Politechnika Śląska
Wydział Matematyki Stosowanej
Kierunek Informatyka

Gliwice, 7 stycznia 2021

Programowanie I

Projekt zaliczeniowy

„BFC64”

Grzegorz Koperwas gr. 2 lab 2C

Spis treści

1. Opis projektu

Projekt jest kompilatorem języka *brainfuck*,[1] 8 bitowym, nie pozwalającym na over-flow'y[2]. Powinno się dać go łatwo portować na inne systemy oparte o procesor `MOS 6502/6510`, czy inne architektury.

2. Wymagania

- Kompilacja do assemblera kickassembler
- Ośmiobitowa „taśma" o długości większej niż 256 bajtów, w formie *non-wrapping*
- Optymailzowanie kodu w postaci `+++++++--+` na pojedyńcze symbole

Wymagania „poza konkursem"

- Możliwość łatwej rozbudowy o inne architektury.
- Stworzenie kompilatora, a nie interpretera w assemblerze, kosztem pamięciożerności.
- Nauka assemblera.
- Kod który da się łatwo edytować, z tego powodu plik `template.cpp` jest generowany skryptem pythona by nie musieć go pisać ręcznie. W innym wypadku musiał bym ręcznie zmieniać kod podobny do tego w załączniku 1.

[1] Nie zawierającym linkera oraz assemblera.Patrz sekcja 4.1.
[2] Typ non-wrapping

```
std::string
subtract(std::string value, std::string label, std::string label2)
    {
        return "    lda ($fb),y\n    clc\n    cmp #" + value
               + "\n    bcs " + label + " //if lower than "
               + "number\n    lda #$00\n    jmp " + label2
               + "\n" + label + ":\n    sec\n    sbc #" + value
               + "\n" + label2 + ":\n    sta ($fb),y\n";
    }
```

Załącznik 1: Nie, nie piszę tego.

3. Przebieg realizacji

Sama implementacja podstawowej logiki kompilatora nie sprawiła większych problemów, toż to nie będę się rozpisywał na jej temat. Parserem jest zwykłą pętlą, podobnie jest z optymalizatorem.

Największe trudności sprawiało zaimplementowanie taśmy o długości większej niż 256 komórek. Okazuje się że koncept pointerów, ze względu na 8 bitowość procesora posiadającego 16 bitową szynę adresową, jest wykonywany przez ten procesor *ciekawie...*

Łącząc to z pierwszymi próbami używającymi rejestru `X` procesora zamiast `Y`, zamiast wartości na taśmie programy zmieniały wartość pointera. Dodając to iż procesor jeszcze ładnie indeksował pointer rejestrem `X` to programy dla paru komórek taśmy działały, potem dziwiłem się czemu taśma nie jest zerowa. Spowodowane to było tym iż właśnie okolica adresu który wybrałem na lokalizację pointera była nie używana przez Kernal, jednak obok były już nie-zerowe rejestry systemowe oraz basic'a.

Po dniu debugowania programem `C64-Debugger`[3], którego obsługa jest ciekawa. Sam interfejs (patrz rysunek 2) nie daje żadnej wskazówki jak z niego korzystać, ale po zrozumieniu breakpointów udało się go opanować.

Dodatkowe szczegóły na temat pracy programów po kompilacji znajdują się w sekcji 4.3.

4. Instrukcja użytkownika i opis działania

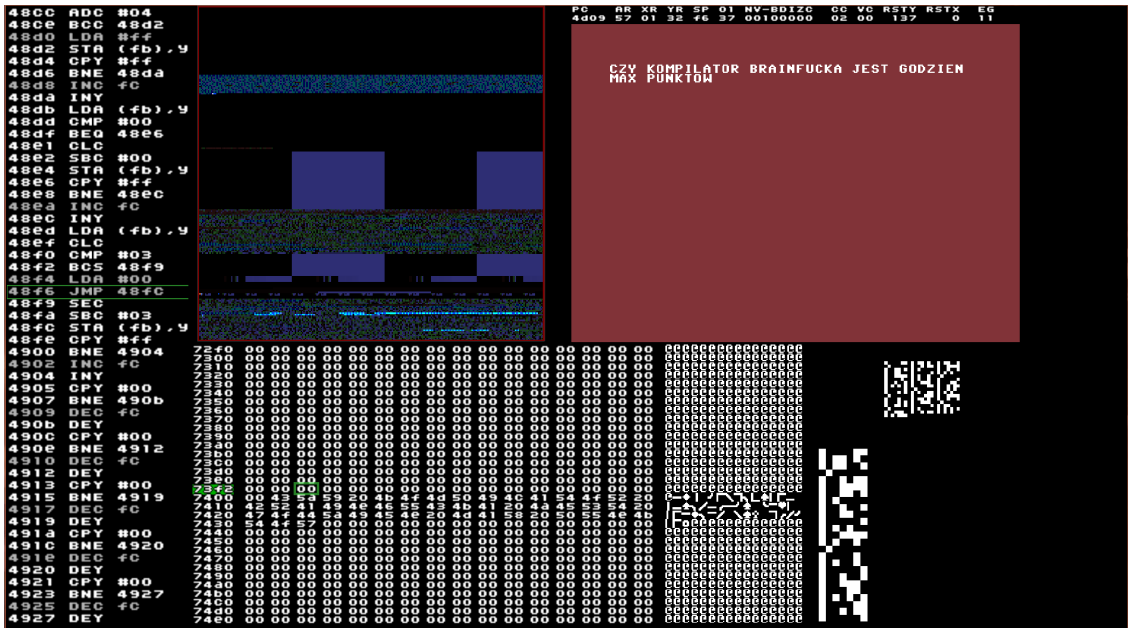
4.1. Kompilowanie

Wymagania do kompilacji:

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

Kompilujemy poleceniem `make`.

[3] https://sourceforge.net/projects/c64-debugger/



Załącznik 2: *Interfejs* programu `C64-debugger`, $\frac{10}{10}$ would debug again

Instalujemy poleceniem `sudo make install`

Dokumentacje kompilujemy poleceniem `make docs`.

Przykładowy test uruchamiamy poleceniem `make test`, powinien uruchomić emulator Vice z programem testowym `tests/test.b`.

Wymagania do używania

- kickassembler[4] - 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[5] uruchomi automatycznie programy poleceniem:

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

4.2. 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.

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

- 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 tabelki 1.

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

Tabelka 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.

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 `template.cpp.template`, do którego podstawia za placeholdery[6] odpowiednie sumy stringów oraz parametrów funkcji według plików-wzorców assemblera, w których podmienia `label()` na argument `label` itd. Opis wzorców w sekcji 4.3..

Wygenerowany `string` program zapisuje do pliku wyjściowego.

[6] W formie `$nazwa`

4.3. 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 pamięci, tak zwanej „zeropage". Kompilator umieszcza w pamięci o adresie `$00FB` adres `$7300`, który jest adresem początku taśmy.

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

```
                    ⋮
    cpy #$00       // compare Y to 0
    bne label()    // if Y != 0 goto label
    dec $00fb + 1 // decrement tape pointer
label():
    dey            // decrement Y
                    ⋮
```

Taśma

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

```
*=$7300 "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 `$73FF`/[7] 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[8]. Z tego powodu przy wypisywaniu na ekran za pomocą „`.`" trzeba ustawiać wartość komórki na wartości PETSCIII.

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`

[7] Początek taśmy `$7300` + początkowa wartość `Y` równa `$ff`

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

itd. Są one przetwarzane przez skrypt pythona do pliku `template.cpp` podczas kompilacji by nie było potrzeby wpisywania ich na sztywno.

Przykładowo w załączniku 3 do rejestru `A` ładowana jest wartość aktualnej komórki, następnie porównywana jest z `#$ff`[9]. 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():
        ⋮
```

Załącznik 3: Zwiększanie komórki o 1

[9] # oznacza że jest to wartość a nie adres