Imię i nazwisko: _____ Numer indeksu: _____

Architektury systemów komputerowych

30 czerwca 2022

czas trwania: 180 minut

Zadanie 1 (10). W prostokąt poniżej wpisz treść procedry w języku `C`, która wykonuje to samo obliczenie, co poniższa procedura `foo` zaprogramowana w asemblerze. Kod w języku `C` może zawierać tylko instrukcje sterujące «`for`» i «`if`». Użycie «`goto`» i «`while`» jest niedozwolone. Parametr «`s`» wskazuje na ciąg 7-bitowych kodów `ASCII`.

```
foo:  movq    %rdi, %rax
.L2:  cmpb    $0, (%rax)
      je      .L4
      incq    %rax
      jmp     .L2
.L4:  decq    %rax
      cmpq    %rax, %rdi
      jnb     .L7
      movb    (%rdi), %dl
      movb    (%rax), %cl
      incq    %rdi
      movb    %cl, -1(%rdi)
      movb    %dl, (%rax)
      jmp     .L4
.L7:  ret
```

```
void foo(char *s) {
  char *e, t;
  for (e = s; *e; e++);
  for (; s < --e; s++)
      t = *s, *s = *e, *e = t;
}
```

Wskazówka: Rejestry `%cl` i `%dl` to najmłodsze bajty odpowiednio rejestrów `%rcx` i `%rdx`. Rozwiązanie wzorcowe ma 4 wiersze.

W prostokąt poniżej wpisz słowne wyjaśnienie, co robi funkcja `foo`.

«`foo`» odwraca w miejscu ciąg znaków «`s`» zakończonym zerem.

Zadanie 2 (8). Przeczytaj poniższy kod w języku `C` i odpowiadający mu kod w asemblerze `x86-64`, po czym wywnioskuj rozmiar struktur `SA` i `SB` oraz wartość stałych `P` i `Q`. W kratce poniżej należy umieścić **zwięzły** opis wnioskowania prowadzący do odpowiedzi.

```
typedef struct {
  int s[P];
  int z;
} SA;

typedef struct {
  SA t[Q];
  int k;
  long y;
} SB;

int foo(SB *p, long i) {
  return p[i].t[p->y].z;
}
```

```
foo:
  lea    (%rsi,%rsi,8),%rax
  lea    (%rsi,%rax,2),%rax
  shl    $0x4,%rax
  add    %rdi,%rax
  mov    0x128(%rdi),%rdx
  lea    (%rdx,%rdx,2),%rdx
  mov    0x14(%rax,%rdx,8),%eax
  retq
```

`P` = 5

`Q` = 12

`sizeof(SA)` = 24

`sizeof(SB)` = 304

Zadanie 3 (10). Posługując się `ABI` dla architektury `x86-64` wyznacz rozmiar struktury `node`, rozmiary pól i ich przesunięcie względem początku struktury. W **pierwszą** kolumnę po lewej stronie wpisz **przesunięcie**, a w **drugą rozmiar** danego pola. W kratkę po prawej stronie należy wpisać zoptymalizowaną wersję struktury i jej rozmiar.

```
                struct packet {
0   | 1   |       char type;
8   | 16  |       struct packet *node[2];
                  union {
                    struct {
24  | 3   |         char n_unit[3];
32  | 8   |         double n_size;
                    };
                    struct {
24  | 1   |         char l_byte;
26  | 2   |         short l_word;
32  | 8   |         long l_long;
                    };
                  };
40  | 4   |       int crcsum;
48  | 8   |       int (*sum)(struct packet *);
                };
```

```
struct packet {
  int (*sum)(struct packet *); //  0   8
  struct packet *node[2];      //  8  16
  union {
    struct {
      int crcsum;              // 24   4
      char type;               // 28   1
    };
    struct {
      char n_pad[5];           // 24   5
      char n_unit[3];          // 29   3
      double n_size;           // 32   8
    };
    struct {
      char l_pad[5];           // 24  5
      char l_byte;             // 29  1
      short l_word;            // 30  2
      long l_long;             // 32  8
    };
  };
};
```

`/* sizeof(struct node) ==` **56** `*/`

`/* sizeof(struct packet) ==` **40** `*/`

Wskazówka: Jeśli chcesz, możesz dołożyć do unii nową strukturę anonimową, a do zagnieżdżonych struktur dodatkowe pola.

Zadanie 4 (8). W kodowaniu liczb *BCD* (ang. *binary-coded decimal*) każde kolejne 4 bity liczby kodują kolejne cyfry dziesiętne. Np. liczba 123 jest kodowana jako `0x123`. Wiele operacji na liczbach *BCD* można wykonać za pomocą prostych operacji bitowych. Uzupełnij poniższe prostokąty, tak by powstała poprawna definicja funkcji `bcddiv2`, która dzieli liczbę w kodowaniu *BCD* przez dwa. W wyrażeniach poza zmiennymi `x` i `m` oraz stałymi możesz użyć wyłącznie operatorów bitowych oraz operatora dodawania, odejmowania i przypisania. Łączna liczba wystąpień operatorów nie może przekraczać 14!

Wskazówka: W rozwiązaniu wzorcowym `bcddiv2` użyto 7 operatorów.

```
uint64_t bcddiv2(uint64_t x) {
  uint64_t m;
  m = x & 0x1111111111111111;
  x >>= 1;
  x -= m >> 4 | m >> 3;
  return x;
}
```

Przykład: `bcddiv2(0x745) = 0x372`.

Imię i nazwisko: _____ Numer indeksu: _____

Zadanie 5 (10). Funkcja `zbl` (ang. *zero byte left*) wyznacza najbardziej znaczący bajt w 32-bitowym słowie, który ma wartość 0. Jeśli `nn` i `dd` oznaczają odpowiednio bajt o niezerowej i dowolnej wartości, to funkcję `zbl` można wyrazić następująco:

$$\texttt{zbl}(x) = \begin{cases} 0, & \text{gdy } x = \texttt{0x00dddddd} \\ 1, & \text{gdy } x = \texttt{0xnn00dddd} \\ 2, & \text{gdy } x = \texttt{0xnnnn00dd} \\ 3, & \text{gdy } x = \texttt{0xnnnnnn00} \\ 4, & \text{gdy } x = \texttt{0xnnnnnnnn} \end{cases}$$

Uzupełnij poniższe prostokąty tak, by powstała poprawna definicja funkcji `zbl`. Najpierw zdefiniuj funkcję pomocniczą `nzb` (ang. *nonzero bytes*), która ustawi na 1 wartość najbardziej znaczącego bitu każdego bajtu wtw., gdy odpowiedni bajt miał niezerową wartość. Reszta bitów może przyjąć dowolne wartości. W wyrażeniach można używać tylko uprzednio zdefiniowanych zmiennych. Poza stałymi możesz użyć wyłącznie operatorów bitowych oraz operatora dodawania i przypisania. Użycie więcej niż 10 instrukcji będzie dyskwalifikowało rozwiązanie. Łączna liczba wystąpień operatorów nie może przekraczać 25!

Wskazówka: W rozwiązaniu wzorcowym `nzb` i `zbl` używają odpowiednio po 3 i 15 operatorów.

```
uint32_t nzb(uint32_t x) {
  x |= (x & 0x7f7f7f7f) + 0x7f7f7f7f;
  return x;
}
```

Przykład: W rozwiązaniu wzorcowym `nzb(0x070050fa) = 0x8f00dffb`, ale wartości nie najstarszych bitów w bajtach mogą być dowolne.

```
uint32_t zbl(uint32_t x) {
  uint32_t t1, t2, t3, t4;
  x = nzb(x);
  t1 = x >> 31;
  t2 = (x >> 23) & t1;
  t3 = (x >> 15) & t2;
  t4 = (x >> 7) & t3;
  x = t1 + t2 + t3 + t4;
  return x;
}
```

Przykład: `zbl(0x070050fa) = 1`

Zadanie 6 (10). Na podstawie poniższego kodu w asemblerze x86-64 uzupełnij w kodzie źródłowym w języku C puste pola. Przy słowie kluczowym «case» brakuje numeru rozpatrywanego przypadku, a w reszcie pól słowa kluczowego break lub specjalnej dyrektywy fallthrough oznaczającej przejście do wykonania następnego przypadku. Tabela skoków rozpoczyna się pod adresem 0x481008 i widnieje po prawej stronie zdeasemblowanego kodu.

```
long lol(long a, unsigned long b) {
  switch (a) {
    case [115]:
      b = (b << a) | (b >> (64 - a));
      [break];
    case [114]:
      b *= b >> (b - 1);
      [break];
    case [112]:
      b = 2022;
      [fallthrough];
    case [110]:
      b |= a ^ (b + 1);
      [fallthrough];
    case [107]:
      b -= 13;
      [break];
    default:
      b += (b >= 0) ? -3 * b : b;
  }
  return b;
}
```

```
401c2d <lol>:
401c2d: lea  -107(%rdi),%rax
401c31: cmp  $8,%rax
401c35: ja   0x401c69
401c37: jmpq *0x481008(,%rax,8)
401c3e: mov  %rsi,%rax
401c41: mov  %edi,%ecx
401c43: rol  %cl,%rax
401c46: retq
401c47: lea  -1(%rsi),%ecx
401c4a: mov  %rsi,%rax
401c4d: shr  %cl,%rax
401c50: imul %rsi,%rax
401c54: retq
401c55: mov  $2022,%esi
401c5a: lea  1(%rsi),%rcx
401c5e: xor  %rcx,%rdi
401c61: or   %rdi,%rsi
401c64: lea  -13(%rsi),%rax
401c68: retq
401c69: neg  %rsi
401c6c: lea  (%rsi,%rsi,1),%rax
401c70: retq
```

Tabela skoków:

0x401c64
0x401c69
0x401c69
0x401c5a
0x401c69
0x401c55
0x401c69
0x401c47
0x401c3e

Zadanie 7 (12). Poniższą jednostkę translacji języka C przetłumaczono do asemblera x86-64 po czym wygenerowano plik relokowalny. Ciąg znaków "bla" umieszczono na początku sekcji «.rodata», a tablicę «tab» na początku sekcji «.data». Uzupełnij podaną tablicę *rekordów relokacji* dla sekcji «.data» – w pierwszej kolumnie jest przesunięcie relokacji względem początku sekcji, a w drugiej *symbol* plus *addend* (stała całkowita). Zakładamy, że typ wszystkich relokacji to «R_X86_64_64», czyli 64-bitowy adres bezwzględny.

```
struct node {
  int (*fn)(const char *);
  long index;
  struct node *next;
  const char *key;
};

extern int foo(const char *);
extern int bar(const char *);
extern char baz[16];
```

```
struct node tab[2] = {
  {
    .key = "bla", .fn = foo,
    .next = tab+1, .index = 0
  }, {
    .fn = bar, .next = &tab[0],
    .index = 1, .key = &baz[3]
  }
};
```

Offset	Wartość
0x00	foo
0x10	tab+32
0x18	.rodata
0x20	bar
0x30	tab
0x38	baz+3

Imię i nazwisko: _____ Numer indeksu: _____

Zadanie 8 (12). Rozważamy procesor *out-of-order* x86-64. Każda instrukcja przechodzi kolejno przez etapy: *dispatch* «D» (wybór jednostki funkcyjnej do przetwarzania instrukcji), *execute* «e» (wyliczenie wyniku), *write-back* «w» (udostępnianie wyliczonej wartości zależnym instrukcjom) i *retire* «R» (wpisywanie wartości do rejestrów widocznych dla programisty). Wpisz w poniższą tabelkę w jaki sposób procesor przeprowadza wymienione instrukcje przez poszczególne etapy przetwarzania. Etapy pierwszej instrukcji oraz etapy *dispatch* pozostałych instrukcji zostały już wpisane. Jeśli w danym cyklu zegarowym instrukcja czekała na wykonanie lub zatwierdzenie, to wpisz w kratkę znak «-». Innymi słowy, należy zasymulować działanie programu `llvm-mca`.

Procesor potrafi w jednym cyklu zlecić (ang. *dispatch*) i zatwierdzić (ang. *retire*) do 4 instrukcji. Ma 3 jednostki funkcyjne wykonujące proste operacje arytmetyczno-logiczne w jednym cyklu oraz jedną jednostkę wykonującą mnożenie w 3 cykle, ale może rozpocząć wykonanie mnożenia co cykl. Zanim procesor rozpocznie wykonywanie instrukcji musi poczekać na jej argumenty. Instrukcja udostępnia swój wynik innym instrukcjom na początku fazy *write-back*, tj. w tym samym cyklu zegarowym instrukcja zależna może rozpocząć swoją pierwszą fazę *execute*. Zatwierdzanie instrukcji następuje w porządku programu.

Instrukcja	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}
`movl %edi, %eax`	D	e	w	R														
`imull %esi, %eax`	D	-	e	e	e	w	R											
`shrl $16, %eax`	D	-	-	-	-	e	w	R										
`imull %edx, %esi`	D	e	e	e	w	-	-	R										
`addl %eax, %esi`	D	e	-	-	-	-	e	w	R									
`imull %ecx, %edi`	D	-	-	e	e	e	w	-	R									
`movzwl %si, %eax`	D	-	-	-	-	-	-	e	w	R								
`addl %edi, %eax`	D	-	-	-	-	-	-	-	e	w	R							
`sarl $16, %eax`	D	-	-	-	-	-	-	-	-	e	w	R						
`imull %ecx, %edx`	D	-	-	-	e	e	e	w	-	-	-	R						
`sarl $16, %esi`	D	-	-	-	-	-	-	e	w	-	-	R						
`addl %esi, %edx`	D	-	D	-	-	-	-	-	e	w	-	R						
`addl %edx, %eax`	D	-	-	D	-	-	-	-	-	-	e	w	R					

Zadanie 9 (10). System posiada dwudrożną sekcyjno-skojarzeniowej pamięć podręczną *indeksowaną i znacznikowaną* adresami fizycznymi i zarządzaną polityką wymiany NRU (ang. *not recently used*). Adresy wirtualne i fizyczne mają 16-bitów i są podzielone następująco (po dwukropku podano liczbę bitów):

VPN:8		VPO:8
CT:4	CI:8	CO:4

Na podstawie powyższych danych uzupełnij kratki w nastepującym zdaniu. Rozmiar pamięci podręcznej wynosi [8192] bajtów, a rozmiar strony pamięci wirtualnej [256] bajtów.

Procesor posługuje się adresami wirtualnymi kiedy robi dostępy do pamięci. Zatem najpierw MMU musi przetłumaczyć adres wirtualny na fizyczny i dopiero potem nastepuje dostęp do pamięci podręcznej na podstawie adresu fizycznego. Interesujący fragment tablicy stron procesu wygląda następująco:

VPN	PPN
0x10	0xF0
0x11	0xA1
0x12	0x42
0x13	0x20

VPN	PPN
0x14	0x41
0x15	0x40
0x16	0x31
0x17	0xA0

Tablica A jest umieszczona w pamięci pod adresem wirtualnym 0x1000 i ma 512 elementów typu «float». Przed wykonaniem programu pamięć podręczna danych jest pusta. Następnie program przechodzi tablicę A sekwencyjnie od pierwszego do ostatniego elementu. Na tak rozgrzanej pamięci podręcznej uruchamiamy poniższą procedurę:

```
float calc(float A[8][64]) {
  float r = 0.0;
  for (int i = 7; i >= 0; i--)
    for (int j = 63; j >= 0; j--)
        r += A[i][j];
  return r;
}
```

Dla powyższej procedury w poniższej tabelce podaj liczbę chybień w pamięć podręczną. Zakładamy, że wyłącznie dostępy do tablicy A mogą generować chybienia.

Dostępy do tablicy	Chybienia
A[7][0..63]	0
A[6][0..63]	0
A[5][0..63]	0
A[4][0..63]	0
A[3][0..63]	16
A[2][0..63]	0
A[1][0..63]	16
A[0][0..63]	16