Politechnika Warszawska
Wydział Elektroniki I Technik Informacyjnych
Instytut Telekomunikacji

Systemy komputerowe: architektura i programowanie (SYKOM)

Laboratorium

Ćwiczenie 1 2023.02.23
Temat: Podstawy tworzenia i uruchamiania oprogramowania dla procesora RISC-V pracującego bez systemu operacyjnego.

Warunki wstępne:
Zapoznać się z:
Architekturą procesora RISC-V
Działaniem programu GDB i QEMU oraz opcjami ich uruchamiania.

1. Przygotowanie środowiska do pracy

1.1.Komputer laboratoryjny

Dla celów ćwiczeń laboratoryjnych przygotowano zestaw maszyn wirtualnych działających pod systemem Linux Debian. W komputerach tych zainstalowano odpowiednie narzędzia dla procesora RISC-V w wersji RV32I oraz odpowiednie pliki nagłówkowe. Aby mieć dostęp do jednej z tych maszyn należy dokonać rezerwację z wykorzystaniem tzw. Systemu Rezerwacji Zasobów, który jest dostępny na komputerze o adresie: zsutresv.tele.pw.edu.pl.

Właściwą rezerwacje realizuje się poprzez przeglądarkę WWW zainstalowaną na swoim komputerze i używając serwowanej przez ten system rezerwacji strony (nazywany z ang. ResourceReservation):

```
http://zsutresv.tele.pw.edu.pl/ResourceReservation/
```

Proszę zwrócić uwagę, że w powyższym adresie użyto „http" a nie „https" – nie jest to błąd. Po zalogowaniu się (hasła otrzymują studenci listem elektronicznym) można zarezerwować dla siebie w tzw. aplikacji „23L SYKOM Lab1" wybrany przez siebie slocie czasowym maszynę wirtualną na której będzie można wykonywać ćwiczenie laboratoryjne.
Aby połączyć się z przydzieloną nam maszyną wirtualną należy ponownie otworzyć przeglądarkę WWW i otworzyć stronę: https://resrepo.tele.pw.edu.pl. Po czym używając danych uwierzytelniających otrzymanych osobnym listem zalogować się na tę stronę. W terminie który wcześniej wybraliśmy, system przydzieli nam maszynę wirtualną a na powyższej stronie dokładnie w tym czasie pojawią się niezbędne dane do zalogowania się do przydzielonej nam maszyny wirtualnej. Danymi tymi będą:
-plik z rozszerzeniem OVPN zawierający: certyfikat i klucz prywatny a umożliwiającym dostęp przez program OpenVPN do maszyn wirtualnych (proszę ten plik pobrać na swój dysk, a potem użyć go jako parametr „--config" w wywołaniu konsolowym programu openvpn lub zaimportować go programem OpenVPN-GUI),
-numer IP przydzielonej nam maszyny wirtualnej (używany do zestawienia połączenia SSH z tą maszyną),
-hasło dostępowe do VPN (serwer VPN wymaga odpowiedniego certyfikatu, klucza i hasła dostępowego do poprawnego zestawienia połączenia),
-czas kiedy dostęp do systemu zostanie nam zabrany.

Proszę pamiętać, że stan każdej z maszyn jest przywracany do stanu oryginalnego a wszystkie dane w niej składowane ulegają usunięciu po upływie 4 godzin od początku przydzielonego slotu. Jednak

aby student miał możliwość pracy ze swoimi plikami – efektami swojej pracy, przygotowano indywidualne i dedykowane każdemu studentowi repozytorium GIT.

1.2.Użytkowanie systemu GIT na przydzielonych maszynach

Przed rozpoczęciem pracy z przydzieloną przez SystemRezerwacji maszyną należy po zalogowaniu się do niej - wykorzystując protokół SSH i uruchamiając na swoim prywatnym komputerze program ssh (pod Linux) lub Putty (pod Windows) połączyć się z przydzieloną maszyną - dokonać klonowania indywidualnego zdalnego repozytorium GIT (dane dostępowe są przesyłane osobnymi listami elektronicznymi).

Pierwsze klonowanie wykonujemy w dowolnym miejscu ale w obrębie katalogu domowego przydzielonej maszyny wirtualnej, np.: wykonując kolejno:

```
cd ~
mkdir moje_prace
cd moje_prace
```

Przed rozpoczęciem klonowania należy wyjaśnić iż program GIT przed rozpoczęciem klonowania zapyta o nasze dane uwierzytelniające (dane te przesłane zostaną studentom listem elektronicznym). Dla celów dalszego opisu będą to dane hipotetycznego i nie istniejącego użytkownika – tj. login: `sykom`. Aby dokonać klonowania wydajemy polecenie:

```
git clone https://sykom@equ.tele.pw.edu.pl/sykomXXXX/sykom
```

Podany powyżej adres (tutaj jest to: `https://sykom@equ.tele.pw.edu.pl/sykomXXXX/sykom`) jest specyficzny dla każdego indywidualnego repozytorium GIT – proszę nie modyfikować go i użyć zgodnie ze schematem tu podanym zmieniając jedynie treści w miejscach, w których występuje login (tu jest to sykom, proszę też zauważyć że występuje on tu dwa razy – co jest poprawne i obowiązkowe!). Proszę pamiętać, że w systemie Linux (czyli na przydzielonych maszynach wirtualnych) pytanie o dane uwierzytelniające mogą być podawane w linii poleceń. Dodatkowo proszę pamiętać, aby nie używać metody kopiuj-wklej, z nieznanych przyczyn proces taki dość często kończy się porażką autoryzacji.

Gdy mamy pewność, że zawartość naszego indywidualnego zdalnego repozytorium GIT jest obecna na udostępnionej maszynie laboratoryjnej, dla pewności możemy wykonać polecenie:

```
git pull
```

Pod koniec pracy z dedykowaną maszyną laboratoryjną (proszę pamiętać że po upływie 4h dostęp może zostać zerwany – trzeba zatem operacje poniższe rozpocząć nieco wcześniej) wykonujemy w lokalnej kopii repozytorium GIT (czyli wewnątrz katalogu: `~/moje_prace`) operacje:

a)Dodanie wszystkich swoich poprawek do lokalnego repozytorium (domyślnie po sklonowaniu danych ze zdalnego repozytorium pracujemy wyłącznie na własnej kopii), kropka użyta w poniższym poleceniu oznacza – dodaj wszystkie zmodyfikowane pliki:

```
git add .
```

Po wydaniu tego polecenia możemy zobaczyć czy wszystkie nasze poprawione pliki zostały uwzględnione, wydając polecenie:

```
git status
```

b)Zatwierdzanie zbioru poprawek/treści (niezbędne, aby system poprawnie rozpoznawał kiedy i co zostało uznane za zrobione):

```
git commit –a –m „Stan z konca zajec lab1”
```

Zaleca się nie stosowanie polskich znaków diakrytycznych w sekcji objętej cudzysłowem a będących zdawkowym opisem czego dotyczy zbiór zatwierdzonych poprawek/treści.

c)Po wydaniu powyższego polecenia możemy swoje poprawki i treści (nowe pliki) „wypchnąć” do zdalnego repozytorium (tak aby np.: prowadzący mogli jest sprawdzić i wystawić ocenę):

```
git push
```

Przypomina się, aby pamiętać, iż procesy „wypychania" danych mogą zajmować pewien nie mały czas (zależnie od liczby i wielkości nowych lub zmienionych plików). Dlatego zalecane jest, aby punkty od a) do c) wykonywać także podczas trwania prac nawet gdy stan prac nie jest w pełni zaawansowany. Dodatkowo dla pewności możemy na swoim prywatnym komputerze zweryfikować czy wszystkie wytworzone przez siebie pliki znalazły się na dedykowanym zalanym repozytorium GIT.
Więcej na temat systemu GIT można znaleźć w Internecie, w tym na stronie: https://git-scm.com
Dodatkowo przed laboratorium w katalogu lab1 prowadzący ćwiczenia umieszcza plik PDF o nazwie niemal identycznej jak nazwisko i imiona osoby dla której utworzono zdalnej repozytorium. W tym przykładzie byłby to plik o nazwie: `Sykom_lab1.pdf`

1.3. Sprawdzenie narzędzi na przydzielonej maszynie

Jedną z wielu części tworzonego w ramach tego laboratorium raportu jest sprawdzenie wersji używanych narzędzi. Aby sprawdzić czy narzędzia są dostępne i w jakich wersjach, należy uruchomić poniższe pliki dodając do ich wywołania opcję „--version" (proszę zauważyć, że argumenty wywołania z czytelnym dla człowieka tekstem, tak jak tu version w systemie Linux na ogół poprzedzone są dwoma znakami – a nie pojedynczym):

```
riscv32-unknown-elf-gcc    - kompilator GCC dla RV32I
qemu-system-riscv32        - emulator procesora RISC-V w wersji RV32I
dtc                        - kompilator Device-Tree-bolb
```

Następnie trzeba zweryfikować, czy qemu-system-riscv32 wspiera właściwą wersję platformy RISC-V, realizuje się to wydając polecenie:

```
qemu-system-riscv32 -machine help
```

i sprawdzenie czy QEMU wspiera wymagane maszyny - powinniśmy ujrzeć (ten wynik także proszę zawrzeć w raporcie):

```
Supported machines are:
none                 empty machine
...
sykt                 RISC-V VirtIO Board SYKT (Privileged ISA v1.00)
...
```

Dla dalszych prac najistotniejsze jest, aby znaleźć emulowaną maszynę o nazwie SYKT. Program `QEMU` dla potrzeb zajęć z tego przedmiotu został wyposażony w odpowiednie peryferia i cechy, i tylko maszyna SYKT właśnie je realizuje. Program `QEMU` pobrany z innych źródeł nie posiada wymaganych peryferii.

2. Pozyskanie konfiguracji docelowego systemu

Każda maszyna (np.: platforma z „krzemowym" CPU, emulator CPU, ...) posiada własną konfigurację. Dla platform z krzemowym CPU konfigurację taką sporządza jej konstruktor i publikuje ją w tzw. „Manualach". W przypadku emulatora `QEMU` istnieje możliwość automatycznego wygenerowania opisu takiej konfiguracji, zgodnie z tym jak ten emulator został skompilowany i jakie komponenty posiada.
Konfigurację tą można odkryć poprzez wydanie polecenia (proszę zwrócić uwagę na podwójne znaki „--"):

```
qemu-system-riscv32 -machine sykt -bios none --machine dumpdtb=sykt.dtb
```

Polecenie to wytworzy plik `sykt.dtb`, jest to tzw. plik opisu deviece-tree-blob, Więcej szczegółów dostarczają materiały wykładowe, oraz zewnętrzne dokumenty [1][2][3].
Plik `sykt.dtb` jest zapisany w postaci binarnej, narzędzie DTC[4] posiada jednak możliwość rekompilacji takiego pliku do postaci źródłowej. Wykonuje to polecenie:

```
dtc -I dtb -O dts -o sykt.dts sykt.dtb
```

Otrzymawszy plik `sykt.dts`, zawierający czytelny dla człowieka zapis, możemy znaleźć wprowadzenie:

```
/dts-v1/;

/ {
        #address-cells = <0x2>;
        #size-cells = <0x2>;
        compatible = "riscv-virtio";
        model = "riscv-virtio,qemu";
```

Potwierdza on iż system będzie emulował procesor Risc-V. Cała zawartość tego pliku na tym etapie prac nie jest istotna, najistotniejsze są jednak informacje o:
a)umiejscowieniu pamięci:

```
memory@80000000 {
                device_type = "memory";
                reg = <0x0 0x80000000 0x0 0x8000000>;
        };
```

Tu widzimy, że pamięć (zakładana jest jednolita przestrzeń) zaczyna się pod adresem: `0x80000000`.
b)umiejscowieniu modułu dostarczającego port diagnostycznej komunikacji szeregowej oraz tzw. unikatowy numer CPU:

```
sykt@100000 {
                reg = <0x0 0x100000 0x0 0x1000>;
                compatible = "syktid0";
        };
```

Dla dalszych rozważań istotne jest, że moduł ten jest dostępny pod adresem `0x100000` i ma wielkość `0x1000`. Zatem na podstawie powyższych informacji wiemy czy są obecne oraz gdzie w przestrzeni pamięciowej umiejscowiono wymagane dla dalszej pracy odpowiednie zasoby – te informacje proszę zamieścić w raporcie z wykonanych w laboratorium prac. W raporcie wymagane jest także zamieszczenie wygenerowanego pliku `sykt.dts`.
Warto wspomnieć jakie cechy i jaka budowę ma moduł SYKT. Najważniejsze jest to, że jest to podprzestrzeń pamięci na której początku pod adresem relatywnym równym 0 względem całego modułu umiejscowiono tzw. rejestr sterowania (`SYKT_CTRL_ADDR`) o wielkości 4 bajtów. Jest to rejestr tylko do zapisu. Jego głównym i jak na razie jedynym zadaniem jest wyłączenie symulatora. Wykonuje się to poprzez wpisanie wartości `0x00005555` lub `0x00YY3333` (gdzie YY to wartość przekazywana jako kod wyjścia podczas tego wyłączania symulatora). W przytoczonej powyżej zawartości pliku DTS wynika, że rejestr ten umieszczono pod adresem 0x00100000.
Następnym rejestrem jest tzw. rejestr unikatowej identyfikacji procesora (`SYKT_ID_ADDR`). Jest to rejestr tylko do odczytu o wielkości 4 bajtów. Jego lokacja jest o cztery większa od adresu `SYKT_CTRL_ADDR` więc w podanym tu przykładzie odpowiada adresowi `0x00100004`. Wartość odczytana z tego rejestru stanowi unikatowy numer egzemplarza procesora tej serii (w ćwiczeniu laboratoryjnym jest tylko jedna seria).
Następny rejestr to element emulującego lekki moduł UART (`SYKT_UART_ADDR`). Moduł ten umożliwia prostą komunikację z użytkownikiem poprzez konsolę w której uruchomiono QEMU bez konieczności żmudnych procedur inicjalizacji tego modułu. Rejestr tego modułu używany jest tylko do zapisu i jest o wielkości także 4 bajtów. Jego lokacja przylega do rejestru unikatowej identyfikacji procesora i względem `SYKT_CTRL_ADDR` jest ta lokalizacja o 8 bajtów większa, więc w podanym przykładzie odpowiada adresowi `0x00100008`.
Następnie w lokacji oddalonej o 12 bajtów (`0x000c`) od `SYKT_CTRL_ADDR` (czyli w podanym przykładzie od adresu `0x0010000c`) rozpoczyna się przestrzeń tzw. `SYKT GPIO`. Przestrzeń `SYKT GPIO` rozciąga się do końca podprzestrzeni pamięciowej całego modułu `SYKT` (czyli do lokacji o

4096B większej od SYKT_CTRL_ADDR), czyli w podanym przykładzie jest to adres 0x101000. Przestrzeń SYKT GPIO w laboratorium 1. jest nie wykorzystywana.

3. Przygotowanie skryptów dla linkera zgodnych z konfiguracją docelowego systemu

Proces kompilacji źródeł zapisanych w języku C, składa się z dwóch etapów: właściwej kompilacji i linkowania (pomijamy tu etap 0: preprocesing). O ile w kodzie w C zapis jest dla kompilatora neutralny względem sprzętu na jakim będzie kod wynikowy działał, o tyle proces linkowania jest bardzo mocno związany z architekturą docelowej maszyny.
Dzięki odkrytej w poprzednim punkcie szczegółom architektury emulowanego CPU, można dostosować skrypt dla linkera do swojego środowiska. Pracę można zacząć od wygenerowania domyślnej konfiguracji linkera dla posiadanych narzędzi:

```
riscv32-unknown-elf-ld --verbose > sykt.ld
```

Jest to jednak podejście w którym trzeba wiele elementów w pliku sykt.ld zmodyfikować. Dlatego dla potrzeb laboratorium w katalogu „Lab1" na serwerze Studia umieszczono także plik sykt.ld-wzorzec. Jest to wzorzec w którym trzeba dokonać odpowiednie modyfikacje.
Zgodnie z zawartością pliku sykt.dts, należy dokonać modyfikacji umiejscowienia pamięci RAM. Odpowiednie wartości przechowuje sekcja MEMORY:

```
OUTPUT_FORMAT("elf32-littleriscv", "elf32-littleriscv",
              "elf32-littleriscv")
OUTPUT_ARCH(riscv)
/* >>> Modification for ldb to point to proper memory <<< */
MEMORY
{
  RAM (rwx)       : ORIGIN = 0xZZZZZZZZ, LENGTH = 128M
}
/* >>> Modified Section End <<< */
ENTRY(_start)
...
```

Elementy, które warto tu skorygować (w zależności od zawartości pliku sykt.dts) to przypisanie wartości do słowa kluczowego ORIGIN (powyżej to 0xZZZZZZZZ). Ustala to słowo gdzie jest podłączona w przestrzeni pamięciowej pamięć RAM. Jej wielkość (LENGTH) nie ma, aż takiego znaczenia – gdyż tworzone będą krótkie pliki wykonywalne, a ważne jest to, aby program QEMU mógł od maszyny hostującej (tej na której jest uruchamiany) otrzymać dla swojej pracy odpowiednią porcję pamięci danych w której mógłby emulować tę pamięć RAM.
Dodatkowo wyjaśnienia wymaga określanie wyłącznie pamięci RAM, w ćwiczeniu używana jest wyłącznie architektura Von Neumana, gdzie cała przestrzeń pamięciowa może być emulowana jako RAM. Wymagane jest zamieszczenie w raporcie zmodyfikowanego pliku linkera (sykt.ld) z przeprowadzonymi przez siebie modyfikacjami.

4. Utworzenie i kompilacja prostego programu demonstracyjnego

Proces uruchamiania typowej aplikacji w środowisku systemów operacyjnych jest realizowany przez ten system. System ten zapewnia odpowiednie alokacje zasobów, załadowanie obrazu aplikacji i wiele innych wspierających usług. W ramach tego ćwiczenia poznawane są sytuacje gdy tego systemu operacyjnego jeszcze nie ma. Jest to przypadek określany często jako tzw. ang. „Bare-metal". Warto tu nadmienić, że samo jądro systemu operacyjnego (Kernel systemu Linux) jest także w podobnej sytuacji – samo musi zadbać o wywołanie odpowiednich mechanizmów i samo sobie zaalokować odpowiednie zasoby.
Rozpatrywane w ramach tego ćwiczenia zadanie jest nieco prostsze i polega na uruchomieniu kodu aplikacji, która sama sobie zapewni styk z niezbędnymi zasobami (stos, dostęp i współpraca z peryferiami). Typowo w takich sytuacjach tworzony jest specjalizowany plik startowy ułatwiający uruchomienie kodu zapisanego w języku C. W naszym przykładzie będzie to plik crt0.s, o następującej treści:

```
.section     .init, "ax"
.global       _start


_start:
    .cfi_startproc
    .cfi_undefined ra
    .option push
    .option norelax

    la       gp, __global_pointer$

    .option pop

    la       sp, __stack_top
    add      s0, sp, zero

    jal      zero, main

    .cfi_endproc
    .end
```

Plik ten zapisano w języku asembler, jego zadaniem jest ustawienie szczytu stosu, ustalonego przez skrypt linkera w definicji `__stack_top`, oraz skok do procedury `main()` zapisanej w pliku main.c. Jest to mocno uproszczony plik startowy, nie inicjuje on wszystkich elementów aplikacji (inicjacja stałych, zmiennych globalnych itp.), jednakże dla potrzeb tego ćwiczenia jest on wystarczający. Plik ten przez swoją konstrukcję także uniemożliwia proste wywołanie funkcji bibliotecznych.
Struktura głównego (i jedynego) pliku w C omawianej aplikacji jest bardziej złożona. Ponieważ nie możemy polegać na systemie operacyjnym dołączanie klasycznych bibliotek jest bez celowe. Aby jednak móc skorzystać z peryferii należy zdefiniować zbór makr i funkcji usługowych:

```
1.  #define RAW_SPACE(addr)      (*(volatile unsigned long *)(addr))

2.  #define SYKT_CTRL_ADDR       (0x00100000)
3.  #define SYKT_ID_ADDR         ((SYKT_CTRL_ADDR)+4)
4.  #define SYKT_UART_ADDR       ((SYKT_CTRL_ADDR)+8)
5.  #define SYKT_EXIT_VAL        (0x00003333)
6.  #define SYKT_UART_VAL        (0x00008888)

7.  void my_simulation_exit(unsigned char ret_code){
8.    RAW_SPACE(SYKT_CTRL_ADDR) = SYKT_EXIT_VAL | (((unsigned long)ret_code)<<16);
9.  }

10. void my_putchar(unsigned char c){
11.   RAW_SPACE(SYKT_UART_ADDR)=SYKT_UART_VAL | ((0x000000FF & (unsigned long)c)<<16);
12. }

13. unsigned long my_get_cpu_id(void){
14.   return RAW_SPACE(SYKT_ID_ADDR);
15. }
```

Jak widać w linii 1 zdefiniowano specyficzny mechanizm dostępu do zasobów peryferyjnych. Jest to prosta metoda odwołania się do peryferii zmapowanych w przestrzeni pamięciowej pod określonym adresem. Bazujące na tym mechanizmie odwołania wykorzystano w m.in. w funkcji `my_simulation_exit()`, linie 7-9. Zadaniem tej funkcji jest wpisanie pod adres `SYKT_CTRL_ADDR` wartości będącej złożeniem wartości `0x00003333` i na bitach 23..16 docelowej wartości tzw. kodu wyjścia[1]. Po wpisaniu tak wypracowanej wartości automatycznie kończona jest praca programu `QEMU`, który zwróci przekazany kod wyjścia powłoce skąd został uruchomiony.

[1] Kod wyjścia to liczba w zakresie 0…255 przekazywana na zakończenie działania każdego procesu

W liniach 10-12 zdefiniowano funkcję przydatną w uruchamianiu kodu a zapewniającą komunikację z użytkownikiem – tę łączność program `QEMU` będzie emulować z wykorzystaniem konsoli w której został uruchomiony.
Zastanawiająca może być implementacja `my_get_cpu_id()` (linie 13-15), jej działanie polega na odczytaniu 32bitowego pola spod adresu `SYKT_ID_CTRL_ADDR`, emulowany CPU został tak skonstruowany aby, pod tym adresem przy próbie odczytu zwracać swój unikatowy 32bitowy identyfikator.
Podsumowując zatem dla uwspólniania danych otrzymanych z pliku `sykt.dts` z tymi zapisanymi w kodzie C dla poprawnego działania systemu, należy zwrócić uwagę na linie: 2, 3 i 4 powyższego kodu – tam zapisano bowiem kod wyznaczający adresy z zasobami istotnymi podczas tych zajęć. Proszę pamiętać, że w otrzymanym dla zajęć laboratoryjnych emulatorze QEMU adres podany w linii 2 może być inny – jego ustalenie opisał ten dokument w rozdziale 3.
Aby właściwe pliki można było skompilować najwygodniej jest utworzyć plik Makefile. Jego zapis pomoże narzędziu `make`[6][7] dokonać wywołania właściwych narzędzi z odpowiednimi opcjami. Poniżej znajduje się wzorcowy zapis takiego pliku:

```
GCC=riscv32-unknown-elf-gcc
OBJDUMP=riscv32-unknown-elf-objdump
MARCH=rv32i
MABI=ilp32
LD_SCRIPT=sykt.ld

CFLAGS=-O0
LDFLAGS=-ffreestanding -Wl,--gc-sections
LDFLAGS+=-nostartfiles -nostdlib -nodefaultlibs -Wl,-T,$(LD_SCRIPT)
LDFLAGS+=-march=$(MARCH) -mabi=$(MABI)

all:
	$(GCC) -g $(CFLAGS) $(LDFLAGS) crt0.s main.c -o sykt
	$(OBJDUMP) -DxS sykt > sykt.lst
clean:
	rm -f sykt.lst
	rm -f sykt
```

Powyższy plik Makefile definiuje cel: „all”, jest on domyślnym i dzięki niemu w trakcie działania polecenia `make` dokonane zostanie wywołanie enigmatycznego polecenia `$(GCC)`. Jest to zmienna zdefiniowana w pierwszej linii pliku Makefile, a kryje się pod nią nazwa programu: `riscv32-unknown-elf-gcc`, który będzie tu uruchomiony. Po automatycznym jego uruchomieniu, przejmie on rolę kompilatora i linkera – stąd nie kompilujemy i linkujemy osobno.
Wartym zwrócenia uwagi w linii wywołania kompilatora jest użycie opcji:

```
-nostartfiles -nostdlib -nodefaultlibs
```

Opcje te zapewniają ominięcie dołączania domyślnie używanych bibliotek. W przypadku systemów Bare-metal jest to niemal norma, należy jednak pamiętać, że dzięki tym opcjom żadne standardowo wykorzystywane funkcje nie będą dostępne (jest zatem wysoce zalecane aby z funkcji standardowych bibliotek nie korzystać).
Po wydaniu polecenia `make` w katalogu z powyższymi plikami powinniśmy otrzymać binarny plik `sykt` - będący obrazem aplikacji oraz `sykt.lst` - będący odwzorowaniem plików źródłowych poddanych kompilacji i linkowaniu, plik ten jest zapisany w asemblerze procesora RISC-V. Analiza tego pliku może nam dostarczyć wiele ważnych informacji np.: czy kompilacja przebiegła pomyślnie – w szczególności czy adres pod jakim umieszczony zostanie kod jest prawidłowo zdefiniowany w pliku `sykt.ld`.
Pliki `Makefile`, `sykt.lst` oraz pliki z rozszerzeniami .c i .S a niezbędne dla uruchomienia aplikacji demonstracyjnej proszę dodać do raportu.

5. Uruchomienie programu z użyciem emulatora QEMU i narzędzia GDB

Po wykonaniu zgodnie z instrukcjami z pkt. 4 kompilacji powinien powstać plik binarny `sykt`. Plik ten jest zapisany w formacie ELF[8] i zawiera on wszystkie elementy niezbędne do kodu zapisanego w funkcji `main()`. Dla potrzeb laboratorium uruchomienie tak utworzonego kodu będzie realizowane za pomocą emulatora QEMU. Projekt QEMU wspiera wielką liczbę CPU i platform z nimi związanych. W ramach tego laboratorium skupmy uwagę na jego narzędzie o nazwie: `qemu-system-riscv32`. Jak już wiadomo wspiera on procesor RISC-V w wersji RV32I a dodatkowo emuluje komponenty SYKT.

Przygotowane dla tego laboratorium narzędzie wspiera także możliwość uruchamiania kodu w tzw. trybie pracy krokowej. Dzięki temu, że emulator wspiera polecenia narzędzia GDB, można uruchamiać program krok po kroku, analizując jego działanie zarówno na poziomie kodu w języku C, jaki i na poziomie maszynowym.

Narzędzie GDB jest częścią pakietu GCC i dla potrzeb laboratorium przygotowano program GDB o pełnej nazwie: `riscv32-unknown-elf-gdb`, który bezpośrednio realizuje pracę krokową. Narzędzie to posiada jeszcze możliwość dostrajania sposobu działania do naszych potrzeb. Dostrajanie to najlepiej zapisać w pliku `.gdbinit` o następującej treści:

```
set style address foreground white
target remote :1234
load
b main
define f
si
end
define r
c 1000000
end
set disassemble-next-line on
show disassemble-next-line
```

Do pracy z programem GDB choć dostrojenie to nie jest niezbędne, to jednak ułatwia pracę, np.: definiuje tzw. `breakpoint` na wywołaniu funkcji `main()`, definiuje także makra programu GDB. Makra te stanowią wsparcie dla użytkownika. W powyższej definicji mamy dwa takie makra `f` i `r`. Makro o nazwie `f` - po wpisaniu litery `f` w okno dialogowe programu GDB i zatwierdzeniu klawiszem `Enter` - wykona jedną instrukcję. Dodatkowo po każdym wywołaniu makra `f`, program GDB automatycznie będzie pokazywał kolejną instrukcję w postaci asemblerowej. Makro `r` wykona 1000000 instrukcji emulując w ograniczonym stopniu tzw. polecenie run.

Przed uruchomieniem programu GDB należy pamiętać, aby już działała maszyna emulowana przez QEMU na której wykonywany będzie kod uruchamianej przez nas aplikacji – program GDB będzie chciał się z nią połączyć poprzez TCP. Taką maszynę można uruchomić według poniższego:

```
qemu-system-riscv32 -machine sykt -bios none -m 128M -gdb tcp::1234 -S -kernel sykt
```

Od tego czasu emulator QEMU będzie czekał na połączenie programu GDB. Dzieje się tak dzięki opcji `-S` (duża litera S), bez tej opcji QEMU automatycznie uruchomi program `sykt` bez możliwości połączenia się z użyciem programu GDB, czyli np.: beż możliwości uruchomienia pracy krokowej.

Aby uruchomić GDB najlepiej otworzyć nowy terminal (czyli zestawić nowe połączenie SSH) – wymagana jest tu zatem równoległa praca QEMU i GDB. Samo wywołanie GDB jest nieco zawiłe:

```
riscv32-unknown-elf-gdb -iex "add-auto-load-safe-path .gdbinit" sykt
```

Program jak widać spodziewa się w aktualnym katalogu pliku `.gdbinit` (proszę uważać na nazwę tego pliku – zaczyna się od znaku „ . ”) oraz `sykt`. Plik `sykt` jak widać jest używany zarówno przez QEMU jak i GDB, wynika to z faktu że QEMU potrzebuje treści tego pliku do zapełnienia pamięci operacyjnej emulowanej maszyny, a GDB używa go do odczytania symboli niezbędnych dla

procesu `debug` – możliwości wyświetlenia informacji o kodzie źródłowym. Proszę przypomnieć sobie że w pliku Makefile (opis roz. 4) zapisano instrukcję informującą kompilator aby dołączył tzw. symbole do skompilowanego kodu – co czyni opcja „-g".
Obsługa GDB jest raczej prosta, jak wspomniano naciskając klawisz `f` można przechodzić przez kod krok po korku. Istnieje też możliwość emulacji w trybie z pełną szybkością – czyli emulacją normalnego działania kodu. Taki tryb uruchamia się za pomocą wbudowanego polecenia „`continue`" lub w skrócie `c`.
Proszę pamiętać, że aplikacje dla systemów Bare-metal, zasadniczo nigdy same z siebie nie powinny kończyć swojego działania – nie mogą one zwrócić sterowania do systemu operacyjnego, gdyż takowego w jej środowisku wywołania nie ma. Zatem, aby zakończyć działanie uruchomionej według powyższych instrukcji aplikacji można wykonać dwie operacje: nacisnąć CTRL-C lub co bardziej jest zalecane - umieszczenie w kodzie wywołania funkcji `my_simulation_exit()`. Funkcja ta wywołuje specjalny mechanizm kończenia pracy emulatora QEMU.
Proszę zwrócić uwagę także na informacje raportowane przez QEMU podczas uruchamiania się. Oprócz informacji o dacie kompilacji program przedstawia on także dane niezbędne do identyfikacji konfiguracji użytej przez program QEMU. Prolog tych informacji wygląda następująco:

```
qemu-system-riscv32: info: Qemu for SYKT lecture made by A.Pruszkowski (Compiled at: Oct
14 2021 09:19:29)
Qemu internal configuration was found.
QEMU start report:
Compilation: Oct  14 2021 09:19:29
NIC:         enp0s3
NIC MAC:     08:00:2a:17:50:e1 (17)
HASH:        0x2d0a7a0da4e92841
GPIO Emulator for QEMU initializing .... (Compiled at: Oct  14 2021 09:19:29)
VNC server running on ::1:5900
```

Po pojawieniu się informacji o tym jak serwer VNC nasłuchuje (co w tym przypadku nie ma znaczenia), program przechodzi do oczekiwania na polecenia programu GDB. W raporcie należy także zamieścić ten prolog – pomoże on zweryfikować poprawność wykonania zadania przez Studenta.
Podczas pracy procesora okienko w którym uruchomiono program QEMU ukazuje także komunikaty wygenerowane poprzez użycie powyżej opisanej funkcji `my_putchar()`

6. Literatura

[1] https://en.wikipedia.org/wiki/Device_tree, ostatnia wizyta 2022.03.01
[2] https://git.kernel.org/pub/scm/utils/dtc/dtc.git/plain/Documentation/manual.txt?id=HEAD, ostatnia wizyta 2022.03.01
[3] https://github.com/devicetree-org/devicetree-specification/releases/download/v0.3/devicetree-specification-v0.3.pdf, ostatnia wizyta 2022.03.01
[4] https://git.kernel.org/pub/scm/utils/dtc/dtc.git/about, ostatnia wizyta 2022.03.01
[5] https://elinux.org/Device_Tree_Reference, ostatnia wizyta 2022.03.01
[6] https://www.gnu.org/software/make, ostatnia wizyta 2022.03.01
[7] https://www.gnu.org/software/make/manual/make.html, ostatnia wizyta 2022.03.01
[8] https://en.wikipedia.org/wiki/Executable_and_Linkable_Format, ostatnia wizyta 2022.03.01