Politechnika Warszawska, Instytut Telekomunikacji

Sprawozdanie z projektu SYKOM

3 czerwca 2023

Spis treści

1. Indywidualnie przypisane dane

Adresy przestrzeni GPIO i rejestrów udostępnianych przez moduł gpioemu (SYKT_GPIO_ADDR_SPACE, A1, A2, W, L i B) a widoczne przez CPU powiny być następujące:
— SYKT_GPIO_ADDR_SPACE - ustalony na podstawie konfiguracji wewnętrznej QEMU - zgodnie z opisem dla lab1,
— A1 - wyznaczony jako SYKT_GPIO_ADDR_SPACE+0x1D7 (w SYS-FS dostępny przez plik: /sys/kernel/sykt/kjaa1)
— A2 - wyznaczony jako SYKT_GPIO_ADDR_SPACE+0x1E0 (w SYS-FS dostępny przez plik: /sys/kernel/sykt/kjaa2),
— W - wyznaczony jako SYKT_GPIO_ADDR_SPACE + 0x1E8 (w SYS-FS dostępny przez plik: /sys/kernel/sykt/kjaw)
— L - wyznaczony SYKT_GPIO_ADDR_SPACE+ 0x1F0 (w SYS-FS dostępny przez plik: /sys/kernel/sykt/kjal),
— B - wyznaczony jako SYKT_GPIO_ADDR_SPACE + 0x1F8 (w SYS-FS dostępny przez plik: /sys/kernel/sykt/kjab).
Dane przekazywane między aplikacją użytkownika a plikami SYS-FS (czyli jądrem systemu) niech będą przekazywane w reprezentacji: HEX.

2. Moduł Verilog

Moduł "gpioemu" realizuje operacje na sygnałach GPIO na podstawie danych wejściowych z interfejsu CPU, takie jak odczytywanie i zapisywanie danych, operacje mnożenia i zliczania jedynek. Wyniki i stany są zapisywane w odpowiednich rejestrach i przypisywane do sygnałów wyjściowych. Składa się on z następujących bloków:

2.1. Deklaracje portów

Moduł posiada różne wejścia i wyjścia, takie jak sygnał zegara clk, sygnał resetu n_reset, adresy saddress, sygnały sterujące srd i swr, dane wejściowe sdata_in i dane wyjściowe sdata_out. Istnieją również sygnały wejściowe z GPIO gpio_in i gpio_latch, sygnały wyjściowe do GPIO gpio_out i gpio_in_s_insp. Niektóre ze zmiennych wejściowych i wyjściowych są zadeklarowane jako rejestry.

```
module gpioemu(n_reset, saddress, srd, swr, sdata_in, sdata_out, gpio_in, gpio_latch, gpio_out, clk, gpio_in_s_insp);
    input clk;
    input n_reset;
    input [15:0] saddress;
    input srd;
    input swr;
    input [31:0] sdata_in;
    output [31:0] sdata_out;
    input [31:0] gpio_in;
    reg [31:0] gpio_in_s;
    input gpio_latch;
    output [31:0] gpio_in_s_insp;
    output [31:0] gpio_out;
    reg [31:0] gpio_out_s;

    reg [23:0] A1; // pierwsza liczba podawana jest przez ten rejestr (24 bity)
    reg [23:0] A2; // druga liczba podawana jest przez ten rejestr (24 bity)
    reg [31:0] W; // wynik mnozenia (32 bity)
    reg [31:0] L; // liczba jedynek w wyniku
    reg [31:0] B; // stan zleconej operacji

        reg [47:0]  temp;

        integer i = 0;
        integer j = 0;
```

Deklaracje portów

2.2. Inicjalizacja sygnałów

Ten blok kodu inicjalizuje różne zmienne i rejestry do wartości zerowych w momencie, gdy sygnał resetu n_reset spada na zbocze opadające.

```
always @(negedge n_reset)
    begin
        sdata_out_s <= 0;
        gpio_out_s <= 0;
        W <= 0;
        L <= 0;
        B <= 0;
        i <= 0;
        j <= 0;
        temp <= 0;
    end
```

Inicjalizacja sygnałów

2.3. Przepisywanie sygnałów GPIO

Ten blok kodu przepisuje wartość sygnału gpio_in do rejestru gpio_in_s na zboczu rosnącym sygnału gpio_latch.

```
always @(posedge gpio_latch)
    begin
        gpio_in_s <= gpio_in;
    end
```

Przepisywanie sygnałów GPIO

2.4. Odczytywanie i zapisywanie danych

Ten blok kodu odczytuje wartość saddress i przypisuje odpowiednie dane do sygnału wyjściowego sdata_out_s. W zależności od wartości saddress, przypisywane są wartości zmiennych A1, A2, W, L lub B. Jeśli saddress nie pasuje do żadnego z przypadków, sdata_out_s jest ustawiane na 0.

```
always @(posedge swr)
    begin
        if (B != 1) begin
            case(saddress)
                16'h1D8: begin // adres pierwszego argumentu
                    A1 <= sdata_in;
                    W <= 0;
                    L <= 0;
                    B <= 0;
                end
                16'h1E0: begin // adres drugiego argumentu
                    A2 <= sdata_in;
                    W <= 0;
                    L <= 0;
                    B <= 1;
                end
            endcase
        end
    end

always @(posedge srd)
    begin
        case(saddress)
            16'h1D8: sdata_out_s <= A1;
            16'h1E0: sdata_out_s <= A2;
            16'h1E8: sdata_out_s <= W;
            16'h1F0: sdata_out_s <= L;
            16'h1F8: sdata_out_s <= B;
            default: sdata_out_s <= 0;
        endcase
    end
```

Odczytywanie i zapisywanie danych

2.5. Wykonywanie operacji mnożenia i zliczania jedynek

Ten blok kodu wykonuje operację mnożenia i zliczania jedynek, gdy wartość B jest równa 1. Zmienne A1 i A2 są używane do przechowywania argumentów mnożenia. Na podstawie wartości bitów A2, wykonuje się mnożenie poprzez przesunięcie bitowe i dodawanie do zmiennej temp. Następnie, w pętli, zliczane są jedyne bity w temp i wynik jest przypisywany do zmiennej L. Jeśli wartość temp przekracza zakres 32-bitowy, B jest

ustawiane na 2, w przeciwnym przypadku B jest ustawiane na 0. Dodatkowo, zmienne i rejestry są resetowane, a wynik mnożenia jest przypisywany do zmiennej W. Dodatkowo, wartość gpio_out_s jest inkrementowana o 1.

```
always @(posedge clk) begin
    if (B == 1) begin
        temp <= 0;
        for (i = 0; i < 23; i = i + 1) begin
            if(A2[i] == 1)
            begin
                temp = temp + (A1<<i);
            end
        end
        for (j = 0; j < 47; j = j + 1) begin
            L = L + (temp[j] == 1);
        end
        if (temp > 2**32 - 1) begin
            B <= 2;
            i <= 0;
            j <= 0;
            A1 <= 0;
            A2 <= 0;
            W <= temp[47:16];
            gpio_out_s <= gpio_out_s + 1;
        end
        else begin
            B <= 0;
            i <= 0;
            j <= 0;
            A1 <= 0;
            A2 <= 0;
            W <= temp;
            gpio_out_s <= gpio_out_s + 1;
        end
    end
end
```

Wykonywanie operacji mnożenia i zliczania jedynek

2.6. Przypisanie sygnałów wyjściowych

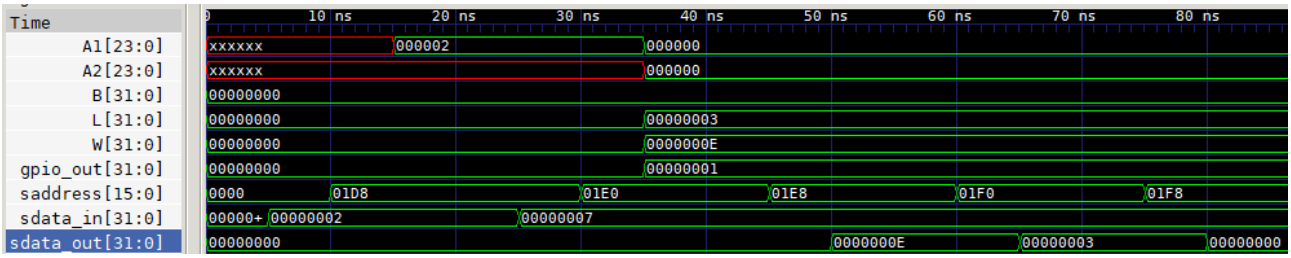
W tym fragmencie kodu następuje przypisanie wartości rejestrów i zmiennych do sygnałów wyjściowych modułu. gpio_out przyjmuje niższe 16 bitów z gpio_out_s. sdata_out jest równa sdata_out_s, a gpio_in_s_insp przyjmuje wartość gpio_in_s.

```
assign gpio_out = gpio_out_s[15:0];
assign sdata_out = sdata_out_s;
assign gpio_in_s_insp = gpio_in_s;
```

Przypisanie sygnałów wyjściowych

3. Testy modułu Verilog

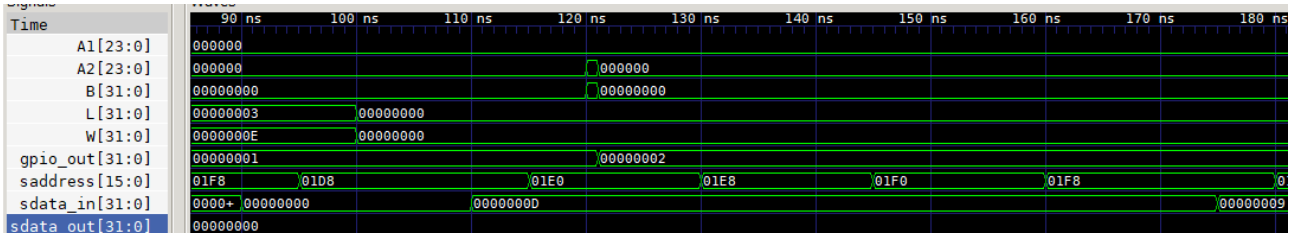
Po utworzeniu modułu Verilog przeszedłem do utworzenia testbench'a, a następnie do sprawdzenia z jego użyciem poprawności działania modułu Verilog.



Zrzut 1: Test *2 * 7 = 14*

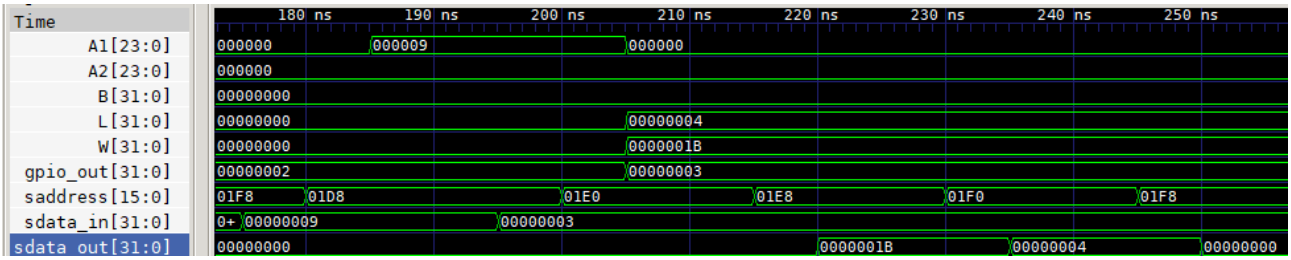
Patrząc na wartości *sdata_in* możemy zauważyć, że oba argumenty zostały prawidłowo wczytane, na prawidłowe adresy- kolejno 0x1D8 oraz 0x1E0. Następnie patrząc na *sdata_out* widać, że na rejestrze 'W' pod adresem

0x1E8 pojawiła się wartość 0xD, czyli zapisana w systemie szesnanstkowym liczba 14, co jest poprawnym wynikiem działania. Dalej, na rejestrze 'L' , którego zadaniem jest zliczanie jedynek, pod adresem 0x1F0 znajduje się liczba 3, co jest prawidłową wartością, ponieważ 14 w reprezentacji binarnej jest równe: *1110*. Następnie, na rejestrze B, pod adresem 0x1F8 pojawia się wartość 0, co oznacza gotowość modułu do wykonywania kolejnych operacji.



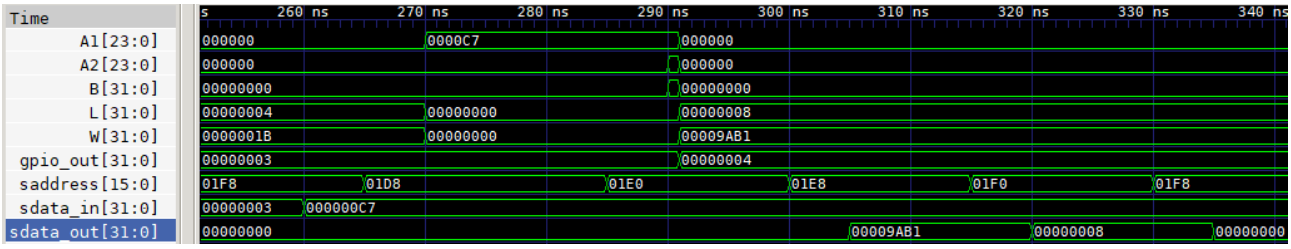
Zrzut 2: Test *0 * 13 = 0*

Podobnie do poprzedniego przykładu, argumenty zostały wczytane w odpowiedni sposób, na odpowiednie adresy. Wyniki na wyjściu również się zgadzają, ponieważ spodziewano się zer na rejestrach: 'W'(0x1E8), 'L'(0x1F0) i 'B'(0x1F8)- operacja została ukończona pomyślnie i moduł jest gotowy do kolejnych działań.



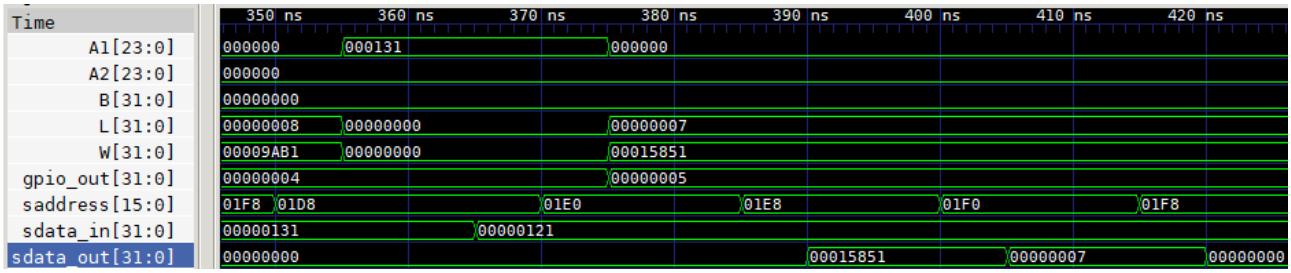
Zrzut 3: Test *9 * 3 = 27*

Analogicznie do pierwszego testu możemy zauważyć, że test przebiegł pomyślnie- dane zostały odpowiednio wczytane, a poprawne wyniki działania trafiły do poprawnych rejestrów: W = 0x1B, L = 4, a B = 0). Warto również zwrócić uwagę na rejestr gpio_out, który zlicza ilość wykonanych operacji.



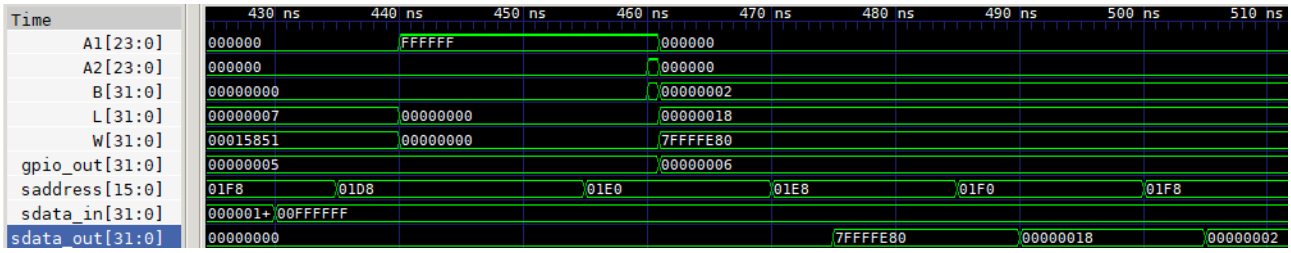
Zrzut 4: Test *199 * 199 = 39601*

W tym teście pomnożone przez siebie zostały dwie duże, takie same liczby. Moduł poradził sobie również z takim działaniem bez problemu, o czym świadczą poprawne wyniki, umieszczone na prawidłowych adresach- liczba 39601 zapisana w systemie szesnastkowym wynosi 9AB1, co jest zgodne z otrzymanym wynikiem, wartość 'L' wynosi 8- 39601 binarnie to 1001101010110001, co oznacza poprawność wyniku, natomiast wartość rejestru B wynosi 0, co oznacza prawidłowe ukończenie wykonywania operacji i gotowość do wykonywania kolejnej.



Zrzut 5: Test *305 * 289 = 88145*

Liczba 88145 w systemie szesnastkowym i binarnym jest równa kolejno: 15851 i 00010101100001010001, co jest zgodne z otrzymanymi wynikami, które umieszczone zostały również na prawidłowych adresach.

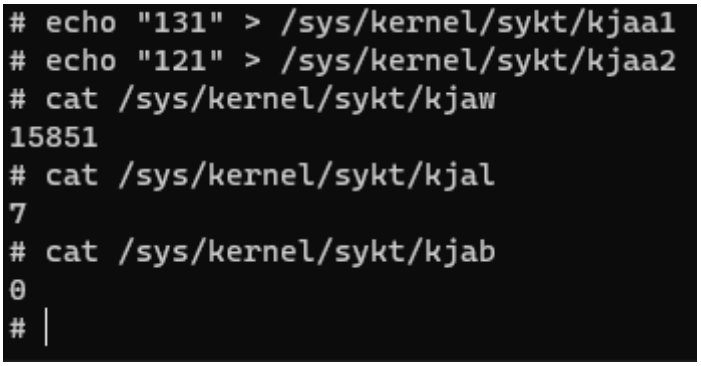


Zrzut 6: Test overflow

W ostatnim teście, uwzględniłem za wysokie liczby, aby spowodować overflow i sprawdzić poprawność jego działania. Dla przykładu podałem dwie takie same liczby 0xFFFFFF które równe są 16777215 w postaci dziesiętnej. Dla przypomnienia maksymalna wartość dla 32-bitowej liczby zapisanej w postaci binarnej to 232 1 czyli 4 294 967 295 (ponad cztery miliardy). Natomiast mnożąc przez siebie dwie takie liczby 16 777 215, otrzymujemy 281 474 943 156 225 czyli około 281 bilionów, co jest bardzo wysoko ponad dopuszczoną liczbę maksymalną. Widać, że kolejny raz argumenty wejściowe zostały ustawione poprawnie, a wynik jest tylko częścią prawdziwego wyniku czyli tylko tym, co zmieściło się na zadanych 32 bitach. Widać także, że jedynki również zostały zliczone, aczkolwiek nie ma to znaczenia, ponieważ wartość na rejestrze 'B(0x2A8)' jasno sygnalizuje nam, że wystąpił błąd i operacja nie została pomyślnie zakończona.

4. Moduł jądra

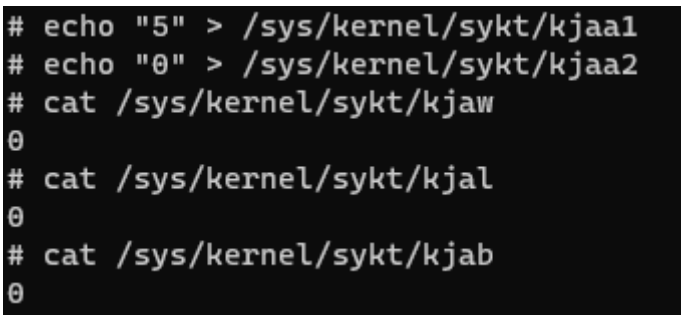
Kod modułu jądra zawiera w sobie najpotrzebniejsze funkcje, m.in:
— Zapisywanie wartośći do rejestrów A1 i A2
— Odczytywanie wartości z rejestrów A1, A2, W, L, B.
W celu sprawdzenia poprawności działania modułu, wykonałem testy jeszcze przed uruchomieniem aplikacji testowej, czego wyniki znajdują się na poniższych zrzutach:



Zrzut 7: Test mnożenia za pomocą modułu jądra: 305 * 289 = 88145(0x15851)

Jak widać, wynik jest poprawny. Należy pamiętać, że wartości zapisywane do rejestrów A1 i A2 muszą być wprowadzone w formacie szesnastkowym- wartości 305 i 289 w formacie szesnastkowym wynoszą kolejno: 131 i 121. Dla pozostałych rejestrów wartości wynosza: *L = 7, B = 0*- wyniki te pokrywają się z tym samym działaniem testowanym wcześniej przy pomocy modułu verilog.

Następnie postanowiłem sprawdzić działanie modułu jądra dla jeszcze jednego przypadku- tym razem zawierającego zero.



Zrzut 8: Test mnożenia za pomocą modułu jądra: 5 * 0 = 0

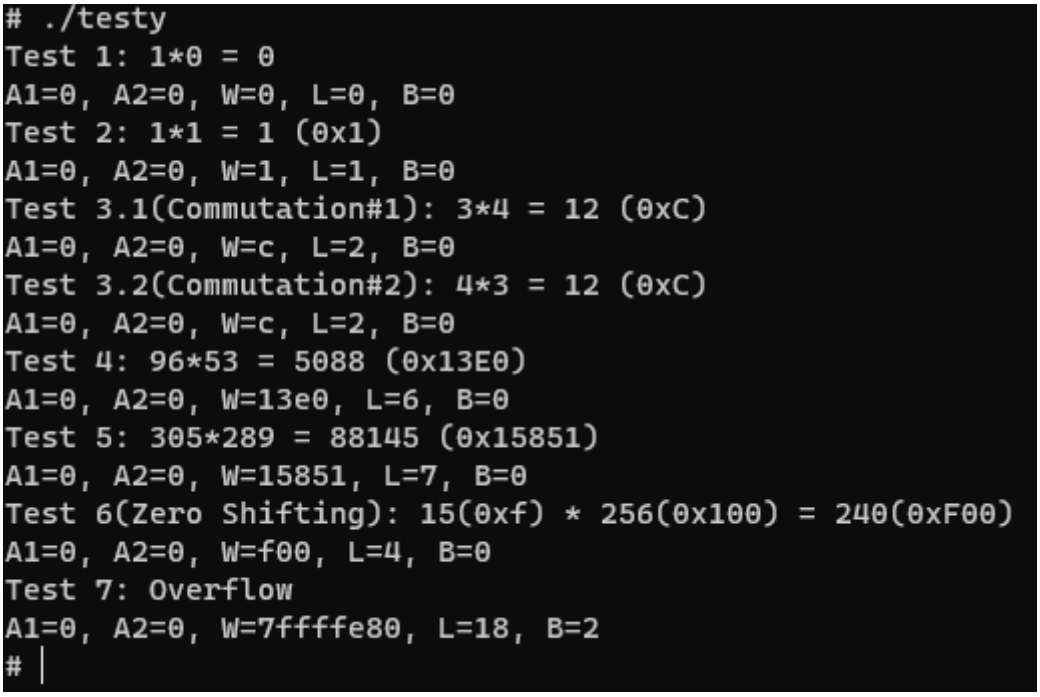
Jak widać wynik operacji się zgadza, rejestry posiadają następujące wyniki: *W = 0, L = 0, B = 0.*

5. Aplikacja Testowa

Po zweryfikowaniu poprawności działania wszystkich pozostałych elementów, do przetestowania została już tylko aplikacja testowa. W celu dokładnego sprawdzenia poprawności działania aplikacji testowej pod uwagę wziąłem kilka wariantów:

— Mnożenie niewielkich liczb
— Mnożenie dużych liczb
— Mnożenie przez zero
— Komutacja
— Overflow

Wynik wywołania aplikacji testowej prezentuje się następująco:



Zrzut 9: Działanie aplikacji testowej

Jak możemy zauważyć na zrzucie, wyniki dla **testów 1 i 2** są zgodne z oczekiwanymi wartościami. W **teście 3** sprawdzono, czy aplikacja poradzi sobie z komutacją- test się powiódł, wyniki były zgodne z oczekiwanymi. Następnie zacząłem przeprowadzać testy dla większych wartości liczbowych. Zarówno **test 4**, jak i **test 5** przebiegły pomyślnie, wyniki były zgodne z oczekiwanymi. **Test numer 6** był testem *zero shiftingu*- w tym przypadku test również się powiódł. Ostatni przeprowadzony test sprawdzał reakcję aplikacji na zbyt duże zadane wartości- jak można zauważyć, wartość rejestru B jest równa 2, co sygnalizuje overflow.

Spis zrzutów