Brainfuck Compiler 64 - Programowanie bardzo niskopoziomowe.

Grzegorz Koperwas

23 stycznia 2021

Jak napisać kompilator?

Zanim zaczniemy pisać kompilator musimy sobie odpowiedzieć na parę pytań:

Jak napisać kompilator?

Zanim zaczniemy pisać kompilator musimy sobie odpowiedzieć na parę pytań:

Czemu? By móc mówić że napisałem kompilator.

Jak napisać kompilator?

Zanim zaczniemy pisać kompilator musimy sobie odpowiedzieć na parę pytań:

Czemu? By móc mówić że napisałem kompilator.

Czego? Brainfuck'a - języka stworzonego by twórca mógł mówić że napisał kompilator w 256 bajtach.

Jak napisać kompilator?

Zanim zaczniemy pisać kompilator musimy sobie odpowiedzieć na parę pytań:

Czemu? By móc mówić że napisałem kompilator.

Czego? Brainfuck'a - języka stworzonego by twórca mógł mówić że napisał kompilator w 256 bajtach.

Na co? Commodore 64 - Ten 40 letni assembler nie może być taki trudny.

Brainfuck 101

Brainfuck - jak sama nazwa wskazuje nie jest zbytnio czytelnym językiem.
Brainfuck nie posiada koncepcji zmiennej, zamiast tego oferuje nam *dużą* taśmę z komórkami na zmienne liczbowe.

Jego cała składnia składa się nie z słów, jak w c++, tylko z paru znaków:

- $<$, $>$ Przesuń taśmę w lewo/prawo
- $+$, - Dodaj/Odejmnij 1 od komórki pamięci
- ., , Wypisz/wczytaj znak do komórki pamięci
- [,] Pętla `while (komórka != 0)`.

Przykładowy program:

```
1  ++++++++
2  [
3      >++++
4      [
5          >++>+++>+++>+<<<<-
6      ]
7      >+>+>->>+
8      [<]
9      <-
10 ]
11 >>.>---.+++++++..+++.>>.<-.<.+++.------.--------.>>+.>++.
```

Przykładowy program:

```
++++++++
[
    >++++
    [
        >++>+++>+++>+<<<<-
    ]
    >+>+>->>+
    [<]
    <-
]
>>.>---.+++++++..+++.>>.<-.<.+++.------.--------.>>+.>++.
```

Jak ktoś pomyślał że to „Hello World” to gratulacje.

Jak działa BFC64?

BFC64 składa się z trzech części:

Parsera Zamienia on znaki bf na `Symbole`

Jak działa BFC64?

BFC64 składa się z trzech części:

Parsera Zamienia on znaki bf na `Symbole`

Optymalizatora Procesor może dodawać liczby, można to wykorzystać.

Jak działa BFC64?

BFC64 składa się z trzech części:

Parsera Zamienia on znaki bf na `Symbole`

Optymalizatora Procesor może dodawać liczby, można to wykorzystać.

Kompilator Generuje kod assemblera w oparciu o wzorce

Jak działa BFC64?

BFC64 składa się z trzech części:

Parsera Zamienia on znaki bf na `Symbole`

Optymalizatora Procesor może dodawać liczby, można to wykorzystać.

Kompilator Generuje kod assemblera w oparciu o wzorce

Jak działa BFC64?

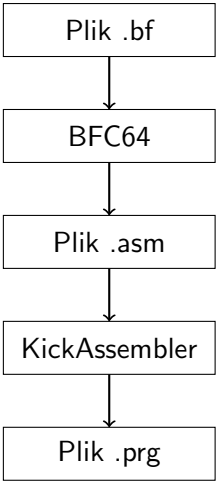
BFC64 składa się z trzech części:

Parsera Zamienia on znaki bf na `Symbole`

Optymalizatora Procesor może dodawać liczby, można to wykorzystać.

Kompilator Generuje kod assemblera w oparciu o wzorce

Funkcje assemblera oraz linkera spełnia `kickassembler`.



Czym są te wzorce

Kompilator w BFC64 działa poprzez wklejanie za kolejne `symbole` fragmentów assemblera.
Wykorzystuje on również stos w celu przydzielania odpowiednich referencji końcom pętli (koniec pętli musi wiedzieć gdzie się ona zaczyna)

Czym są te wzorce

Kompilator w BFC64 działa poprzez wklejanie za kolejne `symbole` fragmentów assemblera.
Wykorzystuje on również stos w celu przydzielania odpowiednich referencji końcom pętli (koniec pętli musi wiedzieć gdzie się ona zaczyna)

Same fragmenty assemblera wkleja w odpowiednie miejsca skrypt *pythona* podczas kompilacji. Czyta on pliki z odpowiedniego folderu i zamienia on je na funkcje c++ za pomocą wzorca.

Czemu skrypt pythona?

```
69  std::string
70  subtract(std::string value, std::string label, std::string label2)
71      {
72          return "    lda ($fb),y\n    clc\n    cmp #" + value
73                 + "\n    bcs " + label + " //if lower than "
74                 + "number\n    lda #$00\n    jmp " + label2
75                 + "\n" + label + ":\n    sec\n    sbc #" + value
76                 + "\n" + label2 + ":\n    sta ($fb),y\n";
77      }
```

Podstawy architektury MOS 6510 w Commodore 64

Co musimy znać by móc programować w assemblerze:

Podstawy architektury MOS 6510 w Commodore 64

Co musimy znać by móc programować w assemblerze:

- Zestaw instrukcji procesora

Podstawy architektury MOS 6510 w Commodore 64

Co musimy znać by móc programować w assemblerze:

- Zestaw instrukcji procesora
- Urządzenia wejścia wyjścia

Podstawy architektury MOS 6510 w Commodore 64

Co musimy znać by móc programować w assemblerze:

- Zestaw instrukcji procesora
- Urządzenia wejścia wyjścia
- Mapę pamięci komputera

Podstawy architektury MOS 6510 w Commodore 64

Co musimy znać by móc programować w assemblerze:

- Zestaw instrukcji procesora
- Urządzenia wejścia wyjścia
- Mapę pamięci komputera
- Kernal lub bios lub system operacyjny (cokolwiek jest dostępne)

Podstawy architektury MOS 6510 w Commodore 64

Co musimy znać by móc programować w assemblerze:

- Zestaw instrukcji procesora
- Urządzenia wejścia wyjścia
- Mapę pamięci komputera
- Kernal lub bios lub system operacyjny (cokolwiek jest dostępne)

Commodore 64 nie posiada żadnego systemu operacyjnego, posiada za to BASIC'a oraz Kernal w pamięci ROM.

Podstawy architektury MOS 6510 w Commodore 64

Co musimy znać by móc programować w assemblerze:

- Zestaw instrukcji procesora
- Urządzenia wejścia wyjścia
- Mapę pamięci komputera
- Kernal lub bios lub system operacyjny (cokolwiek jest dostępne)

Commodore 64 nie posiada żadnego systemu operacyjnego, posiada za to `BASIC`'a oraz Kernal w pamięci `ROM`.

Kernal jest takim odpowiednikiem *biblioteki systemowej*, zawiera funkcje do łatwiejszego wypisywania znaków na ekran i wczytywania ich z klawiatury.

Procesor MOS 6510/6502

Procesor w Commodore 64 posiada cztery ośmiobitowe rejestry:

Procesor MOS 6510/6502

Procesor w Commodore 64 posiada cztery ośmiobitowe rejestry:

- X oraz Y - rejestry którymi możemy indeksować pamięć
- A - Akumulator, na nim są przeprowadzane operacje matematyczne
- S - wskaźnik do stosu (rozmiar to 256 bajtów/128 adresów)

Procesor MOS 6510/6502

Procesor w Commodore 64 posiada cztery ośmiobitowe rejestry:

- X oraz Y - rejestry którymi możemy indeksować pamięć
- A - Akumulator, na nim są przeprowadzane operacje matematyczne
- S - wskaźnik do stosu (rozmiar to 256 bajtów/128 adresów)

Procesor może dodawać oraz odejmować. Może Dokonywać operacji bitowych (and, or, shift, rotate) oraz porównywać liczby, wynik zapisuje jako flagi.

Procesor MOS 6510/6502

Procesor w Commodore 64 posiada cztery ośmiobitowe rejestry:

- X oraz Y - rejestry którymi możemy indeksować pamięć
- A - Akumulator, na nim są przeprowadzane operacje matematyczne
- S - wskaźnik do stosu (rozmiar to 256 bajtów/128 adresów)

Procesor może dodawać oraz odejmować. Może Dokonywać operacji bitowych (and, or, shift, rotate) oraz porównywać liczby, wynik zapisuje jako flagi.

Procesor jest 8 bitowy, ale posiada 16 bitową szynę danych (max 64 kilobajty pamięci). Przez to wskaźniki mogą być tylko w pamięci *zeropage* (pierwsze 256 bajtów).

Assembler 6502 101

Assembler używa mnemonik, skrótów nazw instrukcji procesora, na przykład:

Assembler 6502 101

Assembler używa mnemonik, skrótów nazw instrukcji procesora, na przykład:

`LDA` LoaD A

`JSR` Jump SubRoutine

`INY` INcrease Y

Assembler 6502 101

Assembler używa mnemonik, skrótów nazw instrukcji procesora, na przykład:

`LDA` LoaD A

`JSR` Jump SubRoutine

`INY` INcrease Y

Assembler nie posiada pętli, mamy do dyspozycji tylko skoki (goto) lub skoki warunkowe.

Assembler 6502 101

Assembler używa mnemonik, skrótów nazw instrukcji procesora, na przykład:

`LDA` LoaD A

`JSR` Jump SubRoutine

`INY` INcrease Y

Assembler nie posiada pętli, mamy do dyspozycji tylko skoki (goto) lub skoki warunkowe.

Różne mnemoniki posiadają różne sposoby adresowanie argumentów, mogą być to sztywne adresy, adresy względne, indeksowanie rejestrami czy indeksowanie wskaźników rejestrami.

```
1  BasicUpstart2(start)       // tell BASIC to start execution at 'start'
2  *=$4000 "Program"          // at memory $4000 place
3  start:
4      ldy #$00               // store 0 to register Y
5  print_loop:
6      lda string, Y          // load to accumulator (string + Y)*
7      cmp #$00               // compare it to 0
8      beq loop               // if equal goto loop
9      jsr $ffd2              // jump to kernal function for printing
10     iny                    // Y++
11     jmp print_loop         // goto print_loop
12 loop:
13     jmp loop               // goto loop
14 string:
15     .text "HELLO, WORLD!" // place text in memory
16     .byte 0                // cstring ends with a zero byte
```

Jak wyglądają wzroce w porównaniu do c++

Mamy taki wzorzec:

```
    lda ($fb),y // load from (($fb)* + y)*
    cmp #$ff    // compare to 255
    beq label() // if acc == 255 go to label
    clc         // clear carry flag
    adc #$01    // acc = acc + 1 + carry
    sta ($fb),y // store to (($fb)* + y)
label():
```

Gdzie:

- `$FB` - pointer do pointera taśmy
- Rejestr `Y` to indeks taśmy
- Za `label()` kompilator wsadza jakąś nazwę

Jak wyglądają wzroce w porównaniu do c++

Mamy taki wzorzec:

```
    lda ($fb),y // load from (($fb)* + y)*
    cmp #$ff    // compare to 255
    beq label() // if acc == 255 go to label
    clc         // clear carry flag
    adc #$01    // acc = acc + 1 + carry
    sta ($fb),y // store to (($fb)* + y)
label():
```

Gdzie:

- `$FB` - pointer do pointera taśmy
- Rejestr `Y` to indeks taśmy
- Za `label()` kompilator wsadza jakąś nazwę

W c++:

```
if (tapeptr[i] < 255)
    tapeptr[i]++;
```

Gdzie:

- `tapeptr` - `int*` - wskaźnik do taśmy
- `i` - `int` - indeks taśmy

Inne wzorce

Dla każdego symbolu robimy kolejne wzorce:

- $<$, $>$ - Zmniejsz/zwiększ rejestr `Y`, jeśli jest równy od 0/255, zmniejsz/zwiększ bardziej znaczący bajt pointera.
- . - Załaduj wartość do `A`, skocz do funkcji wypisującej z Kernala.
- , - Skacz do funkcji wczytującej z Kernala aż `A != 0`
- [- Załaduj wartość do `A`, jeśli `A == 0` to skocz do końca pętli
-] - Skocz do początku pętli

Po tym wszystkim pozostaje jedno pytanie:

Po tym wszystkim pozostaje jedno pytanie:

W czym napisano assemblera?

Papier++ go brrrrr

Materiały

- Filmiki Ben'a Eater'a o budowaniu komputera i karty graficznej na płytkach prototypowych
- Zestaw instrukcji 6502
- Dokumentacja KickAssemblera
- Dokumentacja oraz kod na Github'ie