Question 1 This question involves cache lookups. The cache below has the following properties.

- The memory is byte addressable.
- Memory accesses are to 1-byte words.
- Addresses are 12 bits wide.
- The cache is 2-way set associative, with a 4 byte block size and 8 total lines.

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

Set	Tag	Valid	Byte0	Byte1	Byte2	Byte3
S_0	58	0	02	55	AD	87
S_0	23	1	3E	98	47	51
S_1	2B	0	6C	77	89	14
S_1	EF	1	B9	64	78	25
S_2	69	0	00	FF	14	43
S_2	2B	1	92	63	42	21
S_3	75	0	33	BE	AF	31
S_3	C6	1	22	17	02	24

For the given addresses, fill out the two tables below. Give your answer in hexadecimal. If there is a cache miss, enter “N/A” for “Cache Byte returned”.

Address: **0x2B9**	
Parameter	**Value**
Block offset	**0x1**
Set Index	**0x2**
Cache Tag	**0x2b**
Cache Byte returned	**0x63**

Address: **0x75E**	
Parameter	**Value**
Block offset	**0x2**
Set Index	**0x3**
Cache Tag	**0x75**
Cache Byte returned	**N/A**

Question 2 You are given the following functions, both of which compute the product of an array. Assume that the code below has been compiled using `gcc` with the optimization level set to "`-Og`".

A

```
#define MAX 100000

void prod1(int array[MAX], int* prod) {
  int i;
  *prod = 1;
  for (i = 0; i < MAX; i++) {
      *prod *= array[i];
  }
}
```

B

```
#define MAX 100000

void prod2(int array[MAX], int* prod) {
  int i;
  int total = 1;
  for (i = 0; i < MAX; i++) {
      total *= array[i];
  }
  *prod = total;
}
```

Which is true about the above code? Fill in **one** oval.

- ○ Code in block A (prod1) will run faster
- ○ **The code in block B (prod2) will run faster**
- ○ Both functions will take about the same amount of time to run
- ○ The program are not comparable, because they give different answers

Question 3 Recall that floating point numbers are represented in the form $V = (-1)^s \times M \times 2^E$ where `M` is encoded in `frac`, and `E` is encoded in `exp`. Assume we have the following tiny **16-bit** floating point representation where bits 0-9 represent the fraction field, bits 10-14 represent the exponent field, and bit 15 represents the sign bit, as shown below.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exp					frac									

What is the floating point representation of the decimal value **42.25**? Give your answer in **hexadecimal**.

0x5148_____

Question 4 Assume you have a two's complement number system where integers are 20 bits. What is the largest (most positive) number (TMax) and what is the smallest (most negative) number (TMin)? Give your answer in hexadecimal.

TMax: **0x7ffff**_____

TMin: **0x80000**_____

Question 5 Round the following binary fractions to binary whole numbers using the "round to even" rounding mode.

Binary	**Rounded Binary**
100.11	101
011.1	100
010.0111	010
010.1	010
101.101	110

Question 6 Suppose we have the following bytes in memory:

Address	0x640	0x641	0x642	0x643	0x644	0x645	0x646	0x647	0x648	0x649	0x64a	0x64b	0x64c	0x64d	0x64e	0x64f	0x650
Value	0x21	0x18	0x9e	0xad	0xb6	0xcd	0xd1	0x83	0xfd	0x81	0xeb	0x7d	0xfe	0x44	0xe7	0x27	0x1c

What are the values of the following expressions below? **Give your answers in hexadecimal.**

Expression	**Value**
Little-endian **char** at address **0x643**	**0xad**
Little-endian **short** at address **0x64b**	**0xfe7d**
Little-endian **int** at address **0x644**	**0x83d1cdb6**
Big-endian **short** at address **0x646**	**0xd183**
Big-endian **int** at address **0x64c**	**0xfe44e727**

Question 7 Write the following function in RISC-V assembly.

C code

```
struct q7 {
    char a;
    int b;
    int *c;
};

int* getC(struct q7 *s) {
    return s->c;
}
```

RISC-V assembly

```
getC:
        lw      a0,8(a0)
        ret
```

Question 8 (4 points) Write the following function in RISC-V assembly.

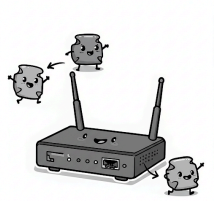
C code

```
int getElement(int array[], int index) {
    return array[index];
}
```

RISC-V assembly

```
getElement:
        slli    a1,a1,2
        add     a0,a0,a1
        lw      a0,0(a0)
        ret
```

Question 9 Router and Packet Synchronization with Pthreads



The following C program models a **network router** using `pthreads`. Each **packet** is simulated by its own thread. The **router**, also a thread, consumes packets from a shared circular buffer. The buffer can only hold a limited number of packets. If the buffer is full, incoming packets are **dropped**. If the buffer is empty, the router goes to sleep.

Fill in the missing code in the **`router_thread`** and **`packet_thread`** functions to correctly implement the described behavior. Use appropriate locking and condition variables to avoid race conditions and to manage synchronization.

Your implementation should:

- Drop packets, by calling the `drop()` function, if the buffer is full.
- Signal the router, by calling the `signal()` function, only when a new packet arrives and the router is sleeping.
- Route packets, by calling the `route()` function, when there are packet IDs in the buffer.
- Allow the router to sleep, by calling the `wait()` function, if there are no packets to process.

Functions to be implemented:

- `router_thread`: waits if the buffer is empty, otherwise removes a packet and routes it.
- `packet_thread`: adds a packet to the buffer or drops it if the buffer is full. It also wakes the router if it is sleeping.

Key structures:

The **`router_t`** structure gets passed into the router thread. It contains a circular buffer of packet IDs, the necessary information state, a lock, and a condition variable, which are needed to perform the proper synchronization between the router and packet threads.

```
typedef struct {
    pthread_mutex_t lock;
    pthread_cond_t not_empty;
    int buffer_size; // the buffer can hold this number of packets
    int *buffer;     // the buffer; can be accessed like an array of ints
    int count;       // count of packets currently in the buffer
    int in;          // put packets into the buffer at this index
    int out;         // take out packets from the buffer at this index
} router_t;
```

The **`packet_t`** structure gets passed into each packet thread. It contains the router info (see above struct) along with an unique packet ID.

```
typedef struct {
    router_t *r; // same struct that the router has
    int id;      // packet id
```

```
} packet_t;
```

Helper functions:

To simplify your design (and to save you from a bunch of writing), here are some helper functions to use.

- **route**`(int packet_id)` -- Simulates routing a packet. **Note**: this should only be called from the router thread.
- **drop**`(int packet_id)` -- Simulates dropping a packet. **Note**: this should only be called from a packet thread.
- **lock**`(router_t *r)` -- Wrapper function for `pthread_mutex_lock()`
- **unlock**`(router_t *r)` -- Wrapper function for `pthread_mutex_unlock()`
- **wait**`(router_t *r)` -- Wrapper function for `pthread_cond_wait()`
- **signal**`(router_t *r)`-- Wrapper function for `pthread_cond_signal()`

These are the only function calls that you need to use, so you should not be calling any other functions in your implementation. **Note:** the last four helper functions all take a pointer to a `router_t` struct. As shown below, the helper functions will supply the correct arguments to the associated pthread functions for you.

```
void route(int packet_id) { // Simulate routing a packet
    printf("Routing packet %d\n", packet_id);
    usleep(100); // simulate time to route the packet
}

void drop(int packet_id) { // Simulate dropping a packet
    printf("Dropping packet %d\n", packet_id);
}

void lock(router_t *r) { // Wrapper function for pthread_mutex_lock
    pthread_mutex_lock(&r->lock);
}

void unlock(router_t *r) { // Wrapper function for pthread_mutex_unlock
    pthread_mutex_unlock(&r->lock);
}

void wait(router_t *r) { // Wrapper function for pthread_cond_wait
    printf("No packets. Router Sleeping...\n");
    pthread_cond_wait(&r->not_empty, &r->lock);
}

void signal(router_t *r) { // Wrapper function for pthread_cond_signal
    pthread_cond_signal(&r->not_empty);
}
```

In addition to the above helper functions, there is also a function which will initialize the router struct:

```
void router_init(router_t *r, int buffer_size) { // initializes the router struct
    pthread_mutex_init(&r->lock, NULL);
    pthread_cond_init(&r->not_empty, NULL);
    r->buffer = malloc(sizeof(int)*buffer_size);
    assert(r->buffer); // check for NULL pointer
    r->buffer_size = buffer_size;
    r->count = 0;
    r->in = 0;
    r->out = 0;
}
```

A small driver to test the code is shown below:

```
 int main() {
    // number of packets the router can store before dropping packets
    int buffer_size = 5;  // size of the packet buffer
    int num_packets = 20; // how many packet threads to create
    pthread_t router;     // router thread
    pthread_t packets[num_packets]; // packet threads
    router_t router_arg;  // router argument passed into the router thread
    packet_t packet_arg[num_packets]; // packet argument passed into packet thread

    router_init(&router_arg, buffer_size);

    pthread_create(&router, NULL, router_thread, &router_arg); // router thread
    for (int i = 0; i < num_packets; i++) { // packet threads
        packet_arg[i].r = &router_arg;
        packet_arg[i].id = i;
        pthread_create(&packets[i], NULL, packet_thread, &packet_arg[i]);
    }

    // wait for packet threads to finish
    for (int i = 0; i < num_packets; i++) {
        pthread_join(packets[i], NULL);
    }

    // For demo purposes only: let router finish remaining packets then exit
    sleep(1);
    printf("[Main] Done.\n");
    exit(0);
}
```

A sample run of the program is shown below.

```
./router
No packets. Router Sleeping...
Routing packet 0
Dropping packet 5
Routing packet 1
Routing packet 2
Routing packet 3
Routing packet 4
Routing packet 6
Dropping packet 12
Dropping packet 13
Routing packet 7
Dropping packet 14
Dropping packet 17
Dropping packet 16
Dropping packet 18
Dropping packet 19
Routing packet 8
Routing packet 9
Routing packet 10
Routing packet 11
Routing packet 15
No packets. Router Sleeping...
[Main] Done.
```

Your job is to finish the implementation of the **`router_thread`** and the **`packet_thread`** functions to correctly implement the described behavior. Use appropriate locking to avoid data races and appropriate calls to `wait` and `signal` to manage synchronization.

```
void* packet_thread(void* arg) {
    router_t *r = ((packet_t *)(arg))->r;    // get a pointer to our router info
    int packet_id = ((packet_t *)(arg))->id; // get our packet_id
    lock(r);
    // if the buffer is full, drop packet
    if (r->count >= r->buffer_size) {
       unlock(r);
       drop(packet_id);
       return NULL;
    }

    // else add packet to the buffer and signal
    r->buffer[r->in] = packet_id;
    r->in = (r->in + 1) % r->buffer_size;
    if (r->count == 0) { // wake the router up
        signal(r);
    }
    r->count++;
    unlock(r);
    return NULL;
}
void* router_thread(void* arg) {
    router_t *r = arg; // get a pointer to our router info
    while (1) {  // router never exits
        lock(r);
        while (r->count == 0) {
            wait(r);
        }
        assert(r->count > 0);

        // get packet id from the buffer
        int packet_id = r->buffer[r->out];
        r->out = (r->out + 1) % r->buffer_size;
        r->count--;
        unlock(r);
        route(packet_id);
    }
    return NULL; // Never reached
}
```

Question 10 The single-lane bridge with semaphores

A bridge over a river is single-lane: cars can travel in either direction, but never simultaneously. Any number of cars may share the bridge at once as long as they are all going the same way. Northbound traffic blocks southbound traffic, and vice versa.

In the starter code below, fill in the bodies of `northbound()` and `southbound()` using POSIX semaphores so that:

1. No two cars are on the bridge at the same time going opposite directions.
2. Multiple cars may be on the bridge simultaneously if they are going the same way.
3. Cars do not block unnecessarily — a northbound car must not wait if other northbound cars are already crossing.

You do not need to prevent starvation for this problem. A continuous stream of one direction may permanently block the other.

Hints:

- You will need more than one semaphore. Think about which ones are binary (mutex-style) and which represent the bridge itself.
- Each direction needs to know whether it is the **first** car claiming the bridge or the **last** car releasing it. A counter protected by a mutex is a natural way to track this.
- Be careful with what you hold while you wait. If a thread is blocked on the bridge while still holding a mutex that the *other* direction's exiting car needs, you have a deadlock. Trace through this carefully when you design your solution.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

#define NUM_NORTH 5
#define NUM_SOUTH 5

/* ----- shared state ------------------------------------------------ */
sem_t north_mutex;       /* protects north_count        */
sem_t south_mutex;       /* protects south_count        */
sem_t bridge;            /* held by the active direction */
int   north_count = 0;
int   south_count = 0;

/* ----- placeholder for the actual crossing ------------------------- */
void cross_bridge(int id, const char *dir)
{
    printf("  [%s%d] on  the bridge\n", dir, id);
    usleep(200000);                  /* simulate ~200 ms crossing */
    printf("  [%s%d] off the bridge\n", dir, id);
}
```

```
void northbound(int id)
{
    sem_wait(&north_mutex);
    north_count++;
    if (north_count == 1) sem_wait(&bridge);   /* first NB claims it */
    sem_post(&north_mutex);

    cross_bridge(id, "N");

    sem_wait(&north_mutex);
    north_count--;
    if (north_count == 0) sem_post(&bridge);   /* last NB releases  */
    sem_post(&north_mutex);
}

void southbound(int id)
{
    sem_wait(&south_mutex);
    south_count++;
    if (south_count == 1) sem_wait(&bridge);
    sem_post(&south_mutex);

    cross_bridge(id, "S");

    sem_wait(&south_mutex);
    south_count--;
    if (south_count == 0) sem_post(&bridge);
    sem_post(&south_mutex);
}
```

```
/* ----- thread entry points ---------------------------------------- */
void *north_thread(void *arg)
{
    int id = *(int *)arg;
    usleep((rand() % 500) * 1000);   /* stagger arrivals */
    northbound(id);
    return NULL;
}

void *south_thread(void *arg)
{
    int id = *(int *)arg;
    usleep((rand() % 500) * 1000);
    southbound(id);
    return NULL;
}

int main(void)
{
    pthread_t n[NUM_NORTH], s[NUM_SOUTH];
    int       n_id[NUM_NORTH], s_id[NUM_SOUTH];

    sem_init(&north_mutex, 0, 1);
    sem_init(&south_mutex, 0, 1);
    sem_init(&bridge,      0, 1);

    for (int i = 0; i < NUM_NORTH; i++) {
        n_id[i] = i;
        pthread_create(&n[i], NULL, north_thread, &n_id[i]);
    }
    for (int i = 0; i < NUM_SOUTH; i++) {
        s_id[i] = i;
        pthread_create(&s[i], NULL, south_thread, &s_id[i]);
    }
    for (int i = 0; i < NUM_NORTH; i++) pthread_join(n[i], NULL);
    for (int i = 0; i < NUM_SOUTH; i++) pthread_join(s[i], NULL);

    sem_destroy(&north_mutex);
    sem_destroy(&south_mutex);
    sem_destroy(&bridge);
    return 0;
}
```