CS 354: Intro to Computer Systems (Spring 2018 @ Epic)

University of Wisconsin-Madison
Department of Computer Sciences

Final Exam

Monday, May 7thth 2018

5:30 pm - 7:30 pm

The Beatles

There are **eighteen (18) total numbered pages** with **ten (10) questions**.

PLEASE READ THESE INSTRUCTIONS CAREFULLY!

1. You should **assume** the following for all questions **unless otherwise specified**.
   (a) The size of an **int** is **4 bytes**, and the size of a **char** is **1 byte**.
   (b) The machine is **little-endian**.
   (c) The machine uses **32 bits (4 bytes)** for representing a **memory address**. In other words, the size of a **pointer** is **4 bytes**.
   (d) The machine uses **2's complement** for representing **signed** numbers.
   (e) The **x86 assembly** instructions follow **AT&T** syntax discussed in class.
2. **Calculators** (and other electronic devices) are **NOT allowed** during this exam.
3. You are allowed to use both sides of one **8.5 x 11 inch cheat sheat**.
4. There is a mix of some **easy and hard questions** on this exam. So, try attempting as many questions as possible without getting struck on a single hard question.
5. Please **read all questions** carefully!

Good luck with your exam!

Please write your **FULL NAME** below.

FULL NAME: GERALD

Grading Page

Question	Points Scored	Maximum Points
1		10
2		10
3		10
4		10
5		10
6		10
7		10
8		10
9		10
10		10
Total		**100**

1. Pointers in Assembly

   (a) For the following assembly routine, fill in the missing parts of the corresponding C function.

Assembly Routine	C Function

`func:`
`  pushl  %ebp`
`  movl  %esp, %ebp`
`  movl  8(%ebp), %eax`
`  cmpl  12(%ebp), %eax`
`  jne .L8`
`  movl  $0, %eax`
`  jmp .L9`
`.L8:`
`  movl  8(%ebp), %eax`
`  movl  (%eax), %edx`
`  movl  12(%ebp), %eax`
`  movl  (%eax), %eax`
`  cmpl  %eax, %edx`
`  jne .L10`
`  movl  $1, %eax`
`  jmp .L9`
`.L10:`
`  movl  $-1, %eax`
`.L9:`
`  popl  %ebp`
`  ret`
	`int func(int *ap, int *bp) {`
	`  // Assume ap and bp are not NULL.`
	`  if (ap == bp )`
	`  return 0;`
	`  else if (*ap == *bp )`
	`  return 1;`
	`  else`
	`  return -1;`
	`}`

(3) (3)

   (b) Which among the three C functions (`f1()`, `f2()`, `f3()`) does this assembly code correspond to?

```
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%edx
movl 12(%ebp),%eax
movl %ebp,%esp
movl (%edx),%edx
addl %edx,(%eax)
movl %edx,%eax
popl %ebp
ret
```

ANSWER: f3 (4)

```
int f1(int *ap, int *bp)
{
    int a = *ap;
    int b = *bp;
    return a+b;
}
```

```
int f2(int *ap, int *bp)
{
    int b = *bp;
    *bp += *ap;
    return b;
}
```

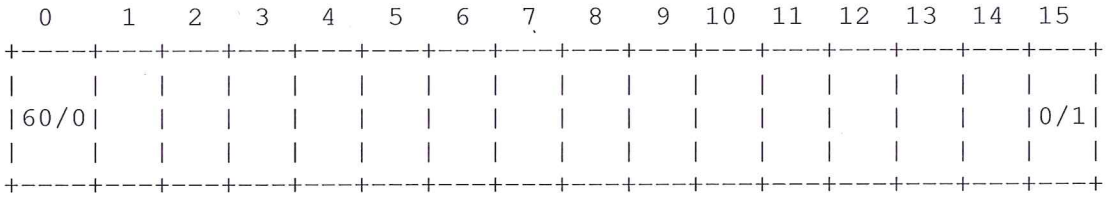
```
int f3(int *ap, int *bp)
{
    int a = *ap;
    *bp += *ap;
    return a;
}
```

-1 for each mistake upto 10 mistakes.

2. Dynamic Memory Allocation - Basics

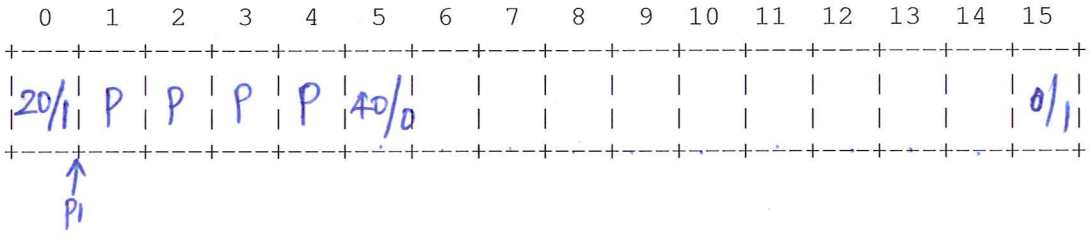
Consider the following small heap which has a total of **64 bytes** (including the block that mark the end of the heap). A block header has the format **x/y** where x denotes the **size of the block including the header** in bytes and y denotes if the block is **free (0)** or **allocated (1)**. The end of the heap is denoted by a special block with the value 0/1 which means that the block size is 0 bytes and that the block is allocated. The dynamic memory allocator is **single word (4 bytes) aligned**. The **indices** in the heap below denote the **word number** (not the byte number). Size of **one word** is **4 bytes**. All blocks in the heap has an **header** of size **4 bytes**. The allocator uses a **best-fit** allocation policy. When allocating memory, blocks are **split** if there is more space to accommodate new free blocks. When freeing memory, **immediate coalescing** with the adjacent free blocks happen, if needed. Blocks with **zero payload** are NOT allowed.

The initial state of this heap is shown below.

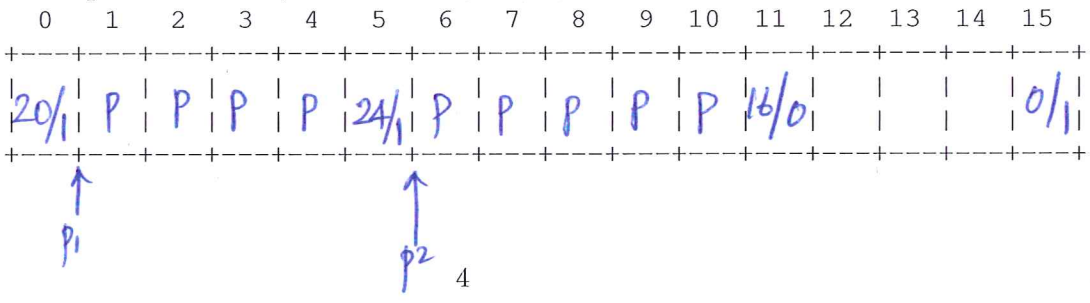


Write the contents (i.e., state) of the heap **after** the following **malloc()/free()** requests. You should assume that these requests are serviced one after the other. i.e., the state of the heap should build on top of the previous state. Mark the **payload** in an allocated block using a **'P'** and the **padding** using a **'X'**. You should write the **header** for both allocated and free blocks in the format **x/y** (as discussed above). If a block is free, you should leave the contents of the block to be empty as shown above in the initial state. `sizeof(int) = 4 bytes`.

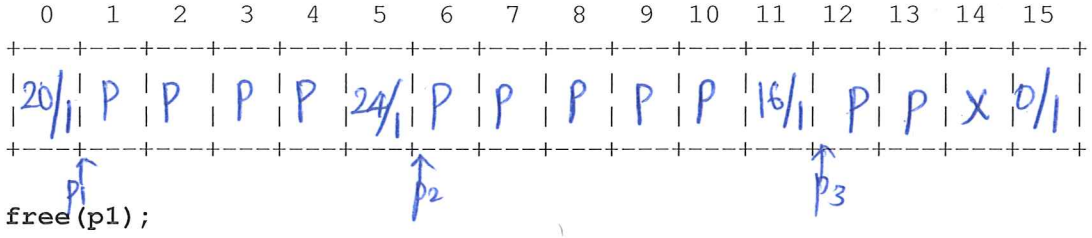
```
int *p1 = malloc(4 * sizeof(int));
```




```
int *p2 = malloc(5 * sizeof(int));
```

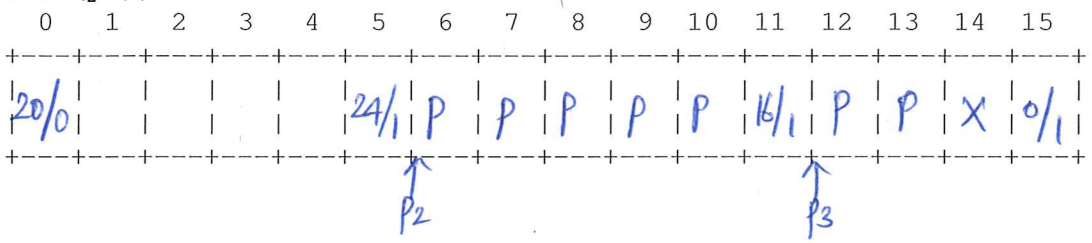


```
int *p3 = malloc(2 * sizeof(int));
```



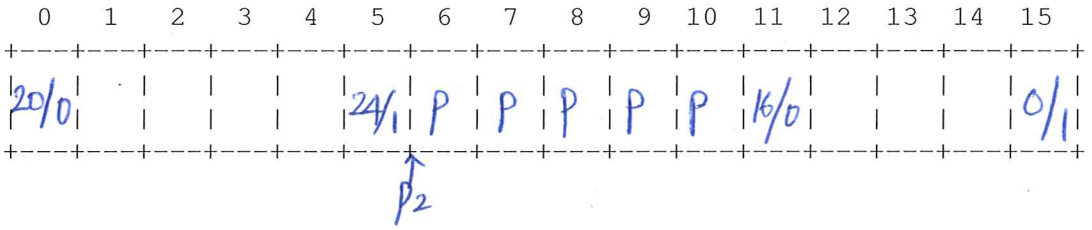
1

```
free(p1);
```



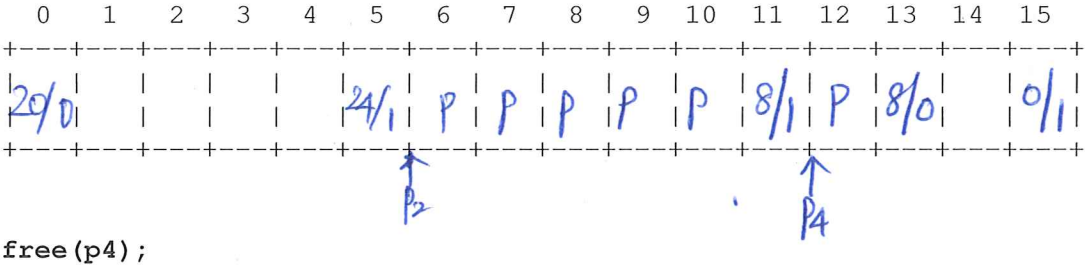
1

```
free(p3);
```



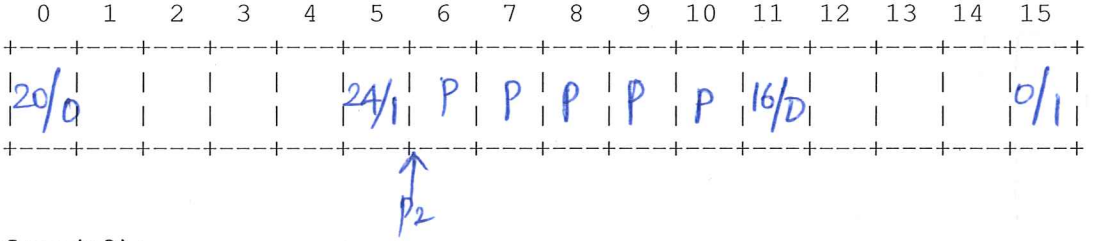
1

```
int *p4 = malloc(sizeof(int));
```



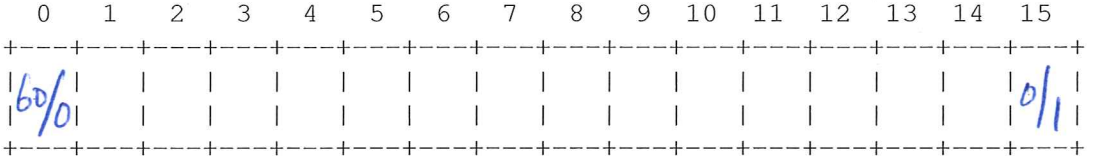
2

```
free(p4);
```



2

```
free(p2);
```



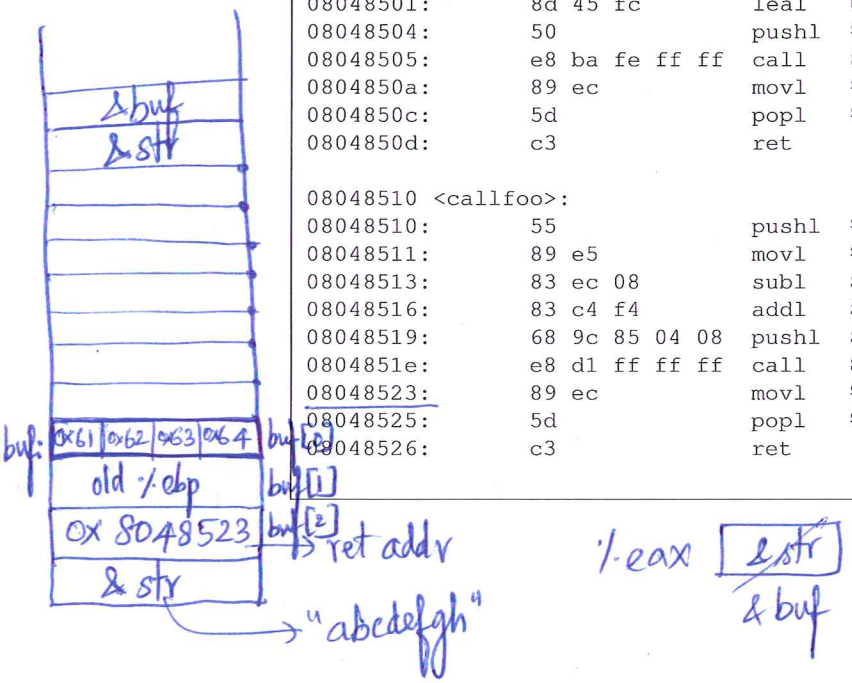
1

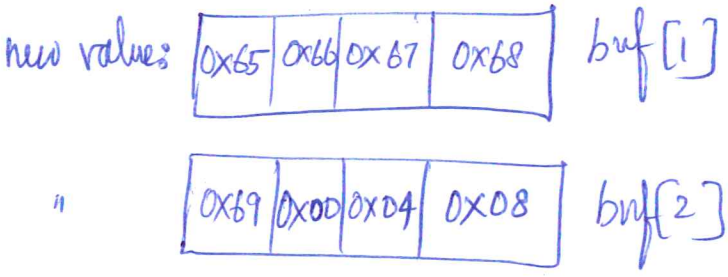
3. Stack Smashing

Consider the following C functions and their corresponding assembly functions that were generated on a Linux/x86 machine.

```
/* copy string x to buf */
void foo(char *x) {
  int buf[1];
  strcpy((char *)buf, x);
}

void callfoo() {
  foo("abcdefghi");
}
```

```
080484f4 <foo>:
080484f4:       55              pushl  %ebp
080484f5:       89 e5           movl   %esp,%ebp
080484f7:       83 ec 18        subl   $0x18,%esp
080484fa:       8b 45 08        movl   0x8(%ebp),%eax
080484fd:       83 c4 f8        addl   $0xfffffff8,%esp
08048500:       50              pushl  %eax
08048501:       8d 45 fc        leal   0xfffffffc(%ebp),%eax
08048504:       50              pushl  %eax
08048505:       e8 ba fe ff ff  call   80483c4 <strcpy>
0804850a:       89 ec           movl   %ebp,%esp
0804850c:       5d              popl   %ebp
0804850d:       c3              ret

08048510 <callfoo>:
08048510:       55              pushl  %ebp
08048511:       89 e5           movl   %esp,%ebp
08048513:       83 ec 08        subl   $0x8,%esp
08048516:       83 c4 f4        addl   $0xfffffff4,%esp
08048519:       68 9c 85 04 08  pushl  $0x804859c   # push string address
0804851e:       e8 d1 ff ff ff  call   80484f4 <foo>
08048523:       89 ec           movl   %ebp,%esp
08048525:       5d              popl   %ebp
08048526:       c3              ret
```





Some useful things to help you solve this problem:

- `strcpy(char *dst, char *src)` copies the string at address `src` (including the terminating '`\0`' character) to address `dst`. It does **not** check the size of the destination buffer.
- Recall that Linux/x86 machines are **Little Endian**.
- You will need to know the hex values of the following characters:

Character	Hex value	Character	Hex value
'a'	0x61	'f'	0x66
'b'	0x62	'g'	0x67
'c'	0x63	'h'	0x68
'd'	0x64	'i'	0x69
'e'	0x65	'\0'	0x00

Now consider what happens on a Linux/x86 machine when `callfoo()` calls `foo()` with the input string "`abcdefghi`".

(a) List the contents of the following memory locations immediately **after** `strcpy` returns to `foo`. Each answer should be an unsigned 4-byte integer expressed as 8 hex digits (e.g., `0x0840A1B1`).

(2) each.

`buf[0]` = 0x64 63 62 61 OR 0x 61 62 63 64 — OK

`buf[1]` = 0x68 67 66 65 OR 0x 65 66 67 68 . — OK

`buf[2]` = 0x08 04 00 69 OR 0x 69 00 04 08 — OK.

(b) Immediately **before** the `ret` instruction at address `0x0804850d` executes, what is the value of the frame pointer register `%ebp`?

`%ebp` = 0x 68 67 66 65 OR 0X 65 66 67 68 -OK.

(c) Immediately **after** the `ret` instruction at address `0x0804850d` executes, what is the value of the program counter register `%eip`?

`%eip` = 0x 08 04 00 69 OR 0X 69 00 04 08 -OK.

4. Caches - Low-level Details

- The memory is **byte** addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Physical addresses are **13 bits** wide.
- The cache is **2-way set associative** (i.e., E = 2), with a **4 byte cache block size** and **16 total lines**.

In the following tables, **all numbers are given in hexadecimal**. The contents of the cache are as follows:

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	09	1	86	30	3F	10	00	0	99	04	03	48
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	2F	81	FD	09	0B	0	8F	E2	05	BD
3	06	0	3D	94	9B	F7	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	B0	39	D3	F7
6	91	1	A0	B7	26	2D	F0	0	0C	71	40	10
7	46	0	B1	0A	32	0F	DE	1	12	C0	88	37

(a) What is the **total size** of the cache (C) in bytes?

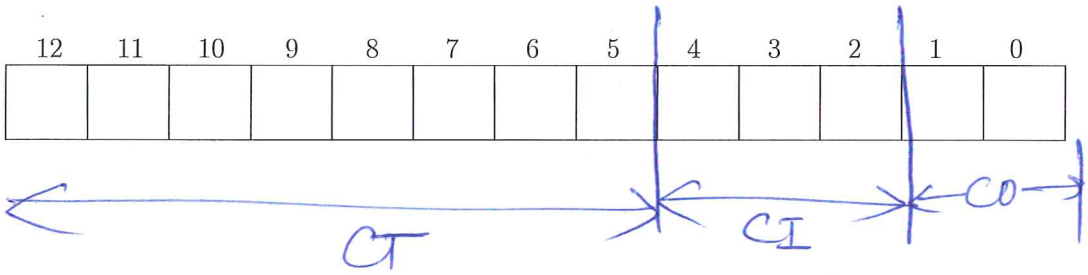
(1) 64 bytes

C = S x E x B = 8 x 2 x 4 = 64

(b) The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

CO The block offset within the cache line
CI The cache index
CT The cache tag

(1)



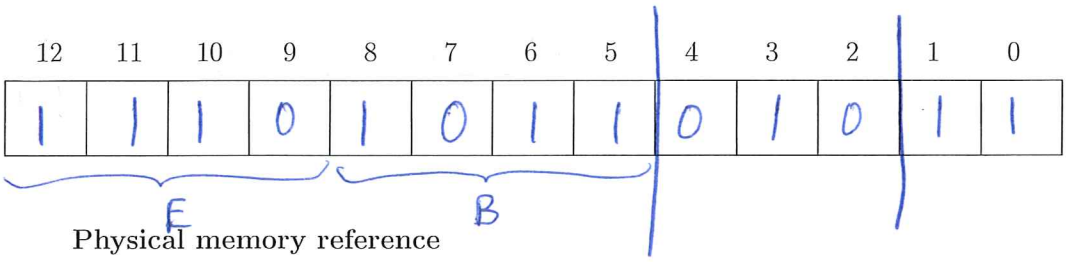
(c) For the given physical address, indicate the cache entry accessed and the cache byte value returned **in hex**. Indicate whether a cache miss occurs.

If there is a cache miss, enter "N/A" for "Cache Byte returned".

Physical address: `0x1D6B`

Physical address format (one bit per box)

(1)



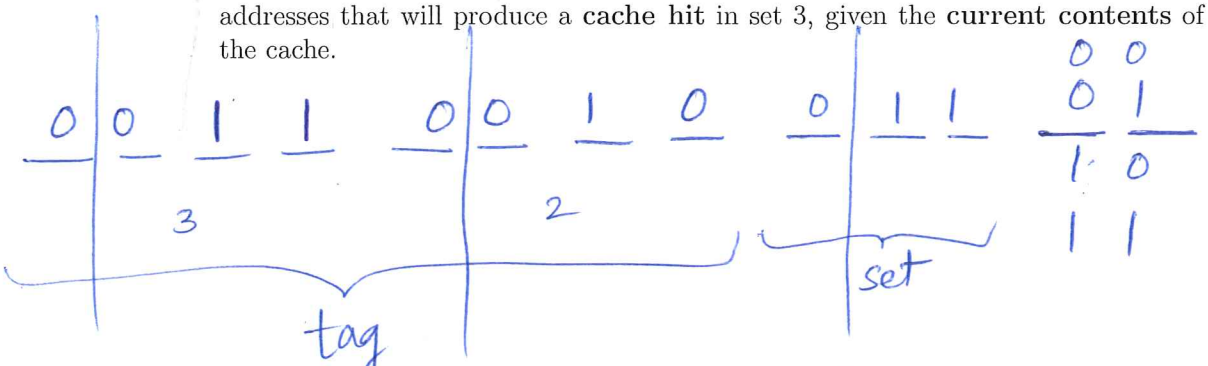
Physical memory reference

(5)

Parameter	Value
Byte offset	0x 3
Cache Index	0x 2
Cache Tag	0x EB
Cache Hit? (Y/N)	N
Cache Byte returned	0x N/A

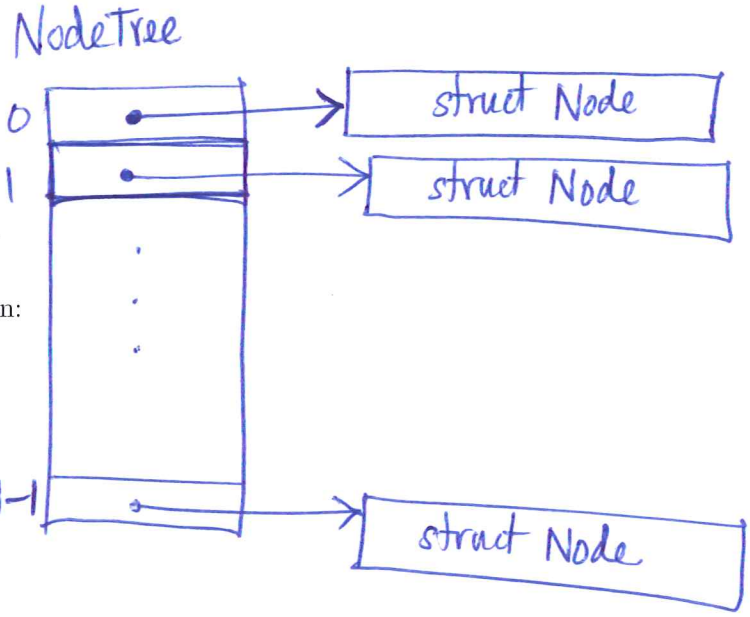
(2)

(d) In the 2-way set associative cache given above, list all the **memory addresses** (in hex) that will **hit** in **set 3**. Remember that you don't need to make an exhaustive list of all addresses that map to set 3. Instead you only need to list the set of addresses that will produce a **cache hit** in set 3, given the **current contents** of the cache.



0x64C, 0x64D, 0x64E, 0x64F

5. Arrays and Structures in Assembly



Consider the following C declaration:

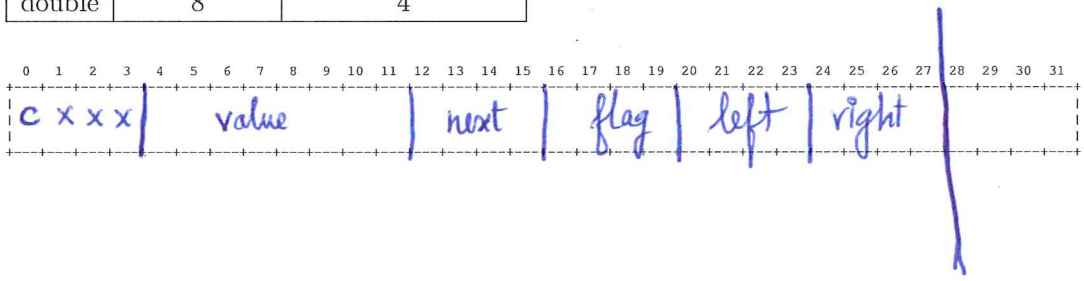
```
struct Node{
   char c;
   double value;
   struct Node* next;
   int flag;
   struct Node* left;
   struct Node* right;
};

typedef struct Node* pNode;

/* NodeTree is an array of N pointers to Node structs */
pNode NodeTree[N];
```

(a) Using the template below (allowing a maximum of 32 bytes), indicate the allocation of data for a **Node struct**. Mark off and label the areas for each individual element (there are 6 of them). Mark the parts that are allocated, but not used (to satisfy alignment) using a 'X' sign. Clearly indicate the **right hand boundary** of the data structure with a **vertical line**.

Assume the following sizes and alignment for the data types.

Type	Size (bytes)	Alignment (bytes)
char	1	1
int	4	4
double	8	4

(2)



(continued on the next page)

To access the base of the i^{th} element in the array of pointers

```
sall $2, %edx      # 4*i in %edx
leal (%eax, %edx), %eax   # addr of the ith array elem in NodeTree
movl (%eax), %eax         # get the value at the ith elem. in NodeTree array which is the base of the struct Node.
```

NodeTree[i]

(b) For each of the four C references below, please indicate which assembly code section (labeled A – F) places the value of that C reference into register %eax. If no match is found, please write "NONE" next to the C reference.

The initial register-to-variable mapping for each assembly code section is:

```
%eax = starting address of the NodeTree array
%edx = i
```

(8)

C References	Matching Assembly Code
`NodeTree[i]->flag`	D
`NodeTree[i]->left->left->c`	NONE
`NodeTree[i]->next->next->flag`	F
`NodeTree[i]->right->left->left`	B

Linux/IA32 Assembly:

A.
```
sall $2, %edx
leal (%eax,%edx),%eax
movl 16(%eax),%eax
```
3rd line missing

B.
```
sall $2,%edx
leal (%eax,%edx),%eax
movl (%eax),%eax
movl 24(%eax),%eax
movl 20(%eax),%eax
movl 20(%eax),%eax
```

same as above.

C:
```
sall $2,%edx
leal (%eax,%edx),%eax
movl 20(%eax),%eax
movl 20(%eax),%eax
movsbl (%eax),%eax
```

D:
```
sall $2,%edx
leal (%eax,%edx),%eax
movl (%eax),%eax
movl 16(%eax),%eax
```

E:
```
sall $2, %edx
leal (%eax,%edx),%eax
movl (%eax),%eax
movl 16(%eax),%eax
movl 16(%eax),%eax
movl 20(%eax),%eax
```

F:
```
sall $2, %edx
leal (%eax,%edx),%eax
movl (%eax),%eax
movl 12(%eax),%eax
movl 12(%eax),%eax
movl 16(%eax),%eax
```

(2) each.

6. Dynamic Memory Allocation - Implementation

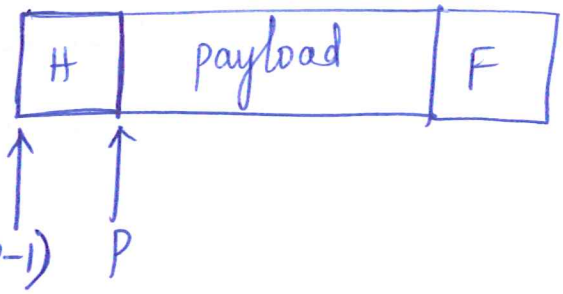
Consider an allocator that uses an **implicit free list**. Each memory block, either allocated or free, has a size that is a multiple of **eight bytes**. Thus, only the **29 higher order bits** in the **header** and **footer** are needed to record block size, which **includes** the header and footer and is represented in units of **bytes**. The usage of the remaining **3 lower order bits** is as follows:

- **bit 0** indicates the use of the current block: 1 for allocated, 0 for free.
- **bit 1** indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- **bit 2** is unused and is always set to be 0.

not relevant to this question →

Five helper routines are defined to facilitate the implementation of `free(void *p)`. The functionality of each routine is explained in the comment above the function definition. Fill in the body of the helper routines, the code section label (i.e., A, B, or C) that implement the corresponding functionality correctly. The **tilde (~) operator** in C is the **ones' complement** operator.

```
/* given a pointer p to an allocated block, i.e., p is a
   pointer returned by some previous malloc()/realloc() call;
   returns the pointer to the header of the block*/
void * header(void* p)
{
  void *ptr;
  ____B____;
  return ptr;
}
```

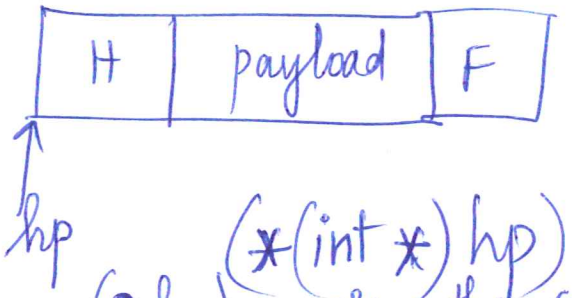



```
A. ptr=p-1
B. ptr=(void *)((int *)p-1)
C. ptr=(void *)((int *)p-4)
```

```
/* given a pointer to a valid block header or footer,
   returns the size of the block */
int size(void *hp)
{
  int result;
  ____C____;
  return result;
}
```

For simplicity, assume 8 bit ints instead of 32 bits.

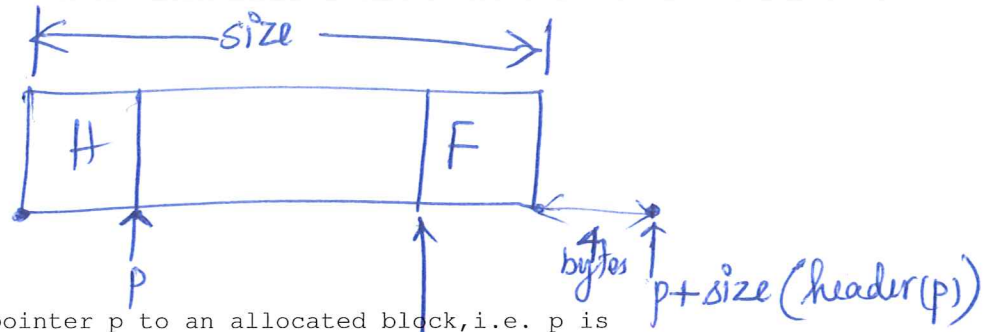
7 = 0000 0111 in 8 bits




```
A. result=(*hp)&(~7)
B. result=((*(char *)hp)&(~5))<<2
C. result=(*(int *)hp)&(~7)
```

~7 = 11111000 (in 8 bits)

⇒ (*(int *)hp) & ~7 gives the size of the block.



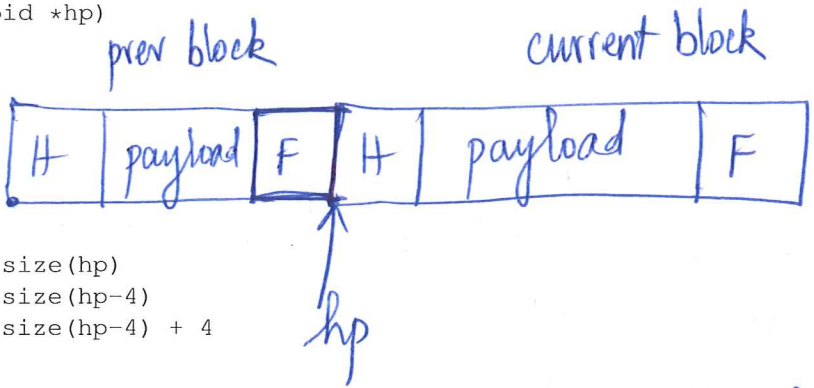

```
/* given a pointer p to an allocated block,i.e. p is
   a pointer returned by some previous malloc()/realloc() call;
   returns the pointer to the footer of the block*/
void * footer(void *p)
{
  void *ptr;
  ____A____;
  return ptr;
}
```

A. ptr=p+size(header(p))-8
B. ptr=p+size(header(p))-4
C. ptr=(int *)p+size(header(p))-2

```
/* given a pointer to a valid block header or footer,
   returns the usage of the currect block,
   1 for allocated, 0 for free */
int allocated(void *hp)
{
  int result;
  ____A____;
  return result;
}
```

A. result=(*(int *)hp)&1
B. result=(*(int *hp)&0
C. result=(*(int *)hp)|1

```
/* given a pointer to a valid block header,
   returns the pointer to the header of previous block in memory */
void * prev(void *hp)
{
  void *ptr;
  ____B____;
  return ptr;
}
```



A. ptr = hp - size(hp)
B. ptr = hp - size(hp-4)
C. ptr = hp - size(hp-4) + 4

size(hp-4) ⇒ size of prev. block as the footer also stores the size of a block.

7. C programming - Iteration vs Recursion

Consider the following two functions to compute the factorial of a number. The function **`ifact()`** computes the factorial in an **iterative** manner and **`rfact()`** computes it in a **recursive** manner.

```
 1 unsigned long long int ifact(unsigned int n) {
 2     long long fact = 1;
 3     for (; n > 1; --n)
 4         fact *= n;
 5     return fact;
 6 }
 7
 8 unsigned long long int rfact(unsigned int n) {
 9     if (n == 0)
10         return 1;
11     else
12         return n * rfact(n - 1);
13 }
```

(a) Assume we run this program on a machine in which memory is very limited (only a few KB). In this case, which version of the factorial function is **better** than the other with respect to **memory usage**? **Explain your answer.**

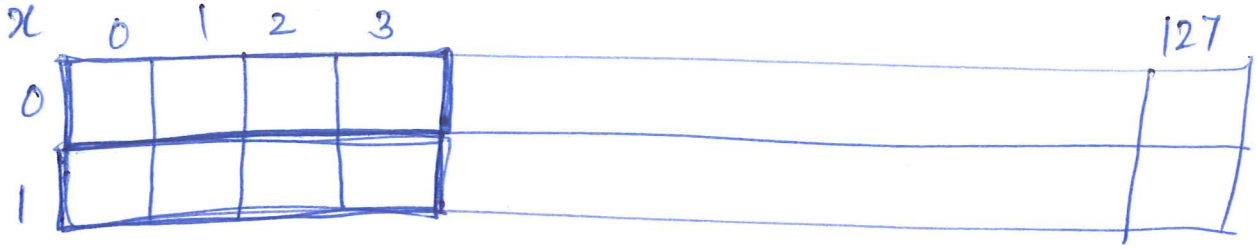
(5)

iterative is better since recursive version may use a lot of stack space.

(5)

(b) Will these factorial functions (ifact, rfact) work for all positive values of n? **Explain your answer.**

No. If the factorial value grows larger than what we can store in unsigned long long int then we get incorrect results.



x[0][i]

$2^7 \times 2^2 = 2^9$

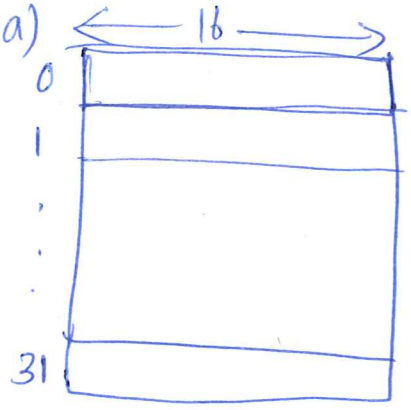
8. Caches - High-level Analysis

(a) You are given the following code to analyze:

```
int x[2][128];
int i;
int sum = 0;
for (i = 0; i < 128; i++) {
    sum += x[0][i] * x[1][i];
}
```

size of $x_{0,0} - x_{0,127}$ = 512 bytes

$\Rightarrow x_{00} - x_{03}$ and $x_{10} - x_{13}$ map to the same set in direct mapped cache of 512 bytes.



Assume we execute this under the following conditions:

- `sizeof(int) = 4`.
- Array x starts at **memory address 0x0** and is stored in **row-major** order.
- In each case below, the **cache is initially empty**.
- The only memory accesses are to the entries of the array x. All other variables are stored in registers.
- Associative caches use **Least Recently Used (LRU)** replacement policy.

b) 64 sets ⇒ whole array x can fit in cache. ∴ only cold misses

Given these assumptions, estimate the **miss rates** for the following cases:

Cache Type	Total cache size (C)	Cache block size (B)	Miss rate (%)
a) Direct-mapped	512 bytes 2^9	16 bytes 2^4	100%
b) Direct-mapped	1024 bytes	16 bytes	25%
c) 2-way set associative	512 bytes	16 bytes	25%
d) 2-way set associative	1024 bytes	16 bytes	25%
e) 2-way set associative	512 bytes	32 bytes	12.5%
f) 2-way set associative	256 bytes	64 bytes	6.25%

S = 32 (a)
S = 64 (b)
S = 16 (c)
S = 32 (d)

6

(b) Assume a fully-associative cache of size 4 blocks. The cache used LRU replacement policy. Given the following sequence of requests, what are the blocks that are found in the cache after all these requests are processed.

```
b0, b1, b2, b3, b4, b3, b2, b0, b4, b1, b3
```

4

State of cache at the end:

0	1	2	3
~~b_0~~ b_4	~~b_1~~ b_0	~~b_2~~ b_3	~~b_3~~ b_1

any order among b_0, b_1, b_3, b_4

order doesn't matter.

c), d)

In a 2-way associative cache irrespective of the total cache size ($C \geq 32$ bytes) the miss rate is going to be the same for a given block size.

e.g. C = 32 bytes, B = 16 bytes, miss% = 25%. Think!

9. Functions in Assembly

Consider the given C function and its corresponding assembly code.

C Function	Assembly Routine

`int proc(void)`
`{`
`  int x,y;`
`  scanf("%x %x", &y, &x);`
`  return x-y;`
`}`
	`proc:`
	`1  pushl  %ebp`
	`2  movl  %esp, %ebp`
	`3  subl  $24, %esp`
	`4  subl  $4, %esp`
	`5  leal  -12(%ebp), %eax`
	`6  pushl  %eax`
	`7  leal  -16(%ebp), %eax`
	`8  pushl  %eax`
	`# .LC0 is pointer to string "%x %x"`
	`9  pushl  $.LC0`
	`10  call  scanf`
	`11  addl  $16, %esp`
	`12  movl  -12(%ebp), %edx`
	`13  movl  -16(%ebp), %eax`
	`14  subl  %eax, %edx`
	`15  movl  %edx, %eax`
	`16  leave`
	`17  ret`

(2) each.

Assume the procedure `proc` starts executing with the following register values. i.e., these are the register values **before** line 1 in `proc` is executed.

Register	Value
`%esp`	`0x80040`
`%ebp`	`0x80060`

Suppose proc calls scanf (line 10), and that scanf reads values `0x46` and `0x53` from the standard input. Assume that the string `"%x %x"` is stored at memory location `0x300070`. Write all values in **hexadecimal**.

(a) What value does `%ebp` get set to on line 2? 0x8003C ∵ pushl %ebp decrements the %esp by 4 bytes.

24 + 4 = 28 = 0x1C
0x8003C − 0x1C = 0x80020

(b) What value does `%esp` get set to on line 4? 0x80020

(c) At what address is local variable x stored? −12(%ebp) = 0x8003C − 0xC = 0x80030

(d) At what address is local variable y stored? −16(%ebp) = 0x8003C − 0x10 = 0x8002C

(e) What is the value of register `%ebp` after `leave` (line 16) is executed?

leave → movl %ebp, %ebp
popl %ebp

0x80060

old value of %ebp that was just popped off the stack.

2.5 each

10. C vs Assembly

Match the C code snippets with their corresponding assembly routines.

C Function	Assembly Routine
`bitwiseOR`	4
`logicalOR`	2
`bitwiseAND`	1
`logicalAND`	3

(See next page for the C and Assembly code snippets)

C Function	Assembly Routine

`int bitwiseOR(int x, int y) {`
`  return x | y;`
`}`
	func1:
	`  pushl  %ebp`
	`  movl  %esp, %ebp`
	`  movl  8(%ebp), %eax`
	`  andl  12(%ebp), %eax`
	`  popl  %ebp`
	`  ret`

`int logicalOR(int x, int y) {`
`  return x || y;`
`}`
	func2:
	`  pushl  %ebp`
	`  movl  %esp, %ebp`
	`  cmpl  $0, 8(%ebp)`
	`  jne .L4`
	`  cmpl  $0, 12(%ebp)`
	`  je  .L5`
	`.L4:`
	`  movl  $1, %eax`
	`  jmp .L6`
	`.L5:`
	`  movl  $0, %eax`
	`.L6:`
	`  popl  %ebp`
	`  ret`

`int bitwiseAND(int x, int y) {`
`  return x & y;`
`}`
	func3:
	`  pushl  %ebp`
	`  movl  %esp, %ebp`
	`  cmpl  $0, 8(%ebp)`
	`  je  .L11`
	`  cmpl  $0, 12(%ebp)`
	`  je  .L11`
	`  movl  $1, %eax`
	`  jmp .L12`
	`.L11:`
	`  movl  $0, %eax`
	`.L12:`
	`  popl  %ebp`
	`  ret`

`int logicalAND(int x, int y) {`
`  return x && y;`
`}`
	func4:
	`  pushl  %ebp`
	`  movl  %esp, %ebp`
	`  movl  8(%ebp), %eax`
	`  orl 12(%ebp), %eax`
	`  popl  %ebp`
	`  ret`