

Zaawansowane Architektury Komputerów



Laboratorium 4 – JIT dla architektury DLX

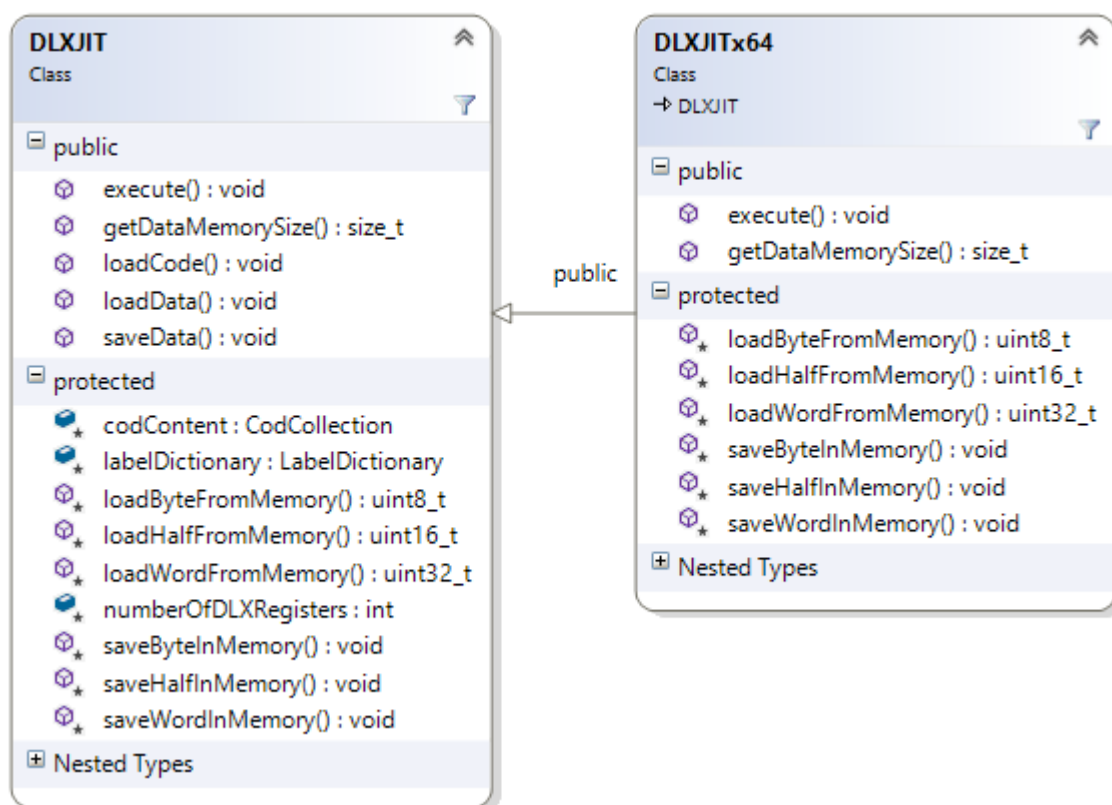
Tomasz Bieliński

2018-04-24

Wstęp

Celem tego ćwiczenia laboratoryjnego będzie implementacja prostego kompilatora JIT (ang. Just in Time), który będzie konwertował kod dla procesora DLX do kodu maszynowego architektury IA-32e (znanej również jako x86_64, x64 albo amd64) [1]. Do instrukcji został załączony projekt dla Visual Studio 2013 w którym zostały zaimplementowane operacje wczytywania kodu DLX, wczytywania stanu pamięci i zapisywanie stanu pamięci zgodnie z formatem symulatora Escape [2] oraz część funkcjonalności kompilatora JIT.

Struktura programu DLXJIT



Rysunek 1 Klasa abstrakcyjna DLXJIT zawierająca funkcjonalności niezależne od architektury docelowej oraz klasa DLXJITx64 zawierająca implementację kompilatora JIT dla architektury IA-32e

Głównymi klasami programu są DLXJIT oraz DLXJITx64 (Rysunek 1). Klasa DLXJIT zawiera następujące metody publiczne:

- `loadCode()` – ładuje plik cod zgodny z formatem symulatora ESCAPE. Program zawarty w pliku jest analizowany i w postaci listy struktur `DLXJITCodLine` jest przechowywany do czasu kompilacji. Ponadto metoda ta tworzy słownik etykiet – `labelDictionary`.

- `loadData()` – ładuje plik dat zgodny z formatem symulatora ESCAPE. Wykorzystuje metody abstrakcyjne `saveByteInMemory()`, `saveHalfInMemory()`, `saveWordInMemory()`.
- `saveData()` – zapisuje plik dat zgodny z formatem symulatora ESCAPE. Wykorzystuje metody abstrakcyjne `loadByteFromMemory()`, `loadHalfFromMemory()`, `loadWordFromMemory()`.
- `execute()` – metoda abstrakcyjna. Kompiluje program do natywnej architektury, a następnie go wykonuje. Zaimplementowana w DLXJITx64.
- `getDataMemorySize()` – metoda abstrakcyjna. Podaje rozmiar pamięci danych. Zaimplementowana w DLXJITx64.

Oraz metody chronione:

- `loadByteFromMemory()` – metoda abstrakcyjna. Wczytuje bajt z pamięci danych spod zadanego adresu. Zaimplementowana w DLXJITx64.
- `loadHalfFromMemory()` – metoda abstrakcyjna. Wczytuje poł słowa z pamięci danych spod zadanego adresu. Kolejność bajtów Big Endian. Zaimplementowana w DLXJITx64.
- `loadWordFromMemory()` – metoda abstrakcyjna. Wczytuje słowo z pamięci danych spod zadanego adresu. Kolejność bajtów Big Endian. Zaimplementowana w DLXJITx64.

Przetwarzanie pliku COD oraz obsługiwane formaty instrukcji

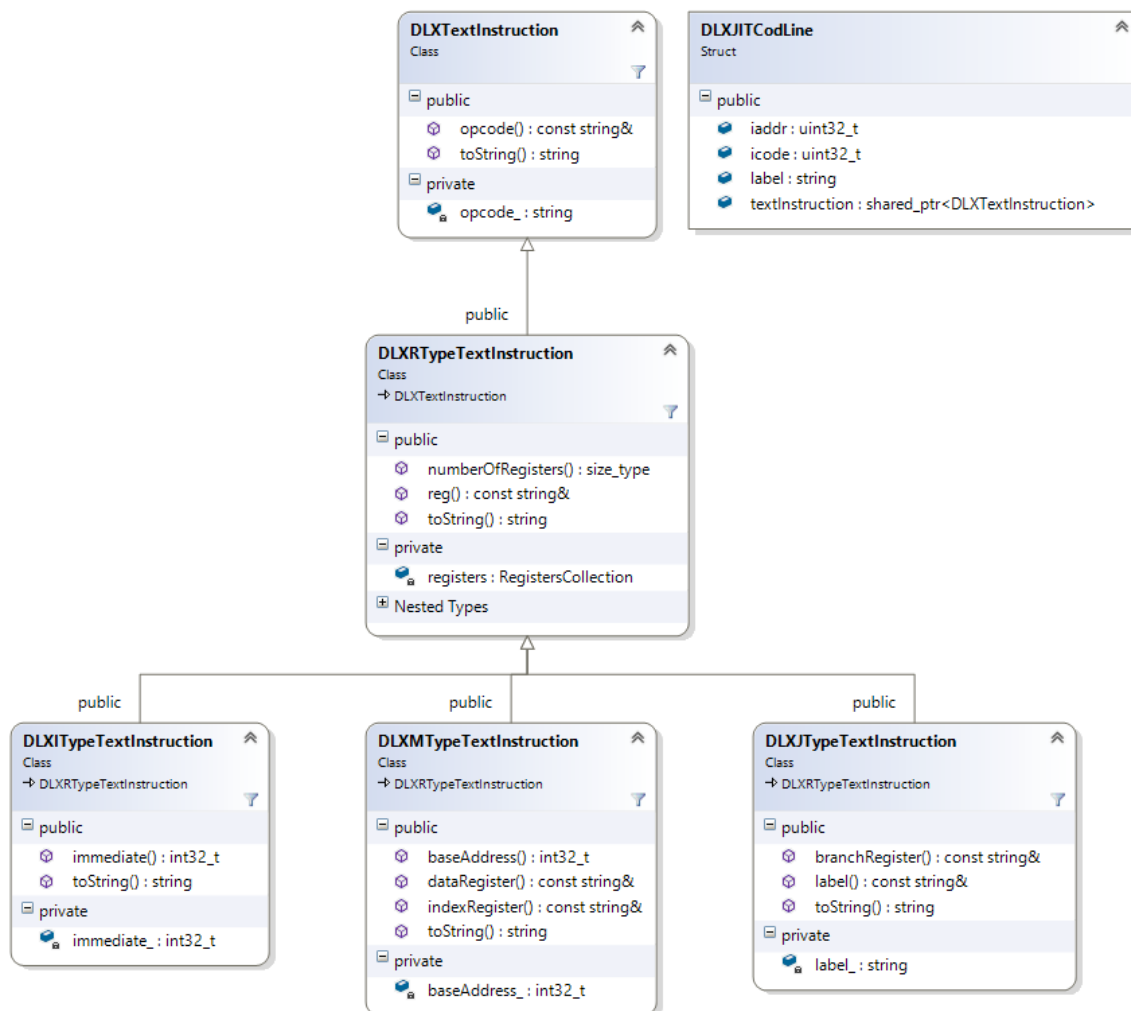
Każda linia pliku cod jest przetwarzana i przechowywana w postaci struktury `DLXCodLine` (Rysunek 2). Składa się z następujących pól:

- `iaddr` – adres instrukcji w architekturze DLX,
- `icode` – binarne kodowanie instrukcji wg. programu ESCAPE,
- `label` – etykieta dla tej instrukcji,
- `textInstruction` – wskaźnik na obiekt przechowujący zanalizowaną instrukcję, która w pliku cod jest w postaci tekstowej.

Instrukcje tekstowe są przechowywane w obiektach których hierarchię klas przedstawia Rysunek 2. W obecnej postaci program przyjmuje następujące formaty instrukcji DLX:

- `oper` – instrukcja zawierająca tylko kod operacji. Brak jakichkolwiek argumentów. Zapisywana jako `DLXTextInstruction`.
- `oper r1,r2,r3,..., rn` – instrukcja zawierająca kod operacji oraz n rejestrów. Zapisywana jako `DLXRTYPETextInstruction`.
- `oper r1,imm,r2` – instrukcja zawierająca kod operacji, rejestr, stałą natychmiastową w zapisie szesnastkowym, oraz drugi rejestr. Zapisywana jako `DLXITYPETextInstruction`.
- `oper r2,imm(r1)` – instrukcja zawierająca kod operacji, rejestr oraz pośredni adres pamięci składający się z adresu bazowego w postaci szesnastkowej oraz rejestru indeksującego. Zapisywana jako `DLXMTYPEInstruction`.
- `oper r1, etykieta` – zawiera kod operacji zawierająca rejestr oraz etykietę. Stosowane dla instrukcji skoku. Zapisywane jako `DLXJTYPEInstruction`.

W razie potrzeby listę powyższych formatów można rozszerzyć lub zmodyfikować.



Rysunek 2 Hierarchia typów instrukcji DLX w programie DLXJIT oraz struktura DLXJITCodLine

Implementacja modelu procesora DLX dla architektury IA-32e

Poniżej przedstawione koncepcje można modyfikować według uznania.

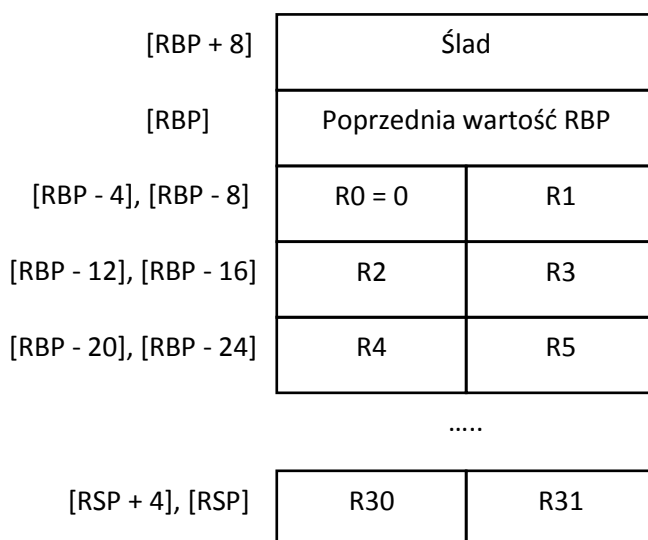
Program DLX po skompilowaniu do architektury IA-32e jest w postaci funkcji przyjmującej jako parametr wskaźnik na pamięć danych. W kodzie typ tej funkcji jest zdefiniowany następująco:

```
typedef void(*DLXProgram)(uint8_t* data_memory);
```

Wskaźnik na pamięć danych jest przechowywany w rejestrze RCX – zgodnie z konwencją wywołania funkcji dla platformy Windows na architekturze IA-32e [3]. W przypadku gdy program byłby kompilowany za pomocą kompilatora g++ dla systemu Linux wskaźnik pamięci znajdowałby się w rejestrze RDI [4].

Rejestry DLX

Rejestry architektury DLX są przechowywane na stosie w postaci 4 bajtowych liczb ze znakiem Little Endian. Obecnie kompilator zakłada 32 rejestry DLX. Ich organizację na stosie przedstawia Rysunek 3.



Rysunek 3 Organizacja rejestrów procesora DLX na stosie architektury IA-32e

W takim modelu implementacja operacji:

ADD R3, R11, R3

może wyglądać w następujący sposób:

```
movsxd rax,dword ptr [rbp-10h]
movsxd rdx,dword ptr [rbp-30h]
add rax,rdx
mov dword ptr [rbp-10h],eax
```

Pierwsze dwa rozkazy (`movsxd [1, s. 1246]`) ładują 32 bitową wartość spod wskazanego adresu i rozszerzają ją z uwzględnieniem znaku i zapisują w rejestrze 64 bitowym. Następnie zostaje wykonana operacja dodawania. Na końcu wynik znajdujący się w młodszych 32 bitach rejestru RAX, czyli w rejestrze EAX jest zapisywana do rejestru docelowego DLX, czyli na stos.

Pamięć procesora DLX

Tak jak wcześniej zostało wspomniane wskaźnik do pamięci procesora DLX znajduje się w rejestrze RCX lub RDI w zależności od tego jakim kompilatorem został skompilowany program DLXJIT. Na potrzeby dalszego opisu założymy, że jest to rejestr RCX.

Dane w pamięci procesora DLX są zapisywane i odczytywane w konwencji Big Endian, natomiast architektura IA-32e zapisuje i odczytuje dane w konwencji Little Endian. Z tego powodu podczas

operowania na pamięci DLX konieczna jest konwersja pomiędzy tymi formatami. Najlepiej jest wykorzystać do tego celu rozkaz bswap [1, s. 684] lub ror [1, s. 1643]. Implementacja operacji:

```
LDW R7, 0x0200(R1)
```

może wyglądać następująco:

```
movsxd rdx,dword ptr [rbp-8]
mov eax,dword ptr [rcx+rdx+200h]
bswap eax
mov dword ptr [rbp-20h],eax
```

W pierwszej linii ładowana jest zawartość rejestru DLX R1 do rejestru RDX. Następnie wykonywany jest odczyt z pamięci procesora DLX do rejestru EAX. Adres jest obliczany poprzez zsumowanie zawartości RCX (wskaźnika), RDX (przesunięcia) oraz stałej 200h (adres bazowy w pamięci DLX). Po odczycie kolejność bajtów jest zamieniana do formatu Little Endian i cała zawartość jest zapisywana do rejestru DLX R7.

Kodowanie rozkazów IA-32e

Legacy Prefixes	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
Grp 1, Grp 2, Grp 3, Grp 4 (optional)	(optional)	1-, 2-, or 3-byte opcode	1 byte (If required)	1 byte (If required)	Address displacement of 1, 2, or 4 bytes	Immediate data of 1, 2, or 4 bytes or none

Rysunek 4 Schemat kodowania instrukcja IA-32e

Schemat kodowanie instrukcji IA-32e został opisany w tomie 2 dokumentacji, w rozdziale 2.1 [1, s. 505]. Instrukcja może się składać z maksymalnie 6 części (Rysunek 4), są to:

- Prefik(s/y) – modyfikują one działanie instrukcji. W tym ćwiczeniu jedynym potrzebnym prefiksem będzie REX. 1 bajt każdy.
- Kod operacji – 1 do 3 bajtów. W niektórych przypadkach należy dodać do kodu operacji liczbę odpowiadającą wykorzystywanemu rejestrowi.
- Bajt ModR/M – bajt zawiera informacje o źródłowym/docelowym rejestrze lub źródłowym/docelowym miejscu w pamięci. Pozwala na adresowanie typu [rejestr + przesunięcie]. Niekiedy fragment kodu operacji jest umieszczany na polu Reg/Opcode.

- Bajt SIB – jest wykorzystywany do bardziej zaawansowanego adresowania. Pozwala na adresowanie względne za według schematu [skala * rejestr + rejestr + przesunięcie]. Pozwala na adresowanie za pomocą rejestru RSP.
- Przesunięcie – 1-4 bajtów (w zależności od typu operacji). Przesunięcie wykorzystywane w adresowaniu oraz w instrukcjach skoku.
- Stała natychmiastowa – 1-4 bajtów (w zależności od typu operacji). Stała natychmiastowa będąca parametrem niektórych instrukcji.

Prefiks REX zawiera trzy flagi:

- W – jeśli jest ustawiona na 1, oznacza to, że operacja będzie wykonywana na 64 bitowych rejestrach (szczegóły znajdują się w opisach konkretnych instrukcji).
- R – Najstarszy, dodatkowy bit dla pola reg w bajcie ModR/M.
- X – Najstarszy, dodatkowy bit dla pola index w bajcie SIB.
- B – Najstarszy, dodatkowy bit dla pola r/m w bajcie ModR/M, pola base w bajcie SIB lub pola register/opcode w bajcie ModR/M.

Bity R,X,B pozwalają na dostęp do rejestrów R8-R15 lub R8D-R15D (w zależności od flagi W).

W tomie 2 dokumentacji Intel'a podane zostały wszystkie szczegóły dotyczące kodowanie każdej instrukcji IA-32e. Informacje te są zgrupowane w tabelach. Opis oznaczeń stosowanych w tabelach znajduje się w rozdziale 3.1.1.1 w tomie 2 [1, s. 574].

Instrukcje skoku

W procesie kompilacji (lub asemblacji) generowanie instrukcji skoku jest zagadnieniem wrażliwym. W czasie przetwarzania kodu DLX należy zapamiętywać początek implementacji instrukcji na którą wskazuje etykieta, aby w odpowiednim momencie móc wykonać do niej skok. Problem pojawia się w przypadku skoku w przód. W takiej sytuacji należy za kodem operacji (2 bajty) zarezerwować 4 bajty dla przesunięcia w rozkazie skoku. Następnie po zakończonej kompilacji gdy wszystkie adresy będą już znane należy obliczyć przesunięcie skoku i uzupełnić.

Instrukcję do której ma skoczyć program koduje się przesunięciem. Przesunięcie jest to liczba która jest dodawana do obecnej wartości rejestru RIP (sam RIP wskazuje na następną instrukcję, za instrukcją skoku). Wynik tej operacji jest zapisywany w RIP. W przypadku skoków w tył przesunięcie jest liczbą ujemną, zaś w przypadku skoków w przód przesunięcie jest liczbą dodatnią.

Zadania

1. Zaimplementować wszystkie rozkazy DLX potrzebne do uruchomienia zadania z filtrem SOI z laboratorium nr 1. (4 pkt.). Przetestować program.
2. Zwiększyć efektywność generowanego kodu (+1 pkt.). Przykładowo można:

- a. Dokonać mapowania części rejestrów DLX na rejestry IA-32e. Przykładowo R1-R8 z architektury DLX na R8-R15 architektury IA-32e. Optymalizacja operacji na R0 - zastąpienie go stałą 0.
- b. Usunięcie niepotrzebnych sekwencji rozkazów (lub ich minimalizacja). Przykładowo dla sekwencji:

```
MUL R9, R10, R11
```

```
ADD R3, R11, R3
```

Zamiast generować następujący kod:

```
movsxd rax,dword ptr [rbp-28h]
```

```
movsxd rdx,dword ptr [rbp-2Ch]
```

```
imul rax,rdx
```

```
mov dword ptr [rbp-30h],eax
```

```
movsxd rax,dword ptr [rbp-10h]
```

```
movsxd rdx,dword ptr [rbp-30h]
```

```
add rax,rdx
```

```
mov dword ptr [rbp-10h],eax
```

można opuścić ponowny odczyt R11, oraz zamienić kolejność parametrów w rozkazie add.

```
movsxd rax,dword ptr [rbp-28h]
```

```
movsxd rdx,dword ptr [rbp-2Ch]
```

```
imul rax,rdx
```

```
mov dword ptr [rbp-30h],eax
```

```
movsxd rdx,dword ptr [rbp-10h]
```

```
add rdx,rax
```

```
mov dword ptr [rbp-10h],edx
```

- c. Analiza zależności pomiędzy instrukcjami DLX i próba optymalizacji kodu na tej postawie poprzez usuwanie niepotrzebnych sekwencji i próby wykorzystania cech architektury IA-32e nieobecnych w architekturze DLX.

Przykładowo dla rozkazów:

```
SUBI R5, 0x003C, R11
```

```
BRLE R11, endif1
```

Zamiast generować kod:

```
movsxd rax,dword ptr [rbp-18h]
```

```
sub rax,3Ch
```

```
mov dword ptr [rbp-30h],eax
```

```
movsxd rax,dword ptr [rbp-30h]
```

```
cmp rax,0
```



```
jle endif1
```

można wygenerować, korzystając z właściwości rozkazu cmp:

```
cmp dword ptr [rbp-18h],3Ch
```

```
jle endif1
```

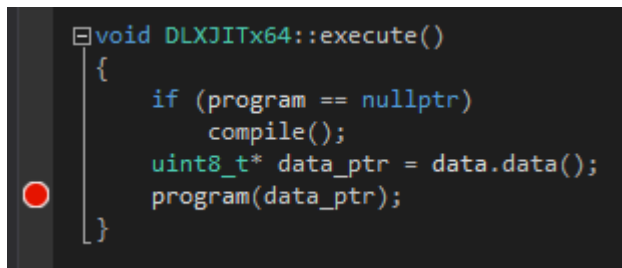
przy założeniu, że R11 nie będzie potem wykorzystywany jako rejestr źródłowy.

Podpowiedzi

Debug

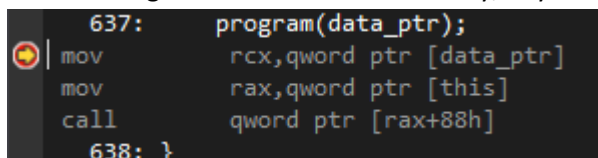
Aby łatwo przeanalizować, czy program DLXJIT wyprodukował poprawny kod asemblera należy:

1. Należy ustawić breakpoint w metodzie execute(), w linii w której ma zostać uruchomiony kod wygenerowany:



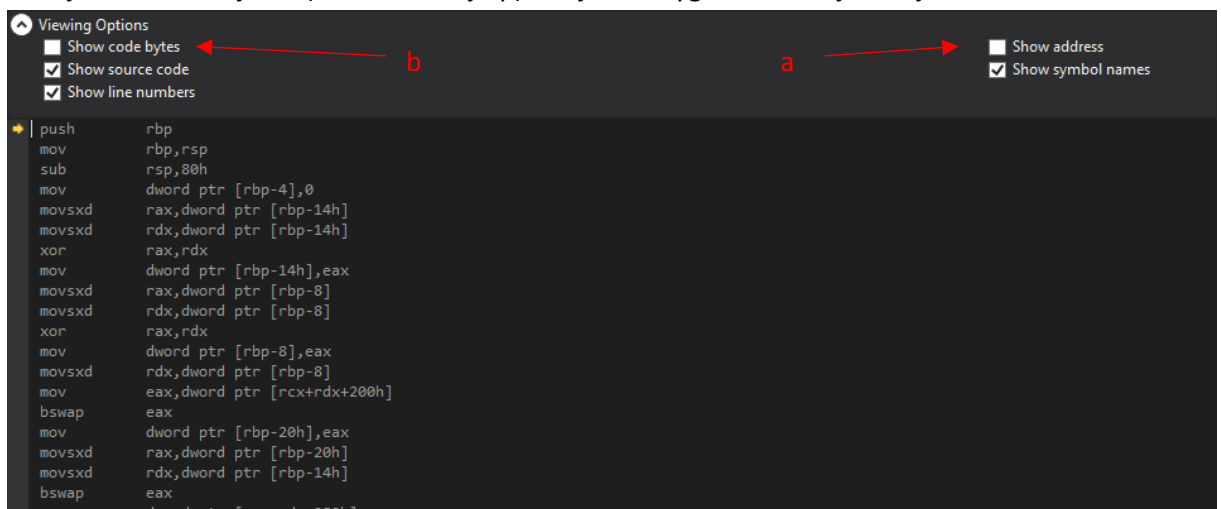
```
void DLXJITx64::execute()
{
    if (program == nullptr)
        compile();
    uint8_t* data_ptr = data.data();
    program(data_ptr);
}
```

2. Uruchomić program. Poczekać na breakpoint i następnie wcisnąć Ctrl+Alt+D lub wybrać z menu Debug -> Windows -> Disassembly, aby otworzyć okno z instrukcjami asemblera.



```
637:  program(data_ptr);
mov     rcx,qword ptr [data_ptr]
mov     rax,qword ptr [this]
call    qword ptr [rax+88h]
638: }
```

3. Podejść do instrukcji call (ewentualnie jmp) i wejść do wygenerowanej funkcji.



Viewing Options

- ☐ Show code bytes
- ☒ Show source code
- ☒ Show line numbers
- ☐ Show address
- ☒ Show symbol names

```
push    rbp
mov     rbp, rsp
sub     rsp, 80h
mov     dword ptr [rbp-4], 0
movsxd  rax, dword ptr [rbp-14h]
movsxd  rdx, dword ptr [rbp-14h]
xor     rax, rdx
mov     dword ptr [rbp-14h], eax
movsxd  rax, dword ptr [rbp-8]
movsxd  rdx, dword ptr [rbp-8]
xor     rax, rdx
mov     dword ptr [rbp-8], eax
movsxd  rdx, dword ptr [rbp-8]
mov     eax, dword ptr [rcx+rdx+200h]
bswap   eax
mov     dword ptr [rbp-20h], eax
movsxd  rax, dword ptr [rbp-20h]
movsxd  rdx, dword ptr [rbp-14h]
bswap   eax
```

4. W tym momencie można zweryfikować czy kod wygenerowany przez program jest zgodny z zamierzeniami.

Dodatkowo można włączyć pokazywanie adresu instrukcji (a) lub wyświetlanie kodu szesnastkowego instrukcji (b). Pierwsza funkcja może przydać się podczas lokalizowania początku instrukcji DLX w kodzie IA-32e – JIT wyświetla odpowiednie adresy po skompilowaniu programu DLX (Rysunek 5). Druga funkcja pozwoli zweryfikować kodowanie instrukcji.

```

XOR      R4, R4, R4:      0x262c0016
XOR      R1, R1, R1:      0x262c002d
for1:    LDW      R7, 0x200(R1): 0x262c0044
        STW      R7, 0x280(R4): 0x262c005a
XOR      R3, R3, R3:      0x262c0071
AND      R4, R4, R5:      0x262c0088
XOR      R2, R2, R2:      0x262c009f
for2:    LDW      R9, 0x280(R5): 0x262c00b6
        LDW      R10, 0x2c0(R2): 0x262c00cc
XOR      R8, R8, R8:      0x262c00e0

```

Rysunek 5 Adresy w których rozpoczyna się implementacja poszczególnych instrukcji DLX

Kodowanie instrukcji

W przypadku problemów z implementacją poprawnego kodowania instrukcji można skorzystać z pomocy asemblera. W tym celu można dodać plik asm do programu implementujący funkcję. W niej można zapisać wybrane instrukcje, a następnie powyższą metodą sprawdzić sposób ich kodowania.

Bibliografia

- [1] „Intel® 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4”. Intel, mar-2017.
- [2] Peter Verplaetse, „ESCAPE v1.1 Manual”. Department of Electronics and Information Systems University of Ghent.
- [3] „MSDN: Calling Convention”. [Online]. Dostępne na: [https://msdn.microsoft.com/en-us/library/9b372w95\(v=vs.120\).aspx](https://msdn.microsoft.com/en-us/library/9b372w95(v=vs.120).aspx). [Udostępniono: 16-maj-2017].
- [4] Michael Matz, Jan Hubička, Andreas Jaeger, i Mark Mitchell, „System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99.7”. 17-lis-2014.