Damian Koper, 241292
SR-TN-17
Wrocław, 08.05.2019r.

Lab 4: FPU, konwencje wywołania, analiza stosu

Cel ćwiczenia

- Złożone działania wykorzystujące FPU - całka oznaczona
- Udostępnienie prostej funkcji na poziomie interfejsu języka C
  - Analiza ramki stosu
  - Konwencja wywołania
- Wyświetlenie śladu stosu
- Pomiar czasu wykonania fragmentu kodu

FPU

FPU jest układem wspomagającym obliczenia, specjalizującym się w obliczeniach na liczbach w formatach stałoprzecinkowych, zmiennoprzecinkowych oraz *packed BCD*. Wyposażony jest w 8 80-bitowych rejestrów, które tworzą stos. Za śledzenie jego szczytu odpowiadają trzy bity `TOP` słowa statusu FPU.

```
(gdb) info float
  R7: Empty   0x00000000000000000000
  R6: Empty   0x00000000000000000000
  ...
=>R0: Empty   0x00000000000000000000
```

Podczas wstawiania wartości na górę stosu `TOP = (TOP-1) mod 8` i w miejsce `TOP` wstawiana jest nowa wartość.

```
(gdb) info float # Po dodaniu liczby 100
Output:
=>R7: Valid   0x4005c800000000000000 +100
  R6: Empty   0x00000000000000000000
  ...
  R0: Empty   0x00000000000000000000
```

Stan i kontrola działania FPU

FPU posiada trzy specjalne rejestry przechowujące jego status i określające sposób jego działania.

Status word

16-bitowe słowo statusu FPU określające aktualny jego stan. Wyróżnione są tam 4 bity flag C0-C3, bit zajętości FPU, wspomniany licznik `TOP`, bit *Interrupt Request*, oraz 7 bitów wyjątków, które moga zostać wyrzucone podczas pracy FPU.

Control word

Control word o długości 16-bit zawiera 12-bitów ustawiające sposób działania FPU. Ustawia sposób interpretowania nieskończoności, tryb zaokrąglenia, precyzję oraz bity określające obsługę wyjątków.

Tag word

Na każde 2 bity tego słowa 16-bitowego przypada numer określający zawartość poszczególnych rejestrów stosu FPU.

ONP - Odwrócona notacja polska

Podczas projektowania algorytmów przydatne okazuje się odpowiednie wyrażenie złożonej operacji arytmetycznej za pomocą jej składowych. Odwrócone notacja polska jest sposobem zapisu działania, gdzie jego operator znajduje się za jego operandami (zapis postfixowy).

```
      2 + 2 -> 2 2 +
  2 + 2 * 2 -> 2 2 2 * +
(2 + 2) * 2 -> 2 2 + 2 *
```

Jednoznacznie określa to kolejność wykonywania działań. Algorytmy wykorzystujące ONP wykorzystują stos, co idealnie sprawdza się w przypadku problemu projektowania algorytmów wykonywanych przez FPU.

Konwencja wywołania `cdecl`

W tej konwencji wywołania fukncji, wykorzystywanej przez większość kompilatorów języka `C`, strona wywołująca jest odpowiedzialna za przekazanie argumentów poprzez stos oraz ich późniejsze z niego zdjęcie. Stałoprzecinkowy wynik funkcji przekazywany jest w rejestrze `eax`, a zmiennoprzecinkowy w `st(0)` FPU. Reszta rejestrów `st(x)` musi być pusta. W tej konwencji funkcja wywoływana jest odpowiedzialna za zachowanie stanów wszystkich rejestrów oprócz `eax`, `ecx` i `edx`.

Programy

- Kody programów `integral`, `my_sinDemo`, `stacktraceDemo`: https://github.com/damiankoper/OiakLab/tree/master/lab_4

Całka sin(x) oraz log10(x)

Program `integral` liczy dwie całki oznaczone: `sin(x)` oraz `log10(x)` pobierając dane wejściowe (A - początek, B - koniec i ilośc przedziałów) ze standardowego wejścia i wyświetlając wynik na standartowe wyjście. Obliczenia wykonywane są metodą trapezów i wykorzystują format 64-bitowy `double` (FLD**L**, Fxxx**L**).

W pierwszej kolejności obliczana jest szerokość przedziału (wysokość trapezu):

```
fildl numOfIntervals
fldl a
fldl b

fsubp
fdivp

fstpl interval # zapisanie przedziału do pamięci
```

co stosując ONP można napisać według kolejności instrukcji FPU:

```
a - początek, b - koniec, i - ilość przedziałów, x - wysokośc trapezu
(a-b) / i = x
ONP: i b a - / (zapisz i zdejmij ze stosu)
```

Całka sin(x)

Na jedną iteracje liczenia całki składają się dwie operacje:

1. Obliczenia pola trapezu i dodanie go do wyniku
2. Przesunięcie punktu `a` o wysokość trapezu

Przed wejściem do pętli należy włożyć na stos FPU 0 za pomoca instrukcji `FLDZ`. To miejsce na stosie będzie sumować pola kolejnych trapezów. Kolejny raz można rozpisać operacje używając ONP:

```
a - początek, b - koniec, x - wysokośc trapezu, w - wynik (już na stosie)

w += (x * (sin(a) + sin(a + x))) / 2
ONP: w x a (sin) a x + (sin) + * 2 / +

a += a + x
ONP: a x + (zapisz a i zdejmij ze stosu)
```

Wykonując pętlę podaną ilość razy na koniec otrzymamy wartość obliczonej całki w `st(0)`. Aby zapis ONP pokrywał się z działaniem FPU należy użyć instrukcji zdejmujących jeden operand ze stosu - instrukcje z literą `P` (`POP`).

Całka log10(x)

Całka z logarytmu obliczana jest analogicznie. Istotną różnicą jest obliczenie samego logarytmu o podstawie 10 (zamiast *sin(x)*). Architektura FPU nie udostępnia instrukcji, która bezpośrednio może policzyć owy logarytm. Skorzystać trzeba z zależności:

```
log10(x) = log2(x) / log2(10)
```

oraz:

```
loga(b) = 1 / logb(a)
```

czyli:

```
log10(x) = log2(x) * log10(2)
```

Wykorzystując instrukcję `FYL2X` możemy obliczyć wynik działania `y * log2(x)`. Mając zatem na uwadze kolejnośc operandów możemy zapisać następujące działanie:

```
a - początek, b - koniec, x - wysokośc trapezu, w - wynik (już na stosie)

w += (x * (log10(a) + log10(a + x))) / 2
w += (x * (log10(2) * log2(a) + log10(2) * log2(a + x))) / 2

ONP: w x log10(2) a (fyl2x) log10(2) a x + (fyl2x) + * 2 / +
```

Stała `log10(2)` ładowana jest za pomocą rozkazu `FLDL2G`.

Pomiar czasu wykonania

Program `integral` odpowiedzialny jest również za pomiar czasu wykonania fragmentu kodu. Elementem pomiaru jest procedura obliczania całki `sin(x)`. Do pomiaru użyto rozkazu `RDTSC`. Rozkaz ten wypełnia rejestry `edx:eax` zawartością rejestru `TSC`, którego zawartość inkrementowana jest po każdym cyklu procesora. Przy długości 64-bit i maksymalnej częstotliwości taktowania procesora *3.0 GHz* rejestr ten nie ulegnie przepełnieniu przez ok. 200 lat. Przez swoją dokładność może być on użyty do ataków czasowych, na którym opierają się między innymi ataki Meltdown i Spectre.

`RDTSC` nie jest instrukcją serializującą - procesor, poprzez spekulatywne wykonywanie kodu, bądź *pipielining*, może odczytać zawartośc rejestru `TSC` zanim wykona poprzednie instrukcje. Aby temu zapobiec trzeba poprzedzić rozkaz `RDTCS` rozkazem `CPUID`, który jest instrukcją serializującą. Alternatywą jest użycie instrukcji `RDTSCP`, która łączy odczyt `TSC` i serializację.

```
# Pomiar czasu
xor %eax, %eax
cpuid
rdtsc
movl %eax, timeTSC1
movl %edx, timeTSC1 + 4
```

Występująca rozbieżność czasów dla tych samych operacji wynika z aktualnego obciążenia systemu operacyjnego, który zarządzając zadaniami, może przełączać kontekst pomiędzy procesami i obsługiwać przerwania. Sprawia to, że w rzeczywistości algorytmy, z punktu widzenia procesora, nie wykonują się jak jeden spójny ciąg instrukcji.

Udostępnienie funkcji na poziomie interfejsu języka C - `my_sin`

Napisany w `C` program `my_sinDemo` wykorzystuje udostępnioną funkcję `my_sin` z poziomu asemblera. Funkcja napisana w asemblerze musi globalnie udostępnić swój symbol dyrektywą:

```
.globl my_sin
```

W celu poinformowania kompilatora o dostępnej zewnętrznej funkcji, w kodzie programu trzeba umieścić jej definicję:

```
extern float my_sin(float a);
```

Parametry i wynik

Parametry funkcji przekazywane są poprzez stos. Ułożone ona są tam w kolejności od najstarszego do najmłodszego.

Wynik funkcji `my_sin`, według konwencji `cdecl`, przekazywany jest w `st(0)`, a reszta rejestrów `st(1) - st(7)` musi być pusta.

Ramka stosu

Ramka stosu tworzona jest dla każdej funkcji i przechowuje jej parametry, określa jej zmienne lokalne, oraz zawiera adres powrotu.

Utworzenie ramki stosu polega na, po wejściu do funkcji, wrzuceniu na stos (zapisaniu) wskaźnika na podstawę stosu, a następnie przypisaniu mu wartości wskaźnika na obecny szczyt stosu.

```
my_sin:
  push %ebp
  mov %esp, %ebp
```

Instrukcją, która automatycznie wykonuje te dwie operacja, a dodatkowo alokuje na stosie miejsce na dane lokalne jest instrukcja `ENTER`.

Tym sposobem, znając położenie podstawy stosu dla funkcji (`ebp`), oraz mając zapisany adres podstawy stosu funkcji wywołującej, możemy łatwo odczytywać i przemieszczać się pomiędzy zagnieżdżonymi ramkami stosu, niezależnie od tego, ile danych funkcja przechowuje obecnie na stosie.

W celu zachowania integralności stosu, oraz zachowania konwencji wywołania, po zakończniu funkcji należy ramkę stosu usunąć. Wykonuje się to analogicznie do procesu jej tworzenia, a równoważną instrukcją jest instrukcja `LEAVE`.

```
  mov %ebp, %esp
  pop %ebp

  ret
```

Analizując stos po wejściu do funkcji `my_sin` i utworzeniu ramki stosu:

```
* - elmenty mogące wystąpić, nieobecne w przypadku funkcji my_sin

   ||    | dane fn. wywołującej |
   ||    | ...                  |
   ||    | dane fn. wywołującej |
   ||    | argument - float a   | 0x3fc90fdb = 1.57
   ||    | adres powrotu        | 0x56555559
   ||    | zapisany ebp         | 0xffffcc38 <-- ebp, esp
   ||    | zmienne lokalne *    |
  \||/   | redzone w AMD64 ABI *|
   \/    | redzone ... *        |
```

Redzone

Konwencja wywołania w *System V AMD64 ABI* opisuje również utrzymanie tzw. *redzone*, która jest 128-bitową przestrzenią znajdującą się za szczytem stosu, a która może być swobodnie używana bez obawy o naruszenie danych przez obsługe sygnałów i przerwań. Funkcja może używać tej przestrzeni do zapisywania danych lokalnych bez dodatkowego narzutu czasowego, który byłby spowodowany modyfikacją wskaźnika na szczyt stosu.

StackTrace - ślad stosu

Program `stacktraceDemo` drukuje na standardowe wyjście ślad stosu - ciąg adresów powrotów, na podstawie których można później uzyskać nazwy funkcji (symboli). Program wywołuje rekurencyjnie funkcję `test`, a przy ostatnim wywołaniu wykonywana jest funkcja `stacktrace`. Drukuje ona wartość zastaną na pozycji `ebp + 4`, która jest adresem powrotu. Za pomocą zapisanego wskaźnika na podstawę stosu poprzedniej ramki, wykonuje te kroki, aż do napotkania wartości zapisanego wskaźnika na podstawę stosu równą `0`, co informuje o napotkaniu ramki niezagnieżdżonej. GCC, tworząc kod za etykietą `_start`, na początku czyści rejestr `ebp`

```
00000400 <_start>:
 400:   31 ed                    xor    %ebp,%ebp
```

```
mov %ebp, %eax
  stackLoop:
    mov 4(%eax), %ebx   # Pobranie adresu powrotu do ebx

    push %eax           # Zachowanie eax
    push %ebx
    push c
    push $formatStr
    call printf
    add $12, %esp
    pop %eax            # Przywrócenie eax

    incl c              # Licznik zagnieżdżenia ramki
    mov (%eax), %eax    # Przejście do kolejnej ramki

    test %eax, %eax
    jnz stackLoop
```

Program generuje następujące dane. Widać, że funkcja `test` wykonana została 5 razy, o czym świadczy powtarzający się adres `0x56555591`:

```
0: 0x5655559d
1: 0x56555591
2: 0x56555591
3: 0x56555591
4: 0x56555591
5: 0x56555591
6: 0x565555ca
7: 0xf7dece81 # adres w dynamicznie ładowanej __libc_start_main
```

Wnioski

Zadania zostały wykonane bez większych trudności.

Podczas analizy wygenerowanego kodu asemblera programu `my_sin` uwagę zwróciło wywołanie funkcji z adresu opisego symbolem z sufixem `@plt`.

```
56d:   e8 3e fe ff ff          call   3b0 <printf@plt>
...
000003b0 <printf@plt>:
3b0:   ff a3 0c 00 00 00       jmp    *0xc(%ebx)
3b6:   68 00 00 00 00          push   $0x0
3bb:   e9 e0 ff ff ff          jmp    3a0 <.plt>
```

Jest to odwołanie do *Procedure Linkage Table*, która razem z *Global Offset Table* odpowiedzialna jest za dynamiczne ładowanie bibliotek. Jeśli adres nie został wcześniej załadowany, wykonany zostaje tu skok do kodu dynamicznego linkera, który, w tym wypadku, w odpowiednim miejscu *GOT* umieści adres funkcji `printf`.

Pozwala to na uniezależnienie lokalizacji adresów owych bibliotek w systemie i utrudnia ataki bazujące na ich znajomości. Ta technika nosi nazwę *Address-Space Layout Randomization (ASLR)*.

Literatura

1. Wikibooks x86 Assembly - https://en.wikibooks.org/wiki/X86_Assembly
2. Laboratorium AK –ATT asembler (LINUX) - http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Linux-AK2-lab-2018%20May.pdf
3. ASLR - http://dustin.schultz.io/how-is-glibc-loaded-at-runtime.html
4. Prezentacja do wykładu
5. RedZone - https://en.wikipedia.org/wiki/Red_zone_(computing)
6. Cdecl - https://en.wikipedia.org/wiki/X86_calling_conventions#cdecl
7. Dokumentacja GDB - https://sourceware.org/gdb/current/onlinedocs/gdb/
8. FPU - http://www.website.masmforum.com/tutorials/fptute/fpuchap1.htm