Real-Time Embedded Systems (ECE-GY 6483)

Homework 2

Saad Zubairi
shz2020

Question 1: Bit Operations and Their Functions

1. $x \& 1 \quad \rightarrow \quad$ (f): Return true if x is odd, false if x is even.
2. $x \& (1 \ll n) \quad \rightarrow \quad$ (c): Return true if th nth bit is set.
3. $x \& \sim (1 \ll n) \quad \rightarrow \quad$ (b): Unset nth bit.
4. $(x \oplus y) < 0 \quad \rightarrow \quad$ (e): Return true iff x and y have opposite signs.
5. $y \oplus ((x \oplus y) \& -(x < y)) \quad \rightarrow \quad$ (d): Return the minimum of x and y.
6. $x \& (x-1) \quad \rightarrow \quad$ (g): Return 0 if x is a power of 2 for $x > 0$.
7. $x \& (x+1) \quad \rightarrow \quad$ (a): Return x without trailing 1s.

Question 2: C Optimizations

(a) Count down to zero, not up to N, in for() loops

1. **Performance factor:** Counting down often allows for a comparison against zero, which can be faster on some architectures due to hardware-level optimizations.
2. **No performance change:** On architectures without special handling for zero comparisons, this optimization might not yield any benefit. Additionally, code readability might suffer.
3. **Improvement:** Trivial, typically saving at most one instruction per loop iteration.

(b) Avoid the % operation

1. **Performance factor:** It is computationally expensive as it involves division. Replacing it with bitwise operations reduces overhead.
2. **No performance change:** On modern processors with optimized division hardware, the difference may be minimal. Additionally, alternate logic to replace % might reduce code clarity and introduce bugs.
3. **Improvement:** Significant for high-iteration loops where % is used frequently, especially with powers of 2.

(c) Use an 8-bit unsigned char for values between 0 and 255

1. **Performance factor:** Using smaller data types can reduce memory usage and improve cache efficiency.
2. **No performance change:** On processors with native word sizes larger than 8 bits, additional instructions might be needed to handle smaller types.
3. **Improvement:** Trivial to moderate, primarily beneficial for memory-constrained systemss.

Question 3: JPL Institutional Coding Standards

(a) Why is recursion not permitted in mission-critical flight software?

1. Predictability: Recursion can result in unpredictable memory usage due to stack growth.
2. Stability: The risk of stack overflow or unbounded recursion makes recursion unsuitable.
3. Traceability: Iterative solutions are easier to analyze and debug.

Thus, Iterative solutions are preferred in mission-critical flight software because they offer better control over execution flow and resource usage.

(b) Why is dynamic memory allocation disallowed after task initialization?

1. Memory Fragmentation: Dynamic memory allocation can lead to fragmentation, making it difficult to allocate large contiguous memory blocks over time.
2. Failure Risks: If the memory allocator fails to find a suitable block, it could cause a crash or undefined behavior during runtime.

Mission-critical systems require deterministic behavior and reliable resource management, making dynamic memory allocation unsuitable after initialization.

Question 4: Signed vs Unsigned in Arithmetic

(a) **Answer: Signed**

(b) **Answer: Unsigned**

(c) **Answer: Unsigned**

(d) **Answer: Signed**

Question 5: CPSR Status Flags After ARM Instructions

(a)

- **N (Negative):** 0
- **Z (Zero):** 0
- **C (Carry):** 1
- **V (Overflow):** 0

(b)

- **N (Negative):** 0
- **Z (Zero):** 0
- **C (Carry):** 1
- **V (Overflow):** 0

(c)

- **N (Negative):** 0
- **Z (Zero):** 0
- **C (Carry):** 1
- **V (Overflow):** 0

(d)

- **N (Negative):** Unchanged
- **Z (Zero):** Unchanged
- **C (Carry):** Unchanged
- **V (Overflow):** Unchanged

(e)

- **N (Negative):** 1
- **Z (Zero):** 0
- **C (Carry):** 0
- **V (Overflow):** 1

Question 6: Euclid Algorithm for GCD

(a) C Algorithm Execution

- Initial values: $a = 54$, $b = 24$
- Execution steps:
  1. $a = 54 - 24 = 30$
  2. $a = 30 - 24 = 6$
  3. $b = 24 - 6 = 18$
  4. $b = 18 - 6 = 12$
  5. $b = 12 - 6 = 6$
  6. $a = b = 6$
- **Result:** $\text{GCD}(54, 24) = 6$

(b) ARM Assembly Without Full Conditional Execution

- Initial values: $r1 = 54$, $r2 = 24$
- Execution steps involve branching and subtraction:
  1. $r1 = 54 - 24 = 30$
  2. $r1 = 30 - 24 = 6$
  3. $r2 = 24 - 6 = 18$
  4. $r2 = 18 - 6 = 12$
  5. $r2 = 12 - 6 = 6$
- **Result:** $\text{GCD}(54, 24) = 6$

(c) ARM Assembly With Full Conditional Execution

- Initial values: $r1 = 54$, $r2 = 24$
- Execution steps with conditional execution:
  1. $r1 = 54 - 24 = 30$
  2. $r1 = 30 - 24 = 6$
  3. $r2 = 24 - 6 = 18$
  4. $r2 = 18 - 6 = 12$
  5. $r2 = 12 - 6 = 6$
- **Result:** $\text{GCD}(54, 24) = 6$

(d) Cycle Comparison

- Without conditional execution: 36 cycles (best case).
  Each iteration involves multiple instructions: CMP, conditional branch (BEQ, BLT), subtraction (SUB), and unconditional branch (B). For each: 6 cycles (best case).
  Total cycles: $6 \times 6 = 36$.
- With conditional execution: 30 cycles (best case).
  Fewer branches are needed due to the use of conditional execution (SUBGT, SUBLT).Each iteration: 5 cycles (best case).
  Total cycles: $6 \times 5 = 30$.
- **Conclusion:** The version with full conditional execution is faster.