

bfc64 - dokumentacja

Grzegorz Koperwas

18 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`

Dokumentację kompilujemy poleceniem `make docs`.

Przykładowy test uruchamiamy poleceniem `make test`, powinien uruchomić emulator Vice z programem.

Wymagania do używania

- kickassembler¹ - 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 Vice² uruchomi automatycznie programy poleceniem:

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

¹<http://theweb.dk/KickAssembler/Main.html#frontpage>

²<https://vice-emu.sourceforge.io/>

2. Działanie kompilatora bfc64

Kompilator składa się z trzech głównych części:

- Parsera - zamieniającego pliki tekstowe na listę symboli
- Optymalizatora zamieniającego sąsiednie ++++-- na operacje dodawania lub odejmowania.
- Kompilatora - zamieniającego listę symboli na kod assemblera korzystając z funkcji z przestrzeni `arch`

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 1

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 linii 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.

Optymalizator

W celu ograniczenia pamięci potrzebnej na załadowanie programu optymalizator zamienia sąsiednie symbole `+` oraz `-` na symbole specjalne `add` oraz `subtract`. W planach jest dodanie zamieniania `>` i `<` na symbole specjalne `jmpLeft` oraz `jmpRight`

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

Wadą tych funkcji jest to iż nie używają zestawu znaków ASCII, tylko własnego PETSCII⁵. Z tego powodu przy wypisywaniu na ekran za pomocą `put` trzeba ustawiać wartość komórki na wartości PETSCII.

Wzorce

Kompilator korzysta z wzorców w folderze `processor/arch/c64`, które są plikami assemblera które są wstawiane za odpowiednie symbole. Za symbol `inc` wstawia `inc.asm` itd. Są one przetwarzane przez skrypt pythona do pliku `template.cpp` podczas kompilacji by nie było potrzeby wpisywania ich na sztywno.

Przykładowo na „Rysunku” 1 do rejestru `A` ładowana jest wartość aktualnej komórki, następnie porównywana jest z `#$ff`⁶. Jeśli zachodzi równość, program przechodzi do `label`, w innym przypadku upewnia się że flaga `carry` jest wyłączona i dodaje do rejestru `A` jedynkę. Następnie zapisuje wynik w pamięci.

```
        :  
        lda ($fb),y  
        cmp #$ff  
        beq label()  
        clc  
        adc #$01  
        sta ($fb),y  
label():  
        :  
        :
```

Rysunek 1: Zwiększanie komórki o 1

⁵<https://www.c64-wiki.com/wiki/PETSCII>

⁶`#` oznacza że jest to wartość a nie adres