Quizzes

- **Quiz 1**
  - What is the decimal form of the unsigned binary number 110111?
    - 55
  - What is the decimal form of the two's complement binary number 110111?
    - -9
  - What is the unsigned binary form of the decimal number 17? Use only as many bits as necessary.
    - 10001
  - Which of the following best represents the conversion from the unsigned binary number 11001 to its decimal equivalent, 25?
    - 16 + 8 + 0 + 0 + 1
  - Which of the following numbers could represent 0 in two's complement binary?
    - 0000
  - Which of the following numbers could represent 0 in sign-magnitude binary?
    - 0000
    - 1000
  - What is the binary equivalent of the 0x82e0? Answer with a 16-bit binary number.
    - 1000 0010 1110 0000
  - What is the hex equivalent of the binary number 0110 1110?
    - 0x6e
- **Quiz 2**
  - What is the opcode in the 32-bit ARM instruction 1110000 0100 0 0010 0000 00000000 0001?
    - 0100
  - For the instruction set from lecture on October 4, which of the below is the best English description of the instruction 00 01 10 10?
    - Add r2 to r2 and store the result in r1
  - There is an instruction in ARM that subtracts a number from a register directly (e.g. using an immediate value), without requiring the value to be stored in a register first.
    - TRUE
  - The opcode in this instruction corresponds to ADD: 1110000 0010 0 0010 0000 00000000 0001
    - FALSE
- **Quiz 3**
  - Assume the decimal value 4 is in r0 and the decimal value 5 is in r1, and the instruction "subs r0, r0, r1" is executed. What will the four status bits in N Z C V be set to?
    - N = 1
    - Z = 0
    - C = 0
    - V = 0
  - Assume the decimal value 4 is in r0 and the decimal value -1 is in r1. Which of the following instructions would not set the N bit to 1?
    - Adds r0, r0, r1

    - Which of these instructions will not have any effect if executed when the status bits are set to N = 0, Z = 1, C = 1, V = 1
        - Movne r0, #45
    - Which instruction suffix corresponds to the condition code for running an instruction no matter what the status bits are set to?
        - AL
- **Quiz 4**
    - Which register is the program counter in ARM?
        - R15
    - How many times will these four instructions update the status register if they run in order?

      ```
              mov r2, #1
              mul r2, r2, r1
              subs r1, #1
              subne r3, #8
      ```
        - 1 time
    - A label always turns into an instruction that executes in the generated machine code
        - FALSE

    - If the value 0x00000004 is in r2 when this program starts, what value will be in r1 when it completes?

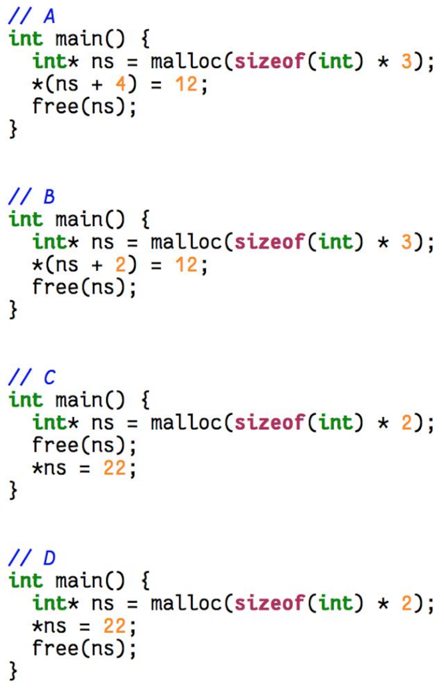
      ```
              mov r1, #0
      start:
              lsls r2, #1
              bcs end
              add r1, #1
              b start
      end:
      ```
        - 29

- **Quiz 5**
    - What is the decimal representation of the signed-magnitude binary number 1100 0100?
        - -68
    - What is the result of adding 0x10F and 0x214?
        - 0x323
    - What is the I (capital i) bit usually for in data-processing instructions in ARM?
        - It specifies whether the second argument will be an immediate value or a register
    - The machine encoding of a branch instruction can express changing the program counter to any 32-bit address
        - FALSE
    - Which of the following instructions set condition flags, but don't change the values in any registers?
        - cmp r1, r2
    - Which of the following condition mnemonics means "The overflow bit AND the zero bit are set?"
        - NE
        - PL
        - GE

        - HS
        - **None of the above**
    - The difference in the machine instructions for “sub r1, r2, r3” and “subne, r3, r2, r3” is in…
        - Bits 15:12 (the Rd part)
        - Bits 31:28 (the cond part)
    - If logically shifted left by THREE bits using the “lsls” instruction, which of the following examples would set the carry bit?
        - 0x20000001
        - 0x20000002
        - ~~0x80000000~~ (Actually, this one does not) (Why?) When you lsls once, the carry bit will be set, but the next two times you lsls you are shifting 0x00000000 so it will not set the carry bit
        - https://piazza.com/class/j7w9htrxqm44j5?cid=1950
- **Quiz 6**
    - Push {lr} is equivalent to…
        - Sub sp, #4 followed by str lr, [sp]
    - Push {r14, pc} and push {lr, pc} behave exactly the same
        - True
    - Push {r1, r2} and push {r2, r1} behave exactly the same
        - True
    - A function call always returns to the same address, no matter where it’s called from
        - False
    - Imagine that a program just started running on a raspberry pi. Which register holds an address close to where the command line arguments are stored?
        - Sp Can someone explain this please? https://piazza.com/class/j7w9htrxqm44j5?cid=1371
        - Ah ok thank you -- you’re welcome
- **Quiz 7**
    - What register does ARM use for the return value?
        - R0
    - If we write a function call “f(34,56,58)” in C, which register will contain the number 56?
        - R1
    - A function always needs to save the arguments to the stack as local variables in order to execute correctly.
        - False
    - How is “bl” different from “b”?
        - It saves the return address to the link register before branching.

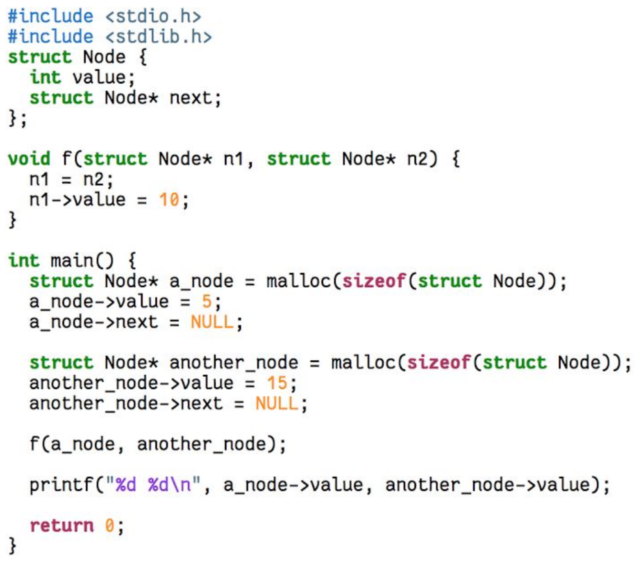
Quiz 8 is course feedback

- **Quiz 9**
  - Which of these programs uses memory in an unsafe way?



```
// A
int main() {
  int* ns = malloc(sizeof(int) * 3);
  *(ns + 4) = 12;
  free(ns);
}


// B
int main() {
  int* ns = malloc(sizeof(int) * 3);
  *(ns + 2) = 12;
  free(ns);
}


// C
int main() {
  int* ns = malloc(sizeof(int) * 2);
  free(ns);
  *ns = 22;
}


// D
int main() {
  int* ns = malloc(sizeof(int) * 2);
  *ns = 22;
  free(ns);
}
```

    - A, C
  - This program prints “5 10”. Which description best describes why?



```
#include <stdio.h>
#include <stdlib.h>
struct Node {
  int value;
  struct Node* next;
};

void f(struct Node* n1, struct Node* n2) {
  n1 = n2;
  n1->value = 10;
}

int main() {
  struct Node* a_node = malloc(sizeof(struct Node));
  a_node->value = 5;
  a_node->next = NULL;

  struct Node* another_node = malloc(sizeof(struct Node));
  another_node->value = 15;
  another_node->next = NULL;

  f(a_node, another_node);

  printf("%d %d\n", a_node->value, another_node->value);

  return 0;
}
```

    - Because the line "n1->value = 10" changes the memory referenced by "another_node", while leaving the memory referenced by "a_node" unchanged
  - Memory used by a pointer must be freed at the end of every function that uses it.
    - False
  - One way to write a program with no memory leaks is to ensure that the program calls “free” on each pointer (address) returned from “malloc” exactly one.

        ■ True

- **Quiz 10**
    - What does this program print? It prints 10

      ```c
      #include <stdlib.h>
      #include <stdio.h>

      void f(int* n) {
        n[1] = 10;
      }

      int main() {
        int* ns = malloc(sizeof(int) * 2);
        ns[0] = 0;
        ns[1] = 1;
        f(ns);
        printf("%d", ns[1]);
      }
      ```

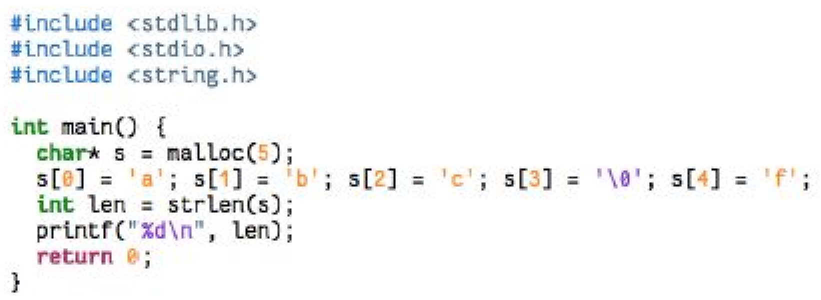
    - What does this program print? It prints 1

      ```c
      #include <stdlib.h>
      #include <stdio.h>

      void f(int* n) {
        n = malloc(sizeof(int) * 2);
        n[1] = 10;
      }

      int main() {
        int* ns = malloc(sizeof(int) * 2);
        ns[0] = 0;
        ns[1] = 1;
        f(ns);
        printf("%d", ns[1]);
      }
      ```

    - How many bytes does the call to free relinquish back to the memory management system? It prints 6

      ```c
      #include <stdlib.h>
      #include <stdio.h>

      char* f() {
        char* m = malloc(sizeof(char) * 10);
        char* n = malloc(sizeof(char) * 6);
        m = n;
        return m;
      }

      int main() {
        char* o = f();
        free(o);
      }
      ```

    - What will this program print?



```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main() {
  char* s = malloc(5);
  s[0] = 'a'; s[1] = 'b'; s[2] = 'c'; s[3] = '\0'; s[4] = 'f';
  int len = strlen(s);
  printf("%d\n", len);
  return 0;
}
```

    - 3

- How many bytes are needed to store the contents of struct C?

```
struct S {
        int v;
        char* c;
        int* w;
 };

struct C {
        struct S* s;
};
```

    - 4 (Bytes to store contents of struct S alone would be 12 bytes)

- Based on the calling convention of functions in ARM, PUSH {lr} at the start of a function should be paired with ___ at the end of said function.
    - POP {pc}
- The push instruction changes the value of sp
    - True
- The STR instruction changes the value of sp
    - False
- The BL instruction changes the value of sp
    - False
- In the compiled output of the C expression "x->a = v", the part of the output that does the real work of assigning the "a" field of the struct will be
    - A STR instruction
- Which of these is a good rule for thinking about malloc, free, and avoiding memory leaks?
    - For each time malloc is called in the running program, there should at some point be a later corresponding call to free for that address
- Temporary values that a function uses and "remembers" while calling other functions can be stored in
    - r4
    - r5
    - Memory addresses that it makes space for by changing the value of sp

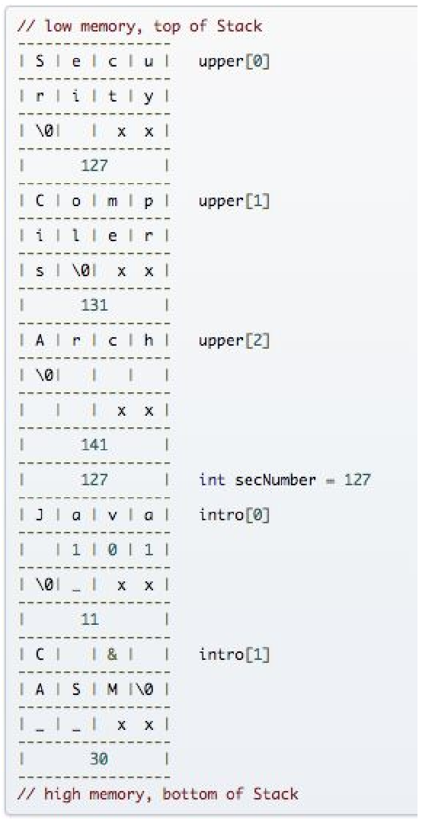
- **Which of the following things is a function responsible for doing before returning?**
    - Restoring the values of r4-r11 that were present before the function began running
    - Restoring the value of sp that was present before the function began running
    - Putting the return value in r0

- Which of the following actions most directly corresponds to the act of returning from a function?
    - Changing the value of pc

- **Quiz 11**
    - This quiz focuses on the program below:

```
struct Course {
        char name[10];
        int number;
};

int main() {
        struct Course intro[] = { { "Java 101", 11}, {"C & ASM", 30} };
        int secNumber = 127;
        struct Course upper[] = { { "Security", secNumber}, {"Compilers",
131}, {"Arch", 141} };
}
```

Stack layout:

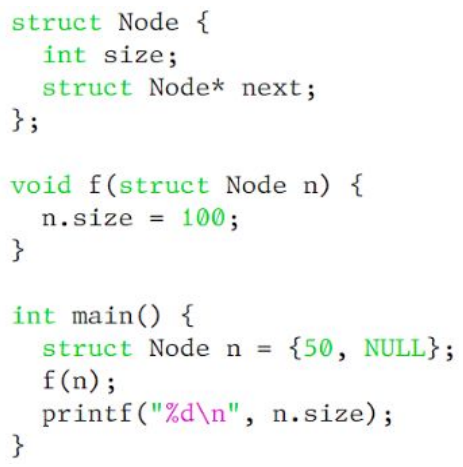


```
// low memory, top of Stack
-------------------
| S | e | c | u |    upper[0]
-------------------
| r | i | t | y |
-------------------
| \0|   |  x  x |
-------------------
|      127      |
-------------------
| C | o | m | p |    upper[1]
-------------------
| i | l | e | r |
-------------------
| s | \0|  x  x |
-------------------
|      131      |
-------------------
| A | r | c | h |    upper[2]
-------------------
| \0|   |   |   |
-------------------
|   |   |  x  x |
-------------------
|      141      |
-------------------
|      127      |    int secNumber = 127
-------------------
| J | a | v | a |    intro[0]
-------------------
|   | 1 | 0 | 1 |
-------------------
| \0| _ |  x  x |
-------------------
|      11       |
-------------------
| C |   | & |   |    intro[1]
-------------------
| A | S | M |\0 |
-------------------
| _ | _ |  x  x |
-------------------
|       30      |
-------------------
// high memory, bottom of Stack
```

Note: Remember that the array elements get laid out from low memory to high memory (top to bottom on the stack), so array element [0] is in lower memory (higher on the stack) than array element [1], which in turn is in lower memory/higher on the stack than [2].

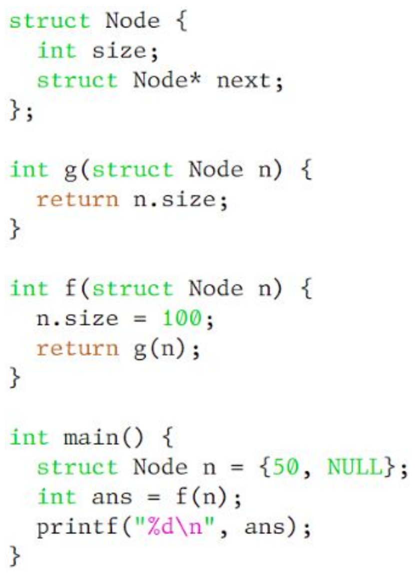
- How many total bytes of **data** are in a Course struct?
  - 14
- What is sizeof(struct Course)
  - 16
- Will the string “Compilers” be at a lower-numbered address or a higher-numbered address that the string “Arch”
  - Lower
- Will the string “Compilers” be at a lower-numbered address or a higher-numbered address than the string “C & ASM”?
  - Lower (https://piazza.coxm/class/j7w9htrxqm44j5?cid=1561)
- How many times will the value 30 be stored on the stack for this main function?
  - 1
- How many times will the value 127 be stored on the stack for this main function?
  - 2
- If we overwrote the string “Java 101” with a string of length 20, which other values would be at least partially overwritten?
  - 11
  - C & ASM

- **Quiz 12**
  - This program below prints **50**.



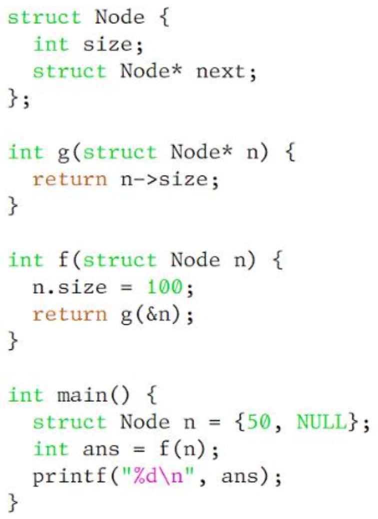
```
struct Node {
  int size;
  struct Node* next;
};

void f(struct Node n) {
  n.size = 100;
}

int main() {
  struct Node n = {50, NULL};
  f(n);
  printf("%d\n", n.size);
}
```

  - This program helow prints **100**



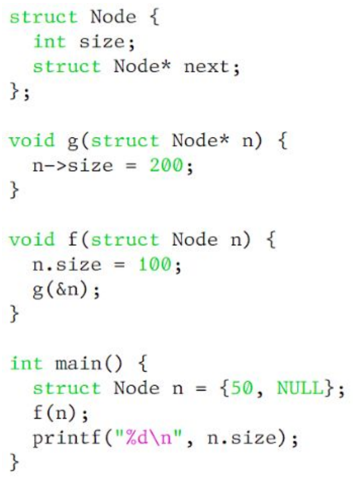
```
struct Node {
  int size;
  struct Node* next;
};

int g(struct Node n) {
  return n.size;
}

int f(struct Node n) {
  n.size = 100;
  return g(n);
}

int main() {
  struct Node n = {50, NULL};
  int ans = f(n);
  printf("%d\n", ans);
}
```

- This program below prints **100**



```
struct Node {
  int size;
  struct Node* next;
};

int g(struct Node* n) {
  return n->size;
}

int f(struct Node n) {
  n.size = 100;
  return g(&n);
}

int main() {
  struct Node n = {50, NULL};
  int ans = f(n);
  printf("%d\n", ans);
}
```

- This program below prints **50**



```
struct Node {
  int size;
  struct Node* next;
};

void g(struct Node* n) {
  n->size = 200;
}

void f(struct Node n) {
  n.size = 100;
  g(&n);
}

int main() {
  struct Node n = {50, NULL};
  f(n);
  printf("%d\n", n.size);
}
```

\

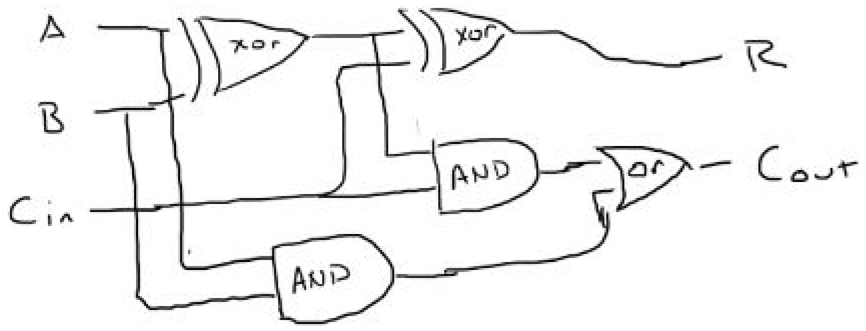
- This program below prints **100**

  ```
  struct Node {
    int size;
    struct Node* next;
  };


  void f(struct Node n[]) {
    n[0].size = 100;
  }

  int main() {
    struct Node n[] = {{50, NULL}, {100, NULL}};
    f(n);
    printf("%d\n", n[0].size);
  }
  ```
- This program below prints **3**

  ```
  struct Node {
    int sizes[4];
    struct Node* next;
  };


  void f(struct Node n) {
    n.sizes[0] = 100;
  }

  int main() {
    struct Node n = {{3, 4, 5, 6}, NULL};
    f(n);
    printf("%d\n", n.sizes[0]);
  }
  ```

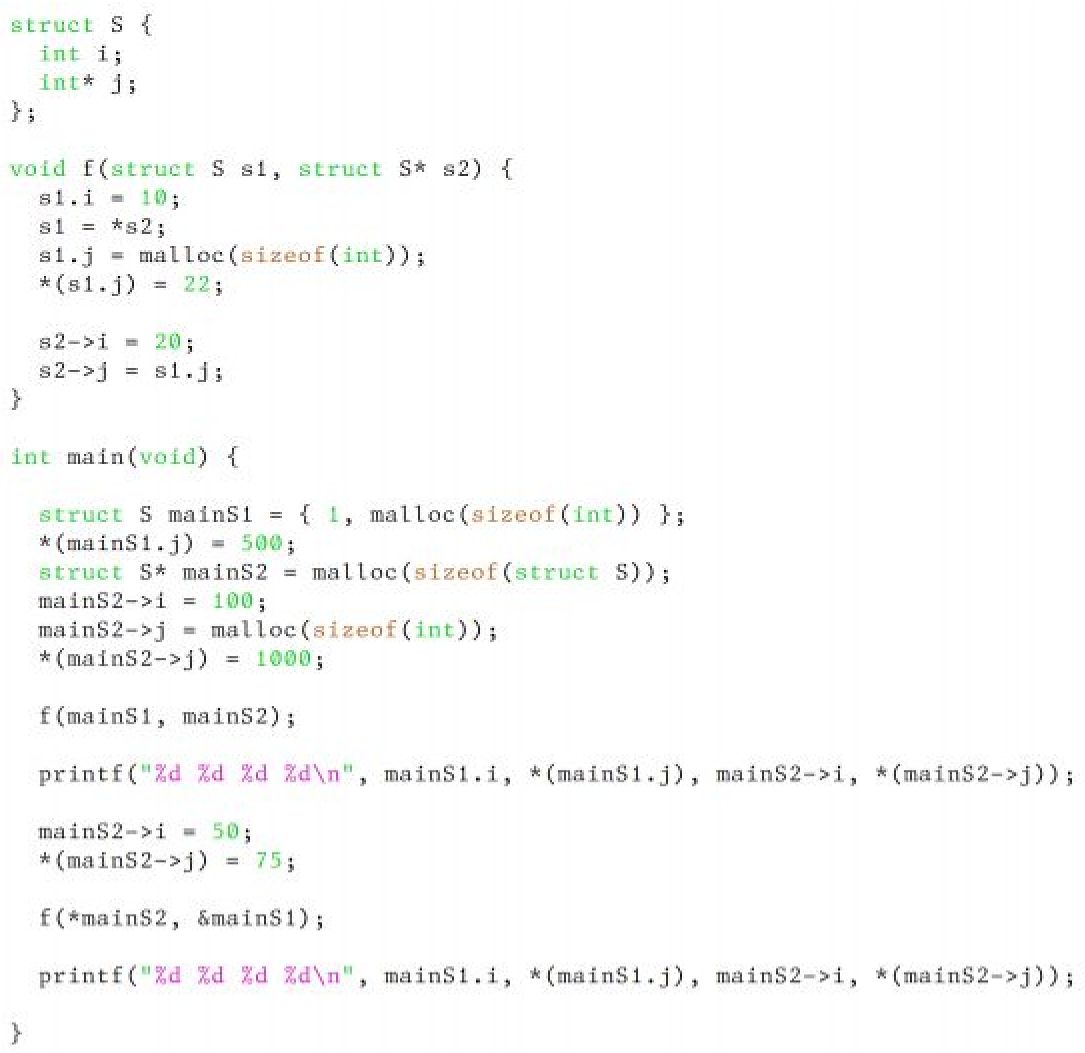
- **Quiz 13**
  - Differences between malloc and calloc
    - Calloc 0-initializes the memory it allocates, whereas malloc doesn't
    - Calloc takes a different number of arguments than malloc
  - Differences between realloc and calloc
    - Calloc 0-initializes the memory it allocates, whereas realloc doesn't
    - Realloc may succeed, but not allocate any new memory, while calloc either allocates new memory or fails
  - In the slow_malloc implementation provided, would a function like consolidate(), which merges free blocks, (potentially) give us extra free space to use?
    - Yes, we could save on the metadata words used to track the free block sizes by merging them
  - What is the worst-case running time of slow_malloc?
    - O(# of total entries)
  - What is the worst-case running time of a malloc that uses a linked free list?
    - O(# of free entries)

- **Quiz 14**
    - 0x4121999a, interpreted as a 32-bit single precision floating point number, is closest to representing what decimal number? For considering ordering, don't rearrange bytes to match endianness
        - 0(100 0001 0)(010 0001 1001 1001 1001 1010)
        - S = 0
        - E = 130 - 127 = 3 -> 2^3
        - M ≈0.26
        - 8*1.26 =10.1
    - What is the hexadecimal representation of the 32-bit floating point representation of decimal 1? 0x3f800000
    - Consider this circuit.



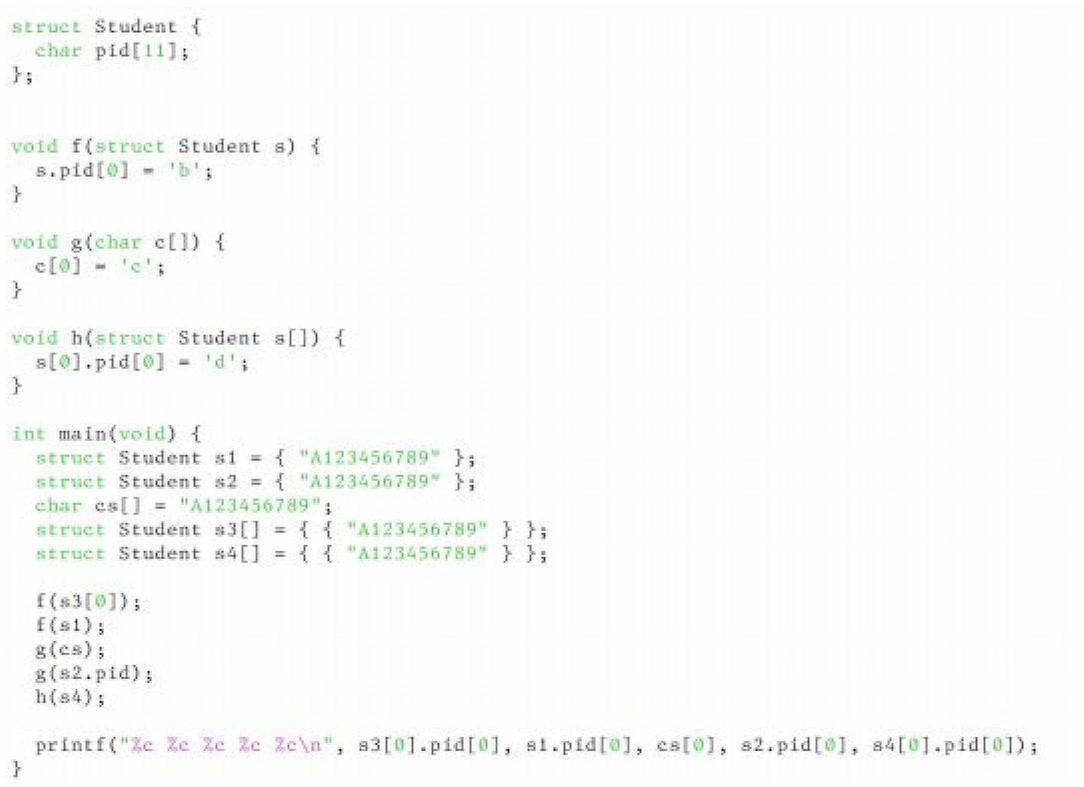
    - If A and B are 1, and Cin is 0, what are the R and Cout outputs?
        - R = 0, Cout = 1
    - If Cout is 1, which of the following could NOT be the inputs?
        - **A = 0, B = 0, C = 1**
        - A = 1, B = 0, C = 0
        - A = 1, B = 0, C = 1
        - A = 0, B = 1, C = 1

- **+Quiz 15**
    - What is the base-10 representation of the binary decimal number 10.11 (note the decimal point)?
        - 2.75
    - True or false: By changing the mantissa of a 32-bit single-precision floating point number, the entire number can be changed from representing a positive number to a negative one.
        - False
    - True or false: The number 4.125 can be exactly represented as a 32-bit single-precision floating point number with no rounding or loss of precision.
        - True
    - If strlen(s) produces 14, and s is a stack-allocated array of char, then s[14] must evaluate to what? (You can assume no undefined behavior happened in strlen())
        - \0
    - Consider this program for the next two questions



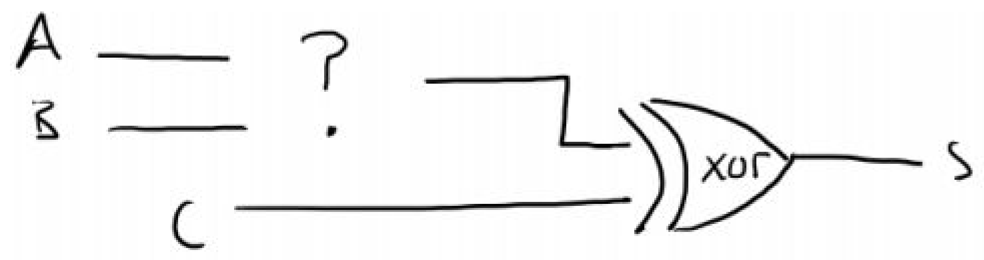
```
struct S {
  int i;
  int* j;
};

void f(struct S s1, struct S* s2) {
  s1.i = 10;
  s1 = *s2;
  s1.j = malloc(sizeof(int));
  *(s1.j) = 22;

  s2->i = 20;
  s2->j = s1.j;
}

int main(void) {

  struct S mainS1 = { 1, malloc(sizeof(int)) };
  *(mainS1.j) = 500;
  struct S* mainS2 = malloc(sizeof(struct S));
  mainS2->i = 100;
  mainS2->j = malloc(sizeof(int));
  *(mainS2->j) = 1000;

  f(mainS1, mainS2);

  printf("%d %d %d %d\n", mainS1.i, *(mainS1.j), mainS2->i, *(mainS2->j));

  mainS2->i = 50;
  *(mainS2->j) = 75;

  f(*mainS2, &mainS1);

  printf("%d %d %d %d\n", mainS1.i, *(mainS1.j), mainS2->i, *(mainS2->j));

}
```

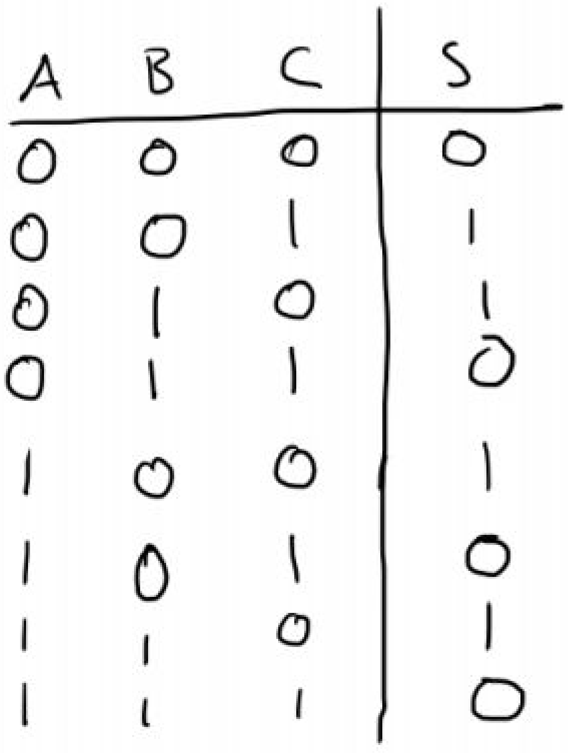
- What is printed by the FIRST printf?
  - 1 500 20 22
- How many bytes are allocated on the HEAP by this program (assume ints are 4 bytes and addresses are 4 bytes).
  - 24 /// **why is this not 20? because size of struct is 8 not 4 (Because you call f twice)**
- What is printed by the SECOND printf?
  - 20 22 50 75 (HOW DOES THIS WORK????)
    - mainS1 is changed by f because it is passed in as a pointer (note the ampersand)
- What does this program print?



```c
struct Student {
  char pid[11];
};


void f(struct Student s) {
  s.pid[0] = 'b';
}

void g(char c[]) {
  c[0] = 'c';
}

void h(struct Student s[]) {
  s[0].pid[0] = 'd';
}

int main(void) {
  struct Student s1 = { "A123456789" };
  struct Student s2 = { "A123456789" };
  char cs[] = "A123456789";
  struct Student s3[] = { { "A123456789" } };
  struct Student s4[] = { { "A123456789" } };

  f(s3[0]);
  f(s1);
  g(cs);
  g(s2.pid);
  h(s4);

  printf("%c %c %c %c %c\n", s3[0].pid[0], s1.pid[0], cs[0], s2.pid[0], s4[0].pid[0]);
}
```

        - A A c c d
    - Assume the caching strategy in figure 8.5 in the Harris textbook, where bits 2-4 are used as the index (same as in lecture). Which of the following addresses has index 101?
        - 0xffffff34
    - True or false: with direct mapping caching, two different addresses with the same tag can be assigned to the same cache entry.
        - True
    - True or false: virtual memory can be used to simulate memory space larger than what's available in physical memory
        - True
    - True or false: the address space of virtual memory addresses must be EQUAL TO OR SMALLER than the address space of physical addresses in order for virtual memory to work.
        - False
    - In our lecture on virtual memory, we u sed 4kb pages, which used 20 bits for the page address and 12 bits for the offset into the page. If we instead had 1MB pages, how many bits would be needed for the offset?
        - 20
        - Because 2^{12} = 4Kb and 2^{20} = 1MB
    - Consider this combinatorial logic circuit and truth table. What kind of gate, filled in for the ? would produce the given truth table?

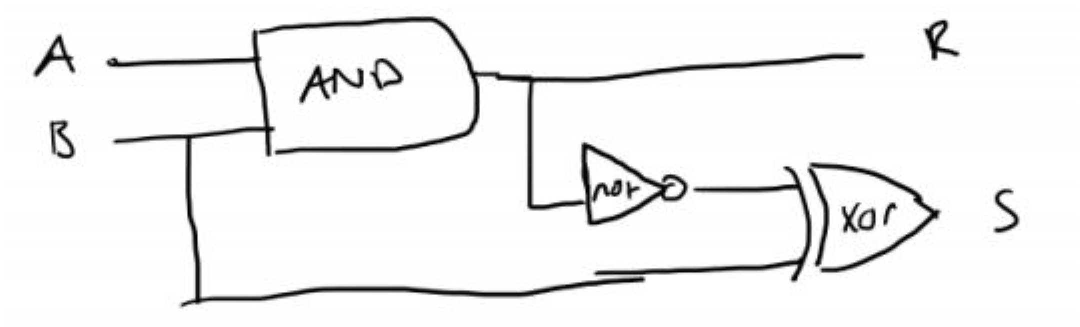


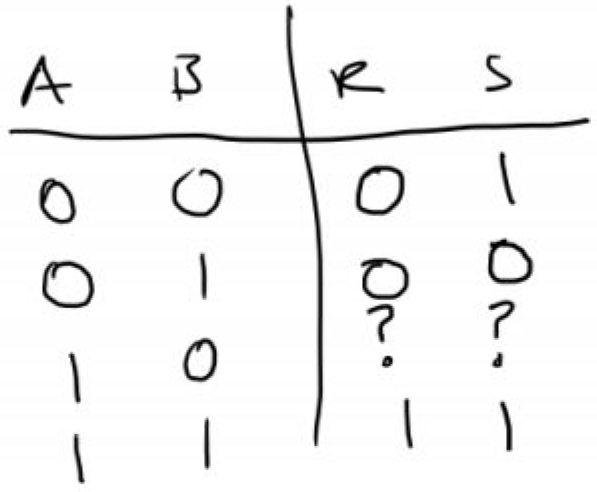


A	B	C	S
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

- Or

○ Fill in the missing row in the truth table for the combinatorial logic circuit below.





A	B	R	S
0	0	0	1
0	1	0	0
1	0	?	?
1	1	1	1

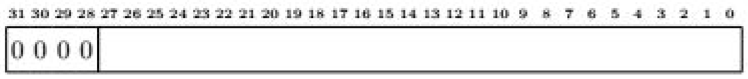
        - 0 1

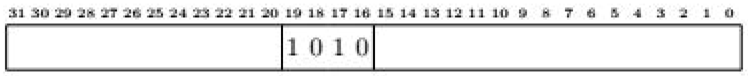
- **Midterm 1**

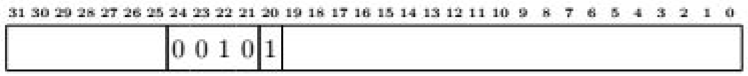
  (The differences of each versions are simply the order of questions/answers)
  (The version shown here: code)
    - How many representations of 0 are possible in the signequizd 2's complement binary representation of numbers?
        - 1 (0000)
    - How many representations of 0 are possible in the signed-magnitude representation of numbers?
        - 2 (1000/0000)
    - For each of the following 8-bit, signed, two's complement binary numbers, give the corresponding decimal representation, including the negative sign if necessary.
    - [1111 1001 / 1111 1111 / 0010 0001]
        - -7/-1/33
    - For each of the following decimal numbers, convert them to their unsigned hexadecimal representation, using as many digits as necessary. Write your answers directly in the answer sheet.
    - [17 / 168 / 512]
        - 0x11/0xA8/0x200
    - Consider these instructions:

A: subne r10, r10, #4
B: subeq r2, r5, r3
C: subs r10, r2, r10
D: adds r2, r10, r3
E: moveq r1, r2

For each of the next three questions, you are given a a machine instruction with specific bits set, with unknown bits set in the other positions. For each question, choose the instruction(s) above whose machine instruction encoding must have the specified bits set. Each instruction could fit in zero or more categories. Give your answer a a sequence of letters.

- 

    - BE
- 

    - AD
- 

    - C
- Consider this program:
  mov r2, #0
  ______ r1, r2, r1

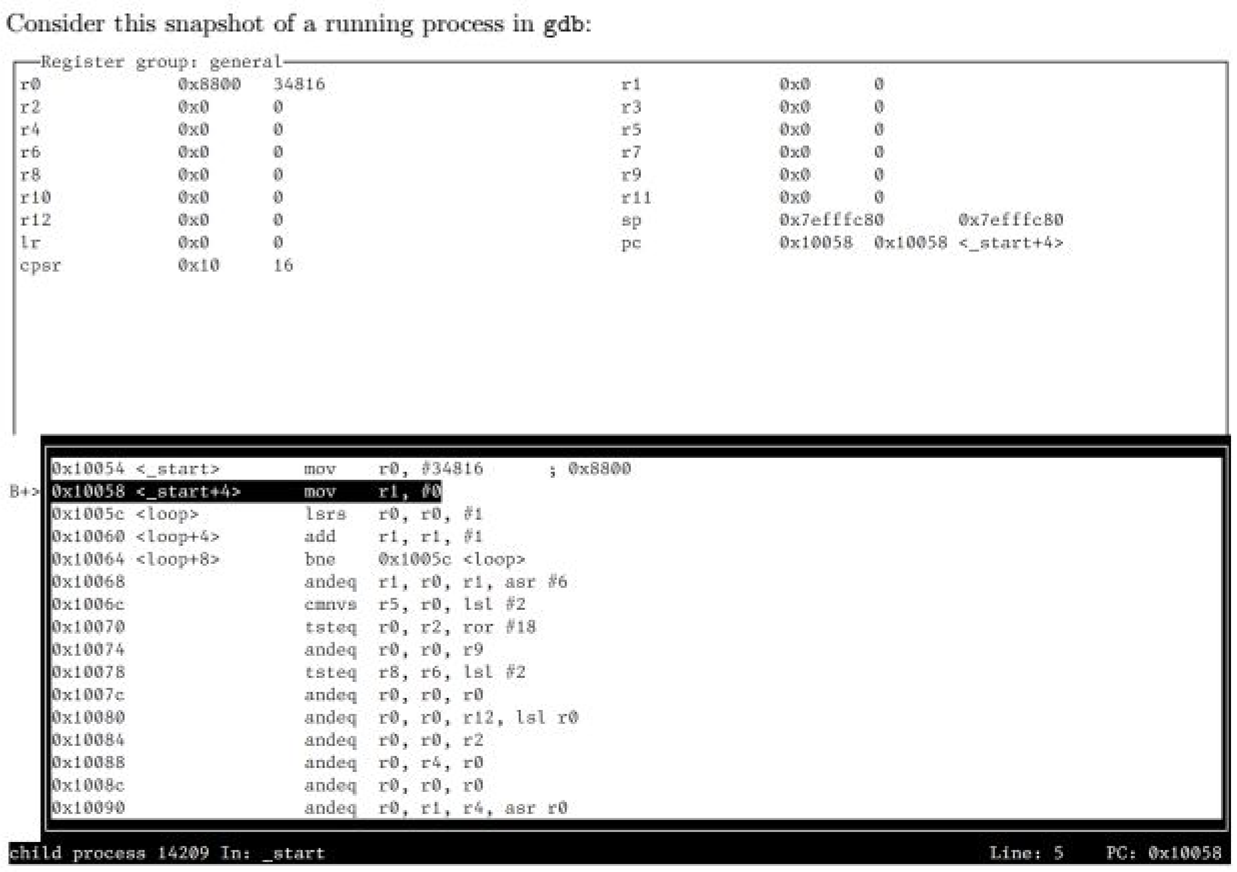
For the next three questions, some property is stated. Consider each of the command choices below filled into the blank in the program. For each answer, give the letter(s) of the commands which would make the program satisfy the property after the two instructions run. The value in r1 is intentionally unknown.

A: and
B: eor
C: orr
D: tst
E: sub

- The value in r1 is always the same as before the two instructions run
    - BCD
- The condition flags (e.g. the first four bits of cpsr) do not change
    - ABCE
- Switching the arguments to instead be r1, r1, r2 would have the same effects as the original order
    - ABCD
- Which of the following programs has the effect of flipping all of the bits in r1 (by “flip” we mean that all 1’s become 0’s and vice versa)? Choose ALL that apply.
  A: mov r0, #0xFFFFFFFF
     eor r1, r1, r0
  B: mov r0, #0xFFFFFFFF
     orr r1, r1, r0
  C: mov r0, #0x0

eor r1, r1, r0
D: mov r0, #0x0
and r1, r1, r0
E: mov r0, #0xFFFFFFFF
and r1, r1, r0
F: None of the above
- A

- Consider this snapshot of a running process in gdb:



Consider this snapshot of a running process in **gdb**:

```
┌─Register group: general─
r0             0x8800   34816                    r1             0x0      0
r2             0x0      0                        r3             0x0      0
r4             0x0      0                        r5             0x0      0
r6             0x0      0                        r7             0x0      0
r8             0x0      0                        r9             0x0      0
r10            0x0      0                        r11            0x0      0
r12            0x0      0                        sp             0x7efffc80       0x7efffc80
lr             0x0      0                        pc             0x10058  0x10058 <_start+4>
cpsr           0x10     16

    0x10054 <_start>        mov    r0, #34816      ; 0x8800
B+> 0x10058 <_start+4>      mov    r1, #0
    0x1005c <loop>          lsrs   r0, r0, #1
    0x10060 <loop+4>        add    r1, r1, #1
    0x10064 <loop+8>        bne    0x1005c <loop>
    0x10068                 andeq  r1, r0, r1, asr #6
    0x1006c                 cmnvs  r5, r0, lsl #2
    0x10070                 tsteq  r0, r2, ror #18
    0x10074                 andeq  r0, r0, r9
    0x10078                 tsteq  r8, r6, lsl #2
    0x1007c                 andeq  r0, r0, r0
    0x10080                 andeq  r0, r0, r12, lsl r0
    0x10084                 andeq  r0, r0, r2
    0x10088                 andeq  r0, r4, r0
    0x1008c                 andeq  r0, r0, r0
    0x10090                 andeq  r0, r1, r4, asr r0
child process 14209 In: _start                                    Line: 5     PC: 0x10058
```

- What are the next 4 values that the pc will hold not including the current one? Choose from the following list; give your answer as a sequence of 4 letters.
A: 0x10054
B: 0x10058
C: 0x1005c
D: 0x10060
E: 0x10064
F: 0x10068
   - CDEC
- What value will be in r0 the first time the program counter holds the address 0x10068, but before that instruction is executed? The given values are in decimal. Give your answer as a single letter.
A: 16
B: 1
C: 32

D: 4
E: Some other value
F: The program counter will never hold the address 0x10068 due to an infinite loop
  - E (https://piazza.com/class/j7w9htrxqm44j5?cid=664)

- Consider this program with holes in it:

```
mov r2, #0
loop:
____________
____________
add r2, #1
skip_add:
____________
```

Assume the goal is to have r2 contain the number of (binary) 1's in the value in r1. For example, if r1 contains
0000 1010 0001 1011 0100 0111 0000 0000
the value in r2 at the end should be (decimal) 10 or (hex) 0xa.
The program should always terminate (it should not produce an infinite loop).
Give your answer as a sequence of 3 letters corresponding to instructions below that create a program to this specification when filled in above in order.
A: lsrs r1, #1
B: lsr r1, #1
C: lsls r1, #1
D: lsl r1, #1
E: bne skip_add
F: beq skip_add
G: bcc skip_add
H: bcs skip_add
I: b skip add
J: bne loop
K: beq loop
L: bcc loop
M: bcs loop
N: b loop
  - CGJ

Tables that were provided:

Table 6.1 ARM register set

Name	Use
R0	Argument / return value / temporary variable
R1–R3	Argument / temporary variables
R4–R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Table B.5 Instructions that affect condition flags

Type	Instructions	Condition Flags
Add	`ADDS, ADCS`	N, Z, C, V
Subtract	`SUBS, SBCS, RSBS, RSCS`	N, Z, C, V
Compare	`CMP, CMN`	N, Z, C, V
Shifts	`ASRS, LSLS, LSRS, RORS, RRXS`	N, Z, C
Logical	`ANDS, ORRS, EORS, BICS`	N, Z, C
Test	`TEQ, TST`	N, Z, C
Move	`MOVS, MVNS`	N, Z, C
Multiply	`MULS, MLAS, SMLALS, SMULLS, UMLALS, UMULLS`	N, Z

Table 6.3 **Condition mnemonics**

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\overline{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\overline{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\overline{N}$
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	$\overline{V}$
1000	HI	Unsigned higher	$\overline{Z}C$
1001	LS	Unsigned lower or same	Z OR $\overline{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	Z OR $(N \oplus V)$
1110	AL (or none)	Always / unconditional	Ignored

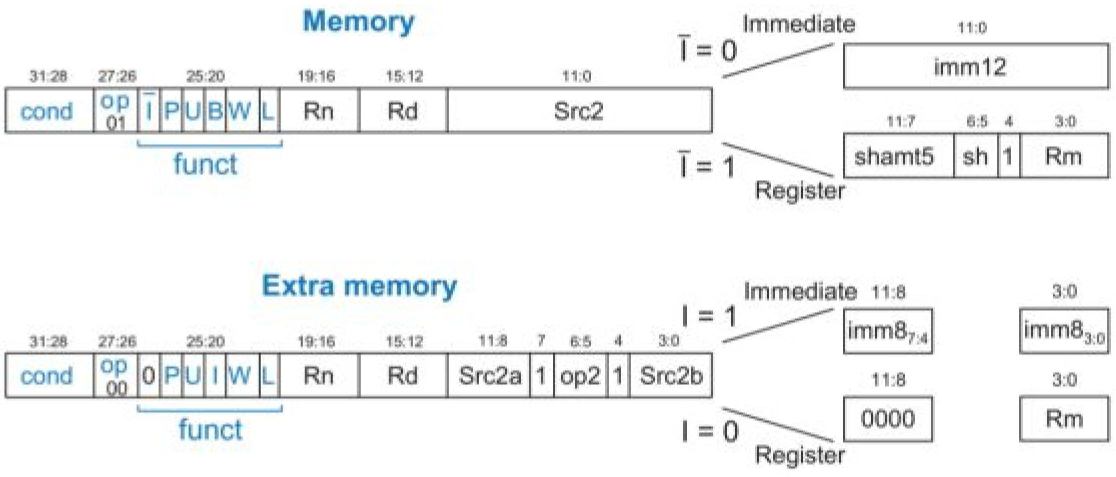


Figure B.3 Memory instruction encodings

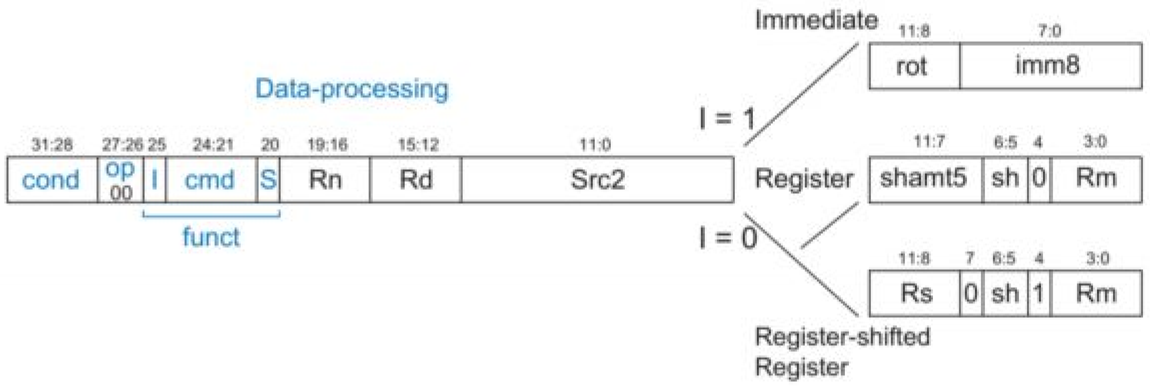


Figure B.1 Data-processing instruction encodings

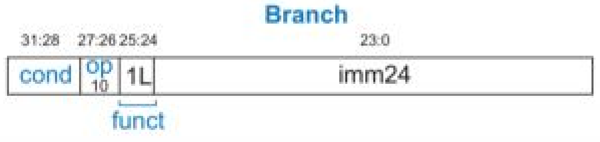


Figure B.4 Branch instruction encoding

Table B.4 Branch instructions

L	Name	Description	Operation
0	`B label`	Branch	`PC ← (PC+8)+imm24 << 2`
1	`BL label`	Branch with Link	`LR ← (PC+8) – 4; PC ← (PC+8)+imm24 << 2`

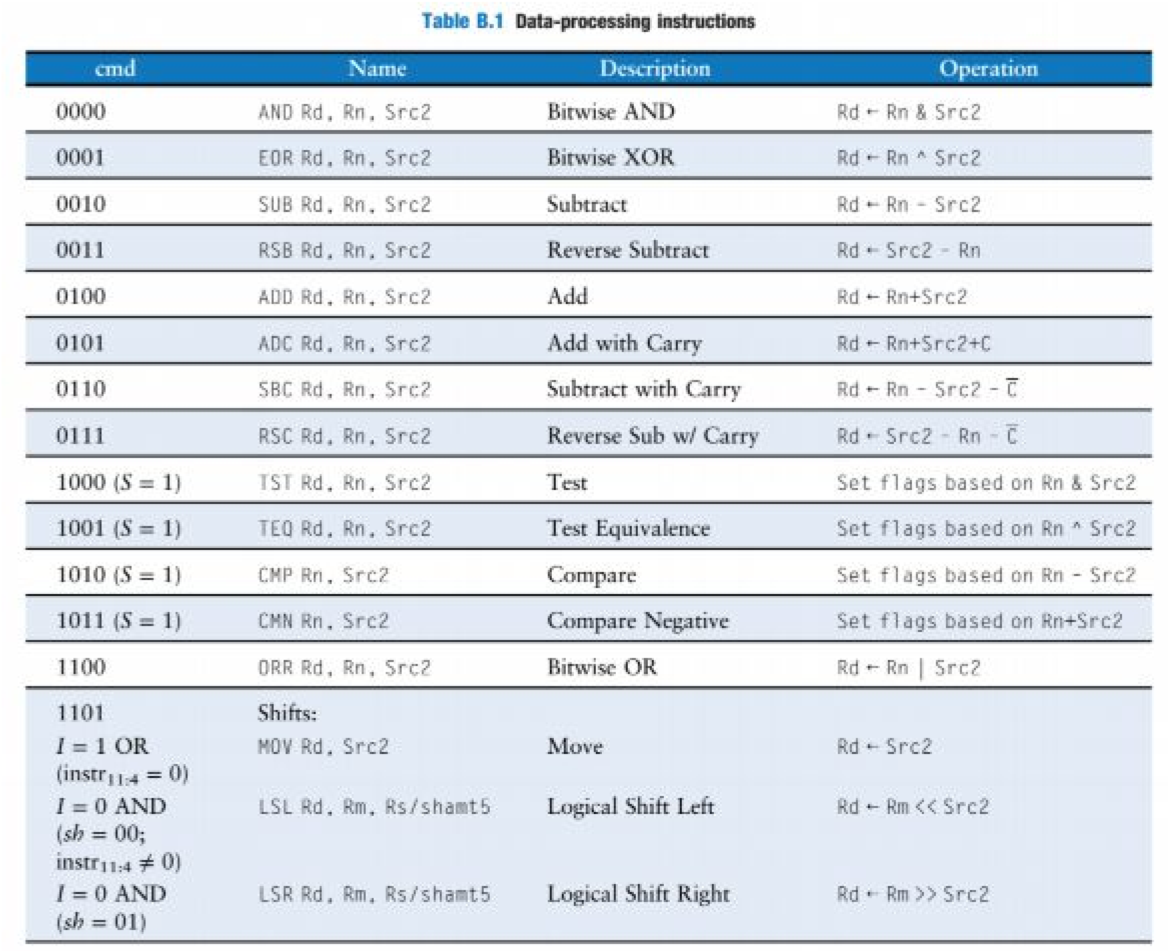


Table B.1 Data-processing instructions

cmd	Name	Description	Operation
0000	`AND Rd, Rn, Src2`	Bitwise AND	`Rd ← Rn & Src2`
0001	`EOR Rd, Rn, Src2`	Bitwise XOR	`Rd ← Rn ^ Src2`
0010	`SUB Rd, Rn, Src2`	Subtract	`Rd ← Rn - Src2`
0011	`RSB Rd, Rn, Src2`	Reverse Subtract	`Rd ← Src2 - Rn`
0100	`ADD Rd, Rn, Src2`	Add	`Rd ← Rn+Src2`
0101	`ADC Rd, Rn, Src2`	Add with Carry	`Rd ← Rn+Src2+C`
0110	`SBC Rd, Rn, Src2`	Subtract with Carry	Rd ← Rn - Src2 - $\overline{C}$
0111	`RSC Rd, Rn, Src2`	Reverse Sub w/ Carry	Rd ← Src2 - Rn - $\overline{C}$
1000 ($S = 1$)	`TST Rd, Rn, Src2`	Test	`Set flags based on Rn & Src2`
1001 ($S = 1$)	`TEQ Rd, Rn, Src2`	Test Equivalence	`Set flags based on Rn ^ Src2`
1010 ($S = 1$)	`CMP Rn, Src2`	Compare	`Set flags based on Rn - Src2`
1011 ($S = 1$)	`CMN Rn, Src2`	Compare Negative	`Set flags based on Rn+Src2`
1100	`ORR Rd, Rn, Src2`	Bitwise OR	`Rd ← Rn | Src2`
1101	Shifts:		
$I = 1$ OR ($instr_{11:4} = 0$)	`MOV Rd, Src2`	Move	`Rd ← Src2`
$I = 0$ AND ($sh = 00$; $instr_{11:4} \neq 0$)	`LSL Rd, Rm, Rs/shamt5`	Logical Shift Left	`Rd ← Rm << Src2`
$I = 0$ AND ($sh = 01$)	`LSR Rd, Rm, Rs/shamt5`	Logical Shift Right	`Rd ← Rm >> Src2`

- **Midterm 2 (**https://piazza.com/class/j7w9htrxqm44j5?cid=1937**)**

  (The differences of each versions are simply the order of questions/answers)
  (The version shown here: sudoku)

  - Consider the following layout of values on the heap, and register values (assume the little-endian layout that we usually see in GDB. For example, the byte at address 0x10001 is 0x53)

r0	0x00010008
r1	0x00010000
r2	0x00000093
r3	0x12345693
r4	0x00000000

Address	Value
0x10000	0x51525354
0x10004	0x61626364

  - Which of the following instructions, if run at this point, would change the byte holding 0x61 to hold 0x93, while leaving the rest of memory unchanged? Consider the values in registers as shown above. Choose ALL that apply:
    A. strb r2, [r0, #-1]
    B. strb r2, [r1, #7]

C. strb r3, [r0, #-1]
D. strb r3, [r1, #-1]
E. str r2, [r0, #-1]
F. str r2, [r1, #7]
G. str r3, [r0, #-1]
H. str r3, [r1, #7]
 - ABC (https://piazza.com/class/j7w9htrxqm44j5?cid=1929)

- What is the address of the byte above that holds the value 0x61? Choose one:
A. 0x10004
B. 0x10007
C. 0x10001
D. 0x10005
E. None of the above
 - B
- Which of the following sets of instructions, if run at this point, would end with the value 0x64 in r4? Choose ALL that apply:
```
A. ldr r4, [r1]
   and r4, #0x000000FF
   add r4, #10
B. ldr r4, [r0, #-4]
   and r4, #0x000000FF
C. ldrb r4, [r1]
   add r4, #10
D. ldrb r4, [r1, #4]
   add r4, #10
E. ldrb r4, [r0, #-4]
```
F. None of the above
 - BE
- Consider the C library function strcat (adapted from the manual page on strcat provided with gcc):

char* strcat(char* dest, char* src)

The strcat() function appends the src string to the dest string, over- writing the terminating null byte ('\0') at the end of dest, and then adds a terminating null byte. The strings may not overlap, and the dest string must have enough space for the result. If dest is not large enough, program behavior is unpredictable.
- Which of the following programs WOULD have unpredictable behavior based on the description above? Note that by the length of a string above, the description above is referring to what strlen would return (which includes the count of characters up to but not including the 0 byte). Choose ALL that apply.
```
A. char* s1 = malloc(7);
   char* s2 = malloc(2);
   s1[0] = 'a'; s1[1] = 'b'; s1[2] = 'c'; s1[3] = '\0';
   s2[0] = 'd'; s2[1] = '\0';
   strcat(s1, s2);
B. char* s1 = malloc(5);
   char* s2 = malloc(3);
```

s1[0] = ’a’; s1[1] = ’b’; s1[2] = ’c’; s1[3] = ’\0’;
s2[0] = ’d’; s2[1] = ’e’; s2[2] = ’\0’;
strcat(s1, s2);

C. char* s1 = malloc(5);
char* s2 = malloc(3);
s1[0] = ’a’; s1[1] = ’b’; s1[2] = ’c’; s1[3] = ’\0’;
s2[0] = ’d’; s2[1] = ’\0’;
strcat(s1, s2);

D. char* s1 = malloc(5);
s1[0] = ’a’; s1[1] = ’b’; s1[2] = ’c’; s1[3] = ’\0’;
strcat(s1, s1 + 1);

E. None of the above

- BD

Can anyone helps with choice D? Thx ← just think of s1+1 as accessing b and further ok
Yes s1 + 1 can accessing b and rest of the string, but when concatenating s1 and s1+1, which is essentially abcbc, we need 6 bytes to store the extra \0.

Consider this assembly implementation of index of, which takes a char* and a char in r0 and r1 respectively, and ends with r2 containing the index of where the character appears (starting with index 0), and -1 if it doesn’t appear.

```
@ r0 contains address of string
@ r1 contains character to search for
@ r2 used for current index
@ r3 used to store current character
Index_of:
mov r2, #0
mov r3, #0
loop:
_______________
_______________
beq not_found
_______________
beq found
add r2, #1
b loop
Not_found:
mov r2, #-1
Found:
@ do nothing, r2 already contains the correct value
```

Fill in the implementation above using the following instructions:

A. cmp r3, r2
B. cmp r3, r1
C. cmp r1, r2
D. cmp r0, r3
E. cmp r3, #0
F. cmp r2, #0
G. cmp r1, #0

H. cmp r0, #0
I. ldrb r3, [r0, r2]
J. ldrb r3, [r1, r2]
K. ldrb r3, [r0, r1]
L. ldrb r0, [r2, r3]
    - IEB
- .text
  .global f
  F:
  sub sp, #8
  str lr, [sp, #4]
  str r4, [sp]
  mov r4, r0
  bl strlen
  lsl r0, #1
  add sp, #8
  ldr r4, [sp, #-8]
  ldr lr, [sp, #-4]
  extern int f(char* s);
  int main() {
  char* s = strdup("abcd");
  int twolen = f(s);
  }
- When called from C main as above, this implementation produces a segmentation fault. What's a likely explanation and fix?
  A. It doesn't save the original value of lr, so that register is overwritten. Add push {lr} at the beginning and pop {lr} at the end.
  B. The program attempts to modify the string, which is read-only. The string shouldn't be allocated on the heap with strdup.
  C. It never returns control to the caller by changing pc. Replace the use of lr in the last line with pc.
  D. It doesn't save the argument to the stack, so it gets overwritten. Create space for a local variable and store r0 there by using push and pop.
  E. It doesn't save the original value of sp, so that register is overwritten. Add push {sp} at the beginning and pop {sp} at the end.
  F. None of the above
    - C
- Which of the following invariants does a callee need to maintain in the ARM 32-bit convention? Choose ALL that apply.
  A. The sp register must to be the same when returning as at the start of the function.
  B. The values of r0-r3 must be the same when returning as at the start of the function.
  C. The values of r4-r10 must be the same when returning as at the start of the Function.
  D. The value of lr must be the same when returning as at the start of the function.
  E. The value of pc must be the same when returning as at the start of the Function.

F. The value of pc must be the same when returning as at the start of the function.
G. All memory that has been allocated on the heap since the beginning of the function must be freed.

- **AC *D***

○ Consider the following program, and the corresponding ARM assembly that GCC generated for it (slightly shortened):

```
check_mod:
  push  {lr}
  sub sp, sp, ___________
  str r0, _____________
  str r1, _____________
  str r2, _____________
  ldr r0, [sp, #12]
  ldr r1, [sp, #8]
  bl  __aeabi_divmod
  mov r2, r1
  ldr r3, [sp, #4]
  cmp r2, r3
  moveq r0, #1
  movne r0, #0
  add sp, sp, #20
  pop {pc}
main:
  push  {r3, lr}
  mov r0, #27
  mov r1, ___________
  mov r2, ___________
  bl  1041c <check_mod>
  mov r0, r3
  pop {r3, pc}
```

```
int check_mod(int n, int d, int c) {
  return n % d == c;
}

int main() {
  check_mod(27, 5, 2);
}
```

There are some holes in the generated assembly. Fill them in, in order, using the instructions and instruction fragments below. Give your answer as a sequence of 6 letters in order.

A. [sp, #20]
B. [sp, #16]
C. [sp, #12]
D. #20
E. #5
F. #2
G. #27
H. #16
I. #8
J. [sp, #8]
K. [sp, #4]
L. [sp]

- DCJKEF

○ Consider this C program for the next two questions:

```
int* make_list(int upto) {
  int i;
  int* nums = malloc(sizeof(int) * upto);
  for(i = 0; i < upto; i += 1) {
    nums[i] = i * i;
  }
  return nums;
}
int main() {
  int* some_nums = make_list(7);
  int i;
  int sum = 0;
  for(i = 0; i < 7; i += 1) {
    sum += some_nums[i];
  }
}
```

- Which of the following statements could replace nums[i] = i * i above without changing the program's meaning? Choose ALL that apply.
  A. nums + i = i * i;
  B. *(nums + i) = i * i;
  C. (*nums) + i =7 * i;
  D. *nums[i] = i * i;
  E. None of the above
  - B
- This program leaks memory – all of the bytes allocated for the array. Where is the most appropriate place to free the memory?
  A. At the end of make list, before the return nums statement.
  B. Before the for loop in the main function, but after sum is declared.
  C. Before the for loop in make list
  D. Immediately after the make list function returns
  E. None of the above
  - E
- Consider this assembly program and two lines of C code that generated them (lightly edited from the raw output of gcc).

```
mov r0, #12
bl  malloc
mov r3, r0          int* m = malloc(sizeof(int) * 3);
add r3, r3, #8      __________ = 10;
mov r2, #10
str r2, [r3]
```

- Which of the following left-hand-sides would fill in the C program to correspond to the generated assembly? Assume that ints take 4 bytes of memory.
  A. nums[0]
  B. nums[8]
  C. nums[7]
  D. nums[2]
  E. nums[3]
  F. None of the above
  - D
- Consider the following program:

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3
 4 struct Point {
 5   int x;
 6   int y;
 7 };
 8
 9 struct Pair {
10   struct Point* left;
11   struct Point* right;
12 };
13
14 struct Point* f(struct Point* p) {
15   p->x = 10;
16   p = malloc(sizeof(struct Point)); // Malloc A
17   p->x = 9;
18   p->y = 11;
19   return p;
20 }
21 int main() {
22   struct Point* pt1 = malloc(sizeof(struct Point)); // Malloc B
23   pt1->x = 1;
24   pt1->y = 2;
25
26   struct Point* pt2 = malloc(sizeof(struct Point)); // Malloc C
27   pt2->x = 3;
28   pt2->x = 4;
29
30   struct Pair* pair = malloc(sizeof(struct Pair)); // Malloc D
31   pair->left = pt1;
32   pair->right = pt2;
33
34   struct Point* pt = f(pair->left);
35
36   printf("%d, %d, %d", pair->left->x, pt1->x, pt->x);
37
38   free(pair->left);
39   free(pair->right);
40   free(pair);
41 }
```

When run on this program, Valgrind reports (among other things):

```
==4112== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4112==    at 0x4833970: malloc (vg_replace_malloc.c:263)
==4112==    by 0x104A3: f (structs.c:16)
==4112==    by 0x10567: main (structs.c:34)
```

- Assuming ints are 4 bytes, and pointers are 4 bytes, and structs take space equal to the sum of their members, how much total memory does this program allocate with malloc, whether it's freed or not by the end? Give your answer directly in the answer sheet as a number of bytes.
    - 32
- For each of the three free expressions at the bottom, indicate the line of the malloc that produced the address being freed. Use the letter given on the malloc line (A-D), give your answer as a sequence of three letters.
    - BCD

- What does this program print? Write your answer directly into the answer sheet as three numbers separated by commas.
    - 10, 10, 9