Exam Review

Tips: Anything from the assigned readings is fair game. Content in the slide decks and assigned homework/laboratory assignments will have a somewhat higher priority.

1. T
2. F
3. T
4. T
5. F (it also includes stack, general registers, file descriptors, pc, and environment variables)
6. F (The Instruction Set Architecture (ISA) defines the set of instructions that a CPU can understand and execute. The microarchitecture, on the other hand, describes the specific implementation details of how that ISA is realized in a particular processor design (e.g., pipeline structure, cache organization, execution units).
7. F (While their primary function is graphics rendering, modern GPUs are highly parallel processors capable of performing a wide range of general-purpose computations, a concept known as GPGPU (General-Purpose computing on Graphics Processing Units). This makes them suitable for tasks like scientific simulations, machine learning, and cryptocurrency mining.)
8. c) 2^{-4} In an unsigned fixed-point number with an I-bit integer part and an F-bit fractional part, the total bits are I+F. The value is interpreted as the integer value divided by 2F, so the scaling factor is 2^{-F}. Here, F=4, so the factor is 2^{-4}.
9. c) Performing A + (two's complement of B). Subtraction A−B is implemented as A+(−B). In two's complement, −B is represented by the two's complement of B.
10. b) Division algorithms are inherently iterative and can be more complex to control. Hardware division typically involves a sequence of subtractions and shifts within a loop, requiring more complex control logic compared to common multiplication algorithms like array multipliers or Booth's algorithm implementations
11. B.) Its logic is implemented using fixed logic gates and flip-flops, making it faster but harder to modify.
12. b) Connect slower peripheral devices to the system.
13. a.)The need for instructions and data to share a single bus for fetching and storing. (Harvard architecture uses dual buses)
14. d.)Data-Level Parallelism (SIMD)
15. 64 bits, 4 bits
16. Combinational logic, clock signals
17. Parallel flows
18. Arithmetic Logic Unit
19. Access

20. Clock
21. EEPROM (Electrically Erasable Programmable ROM) or Flash Memory
22. Exceptoin
23. Running, stopped, sleeping (user input, I/O needs to complete), zombie (completed execution, but not cleaned up by parent), idle, or orphan (abandoned by parent process, no wait on child termination)
24. 0 to 2^(n-1), 0x400000
25. Privileged instructions
26. /proc (/proc/cpuinfo, /proc/process-id/maps); also /sys since Linux kernel 2.6

0X2F42000

$(-1)^S \times 2^E \times 2^F$

0 $\boxed{010\ 1111\ 0}$100 0001 0000

$S = 0$ $\quad E = 2^1 + 2^2 + 2^3 + 2^4 + 2^6$ $\quad$ bias = 127

$= 6 + 8 + 16 + 64 = 94$ $\quad 127 - 94 = 33$

$F = 2^{-1} + 2^{-7} = 0.5 + 0.0078125 + 1$

$= 1.5078125$

27. $(-1)^0 \times 2^{33} \times 1.5078125 = 12{,}952{,}010{,}752$

4.75 sign bit = 0

$2^2 \Rightarrow$ 0000 0100 So

0000 0100.00 $\times 2^0$

1.00 $\times 2^2$

0.75 × 2 = 1.5
0.5 × 2 = 1.0
0.0 × 2 = 0

1
1
0

bias 127 + 2 = 129

thus

0 | 1000 0001

0 0 1 1 0 0 0 0 0000...

Add bits from normalization 1st then fraction

28.

29. Here are the C expressions for each value of K:

a.) K = 9 `x << 3 + x` (This is equivalent to x * 8 + x = x * 9)

b.) K = -15 `x - (x << 4)` (This is equivalent to x - (x * 16) = x - 16x = -15x)

c.) K = 72 `(x << 6) + (x << 3)` (This is equivalent to x * 64 + x * 8 = x * (64 + 8) = x * 72)

d.) K = 80 `(x << 6) + (x << 4)` (This is equivalent to x * 64 + x * 16 = x * (64 + 16) = x * 80)

30. “A collection of intertwined hardware and systems software that must cooperate in order to achieve the ultimate goal of running application programs”

31.The processor state

The format of the instructions

The effect each instruction will have on the processor state

32. SRAM (Static Random-Access Memory) and DRAM (Dynamic Random-Access Memory) are two primary types of volatile random-access memory used in computers and other electronic devices. While both provide temporary data storage that is lost when power is removed, they differ significantly in their internal design, performance characteristics, and typical applications.

Feature	SRAM (Static Random-Access Memory)	DRAM (Dynamic Random-Access Memory)
Data Storage	Uses latches (typically 6 transistors) to store each bit. The latch holds the data as long as power is supplied without needing to be refreshed.	Uses a capacitor and a transistor to store each bit as a charge in the capacitor. The charge leaks over time, requiring periodic refreshing to retain the data.
Speed	Faster access times due to not needing to be refreshed.	Slower access times because of the refresh cycles.
Complexity	More complex internal structure per bit (more transistors).	Simpler internal structure per bit (fewer components).
Density	Lower density (less memory capacity per unit area) due to more components per cell.	Higher density (more memory capacity per unit area) due to fewer components per cell.
Cost	More expensive to manufacture per bit due to complexity.	Less expensive to manufacture per bit due to simplicity and higher density.

Power Consumption	Generally consumes less power when idle. Can consume more power when actively being read from or written to.	Consumes more power due to the constant refreshing required, even when idle.
Applications	Typically used for smaller, faster memory caches (like CPU cache - L1, L2, L3), register files, and in applications where speed is critical.	Typically used for the main system memory (RAM) in computers, graphics cards, and other devices where high capacity is needed at a lower cost.
Refresh	Does not require periodic refreshing.	Requires periodic refreshing to maintain data.
Volatility	Volatile (data is lost when power is removed).	Volatile (data is lost when power is removed).

SRAM prioritizes speed and performance at a higher cost and lower density, making it suitable for small, fast caches. DRAM prioritizes higher density and lower cost, making it the dominant technology for the larger main memory found in most computing systems, despite being slower and requiring constant refreshing.

33.

```
long transform(long p, long q, long r) {
  // Parameters: p in %rdi, q in %rsi, r in %rdx
  // Return value in %rax
    // (p - (q + r)) ^ ((q + r) >> 63).

  // Assembly line 1: addq %rdx, %rsi
  // %rsi = %rsi + %rdx  (q = q + r)
  q = q + r;
```

```
// Assembly line 2: subq %rsi, %rdi
// %rdi = %rdi - %rsi  (p = p - (q+r))
p = p - q; // q already holds q+r

// Assembly line 3: movq %rdi, %rax
// %rax = %rdi  (Initialize return value with p - (q+r))
long result = p; // p holds p - (q+r)

// Assembly line 4: movq %rsi, %rcx
// %rcx = %rsi  (Copy q+r into %rcx)
long temp_rcx = q; // q holds q+r

// Assembly line 5: sarq $63, %rcx
// %rcx = %rcx >> 63 (Arithmetic right shift by 63 bits)
// This operation extracts the sign bit of the 64-bit value in %rcx.
// If %rcx was non-negative, result is 0. If %rcx was negative, result is -1 (all 1s).
temp_rcx = temp_rcx >> 63; // Using C's arithmetic right shift for signed types

// Assembly line 6: xorq %rcx, %rax
// %rax = %rax ^ %rcx (XOR the current result with the sign value)
result = result ^ temp_rcx;

// Assembly line 7: ret
// The value in %rax is returned.

return result;
```

}

34. See below

```
.data
    .align 8             # Ensure the variable is 8-byte aligned
calculation_limit:
    .quad   10           # The value for the 'limit' parameter
cumulative_calc:
    movq    $0, %rax        # Initialize cumulative sum = 0
    movq    $1, %rcx        # Initialize counter = 1
    cmpq    %rcx, %rdi      # Compare limit with counter (1)
    jl      .Lend_calc      # If limit < 1, exit loop

.Lcalc_loop:
    movq    %rcx, %rdx      # Copy counter to %rdx
    andq    $1, %rdx        # Check if counter is odd (%rdx =
counter & 1)
    testq   %rdx, %rdx      # Test the result of the AND
    jnz     .Lcall_helper   # If odd (result != 0), call helper

    ; Even case
    leaq    (%rcx,%rcx), %rdx # Calculate counter * 2 (%rdx =
counter * 2)
    addq    %rdx, %rax      # Add to sum (%rax = %rax + counter
* 2)
    jmp     .Lcontinue_loop   # Continue to next iteration

.Lcall_helper:
    # Odd case - call helper
```

```
    pushq   %rcx            # Save %rcx (counter) - caller-save
    movq    %rcx, %rdi      # Move counter to %rdi for function
call
    call    square_and_add_5 # Call helper function
    addq    %rax, %rax      # Add helper's result to cumulative
sum (%rax = %rax + result)
    popq    %rcx            # Restore %rcx

.Lcontinue_loop:
    incq    %rcx            # Increment counter
    cmpq    %rdi, %rcx      # Compare counter with limit
    jle     .Lcalc_loop     # If counter <= limit, continue loop

.Lend_calc:
    ret                     # Return sum in %rax

square_and_add_5:
    movq    %rdi, %rax      # Move parameter val to %rax
    imulq   %rax, %rax      # %rax = %rax * %rax (val * val)
    addq    $5, %rax        # %rax = %rax + 5 (val * val + 5)
    ret                     # Return result in %rax
```

35. See below

```
.section .rodata   # Read-only data section
    .align 8       # Align the jump table
category_table:
    .quad .L_is_digit
    .quad .L_is_upper
    .quad .L_is_lower
```

```
    .quad .L_is_other  # Default case

    .text              # Code section
    .globl check_char_category
    .type check_char_category, @function
check_char_category:
    # input_char is in %rdi (lower 8 bits)
    # Derived index will be used for jump table

    movq    %rdi, %rax          # Copy input_char to %rax

    # --- Determine Index ---
    # If char is digit, index = 0
    # If char is upper, index = 1
    # If char is lower, index = 2
    # Otherwise, index = 3 (for the default jump table entry)

    cmpb    $'0', %dil          # Compare char with '0'
    jl      .L_check_upper      # If char < '0', maybe
uppercase?

    cmpb    $'9', %dil          # Compare char with '9'
    jle     .L_is_digit_idx     # If char <= '9', it's a digit
(index 0)

.L_check_upper:
    cmpb    $'A', %dil          # Compare char with 'A'
```

```
    jl      .L_check_lower      # If char < 'A', maybe
lowercase?

    cmpb    $'Z', %dil          # Compare char with 'Z'
    jle     .L_is_upper_idx     # If char <= 'Z', it's uppercase
(index 1)

.L_check_lower:
    cmpb    $'a', %dil          # Compare char with 'a'
    jl      .L_is_other_idx     # If char < 'a', it's neither
(index 3)

    cmpb    $'z', %dil          # Compare char with 'z'
    jle     .L_is_lower_idx     # If char <= 'z', it's lowercase
(index 2)

.L_is_other_idx:
    movq    $3, %rcx            # Set index to 3 (other)
    jmp     .L_do_jump          # Go to jump

.L_is_digit_idx:
    movq    $0, %rcx            # Set index to 0 (digit)
    jmp     .L_do_jump          # Go to jump

.L_is_upper_idx:
    movq    $1, %rcx            # Set index to 1 (upper)
    jmp     .L_do_jump          # Go to jump
```

```
.L_is_lower_idx:
    movq    $2, %rcx            # Set index to 2 (lower)
    ; Fall through to .L_do_jump

.L_do_jump:
    # Jump to table entry: category_table + index * 8
    jmp     *category_table(,%rcx,8) # Jump to the address in
the table

.L_is_digit:
    movq    $0, %rax            # Return code 0
    jmp     .L_end_check

.L_is_upper:
    movq    $1, %rax            # Return code 1
    jmp     .L_end_check

.L_is_lower:
    movq    $2, %rax            # Return code 2
    jmp     .L_end_check

.L_is_other:
    movq    $-1, %rax           # Return code -1

.L_end_check:
    ret                         # Return value in %rax
```

36. The jle (Jump if Less than or Equal) instruction jumps if the Zero Flag (ZF) is set (equal) or the Sign Flag (SF) is not equal to the Overflow Flag (OF) (less than), interpreting the result of the comparison as signed integers.