**University of California, Berkeley – College of Engineering**

Department of Electrical Engineering and Computer Sciences

Summer 2018 Instructors: Steven Ho, Nick Riasanovsky August 9, 2018

# CS61C FINAL

| | | |
|---|---|---|
| ***Last*** *Name (Please print clearly)* | Perfect | |
| ***First*** *Name (Please print clearly)* | Petra | |
| *Student ID Number* | 61C | |
| *Circle the name of your Lab TA* | **Damon Jonathan Sean Sruthi Emaan Suvansh Sukrit** | |
| *Name of the person to your: Left \| Right* | Steven | Nick |
| *All my work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61C who haven't taken it yet.* ***(please sign)*** | | |

## Instructions

- This booklet contains 24 pages including this cover page.
- Please turn off all cell phones, smartwatches, and other mobile devices. Remove all hats, headphones, and watches. Place *everything* except your writing utensil(s), cheat sheet, and beverage underneath your seat.
- You have 170 minutes to complete this exam. The exam is closed book: no computers, tablets, cell phones, wearable devices, or calculators. You are allowed three pages (US Letter, double-sided) of *handwritten* notes.
- There may be partial credit for incomplete answers; write as much of the solution as you can and show work in the white spaces provided.
- Please write your answers within the blanks provided within each problem!
- Please clearly bubble in or circle the correct answer choice(s) for multiple choice questions!

| **Question** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **Total** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Possible Points** | 16 | 8 | 9 | 9 | 8 | 16 | 16 | 6 | 10 | 10 | 10 | 10 | 6 | 6 | 140 |

If you have the time, **feel free to doodle on the front page!**

## Question 1: Main [Memory] Stacks (16 pts)

Recall the idea of the Stack section in our memory hierarchy: when functions are called, their function frames are **pushed** onto the stack; when a function returns, it is **popped** off the stack. We will be implementing the Stack section of memory in software by representing each function frame on the stack as a struct, defined below. Assuming we are running on a **32-bit machine** with **32-bit integers**.

```
typedef struct stack {
      void *frame;          // Pointer to the space allocated for this function frame
      struct stack *prev;   // Pointer to previous stack node
      char *func_name;      // Name of the stack frame’s function
      int threads;          // The number of threads below this stack frame
      int in_use;           // The number of threads currently pointing to this stack frame
} StackNode;

StackNode *sp = NULL;
```

1) What would `sizeof (StackNode)` return?

5 * 4B = 20 B

Suppose that the staff has implemented a function `frame_size` which takes in a C function’s name as a string and outputs how many **bytes** that the function’s local variables would need in the function frame:

```
int frame_size (char *func_name) {
      \\ Assume this function has been correctly implemented for you.
}
int main_stacks (int argc, char *argv[]) {
      char arr[] = “61C rox”;
      return 0;
}
```

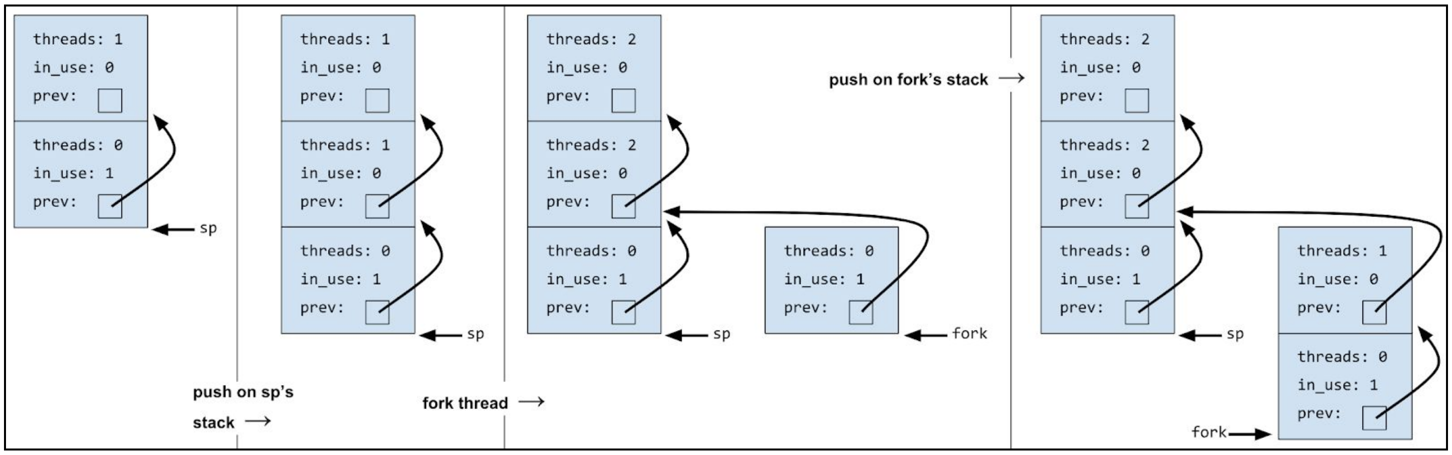
2) What would `frame_size (“main_stacks”)` return?

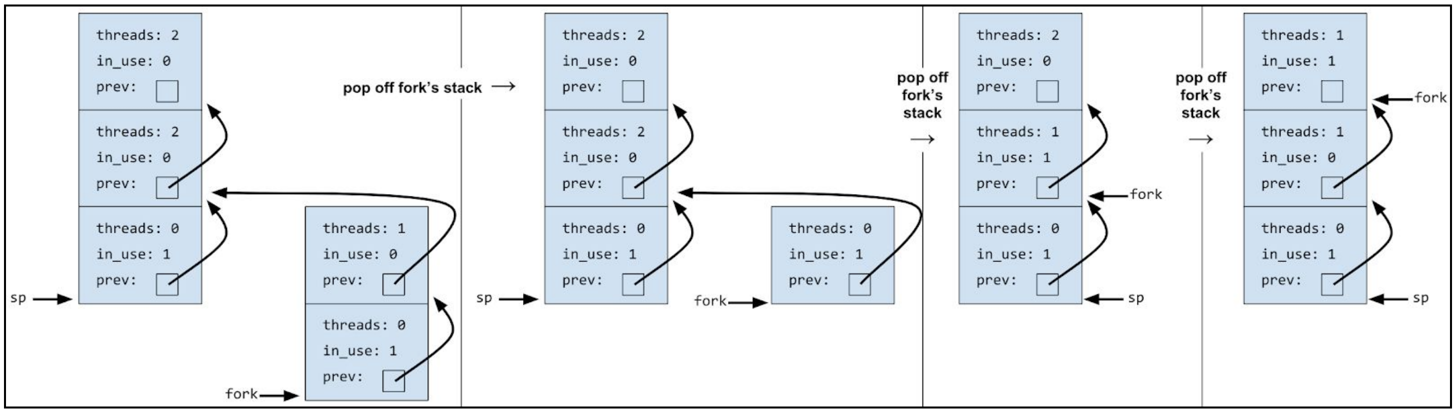
2 * 4B + 8B = 16 B

Recall that for a single thread, its Stack section is essentially a linked list where the stack pointer **sp** points to the end of the stack. Also recall that when you **fork** a thread, the new thread will have their own copy of the stack. Instead of actually copying the function frames, we just point to the “shared” stack frames.

On the next page is a visual of the stack that we will be designing. The first set is an example of pushing function frames on the stack along with creating a new fork. The second set is continuously popping off from the fork’s stack.

SID: ________________





Fill in the following code sequence that will push the function frame onto the stack and modify the sp passed in. Assume sp is never NULL and func_name is always stored in static data—don't allocate space for it.

```
void push (char *func_name, StackNode **sp) {
      StackNode *temp = (StackNode *) malloc(sizeof(StackNode));
      temp->frame = malloc( frame_size(func_name) );
      temp->prev = *sp;
      temp->func_name = func_name;
      temp->threads = 0;
      temp->in_use = 1;
      if (*sp == NULL) {
            *sp = temp;
            return;
      }
      if ((*sp)->in_use > 0) {
            (*sp)->threads += 1;
            (*sp)->in_use -= 1;
      } else {    /* We are making a fork. */
            for (StackNode *trace = *sp; trace != NULL; trace = trace->prev) {
                  trace->threads += 1;
            }
      }
      *sp = temp;
}
```

Fill out the following code sequence that will pop the function frame off the stack, modifying the sp passed in. Also assume that sp is never NULL. Since our stack frame is possibly shared by multiple "threads" you will want to consider when we should free the memory for a frame we pop off.

```
void pop (StackNode **sp) {
          printf("%s has been returned\n", (*sp)->func_name);
          (*sp)->in_use--;
          bool free_frame = false;
          if ( (*sp)->threads == 0 && (*sp)->in_use == 0 ) {
             free_frame = true;
          }
          StackNode *temp = (*sp)->prev;
          if (free_frame) {
              free((*sp)->frame);
              free(*sp);
          }
          *sp = temp;
          if ((*sp) != NULL) {
              (*sp)->threads -= 1;
              (*sp)->in_use += 1;
          }
}
```

## Question 2: Main [Memory] Stacks Management (8 pts)

In this question, we will continue the Stack implementation from Question 1, assuming it functions correctly. Consider the following program (and feel free to draw the stack in the space to the right :D ):

```
StackNode *sp = NULL;
int main(int argc, char *argv[]) {
      push("main", &sp);
      push("foo", &sp);
      push("bar", &sp);
      StackNode *fork = sp;
      push("orig", &sp);
      push("split", &fork);
      return 0;
}
```

Each of the following values on the next page evaluate to an address in the C code above. Select the region of memory that these addresses point to right before the main function returns.

SID: ________________

| | | | | | |
|---|---|---|---|---|---|
| 1. | `main` | Ⓐ Stack | Ⓑ Heap | Ⓒ Static | [✓ Ⓓ Code] |
| 2. | `&sp` | Ⓐ Stack | Ⓑ Heap | [✓ Ⓒ Static] | Ⓓ Code |
| 3. | `sp` | Ⓐ Stack | [✓ Ⓑ Heap] | Ⓒ Static | Ⓓ Code |
| 4. | `*sp` | Ⓐ Stack | [✓ Ⓑ Heap] | Ⓒ Static | Ⓓ Code |
| 5. | `sp->func_name` | Ⓐ Stack | Ⓑ Heap | [✓ Ⓒ Static] | Ⓓ Code |
| 6. | `&fork` | [✓ Ⓐ Stack] | Ⓑ Heap | Ⓒ Static | Ⓓ Code |
| 7. | `argv` | [✓ Ⓐ Stack] | Ⓑ Heap | Ⓒ Static | Ⓓ Code |

Suppose we had a simple recursive function defined as follows:

```
long factorial(long n):
    if (n == 1):
        return 1;
    else:
        return n * factorial(n-1)
```

Assume that function frames only require space for the local variables (i.e. the return value of `frame_size("factorial")` ). You are given the following specifications:

Stack and Heap: 16 KiB
Static: 12 KiB
Code: 4 KiB

`frame_size("factorial") = 8`
`sizeof(StackNode) = 56`

Suppose we call the `factorial` function on some number N **using our Stack data structure from Question 1** (note: we allocated data for our `StackNode` structs):

```
StackNode *sp = NULL;
int main () {
      push("factorial", &sp);
      push("factorial", &sp);
      push("factorial", &sp);
      ...

      // the N'th call to factorial
      push("factorial", &sp);

      // the first return from factorial
      pop(&sp);
      ...
}
```

What is the smallest value of N that will cause a maximum recursion depth error (meaning no more function frames can be created)? Ignore the stack space required for the `main` function. If convenient, put your answer as a power of 2.

$2^{14}$ B of Heap Space; Each function call needs 8 + 56 = 64 = $2^6$ B of heap space

N = $2^{14} / 2^6 = 2^8$

## Question 3: Go With the Overflow (9 pts)

So far in RISC-V, we have not dealt with overflow exceptions. Implement the following using exactly the lines given such that if there is overflow done by the `add/addi` instruction, you **branch** to the `overflow` label where the exception handler is. Otherwise, if there is no overflow, you jump to `end`.

```
# Unsigned addition overflow
Q1:
           add  t0, t1, t2
           bltu t0, t1, overflow
           j end

# Signed addition overflow with
positive immediate
Q2:
           addi t0, t1, POS_IMM
           blt t0, t1, overflow
           j end

# General signed addition
Q3:
           add  t0, t1, t2
           slt t3, t2, x0
           slt t4, t0, t1
           bne t3, t4, overflow
           j end

overflow:  ...

end:       ...
```

```
Scratch space (not graded)
```

**`Hint: It is true that the sum should be less than one of the operands if and only if the other operand is negative`**

Suppose that the label `Q1` is at address 0x4000 0000. If the label `end` is at address 0x40**XY Z**800, what are all the possible values for **X**, **Y**, and **Z** such that `j end` can be resolved in the assembler? Formulate your answer in the form `[A - B]` where `A` and `B` are both hexadecimal digits.

**X**: 0
**Y**: [0 - F]
**Z**: [0 - F]

SID: ________________

## Question 4: 🅱loating Point (9 pts)

Being the feisty floating point fanatic you are, you devise a new scheme to interpret 32-bit floats. You keep the breakdown of the Sign/Exponent/Significand fields the same. However, there is no implicit 1. or 0. before the significand bits; also, you evaluate the 23 significand bits as an **unsigned number**, which you then multiply with $2^{Exp\ -\ Bias}$ and $(-1)^{Sign}$ as you normally would.

$$(-1)^{Sign} \times Significand_{unsigned} \times 2^{Exp - Bias}$$

where $Bias$ = 127.
There are no denormalized numbers.

| Exponent | Significand | Value |
|---|---|---|
| Largest | Zero | ±Infinity |
| Largest | Nonzero | NaN |

We walk through one way of representing 3 in this scheme, using a Significand of 0b00...011 = 3

$(-1)^0 \times 3 \times 2^{127 - Bias} = 1 \times 3 \times 2^0 = 3$ **Sign**: 0, **Exponent**: 127 = 0x7F, **Significand**: 3 = 0x000003

Answer the following questions comparing the standard IEEE Floating Point and your new scheme (let's call it Bloating Point).

1) Bloating Point represents a larger amount of unique numeric values than Floating Point. T [✓ F]

2) 0x3D80001E is a valid representation of 1.875 in Bloating Point. (Hint: 1.875 = 15/8 [✓ T] F

3) There exists a 32-bit number whose Floating Point value and Bloating Point value are the same. [✓ T] F

4) How many Bloating Point numbers evaluate to $+2^{-124}$?

| | |
|---|---|
| $1 * 2^{-124}$ | $4 * 2^{-126}$ |
| $2 * 2^{-125}$ | $8 * 2^{-127}$ |

4

5) What is the smallest positive number divisible by 5 that Bloating Point cannot represent? You may leave all or part of your answer as a power of 2.

The smallest positive Bloating Point number divisible by 5 involves a significand of ~ all 1s, since that is when we lose precision. We start our search with a significand of all 1s, which is $2^{23}$- 1. The closest number above this number that is divisible by 5 is $2^{23}$+2 (using modular arithmetic / finding a pattern). However, $2^{23}$+2 can be represented in Bloating Point: $(2^{22} + 1) * 2^1$ . Thus, we find the next number divisible by 5 is $2^{23}$+7, which can't be represented in Bloating Point since it's not even.

$2^{23}+7$

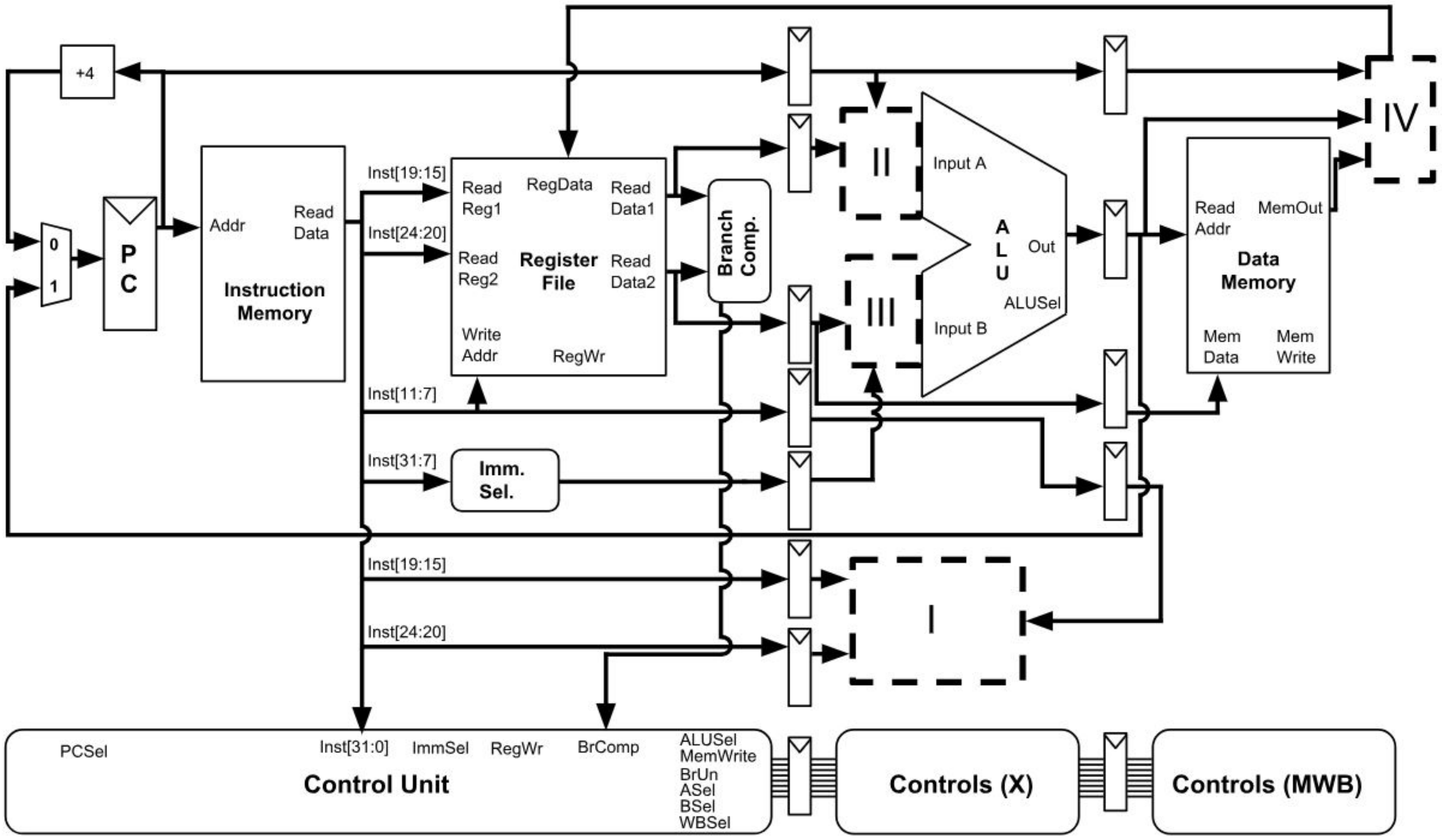
## Question 5: DamonPath (8 pts)

Below we have implemented 3-stage CPU with the stages IFD, EX, and MWB. We're interested in implementing forwarding from the output of the MWB stage to the input of the ALU in the EX stage.This datapath should still implement the regular RISC-V instruction set as well as forwarding.

Make sure to read through all parts of the question (I, II, III, IV) as some variables may defined in different parts.

**Assume that the control values generated by the control unit are driving their respective control inputs in the datapath.**

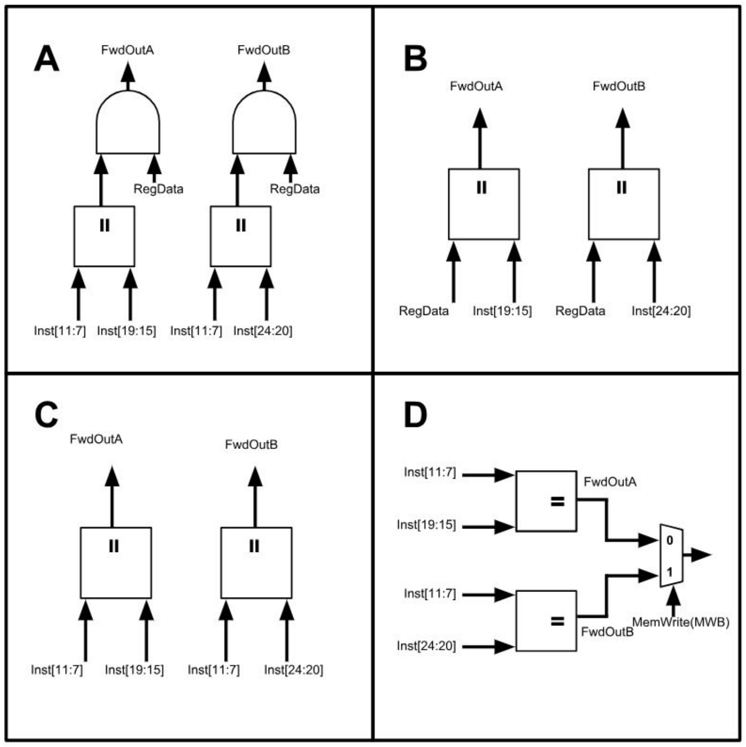
**(Note: In cases that may be ambiguous, we have marked certain values with the stage that they come from. For example, if we write ASel(X) this means the ASel coming from Controls(X) unit.)**



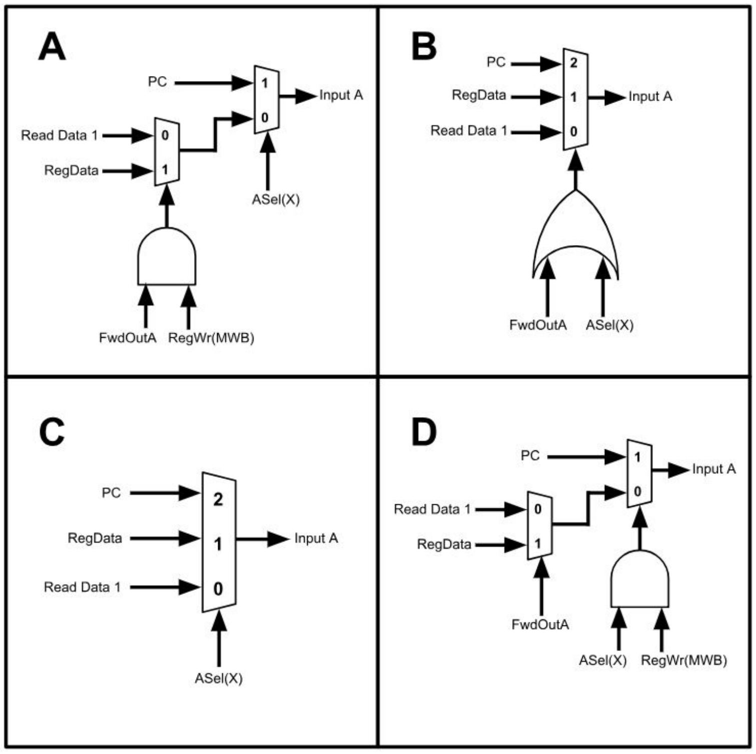
SID: ________________

Choose the correct implementation for I - IV from the options below:

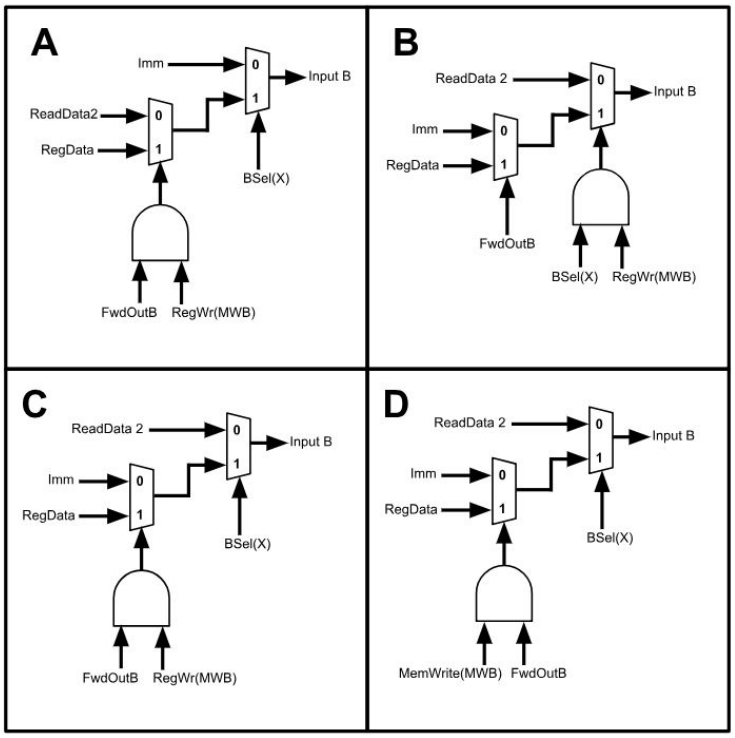
I. C



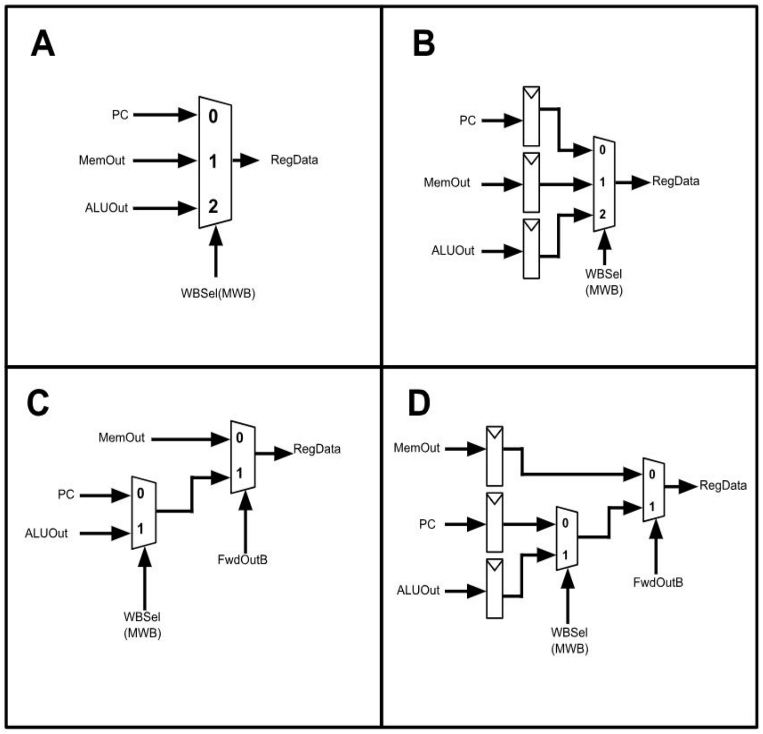
II. A



III. A



IV. A



## Question 6: Cache These Hands (16 pts)

A machine with a 19 bit address space has a single 256 B cache. The cache is 4-way set associative with 8 total entries.

1. Determine the number of bits in Tag, Index, and Offset fields for an address on this machine.

Tag: 13 Index: 1 Offset: 5

The following piece of code is executed on the aforementioned machine. This code computes an outer product of a **N x 1** vector A and a **1 x N** vector B, placing the result in a **N x N** matrix C. Use this code to answer the follow questions about the hit rate the code was produced.

For all questions assume the following:

- sizeof (double) == 8
- A = 0x10000
- B = 0x20000
- C = 0x30000
- The cache begins cold before each question.
- Code is executed from left to right.

```
#define N 16

void outer_product (double *A, double *B, double *C) {
      for (int i = 0; i < N; i++) {
                  for (int j` = 0; j < N; j++) {
                              C [j + i * N] = A[ i ] * B[ j ];
                  }
      }
}
```

2. What is the hit rate for executing this code if it uses LRU replacement and is a write back cache with write allocate on a miss? Fill in all blanks for credit.

**HR for accesses to A: 63/64**

**HR for accesses to B: 27/32**

**HR for accesses to C: 3/4**

**OVERALL HR: ⅓ * 63/64 + ⅓ * 27/32 + ⅓ * ¾ = 21 / 64 + 9 / 32 + 1/4 = 55/64**

**Explanation: We start by attempting to find a pattern when i = 0. Accesses occur left to right so first A is read, then B, and then C. On first iteration all 3 elements will miss and all 3 blocks will be loaded into the cache because the cache is 4 way set associative. Since each block is 32 Bytes in size and each access is 8 Bytes (1 double), the all access will hit giving this result:**
**i = 0, j = 0**
**M M M**
**H H H**
**H H H**
**H H H**

**where the first access is always A, the second B and the third C. Then when j = 4, B and C will move to a new block but A is still in the cache, so it will hit. Otherwise the pattern is the same**

**i = 0, j = 4**
**H M M**
**H H H**
**H H H**
**H H H**

**Then when j = 8, B and C once again enter set 0. B will fit but then that set will be full. Since A keeps being accessed, B's old value will be evicted. So the pattern will remain for j = 8. However when j = 12 B's old value will not be evicted.**

**i = 0, j = 8**
**H M M**
**H H H**
**H H H**
**H H H**

**i = 0, j = 12**

**When i = 1 only two notable changes occur. A will still be loaded in, because this will not be a miss again until i is divisible by 4 and B[4] and B[12] will still be in the cache.**

**This means that from i = 1 to i = 3**

**A will not miss, B will miss only twice per iteration (6 times total) and C will maintain its current hit rate. When i = 4 then the pattern will repeat, although it switches sets (B[0] and B[8] will remain in the cache starting from i = 5). As a result the overall hit rate for each part is:**

**A has 1 miss, so 63/64**
**B has 10 misses so 54/64 = 27/32**
**C misses one in every 4 access, so 3/4**

3. What is the hit rate for executing this code if it uses LRU replacement and is a write back cache with no write allocate on a miss? Fill in all blanks for credit.

**HR for accesses to A: 63/64**

**HR for accesses to B: 63/64**

**HR for accesses to C: 0**

**OVERALL HR: 63/64**

**Explanation: There are many ways to do this question. The easiest barely looks at the access patterns. Since C is a different address from A and B, no part of C will ever enter the cache. The total size of the cache is 256 B, A contains 128 B and B contains 128 B so A and B both fill the cache. Each element of A and each element of B is accessed 16 times, so each Vector is accessed 256 times with only 4 compulsory misses.**

**As a result,**

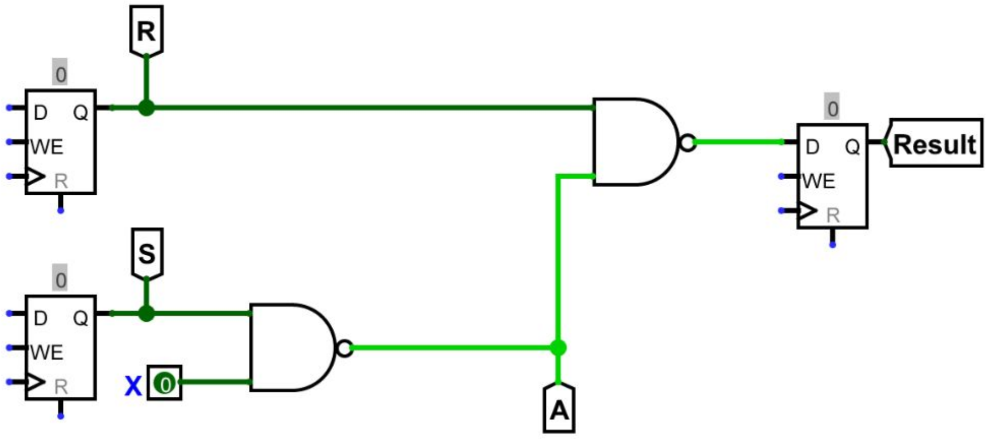
**A = 252/256 = 63/64**
**B = 252/256 = 63/64**
**C = 0**

SID: ________________

## Question 7: NANDerson (16 pts)

Part 1: We have three inputs: R, S, and X. R and S are the outputs of the registers shown below, while X appears **instantaneously** at the rising edge of the clock. We also know the following:

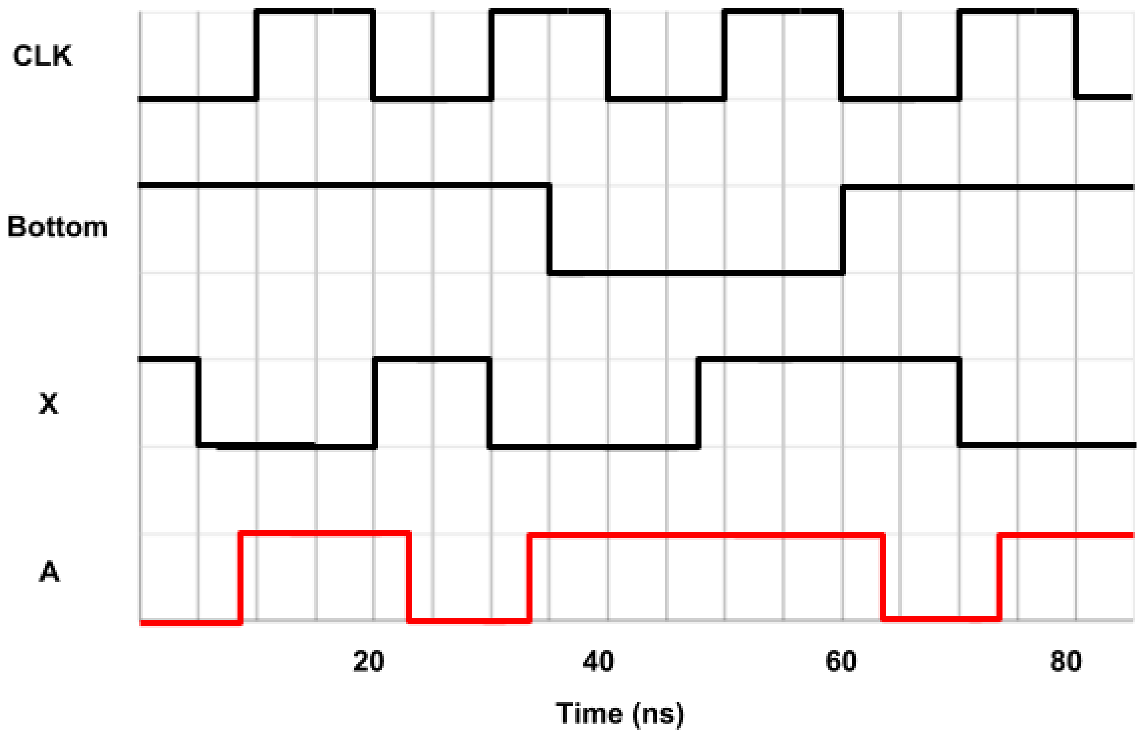
- ➔ CLK→ q = 5ns
- ➔ Setup Time = 5ns
- ➔ NAND Gate Delay = 3ns
- ➔ Assume that hold time is negligible



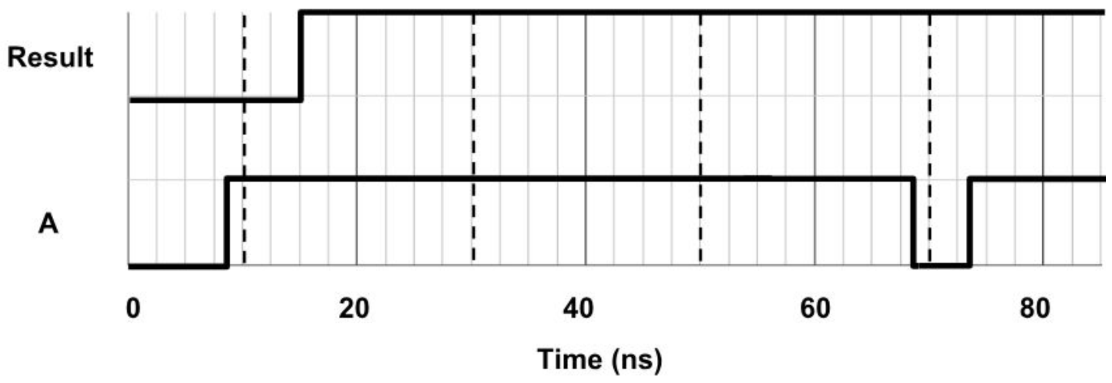
a) Given the following waveform diagram, list all the times that A changes its value. For example, if A flipped at 20, 40, and 60 ns the answer would be "20, 40, 60"

Anything drawn will be considered as scratch work, and **may** be used for partial credit. (The dotted lines show the rising edge, and each tick is 2.5ns)

Answer: 8, 23, 33, 63, 73

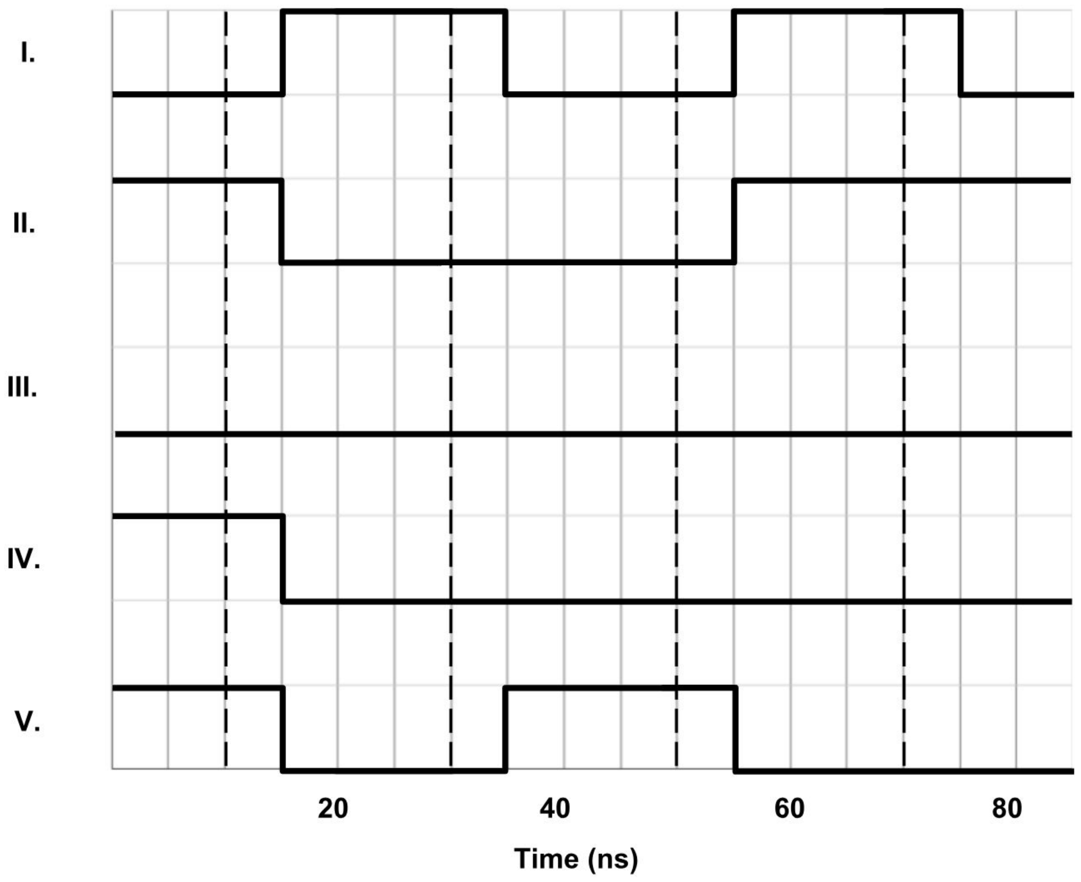


b) Now, we've generated an arbitrary A and Result waveform shown below.



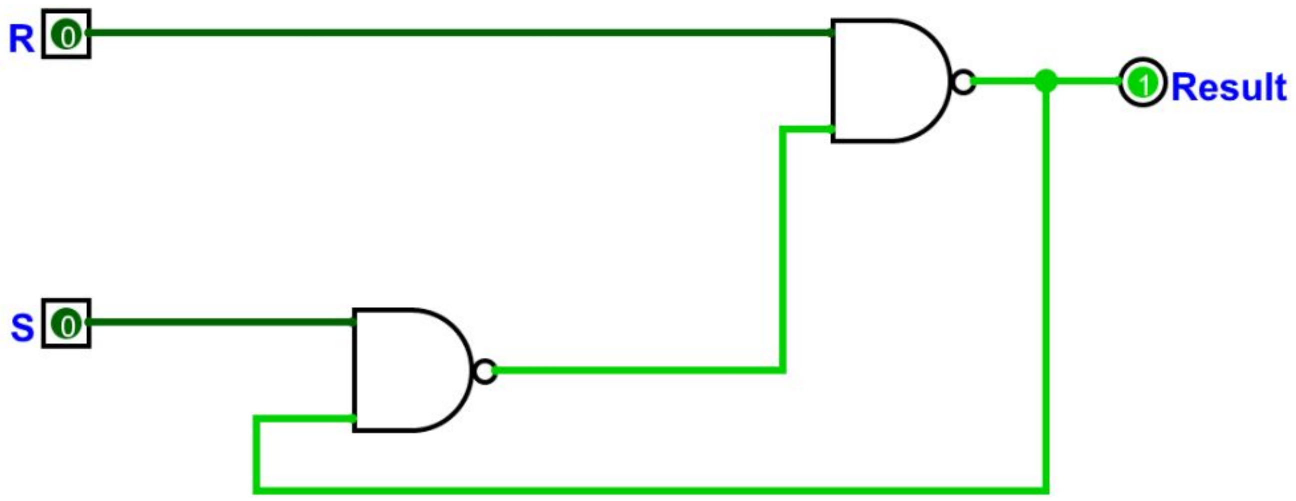
Choose all of the answers below that may be the waveform for **R** such that the correct **Result** waveform is output.

I II III IV V



SID: _______________

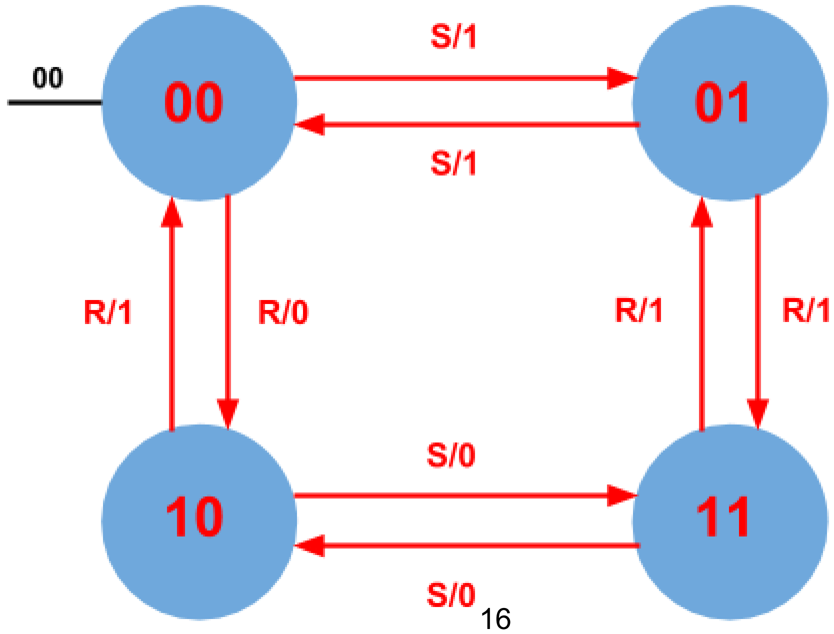
Part 2: We now connect the "X" output to the output of our second NAND gate with the hope of making something useful. The corresponding diagram is shown below.



a) In order to determine the functionality of this circuit, we decide to model it as an FSM. In this FSM, **we must flip one of either R or S (and not both) in each time step.**

Represent this as an FSM using the fewest states below. Each transition should be labeled with the bit that you're flipping, and the output of the FSM where the output is "Result".
It should be of the form: `<R or S> / <Result>.` (e.g. `"R/1"`).

We arbitrarily start at state "00". The first bit represents R and the second bit represents S.



SID: ________________

## Question 8: Mr. MOESI (6 pts)

For this question you will be asked to determine which cache coherence scheme(s) can fulfill tasks efficiently based upon assumptions of what task our machine must perform. For each question there will be two parts:

- **Expected Behavior**: the behavior that your scheme must perform efficiently.
- **Necessary Behavior**: the behavior that doesn't need to be performed efficiently but must be supported.

You should select **all** schemes that can handle the expected behavior with maximum efficiency.

For all scenarios, we have the following assumptions:

- The machine has multiple cores
- **Writing** to memory is very very slow (a performance bottleneck)

If this is unclear, consider an example. If the expected behavior involves writing and **MSI** and **MOESI** can do fewer writes to memory than **MESI**, then you would select **MSI** and **MOESI** and not **MESI**.

1. **Expected Behavior:** Processing completely read only data.
   **Necessary Behavior:** Nothing additional.

   [✓ Ⓐ **MSI**] [✓ Ⓑ **MESI**] [✓ Ⓒ **MOESI**]

2. **Expected Behavior:** A single program with threads for different purposes. A single thread will write in the information about a user. Then after this write, the other three threads will in parallel perform different computations based upon the user's data (each written to different, unique location).
   **Necessary Behavior:** Writing to a final shared location may be necessary to accumulate results once all threads have completed.

   Ⓐ **MSI** Ⓑ **MESI** [✓ Ⓒ **MOESI**]

3. **Expected Behavior:** Processing a unique program on each core (with its own memory space) and quick reading from memory shared by programs.
   **Necessary Behavior:** Writing to data that is shared across programs.

   Ⓐ **MSI** [✓ Ⓑ **MESI**] [✓ Ⓒ **MOESI**]

## Question 9: SIMD Once Told Me (10 pts)

In lecture we showed you how to use Intel Intrinsics to perform an efficient Matrix Multiplication. For this question you are going to perform a similar operation, an inner product between an **m x n** matrix **A** and an **n x 1** vector **b**. This formula can be expressed as:

$$\mathbf{C_i} = \sum_{j=1}^{n} A_{i,j}\, b_j$$

To make this question more interesting, rather than working with the 128 bit registers we showed in lecture, we will instead work with 256 bit registers. You can select from the following SIMD functions, each of which has a brief explanation.

| | |
|---|---|
| `__m256d _mm256_setzero_pd ()` | Produce a `__m256d` with all 0s. |
| `__m256d _mm256_loadu_pd (double *)` | Loads the doubles at the ptr's address into a `__m256d`. |
| `__m256d _mm256_load_pd (double *)` | Loads the doubles at the ptr's address into a `__m256d`. This is faster, but the ptr passed in must always be divisible by 32 (or else a segfault will occur). |
| `void _mm256_storeu_pd (double *, __m256d)` | Stores the contents of the `__m256d` in the memory location pointed to. |
| `void _mm256_store_pd (double *, __m256d)` | Stores the contents of the `__m256d` in the memory location pointed to. This is faster, but the ptr passed in must always be divisible by 32 (or else a segfault will occur). |
| `__m256d _mm256_fmadd_pd (__m256d A, __m256d B, __m256d C)` | Returns `(A * B) + C` by performing operations on packed doubles. |

You may assume that all pointers passed into the function and any stack variables allocated are at addresses divisible by 32. You may also assume **`sizeof (double) = 8`.** Fill in the function below. You must use the **fastest load and store** whenever possible.

/* Computes an inner product between **SM**, which is a **M x N** matrix, and **SV**, which is
 * is a **N x 1** vector. This result is stored in **D**, which is a **M x 1** vector. */

```
void inner_product_simd (double *d, double *sm, double *sv, unsigned m, unsigned n) {
      __m256d matrix_part;
      __m256d vector_part;
      __m256d total;
      double arr[4];
      for (int i = 0; i < m; i += 1) {
            total = _mm256_setzero_pd ();
            for (int j = 0; j < n / 4 * 4; j += 4) {
                  if ((i * n + j) % 4 == 0) {  // Check if the address is aligned.
                        matrix_part = _mm256_load_pd (sm + i * n + j);
                  } else {
                        matrix_part = _mm256_loadu_pd (sm + i * n + j);
                  }
                  vector_part =  _mm256_load_pd (sv + j);
                  total = _mm256_fmadd_pd (matrix_part, vector_part, total);
            }
           _mm256_store_pd (arr, total);
            d[i] = arr[0] + arr[1] + arr[2] + arr[3];
      }
      if (n / 4 * 4 != n) {  // Check if there is a tail case.
            for (int i = 0; i < m; i += 1) {
                  for (int j = n / 4 * 4; j < n; j += 1) {
                        d[i] += sm[i * n + j] * sv[j];
                   }
            }
      }
}
```

SID: ________________

## Question 10: Hashtag Pragma (10 pts)

In OMP we have critical sections which need to be resolved via atomic instructions in assembly. When trying to execute this OpenMP code in multiple threads:

```
#pragma omp critical
{
      x++
}
```

a student produces this assembly to implement it:

```
try:  lw t0 0(s0) #load in the value of x from the address
      addi t2 t0 1
      amoswap.w.aq t1, t2, (s0)
      bne t1 t2 try #check if the value of x was the one previously loaded
End:
```

1. This code DOES NOT ensure the critical section works properly. Select **all of the following** options that accurately describes the code the student produced.
   - Ⓐ The code increments x every time it attempts store. This problem can be solved by restoring x to the value of `t0` before trying again.
   - [✓ Ⓑ] Amoswap successfully changes the value at the address of `s0` and returns the old value in memory in a single, uninterrupted operation.
   - [✓ Ⓒ] If instead of a `lw`, `addi`, and `amoswap`, a single `amoadd` instruction was used to add 1 to x, then the code would work properly.
   - [✓ Ⓓ] The code can still work properly in some cases.

When working with fully associative caches, you learned that a **cache will search each block in parallel**. Let's envision how this could be simulated in software using openmp. Imagine you have a cache struct that looks like this:

```
typedef struct cache {
      uint32_t *blocks;
      unsigned num_blocks;
} cache;
```

Additionally imagine a cache block stores both data and metadata in a **single 32 bit value** and does so **big endian**. It has the following details:

- Bit 31 (MSB): Valid Bit
- Bit 30: Dirty Bit
- Bits 29 - 16 (14 total bits): Tag
- Bits 15 - 0 (16 total bits): Data

SID: ________________

2. Complete the function for handling the **parallel search of each cache block**. It takes in a cache struct, a pointer to data (where data should be stored), and a tag. It should update two shared variables; the first a boolean value indicating if the data was stored in the cache; the second a pointer whose contents should be updated with the data. For this portion **assume we will have as many hardware threads as we do cache blocks.**

```
bool search_cache (cache *cache, uint16_t *data_ptr, uint16_t tag) {
      bool found = false;
      unsigned num_blocks = cache->num_blocks;
      #pragma omp parallel for
      {
      for (int i = 0; i < num_blocks; i++) {
            uint32_t contents = cache->blocks[i];
            uint32_t valid = contents & (0x80000000);
            if (valid) {
                  uint32_t block_tag = (contents >> 16) & (0x3FFF);
                  if (block_tag == tag) {
                        uint32_t data = contents & (0xFFFF);
                        #pragma omp critical
                        {
                              found = true;
                              *data_ptr = data;
                        }
                  }
            }
      }
      }
      return found;
}
```

3. Now let's **assume we may have fewer hardware threads than cache blocks**. Which of the follow statements about the code above represent ways the code can be optimized? Select **all** that are true:

- Ⓐ When a thread finds that it contains the data, it can break out of the loop.
- Ⓑ As we increase our number of threads we will see little to no speedup because a critical section is included.
- [✓ Ⓒ] If our cache works properly, no matter how many threads exist in hardware we can completely remove the critical pragma.
- Ⓓ If the number of hardware threads is near the number of cache blocks, then we run the risk of having false sharing occur in our code.

SID: ________________

## Question 11: TL;Br (Too Long; But read) (10 pts)

Consider a machine with 4 KiB pages, a 32-bit virtual address space with 256 MiB of DRAM for main memory. It has a single level of page table and a TLB containing 4 entries which is fully associative.

1. How many bits are there for the VPN? How many bits are there for the offset of the virtual address?

VPN: **20** Offset: **12**

2. How many bits are there for the PPN? How many bits are there for the offset of the physical address?

PPN: **16** Offset: **12**

Next let's identify how translations of various **virtual addresses** will be resolved. For each translation **identify if the result is a TLB hit, a Page Table Hit, or a Page Fault**. **Assume each access restarts from the original layout of the TLB and Page Table**. Assume **any page table entries not shown have a valid bit of 0.**

**TLB**

| VPN | PPN |
|---|---|
| 0x6 | 0x15 |
| 0x4 | 0x31 |

**PAGE TABLE**

| VPN | Valid Bit | PPN |
|---|---|---|
| 0x0 | 1 | 0x3 |
| 0x1 | 1 | 0x7 |
| ………………………………... | ………………………………... | ………………………………... |
| 0x4 | 1 | 0x31 |
| 0x5 | 0 | 0x3 |
| 0x6 | 1 | 0x15 |
| 0x7 | 1 | 0x11 |
| ………………………………... | ………………………………... | ………………………………... |

3. 0x7ABC
   Ⓐ TLB Hit [✓ Ⓑ Page Table Hit] Ⓒ Page Fault

4. 0x3000
   Ⓐ TLB Hit Ⓑ Page Table Hit [✓ Ⓒ Page Fault]

5. 0x6423
   [✓ Ⓐ TLB Hit] Ⓑ Page Table Hit Ⓒ Page Fault

6. 0x5221
   Ⓐ TLB Hit Ⓑ Page Table Hit [✓ Ⓒ Page Fault]

7. 0x20282
   Ⓐ TLB Hit Ⓑ Page Table Hit [✓ Ⓒ Page Fault]

Finally let's consider how the TLB and Page Table change (or don't) as a result of memory accesses. Assume we have **256 B Pages, 2 TLB entries, 4 physical pages and 8 virtual pages in our machine**. These initially appear as shown below. After each access fill in the new contents of the TLB and PT. Assume we evict from main memory and the TLB by evicting the smallest VPN. **Once again assume each access restarts from the original layout of the TLB and Page Table**. If a row in either the TLB or the Page Table does not change from the original, you can either fill it in again or **leave it blank** in the same location.

**TLB**

| VPN | PPN |
|---|---|
| 0x1 | 0x2 |
| 0x5 | 0x3 |

**PAGE TABLE**

| VPN | Valid Bit | PPN |
|---|---|---|
| 0x0 | 0 | 0x3 |
| 0x1 | 1 | 0x2 |
| 0x2 | 0 | 0x1 |
| 0x3 | 1 | 0x0 |
| 0x4 | 0 | 0x0 |
| 0x5 | 1 | 0x3 |
| 0x6 | 1 | 0x1 |
| 0x7 | 0 | 0x2 |

**8. 0x608 (All changes are in red. Everything else remained the same)**

**TLB**

| VPN | PPN |
|---|---|
| **0x6** | **0x1** |
| | |

**PAGE TABLE**

| VPN | Valid Bit | PPN |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

**9. 0x2F4**

**TLB**

| VPN | PPN |
|---|---|
| **0x2** | **0x2** |
| | |

**PAGE TABLE**

| VPN | Valid Bit | PPN |
|---|---|---|
| | | |
| **0x1** | **0** | **X (doesn't matter)** |
| **0x2** | **1** | **0x2** |
| | | |
| | | |
| | | |
| | | |
| | | |

SID: ________________

## Question 12: Go Ham[ming] (10 pts)

We will consider a system that works with 7-bit codewords encoded with Single Error Correction (SEC) parity bits. The Hamming code table is provided for your reference to the right.

| Bit | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Data | P1 | P2 | D1 | P4 | D2 | D3 | D4 |
| P1 | X | | X | | X | | X |
| P2 | | X | X | | | X | X |
| P4 | | | | X | X | X | X |

Suppose we read a codeword of 0b1010101 from our disk. What are the data bits that were encoded?

$p_1$ = 0, $p_2$ = 0, and $p_4$ = 0 so there is no error

0b 1101

Suppose two pesky alpha particles struck our disk (darn!) and changed the first two bits of our codeword so that now, we retrieve 0b**01**10101 from our disk. What data bits will be interpreted?

$p_1$ = 1, $p_2$ = 1, and $p_4$ = 0 so there is an error in the 1 + 2 = 3rd bit

0b 0101

What is the Hamming distance between two valid 7-bit SEC codewords? Hint: you can solve this with or without the previous two codewords.

SEC scheme codewords always have a Hamming distance of three. You could also compare the two previous valid and corrected codewords 0b1010101 and 0b0100101 which do have a Hamming distance of 3.

3

We will now add in a fourth parity bit $\mathbf{p_8}$ to allow our disk to also detect two errors (DED). We read 0b0110101$\mathbf{p_8}$ from our disk, where $\mathbf{p_8}$ is the DED parity bit for the 7-bit string. What **must** be the value of $\mathbf{p_8}$ such that our 8-bit codeword contains a single correctable error?

We know from the previous questions that 0b0110101 has a nonzero parity check. If there was a single error, then $p_8$ would have to be 1 in order for the parity of the entire 8-bit codeword to be 1 and confirm a single error.

$\mathbf{p_8}$ = [✓ 1] 0

What is the Hamming distance between two valid 8-bit DED codewords?

You can compare any two valid 8-bit codewords (i.e. from above) or know that DED codewords have H.Dist of 4

4

Suppose we have a 5-disk RAID 3 system. Compared to a 5-disk RAID 5 system that is **byte striped**, fill in all the bubbles for the data request patterns / characteristics where RAID 5 **strictly outperforms** RAID 3.

| | | |
|---|---|---|
| Ⓐ long sequential reads | Ⓑ long sequential writes | Ⓒ recovering from 1 disk failure |
| Ⓓ small random reads | [✓ Ⓔ] small random writes | Ⓕ capacity |

Consider adding a DMA controller to facilitate bringing in data from our external disk/devices into main memory. The DMA controller can either make transfers in **burst mode**, where it stalls the CPU and writes 32B of data per clock cycle, or **cycle-stealing mode** where it doesn't stall the CPU (i.e. it runs normally) and writes 4B of data per clock cycle. The CPU returns to normal instruction execution after the transfer is completed. Which mode would be better for the following situations? Assume every instruction uses 1 clock cycle with no stalls.

| | | |
|---|---|---|
| • Dealing with page fault; the page size is 4KiB | [✓ Ⓐ] burst | Ⓑ cycle stealing |
| • Playing a 200MB 5-minute video | [✓ Ⓐ] burst | Ⓑ cycle stealing |
| • Processing 10 keyboard strokes per second | Ⓐ burst | [✓ Ⓑ] cycle stealing |

## Question 13: South Spark (6 pts)

In this question you will write Spark to find the mode of a list of values and how often it occurs. As a refresher the mode is the number that appears most often. If there is a tie you can select any of the options. **Fill in the blanks for the Python code below.** Use the following Spark Python functions when necessary: **map**, **flatmap**, **reduce**, **reduceByKey**. Here is a sample input and output:

#Input: [**1**, 2, **1**, 2, 3, 4, 5, 6, 4, 2, **1**, 3, 3, **1**, **1**, 2, 2, **1**]
#Output: (1, 6)

```
def output_data (val):
        return  (val, 1)

def compute_count (a, b):
        return  a + b

def find_max_occurrence (a, b):
        if a[1] > b[1]:
                return a
        return b

#values = list (numbers)
modeData = sc.parallelize (values)

modeData.map (output_data)
            .reduceByKey (compute_count)
            .reduce (find_max_occurrence)
```

## Question 14: CALLs well that ends well (6 pts)

For each question select all options which correctly answers the question. The options are

- A. Compiler
- B. Assembler
- C. Linker
- D. Loader
- E. None of the other options

1. Output includes pseudoinstructions [✓ Ⓐ] Ⓑ Ⓒ Ⓓ Ⓔ
2. Is itself a program [✓ Ⓐ] [✓ Ⓑ] [✓ Ⓒ] [✓ Ⓓ] Ⓔ
3. Determines the virtual address space for the static section Ⓐ Ⓑ [✓ Ⓒ] Ⓓ Ⓔ
4. Initializes the physical address space for the static section Ⓐ Ⓑ Ⓒ [✓ Ⓓ] Ⓔ
5. Resolves pseudoinstructions Ⓐ [✓ Ⓑ] Ⓒ Ⓓ Ⓔ
6. Fills in the immediate for jump instructions Ⓐ [✓ Ⓑ] [✓ Ⓒ] Ⓓ Ⓔ