denotacyjne dla pierwszej z nich, przy założeniu, że następuje aktualizacja z maszyny A do pamięci dzielonej.

$$\begin{aligned}
&\mathcal{C}[[DM_TO_SH: LocalAddress: SharedAddress: ByteNumber]] = \lambda s.\\
&((cm_A, dm_A, cs_A, ds_A, cr_A, dr_A, flg_A), (cm_B, dm_B, cs_B, ds_B, cr_B, dr_B,\\
&flg_B), sm, sf) := s\\
&srcaddr := GetAddress(cr_A, cm_A)\\
&cr_1 := cr_A \oplus AddressSize\\
&dstaddr := GetAddress(cr_1, cm_A)\\
&cr_2 := cr_1 \oplus AddressSize\\
&size := ByteOf(Get1BMem(cr_2\ cm_A))\\
&cr_3 := cr_2 \oplus 1\\
&usm := MemMove(dm_A, srcaddr, sm, dstaddr, size)\\
&s_1 := ((cm_A, dm_A, cs_A, ds_A, cr_3, dr_A, flg_A), (cm_B, dm_B, cs_B, ds_B, cr_B,\\
&dr_B, flg_B), sm, sf)\\
&s_1
\end{aligned}$$

Listing 3. Równanie semantyczne funkcji DM_TO_SH
Listing 3. Semantic equation of the function DM_TO_SH

Równanie dla procedury synchronizacji w drugą stronę, tj. `SH_TO_DM` jest analogiczne.

Jak wspomniano, należy zapobiec aktualizacji pamięci współdzielonej przez dwie maszyny jednocześnie. W tym celu można zastosować flagę `SharedFlags.InUse`, która określa zajętość pamięci współdzielonej. Algorytm dostępu do pamięci w celu jej synchronizacji można wtedy zapisać jak na listingu 3.

```
while (SharedFlags.InUse) {
    if (TaskCycleExceeded)
        SetFailureFlag;
    }
}
SharedFlags.SharedInUse := true;
DM_TO_SH(LocalAddr, SharedAddr, ByteNumber);
SharedFlags.SharedInUse := false;
```

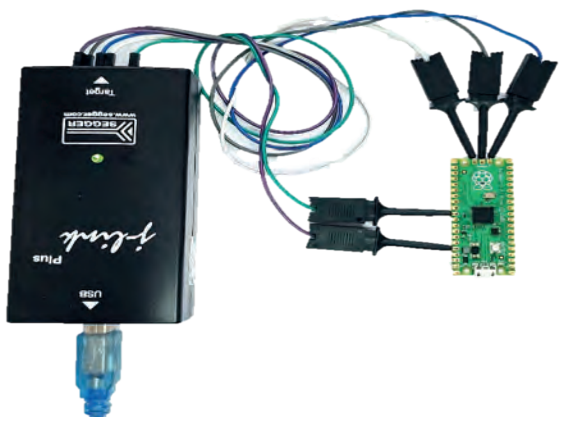
Listing 4. Algorytm aktualizacji pamięci współdzielonej
Listing 4. Shared memory synchronization algorithm

6. Implementacja

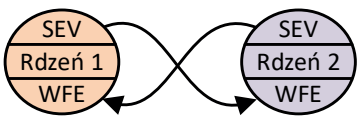
Model formalny został zaimplementowany w języku C na popularnym dwurdzeniowym mikrokontrolerze homogenicznym RP2040 firmy Raspberry Pi Foundation. Układ ten jest złożony z dwóch rdzeni Cortex-M0+ o maksymalnej częstotliwości taktowania wynoszącej 133 MHz. Płytę użytą do aplikacji rozwiązania przedstawiono na rys. 5.

W sytuacji, gdy dwie lub więcej współpracujących aplikacji pracuje w czasie rzeczywistym, zazwyczaj zachodzi potrzeba synchronizacji zasobów. Układ RP2040 jest wyposażony w funkcje SEV (*Send Event*) oraz WFE (*Wait For Event*) [21], co umożliwia wzajemne sprzętowe powiadamianie między rdzeniami. To rozwiązanie pozwala na wymuszenie zaplanowanej akcji w drugim rdzeniu, na przykład aktywacji lub wstrzymania programu sterującego, bądź aktualizacji pamięci współdzielonej. Relacja między rdzeniami oraz kierunek wyzwalania takiego przerwania dla układu zostały przedstawione na rys. 6.

W układach wielordzeniowych, w celu ochrony współdzielonego obszaru pamięci przed potencjalnymi konfliktami odczytu i zapisu, często stosuje się specjalne sprzętowe warianty semaforów. W wykorzystanym układzie zaimplementowano 32 instancje mechanizmu Spinlock [21], który można interpretować jako jednobitową flagę. Temu mechanizmowi odpowiada konstrukcja `SharedFlags.InUse` (pkt 5). Przykładowy pełny cykl pracy dwurdzeniowego sterownika PLC dla jednego z rdzeni został przedstawiony w listingu 5. W fazie *precycle* (przed wykonaniem cyklu sterowania), lokalny obszar pamięci danych maszyny wirtualnej jest aktualizowany przy użyciu globalnych zmiennych o atrybucie READ. Następnie realizowana jest faza obliczeniowa, w której maszyna wirtualna interpretuje przygotowany binarny kod sterowania. Na zakończenie, w fazie *postcycle*, pamięć współdzielona jest aktualizowana o nowe wartości zmiennych z atrybutem WRITE. Każda operacja modyfikacji obszaru pamięci współdzielonej podlega obowiązkowemu zablokowaniu semafora przed rozpoczęciem kopiowania, a następnie zwolnieniu go po zakończeniu. Odczyt bieżących wartości do pamięci współdzielonej odbywa się w analogiczny sposób. Funkcje `CPDev.CopyDataMemToShareMem`() oraz `CPDev.CopyDataMemToShareMem`() kopiują wartości zmiennych globalnych między



Rys. 5. Płyta deweloperska z układem RP2040
Fig. 5. Development board with RP2040 chip



Rys. 6. Zależność między rdzeniami układu RP2040 w oparciu o mechanizm SEV i WFE
Fig. 6. The relationship between the cores of the RP2040 system based on the SEV and WFE mechanism

pamięcią lokalną i globalną. Efektywność tego kopiowania zależy od ułożenia zmiennych w pamięci oraz ograniczeń platformy sprzętowej [4].

```
//precycle
while (SpinlockInUse(spinlock1)) {
    if (CPDev.CheckTaskCycleExceeded())
        CPDev.SetFailureFlag();}
SpinlockTake(spinlock1);
CPDev.CopyShareMemToDataMem();
SpinlockGive(spinlock1);
//cycle
CPDev.RunCycle();
//postcycle
while (SpinlockInUse(spinlock1)) {
    if (CPDev.CheckTaskCycleExceeded())
        CPDev.SetFailureFlag();}
SpinlockTake(spinlock1);
CPDev.CopyDataMemToShareMem();
SpinlockGive(spinlock1);
```

Listing 5. Fragment implementacji jednej z instancji dwurdzeniowego sterownika PLC na układzie RP2040
Listing 5. A fragment of the implementation of one of the instances of a dual-core PLC controller on the RP2040 chip

7. Podsumowanie

W ramach prezentowanej pracy przedstawiono architekturę i model dwurdzeniowego sterownika PLC, który bazuje na maszynie wirtualnej. Denotacyjny model semantyczny opisuje jej działanie, polegające na wykonywaniu projektów sterujących opracowanych zgodnie z normą IEC 61131-3. Do realizacji dwurdzeniowego sterownika opracowano mechanizm wymiany informacji między projektami, wykorzystujący współdzielony obszar pamięci. Ten fragment został zabezpieczony przed potencjalnymi błędami odczytu aktualnie modyfikowanego obszaru.

Implementacja oparta jest na przygotowanych równaniach denotacyjnych, które modelują dwie współpracujące maszyny wirtualne, co umożliwia wykonywanie dwóch projektów. Rozwiązanie zostało zaimplementowane w języku C/C++ na dwurdzeniowym mikrokontrolerze homogenicznym RP2040. Do zapewnienia wzajemnego wykluczania podczas dostępu do współdzielonego obszaru pamięci wykorzystano sprzętowe mechanizmy układu. Utworzony prototyp, wykorzystujący ekonomiczne układy sprzętowe, wskazuje, że jest możliwe jednoczesne sterowanie dwoma obiektami w czasie rzeczywistym, co zazwyczaj wymagałoby zastosowania dwóch osobnych sterowników.

Bibliografia

1. Becker M., Sandström K., Behnam M., Nolte T., *A Many-Core Based Execution Framework for IEC 61131-3.* [In:] Proceedings of the IECON 2015 – 41st Annual Conference of the IEEE Industrial Electronics Society, Yokohama, Japan, 2015, DOI: 10.1109/IECON.2015.7392805.
2. Cisek J., Mikluszka W., Swider Z., Trybus L., *A Low-Cost DCS with Multifunction Instruments and CAN Bus*, „IFAC Proceedings Volumes", Vol. 34, No. 29, 2001, 64–69, DOI: 10.1016/S1474-6670(17)32794-5.
3. Gordon M., *The Denotational Description of Programming Languages*, Springer-Verlag, New York, 1979.
4. Hubacz M., Trybus B., *Data Alignment on Embedded CPUs for Programmable Control Devices*, „Electronics", Vol. 11, No. 14, 2022, DOI: 10.3390/electronics11142174.
5. Hubacz M., Trybus B., *Dual-Core PLC for Cooperating Projects with Software Implementation*, "Electronics", Vol. 12, No. 23, 2023, DOI: 10.3390/electronics12234730.
6. John K.H., Tiegelkamp M., *IEC 61131-3: Programming Industrial Automation Systems*, Springer, Berlin/Heidelberg, Germany, 2010.
7. Papaspyrou N.S., *Denotational semantics of ANSI C*, "Computer Standards & Interfaces", Vol. 23, No. 3, 2001, 169–185, DOI: 10.1016/S0920-5489(01)00059-9.
8. Sadolewski J., Trybus B., *Compiler and virtual machine of a multiplatform control environment*, "Bulletin of the Polish Academy of Sciences Technical Sciences", Vol. 70, No. 2, 2022, DOI: 10.24425/bpasts.2022.140554.
9. Schmidt D., *Denotational Semantics: A Methodology for Language Development*. Kansas State University, Department of Computing and Information Sciences, Manhattan, 1997.
10. Slonneger K., Kurtz B.L., *Formal Syntax and Semantics of Programming Languages: A Laboratory-Based Approach,* Addison-Wesley Publishing Company, 1995.
11. Stoy J., *Denotational Semantics: The Scott–Strachey approach to programming language theory*, Massachusetts Institute of Technology, 1979.
12. Trybus B., *Development and Implementation of IEC 61131-3 Virtual Machine*, „Theoretical and Applied Informatics", Vol. 23, No. 1, 2011, 21–35.

Inne źródła

13. [www.beckhoff.com/en-en/products/ipc/embedded-pcs/cx8200-arm-cortex-a53/] – CX8200|Embedded PC Series (Compact Controller). Beckhoff Automation (1 grudnia 2023).
14. [www.phoenixcontact.com/en-gb/products/plcs-controllers-and-i-os/automation-technology-for-plcnext-technology] – Automation Technology for PLCnext Technology. Phoenix Contact (1 grudnia 2023).
15. [www.openampproject.org/] – The OpenAMP Project (1 grudnia 2023).
16. [www.xilinx.com/products/silicon-devices/soc/zynq-7000.html] – Zynq 7000 SOC. AMD (XILINX) (1 grudnia 2023).
17. Lindholm T., Yellin F., Bracha G., Buckley A., *The Java® Virtual Machine Specification*, Oracle America, 2013.
18. Thai T.L., Lam H., *.NET Framework Essentials*, O'Reilly Media, 2001
19. *ECMA-335 Standard Common Language Infrastructure (CLI)*, ECMA, Geneva, 2012.
20. Barendregt H., Barendsen E., *Introduction to Lambda Calculus*, 2000, [ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf].
21. [https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf] – RP2040 Datasheet (1 grudnia 2023).

Model and Implementation of a Dual-Core Programmable Controller Based on a Virtual Machine

Abstract: The concept, semantic model, and prototype implementation of a dual-core programmable controller have been presented. The controller's design concept involves parallel processing of two execution programs through a virtual machine, utilizing a shared memory area for global variables. The presented model provides a formal description of the execution of portable binary programs created based on the languages of the IEC 61131-3 standard in the CPDev programming environment. The architecture described outlines the operation of the virtual machine using abstract algebraic objects. The solution has been implemented in C/C++ on a dual-core microcontroller platform.

Keywords: PLC, IEC 61131-3, formal model, virtual machine

mgr inż. Marcin Hubacz
m.hubacz@prz.edu.pl
ORCID: 0000-0002-2748-1145

W 2019 r. ukończył studia na Wydziale Elektrotechniki i Informatyki Politechniki Rzeszowskiej – kierunek Automatyka i Robotyka oraz Informatyka. Obecnie Asystent w Katedrze Informatyki i Automatyki Politechniki Rzeszowskiej. Jego główne zainteresowania dotyczą robotyki, elektroniki, systemów wbudowanych oraz druku 3D.

dr inż. Jan Sadolewski
jsad@prz.edu.pl
ORCID: 0000-0001-7370-9027

Absolwent Wydziału Elektrotechniki i Informatyki Politechniki Rzeszowskiej (2006 r.). W 2012 r. uzyskał stopień doktora nauk technicznych w dyscyplinie informatyka na Wydziale Automatyki, Elektroniki i Informatyki Politechniki Śląskiej w Gliwicach. Jego zainteresowania naukowe koncentrują się wokół języków programowania, tworzenia kompilatorów oraz środowisk wykonawczych.

dr inż. Bartosz Trybus
btrybus@kia.prz.edu.pl
ORCID: 0000-0002-4588-3973

Adiunkt w Katedrze Informatyki i Automatyki Politechniki Rzeszowskiej. Ukończył studia na Wydziale Elektrycznym, Automatyki, Informatyki i Elektroniki AGH w Krakowie. Doktorat z informatyki uzyskał w 2004 r. Jego główne badania dotyczą systemów czasu rzeczywistego i środowisk wykonawczych oprogramowania sterującego.