Systemy komputerowe

Lista zadań nr 4

Karolina Szlęk 300411

Zadanie 1

Zadanie 1. Rejestry %reg1% i %reg2 są tego samego rozmiaru. Wykaż, że niezależnie od zapisanych w nich wartości, interpretowanych jako liczby ze znakiem, instrukcja `cmp %reg1, %reg2` ustawia flagi tak, że `setl %reg3` zadziała zgodnie z oczekiwaniami. Podobnie, wykaż, że jeśli te wartości intepretujemy jako liczby bez znaku, to `setb %reg3` zadziała zgodnie z oczekiwaniami. Wywnioskuj stąd, że pozostałe instrukcje rodziny `set` działają stosownie do swoich sufiksów.

cmp y,x => x-y
Ustawia flagi, nie zapisuje niczego do destination.

Nas będą interesowały następujące :

ZF (Zero Flag) = 1	wtw x-y = 0 (x ==y)
CF (Carry Flag, bez znaku) =1	wtw gdy wystąpi carry lub borrow
SF (Sign Flag, ze znakiem) = 1	wtw x-y<0
OF (Overflow Flag, ze znakiem) = 1	wtw wystąpi overflow/underflow

camp y,x // liczy x-y

setl z // z == 1 wtw. x<y, więc x-y<0

// wpp. 0

setl – 1 wtw x<y (SF ^ OF)

Rozpatrzmy teraz wszystkie przypadki:

y> 0; x> 0; x>y

y< 0; x< 0; x<y

y< 0; x> 0; x - y > INT64_MAX

y< 0; x> 0; x - y < NT64_MAX

y> 0; x< 0; x - y < NT64_MAX

y> 0; x< 0; x - y > NT64_MAX

setb = 1 wtw x<y

Wtedy, gdy nastąpiło borrow w trakcie cmp patrzymy na CF.

x>y CF = 0;

x<y CF =1;

Każda instrukcja z rodziny set działa odpowiednio do swoich sufiksów, ponieważ “sprawdza’ kody warunkowe tak, aby pokryć wszystkie możliwe przypadki.

```

cmp zwraca flagi, setl zwraca 0 albo 1

ZF - Zero Flag

SF - Sign Flag (%reg2 - %reg1) < 0

OF - Overflow Flag (Signed)

CF - Carry Flag (Unsigned)

%reg1 i %reg2 są tego samego rozmiaru.

cmp %reg1, %reg2 (%reg2 - %reg1)

%reg1 i %reg2 interpretujemy jako liczby ze znakiem

setl: SF ^ OF (reg2 > reg1 AND not Overflow) OR (reg2 <= reg1 AND Overflow)

setl %reg3 (SF = 1, gdy %reg2 - %reg1 <0)

załóżmy, że a = reg2, b = reg1

Przypadek 1:

SF = 1, OF = 0

a - b < 0 i nie ma przepełnienia

Przypadek 2:

SF = 0, OF = 1

a - małe

b - duże

więc a - b > 0, overflow, ale a jest mniejsze niż b
```

Przypadek 3:

SF = 1, OF = 1

a - duże

b - małe np. MIN_INT

więc a - b < 0, overflow, ale a nie jest mniejsze niż b (1 ^ 1 = 0)

Przypadek 4:

SF = 0, OF = 0

a jest większe niż b i nie ma przepełnienia więc git xD

%reg1 i %reg2 interpretujemy jako liczby bez znaku

setb: CF

setb %reg3

Porównujemy liczby bez znaku, a porównanie wykonujemy a - b, więc

występuje borrow, kiedy b > a.

Wnioskując w podobny sposób możemy dojść do tego, że każdy z set'ów działa

stosownie do sufiksu.

```

## Zadanie 2

**Zadanie 2.** Poniżej znajduje się kod funkcji o sygnaturze «`void who(int16_t v[], size_t n)`». Przetłumacz go na język C i odpowiedz, jaki jest efekt jego wykonania. Czy znajomość sygnatury jest istotna?

```
1 who: subq $1, %rsi   8   movzwl (%rdx), %r9d
2   movl $0, %eax   9   movw   %r9w, (%rcx)
3 .L2:   cmpq   %rsi, %rax   10   movw   %r8w, (%rdx)
4   jnb   .L4   11   addq $1, %rax
5   leaq   (%rdi,%rax,2), %rcx   12   subq $1, %rsi
6   movzwl (%rcx), %r8d   13   jmp   .L2
7   leaq   (%rdi,%rsi,2), %rdx   14 .L4:   ret
```

Kod funkcji o sygnaturze «void who(int16_t v[], size_t n)».

void who (int16_t v[ ], size_t n){
```

```
        n = n –1;
        int i = 0;
        while (i < n)
        {
                int64_t a = v[i];
                int64_t b = v[n];
                v[i] = b;
                v[n] = a;
                i ++;
                n --;
        }
}
```

Ta funkcja służy do odwracania tablicy (tablica, która miała wartości w kolejności od pierwszej do ostatniej, teraz ma wartości w kolejności od ostatniej do pierwszej).

Znajomość sygnatury jest istotna, ponieważ dzięki nie wiemy czy pierwszym argumentem jest tablica a nie coś innego, choćby pojedyncza zmienna.

```
void who(int16_t v[]/*rdi*/, size_t n /*rsi*/){
   int16_t a, b;              // ?. a = %r8d, b = %r9d
   n--;                       // 1. subq $1, %rsi
   size_t k = 0;              // 2. movl $0, %eax
   while(k < n){              // 3. cmpq %rsi,  %rax
                              // 4. jnb .L4  wtedy, gdy (k=>n), czyli CF=0
      a = v[k];               // znajdz adres rdi, przesuń o k w prawo
                              // 5. leaq(%rdi,%rax,2), %rcx czyli %rcx=v[k]
                              // %rdi + %rax * 2 (czyli k-tą liczbę 2 bajtową)
                              // 6. przypisz v[k] do a
      b = v[n];               // znajdz adres rdi, przesuń o n w prawo
                              // 7. leaq(%rdi,%rsi,2), %rdx czyli %rdx=v[n]
                              // 8. przypisz v[n] do b
```

```
        v[k] = b;           // 9. przypisz b do v[k]
        v[n] = a;           // 10. przypisz a do v[n]
        k++;                // 11.
        n--;                // 12.
    }
}
```

Zadanie 3

Zadanie 3. Poniżej znajduje się kod funkcji o sygnaturze «`bool zonk(char* a, char* b)`», jako argumenty przyjmującej C-owe łańcuchy znaków. Przetłumacz ją na język C (bez instrukcji goto). Jaką wartość liczy ta funkcja?

```
1  zonk:   movl    $0, %ecx
2  .L2:    movslq  %ecx, %rax
3          movzbl  (%rdi,%rax), %edx
4          testb   %dl, %dl
5          je      .L6
6          cmpb    %dl, (%rsi,%rax)
7          jne     .L5
8          addl    $1, %ecx
9          jmp     .L2
10 .L6:    orb     (%rsi,%rax), %dl
11         sete    %al
12         ret
13 .L5:    movl    $0, %eax
14         ret
```

Integer registers

%rax	Return value
%rbx	Callee saved
%rcx	4th argument
%rdx	3rd argument
%rsi	2nd argument
%rdi	1st argument
%rbp	Callee saved
%rsp	Stack pointer
%r8	5th argument
%r9	6th argument
%r10	Scratch register
%r11	Scratch register
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

MOVSLQ is move and sign-extend a value from a 32-bit source to a 64-bit destination.

test S_2, S_1	Set condition codes according to S_1 & S_2
JE rel8	Jump short if equal (ZF=1)

CASSEMBLER

```
bool zonk
{
    (1)%ecx = 0;
    while(1)
    {
        (2)%rax=%ecx;
        (3)%edx=[%rdi+%rax];
        (4)(5) if(%dl==0)
        {
            (10)%dl=([%rsi+%rax] | %dl);
            (11)%al=1;
            (12)return %al;
        }
        (6)(7)if(([%rsi+%rax] != %dl)
        {
            (13)%eax=0;
            (14)return %eax;
        }
        (8)%ecx=%ecx+1;
    }
}
```

Kod funkcji o sygnaturze «bool zonk(char* a, char* b)», jako argu-menty przyjmującej C-owe łańcuchy znaków

bool zonk (char* a, chat* b)

{

int i;

for (i=0; a[i]; i++) {

if (b[i] != a[i]) {

return 0;

}

}

return !(b[i] | a[i]);

}

Ta funkcja zwraca true, gdy ciągi a i b są równe, czyli sprawdza, czy a==b.

---

orb to OR logiczny

Rozwiązanko (Sprawdzanie czy a == b):

``` c

bool zonk(char* a, char* b)
```

```
{
   int iter = 0  // 1.
   for (; a[iter]; iter++){ // 3. %edx = a[iter]
                            // 4. %dl = a[iter] w for jak false
                            //    czyli, gdy a[iter] == NULL to L6.
      if(b[iter] != a[iter]) {    // 6. cmp a[iter], b[iter] i jeśli są
                            //    różne to L5
         return 0;              // 13. %eax=0
                            // 14. ret
      }
   }
   return (b[iter] == NULL);       // 10. bo w %dl jest null
                            // 11. sete %al ustawia wartość ZF w %al
}
```

Zadanie 4. Zastąp instrukcje `pushq %reg1` oraz `popq %reg2` równoważnymi ciągami instrukcji jawnie operującymi na stosie.

Zadanie 4

```
 push %reg1
leaq –8(%rsp), %rsp
movq %reg1, (%rsp)


 popq %reg2
movq (%rsp), %reg2
leaq 8(%rsp), %rsp
```

Funkcje pushq i popq nie ustawiają flag, ale addq i subq już tak (ustawiają flagi). Dlatego zamiast nich skorzystamy z leaq.

Rozwiązanie gorsze z addq i subq:

```
pushq %reg1:                    //Wykonując pushq musisz przesunąć %rsp o 8 w tył.
subq $8,%rsp
movq %reg1,(%rsp)
```

popq %reg2:
movq (%rsp),%reg2
addq $8,%rsp

Zadanie 5

Zadanie 5. Poniżej znajduje się kod funkcji o sygnaturze «`foo(int16_t v[], size_t n)`». Przetłumacz ją na język C. Narysuj ramkę stosu tej funkcji i wytłumacz, jaka jest rola poszczególnych komórek ramki

oraz jak jej zawartość zmienia się w trakcie działania. Jaki jest efekt ma ten kod? Jaka jest rola rejestru %rbp w tym kodzie?

```
1  foo:    pushq   %rbp
2          movq    %rsp, %rbp
3          movq    %rdi, -24(%rbp)
4          movq    %rsi, -32(%rbp)
5          movq    $0, -8(%rbp)
6          jmp     .L2
7  .L3:    movq    -8(%rbp), %rax
8          leaq    (%rax,%rax), %rdx
9          movq    -24(%rbp), %rax
10         addq    %rdx, %rax
11         movzwl  (%rax), %eax
12         leal    (%rax,%rax), %edx
13         movq    -8(%rbp), %rax
14         leaq    (%rax,%rax), %rcx
15         movq    -24(%rbp), %rax
16         addq    %rcx, %rax
17         movw    %dx, (%rax)
18         addq    $1, -8(%rbp)
19 .L2:    movq    -8(%rbp), %rax
20         cmpq    -32(%rbp), %rax
21         jb      .L3
22         nop
23         popq    %rbp
24         ret
```

Wskazówka: Instrukcja nop to tzw. 'no operation', nie ma efektu poza przejściem do wykonania kolejnej instrukcji kodu

Kod funkcji o sygnaturze «foo(int16_t v[], size_t n)»

Narysuj ramkę stosu tej funkcji i wytłumacz, jaka jest rola poszczególnych komórek ramki

oraz jak jej zawartość zmienia się w trakcie działania. Jaki jest efekt ma ten kod? Jaka jest rola rejestru %rbpw tym kodzie.

1. wkładamy rbp na początek stosu
2. rbp ustaw jako wskaźnik na drugie miejsce w stosie
3. wkładamy na 4 miejsce v[]
4. na piąte miejsce wkładamy n
5. zerujemy drugie miejsce na stosie
6. skaczemy do L2

19. wstaw drugie miejsce na stosie do warości return (rax)

20. porównaj n(piąte miejsce) z returnem

21. skacz do L3

     i - drugie miejsce na stosie | 2 * i - iter

-> 7.wstaw i do return (rax)

-> 8. rdx = iter

-> 9. przypisz v.begin do rax(return)

-> 10. rax = v.begin + iter

-> 11. eax (rax) = v[i]

-> 12. edx (rdx) = 2* v[i]

-> 13. wstaw i do return (rax)

-> 14. wstaw iter do rcx

-> 15. rax = v.begin (czwarte miejsce w pamięci)

-> 16. rax = rcx + rax = v.begin + iter

-> 17. v[i](rax) = edx (2*v[i])

-> 18. drugie miejsce na stosie += 1

22. pusta operacja

23. ściągamy rbp ze stosu

24. zwróć coś

```cpp=
void foo(int16_t v[], size_t n) {
    for (size_t i = 0; i < n; i++) {
        v[i] = 2*v[i];
    }
}
```

```cpp=
//'i' to jest wartość -8(%rsp)
foo(int16_t v[], size_t n) {
    rax = 0;

    for (i = 0; i < n; i++) {
        rax = i;
```

```
        rdx = 2 * rax; // leaq (%rax,%rax), %rdx
        rax = v; //wskaźnik na v
        rax = rax+rdx; //&v[i]  (wskaźnik na v[i]) bo v to &v[0] + 2 * i
        rax = *rax; //v[i]
        edx = 2 * rax; //2*v[i]
        rax = i;
        rcx = 2 * rax; //2*i
        rax = v;
        rax = rax+rcx; //v + 2*i = v[i];
        *rax = edx;// v[i] = 2*v[i];
    }
}
```

schemat stosu
old %rbp
saved registers
local variables
addidtional args >= 7

Rola poszczególnych komórek stosu
%rsp ==%rbp <-- %rsp
%rsp-8 <-- i
%rsp-24 <-- v
%rsp - 32 <-- n

Zadanie 6

Zadanie 6. Poniżej znajduje się kod funkcji rekurencyjnej o nieznanej sygnaturze. Przetłumacz tę funkcję na język C, odkryj jej sygnaturę i odpowiedz, jaką wartość ona liczy.

```
1  reccur:
2          pushq   %rbp
3          movq    %rsp, %rbp
4          subq    $16, %rsp
5          movl    %edi, -4(%rbp)
6          cmpl    $0, -4(%rbp)
7          jne     .L2
8          movl    $1, %eax
9          jmp     .L3
8  .L2:    movl    -4(%rbp), %eax
9          subl    $1, %eax
10         movl    %eax, %edi
11         call    reccur
12         imull   -4(%rbp), %eax
13 .L3:    leave
14         ret
```

Wskazówka Instrukcja leave podstawia %rbp pod %rsp oraz wykonuje popq %rbp.

```
reccur:
   pushq %rbp
   movq %rsp, %rbp
   subq $16, %rsp
   movl %edi, -4(%rbp)
   cmpl $0, -4(%rbp)          // jeśli x==0 to ZF=1
   jne .L2                    // if (x != 0) { jump L2
}
   movl $1, %eax              // |
   jmp .L3                    // else return 1
.L2:
   movl -4(%rbp), %eax        // eax = x
   subl $1, %eax              // eax = x-1
   movl %eax, %edi            // rdi = x-1
   call reccur                // wynik w raxie
   imull -4(%rbp), %eax       // eax*reccur(eax-1)
.L3:
   leave
   ret
```

1. Nazwa funkcji
2. Włożenie base pionter na stos (rbp na początek stosu)
3. Zapisanie nowego szczytu stosu w %rbp.
4. Dekrementacja szczytu stosu o 16
5. Zapisanie jedynego argumentu (pierwszego) we właściwym miejscu na stosie.
6. Porównanie z 0
7. Jeżeli równe 0 to
8. %eax = 1
9. Delokacja obecnej ramki stosu i powrót do poprzedniej funkcji zwracając %eax
10. Jeżeli różne od 0 to
11. %eax = %edi (pierwszy argument – argument funkcji)
12. eax -= 1
13. edi = eax

14. wywołaj reccur(%edi) (rekurencyjne wywołanie funkcji z arg %edi)
15. mnożenie obecnego argumentu razy zwrócona wartość wywołania rekurencyjnego - innymi słowy %eax = %eax(z reccur) * -4(%rbp)(obecne %rdi)
(eax bo pod -4(%rbp) mamy nasz edi z poprzedniego wywołania,
a reccur(edi) bo wywołanie rekurencyjnie przypisuje się pod rax (bo to return value))

Język C

```
int reccur(int x) {
  if (x == 0) {
    return 1;
  }
  return x * reccur(x-1);
}
```

Funkcja z zadania wylicza silnie. Jej sygnaturą jest int reccur(int x).

Tłumaczenie ALA	C

1. nazwa funkcji
2. wkładamy base pionter na stos (rbp na początek stosu)
3. rbp ustaw jako wskaźnik na drugie miejsce w stosie
4. przesuń wskaźnik stosu(rsp) o dwa miejsca (na 4 miejsce)
5. na czterech najstarszych bitach (tu -> ____xxxx) drugiego miejsc na stosie(rbp) wstawiamy edi
6. if(5. != 0)
7. then L2
(8,9). else return 1
10. L2:
11. eax = edi (pierwszy argument)
12. eax -= 1
13. edi = eax
14. wywołaj reccur(%edi) (rekurencja więc wynik w raxie (bo to return))
15. eax = eax(z reccur) * -4(%rbp)(obecne %rdi) (eax bo pod -4(%rbp) mamy nasz edi z poprzedniego wywołania, a reccur(edi) bo wywołanie rekurencyjnie przypisuje się pod rax (bo to return value))
	int reccur(int x) {
	  if (x == 0) {
	  return 1;
	  }
	  return x * reccur(x-1);
	}

Zadanie 7

Zadanie 7. Dana jest funkcja o sygnaturze postaci «`int32_t bar(int32_t a1,...,int32_t an)`», gdzie `n` jest nieznane. Jaka jest minimalna wartość `n`, jeżeli wiadomo, że funkcja zwraca wartość jednego ze swoich argumentów, a jej kod wygląda tak

```
1 bar:    pushq   %rbp                    4         movl    16(%rbp), %eax
2         movq    %rsp, %rbp              5         popq    %rbp
3           .....                         6         ret
```

Napisz szkic kodu asemblerowego wywołującego funkcję `bar` z liczbą parametrów równą takiemu minimalnemu `n`. Zadbaj o poprawne przekazanie argumentów do funkcji. Jak zmieni się napisany przez Ciebie kod, gdy `n` będzie większe?

Oto ramka funkcji bar:

Itd..

Ósmy arg. ...

Siódmy arg. (%rsp +16)

Adres powrotu (%rsp +8)

Z wykładu wiemy, że %rsp wskazuje na adres %rbp,

Numer argumentu	Miejsce zapisu	Zawartość rbp po wywołaniu bar
arg1	rdi	rbp -4
arg2	rsi	rbp -8
arg3	rdx	rbp -12
arg4	rcx	rbp -16
arg5	r8d	rbp -20
arg6	r9d	rbp -24
arg7	rbp + 16	rbp + 16

Począwszy od siódmego argumentu, argumenty są zapisywane na stosie rbp.
Już 7 argument jest zapisany w rbp+16.

Widzimy w zadaniu, że kod asemblerowy odnosi się do tego właśnie miejsca w pamięci.
Oznacz to, że funkcja bar musiała otrzymać co najmniej 7 argumentów.

Zatem n>=7.

Asembler – szkic wywołania funkcji bar

```
push    %rbp

mov     %rsp, %rbp

push    $7,
```

```
mov     $6, %r9d
mov     $5, %r8d
mov     $4, %ecx
mov     $3, %edx
mov     $2, %esi,
mov     $1, %edi
call bar
```

Dla większego n, każdy dodatkowy argument byłby pushowany na stos. (przykład)

```
push $10
push $9
push $8
push $7
```