ĐÁP ÁN BÀI TẬP ÔN TẬP
LẬP TRÌNH HỆ THỐNG

Bài tập 1. Thực hiện các phép chuyển đổi và tính toán sau:

a. Chuyển các số hexan sang hệ nhị phân:

0x39A7F8$_{16}$ = **0011 1001 1010 0111 1111 1000**$_{2}$

0xD5E4C$_{16}$ = **1101 0101 1110 0100 1100**$_{2}$

b. Chuyển số nhị phân sang hệ hexan (16):

1100100101110112 = **0x64BB**$_{16}$

1001101110011101101012 = **0x1373B5**$_{16}$

c. Thực hiện tính toán:

0x506 + 0x12 = **0x518**

0x503C – 0x42 = **0x4FFA**

0x6653 + 98 = **0x66B5**

Bài tập 2. Cho đoạn chương trình:

```
/* Biến val gồm 4 byte đánh thứ tự từ 1 đến 4 */
int val = 0x87654321;
/* pointer trỏ đến ô nhớ lưu trữ biến val */
byte_pointer valp = (byte_pointer) &val;
/* A. hàm trả về byte thứ 1 kể từ địa chỉ ô nhớ */
show_bytes(valp, 1);
/* B. hàm trả về byte thứ 2 kể từ địa chỉ ô nhớ */
show_bytes(valp, 2);
```

Kết quả trả về của 2 hàm **show_bytes()** sẽ khác nhau như thế nào trong trường hợp chạy trên hệ thống sử dụng little-endian và big-endian?

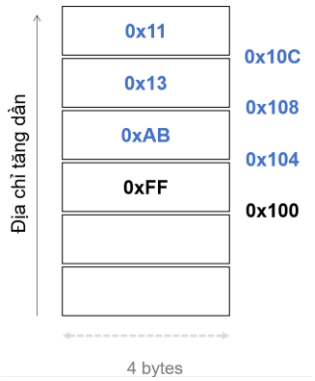
Hệ thống	show_bytes(valp,1)	show_bytes(valp,2)
Little-endian	**0x21**	**0x43**
Big-endian	**0x87**	**0x65**

Bài tập 3. Giả sử có các giá trị sau đang được lưu trong các ô nhớ và các thanh ghi:

Địa chỉ	Giá trị
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Thanh ghi	Giá trị
%eax	0x100
%ecx	0x1
%edx	0x3

a. Hãy điền vào hình minh hoạ bên dưới các địa chỉ và giá trị tương ứng của các ô nhớ.



b. Giả sử ta có câu lệnh **movl [toán hạng x], %ebx** để lấy giá trị dựa trên toán hạng x và đưa vào thanh ghi %ebx. Dựa vào các giá trị trong ô nhớ và thanh ghi ở trên, điền các giá trị sẽ lấy được nếu sử dụng các toán hạng sau:

Toán hạng x	Giá trị lấy được
%eax	Giá trị lưu trong thanh ghi eax: **0x100**
0x104	Giá trị trong ô nhớ ở địa chỉ 0x104: **0xAB**
$0x108	Giá trị hằng số **0x108**
(%eax)	Giá trị trong ô nhớ có địa chỉ 0x100: **0xFF**
4(%eax)	Giá trị trong ô nhớ có địa chỉ 0x104: **0xAB**
9(%eax, %edx)	Giá trị trong ô nhớ có địa chỉ 0x10C: **0x11**
0xFC(,%ecx,4)	Giá trị trong ô nhớ có địa chỉ 0x100: **0xFF**
(%eax, %edx, 4)	Giá trị trong ô nhớ có địa chỉ 0x10C: **0x11**

c. Điền vào chỗ trống ảnh hưởng của những câu lệnh dưới đây, bao gồm thanh ghi/ô nhớ nào bị thay đổi giá trị và giá trị đó là bao nhiêu?
Lưu ý: Giá trị các thanh ghi/ô nhớ ở mỗi câu lệnh vẫn lấy từ bảng trên.

Câu lệnh	Thanh ghi/ô nhớ bị thay đổi	Giá trị
addl %ecx, (%eax)	Ô nhớ có địa chỉ **0x100**	0xFF + 0x1 = **0x100**
imull $16, (%eax, %edx, 4)	Ô nhớ có địa chỉ **0x10C**	0x11 * 16 **= 0x110**
subl %edx, %eax	Thanh ghi %eax	0x100 – 0x3 **= 0xFD**
movl (%eax, %edx, 4), %eax	Thanh ghi %eax	Giá trị ô nhớ 0x10C: **0x11**
leal (%eax, %edx, 4), %eax	Thanh ghi %eax	**0x10C**

Bài tập 4. Giả sử 1 lập trình viên muốn viết 1 đoạn mã thực hiện chức năng sau:

```
1  int req_fun(int a, int b){
2        int val = 0;
3        int x = a & 0xFFFF0000;
4        int y = b & 0xFFFF;
5        val = x | y;
6        return val;
7  }
```

a. Hoàn thành đoạn mã assembly bên dưới để có chức năng tương tự, kết quả val lưu trong %eax.

```
1  movl   8(%ebp), %ebx
2  movl   12(%ebp), %ecx
3  andl   $0xFFFF0000, %ebx
4  andl   $0xFFFF , %ecx
5  orl    %ecx, %ebx
6  movl   %ebx, %eax
```

b. Một lập trình viên khác có ý tưởng khác với đoạn code tối ưu hơn, hãy hoàn thành đoạn code assembly bên dưới?

```
1  movl   8(%ebp), %ebx
2  movl   12(%ebp), %ecx
3  movl   %ebx, %eax
4  movw   %cx, %ax
```

Bài tập 5. Cho đoạn mã assembly như bên dưới:

x lưu tại ô nhớ (%ebp + 8), y lưu tại ô nhớ (%ebp + 12), z lưu tại ô nhớ (%ebp + 16), giá trị trả về lưu trong thanh ghi %eax

```
1  movl   8(%ebp), %ecx
2  movl   12(%ebp), %eax
3  imull  %ecx, %eax
4  subl   %ecx, %eax
5  leal   (%eax,%eax,4), %eax
6  addl   16(%ebp), %eax
7  sarl   $2, %eax
```

Dựa vào mã assembly, điền vào những phần còn trống trong các hàm C tương ứng:

a. Hàm **arith**() phiên bản 1

```
1  int arith(int x, int y, int z)
2  {
3        int t1 = x * y;
4        int t2 = t1 - x;
5        int t3 = 5*t2;
6        int t4 = t3 + z;
7        int t5 = t4 >> 2;
8        return t5;
9  }
```

b. Hàm **arith()** phiên bản 2 (rút gọn)

```
1  int arith(int x, int y, int z)
2  {
3        int t1 = [(x* y – x)*5 + z] >> 2;
4        return t1;
5  }
```

Bài tập 6. Cho đoạn mã assembly dưới đây được tạo bởi GCC:

x lưu tại ô nhớ (%ebp+8), y lưu tại ô nhớ (%ebp+12)

```
1        movl   8(%ebp), %eax
2        movl   12(%ebp), %edx
3        cmpl   $-3, %eax
4        jge    .L2
5        cmpl   %edx, %eax
6        jle    .L3
7        imull  %edx, %eax
8        jmp    .L4
9    .L3:
10       leal   (%edx,%eax), %eax
11       jmp    .L4
12   .L2:
13       cmpl   $2, %eax
14       jg     .L5
15       xorl   %edx, %eax
16       jmp    .L4
17   .L5:
18       subl   %edx, %eax
19   .L4:                              // thoát if/else
20       // return val
```

a. Dưới đây là đoạn mã C tương ứng với đoạn mã assembly trên, trong đó giá trị cuối cùng của **val** được lưu trong **%eax** để trả về tại **.L4**. Hãy điền các vị trí còn trống?

(Lưu ý: bài tập này có nhiều đáp án có thể thoả mãn đoạn code C bên dưới)

```
1   int test(int x, int y) {
2               int val = x * y;
3               if ( x >= -3 ) {
4                       if ( x > 2 )
5                               val = x - y;
6                       else
7                               val = x ^ y;
8               } else if ( x <= y )
9                       val = x + y;
10      return val;
11  }
```

Sinh viên chỉ cần tìm được các nhánh logic đúng với các trường hợp so sánh x, y khớp với đáp là được. Các trường hợp đúng:

- TH1: x < -3 và x > y: val = x*y
- TH2: x < -3 và x <=y: val = x + y
- TH3: x >=-3 và x <=2: val = x ^ y
- TH4: x >= -3 và x > 2: val = x - y

b. Giả sử với tham số **x = 4**, **y = 2**. Khi đó **val =** 2

c. Giả sử với tham số **x = 1, y = 9**. Khi đó **val =** 8

Bài tập 7. Cho đoạn mã assembly như bên dưới: *x lưu tại ô nhớ (%ebp+8)*

```
1        movl   8(%ebp), %ebx
2        movl   $0, %eax
3        movl   $0, %ecx
4        jmp    .L2
5    .L1:
6        leal (%eax,%eax), %edx
7        movl   %ebx, %eax
8        andl   $1, %eax
9        orl    %edx, %eax
10       shrl   %ebx
11       addl   $1, %ecx
12   .L2:
13       cmpl   $5, %ecx
14       jle .L1
```

a. Dưới đây là đoạn mã C tương ứng với đoạn mã assembly. Biết giá trị cuối cùng của **val** được lưu trong **%eax** để trả về sau khi thoát vòng lặp **for**. Hãy điền vào vị trí còn trống để hoàn thiện đoạn mã bên dưới?

```
1    int fun_b(unsigned x) {
2            int val = 0;
3            int i;
4            for (i = 0; i <= 5; i ++) {
5                    val = (x & 1) | (2*val);
6                    x >>= 1;
7            }
8            return val;
9     }
```

b. Với **x = 32**, xác định giá trị của **val**?

val = 1

Bài tập 8. Cho hàm C như sau:

```
1   int my_function()
2   {
3       int first_var = 0;
4       int second_var = 0xdeadbeef;
5       char str[2] = ?;
6
7       char buf[10];
8       gets(buf);
9       return len(buf);
10  }
```

GCC tạo ra mã assembly tương ứng như sau:

```
.section .data
.LC0:
    .byte 0x68,0x69,0x74,0x68,0x75,0x0
.section .text
1  my_function:
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $24, %esp
5      movl    $0, -4(%ebp)
6      movl    $0xdeadbeef, -8(%ebp)
7      movw    (.LC0), %dx
8      movw    %dx, -12(%ebp)
9      leal    -24(%ebp), %eax
10     pushl   %eax
11     call    gets
12     leal    -24(%ebp), %eax
13     pushl   %eax
14     call    len
15     leave
16     ret
```

Giả sử hàm **my_function** bắt đầu thực thi với những giá trị thanh ghi như sau:

Thanh ghi	Giá trị
%esp	0x800168
%ebp	0x800180

Biết **.LC0** là label của 1 vùng nhớ.

a. Giá trị của thanh ghi **%ebp** sau khi thực thi dòng lệnh assembly thứ 3? Giải thích.

0x800164

b. Giá trị của thanh ghi **%esp** sau khi thực thi dòng lệnh assembly thứ 4? Giải thích.

0x80014C

c. Hàm **my_function** có 1 biến cục bộ **str,** là 1 mảng char gồm 2 ký tự. Quan sát mã assembly, hãy cho biết 2 ký tự được gán cho mảng **str** là gì?

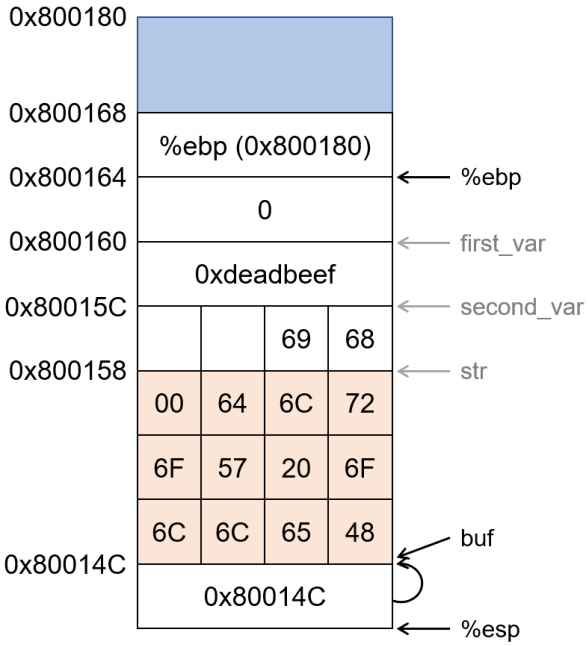
str[0] = 'h', str[1] = 'i'.

d. Xác định địa chỉ cụ thể của vị trí sẽ lưu chuỗi input nhận về từ hàm **gets()**? Giải thích?

Vị trí lưu chuỗi input là địa chỉ **0x80014C**.

e. Giả sử khi gọi **gets** ở dòng code assembly thứ 13, nhận được 1 chuỗi **"Hello world"**. Vẽ stack frame của **my_function** ngay sau khi hàm **gets** trả về.

Lưu ý: Cần chú thích *địa chỉ, giá trị của các ô nhớ* trong stack frame của **my_function**, bao gồm cả các ô nhớ chứa biến cục bộ, tham số và chuỗi đã nhập với **gets**.



f. **gets** không giới hạn độ dài chuỗi mà nó nhận. Dựa vào stack frame của **my_function** đã vẽ, hãy tìm độ dài tối đa của **buf** (số ký tự) sao cho khi nhập vẫn chưa ghi đè lên bất kỳ ô nhớ quan trọng nào trong stack của **my_function**?

Có thể nhập tối đa chuỗi dài 23 ký tự cho buf mà chưa ghi đè lên các ô nhớ lưu giá trị %ebp cũ của hàm mẹ.

g. Chương trình có thể có lỗ hổng buffer overflow. Thử tìm một chuỗi buf sao cho có thể ghi đè lên biến cục bộ **second_var** một giá trị mới là **0xABDCEF**.

Nhập 1 chuỗi buf dài 20 bytes, trong đó 16 byte đầu tuỳ ý (khác 0x0A), 4 byte cuối là 4 byte hexan EF DC AB 00.

"a"*16 + "\xEF\xDC\xAB\00".

Bài tập 9. Trong hệ thống 32 bit, cho mảng **T A[N]** với **T** và **N** chưa biết. Biết tổng kích thước của mảng A là **28 bytes**, T là 1 kiểu dữ liệu cơ bản.

a. Xác định **T** và **N**? Giải thích? Liệt kê tất cả các trường hợp thỏa mãn và dạng khai báo của mảng A với mỗi **T** và **N** tìm được.

- sizeof(T) = 1 → N = 28, ta có char A[28]
- sizeof(T) = 2 → N = 14, ta có short A[14]
- sizeof(T) = 4 → N = 7, ta có int A[7] hoặc float A[7]...

b. Giả sử với 1 trường hợp của **T A[N]** ở trên, ta có đoạn mã C và assembly tương ứng truy xuất các phần tử của mảng như bên dưới. Hãy điền vào những chỗ còn trống ở 2 đoạn code?

Code C

```
1  short A[7];   // khai báo mảng A
2  A[0] = 0;
3  for (int i = 1; i < 7; i ++)
4  {
5      A[i] = A[i-1] + i;
6  }
```

Code assembly

```
1       movl    $A, %eax    // address of A
2       movw    $0, (%eax)
3       movl    $1, %ecx    // chỉ số i
4   .L1:
5       xorl     %ebx, %ebx
6       movw    -2(%eax, %ecx, 2), %bx
7       addl    %ecx, %ebx
8       movw    %bx, (%eax, %ecx, 2)
9       incl    %ecx
10      cmpl    $7, %ecx
11      jl      .L1
```

c. Cho biết sau khi thực hiện đoạn code trên, ta thu được mảng A với các giá trị phần tử như thế nào? Mảng short A[7] gồm 7 phần tử có giá trị lần lượt là 0, 1, 3, 6, 10, 15, 21.

Bài tập 10. Cho các định nghĩa sau trong code C, với giá trị N chưa biết.

```
1   # define N ?
2   void matrix_set_val(int A[N][N], int val)
3   {
4       int i;
5       for (i = 0 ; i < N; i++)
6               A[i][i] = val;
7   }
```

Và đoạn code assembly tương ứng được tạo bởi GCC: *Địa chỉ mảng A lưu tại ô nhớ (%ebp+8), giá trị val lưu tại ô nhớ (%ebp+12)*

```
1       movl   8(%ebp), %ecx
2       movl   12(%ebp), %edx
3       movl   $0, %eax
4   .L14:
5       movl   %edx, (%ecx,%eax)
6       addl   $68, %eax
7       cmpl   $1088, %eax
8       jne    .L14
```

Hãy phân tích đoạn mã assembly trên và xác định giá trị của N? **N = 16**.

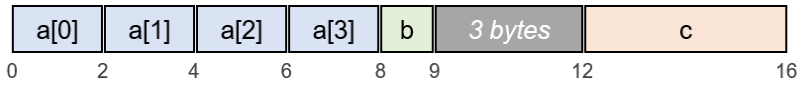
Bài tập 11. Cho struct có định nghĩa như bên dưới trong Linux 32-bit, có yêu cầu alignment.

```
1  typedef struct {
2      short a[4];
3      char b;
4      int c;
5  } str1;
```

Một hàm func được dùng để gán giá trị cho thành phần a[i] và c của struct, kết quả trả về là giá trị của thành phần c như bên dưới.

```
1  int func(int i, int val)
2  {
3      str1 s;
4      s.c = 1;
5      s.a[i] = val;
6      return s.c;
7  }
```

a. Vẽ hình minh họa việc cấp phát struct trên trong bộ nhớ?



b. Tổng kích thước của struct trên là bao nhiêu? 16 bytes.

c. Tìm giá trị trả về của hàm **func** với các tham số sau? Giải thích các thay đổi có trong vùng nhớ của struct?

Giả định chương trình được biên dịch với compiler chỉ warning khi có truy xuất ngoài mảng, vẫn cho chương trình chạy bình thường.

- **func**(2, 2) = 1, s.c được gán bằng 1, s.a[2] được gán = 2
- **func**(4, 2) = 1, s.c được gán bằng 1, s.a[4] (ngoài mảng) được gán = 2
- **func**(6, 2) = 2, s.c được gán bằng 1, s.a[6] (ngoài mảng) được gán = 2, ghi đè s.c nên s.c = 2

Bài tập 12. Cho 2 định nghĩa struct với 2 giá trị A và B chưa biết.

```
1   typedef struct {
2       short x[A][B];          /* Hằng số A và B chưa biết */
3       int y;
4   } str1;
5
6   typedef struct {
7       char array[B];          /* Hằng số B chưa biết */
8       int t;
9       short s[B];
10      int u;
11  } str2;
```

Cho đoạn code C cùng đoạn mã assembly tương ứng như bên dưới:

```
1  void setVal(str1 *r1, str2 *r2)
2  {
3      int v1 = r2->t;
4      int v2 = r2->u;
5      r1->y = v1+v2;
6  }
```

```
1  setVal:
2      movl   12(%ebp), %eax
3      movl   36(%eax), %edx
4      addl   12(%eax), %edx
5      movl   8(%ebp), %eax
6      movl   %edx, 92(%eax)
```

Dựa vào tương quan giữa 2 đoạn mã C và assembly, xác định 2 giá trị **A** và **B**, biết hệ thống 32 bit và có yêu cầu alignment.

A = 5, B = 9.

Bài tập 13. Cho 2 file **main.c** và **fib.c** như sau.

```
/* main.c */

1.  void fib (int n);
2.  int main (int argc, char** argv
      ) {
3.      int n = 0;
4.      sscanf(argv[1], "%d", &n);
5.      fib(n);
6.  }
```

```
/* fib.c */

1.  #define N 16
2.  static unsigned int ring[3][N];
3.  void print_bignat(unsigned int*
     a){
4.      int i;
5.      ...
6.  }
7.  void fib (int n) {
8.      int i;
9.      static int carry;
10.     ...
11. }
```

Hoàn thành bảng sau về các symbol có trong symbol table có trong 2 mô-đun main.o và fib.o, xác định các symbol là ***local/global*** hay ***external***, ***strong*** hay ***weak***.

- Ghi '-' ở cả 2 cột nếu tên không có trong symbol table của mô-đun tương ứng.
- Ghi N/A ở cột ***Strong hay weak*** nếu loại symbol là local.

Symbol table của main.o

Tên symbol	Loại symbol	Strong hay weak
main	global	strong
fib	external	strong
n	-	-

Symbol table của fib.o

Tên symbol	Loại symbol	Strong hay weak
ring	local	N/A
print_bignat	global	strong
fib	global	strong
canary	local	N/A

HƯỚNG DẪN GIẢI CHI TIẾT BÀI TẬP ÔN TẬP
LẬP TRÌNH HỆ THỐNG

Bài tập 1. Thực hiện các phép chuyển đổi và tính toán sau:

a. Chuyển các số hexan sang hệ nhị phân:

0x39A7F8$_{16}$ = **0011 1001 1010 0111 1111 1000**$_2$

0xD5E4C$_{16}$ = **1101 0101 1110 0100 1100**$_2$

b. Chuyển số nhị phân sang hệ hexan (16):

1100100101110112 = **0110 0100 1011 1011$_2$ = 0x64BB$_{16}$**

1001101110011101101012 = **1 0011 0111 0011 1011 0101$_2$ = 0x1373B5**$_{16}$

c. Thực hiện tính toán:

0x506 + 0x12 = **0x518**

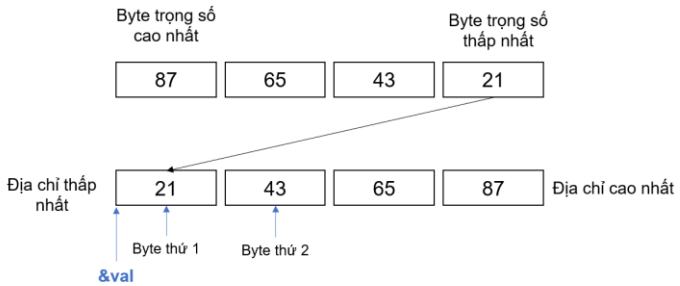
0x503C – 0x42 = **0x4FFA**

0x6653 + 98 = **0x6653 + 0x62 = 0x66B5**

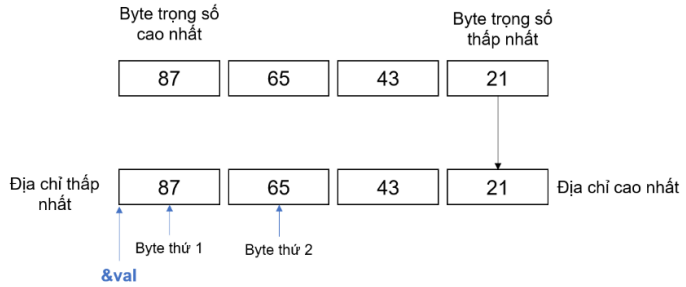
Bài tập 2. Cho đoạn chương trình:

```
/* Biến val gồm 4 byte đánh thứ tự từ 1 đến 4 */
int val = 0x87654321;
/* pointer trỏ đến ô nhớ lưu trữ biến val */
byte_pointer valp = (byte_pointer) &val;
/* A. hàm trả về byte thứ 1 kể từ địa chỉ ô nhớ */
show_bytes(valp, 1);
/* B. hàm trả về byte thứ 2 kể từ địa chỉ ô nhớ */
show_bytes(valp, 2);
```

Kết quả trả về của 2 hàm **show_bytes()** sẽ khác nhau như thế nào trong trường hợp chạy trên hệ thống sử dụng little-endian và big-endian?



Little endian: byte **thấp nhất** ở địa chỉ **thấp nhất**



Big endian: byte **thấp nhất** ở địa chỉ **cao nhất**

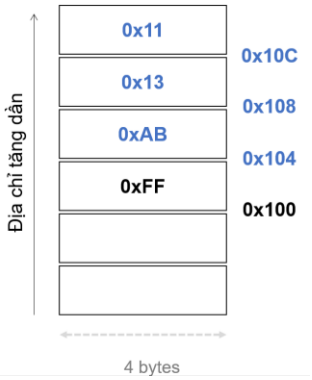
Hệ thống	show_bytes(valp,1)	show_bytes(valp,2)
Little-endian	**0x21**	**0x43**
Big-endian	**0x87**	**0x65**

Bài tập 3. Giả sử có các giá trị sau đang được lưu trong các ô nhớ và các thanh ghi:

Địa chỉ	Giá trị
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Thanh ghi	Giá trị
%eax	0x100
%ecx	0x1
%edx	0x3

d. Hãy điền vào hình minh hoạ bên dưới các địa chỉ và giá trị tương ứng của các ô nhớ.



e. Giả sử ta có câu lệnh **movl [toán hạng x], %ebx** để lấy giá trị dựa trên toán hạng x và đưa vào thanh ghi %ebx. Dựa vào các giá trị trong ô nhớ và thanh ghi ở trên, điền các giá trị sẽ lấy được nếu sử dụng các toán hạng sau:

Toán hạng x	Giá trị lấy được
%eax	Giá trị lưu trong thanh ghi eax: **0x100**
0x104	Giá trị trong ô nhớ ở địa chỉ 0x104: **0xAB**
$0x108	Giá trị hằng số **0x108**
(%eax)	Giá trị trong ô nhớ có địa chỉ lưu trong %eax = 0x100: **0xFF**
4(%eax)	Giá trị trong ô nhớ có địa chỉ %eax + 4 = 0x100 + 4 = 0x104: **0xAB**
9(%eax, %edx)	Giá trị trong ô nhớ có địa chỉ %eax + %edx + 9 = 0x100 + 0x3 + 9 = 0x10C: **0x11**
0xFC(,%ecx,4)	Giá trị trong ô nhớ có địa chỉ 4*%ecx + 0xFC = 4 + 0xFC = 0x100: **0xFF**
(%eax, %edx, 4)	Giá trị trong ô nhớ có địa chỉ %eax + 4*%edx = 0x100 + 4*0x3 = 0x10C: **0x11**

f. Điền vào chỗ trống ảnh hưởng của những câu lệnh dưới đây, bao gồm thanh ghi/ô nhớ nào bị thay đổi giá trị và giá trị đó là bao nhiêu?
Lưu ý: Giá trị các thanh ghi/ô nhớ ở mỗi câu lệnh vẫn lấy từ bảng trên.

Câu lệnh	Thanh ghi/ô nhớ bị thay đổi	Giá trị
addl %ecx, (%eax)	Ô nhớ có địa chỉ **0x100**	0xFF + 0x1 = **0x100**
imull $16, (%eax, %edx, 4)	Ô nhớ có địa chỉ **0x10C**	0x11 * 16 **= 0x110**
subl %edx, %eax	Thanh ghi %eax	0x100 – 0x3 **= 0xFD**
movl (%eax, %edx, 4), %eax	Thanh ghi %eax	Giá trị ô nhớ 0x10C: **0x11**
leal (%eax, %edx, 4), %eax	Thanh ghi %eax	**0x10C**

Bài tập 4. Giả sử 1 lập trình viên muốn viết 1 đoạn mã thực hiện chức năng sau:

```
1  int req_fun(int a, int b){
2          int val = 0;
3          int x = a & 0xFFFF0000;
4          int y = b & 0xFFFF;
5          val = x | y;
6          return val;
7  }
```

a. Hoàn thành đoạn mã assembly bên dưới để có chức năng tương tự, kết quả val lưu trong %eax.

```
1  movl   8(%ebp), %ebx
2  movl   12(%ebp), %ecx
3  andl   $0xFFFF0000, %ebx
4  andl   $0xFFFF , %ecx
5  orl    %ecx, %ebx
6  movl   %ebx, %eax
```

Lưu ý: Lệnh 5 có thể thay đổi thứ tự 2 thanh ghi, khi đó cần điều chỉnh thanh ghi source ở lệnh 6 tương ứng.

b. Một lập trình viên khác có ý tưởng khác với đoạn code tối ưu hơn, hãy hoàn thành đoạn code assembly bên dưới?

```
1  movl   8(%ebp), %ebx
2  movl   12(%ebp), %ecx
3  movl   %ebx, %eax
4  movw   %cx, %ax
```

Ý tưởng: đoạn code C thực hiện lấy 2 byte cao của a và 2 byte thấp của b ghép lại tạo thành 2 byte cao và thấp của val.

Đoạn code của phần b ban đầu gán toàn bộ 4 byte của số a vào thanh ghi %eax (val), sau đó đè 2 byte thấp của b (%cx trong %ecx) vào %ax. Như vậy 2 byte cao của %eax vẫn giữ nguyên là 2 byte cao của a, còn 2 byte thấp của %eax được gán bằng 2 byte thấp của b.

Bài tập 5. Cho đoạn mã assembly như bên dưới:

x lưu tại ô nhớ (%ebp + 8), y lưu tại ô nhớ (%ebp + 12), z lưu tại ô nhớ (%ebp + 16), giá trị trả về lưu trong thanh ghi %eax

```
1  movl   8(%ebp), %ecx          // %ecx = x
2  movl   12(%ebp), %eax         // %eax = y
3  imull  %ecx, %eax             // %eax = %eax * %ecx = x * y (t1)
4  subl   %ecx, %eax             // %eax = %eax - %ecx = x * y – x = t1 - x (t2)
5  leal   (%eax,%eax,4), %eax    // %eax = %eax* 4 + %eax = 5*%eax = 5*t2 (t3)
6  addl   16(%ebp), %eax         // %eax = %eax + z = t3 + z (t4)
7  sarl   $2, %eax               // %eax = %eax >> 2 = t4 >> 2 (t5)
```

Dựa vào mã assembly, điền vào những phần còn trống trong các hàm C tương ứng:

a. Hàm **arith**() phiên bản 1

```
1  int arith(int x, int y, int z)
2  {
3          int t1 = x * y;
4          int t2 = t1 - x;
5          int t3 = 5*t2;
6          int t4 = t3 + z;
7          int t5 = t4 >> 2;
8          return t5;
9  }
```

b. Hàm **arith()** phiên bản 2 (rút gọn)

```
1  int arith(int x, int y, int z)
2  {
3          int t1 = [(x* y – x)*5 + z] >> 2;
4          return t1;
5  }
```

Bài tập 6. Cho đoạn mã assembly dưới đây được tạo bởi GCC:

x lưu tại ô nhớ (%ebp+8), y lưu tại ô nhớ (%ebp+12)

```
1        movl   8(%ebp), %eax        // %eax = x
2        movl   12(%ebp), %edx       // %edx = y
3        cmpl   $-3, %eax            + So sánh %eax (x) với -3: x ? -3   (if 1)
4        jge    .L2                  - nhảy đến .L2 nếu x >= -3
5        cmpl   %edx, %eax           - Vế 1 của if 1: So sánh %eax (x) với %edx (y): x ? y (if 2)
6        jle    .L3                  // nhảy đến .L3 nếu x <= y
7        imull  %edx, %eax           // Vế 1 của if 2: %eax = %eax*%edx = x*y
8        jmp    .L4                  // nhảy k điều kiện đến .L4 -> thoát if/else
9    .L3:                            // Vế 2 của if 2: nhảy đến từ lệnh số 6
10       leal   (%edx,%eax), %eax    // %eax = %eax + %edx = x + y
11       jmp    .L4                  // nhảy k điều kiện đến .L4 -> thoát if/else
12   .L2:                            - Vế 2 của if1: nhảy đến từ lệnh số 4
13       cmpl   $2, %eax             // so sánh %eax (x) với 2: x ? 2  (if 3)
14       jg     .L5                  // nhảy đến .L5 nếu x > 2
15       xorl   %edx, %eax           // Vế 1 của if 3: %eax = %eax ^ %edx = x ^ y
16       jmp    .L4                  // nhảy k điều kiện đến .L4 -> thoát if/else
17   .L5:                            // Vế 2 của if 3: nhảy đến từ lệnh 14
18       subl   %edx, %eax           // %eax = %eax - %edx = x - y
19   .L4:                            // thoát if/else
20       // return val
```

Từ các đoạn mã phân biệt bằng màu sắc (cùng màu là cùng khối if/else), ta thấy có 1 số if/else lồng nhau như sau: if 1 có 1 vế chứa if 2 và 1 vế chứa if 3. Cụ thể, vế 1 màu đỏ của if 1 chứa if 2 màu xanh lá, vế 2 màu đỏ của if 1 chứa if 3 màu xanh dương.

Ở đoạn code bên dưới, có thể viết bất kỳ vế nào của if 1 ở dòng lệnh 4 – 7. Vế còn lại sẽ từ dòng 8 – 9. Tuy nhiên, if 2 và if 3 đều có 2 vế true/false, chỉ có dòng lệnh 4 – 7 đủ chỗ để điền 2 vế. Vế được viết ở dòng 8 - 9 của if 1 chỉ có thể ghi 1 vế, như vậy ta sẽ tận dụng dòng code 2 để ghi giá trị của vế còn thiếu như giá trị mặc định. Nếu tất cả các điều kiện if/else bên dưới đều sai, ta sẽ thu được val là giá trị của vế còn thiếu.

a. Dưới đây là đoạn mã C tương ứng với đoạn mã assembly trên, trong đó giá trị cuối cùng của **val** được lưu trong **%eax** để trả về tại **.L4**. Hãy điền các vị trí còn trống?

(Lưu ý: bài tập này có nhiều đáp án có thể thoả mãn đoạn code C bên dưới)

Ví dụ: Chọn ghi vế if 3 trước, ghi vế if 2 sau. Có 1 vế của if 2 bị thiếu sẽ được điền ở dòng lệnh 2.

```
1   int test(int x, int y) {
2           int val = x * y;            // vế còn thiếu của if 2→ điền cuối cùng
3           if ( x >= -3 ) {            // cmp dòng 3 và jge dòng 4
4                   if ( x > 2 )        // .L2: cmp dòng 13 và jg dòng 14
```

```
5                       val = x - y;      // .L5
6               else
7                       val = x ^ y;      // dòng 15 khi điều kiện dòng 14 không đúng
8           } else if ( x <= y )          // Dòng 5 khi điều kiện dòng 4 không đúng, jle dòng 6
9               val = x + y;              // .L3
10      return val;                       // .L4
11  }
```

Một trường hợp khác, ghi vế if 2 trước, ghi vế if 3 sau. Có 1 vế của if 3 bị thiếu ghi lên dòng lệnh 2.

```
1       int test(int x, int y) {
2           int val = x - y;              // vế còn thiếu của if 3 → điền cuối cùng
3           if ( x < -3 ) {           // cmp dòng 3 và jge dòng 4
4               if ( x > y )              // .L2: cmp dòng 13 và jg dòng 14
5                       val = x * y;      // .L5
6               else
7                       val = x + y;      // dòng 15 khi điều kiện dòng 14 không đúng
8           } else if ( x <= 2 )          // Dòng 5 khi điều kiện dòng 4 không đúng, jle dòng 6
9               val = x ^ y;              // .L3
10      return val;                       // .L4
11  }
```

Sinh viên chỉ cần tìm được các nhánh logic đúng với các trường hợp so sánh x, y khớp với đáp là được. Các trường hợp đúng:

- TH1: x < -3 và x > y: val = x*y
- TH2: x < -3 và x <=y: val = x + y
- TH3: x >=-3 và x <=2: val = x ^ y
- TH4: x >= -3 và x > 2: val = x - y

b. Giả sử với tham số **x = 4**, **y = 2**. Khi đó **val =** .....

x = 4, y = 2, thuộc TH 4: do vậy val = x – y = 4 – 2 = 2

c. Giả sử với tham số **x = 1, y = 9**. Khi đó **val =** .....

x = 1, y = 9, thuộc TH3: do vậy val = x ^ y = 1 ^ 9 = 0001_2 ^ 1001_2 = 1000_2 = 8

Bài tập 7. Cho đoạn mã assembly như bên dưới: *x lưu tại ô nhớ (%ebp+8)*

```
1       movl   8(%ebp), %ebx          // %ebx = x
2       movl   $0, %eax               // %eax = 0 = val
3       movl   $0, %ecx               // %ecx = 0
4       jmp    .L2
5   .L1:
6       leal (%eax,%eax), %edx        // %edx = 2*%eax = 2*val → sử dụng %eax lần đầu ở mỗi vòng
                                      lặp → là val của vòng lặp trước
7       movl   %ebx, %eax             // %eax = %ebx = x
8       andl   $1, %eax               // %eax = %eax & 1 = x & 1
9       orl    %edx, %eax             // %eax = %eax | %edx = (x & 1) | 2*val → lần thay đổi %eax
                                      cuối cùng của mỗi vòng lặp → val sau mỗi vòng lặp
10      shrl   %ebx                   // %ebx >> 1 = x >> 1
11      addl   $1, %ecx               // %ecx ++  → i được cập nhật
12  .L2:
13      cmpl   $5, %ecx               // So sánh %ecx ? 5
14      jle .L1                       // %ecx <= 5 thì quay về .L1 → còn thực hiện vòng lặp
```

- Dòng lệnh 13 so sánh %ecx với 5, nếu chưa lớn hơn 5 thì nhảy về nhãn .L1, thực thi đoạn code sau đó lại kiểm tra điều kiện → đoạn code từ .L1 liên quan đến vòng lặp (dòng 5 – 14).

- %eax chứa val, được khởi tạo là 0, %ecx có thể chứa giá trị biến còn lại là i, được khởi tạo là 0.

- Đoạn code trên thực hiện khởi tạo giá trị, sau đó nhảy đến .L2 kiểm tra điều kiện rồi mới thực thi đoạn code liên quan đến vòng lặp → nên chuyển về for hoặc while.

- Các thông tin cần xác định:

  + Khởi tạo: dòng 2, 3, gán val = 0, i = 0;

  + Điều kiện kiểm tra: dòng 13 – 14: kiểm tra nếu i <= 5 thì còn lặp (quay về .L1), không thì thoát.

  + Cập nhật: cập nhật biến dùng trong điều kiện kiểm tra là i → dòng 11, tăng i (%ecx) lên 1 đơn vị.

  + Phần thân: các lệnh còn lại (6 – 10).

a. Dưới đây là đoạn mã C tương ứng với đoạn mã assembly. Biết giá trị cuối cùng của **val** được lưu trong **%eax** để trả về sau khi thoát vòng lặp **for**. Hãy điền vào vị trí còn trống để hoàn thiện đoạn mã bên dưới?

```
1   int fun_b(unsigned x) {
2         int val = 0;
3         int i;
4         for (i = 0; i <= 5; i ++) {
5               val = (x & 1) | (2*val);
6               x = x >> 1;
7         }
8         return val;
9    }
```

b. Với **x = 32**, xác định giá trị của **val**?

i = 0: val = (32 & 1) | (2*0) = 0 | 0 = 0. x = 16
i = 1: val = (16 & 1) | (2*0) = 0 | 0 = 0. x = 8
i = 2: val = (8 & 1) | (2*0) = 0 | 0 = 0. x = 4
i = 3: val = (4 & 1) | (2*0) = 0 | 0 = 0. x = 2
i = 4: val = (2 & 1) | (2*0) = 0 | 0 = 0. x = 1
i = 5: val = (1 & 1) | (2*0) = 1 | 0 = 1. x = 0
i = 6: dừng
Vậy val = 1

Bài tập 8. Cho hàm C như sau:

```
1   int my_function()
2   {
3       int first_var = 0;
4       int second_var = 0xdeadbeef;
5       char str[2] = ?;
6
7       char buf[10];
8       gets(buf);
9       return len(buf);
10  }
```

GCC tạo ra mã assembly tương ứng như sau:

```
.section .data
.LC0:
    .byte 0x68,0x69,0x74,0x68,0x75,0x0
.section .text
1   my_function:
2       pushl   %ebp
3       movl    %esp, %ebp
4       subl    $24, %esp
5       movl    $0, -4(%ebp)
6       movl    $0xdeadbeef, -8(%ebp)
7       movw    (.LC0), %dx
8       movw    %dx, -12(%ebp)
9       leal    -24(%ebp), %eax
10      pushl   %eax
11      call    gets
12      leal    -24(%ebp), %eax
13      pushl   %eax
14      call    len
15      leave
16      ret
```

Giả sử hàm **my_function** bắt đầu thực thi với những giá trị thanh ghi như sau:

Thanh ghi	Giá trị
%esp	0x800168
%ebp	0x800180

Biết **.LC0** là label của 1 vùng nhớ.

a. Giá trị của thanh ghi **%ebp** sau khi thực thi dòng lệnh assembly thứ 3? Giải thích.

- Sau dòng lệnh số 2, %esp bị giảm xuống 4: %esp = %esp – 4 = 0x800168 – 4 = 0x800164.
- Sau dòng lệnh số 3, %ebp được gán bằng %esp, do đó %ebp = %esp = **0x800164**.

b. Giá trị của thanh ghi **%esp** sau khi thực thi dòng lệnh assembly thứ 4? Giải thích.

- Từ câu a, sau dòng lệnh 2, %esp = 0x800164.
- Dòng lệnh 3 không làm thay đổi %esp.
- Dòng lệnh 4 trừ %esp xuống 24 đơn vị, do vậy %esp = 0x800164 – 24 = 0x800164 – 0x18 = **0x80014C**.

c. Hàm **my_function** có 1 biến cục bộ **str,** là 1 mảng char gồm 2 ký tự. Quan sát mã assembly, hãy cho biết 2 ký tự được gán cho mảng **str** là gì?

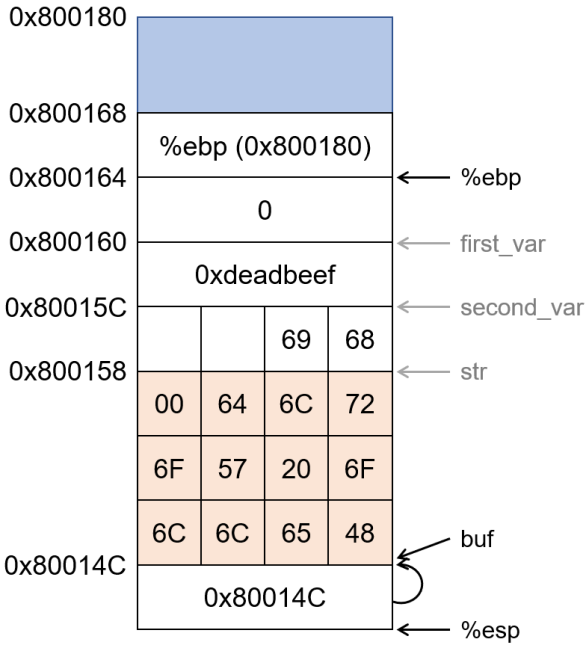
- Tại dòng lệnh assembly 5, 6: giá trị 0 ở vị trí -4(%ebp) → đây là first_var, giá trị 0xdeafbeef ở -8(%ebp) → đây là second_var.
- Tại dòng lệnh thứ 7-8: chương trình lấy 1 số giá trị từ chuỗi các byte ở ô nhớ có nhãn .LC0 đưa vào vị trí -12(%ebp) → đây là mảng str[2].
  - Dòng 7: Lệnh movw lấy 2 byte từ vị trí ô nhớ .LC0 đưa vào %dx, ta được 0x68 và 0x69 → %dx sẽ có giá trị 0x6968 (Little Endian)
  - Dòng 8: Lệnh movw đưa 2 byte trong %dx (0x6968) vào vị trí -12(%ebp) của str[2]. Theo Little Endian, 0x68 được lưu trước, 0x69 lưu sau → **str[0] = 'h', str[1] = 'i'**.

d. Xác định địa chỉ cụ thể của vị trí sẽ lưu chuỗi input nhận về từ hàm **gets()**? Giải thích?

- Hàm gets() cần 1 tham số là vị trí sẽ lưu chuỗi input. Trong assembly, đó là địa chỉ được đẩy vào stack ở dòng 9 - 10 trước khi call gets ở dòng assembly thứ 11.
- Dòng 9 tính toán địa chỉ %ebp – 24 = 0x800164 – 0x18 = 0x80014C lưu vào %eax sau đó push %eax vào stack để làm tham số và gọi gets. Vậy vị trí lưu chuỗi input **là 0x80014C**.

e. Giả sử khi gọi **gets** ở dòng code assembly thứ 13, nhận được 1 chuỗi **"Hello world"**. Vẽ stack frame của **my_function** ngay sau khi hàm **gets** trả về.

Lưu ý: Cần chú thích *địa chỉ, giá trị của các ô nhớ* trong stack frame của **my_function**, bao gồm cả các ô nhớ chứa biến cục bộ, tham số và chuỗi đã nhập với **gets**.



f. **gets** không giới hạn độ dài chuỗi mà nó nhận. Dựa vào stack frame của **my_function** đã vẽ, hãy tìm độ dài tối đa của **buf** (số ký tự) sao cho khi nhập vẫn chưa ghi đè lên bất kỳ ô nhớ quan trọng nào trong stack của **my_function**?

Ô nhớ quan trọng có thể là ô nhớ chứa %ebp cũ của hàm mẹ hoặc địa chỉ trả về trong stack. Dựa vào stack đã vẽ, có thể nhập tối đa chuỗi dài 23 ký tự cho buf. Khi lưu hệ thống sẽ cộng thêm 1 byte của ký tự NULL kết thúc chuỗi, thành chuỗi thành 24 bytes, sử dụng tối đa vùng nhớ của buf mà chưa ghi đè lên các ô nhớ lưu giá trị %ebp cũ của hàm mẹ.

g. Chương trình có thể có lỗ hổng buffer overflow. Thử tìm một chuỗi buf sao cho có thể ghi đè lên biến cục bộ **second_var** một giá trị mới là **0xABDCEF**.

Dựa vào stack đã vẽ, khoảng cách từ vị trí của buf đến biến second_var là 12 + 4 = 16 bytes. Thêm vào đó để ghi đè được hết second_var, còn thêm 4 byte biểu diễn giá trị 0xABDCEF, dưới dạng Little Endian là EF DC AB 00.

Vậy ta có thể nhập 1 chuỗi buf dài 20 bytes, trong đó 16 byte đầu tuỳ ý (khác 0x0A), 4 byte cuối là 4 byte hexan EF DC AB 00.

“a”*16 + “\xEF\xDC\xAB\00”.

Bài tập 9. Trong hệ thống 32 bit, cho mảng **T A[N]** với **T** và **N** chưa biết. Biết tổng kích thước của mảng A là **28 bytes**, T là 1 kiểu dữ liệu cơ bản.

a. Xác định **T** và **N**? Giải thích? Liệt kê tất cả các trường hợp thỏa mãn và dạng khai báo của mảng A với mỗi **T** và **N** tìm được.

Ta có sizeof(T)*N = 28, trong đó N là 1 số nguyên dương, sizeof(T) ∈ {1, 2, 4, 8}
Ta lập luận các trường hợp có thể có:

Sizeof(T)	1	2	4	8
N	28	14	7	28/8 (loại)

Như vậy ta có các trường hợp:
- sizeof(T) = 1 → N = 28, ta có char A[28]
- sizeof(T) = 2 → N = 14, ta có short A[14]
- sizeof(T) = 4 → N = 7, ta có int A[7] hoặc float A[7]...

b. Giả sử với 1 trường hợp của **T A[N]** ở trên, ta có đoạn mã C và assembly tương ứng truy xuất các phần tử của mảng như bên dưới. Hãy điền vào những chỗ còn trống ở 2 đoạn code?
Từ đoạn mã assembly, %eax là địa chỉ của mảng A, truy xuất các phần tử của mảng A sẽ cần tới các biểu thức tính toán địa chỉ dựa trên %eax.
- Dòng lệnh 2: gán 0 vào vị trí (%eax) → phần tử A[0], sử dụng lệnh movw tức sẽ gán 2 bytes.
- Dòng lệnh 6: sử dụng %eax là địa chỉ của A, %ecx là chỉ số i trong biểu thức để tính địa chỉ là &A + 2*i - 2 = &A + 2*(i-1), cũng sử dụng movw
- Dòng lệnh 8: tương tự dòng 6, địa chỉ tính được là &A + 2*i, cũng sử dụng movw.

Từ biểu thức tính địa chỉ phần tử và lệnh movw, nhận định kích thước của 1 phần tử mảng A là 2 byte → short A[7].

Code C

```
1  short A[7];   // khai báo mảng A
2  A[0] = 0;
3  for (int i = 1; i < 7; i ++)
4  {
5      A[i] = A[i-1] + i;
6  }
```

Code assembly

```
1      movl    $A, %eax    // address of A
2      movw    $0, (%eax)
3      movl    $1, %ecx    // chỉ số i
4  .L1:
5      xorl    %ebx, %ebx  // %ebx  = 0
6      movw    -2(%eax, %ecx, 2), %bx  //%bx = A[i-1]
7      addl    %ecx, %ebx              // %ebx += i
8      movw    %bx, (%eax, %ecx, 2)    // A[i] = %ebx
9      incl    %ecx                    // i++
10     cmpl    $7, %ecx
11     jl      .L1
```

c. Cho biết sau khi thực hiện đoạn code trên, ta thu được mảng A với các giá trị phần tử như thế nào?

A[0] = 0. Từ i = 1, A[i] = A[i-1] + i.
Vậy ta có mảng A[7] gồm 7 phần tử có giá trị lần lượt là 0, 1, 3, 6, 10, 15, 21.

Bài tập 10. Cho các định nghĩa sau trong code C, với giá trị N chưa biết.

```
1       # define N ?
2   void matrix_set_val(int A[N][N], int val)
3   {
4       int i;
5       for (i = 0 ; i < N; i++)
6               A[i][i] = val;
7   }
```

Và đoạn code assembly tương ứng được tạo bởi GCC:

Địa chỉ mảng A lưu tại ô nhớ (%ebp+8), giá trị val lưu tại ô nhớ (%ebp+12)

```
1       movl   8(%ebp), %ecx
2       movl   12(%ebp), %edx
3       movl   $0, %eax
4   .L14:
5       movl   %edx, (%ecx,%eax)
6       addl   $68, %eax
7       cmpl   $1088, %eax
8       jne    .L14
```

Hãy phân tích đoạn mã assembly trên và xác định giá trị của N?

Trong vòng lặp có truy xuất phần tử A[i][i] thì trong code assembly sẽ có dòng code tính toán địa chỉ của phần tử này trong mảng A dựa trên i.

Từ các dòng code assembly 1, ta có **%ecx = A** là địa chỉ của mảng**,** dòng 4 – 8 là vòng lặp truy xuất các phần tử của mảng A[N][N].

- Dòng lệnh 5: Địa chỉ của phần tử **A[i][i] = %ecx + %eax = A + distance**, sau mỗi vòng lặp distance sẽ được cập nhật

→ Code assembly tính địa chỉ của phần tử thông qua khoảng cách giữa phần tử đó đến đầu mảng, không trực tiếp dùng chỉ số i. (Xem slide 11 trong Tuần Array-Structure).

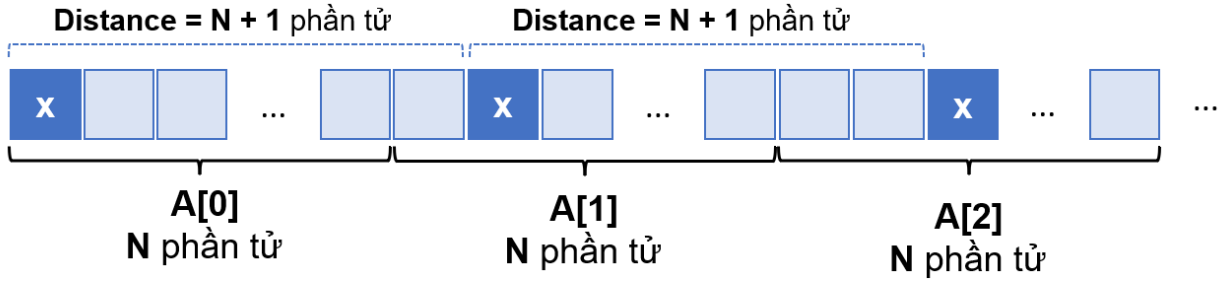
Có 2 cách để tìm kích thước N của mảng A[N][N]

- **Cách 1: dựa trên khoảng cách được cập nhật qua từng vòng lặp**

- Từ code C ta có, mỗi vòng lặp sẽ truy xuất 1 phần tử của 1 mảng con khác nhau trong A.
  Vòng lặp **0**: Phần tử thứ **0** của mảng con **A[0]: A[0][0]**
  Vòng lặp **1**: Phần tử thứ **1** của mảng con **A[1]: A[1][1]**
  Vòng lặp **2**: Phần tử thứ **2** của mảng con **A[2]: A[2][2]**
  …
  Vòng lặp **i**: Phần tử thứ **i** của mảng con **A[i]: A[i][i]**

Vòng lặp **i+1**: Phần tử thứ **i+1** của mảng con **A[i+1]: A[i+1][i+1]**

Ta có thể vẽ minh họa vị trí của những phần tử này trong mảng A như bên dưới, các phần tử được truy xuất sẽ được đánh dấu x.



- Xác định distance sau mỗi vòng lặp được cập nhật lên bao nhiêu, chính là khoảng cách giữa 2 phần tử cần truy xuất liên tiếp là A[i][i] và A[i+1][i+1]:

Từ hình minh họa, có thể thấy, **distance** giữa các vị trí của các phần tử cần truy xuất liên tiếp bằng **kích thước của (N + 1) phần tử = 4*(N + 1)** → đây là khoảng được cập nhật thêm vào distance sau mỗi vòng lặp để đi đến địa chỉ của phần tử tiếp theo cần truy xuất.

- Mặt khác ở dòng lệnh 6, giá trị distance được cập nhật tăng thêm 68, vậy ta có **4*(N+1) = 68 → N = 16** hay **A[16][16]**.
- Có thể kiểm chứng lại với dòng lệnh 7, phần tử cuối cùng có thể truy xuất trong mảng A là A[15][15], khi đó distance lớn nhất sẽ là 15*16*4 + 15*4 = 1020, là hợp lệ. Giá trị distance cập nhật ngay sau đó 1020 + 68 = 1088 sẽ là điều kiện dừng.

- **Cách 2: dựa trên điều kiện dừng**

- Từ code C, ta có mỗi vòng lặp truy xuất phần tử A[i][i] của ma trận. Với định nghĩa A[N][N], ta có phần tử cuối cùng có thể truy xuất được sẽ là A[N-1][N-1], ở vòng lặp kế tiếp sẽ là A[N][N], tuy nhiên điều kiện kiểm tra sẽ dừng khi gặp phần tử này.
- Sau mỗi vòng lặp, distance sẽ được cập nhật thành khoảng cách từ địa chỉ của A đến phần tử cần truy xuất. Với khai báo mảng integer, ta có distance hợp lệ lớn nhất tương ứng với phần tử **A[N-1][N-1]**, và phần tử ngay tiếp theo **A[N][N]** sẽ không hợp lệ, do đó distance của A[N][N] cũng sẽ không hợp lệ.
- Từ dòng assembly 7, ta thấy distance này nếu **bằng 1088** thì sẽ dừng vòng lặp, các trường hợp nhỏ hơn vẫn truy xuất phần tử bình thường, có thể thấy 1088 là giá trị distance không hợp lệ đầu tiên, theo logic sẽ ứng với A[N][N] nên $distance_{A[N][N]}$ = 1088.

  Ta có $distance_{A[N][N]}$ = 4*N*N + 4*N = 1088 (cách đầu mảng N mảng 1 chiều N phần tử integer + N phần tử)

  → **N = 16**.

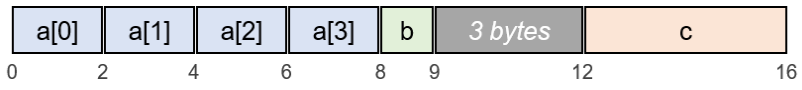
Bài tập 11. Cho struct có định nghĩa như bên dưới trong Linux 32-bit, có yêu cầu alignment.

```
1  typedef struct {
2      short a[4];
3      char b;
4      int c;
5  } str1;
```

Một hàm func được dùng để gán giá trị cho thành phần a[i] và c của struct, kết quả trả về là giá trị của thành phần c như bên dưới.

```
1  int func(int i, int val)
2  {
3      str1 s;
4      s.c = 1;
5      s.a[i] = val;
6      return s.c;
7  }
```

a. Vẽ hình minh họa việc cấp phát struct trên trong bộ nhớ?



Khi tính alignment trong Linux 32 bit:

- Các phần tử của mảng a là short (2 bytes), K = 2 nên cần đặt ở vị trí địa chỉ (offset) chia hết cho 2.
- b là char (1 byte), K = 1 nên có thể đặt tùy ý.
- c là int (4 byte), K = 4 nên cần đặt ở vị trí địa chỉ (offset) chia hết cho 4. Địa chỉ (offset) sau khi cấp phát xong b là 9, chưa chia hết cho 4 nên cần chèn thêm 3 bytes trống để đi đến vị trí gần nhất chia hết cho 4 là offset 12, tại đây có thể lưu c.

Trong Linux 32 bit, yêu cầu căn chỉnh chung K của struct này là 4, tổng kích thước cần chia hết cho 4. Cấp phát xong c, tổng kích thước là 16 bytes, đã chia hết cho 4 nên không cần thêm byte trống.

b. Tổng kích thước của struct trên là bao nhiêu?

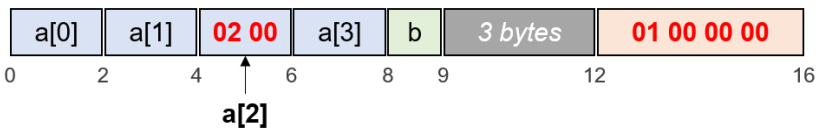
Tổng kích thước của struct là 16 bytes.

c. Tìm giá trị trả về của hàm **func** với các tham số sau? Giải thích các thay đổi có trong vùng nhớ của struct?

Giả định chương trình được biên dịch với compiler chỉ warning khi có truy xuất ngoài mảng, vẫn cho chương trình chạy bình thường.

- **func**(2, 2)

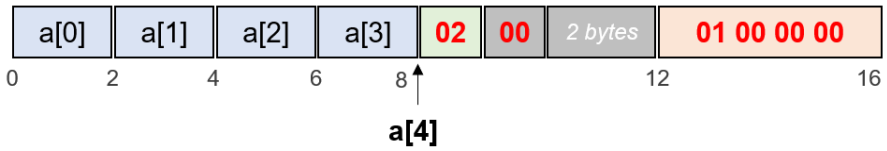
Hàm **func(2,2)** gán giá trị cho c và a[2] trong str1. Cụ thể gán c = 1, sau đó gán a[2] = 2, các byte được lưu trữ dưới bộ nhớ như bên dưới (Little Endian).



Hàm trả về s.c, đọc 4 byte từ vị trí của c được giá trị 1, nên func(2,2) = 1.

- **func**(4, 2)

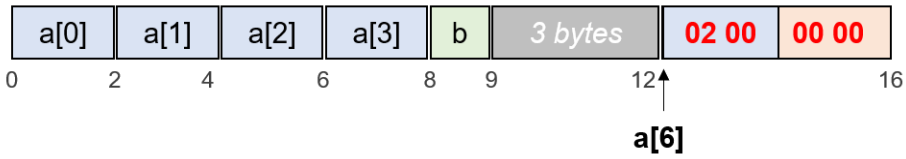
Hàm **func(2,2)** gán giá trị cho c và a[4] trong str1, cụ thể gán c = 1, sau đó gán a[4] = 2. Tuy nhiên mảng a có 4 phần tử, index lớn nhất là 3, với index = 4, hệ thống sẽ truy xuất ra vùng nhớ bên ngoài phía sau mảng. Cụ thể, a[4] sẽ có offset là 8, do chiếm 2 bytes nên ứng với vị trí của thành phần b và 1 byte trong 3 byte trống được chèn thêm. Khi gán giá trị, các byte được lưu trữ dưới bộ nhớ như bên dưới (Little Endian).



Hàm trả về s.c, đọc 4 byte từ vị trí của c được giá trị 1, nên func(2,2) = 1.

- **func**(6, 2)

Hàm **func(6,2)** gán giá trị cho c và a[6] trong str1, cụ thể c = 1, gán a[6] = 2. Tuy nhiên mảng a có 4 phần tử, index lớn nhất là 3, với index = 6, hệ thống sẽ truy xuất ra vùng nhớ bên ngoài phía sau mảng. Cụ thể, a[6] sẽ có offset là 12, chiếm 2 bytes nên sẽ ứng với vị trí của 2 byte đầu tiên của thành phần c (xem hình dưới).
Do a[6] được gán sau nên giá trị gán cho a[6] sẽ ghi đè lên vùng nhớ của thành phần c, làm thay đổi giá trị của c như hình dưới.



Hàm trả về s.c, đọc 4 byte từ vị trí của c được giá trị 2, nên func(2,2) = 2.

Bài tập 12. Cho 2 định nghĩa struct với 2 giá trị A và B chưa biết.

```
typedef struct {
    short x[A][B];          /* Hằng số A và B chưa biết */
    int y;
} str1;

typedef struct {
    char array[B];          /* Hằng số B chưa biết */
    int t;
    short s[B];
    int u;
} str2;
```

Cho đoạn code C cùng đoạn mã assembly tương ứng như bên dưới:

```
void setVal(str1 *r1, str2 *r2)
{
    int v1 = r2->t;
    int v2 = r2->u;
    r1->y = v1+v2;
}
```

```
setVal:
    movl   12(%ebp), %eax
    movl   36(%eax), %edx
    addl   12(%eax), %edx
    movl   8(%ebp), %eax
    movl   %edx, 92(%eax)
```

Dựa vào tương quan giữa 2 đoạn mã C và assembly, xác định 2 giá trị **A** và **B**, biết hệ thống 32 bit và có yêu cầu alignment.

Trong code C có dòng lệnh truy xuất các phần tử q->t và q->u (kiểu str2) và phần tử p->y (kiểu str1) thì trong code assembly sẽ có dòng tính toán địa chỉ của các phần tử này (lưu ý có alignment).

- Xét truy xuất phần tử của **r2** kiểu **str2**:

+ **str2 *r2** (địa chỉ của **r2**) là tham số thứ 2 trong hàm **setVal** nên ở vị trí 12(%ebp), sau đó lưu vào %eax ở dòng assembly thứ 2 → 2 dòng code assembly 3 và 4 có truy xuất các ô nhớ dựa trên địa chỉ của **r2** đang lưu %eax là 36(%eax) và 12(%eax), chính là truy xuất t và u.
+ Trong định nghĩa **str2** của **r2**, t khai báo trước u → t có offset nhỏ hơn u → 12(%eax) là t, 36(%eax) là u.
+ Khoảng cách (hay offset) của t trong **str2** sẽ phụ thuộc vào kích thước thành phần phía trước là mảng gồm **B** ký tự **array**. Từ code assembly, t có địa chỉ cách địa chỉ của struct 12 byte nên ta có: B + x = 12, với B là số phần tử của mảng char array, x có thể là các byte trống chèn thêm để đảm bảo căn chỉnh cho t. Số byte trống được chèn thêm sẽ luôn nhỏ hơn yêu cầu căn chỉnh của t, do đó t kiểu int có yêu cầu alignment là 4 nên 0 <= x < 4 nên **8 < B <= 12** (9, 10, 11 hoặc 12).
+ t có offset 12, chiếm 4 byte nên mảng **short s[B]** bắt đầu từ offset 16. Tiếp đó u có offset là 36 nên ta có 16 + 2*B + y = 36 hay 2*B + y = 20, với y là các byte trống cần thêm để đảm bảo căn chỉnh cho u có kiểu dữ liệu int 4 byte. Ta có 0 <= y < 4 nên 16 < 2*B <= 20 → 8 < B <= 10. Vậy **B = 9** hoặc **B = 10.**

- Xét truy xuất phần tử của **r1** kiểu **str1**:

  + **str1 * r1** (địa chỉ của **r1**) là tham số thứ 1 trong setVal nằm ở vị trí 8(%ebp), sau đó lưu trong %eax ở dòng assembly thứ 5 → dòng code assembly thứ 6 truy xuất ô nhớ dựa trên địa chỉ của **r1** (kiểu str1) đang lưu %eax là 92(%eax), chính là truy xuất y.
  + Trong định nghĩa str1, offset của y trong struct sẽ phụ thuộc vào kích thước mảng 2 chiều **short x[A][B]** phía trước. Ta có kích thước của mảng x sẽ bằng 2*A*B.
  + Để đảm bảo vị trí offset của y sẽ bắt đầu ở vị trí địa chỉ chia hết cho yêu cầu alignment là 4 của int thì offset sẽ có dạng 2*A*B + z với 0 <= z < 4.
  + Từ dòng lệnh assembly 6 ta có 2*A*B + z = 92 với B = 9 hoặc B = 10. Thay thế lần lượt 2 giá trị B vào biểu thức sao cho 0 <= z < 4 và A nguyên dương.
    - Với B = 9: 18A + z = 92.

      z = 0 → A = 5.11 (loại)

      z = 1 → A = 5.05 (loại)

      z = 2 → A = 5

      z = 3 → A = 4.94 (loại)

    - Với B = 10: 20A + z = 92, không có trường hợp nào của z để 0 <= z < 4 và A nguyên.
  + Như vậy, chỉ tìm được 1 trường hợp duy nhất với B = 9 và A = 5 thì thoả 0 <= z < 4 để đảm bảo căn chỉnh và offset tính toán được phù hợp với đề bài.

Vậy **A = 5, B = 9**.

Bài tập 13. Cho 2 file **main.c** và **fib.c** như sau.

```
/* main.c */

1. void fib (int n);
2. int main (int argc, char** argv
3. {
4.     int n = 0;
5.     sscanf(argv[1], "%d", &n);
6.     fib(n);
7. }
```

```
/* fib.c */

1. #define N 16
2. static unsigned int ring[3][N];
3. void print_bignat(unsigned int* a){
4.     int i;
5.     ...
6. }
7. void fib (int n) {
8.     int i;
9.     static int carry;
10.    ...
11.}
```

Hoàn thành bảng sau về các symbol có trong symbol table có trong 2 mô-đun main.o và fib.o, xác định các symbol là ***local/global*** hay ***external***, ***strong*** hay ***weak***.

- Ghi '-' ở cả 2 cột nếu tên không có trong symbol table của mô-đun tương ứng.
- Ghi N/A ở cột ***Strong hay weak*** nếu loại symbol là local.

Symbol table của main.o

Tên symbol	Loại symbol	Strong hay weak
main	global	strong
fib	external	strong
n	-	-

Symbol table của fib.o

Tên symbol	Loại symbol	Strong hay weak
ring	local	N/A
print_bignat	global	strong
fib	global	strong
canary	local	N/A