Malloc Lab
&
Midterm Solutions

Recitation 11: Tuesday: 11/08/2016

Malloc

Important Notes about Malloc Lab

- **Malloc lab has been updated from previous years**
- **Supports a full 64 bit address space rather than 32 bit**
- **Addresses have to be 16 bytes aligned rather than 8 bytes**
- **Encourages a new programming style**
  - Use structures instead of macros
  - Study the baseline implementation of implicit allocator to get a better idea
- **Divided into two phases:**
  - Checkpoint 1: Due date: 11/17
  - Final: Due date: 11/28
- **Get a correct, reasonably performing malloc by checkpoint**
- **Optimize malloc by final submission**

Playing with structures

- Consider the following structure, where a ‘block’ refers to an allocation unit
- Each block consists of some metadata (header) and the actual data (payload)

```
/* Basic declarations */

typedef uint64_t word_t;
static const size_t wsize = sizeof(word_t);

typedef struct block {
  // Header contains size + allocation flag
      word_t header;
      char payload[0];
} block_t;
```

Playing with structures

- The contents of the header is populated as follows

```
/* Pack size and allocation bit into single
word */

static word_t pack(size_t size, bool alloc) {

      return size | alloc;

}
```

```
/* Basic declarations */

typedef uint64_t word_t;
static const size_t wsize = sizeof(word_t);

typedef struct block {
  // Header contains size + allocation flag
      word_t header;
      char payload[0];
} block_t;
```

Playing with structures

- How do we set the value in the header, given the block and values ?

```
/* Set fields in block header */

static void write_header(block_t *block,
                         size_t size, bool alloc) {

     block->header = pack(size, alloc);

}
```

```
/* Basic declarations */

typedef uint64_t word_t;
static const size_t wsize = sizeof(word_t);

typedef struct block {
  // Header contains size + allocation flag
      word_t header;
      char payload[0];
} block_t;
```

Playing with structures

- How do we extract the value of the size, given the header ?
- How do we extract the value of the size, given pointer to block ?

```
/* Extract size from header */

static size_t extract_size(word_t word) {

      return (word & ~(word_t) 0x7);

}

/* Get block size */

static size_t get_size(block_t *block) {

      return extract_size(block->header);
}
```

```
/* Basic declarations */

typedef uint64_t word_t;
static const size_t wsize = sizeof(word_t);

typedef struct block {
  // Header contains size + allocation flag
      word_t header;
      char payload[0];
} block_t;
```

Playing with structures

- How do we write to the end of the block ?

```
/* Set fields in block footer */

static void write_footer(block_t *block,
                         size_t size,
                         bool alloc) {

     word_t *footerp = (word_t *)((block->payload) +
                       get_size(block) - 2*wsize);

     *footerp = pack(size, alloc);

}
```

```
/* Basic declarations */

typedef uint64_t word_t;
static const size_t wsize = sizeof(word_t);

typedef struct block {
  // Header contains size + allocation flag
     word_t header;
     char payload[0];
} block_t;
```

Playing with structures

- How do we get to the start of the block, given the pointer to the payload ?

```
/* Locate start of block, given pointer to payload */

static block_t *payload_to_header(void *bp) {

     return (block_t *)(((char *)bp) -
                offsetof(block_t, payload));

}
```

```
/* Basic declarations */

typedef uint64_t word_t;
static const size_t wsize = sizeof(word_t);

typedef struct block {
  // Header contains size + allocation flag
     word_t header;
     char payload[0];
} block_t;
```

Pointers: casting, arithmetic, and dereferencing

Pointer casting

- **Cast from**
  - <type_a>* to <type_b>*
    - Gives back the same value
    - Changes the behavior that will happen when dereferenced
  - <type_a>* to integer/ unsigned int
    - Pointers are really just 8-byte numbers
    - Taking advantage of this is an important part of malloc lab
    - Be careful, though, as this can easily lead to errors
  - integer/ unsigned int to <type_a>*

Pointer arithmetic

- The expression `ptr + a` doesn't mean the same thing as it would if `ptr` were an integer.
- Example:

```
type_a* pointer = …;
(void *) pointer2 = (void *) (pointer + a);
```

- This is really computing:
  - `pointer2 = pointer + (a * sizeof(type_a))`
  - `lea (pointer, a, sizeof(type_a)), pointer2;`
- Pointer arithmetic on `void*` is undefined

Pointer arithmetic

- ```
  int * ptr = (int *)0x12341230;
  int * ptr2 = ptr + 1;
  ```

- ```
  char * ptr = (char *)0x12341230;
  char * ptr2 = ptr + 1;
  ```

- ```
  int * ptr = (int *)0x12341230;
  int * ptr2 = ((int *) (((char *) ptr) + 1));
  ```

- ```
  char * ptr = (char *)0x12341230;
  void * ptr2 = ptr + 1;
  ```

- ```
  char * ptr = (int *)0x12341230;
  void * ptr2 = ptr + 1;
  ```
  ```

Pointer arithmetic

- ```
  int * ptr = (int *)0x12341230;
  int * ptr2 = ptr + 1; //ptr2 is 0x12341234
  ```

- ```
  char * ptr = (char *)0x12341230;
  char * ptr2 = ptr + 1; //ptr2 is 0x12341231
  ```

- ```
  int * ptr = (int *)0x12341230;
  int * ptr2 = ((int *) (((char *) ptr) + 1));
  //ptr2 is 0x12341231
  ```

- ```
  char * ptr = (char *)0x12341230;
  void * ptr2 = ptr + 1; //ptr2 is 0x12341231
  ```

- ```
  char * ptr = (int *)0x12341230;
  void * ptr2 = ptr + 1; //ptr2 is still 0x12341231
  ```
  ```

Pointer dereferencing

- **Basics**
  - It must be a POINTER type (or cast to one) at the time of dereference
  - Cannot dereference expressions with type `void*`
  - Dereferencing a `t*` evaluates to a value with type `t`

Pointer dereferencing

- What gets “returned?”

```
 int * ptr1 = malloc(sizeof(int));
*ptr1 = 0xdeadbeef;

 int val1 = *ptr1;
 int val2 = (int) *((char *) ptr1);
```

What are val1 and val2?

Pointer dereferencing

- What gets "returned?"

```
int * ptr1 = malloc(sizeof(int));
*ptr1 = 0xdeadbeef;

int val1 = *ptr1;
int val2 = (int) *((char *) ptr1);

// val1 = 0xdeadbeef;
// val2 = 0xffffffef;
```

What happened??

Malloc basics

- **What is dynamic memory allocation?**

- **Terms you will need to know**
  - malloc/ calloc / realloc
  - free
  - sbrk
  - payload
  - fragmentation (internal vs. external)
  - coalescing
    - Bi-directional
    - Immediate vs. Deferred

Allocation Example

```
p1 = malloc(4)
```

```
p2 = malloc(5)
```

```
p3 = malloc(6)
```

```
free(p2)
```

```
p4 = malloc(2)
```

Fragmentation

- **Internal fragmentation**
  - Result of **payload** being smaller than block size.
  - `void * m1 = malloc(3); void * m2 = malloc(3);`
  - `m1,m2` both have to be aligned to 16 bytes…
- **External fragmentation**

External Fragmentation

- Occurs when there is enough aggregate heap memory, but no single free block is large enough

```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(6)
```

Oops! (what would happen now?)

- Depends on the pattern of future requests
  - Thus, difficult to measure

Implementation Hurdles

- **How do we know where the blocks are?**
- **How do we know how big the blocks are?**
- **How do we know which blocks are free?**
- **Remember: can't buffer calls to malloc and free… must deal with them real-time.**
- **Remember: calls to `free` only takes a pointer, not a pointer and a size.**
- **Solution: <u>Need a data structure to store information on the "blocks"</u>**
  - Where do I keep this data structure?
  - We can't allocate a space for it, that's what we are writing!

The data structure

- **Requirements:**
  - The data structure needs to tell us where the blocks are, how big they are, and whether they're free
  - We need to be able to CHANGE the data structure during calls to malloc and free
  - We need to be able to find the **next free block** that is "a good fit for" a given payload
  - We need to be able to quickly mark a block as free/allocated
  - We need to be able to detect when we're out of blocks.
    - What do we do when we're out of blocks?

The data structure

- **Common types**
  - Implicit List
    - Root -> block1 -> block2 -> block3 -> …
  - Explicit List (Encouraged for Checkpoint 1)
    - Root -> free block 1 -> free block 2 -> free block 3 -> …
  - Segregated List
    - Small-malloc root -> free small block 1 -> free small block 2 -> …
    - Medium-malloc root -> free medium block 1 -> …
    - Large-malloc root -> free block chunk1 -> …

Implicit List

- **From the root, can traverse across blocks using headers which store the size of the block**
- **Can find a free block this way**
- **Can take a while to find a free block**
  - How would you know when you have to call sbrk?

Explicit List

- **Improvement over implicit list**
- **From a root, keep track of all free blocks in a (doubly) linked list**
  - Remember a doubly linked list has pointers to next and previous
  - Optimization: using a singly linked list instead (how could we do this?)
- **When malloc is called, can now find a free block quickly**
  - What happens if the list is a bunch of small free blocks but we want a really big one?
  - How can we speed this up?

Segregated List

- **An optimization for explicit lists**
- **Can be thought of as multiple explicit lists**
  - What should we group by?
- **Grouped by size – let us quickly find a block of the size we want**
- **What size/number of buckets should we use?**
  - This is up to you to decide

Malloc Lab is Out!

- **Incrementally improve your design**
- **Start from an implicit allocator**
- **Heap Checker and GDB, the keys to debugging**
- **Read the handout carefully**
- **More on the design and data structures to use in next recitation**
- **Warnings:**
  - Most existing Malloc literature from the book has slightly different guidelines, may be out of date

Midterm Review

Question 1a. ISA Interface

1. (10 points) **The Big Picture**

   (a) (3 points) For each term below, indicate whether it is above or below the ISA interface. Circle your answer.

   A. Compiler (above or below)

   B. Cache Memory (above or below)

   C. Processor Pipeline (above or below)

   D. Operating System (above or below)

   E. Amdahl's Law (above or below)

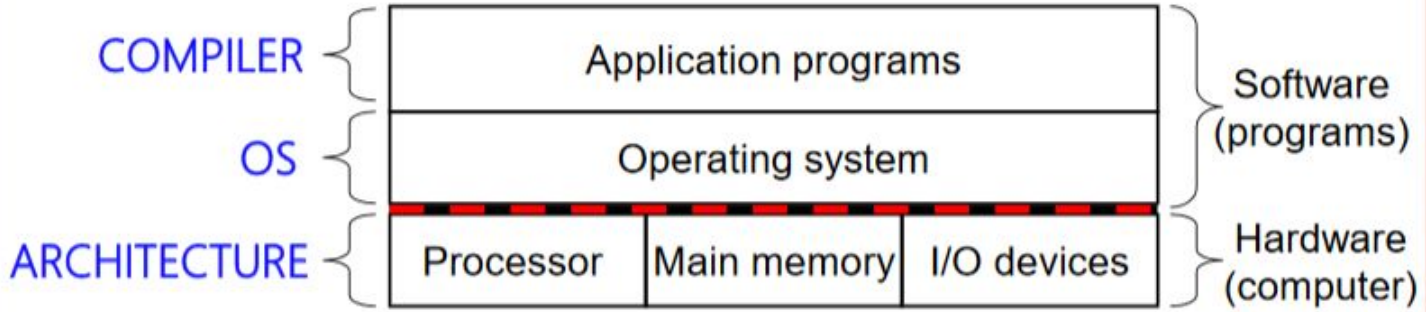
   F. Moore's Law (above or below)

What is the ISA?

Recall Lec. 1 Slide 19...

Anatomy of a Computer System: SW/HW

- What is a Computer System?
  - Software + Hardware
  - Programs + Computer ➔ [Application program + OS] + Computer
  - Programming Languages + Operating Systems + Computer Architecture



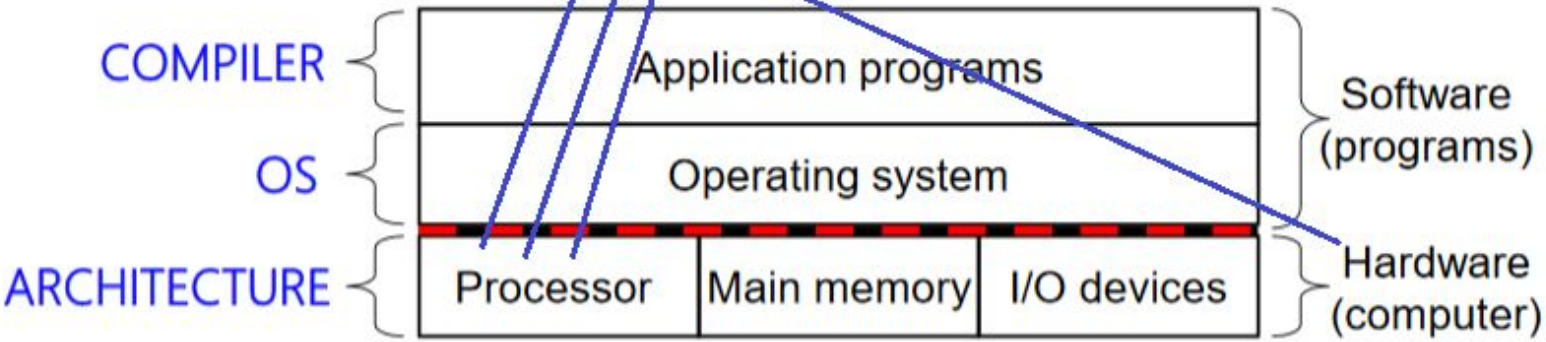
Question 1a. ISA Interface

1. (10 points) **The Big Picture**

   (a) (3 points) For each term below, indicate whether it is above or below the ISA interface. Circle your answer.

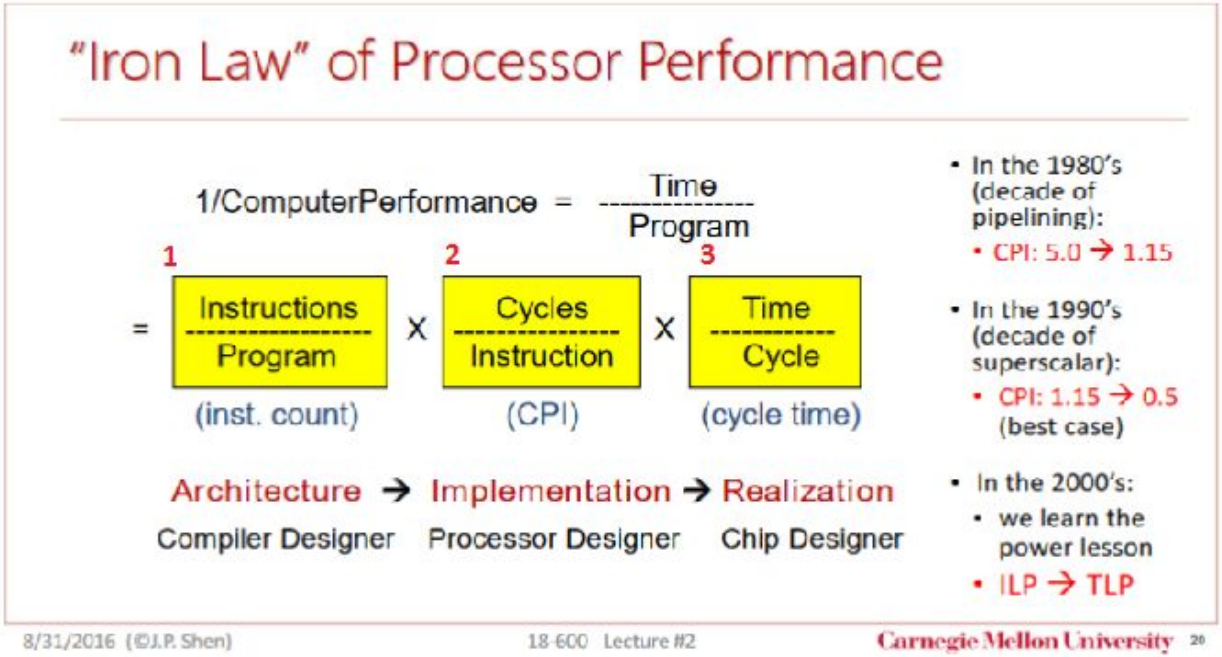
   A. Compiler (above or below)
   B. Cache Memory (above or below)
   C. Processor Pipeline (above or below)
   D. Operating System (above or below)
   E. Amdahl's Law (above or below)
   F. Moore's Law (above or below)

Rubric: each correct option is half point each



Question 1b. Iron Law (Lec. 2 Slide 20)

(b) (3 points) Based on the Iron-Law of processor performance, name the three fundamental ways of improving performance: Rubric: each correct answer is worth 1 point.



Question 1c. Anatomy of Comp Systems (Lec. 1 Slide 21)

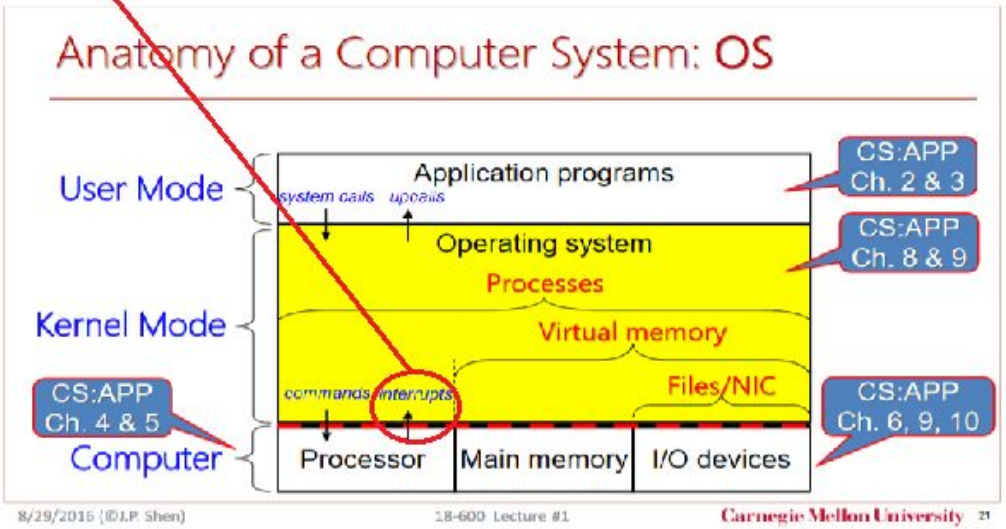
(c) (1 point) Indicate which of the following is CORRECT.

A. Operating system issues system calls to Application Programs.

B. Operating System issues interrupts to the Processor.

C. Processor issues interrupts to the Operating System. (**correct**)

D. I/O Devices issue system calls to the Operating System.



Question 1d. Mem Access Time (Lec. 1 Slide 25)

(d) (1 point) Indicate which of the following is INCORRECT regarding access time.

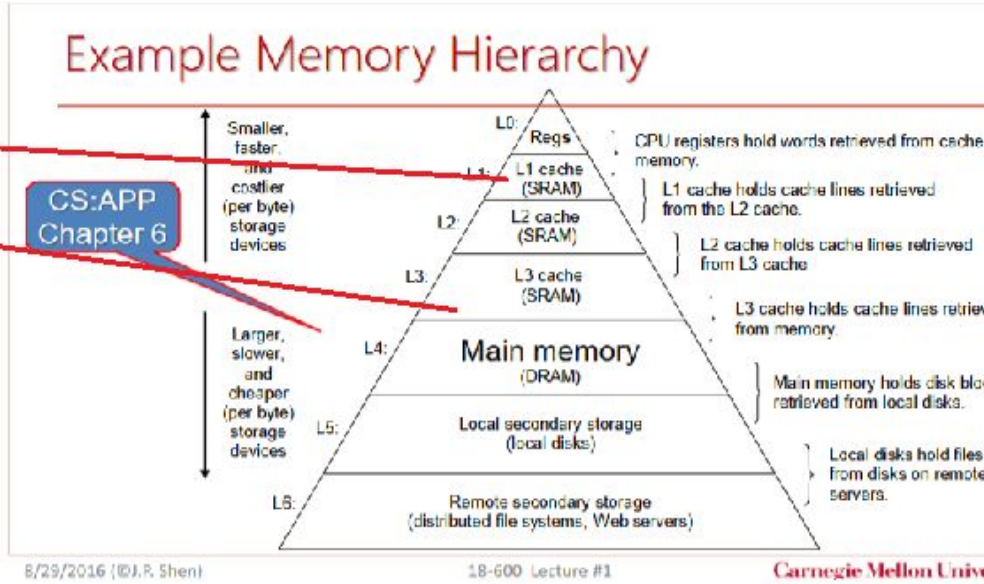
A. L3 cache is faster than L1 cache. (correct)

B. Registers are faster than main memory.

C. Local disks are faster than distributed file systems.

D. SRAM is faster than DRAM.

CS:APP Chapter 6



Question 1e. RISC vs CISC (Lec. 2 Slides 9/10)

(e) (1 point) Indicate which of the following is CORRECT.

A. RISC aims to reduce the number of instructions in a program.

B. CISC aims to reduce the number of instructions in the ISA.

C. RISC aims to reduce the number of cycles per instruction. (**correct**)

D. CISC aims to reduce the number of memory references.

What is RISC?

Reduced Instruction Set Computing

Reduces the cycles per instruction

Question 1f...

(f) (1 point) Which of the following actions constitute as cheating?

A. Looking at someone's code but not copying it.

B. Copying code from StackOverflow or open source projects.

C. Reusing solutions from previous semesters.

D. All of the above. (**correct**)

Question 2 Bits and Bytes

1. An 8-bit machine using two's complement arithmetic for signed integers.
2. Right shifts on signed integers are arithmetic.
3. Right shifts on unsigned integers are logical.
4. `x` and `y` are signed integers, unless otherwise specified.

Expression	Decimal	Binary
$-T_{min}$	-128	0b10000000
`x`	-104	**10011000**
`y`	**63**	0b00111111
`(unsigned)x`	152	0b10011000
`x || 0xdeadbeef`	1	0b00000001
`-x`	104	0b01101000
`x >> 2`	-26	0b11100110
`0x18 & y`	24	0b00011000
`x > y`	0	0b00000000
`((unsigned) x) >> 2`	38	0b00100110

Q3. Floating Point

Grading scheme:

- 1 point for every correct entry. No partial credits.
- Entry regarded incorrect if Rounded Value is simply worded and the value is not written. e.g: largest denorm (except infinity)

3. (10 points) **Floating Point**

Consider the following 5-bit floating point representations based on the IEEE floating point format. This format does not have a sign bit - it can only represent positive floating point numbers.

- There are $k = 3$ exponent bits.
- There are $n = 2$ fraction bits.

Bias = $2^{3-1}-1 = 3$

Below, you are given some decimal values, and your task is to encode them in floating point format. If rounding is necessary, you should use round-to-even, as you did in Data Lab. In addition, you should give the rounded value of the encoded floating point number. Give these as whole numbers (e.g., 17) or as fractions in reduced form (e.g., 3/4).

Value	Floating Point Bits	Rounded Value
9/32	001 00	1/4
3/16		
1		
9		
20		
15/2		

Q3. Floating Point

Value	Floating Point Bits	Rounded Value
9/32	001 00	1/4
3/16	000 11	3/16
1	011 00	1
9	110 00	8
20	111 00	infinity
15/2	110 00	8

When a value requires E= 2^3-1=7 or above, the value becomes infinity and the fractional part must be cleared. NaN is only for values that actually aren't numbers

Q4 Assembly

(a) (12 points) **Jump Table**

The next problem concerns code generated by GCC for a function involving a switch statement. The code uses a jump to index into the jump table:

```
400519: jmpq *0x400640(,%rdi,8)
```

Using GDB, we extract the 8-entry jump table as:

```
0x400640: 0x40052a
0x400648: 0x400529
0x400650: 0x400530
0x400658: 0x400529
0x400660: 0x400530
0x400668: 0x400520
0x400670: 0x400529
0x400678: 0x400535
```

The following block of disassembled code implements the branches of the switch statement:

```
# on entry: %rdi = a, %rsi = b, %rdx = c
    400510: movq $0x2,%rax
    400513: cmp $0x7,%rdi
    400517: ja 400529
    400519: jmpq *0x400640(,%rdi,8)
    400520: movq %rdx,%rax
    400523: addq %rsi,%rax
    400526: salq $0x1,%rax
    400529: retq
    40052a: movq %rsi,%rdx
    40052d: xorq $0xa,%rdx
    400530: leaq 0xa(%rdx),%rax
    400534: retq
    400535: movq $0x8,%rax
    400538: retq
```

(a) (12 points) **Jump Table**

The next problem concerns code generated by GCC for a function involving a switch statement. The code uses a jump to index into the jump table:

```
400519: jmpq *0x400640(,%rdi,8)
```

Using GDB, we extract the 8-entry jump table as:

```
0x400640: 0x40052a   0
0x400648: 0x400529   1
0x400650: 0x400530   2
0x400658: 0x400529   3
0x400660: 0x400530   4
0x400668: 0x400520   5
0x400670: 0x400529   6
0x400678: 0x400535   7
```

The following block of disassembled code implements the branches of the switch statement:

```
# on entry: %rdi = a, %rsi = b, %rdx = c
    400510: movq $0x2,%rax      initialization
    400513: cmp $0x7,%rdi
    400517: ja 400529
    400519: jmpq *0x400640(,%rdi,8)
5   400520: movq %rdx,%rax
    400523: addq %rsi,%rax
    400526: salq $0x1,%rax
    400529: retq
0   40052a: movq %rsi,%rdx
    40052d: xorq $0xa,%rdx
2,4 400530: leaq 0xa(%rdx),%rax
    400534: retq
7   400535: movq $0x8,%rax
    400538: retq
```

```
long test(long a, long b, long c)
{
  long answer = ___2____;
  switch(a)
  {
    case _0__:
      c = __b ^ 10___;
      /* Fall through */
    case _2_(4)_:
    case _4_(2)_:
      answer = _10 + c____;
      break;
    case _5__:
      answer = _(b + c) << 1__;
      break;
    case _7__:
      answer = __8___;
      break; (case 5 and 7 are interchangeable)
    default:
      ;
  }
  return answer;
}
```

Q4 Assembly

(b) (8 points) **Array**

Consider the C code below, where H and J are constants declared with #define.

```
int array1[H][J];
int array2[J][H];

int copy_array(int x, int y) {
    array2[y][x] = array1[x][y];
    return 1;
}
```

Suppose the above C code generates the following x86-64 assembly code:

```
# On entry:
#     %rdi = x
#     %rsi = y
#
copy_array:
    movq %rdi, %rax
    leaq (%rsi,%rsi,8), %rdx
    addq %rdi, %rdx
    salq $5, %rax
    subq %rdi, %rax
    leaq (%rsi,%rax,2), %rax
    movl array1(,%rax,4), %eax
    movl %eax, array2(,%rdx,4)
    movl $1, %eax
    ret
```

What are the values of H and J?

H = 9

J = 62

Q5 Smashing the Stack

5. (16 points) **Smashing the Stack**

After diffusing Dr. Evil's nefarious bombs, you've decided to take it a step farther and take his evil operations down by exploiting security flaws in his codebase. You have located a particular program, evil-prog, and decided to disassemble it and check out the assembly code:

- Main idea testing here is knowledge of the stack layout in 64 bit systems & assembly
- Many students got this question wrong, please refer to the slides, do attack lab & read the book to improve understanding
  - Some confusion between 32 bit & 64 bit systems

Assembly Code

```
00000000004005b0 <nonsense>:
  4005b0:        53                      push   %rbx
  4005b1:        48 89 d3                mov    %rdx,%rbx
  4005b4:        e8 7f fe ff ff          callq  400438 <strcpy@plt>
  4005b9:        48 8b 53 08             mov    0x8(%rbx),%rdx
  4005bd:        48 8b 33                mov    (%rbx),%rsi
  4005c0:        bf fc 06 40 00          mov    $0x4006fc,%edi
  4005c5:        b8 00 00 00 00          mov    $0x0,%eax
  4005ca:        e8 49 fe ff ff          callq  400418 <printf@plt>
  4005cf:        5b                      pop    %rbx
  4005d0:        c3                      retq

00000000004005d1 <main>:
  4005d1:        48 83 ec 28             sub    $0x28,%rsp
  4005d5:        48 c7 44 24 10 05 00    movq   $0x5,0x10(%rsp)
  4005dc:        00 00
  4005de:        48 c7 44 24 18 07 00    movq   $0x7,0x18(%rsp)
  4005e5:        00 00
  4005e7:        48 8b 76 08             mov    0x8(%rsi),%rsi
  4005eb:        48 8d 54 24 10          lea    0x10(%rsp),%rdx
  4005f0:        48 89 e7                mov    %rsp,%rdi
  4005f3:        e8 b8 ff ff ff          callq  4005b0 <nonsense>
  4005f8:        b8 00 00 00 00          mov    $0x0,%eax
  4005fd:        48 83 c4 28             add    $0x28,%rsp
  400601:        c3                      retq
```

This program expects a single argument on the command-line.

Assume that when main begins executing,

`%rsp = 0x100, %rbx = 0xA`

Assume that the memory address 0x4006fc contains the format string

`"0x%x 0x%x"`

Some helpful function definitions are

```
char *strcpy(char *dest, const char *src);
int printf(const char *format, ...);
```

First, lets begin our investigation by running the following command

```
./evil-prog deadbeef
```

Where the hexadecimal byte conversion for "deadbeef" is 64 65 61 64 62 65 65 66

(6 points) Please fill out the following stack diagram for this program execution, representing the layout of the stack right before strcpy is called, address 0x4005b4. Each address represents an 8-byte section of the stack. Please put all answers in hexadecimal format and for any areas of the stack which are unkown, please leave the box blank.

Address	Memory Value
0x100	
0xF8	
0xF0	
0xE8	
0xE0	
0xD8	
0xD0	
0xC8	
0xC0	
0xB8	

Assembly Code

```
00000000004005b0 <nonsense>:
  4005b0:       53                      push   %rbx
  4005b1:       48 89 d3                mov    %rdx,%rbx
  4005b4:       e8 7f fe ff ff          callq  400438 <strcpy@plt>
  4005b9:       48 8b 53 08             mov    0x8(%rbx),%rdx
  4005bd:       48 8b 33                mov    (%rbx),%rsi
  4005c0:       bf fc 06 40 00          mov    $0x4006fc,%edi
  4005c5:       b8 00 00 00 00          mov    $0x0,%eax
  4005ca:       e8 49 fe ff ff          callq  400418 <printf@plt>
  4005cf:       5b                      pop    %rbx
  4005d0:       c3                      retq

00000000004005d1 <main>:
  4005d1:       48 83 ec 28             sub    $0x28,%rsp
  4005d5:       48 c7 44 24 10 05 00    movq   $0x5,0x10(%rsp)
  4005dc:       00 00
  4005de:       48 c7 44 24 18 07 00    movq   $0x7,0x18(%rsp)
  4005e5:       00 00
  4005e7:       48 8b 76 08             mov    0x8(%rsi),%rsi
  4005eb:       48 8d 54 24 10          lea    0x10(%rsp),%rdx
  4005f0:       48 89 e7                mov    %rsp,%rdi
  4005f3:       e8 b8 ff ff ff          callq  4005b0 <nonsense>
  4005f8:       b8 00 00 00 00          mov    $0x0,%eax
  4005fd:       48 83 c4 28             add    $0x28,%rsp
  400601:       c3                      retq
```

This program expects a single argument on the command-line.

Assume that when main begins executing,

```
%rsp = 0x100, %rbx = 0xA
```

Assume that the memory address 0x4006fc contains the format string

```
"0x%x 0x%x"
```

Some helpful function definitions are

```
char *strcpy(char *dest, const char *src);
int printf(const char *format, ...);
```

Address	Memory Value
0x100	
0xF8	
0xF0	0x7
0xE8	0x5
0xE0	
0xD8	
0xD0	0x4005f8
0xC8	0xA
0xC0	
0xB8	

(1 point) What is the security issue with this program called?

Buffer Overflow

(3 points) What would be printed to the console?

0x5 0x7

Some notes:

- Buffer Overflow != Stack Overflow! No partial credit given here
- For the first example, buffer does not overflow! So regular output is printed

Next, we run the following command

```
./evil-prog deadbeefdeadbeefdeadbeef
```

(3 points) What would be printed to the console now?

'0x6665656264616564 0x0' OR '0x6665656264616564 0x7'
(First is correct, both get full credit (null terminator on string))

(3 points) We discover a useful function elsewhere in the program, disable-launch, located at address 0x4142434441424344 ('ABCDABCD' in ascii). What command could we give in order to exploit evil-prog to run the disable-launch code?

./evil-prog deadbeefdeadbeefdeadbeefdeadbeefdeadbeefDCBADCBA
(40 bytes of junk, address bytes in little-endian order)

Some notes:

- Remember little endian format! Review recitation slides, this concept is key
- Were generous when giving points on these, decent efforts were awarded points

Q6. Pipelined Processor Architecture

Grading scheme:

- Part A
  - 1 point for each correct answer. No partial credits
- Part B
  - 2 points for the correct answer. No partial credits
- Part C
  - 1 point for each sub question. No partial credits

6. (10 points) **Pipelined Processor Architecture**

(a) (5 points) Each table below illustrates the pipeline stages of different Y86 instructions. Match each table to the instructions: popq, call, ret, rmmovq, addq. Write your answer in the first row of each table.

Instruction	

Fetch	`icode:ifun = M[PC]`
	`rA:rB = M[PC+1]`
	`valC = M[PC+2]`
	`valP = PC+10`

Decode	`valA = R[rA]` `valB = R[rB]`
Execute	`valE = valB + valC`
Memory	`M[valE] = valA`
Write back	
PC update	`PC = valP`

Instruction	

Fetch	`icode:ifun = M[PC]`
	`rA:rB = M[PC+1]`
	`valP = PC+2`

Decode	`valA = R[rA]` `valB = R[rB]`
Execute	`valE = valB OP valA` `Set CC`
Memory	
Write back	`R[rB] = valE`
PC update	`PC = valP`

Instruction	

Fetch	`icode:ifun = M[PC]`
	`valC = M[PC+1]`
	`valP = PC+2`

Decode	`valA = R[%rsp]` `valB = R[%rsp]`
Execute	`valE = valB + 8`
Memory	`valM = M[valA]`
Write back	`R[%rsp] = valE` `R[rA] = valM`
PC update	`PC = valP`

Instruction	
Fetch	`icode:ifun = M[PC]`
Decode	`valA = R[%rsp]` `valB = R[%rsp]`
Execute	`valE = valB + 8`
Memory	`valM = M[valA]`
Write back	`R[%rsp] = valE`
PC update	`PC = valM`

Instruction	

Fetch	`icode:ifun = M[PC]`
	`valC = M[PC+1]`
	`valP = PC+9`

Decode	`valB = R[%rsp]`
Execute	`valE = valB - 8`
Memory	`M[valE] = valP`
Write back	`R[%rsp] = valE`
PC update	`PC = valC`

Q6. Pipelined Processor Architecture

- Part A
  - Condition codes are set only for arithmetic instructions
  - popq and ret both increment the stack pointer in the execute stage
  - call is the only operation that has to decrement the stack pointer in the execute stage
  - The execute() stage contains an OP code for arithmetic operations
  - The execute stage calculates effective memory address (offset + displacement) for rmmovq or mrmovq instructions

6. (10 points) **Pipelined Processor Architecture**

(a) (5 points) Each table below illustrates the pipeline stages of different Y86 instructions. Match each table to the instructions: popq, call, ret, rmmovq, addq. Write your answer in the first row of each table.

rmmovq

Instruction	
Fetch	icode:ifun = M[PC] rA:rB = M[PC+1] valC = M[PC+2] valP = PC+10
Decode	valA = R[rA] valB = R[rB]
Execute	valE = valB + valC
Memory	M[valE] = valA
Write back	
PC update	PC = valP

Instruction	
Fetch	icode:ifun = M[PC] rA:rB = M[PC+1] valP = PC+2
Decode	valA = R[rA] valB = R[rB]
Execute	valE = valB OP valA Set CC
Memory	
Write back	R[rB] = valE
PC update	PC = valP

popq

Instruction	
Fetch	icode:ifun = M[PC] valC = M[PC+1] valP = PC+2
Decode	valA = R[%rsp] valB = R[%rsp]
Execute	valE = valB + 8
Memory	valM = M[valA]
Write back	R[%rsp] = valE R[rA] = valM
PC update	PC = valP

Instruction	
Fetch	icode:ifun = M[PC]
Decode	valA = R[%rsp] valB = R[%rsp]
Execute	valE = valB + 8
Memory	valM = M[valA]
Write back	R[%rsp] = valE
PC update	PC = valM

addq

call

Instruction	
Fetch	icode:ifun = M[PC] valC = M[PC+1] valP = PC+9
Decode	valB = R[%rsp]
Execute	valE = valB - 8
Memory	M[valE] = valP
Write back	R[%rsp] = valE
PC update	PC = valC

ret

Q6. Pipelined Processor Architecture

- Part B
  - 3 stalls
    - Wait till the write back stage of popq passes back the results to the decode stage of add
- Part C
  - Memory
    - Pop loads from the memory into a register
    - All loads and stores happen during memory stage
  - Execute
    - Arithmetic operations happen in the execute stage
  - Yes, 1
    - Memory and Execute are adjacent stages
    - Forwarding between adjacent stages requires 1 stall

(b) (2 points) Assume the following sequence of instructions in the pipeline:

1. pop r2
2. add r1, r2

How many stalls are required to be inserted after the pop instruction to avoid a data hazard ?

3 stalls

(c) (3 points) We can avoid some of the stalls above by forwarding. Let us see how we can do this. Assume the stages of a pipeline are represented by IF, ID, EX, MEM, WB. The answer to each of the first 2 questions below should be one of these stages.

- Which is the earliest stage of the pop instruction when the result becomes available in r2?

Memory

- Which is the latest stage by which the value of r2 is needed by add?

Execute

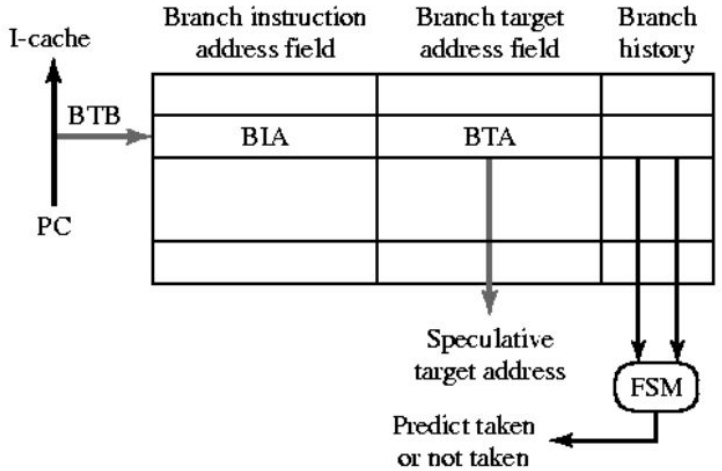
- If we forward between the stages that you mention above, do we still need stalls between pop and add to avoid a data hazard? If so, how many?

Yes, 1

Problem 7a: Branch Target Buffer & Branch History Table

Compare: Both used in dynamic branch prediction.

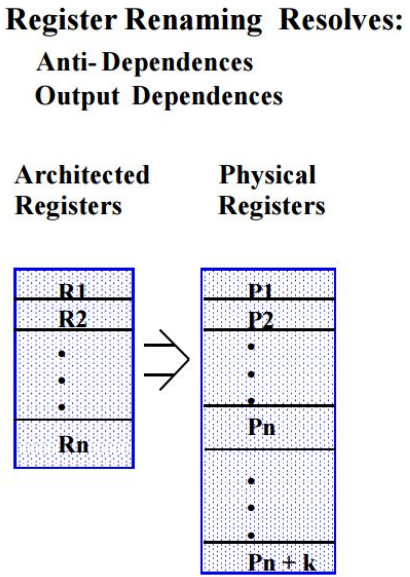
Contrast: BHT stores direction history, and used to predict the direction of branch instruction. BTB stores history of branch targets, and is used to predict the target of a branch instruction.



Problem 7b: Register Allocation & Register Renaming

Compare: Both deal with mapping of registers.

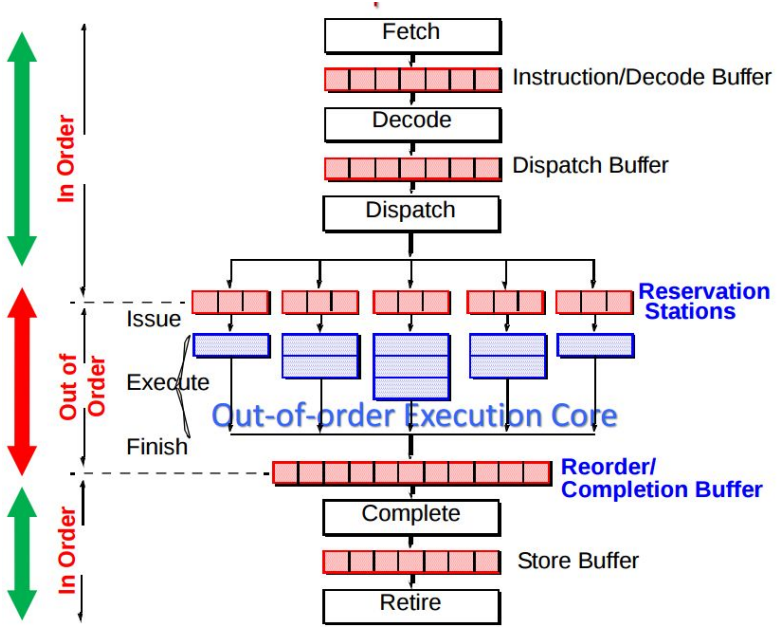
Contrast: Register allocation maps virtual to architecture registers; done by compiler at compile time. Register renaming maps architecture registers to physical registers; done by hardware at run time.



Problem 7c: Reservation Station & Reorder Buffer

Compare: Both form the boundaries (front and back) of Out-of-order Execution core.

Contrast: Reservation Station receives in-order instructions and outputs out-of-order instructions. Reorder Buffer receives out-of-order instructions and outputs in-order instructions.

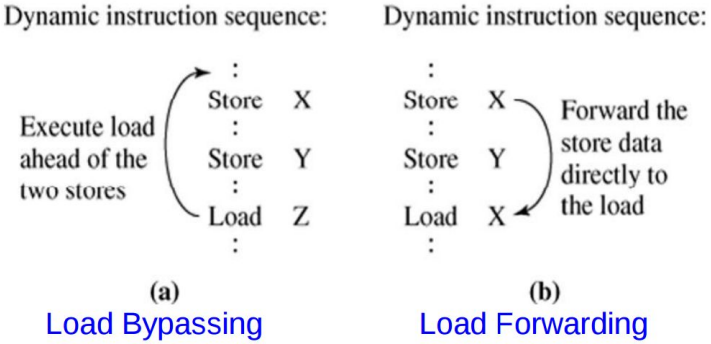


Problem 7d: Load Bypassing and Load Forwarding

Compare: Both try to accelerate the execution of load instructions.

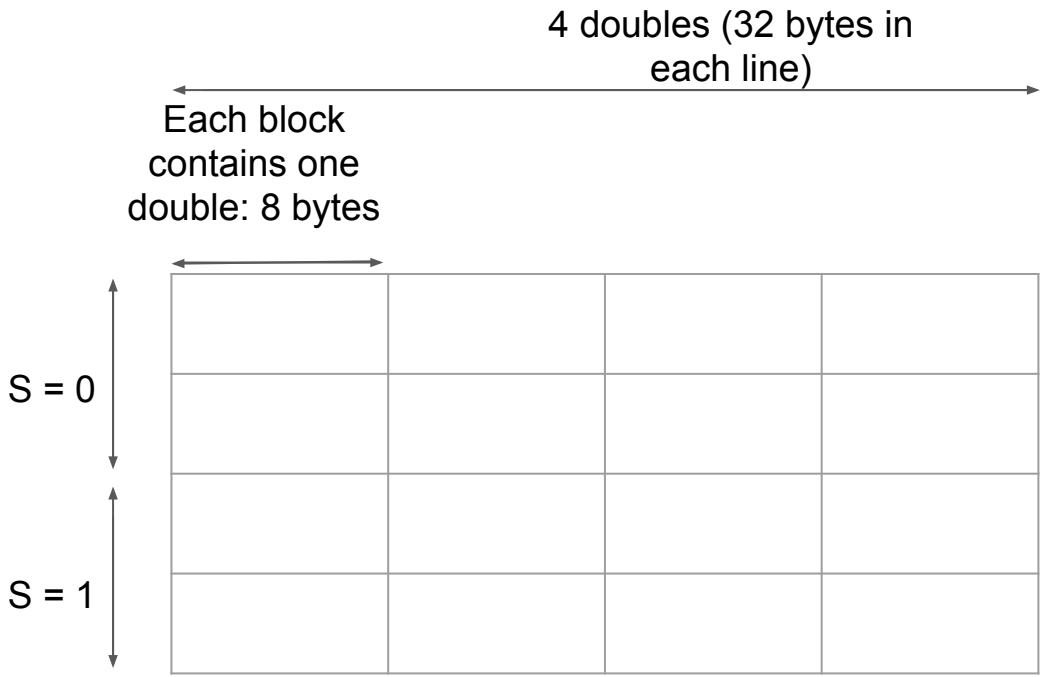
Contrast: LB allows load instructions to execute earlier than a store instruction that precedes the load, if determined that the load will not alias with store.

LF allows a load to get the data directly from the preceding store if aliasing exists.



Problem 8

- 128 byte data cache
- 2-way associative
- 4 doubles in each line
- Each double is 8 bytes
- A is cache aligned



- Problem 8a:
  - Since there are 4 cache lines and each can hold 4 doubles, in total the cache can hold **16** doubles
- Problem 8b:
  - There are 4 cache lines and it's a 2-way associative cache, so there are **2** sets.

Problem 8c (m = 1)

```
double A[32], t = 0;
for(int i = 0; i < m; i++)
  for(int j = 0; j < n; j++)
    t += A[j * m + i];
```

Since m = 1 and m*n = 32

```
for(int i = 0; i < 1; i++)
  for(int j = 0; j < 32; j++)
    t += A[j + i];
```

A[0]	A[1]	A[2]	A[3]
A[8]	A[9]	A[10]	A[11]
A[4]	A[5]	A[6]	A[7]
A[12]	A[13]	A[14]	A[15]

- From the code, you see that we're loading A[0] to A[31] in that order

- A[0] - cold miss, hits for A[1] to A[3] in S = 0
  A[4] - cold miss, hits for A[5] to A[7] in S = 1
  A[8] - cold miss, hits for A[8] to A[11] in S = 0 and so on
- For every miss there are 3 hits.
- And there are no conflict misses, only cold misses

Answers for problem 8c

A. Miss rate = ¼
B. Kinds of misses = Cold or compulsory misses. And optionally capacity misses.
C. Kind of locality? Spatial locality as it's a stride-1 access of the elements from A[0] to A[31]

Problem 8d (m = 2)

```
double A[32], t = 0;
for(int i = 0; i < m; i++)
  for(int j = 0; j < n; j++)
    t += A[j * m + i];
```

Since m = 2 and m*n = 32

```
for(int i = 0; i < 2; i++)
  for(int j = 0; j < 16; j++)
    t += A[2*j + i];
```

- From the code, you see that we're loading A[0], A[2], A[4]... A[30] followed by A[1], A[3], A[5]... A[31]

A[0]	A[1]	A[2]	A[3]
A[8]	A[9]	A[10]	A[11]
A[4]	A[5]	A[6]	A[7]
A[12]	A[13]	A[14]	A[15]

- A[0] - cold miss, hit for A[2] in S = 0
  A[4] - cold miss, hit for A[6] in S = 1
  A[8] - cold miss, hit for A[10] in S = 0
  A[12] - cold miss, hit for A[14] in S = 1

Problem 8d (m = 2)

A[16]	A[17]	A[18]	A[19]
A[24]	A[25]	A[26]	A[27]
A[20]	A[21]	A[22]	A[23]
A[28]	A[29]	A[30]	A[31]

1. A[16] - miss & eviction, hit for A[18] in S = 0
   A[20] - miss & eviction, hit for A[22] in S = 1
   A[24] - miss & eviction, hit for A[26] in S = 0
   A[28] - miss & eviction, hit for A[30] in S = 1

2. After A[28], we need to load A[1] and this maps to the first line of the cache at S=0 and is a conflict miss.

A[1] - conflict miss, hit for A[3] in S = 0
A[5] - conflict miss, hit for A[7] in S = 1
A[9] - conflict miss, hit for A[11] in S = 0
A[13] - conflict miss, hit for A[15] in S = 1 and so on

3. Therefore, for every hit there is a miss. And there are both cold and conflict misses.

Answers for problem 8d

A. Miss rate = **1/2**
B. Kinds of misses = Cold or compulsory misses and conflict misses

Problem 8e (m = 16)

```
double A[32], t = 0;
for(int i = 0; i < m; i++)
  for(int j = 0; j < n; j++)
    t += A[j * m + i];
```

Since m = 16 and m*n = 32

```
for(int i = 0; i < 16; i++)
  for(int j = 0; j < 2; j++)
    t += A[16*j + i];
```

1. From the code, you see that we're loading A[0], A[16], A[1], A[17], A[2], A[18], A[3], A[19]...

S = 0

S = 1

A[0]	A[1]	A[2]	A[3]
A[16]	A[17]	A[18]	A[19]
A[4]	A[5]	A[6]	A[7]
A[20]	A[21]	A[22]	A[23]

2. A[0] is a cold miss and A[1], A[2] and A[3] are hits.
3. Since it's two way set associative, A[16] gets mapped to the second line of the first set (S = 1) and A[20] gets mapped to the second line of the second set (S = 2)

Answers to problem 8e

Therefore, for every miss in the cache, there are three hits

A. Miss rate = ¼
B. Kinds of misses = Cold or compulsory misses.

IMPORTANT: There was a mistake in the rubric where the correct answer was
A. Miss rate = 1
B. Kinds of misses = Cold or compulsory misses and conflict misses.
This is wrong.

- If you've answered A with ¼, you get 2 points
- And if you've answered B with only cold miss you get .5 points more (should have been awarded .5 points already)