bfc64 - dokumentacja

Grzegorz Koperwas

16 grudnia 2020

1. Kompilowanie

Wymagania do kompilacji:

- gcc - kompilator
- make
- python - generowanie pliku `template.cpp` (patrz kompilator)
- xelatex - dokumentacja

Kompilujemy poleceniem `make`.
Instalujemy poleceniem `sudo make install`
Dokumentacje kompilujemy poleceniem `make docs`.

Wymagania do używania

- kickassembler[1] - bfc64 generuje pliki `.asm` dla tego assemblera
- System Linux, testowane wyłącznie na Arch'u

Program uruchamiamy poleceniem `bfc64 <ścieżka do pliku źródłowego>`, utworzy on plik a.asm gotowy do przetworzenia kickassemblerem za pomocą polecenia `kickass a.asm`. Utworzy on nam plik wykonywalny `a.prg` dla commodore 64 lub emulatora.
Emulator Vice[2] uruchomi automatycznie programy poleceniem:

```
x64sc -autostart $PWD/a.prg
```

2. Działanie kompilatora bfc64

Kompilator składa się z dwóch głównych części:

- Parsera - zamieniającego pliki tekstowe na listę symboli
- Kompilatora - zamieniającego listę symboli na kod assemblera korzystając z funkcji z przestrzeni `arch`

[1] http://theweb.dk/KickAssembler/Main.html#frontpage
[2] https://vice-emu.sourceforge.io/

Parser

Parser jest funkcją `parseSourceFile`, która przyjmuje jako argument plik z kodem źródłowym. Zamienia ona wewnętrznie znaki języka *brainfuck* na odpowiadające im symbole według tablicy 2.

Symbol	Wartość
+	inc
-	dec
<	left
>	right
[	loopBegin
]	loopEnd
,	in
.	out

Tablica 1: Symbole oraz odpowiadające im wartości z `SymbolType`

Dodatkowo parser wyświetla ostrzeżenia w przypadku jeśli w trakcie pętli za każdą iteracją jest porównywalna inna komórka pamięci, na przykład dla pętli `[>><]` zostanie wyświetlone ostrzeżenie wraz z numerem lini początkowym i końcowym.

Jeśli parser napotka `]` bez poprzedniego `[` czy nie znajdzie w pliku końca pętli przed jego końcem zgłosi on błąd użytkownikowi i nie skompiluje programu.

Parser traktuje wszystkie inne znaki jako komentarz.

Dodatkowo parser może zostać rozbudowany o zoptymalizowane symbole `add, subtract, jmpLeft, jmpRight` które łączą sąsiednie symbole w jeden[3], oszczędzając przy tym pamięć oraz cykle procesora, jednak w obecnej wersji nie zostało to jeszcze zaimplementowane.

Kompilator

Kompilator tworzy stringa na podstawie listy symboli z parsera za pomocą wzorców generowanych podczas kompilacji z plików w folderze `processor/arch/c64`. Dodatkowo na początek dołącza `arch::begin` a na koniec `arch::end`

Wzorce, z których korzysta kompilator są generowane automatycznie skryptem pythona `templateGen.py`. Generuje on plik źródłowy `template.cpp` z pomocą pliku-wzorca `template.cpp.template`, do którego podstawia za placeholdery[4] odpowiednie sumy stringów oraz parametrów funkcji według plików-wzorców assemblera, w których podmienia `label()` na argument `label` itd. Opis wzorców w sekcji 3..

Wygenerowany `string` program zapisuje do pliku wyjściowego.

3. Opis działania skompilowanego programu

Wyzwania architektury 6502/6510/commodore 64

Procesory *MOS 6502/6510* posiadają 8 bitową szynę danych oraz 16 bitową szynę adresową. Z tego powodu „pointery” muszą się znajdować w pierwszych 256 bajtach pa-

[3]Na przykład `++-+++` zostanie zamienione na `add`

[4]W formie `$nazwa`

mięci, tak zwanej „zeropage". Kompilator umieszcza w pamięci o adresie `$00FB` adres `$C000`, który jest adresem początku taśmy.

Procesor posiada opcje indeksowania pamięci rejestrem X lub Y, taki odpowiednik $(\texttt{pointer} + \texttt{rejestr}_X)^*$ w języku `C++`. Jednak jako iż rejestr `X` jest 8 bitowy, a chcemy taśmę o długości większej od 256 to ruch po taśmie w lewo wygląda tak:

```
                    ⋮
    cpx #$00 // compare X and 0
    bne label // if X != 0, go to label
    dec $00fb + 1 // decrement most significant
                  // byte of tape pointer by 1
label:
    dex // decrement X by 1
                    ⋮
```

Taśma

Definicja taśmy, w pliku `end.asm`, wygląda tak:

```
*=$C000 "Tape" // at address $C000

.fill 1024, 0 // place 1024 zeros
```

Generuje ona taśmę o długości 1024 bajtów, jednak jako iż kompilator nie zabezpiecza nas przed „wyjściem" z jej przestrzeni, zatem jeśli komuś chce się pisać dużo `>` i wiedząc iż adres pierwszej komórki to `$C0FF` możemy zmieniać kolory tła, tekstu, odtwarzać muzykę oraz nadpisywać program.

Wypisywanie znaków na ekran i ich odczyt

Kernal commodore 64 posiada *funkcje*, które realizują te zadania.

Pod adresem `$FFCF` jest funkcja, która umieszcza w rejestrze A wartość wprowadzoną przez użytkownika.

Pod adresem `$FFD2` jest funkcja, która wartość w rejestrze A wypisuje na ekran.

Wadą tych funkcji jest to iż nie używają zestawu znaków ASCII, tylko własnego PETSCII[5]. Z tego powodu przy wypisywaniu na ekran za pomocą `.` trzeba ustawiać wartość komórki na wartości PETSCIII

[5] https://www.c64-wiki.com/wiki/PETSCII