Architektura komputerów 2

Projekt

Funkcje w języku Asemblera pozwalające na obliczenie iloczynu i ilorazu dużych liczb (>1024b)

Anna Antończak, 226168
Hanna Grodzicka, 226154

1. Wstęp.

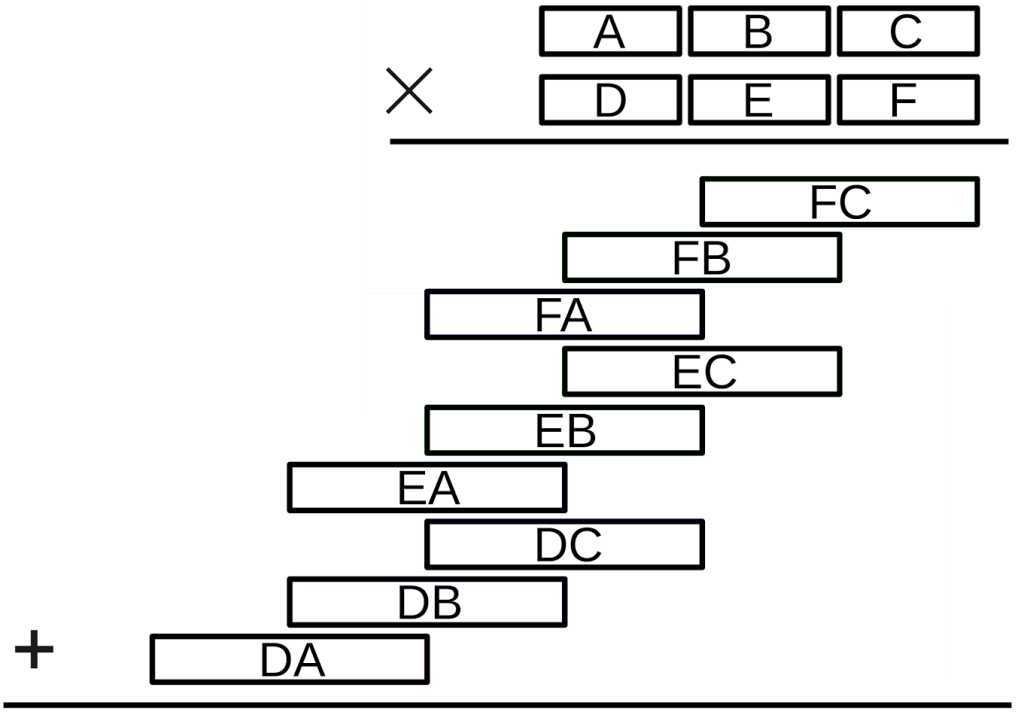
Zaimplementowane operacje arytmetyczne to dzielenie i mnożenie. Algorytm obliczania iloczynu wymagał również opracowania dodawania dla dużych liczb.

Zadanie zostało zrealizowane wykorzystując język Asemblera w składni AT&T oraz kompilator GNU Assembler w systemie Linux w architekturze 64-bitowej.

Obie operacje korzystają z dostępnych rozkazów takich jak *mul* czy *div*. Wykorzystywane liczby (w zależności od operacji mnożna i mnożnik lub dzielna i dzielnik) są zadeklarowane bezpośrednio w programie jako zmienne globalne. Wynik każdej z operacji jest podawany na strumień wyjścia do pliku.

2. Iloczyn dużych liczb.

2.1. Algorytm mnożenia.



Algorytm mnożenia obejmuje tworzenie iloczynów częściowych poprzez przemnażanie kolejnych 64-bitowych bloków za pomocą instrukcji *mul*. Każda suma częściowa o długości 128 bitów jest od razu dodawana do bufora z wynikiem końcowym na pozycję o odpowiednim indeksie. W przypadku generacji przeniesienia jest ono dodawane do następnego 64-bitowego bloku z bufora wyniku w pętli tak długo, dopóki kolejne bloki propagują takie przeniesienie.

Mnożenie to działa dla liczb dodatnich.

2.2. Opis programu.

2.2.1. Deklaracja sekcji danych i opis wczytywania liczb z pliku.

Zadeklarowane zostały nazwy plików wejściowych i wyjściowych oraz bufory i ich długości.

Początkowo liczby deklarowane były w kodzie, a następnie przenoszone do odpowiednich buforów za pomocą instrukcji

kopiującej *rep movsq*. Polecenie to przenosi zawartość bufora spod adresu w rejestrze *rsi* do bufora wskazanego przez adres umieszczony w *rdi*. Przenoszone są bloki 64-bitowe, a liczbę powtórzeń tych operacji określa licznik w rejestrze *rcx*, który osiągnąwszy zero wychodzi z instrukcji.

W następnym etapie projektu dopisano funkcję wczytującą liczby z dwóch osobnych plików i zapisującą wynik do trzeciego. Ostatecznie deklaracja nazw plików wejściowych i wyjściowych oraz buforów i ich długości wygląda następująco:

```
.data
# Deklaracja nazw plikow
multiplicand_in:     .ascii "multiplicand.txt\0"    # mnozna
multiplier_in:       .ascii "multiplier.txt\0"      # mnoznik
result_out:          .ascii "result.txt\0"          # wynik

# Dlugosci wczytanych liczb
mnozna_len:    .quad 0
mnoznik_len:   .quad 0
BUF_LEN = 1024
first:         .fill 512          # mnozna
mnozna:        .fill 1024         # mnozna odczytana z pliku
second:        .fill 512          # mnoznik
mnoznik:       .fill 1024         # mnoznik odczytany z pliku
partial:       .fill 1024         # suma czastkowa
result:        .fill 2048         # wynik do zapisu do pliku
```

Bufory *mnozna* i *mnoznik* są używane do wczytania liczb z plików. Ich długość wynika z ilości pamięci zajmowanej przez jeden kod ASCII równy jednemu bajtowi. Następnie zostają one przepisane do buforów *first* i *second,* w których najpierw następuje konwersja kodów ASCII na liczby heksadecymalne, a następnie sklejenie liczb - każde dwie cyfry zostają zapisane na jednym bajcie. Stąd wynika dwa razy krótsza długość tych buforów w porównaniu do *mnozna* i *mnoznik.* Do sklejenia ze sobą dwóch liczb wykorzystano przesunięcie bitowe w lewo o 4 pozycje.

Przy przepisywaniu wszystkich buforów istotna była kolejność bajtów, ponieważ dane wczytane z pliku są ułożone w konwencji *big endian*. Zaimplementowane mnożenie działa jednak dla *little endian*,

dlatego konieczne było zapisywanie najmłodszych bajtów liczb w miejsca najmłodszych adresów w buforach *first* i *second* podczas wspomnianej konwersji na ASCII.

Bufor *partial* to pomocniczy bufor, w którym dodawane są sumy częściowe z operacji mnożenia. Jego zawartość w formie liczbowej, na koniec działania programu zostaje przepisana na ASCII do bufora *result*, tak aby możliwe było jego zapisanie do pliku.

2.2.2. Wywołanie głównej funkcji.

```
.type main_function, @function
main_function:
call    iloczyn

# %r11 - dlugosc mnoznej
# %r12 - dlugosc mnoznika
.type iloczyn, @function
iloczyn:
pushq     %rbp
movq      %rsp, %rbp

# zaokraglanie mnoznej
xorq      %rdx, %rdx
movq      %r11, %rax
movq      $8, %rcx
idiv      %rcx
movq      %rax, %r9
```

Na początku funkcji wykonującej mnożenie obliczana jest długość mnożnej tak, aby umożliwić posłużenie się nią jako licznikiem wykonywanych mnożeń w dalszej części programu. W tym celu pierwotna długość liczby danej w bajtach zostaje podzielona przez 8. W ten sposób otrzymujemy zaokrągloną w dół długość *quad* (64 bity). Przekazana wartość trafia do rejestru *R9*.

Analogiczne działanie (zaokrąglenie do długości 64 bitów) zostało wykonane dla mnożnika. Wynik z tej operacji trafił do *R10*.

```
# mnozenie: rax * rdx = rdx | rax
movq     %r9, %rcx          # maksymalny indeks FIRST
movq     %r10, %rdi         # maksymalny indeks SECOND
leaq     first, %r8
leaq     second, %r9
```

```
leaq    partial, %r10
xorq    %rsi, %rsi          # licznik petli wewnetrznej
xorq    %rbx, %rbx          # licznik petli zewnetrznej
```

Za pomocą komendy *leaq* do rejestrów zostają przekazane adresy buforów, w których znajduje się mnożnik i mnożna (*first* i *second*). Posłużą one jako bazy w adresowaniu pośrednim we właściwym algorytmie mnożenia.

2.2.3. Mnożenie.

```
# %r9 - adres mnoznej
# %r8 - adres mnoznika
outer_loop:
movq    (%r8,%rbx,8), %r13

inner_loop:
movq    %r13, %rdx
movq    (%r9,%rsi,8), %rax
mulq    %rdx
# rdx | rax    <- wynik mnozenia
pushq   %rsi
pushq   %rbx
call    dodawanie
popq    %rbx
popq    %rsi
incq    %rsi
cmpq    %rdi, %rsi
jle     inner_loop

xorq    %rsi, %rsi
incq    %rbx
cmpq    %rcx, %rbx
jle     outer_loop
```

Powyżej zaprezentowany kod obejmuje cały główny algorytm mnożenia. W pętli zewnętrznej następuje wpisywanie do rejestru *R13* kolejnych 64-bitowych bloków mnożnej zaczynając od początku bufora *first*, ponieważ tam znajdują się najmniej znaczące bajty liczby (konwencja *little endian*). Ta pętla pozwala na poruszanie się po kolejnych blokach mnożnika, a jej licznik równy jest jego długości.

W pętli wewnętrznej wykonują się kolejne mnożenia. Ta pętla pozwala na poruszanie się po kolejnych blokach mnożnej, a jej

licznik równy jest długości mnożnej.

Analizując rysunek z punktu 2.1 dla pierwszego obrotu pętli wewnętrznej otrzymana suma częściowa to blok *FC*. Dla drugiego obrotu jest to *FB*, dla trzeciego *FA* itd. Po każdym mnożeniu dwóch bloków, wynik zostaje dodany w odpowiednie miejsce w buforze *partial.*

2.2.4. Dodawanie.

```
.type dodawanie, @function
dodawanie:
pushq   %rbp
movq    %rsp, %rbp
addq    %rbx, %rsi    # suma licznikow obu petli (wew. i zew.)

continue:
# wynik mnozenia w: rdx | rax
addq    (%r10,%rsi,8), %rax
movq    %rax, (%r10,%rsi,8)
incq    %rsi
adcq    (%r10,%rsi,8), %rdx
movq    %rdx, (%r10,%rsi,8)
jnc     end
```

W funkcji dodawania wykorzystano liczniki pętli mnożenia - zauważono, że blok wyniku mnożenia z mniej znaczącymi bajtami znajdujący się w rejestrze *rax* należy dodać w miejscu, którego indeksem będzie suma liczników pętli zewnętrznej i wewnętrznej. Następnie blok wyniku mnożenia z bardziej znaczącymi bajtami znajdujący się w rejestrze *rdx* zostaje dodany 64 bity dalej.

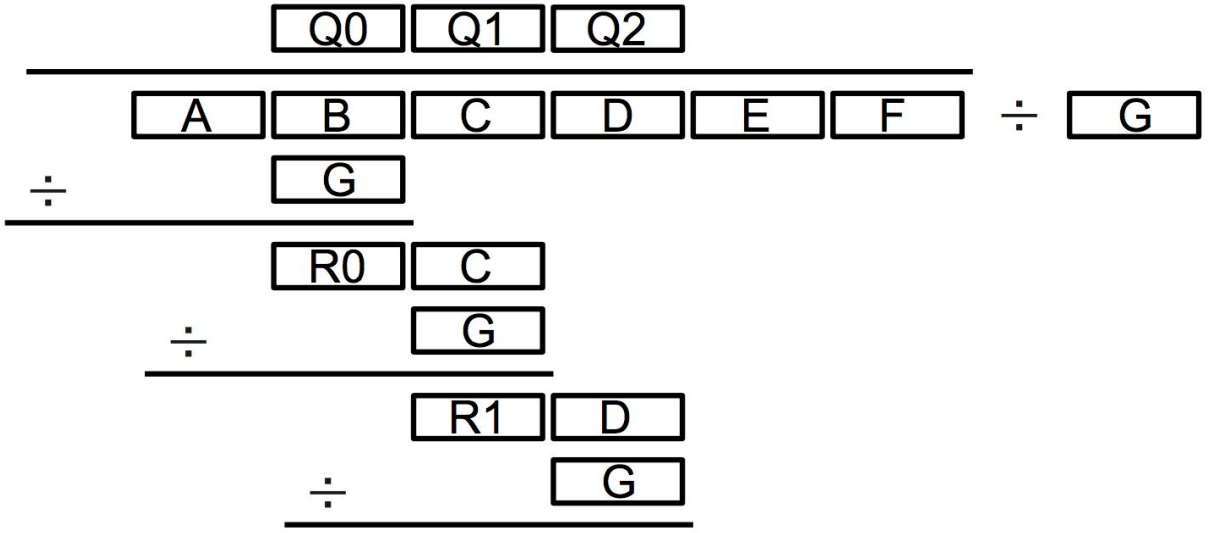
```
flag_loop:
movq    $0, %rax
incq    %rsi
adcq    (%r10,%rsi,8), %rax
movq    %rax, (%r10,%rsi,8)
jc      flag_loop

end:
movq    %rbp, %rsp
popq    %rbp
ret
```

Jeśli została ustawiona flaga przeniesienia to przeniesienie jest dodawane do kolejnego bloku w *partial* lub propagowane dalej w pętli przez kolejne bloki tego bufora aż do momentu, gdy zostanie ono wygaszone. Instrukcja warunkowa *jc* to skok we wskazaną etykietę, jeśli ustawiona jest flaga przeniesienia (*jump if carry*).

3. Iloraz dużych liczb

3.1. Algorytm dzielenia.



Algorytm dzielenia polegał na dzieleniu kolejnych 128-bitowych bloków przez dzielnik mieszczący się w jednym rejestrze (64-bity). Do dzielenia wykorzystano instrukcję *div*. Wynik każdego dzielenia zapisywany był do bufora z wynikiem końcowym, a reszta (na ilustracji R0, R1) stawała się blokiem ze starszymi bitami w kolejnym 128-bitowym bloku. W przypadku dzielenia nie jest generowane przeniesienie, więc liczby wpisane do bufora z wynikiem określały wynik końcowy operacji.

3.2. Opis programu.

3.2.1. Deklaracja sekcji danych i opis wczytywania liczb z pliku.

```
.data
# Nazwy plikow
dividend_in:  .ascii "dividend.txt\0"
divisor_in:   .ascii "divisor.txt\0"
result_out:   .ascii "result.txt\0"
dividend_len: .quad 0
BUF_LEN = 512
first:    .fill 256       # dzielna
dzielna:  .fill 512       # dzielna odczytana z pliku
second:   .fill 8         # dzielnik liczbowo
dzielnik: .fill 16        # dzielnik odczytany z pliku
quotient: .fill 128       # wynik dzielenia
```

```
quotient_buf_len = 128
result:   .fill 256        # wynik do zapisu do pliku
```

Deklaracja zmiennych w przypadku dzielenia wygląda bardzo podobnie jak w przypadku mnożenia. Bufory różnią się jednak długościami - wynika to z maksymalnej liczby cyfr wyniku zaimplementowanego algorytmu. Bufory *quotient* i *result,* do których zapisany będzie wynik, są równe odpowiednio długości buforów *first* i *dzielna*. Długość bufora *second* wynosi 64 bity i w zaimplementowanym algorytmie dzielnik nie może przekraczać długości jednego rejestru.

Samo wczytywanie liczb zostało zrealizowane tak samo jak dla wcześniej opisanego algorytmu mnożenia.

3.2.2. Wywołanie głównej funkcji dzielenia.

```
.type iloraz, @function
iloraz:
pushq   %rbp
movq    %rsp, %rbp

xorq    %r9, %r9                # rejestr indeksujacy
movq    second(,%r9,8), %r14    # R14 - dzielnik

xorq    %rdx, %rdx
movq    dividend_len(,%r9,8), %rax
movq    $8, %rcx
idiv    %rcx                    # dividennd_len / 8
```

Ze względu na to, że cały dzielnik mieści się w rejestrze, zostaje on zapisany na stałe do *R14*, aby w algorytmie głównym można było wykonywać działania bezpośrednio na nim, bez odwoływania się do bufora. W przypadku dzielnej obliczono licznik do późniejszej iteracji w taki sam sposób jak w przypadku mnożnej i mnożnika w mnożeniu, czyli zaokrąglając go w dół do ośmiu bajtów. Licznik ten (w *rcx*) zawiera indeks ostatniego bloku dzielnej, czyli najstarsze jej bity.

3.2.3. Dzielenie.

```
# dzielenie: rdx|rax : r14  = rax , rdx
movq    $quotient_buf_len, %r12
```

```
subq    $8, %r12
movq    %rax, %rcx          # maksymalny indeks FIRST
leaq    first, %r8          # adres bufora FIRST
cmpq    $0, %rcx
jl      end_iloraz
movq    (%r8,%rcx,8), %rax
decq    %rcx
cmpq    %r14, %rax
jae     zacznij_wczesniej
movq    %rax, %rdx          # rdx|rax = 1.quad|2.quad (z FIRST)
jmp     iloraz_loop

zacznij_wczesniej:          # gdy 1.quad FIRST >= dzielnik,
xorq    %rdx, %rdx
divq    %r14                # dzielenie rdx|rax = 0|1.quad
movq    %rax, quotient(,%r12,)
subq    $8, %r12
```

W algorytmie dzielenia konieczne było przesunięcie się w buforze *first* do adresu wskazującego na ostatni 64-bitowy blok dzielnej, aby dotrzeć do najstarszych bitów liczby. W pierwszym kroku wykonano skalowanie, które polega na porównaniu pierwszego (ale ostatniego w buforze) bloku dzielnej z dzielnikiem. Jeśli dzielnik jest mniejszy, wtedy następuje skok do etykiety *zacznij_wczesniej*, gdzie następuje podzielenie 64 najstarszych bitów dzielnej znajdujących się w *rax* (*rdx* zostaje wyzerowany).

Jeśli dzielnik jest większy od pierwszego bloku dzielnej, algorytm przechodzi od razu do wykonania pętli dzielenia.

Do porównania bloków dzielnika z dzielną została wykorzystana instrukcja *jae*. Działa ona tak samo jak *jge* (*jump if greater or equal*), z tym wyjątkiem, że porównuje wartości bezwzględne liczb (*unsigned*). Nasz projekt zakładał operacje na liczbach w systemie naturalnym heksadecymalnym, a nie uzupełnieniowym.

Kolejne 64-bitowe bloki wyniku dzielenia dopisywane są na koniec bufora *quotient*, którego licznik (w *R12*) dany jest w bajtach. W celu dopisania kolejnego bloku wyniku należy odjąć wartość 8 z licznika, aby przesunąć się o długość *quad*.

Algorytm kończy swoje działanie w przypadku przesunięcia się do ujemnych indeksów z bufora *first*. Taki warunek na początku

dzielenia (nie w pętli) zabezpiecza program przed ewentualnymi błędami, które powstały na etapie wczytywania liczb.

```
iloraz_loop:
cmpq    $0, %rcx
jl      konwersja          # zakonczenie dzielenia

movq    (%r8,%rcx,8), %rax
decq    %rcx
divq    %r14
movq    %rax, quotient(,%r12,)
subq    $8, %r12
jmp     iloraz_loop
```

W pętli *iloraz_loop* wykonuje się cały główny algorytm dzielenia. Na początku sprawdzane jest, czy przesunięto się do ujemnego indeksu bufora *first*, czyli czy wszystkie jej bloki zostały już podzielone. Jeśli tak, to następuje wyjście z pętli i konwersja liczb na kody ASCII. Jeśli nie, to do *rax* przekazany jest kolejny quad dzielnej, podczas gdy w *rdx* znajduje się reszta z dzielenia. Następnie liczba składająca się z tych dwóch rejestrów tj. *rdx*|*rax* dzielona jest przez *R14*, czyli dzielnik.

Wynik jest wpisywany do bufora *quotient* zaczynając od najstarszego jego indeksu lub kontynuując cofanie się w tym buforze, jeśli została wykonana procedura spod etykiety *zacznij_wczesniej*. Tak jak było wcześniej wspomniane, licznik tego bufora jest w bajtach, dlatego jego dekrementacja polega na odejmowaniu wartości 8. Licznik ten jest później wykorzystywany w algorytmie konwersji, w którym każdy obrót pętli to poruszanie się co bajt w *quotient*.

Posługując się mnemonikiem *div*, reszta z dzielenia jest zapisywana do rejestru *rdx*, który jest również wykorzystywany jako górna część dzielnej. Dlatego jedyną operacją jaką należało wykonać, było przekazanie odpowiedniego bloku z dzielnej do *rax*, a po wykonaniu dzielenia przekazanie wyniku z *rax* do bufora *quotient.*

4. Testowanie poprawności.

Otrzymane wyniki z iloczynu i ilorazu były porównywane z rezultatem tych samych działań w Pythonie. Język ten ma zaimplementowane mnożenie i dzielenie dużych liczb, które można wpisywać i wypisywać szesnastkowo za pomocą np. “`print hex (0xABC * 0x123)`”. Zapewniło to proste i bezbłędne testowanie, ponieważ liczbę wpisaną do pliku można było od razu skopiować do Pythona.

Przykładowe testowanie działania:

- Iloczyn

```
[17:15:54-s226154@lak:~/Projekt/Mnozenie $ ls                                                  ]
makefile  multiplicand.txt  multiplier.txt  plik*  plik.s
[17:16:11-s226154@lak:~/Projekt/Mnozenie $ cat multiplicand.txt                                ]
7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
[17:16:22-s226154@lak:~/Projekt/Mnozenie $ cat multiplier.txt                                  ]
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[17:16:26-s226154@lak:~/Projekt/Mnozenie $ ./plik                                              ]
[17:16:31-s226154@lak:~/Projekt/Mnozenie $ ls                                                  ]
makefile  multiplicand.txt  multiplier.txt  plik*  plik.s  result.txt
[17:16:33-s226154@lak:~/Projekt/Mnozenie $ cat result.txt                                      ]
555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555
555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555
555555555555555555555555555555555555555555555555555555555555555555555555555555555555545555555
555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555
555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555
5555555555555555555555555555555555555555555555555555555555555555555555555555556
17:16:35-s226154@lak:~/Projekt/Mnozenie $
```

```
[17:03:36-hg@hvvka:~ $ python                                                                  ]
Python 2.7.13 (default, Mar  4 2017, 18:19:38)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> print hex(0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF]
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFF * 0xAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA)
0x5555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555
555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555
555555555555555555555555555555555555555555555555555555555555555555555555555555555555555455555
555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555
555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555
555555555555555555555555555555555555555555555555555555555555555555555555555555556L
>>> 
```

Liczby z powyższego przykładu są równej długości, która wynosi 1088

bitów. Jak widać na zrzucie ekranu bardziej charakterystyczne liczby wyniku tj. “4” i “6” ustawiły się na prawidłowych pozycjach.

Zaimplementowane mnożenie działa także dla liczb o różnej długości:

```
18:53:11-s226154@lak:~/Projekt/Mnozenie $ cat multiplicand.txt
FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543
210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876
543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9
876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDC
BA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210F
EDCBA9876543210FEDCBA9876543210FEDCBA9876543210
18:53:16-s226154@lak:~/Projekt/Mnozenie $ cat multiplier.txt
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
18:53:22-s226154@lak:~/Projekt/Mnozenie $ ./plik
18:53:26-s226154@lak:~/Projekt/Mnozenie $ cat result.txt
FEDCBA9876543210FEDCBA987654320FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0123456789ABCDEF0123456789ABCDF0
18:53:39-s226154@lak:~/Projekt/Mnozenie $
```

```
>>> print hex(0xFEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA987654
3210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA98
76543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDC
BA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210
FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA987654
3210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210 * 0xFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFF)
0xfedcba9876543210fedcba987654320fffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffff0123456789abcdef0123456789abcdf0L
>>>
```

Powyższy przykład to mnożenie liczby 2048-bitowej ze 128-bitową.

Liczby te nie muszą być też wielokrotnością rejestrów roboczych:

```
19:00:32-s226154@lak:~/Projekt/Mnozenie $ cat multiplicand.txt
9999FFFFAAAA11117654
19:00:42-s226154@lak:~/Projekt/Mnozenie $ cat multiplier.txt
FEDCBA9876543210123456789
19:00:48-s226154@lak:~/Projekt/Mnozenie $ ./plik
19:00:51-s226154@lak:~/Projekt/Mnozenie $ cat result.txt
098EB3C4D097AA97474E468ACDEA128AF7091789981EF4
19:00:54-s226154@lak:~/Projekt/Mnozenie $
```

```
>>> print hex(0x9999FFFFAAAA11117654 * 0xFEDCBA9876543210123456789)
0x98eb3c4d097aa97474e468acdea128af7091789981ef4L
>>>
```

W takim przypadku drukowanie liczby jest wciąż poprawne, ale w wyniku może się pojawić zera na początku (maksymalnie 3).

- Iloraz

W tym wypadku drukowany wynik również nie jest idealny, bo może wystąpić jedno zero na jego początku, ale poprawność jest zachowana.

```
19:08:07-s226154@lak:~/Projekt/Dzielenie $ ls
dividend.txt  divisor.txt  makefile  plik*  plik.s
19:08:12-s226154@lak:~/Projekt/Dzielenie $ cat divisor.txt
AAA
19:08:15-s226154@lak:~/Projekt/Dzielenie $ cat dividend.txt
123456789123456789123456789123456789
19:08:23-s226154@lak:~/Projekt/Dzielenie $ ./plik
19:08:28-s226154@lak:~/Projekt/Dzielenie $ ls
dividend.txt  divisor.txt  makefile  plik*  plik.s  result.txt
19:08:33-s226154@lak:~/Projekt/Dzielenie $ cat result.txt
01B5036B845393BAF088BD73F28CC41AC3
19:08:35-s226154@lak:~/Projekt/Dzielenie $
```

```
>>> print hex(0x123456789123456789123456789123456789 / 0xAAA)
0x1b5036b845393baf088bd73f28cc41ac3L
>>>
```

Przykład dla dzielenia dużych liczb tj. 1088 bitów przez 64 bity:

```
19:33:55-s226154@lak:~/Projekt/Dzielenie $ cat dividend.txt
7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
19:34:06-s226154@lak:~/Projekt/Dzielenie $ cat divisor.txt
AAAAAAAAAAAAAAAC
19:34:10-s226154@lak:~/Projekt/Dzielenie $ cat result.txt
BFFFFFFFFFFFFFFE8000000000000002FFFFFFFFFFFFFFFA000000000000000BFFFFFFFFFFFFFFE8000000000000
002FFFFFFFFFFFFFFFA000000000000000BFFFFFFFFFFFFFFE8000000000000002FFFFFFFFFFFFFFFA0000000000
00000BFFFFFFFFFFFFFFE8000000000000002FFFFFFFFFFFFFFFA000000000000000BFFF
19:34:13-s226154@lak:~/Projekt/Dzielenie $
```

```
>>> print hex (0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFF / 0xAAAAAAAAAAAAAAAC)
0xbffffffffffffffe8000000000000002fffffffffffffffa000000000000000bfffffffffffffffe80000000000
00002fffffffffffffffa000000000000000bfffffffffffffffe8000000000000002fffffffffffffffa00000000
0000000bfffffffffffffffe8000000000000002fffffffffffffffa000000000000000bfffL
>>>
```

5. Wnioski

Realizacja projektu wiązała się z dokładnym poznaniem języka Asemblera w składni AT&T. Największą trudnością było wymyślenie algorytmów mnożenia i dzielenia, mając na uwadze główne ograniczenie jakim była wielkość rejestru. W tym celu konieczne było odświeżenie wiedzy z kursu *Arytmetyka komputerów 1*. Proces implementacji poszczególnych funkcji również nie należał do najłatwiejszych ze względu na język Asemblera poznany dopiero w tym semestrze.

Podczas tworzenia projektu konieczne było zrozumienie architektury procesora z rodziny x86, a w szczególności formy zapisu danych w konwencji *little endian*. Pomógł nam w tym żmudny, ale najefektywniejszy sposób na znalezienie błędu w programie - proces debugowania.

Dzięki testom, które były przeprowadzane bardzo rzetelnie, na różnych liczbach o różnych długościach, udało nam się wykryć wyjątki, w których algorytm nie działał poprawnie. Było to spowodowane np. niepoprawnie obliczonymi licznikami pętli czy nieuważnym nadpisywaniem rejestrów.

Bufory wykorzystywane w obu programach były tworzone za pomocą dyrektywy *.fill*, która fizycznie wypełnia zerami takie bufory jeszcze przed uruchomieniem pliku wykonywalnego. Taki plik waży więcej o tyle bajtów ile zostało zadeklarowanych w sekcji *.data*. Rozwiązanie to nie jest wymogiem w programach - równie dobrze zadziałałoby deklarowanie buforów w sekcji *.bss*.

6. Bibliografia

- *Programming from the Ground Up* - Jonathan Bartlett
- *Intel Assembler CodeTable 80x86*
- https://pl.wikibooks.org/wiki/Asembler_x86
- http://www.zak.ict.pwr.wroc.pl/materials/architektura/
- http://www.zak.ict.pwr.wroc.pl/materials/Arytmetyka%20komputerow/