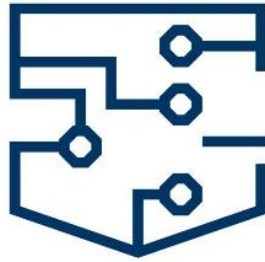


Zaawansowane Architektury Komputerów



Laboratorium 5 – JIT dla architektury DLX, implementacja na procesorze ARM

Tomasz Bieliński

2024-05-09



Wstęp

Celem tego ćwiczenia laboratoryjnego, analogicznie jak w ćwiczeniu poprzednim, będzie implementacja prostego kompilatora JIT (ang. Just in Time). Tym razem jednak docelową architekturą będzie 32 bitowa architektura ARMv7 [1]. Do instrukcji został załączony projekt dla IDE Visual Studio [2] w którym zostały zaimplementowane operacje wczytywania kodu DLX, wczytywania stanu pamięci i zapisywanie stanu pamięci zgodnie z formatem symulatora Escape [3] oraz część funkcjonalności kompilatora JIT. Visual Studio pozwala na uruchamianie oraz debugowanie programów zdalnie na innej maszynie poprzez ssh. Mechanizm ten zostanie wykorzystany aby połączyć się z emulatorem qemu-arm [4] (więcej szczegółów na temat środowiska laboratoryjnego znajduje się w załączniku na stronie 13). Na maszynie są dwaj użytkownicy **debian** (hasło: **debian**) pod którym Visual Studio kompiluje i uruchamia program DLXJIT oraz **root** (hasło: **root**) do ewentualnego zarządzania maszyną (np. jej wyłączenia).

Architektura ARMv7

Jest to 32 bitowa architektura zaprojektowana przez firmę ARM. W tej architekturze dla programisty jest dostępnych m.in. 16 rejestrów, każdy po 32 bitów. Rejestry R0 do R12 są rejestrami ogólnego przeznaczenia. Rejestry R13 do R15 są rejestrami specjalnymi, które nazywane są odpowiednio: **SP**, **LR**, **PC**. Pełnią one następujące role [1, Rozdz. A2.3]:

- SP – wskaźnik stosu,
- LR – Link Register, przechowuje on adres powrotu z wywoływanej procedury,
- PC – licznik instrukcji – odpowiednik RIP w architekturze IA-32e

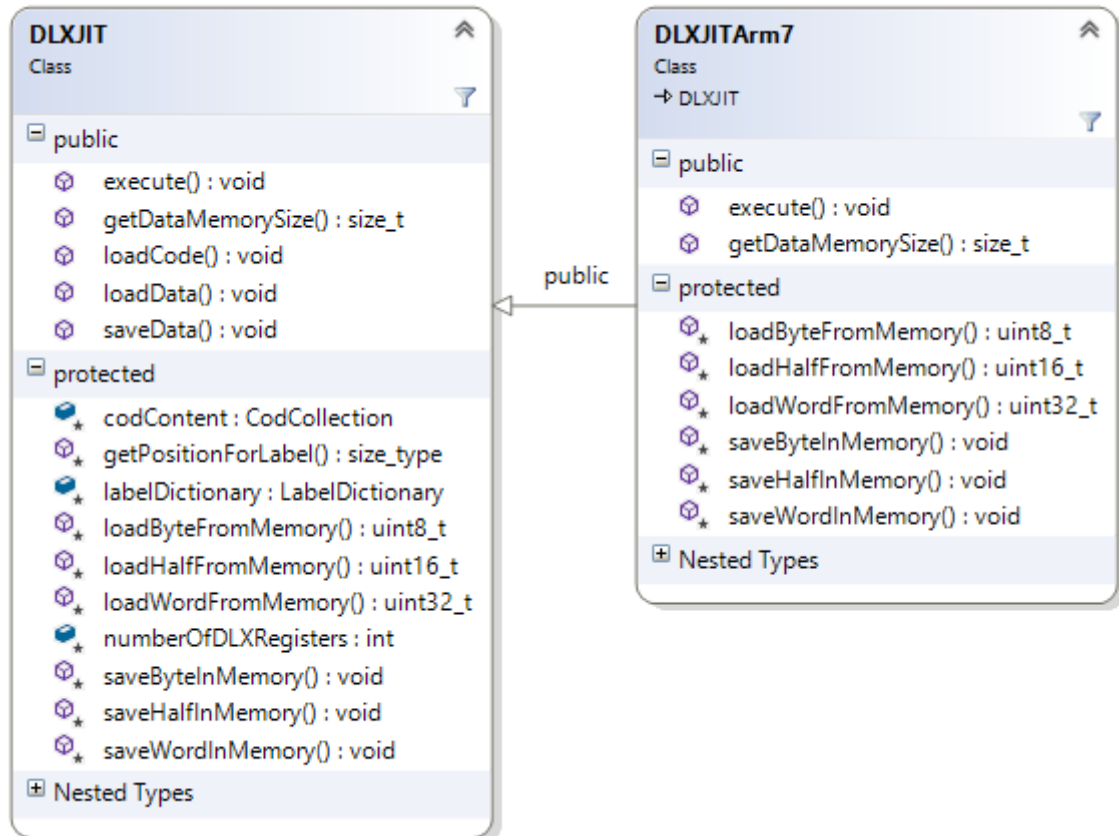
W architekturze ARMv7 występuje stos na którego wierzchołek wskazuje rejestr SP. Stos analogicznie jak w architekturze IA-32e rośnie w dół.

W architekturze ARMv7 jest dostępnych kilka zestawów instrukcji z pośród których najważniejsze są [1, Rozdz. A1.2, A4.1]:

- ARM - podstawowy zestaw instrukcji. Każda z instrukcji ma rozmiar 4 bajtów,
- Thumb – podzbiór ARM. Większość instrukcji kodowana jest na 2 bajtach. Po pojawieniu się rozszerzenia Thumb 2 niektóre dodatkowe instrukcje kodowane są na 4 bajtach. Pozwala na zmniejszenie rozmiaru kodu, kosztem elastyczności samych instrukcji.

Dostarczony program DLXJIT generuje instrukcje ARM.

Struktura programu DLXJIT

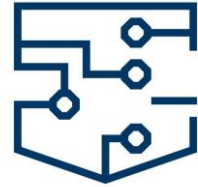


Rysunek 1 Klasa abstrakcyjna DLXJIT zawierająca funkcjonalności niezależne od architektury docelowej oraz klasa DLXJITArm7 zawierająca implementację kompilatora JIT dla architektury ARMv7

Głównymi klasami programu są DLXJIT oraz DLXJITArm7 (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 DLXJITArm7.
- `getDataMemorySize()` – metoda abstrakcyjna. Podaje rozmiar pamięci danych. Zaimplementowana w DLXJITArm7.

Oraz metody chronione:



- `getPositionForLabel()` – pobiera adres instrukcji DLX na którą wskazują etykieta kodu. Wykorzystuje chronione pole `labelDictionary`.
- `loadByteFromMemory()` – metoda abstrakcyjna. Wczytuje bajt z pamięci danych spod zadanego adresu. Zaimplementowana w `DLXJITArm7`.
- `saveByteInMemory()` – metoda abstrakcyjna. Zapisuje bajt do pamięci danych pod zadany adres. Zaimplementowana w `DLXJITArm7`.
- `loadHalfFromMemory()` – metoda abstrakcyjna. Wczytuje pół słowa z pamięci danych spod zadanego adresu. Kolejność bajtów Big Endian. Zaimplementowana w `DLXJITArm7`.
- `saveHalfInMemory()` – metoda abstrakcyjna. Zapisuje pół słowa do pamięci danych pod zadany adres. Kolejność bajtów Big Endian. Zaimplementowana w `DLXJITArm7`.
- `loadWordFromMemory()` - metoda abstrakcyjna. Wczytuje słowo z pamięci danych spod zadanego adresu. Kolejność bajtów Big Endian. Zaimplementowana w `DLXJITArm7`.
- `saveWordInMemory()` - metoda abstrakcyjna. Zapisuje słowo do pamięci danych pod zadany adres. Kolejność bajtów Big Endian. Zaimplementowana w `DLXJITArm7`.

Przetwarzanie pliku COD oraz obsługiwane formaty instrukcji

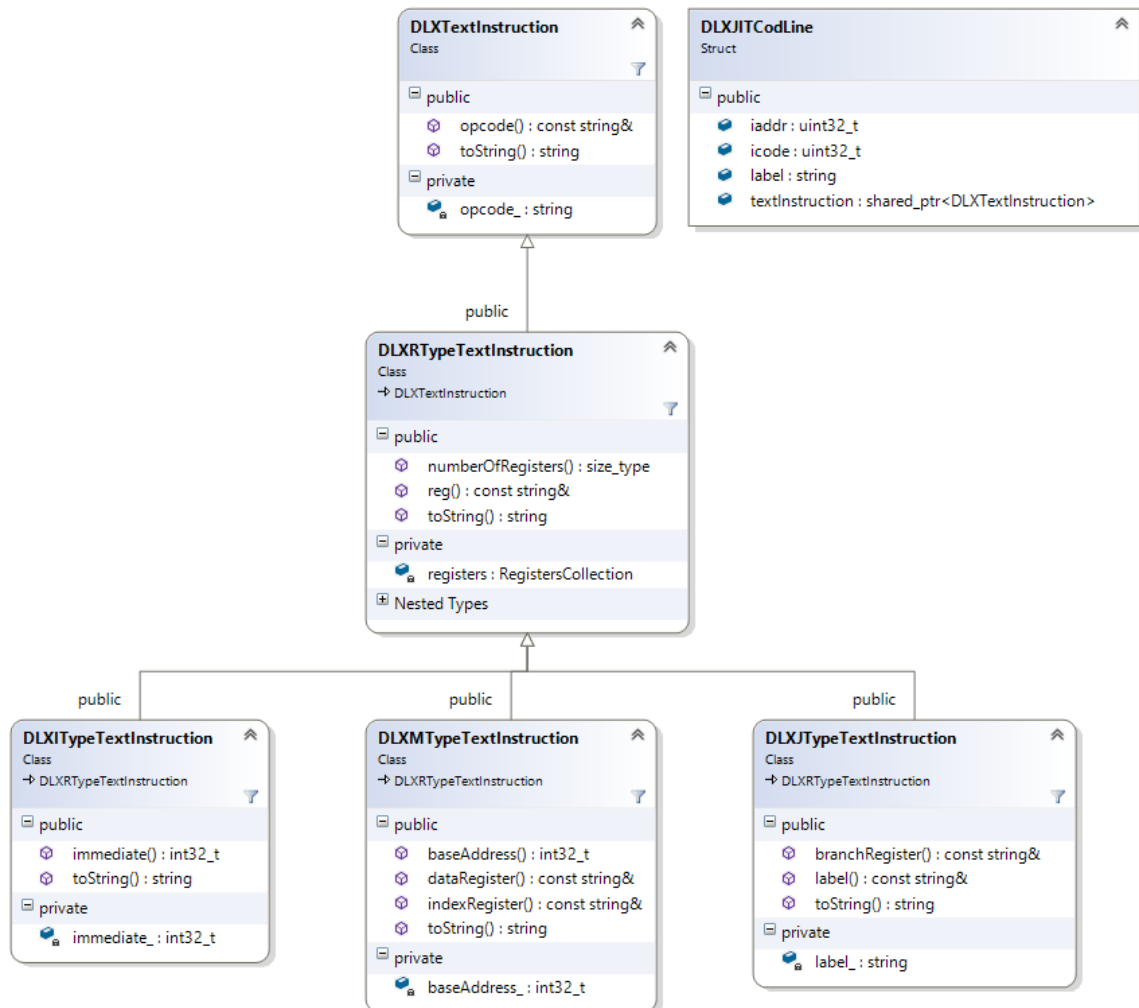
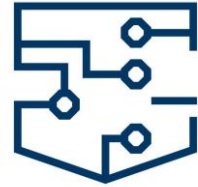
Każda linia pliku cod jest przetwarzana i przechowywana w postaci struktury `DLXCodLine` (Rysunek 2). Składa się z następujących pól:

- `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 zawierający 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`

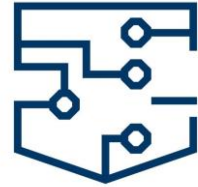
Implementacja modelu procesora DLX dla architektury ARMv7

Poniżej przedstawione koncepcje można modyfikować według uznania.

Analogicznie jak w ćwiczeniu dla architektury IA-32e program DLX po skompilowaniu do rozkazów architektury ARMv7 jest w postaci funkcji przyjmującej jako parametr wskaźnik do pamięci danych. Typ tej funkcji jest zdefiniowany następująco:

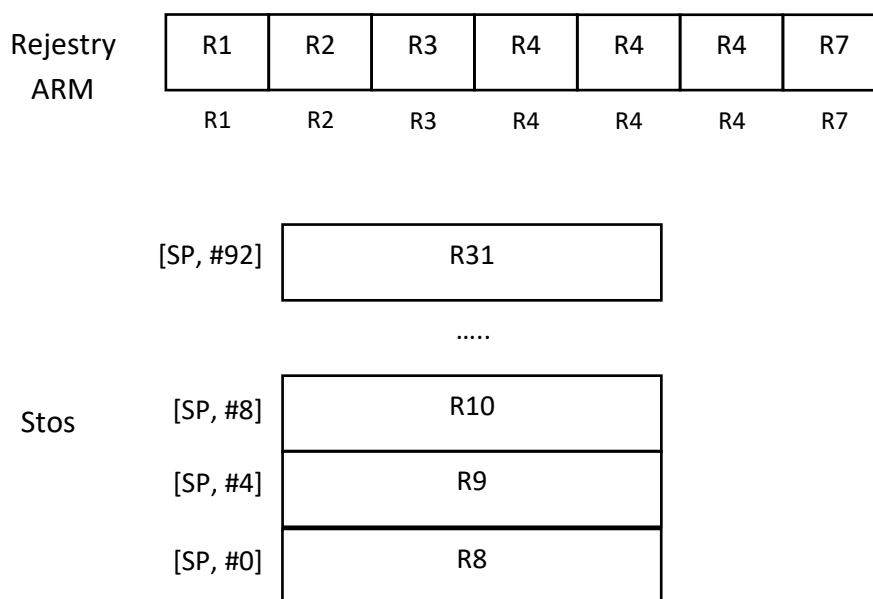
```
typedef void(*DLXProgram)(uint8_t* data_memory);
```

Wskaźnik pamięci danych jest przechowywany w rejestrze R0 zgodnie z standardem wywołań procedur dla architektury ARM [5].



Rejestry DLX

Rejestry DLX od R1 do R7 są mapowane na rejestry R1 do R7 architektury ARM, natomiast rejestry R8 do R31 przechowywane są na stosie w postaci 4 bajtowych liczb ze znakiem w formacie Little Endian. Rejestr R0 jako rejestr źródłowy jest zastępowany stałą 0, zaś gdy jest rejestrem docelowym instrukcja jest pomijana. Obecnie kompilator zakłada 32 rejestry DLX. Ich organizację przedstawia Rysunek 3.



Rysunek 3 Organizacja rejestrów DLX na stosie i w banku rejestrów architektury ARM

Rejestry ARM R8 oraz R9 służą jako rejestry tymczasowe. Do nich jest ładowana zawartość rejestrów DLX przechowywanych na stosie. Rejestr R10 służy jako rejestr tymczasowy dla rejestrów docelowych DLX które są przechowywane na stosie. Rejestry R11 oraz R12 nie są wykorzystywane ze względu na problemy z debuggerem¹.

W takim modelu implementacja operacji:

ADD R3, R11, R3

¹ Po wpisaniu do R11 wartości nie będącej poprawnym adresem w pamięci debugger gdb w dostarczonej wersji maszyny wirtualnej Debian wyłączał się wyświetlając błąd. Prawdopodobnie R11 i/lub R12 jest traktowany jako wskaźnik ramki stosu (analogicznie do RBP) i nieprawidłowa wartość powoduje awarię debugera. Program nienadzorowany używający rejestrów R11 oraz R12 wykonuje się poprawnie.

może wyglądać następująco:

```
ldr r9, [sp, #12]
add r3, r9, r3
```

Zaś następująca instrukcja DLX:

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

może zostać skompilowana następująco:

```
ldr r8, [sp, #4]
ldr r9, [sp, #8]
mul r10, r8, r9
str r10, [sp, #12]
```

Operacje na rejestrach DLX są implementowane z pomocą następujących metod w klasie DLXJITArm7:

- `loadDLXRegister()` – jeżeli rejestr zawartość rejestru DLX jest przechowywana w rejestrach ARM metoda ta zwraca odpowiedni rejestr ARM, jeśli rejestr przechowywany jest na stosie koduje ładowanie jego zawartości do tymczasowego rejestru ARM (R8 lub R9) i zwraca go. Jeśli został użyty rejestr DLX R0 to jest kodowane wpisywanie do rejestru tymczasowego wartości 0.
- `getRegisterForTargetDLXRegister()` – zamienia numer docelowego rejestru DLX na docelowy rejestr ARM, jeżeli docelowy rejestr DLX znajduje się na stosie zwraca rejestr R10.
- `storeTargetDLXRegister()` – jeżeli docelowy rejestr DLX jest implementowany jako rejestr ARM metoda ta nic nie robi. W innym wypadku koduje zapisywanie zawartości R10 do odpowiedniego miejsca na stosie.

Pamięć procesora DLX

Tak jak wcześniej zostało wspomniane wskaźnik do pamięci procesora DLX znajduje się w rejestrze R0.

Dane w pamięci procesora DLX są zapisywane i odczytywane w konwencji Big Endian, natomiast architektura ARMv7 domyślnie 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 **rev** [1, Rozdz. A8.8.145].

Implementacja operacji:

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

może wyglądać następująco:

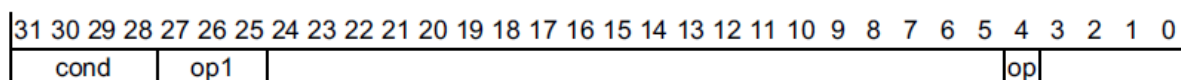
```
add r8, r0, r1
ldr r7, [r8, #512] ;0x200
```

rev r7, r7

W pierwszej linii sumowany jest adres pamięci danych (r0) z zawartością rejestru DLX R1 (który jest implementowany jako rejestr r1 w architekturze ARM). Następnie wykonywany jest odczyt z pamięci procesora DLX do docelowego rejestru R7. W instrukcji ldr jest podawane stałe przesunięcie które jest zakodowane w instrukcji LDW (0x200). Po odczycie kolejność bajtów jest zamieniana do formatu Little Endian.

Kodowanie rozkazów ARMv7

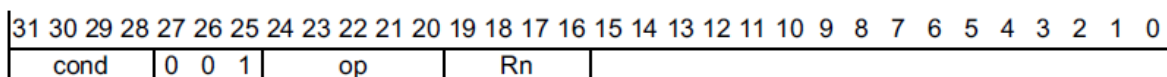
Rozkazy w architekturze ARM posiadają dwa rodzaje kodowania: podstawowy - ARM oraz skrócony Thumb. W tym rozdziale zostanie opisane kodowanie instrukcji ARM. W przypadku architektury IA-32e kod instrukcji jest podzielony na bajty, z których każdy pełni odpowiednią funkcję. Natomiast w architekturze ARM w kodowaniu podstawowym instrukcja jest 4 bajtowym polem bitowym w którym w zależności od typu instrukcji kolejne bity pełni inne funkcję.



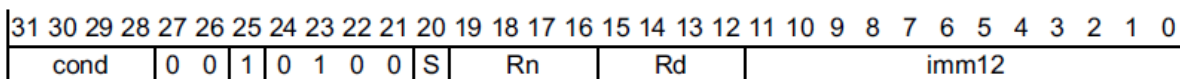
Rysunek 4 Ogólny schemat kodowania instrukcji ARM

Rysunek 4 przedstawia ogólny schemat kodowania instrukcji ARM. Pole **cond** zawiera warunek wykonania instrukcji. Dla instrukcji bezwarunkowych to pole ma wartość 1110 (AL - Always). Jeśli jednak istnieje konieczność warunkowego wykonania instrukcji (np. warunkowego skoku), to pole należy ustawić na inną wartość (zob. [1, Rozdz. A8.3]). Warunek jest obliczany na podstawie zawartości rejestru Application Program Status Register (APSR) [1, Rozdz. A2.4]. Wartość 1111 ma specjalne znaczenie i jest stosowana do kodowania innych instrukcji. Pola **op1** oraz **op** określają poszczególne grupy instrukcji (szczegóły patrz [1, Rozdz. A5.1]). Podział na grupy i podgrupy jest uszczegółowiony w dalszej części dokumentacji ARM.

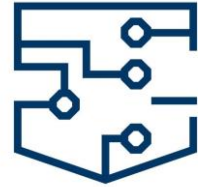
Dla przykładu zostanie przeanalizowane kodowanie instrukcji **add** w wersji immediate - dodającej do rejestru wartość stałą. Znajduje się ona w grupie instrukcji „Data-processing (immediate)” [1, Rozdz. A5.2.3].



Rysunek 5 Ogólny schemat kodowania instrukcji typu Data-processing (immediate)



Rysunek 6 Schemat kodowania instrukcji ADD (immediate)



Rysunek 5 przedstawia ogólny schemat kodowania instrukcji typu Data-processing (immediate), natomiast Rysunek 6 szczegółowe kodowanie dla instrukcji **add** (immediate). Na podstawie tych informacji można zaprojektować strukturę w języku C++ zawierającą pola bitowe opisujące instrukcje Data-processing (immediate). Struktura ta w programie DLXJIT wygląda następująco:

```
struct DataProcessingImmediateInstruction
{
    uint16_t details : 16;
    uint8_t Rn : 4;
    uint8_t op : 5;
    bool b25 : 1;
    bool b26 : 1;
    bool b27 : 1;
    Condition cond : 4;
}__attribute__((__packed__));
```

Jak widać w powyższym przykładzie pola odpowiadające najmłodszym bitom znajdują się na początku struktury, zaś pola odpowiadające najstarszym bitom na końcu struktury. Kolejność ta jest zależna od kompilatora, architektury procesora, a w szczególności od tego czy procesor pracuje w trybie Little Endian, czy Big Endian. Kształt powyższej struktury jest kompromisem między tym co przedstawia Rysunek 4, Rysunek 5, Rysunek 6 oraz kodowaniem innych instrukcji typu Data-processing (immediate), które zostały wykorzystane w programie. Atrybut `__packed__` informuje kompilator, że pola struktury mają być jak najciaśniej upakowane. Na podstawie tego co przedstawia Rysunek 6 oraz opisu instrukcji ADD (immediate) [1, Rozdz. A8.8.5] procedura kodująca rozkaz wygląda następująco:

```
void DLXJITArm7::writeAdd(Condition cond, bool updateFlags,
    Register dest, Register src1, int16_t imm) {
    DataProcessingImmediateInstruction instr;
    instr.cond = cond;
    instr.b27 = false;
    instr.b26 = false;
    instr.b25 = true;
    instr.op = 0x4 << 1 | (updateFlags? 1 : 0);
    instr.Rn = src1;
    instr.details = dest << 12 | (imm & 0xFFF);
    serialize(rawCode, instr);
}
```

Wykonywane są następujące czynności:

- Na bitach 31:28 wpisywany jest rodzaj warunku,
- Bity 27:25 ustawiane na wartości 001 (patrz Rysunek 5 lub Rysunek 6),



- Bity 24:20 z pola op (patrz Rysunek 5) ustawiane są na 01000 lub 01001 w zależności od tego czy instrukcja ma wpływać na flagi w APSR,
- Do bitów 19:16 z pola Rn wpisywany jest numer rejestru źródłowego,
- Na bitach 15:12 (fragment pola details) zapisywany jest numer rejestru docelowego (Rd),
- Bity 11:0 (fragment pola details) zawierają 12 bitową stałą natychmiastową.

Np. rozkaz:

`add r10, r5, #60`

zostanie zakodowany w następujący sposób:

- Bity 31:28 zostaną ustawione na 1110 (AL) – rozkaz zostanie wykonany bezwarunkowo,
- Bity 27:25 ustawiane na wartości 001,
- Bity 24:20 zostaną ustawione na wartość 01000 – rozkaz nie zmienia wartości flag,
- Bity 19:16 zostaną ustawione na 0101 (rejestr R5 jako rejestr źródłowy),
- Bity 15:12 zostaną ustawione na 1010 (rejestr R10 jako rejestr docelowy),
- Bity 11:0 zostaną ustawione na 0000 0011 1100 (stała natychmiastowa to 60, czyli 0x3c).

czyli:

1110 001 01000 0101 1010 0000 0011 1100

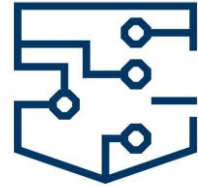
albo:

0xE285A03C

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 najlepiej w kodzie rozkazu **b** wpisać offset 0. Następnie po zakończonej kompilacji gdy wszystkie adresy będą już znane należy obliczyć przesunięcie skoku i je uzupełnić.

Instrukcję do której ma skoczyć program koduje się przesunięciem. Przesunięcie jest to liczba która jest dodawana do obecnej wartości rejestru PC (sam PC w trybie ARM wskazuje na 2 instrukcje, za instrukcją skoku [1, Rozdz. A2.3]). Wynik tej operacji jest zapisywany w PC. 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.

Aby zweryfikować, czy program wykonuje poprawne obliczenia należy podejrzeć plik ze stanem pamięci DLX który znajduje się na maszynie wirtualnej. Zalogować się na maszynę wirtualną (przez konsolę serial0 lub ssh) na użytkownika **debian** i przejść do katalogu

`~/projects/Lab5_vs`

Tam za pomocą programu **less** podejrzeć plik **out.dat**.

2. Zwiększyć efektywność generowanego kodu (+1 pkt.). Przykładowo można:
 - a. Usunąć niepotrzebne sekwencje rozkazów (lub je zminimalizować). Przykładowo dla sekwencji:

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

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

Zamiast generować następujący kod:

```
ldr r8, [sp, #4]
```

```
ldr r9, [sp, #8]
```

```
mul r10, r8, r9
```

```
str r10, [sp, #12]
```

```
ldr r9, [sp, #12]
```

```
add r3, r9, r3
```

można opuścić ponowny odczyt R11 (zapisanego pod [sp, #12]), oraz zmienić a rozkazie **add** rejestr R9 na R10.

```
ldr r8, [sp, #4]
```

```
ldr r9, [sp, #8]
```

```
mul r10, r8, r9
```

```
str r10, [sp, #12]
```

```
add r3, r10, r3
```

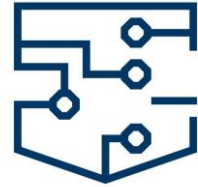
- b. Analizować zależności pomiędzy instrukcjami DLX i próbować optymalizować kodu na tej postawie poprzez usuwanie niepotrzebnych sekwencji i próby wykorzystania cech architektury ARMv7 nieobecnych w architekturze DLX.

Przykładowo dla rozkazów:

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

```
BRLE  R11, endif1
```

Zamiast generować kod:

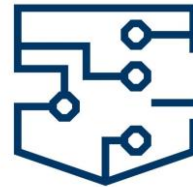


```
sub r10, r5, #60
str r10, [sp, #12]
ldr r8, [sp, #12]
movs r8, r8
ble endif
```

można wygenerować, korzystając z właściwości rozkazu **subs** który modyfikuje flagi (**str** ich nie modyfikuje):

```
subs r10, r5, #60
str r10, [sp, #12]
ble endif
```

- c. Zmienić kodowanie instrukcji z ARM na Thumb. Wymaga to przepisania kodu DLXJIT generującego kod maszynowy poszczególnych instrukcji łącznie z strukturami opisującymi podział kodowania instrukcji na pola bitowe. Aby skompilowany kod uruchomić jako funkcję z kodem typu Thumb należy dodać do adresu zapisanego w polu program 1 (czyli ustawić najmłodszy bit na 1). Jest to właściwość instrukcji BLX, którą kompilator używa aby wywołać funkcję przez wskaźnik [1, Rozdz. A8.8.26], szczegółowe działanie instrukcji BLX jest opisane w procedurze BXWritePC() [1, Rozdz. A2.3.2].








Podpowiedzi

Uruchamianie debiana dla ARM za pomocą QEMU

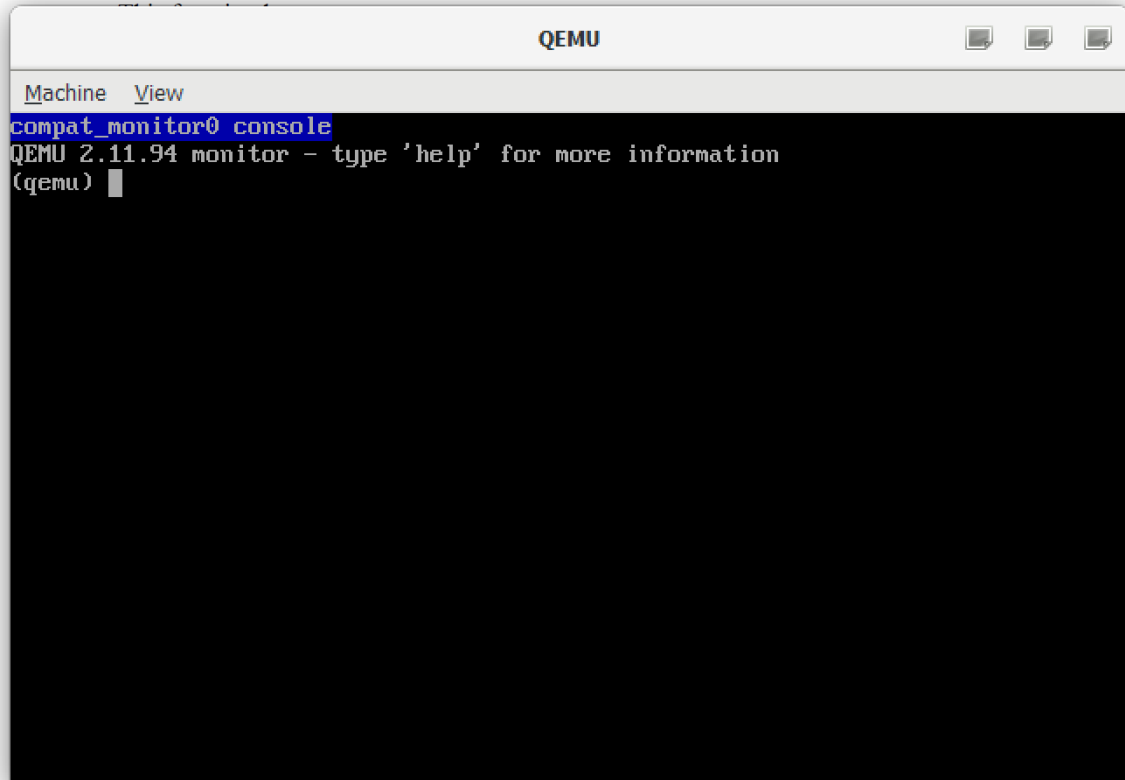
Na potrzeby tego laboratorium została przygotowana maszyna wirtualna z systemem operacyjnym Debian 8 dla architektury ARM. Uruchamiana jest ona w emulatorze qemu-arm, który dostarczony jest wraz z maszyną wirtualną.

Należy rozpakować plik debian_arm.zip w wybranym przez siebie katalogu, następnie przejść do niego i wejść do katalogu debian_arm i uruchomić plik run_machie.bat (Rysunek 7).

Nazwa	Data modyfikacji	Typ	Rozmiar
 debian.qcow	2018-04-23 15:55	Plik QCOW	1 165 360 KB
 initrd.img-3.16.0-4-armmp-lpae	2018-04-19 15:30	Plik 0-4-ARMMP-L...	12 540 KB
 run_machie.bat	2018-04-19 19:10	Plik wsadowy Wind...	1 KB
 run_machine.sh	2018-04-19 15:30	Plik SH	1 KB
 vmlinuz-3.16.0-4-armmp-lpae	2018-04-19 15:30	Plik 0-4-ARMMP-L...	3 195 KB

Rysunek 7

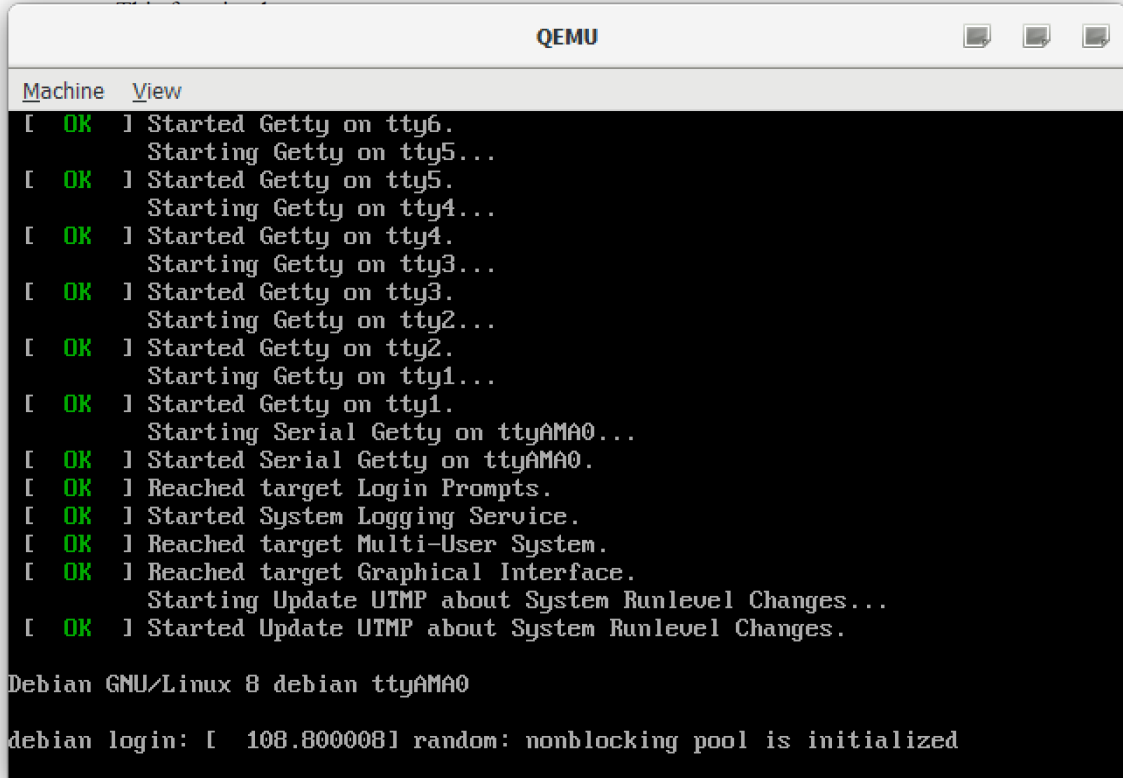
Powinno się otworzyć okno qemu (Rysunek 8).



BranchWritePC(bits(32) address)
if CurrentInstrSat() == InstrSat_ARM then

Rysunek 8 Okno QEMU

Aby przełączyć się na konsolę serial0 i zobaczyć proces ładowania systemu operacyjnego Debian oraz ewentualnie zalogować się na konsolę należy wcisnąć Ctrl+Alt+2.



```

Machine  View
[ OK ] Started Getty on tty6.
        Starting Getty on tty5...
[ OK ] Started Getty on tty5.
        Starting Getty on tty4...
[ OK ] Started Getty on tty4.
        Starting Getty on tty3...
[ OK ] Started Getty on tty3.
        Starting Getty on tty2...
[ OK ] Started Getty on tty2.
        Starting Getty on tty1...
[ OK ] Started Getty on tty1.
        Starting Serial Getty on ttyAMA0...
[ OK ] Started Serial Getty on ttyAMA0.
[ OK ] Reached target Login Prompts.
[ OK ] Started System Logging Service.
[ OK ] Reached target Multi-User System.
[ OK ] Reached target Graphical Interface.
        Starting Update UTMP about System Runlevel Changes...
[ OK ] Started Update UTMP about System Runlevel Changes.

Debian GNU/Linux 8 debian ttyAMA0

debian login: [ 108.800008] random: nonblocking pool is initialized

```

BranchWritePC(bits(32) address)
if CurrentInstrSat() == InstrSat_ARM then

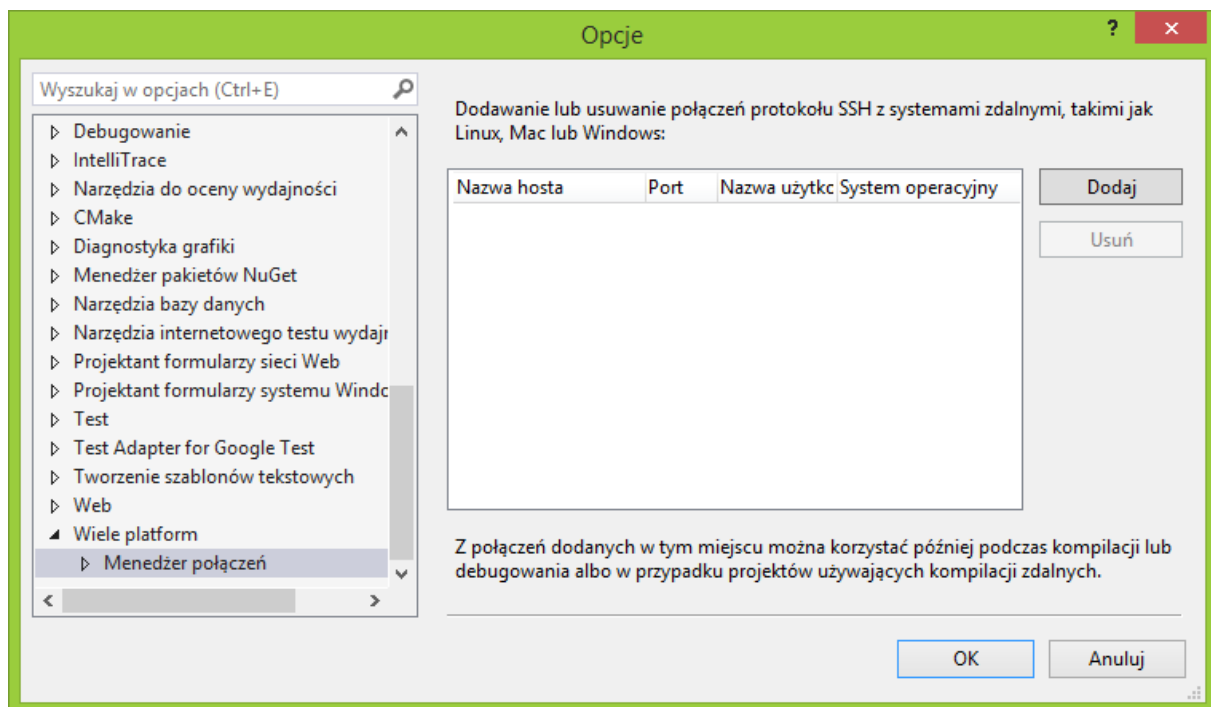
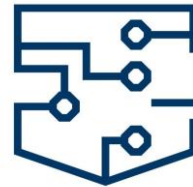
Rysunek 9 Konsola serial0

Na maszynę wirtualną można zalogować się również przez ssh, np. za pomocą programu putty łącząc się z adresem localhost:2222 (logowanie na użytkownika root nie działa w ten sposób).

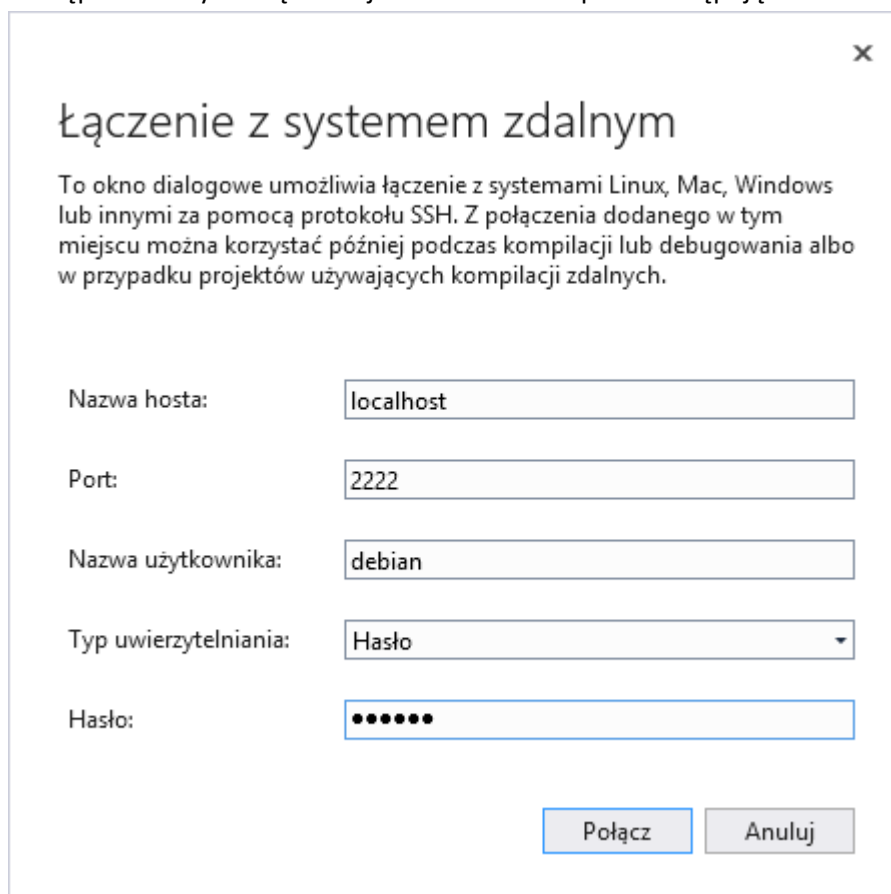
Po uruchomieniu maszyna wirtualna powinna być gotowa do współpracy z Visual Studio.

Konfiguracja Visual Studio

1. Aby dodać połączenie z maszyną wirtualną ARM należy w Visual Studio 2017 otworzyć Menadżer Połączeń wybierając Narzędzia->Opcje, następnie Wiele Platform->Menadżer Połączeń



2. Następnie należy kliknąć dodaj i w formularzu wpisać następujące dane



Łączenie z systemem zdalnym

To okno dialogowe umożliwia łączenie z systemami Linux, Mac, Windows lub innymi za pomocą protokołu SSH. Z połączenia dodanego w tym miejscu można korzystać później podczas kompilacji lub debugowania albo w przypadku projektów używających kompilacji zdalnych.

Nazwa hosta: localhost

Port: 2222

Nazwa użytkownika: debian

Typ uwierzytelniania: Hasło

Hasło:

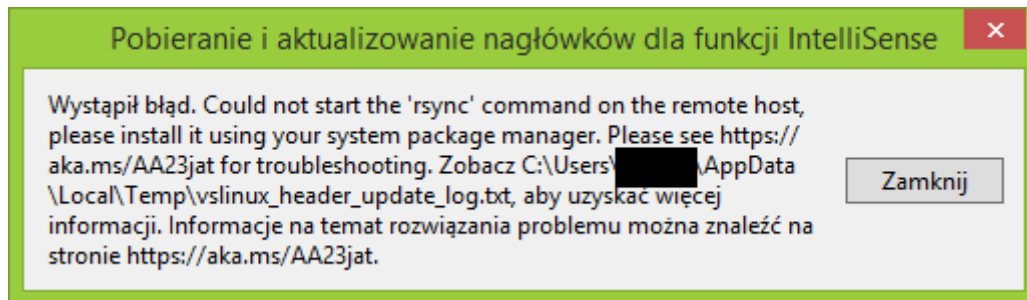
Połącz Anuluj

Login: debian

Hasło: debian

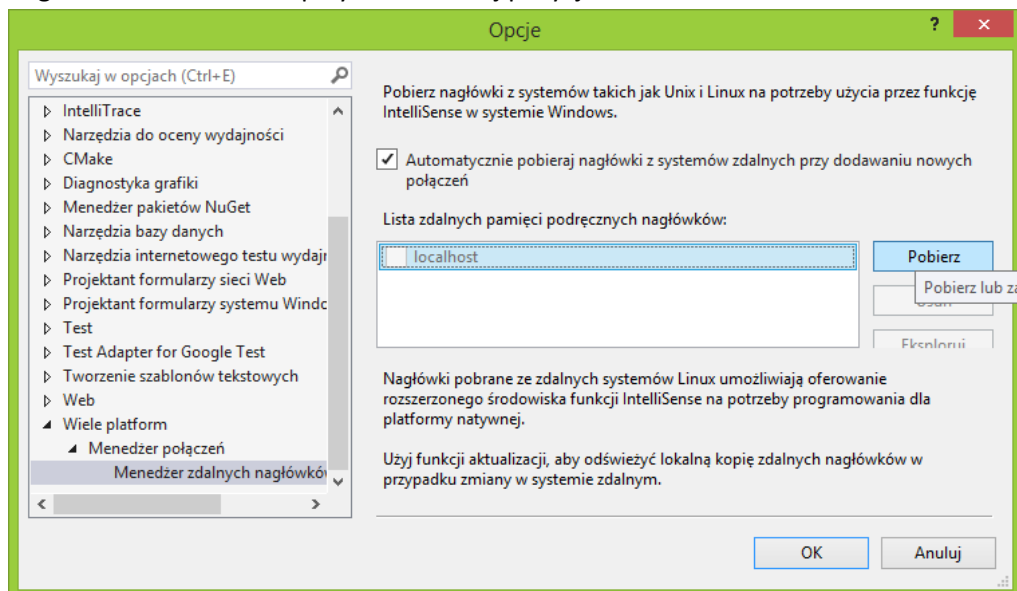
Następnie należy kliknąć „Połącz”.

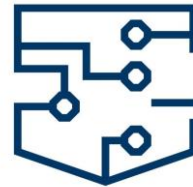
3. Może się pojawić następujący błąd



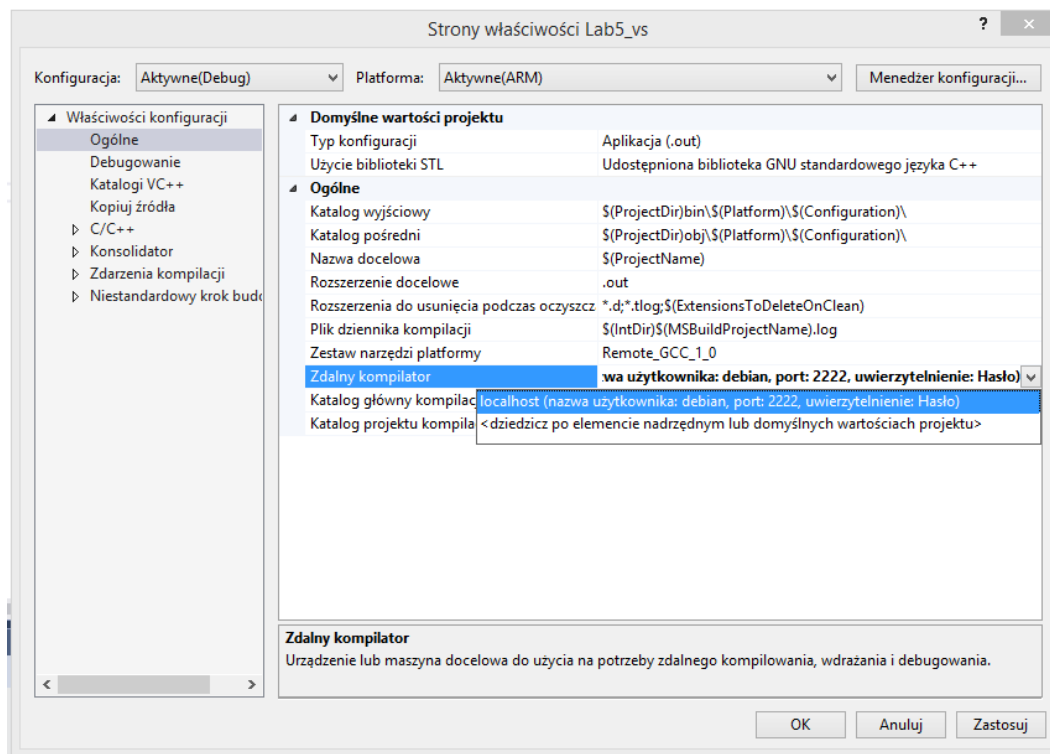
Visual Studio nie może pobrać plików nagłówkowych.

4. Aby możliwe było pobranie plików nagłówkowych należy zainstalować na maszynie wirtualnej program **rsync**. Można to zrobić w następujący sposób:
- Należy zalogować się jako root (hasło: root)
 - Należy wydać następujące polecenia:
apt-get update
apt-get install rsync
5. Po zainstalowaniu programu rsync wystarczy kliknąć „Pobierz” w „Menadżerze zdalnych nagłówków IntelliSense” przy zaznaczonej pozycji localhost.

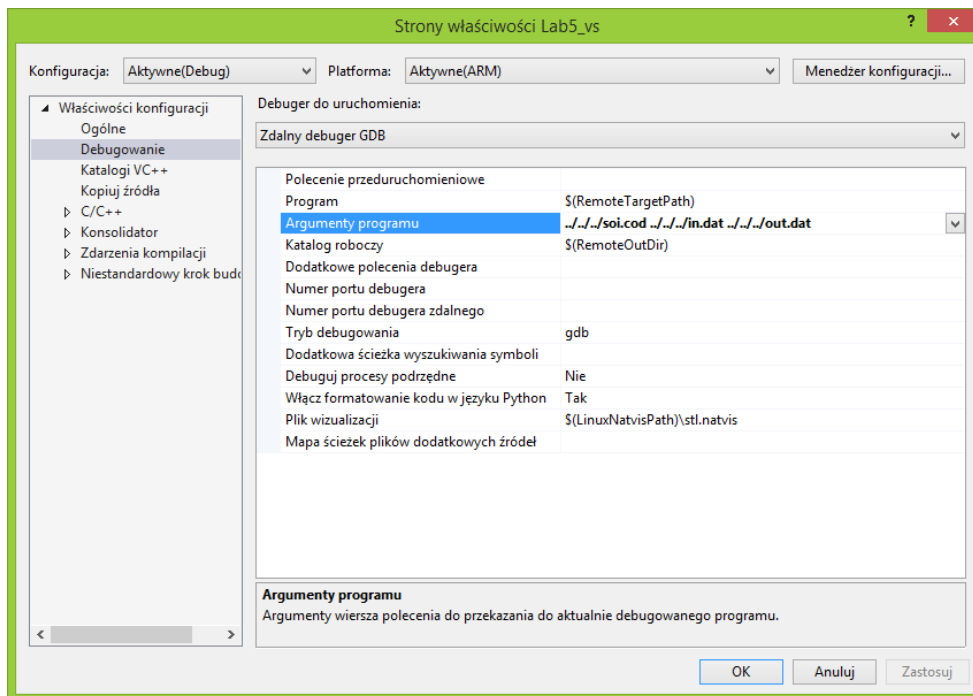




Aby uruchomić projekt na maszynie wirtualnej należy w właściwościach projektu dla platformy ARM ustawić zdalną maszynę localhost:2222



Należy również pamiętać, aby podać odpowiednie argumenty z którymi program będzie uruchomiony. Należy podać lokalizacje plików *.cod i *.dat względem katalogu w którym znajduje się skompilowany program.



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:

```

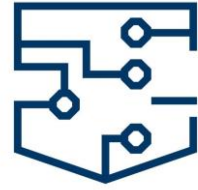
476
477 void DLXJITArm7::execute() {
478     if(program == nullptr)
479         compile();
480     auto data_ptr = data.data();
481     program(data_ptr);
482 }
```

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

```

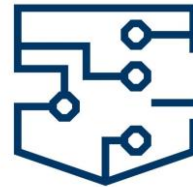
ldr    r3, [r7, #4]
ldr    r3, [r3, #80]    ; 0x50
ldr    r0, [r7, #12]
blx    r3
```

3. Podejść do instrukcji `blx` i wejść do wygenerowanej funkcji.
4. Na początku może się pojawić taki widok

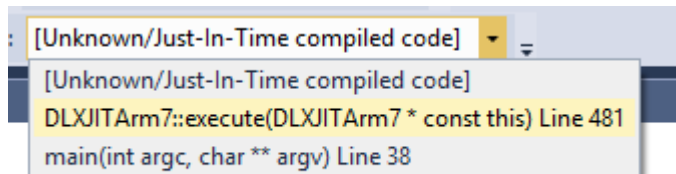


```
??  
??  
??  
??  
??  
??  
??  
??  
??  
??  
??  
??  
??  
mulpl r0, r8, r5  
addge lr, r0, #132, 4 ; 0x40000008  
mulge r4, r8, r5
```

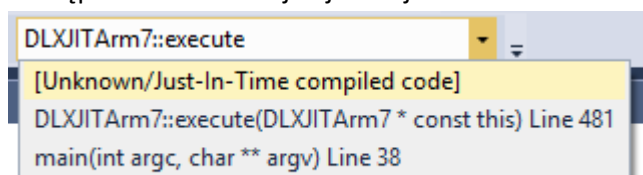
Prawdopodobnie, Visual Studio nie dało rady poprawnie pobrać zdeasemblowanego kodu od debuggera. W związku z tym należy.



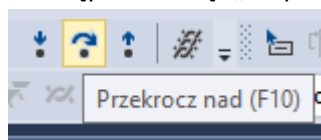
5. Przełączyć ramkę stosu na inną



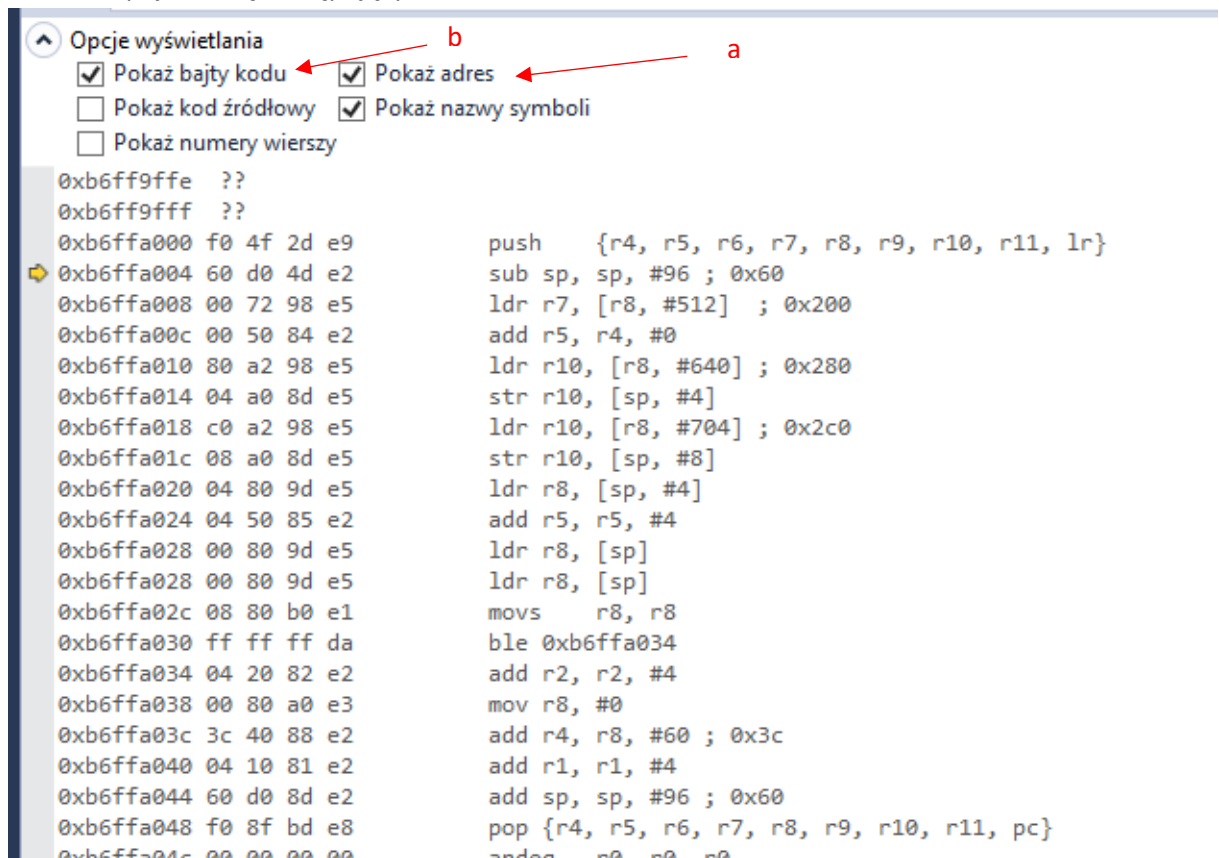
6. Następnie wrócić do tej najniższej



7. I następnie kliknąć „step over”



Powinien pojawić się następujący widok



8. 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 ARM – JIT wyświetla odpowiednie adresy w oknie konsoli systemu Linux po skompilowaniu programu DLX (Rysunek 10). Druga funkcja pozwoli zweryfikować kodowanie instrukcji.

Okno konsoli systemu Linux			
	XOR	R4, R4, R4:	0xb6ffa008
	XOR	R1, R1, R1:	0xb6ffa008
loop1:	SUBI	R1, 80, R8:	0xb6ffa008
	BRGE	R8, endLoop1:	0xb6ffa008
	LDW	R7, 0x200(R1):	0xb6ffa008

Rysunek 10 Adresy w których rozpoczyna się implementacja poszczególnych instrukcji DLX

Aby podejrzeć zawartość rejestrów należy podać ich nazwy poprzedzone \$ w oknie „Watch” („Wyrażenia kontrolne”).

Bibliografia

- [1] „ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition”. ARM Limited, 20 maj 2014. Dostęp: 17 listopad 2017. [Online]. Dostępne na:
<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html>
- [2] „Visual Studio: środowisko IDE i edytor kodu dla deweloperów i zespołów”, *Visual Studio*.
<https://visualstudio.microsoft.com/pl/> (dostęp 22 maj 2023).
- [3] Peter Verplaetse, „ESCAPE v1.1 Manual”. Department of Electronics and Information Systems University of Ghent.
- [4] „QEMU”. <https://www.qemu.org/> (dostęp 7 maj 2018).
- [5] „Procedure Call Standard for the ARM Architecture”. ARM Limited, 24 listopad 2015. Dostęp: 5 lipiec 2018. [Online]. Dostępne na:
http://infocenter.arm.com/help/topic/com.arm.doc.ihl0042f/IHL0042F_aapcs.pdf