Indian Institute of Technology Kharagpur

AUTUMN Semester 2021
COMPUTER SCIENCE AND ENGINEERING

Computer Organization and Architecture

Students: 133 | Model Solutions | Date: 04 October 2021
Total points: 80 | Time: **Opens** 11:55 AM, 04-10-2021
Credit: 30% | **Closes** 1:55 PM, 04-10-2021

INSTRUCTIONS: This is an OPEN-BOOK, OPEN-NOTES test. You may use calculators if required. This question paper has three pages. **Answer all questions**.

Submission of answers: The questions are such that they require either numerical answers or short answers. Please write ONLY THE ANSWERS on a sheet of paper, convert it to **pdf** directly (or by scanning), and submit it on Moodle by **2:10 pm** on Monday, 04 October 2021.

---

1. (**5 points**) Let $t1, $t2 hold two 32-bit integers in two's complement. We want to determine whether a *final* carry (emitted out of the 32nd bit, which is usually discarded) is being generated after addition of these two integers. A register ($t3) should be set to 0 if the final carry bit is 0, else it should be set to 1. Overflow, if any, should be ignored. Write a MIPS code to accomplish this, with *justification why it works correctly*. This can be done using only two lines of codes! (2 + 3)

Solution: The following MIPS code will take of whether or not the final carry-bit is 1:
addu $t3, $t1, $t2
sltu $t3, $t3, $t1 (or *sltu* $t3, $t3, $t2)

Justification: when two binary numbers are added generating a final carry, we claim that the "unsigned sum" (disregarding the final carry) must be smaller than each of the two unsigned operands. Both of these arithmetic operations should therefore be performed in unsigned mode to overrule overflow and to take care of the above issue. To prove this claim, assume that addition of two *n*-bit unsigned integers A and B generates final carry-bit = 1, and let the *n*-bit sum be denoted as S′. Therefore, $A + B = 2^n + S'$. We need to show that S′ is less than both A and B; assume not true $\Rightarrow S' = A + \delta$, $\delta \geq 0$. Hence, $A + B = 2^n + A + \delta \Rightarrow B = 2^n + \delta$; this contradicts the fact the maximum value of *n*-bit unsigned binary number B is $2^n - 1$.

2 (**10 points**). We have a CPU with an 8-bit ALU equipped with an 8-bit integer adder, registers, shift-registers, and other basic control blocks but hardware circuits for multiplier, divider, or for performing bit-wise logical operations (AND, NOT, etc.), or floating-point operations, are **not available**. We need to compute $X = \lfloor((34 \times A) + B)/7\rfloor$ where A, B, X are all integers expressed in *unsigned* 8-bit numbers. In particular, assume that A = 0000 0011, and B = 0001 1001. Show a procedure that computes X with the exact desired result on this CPU while **minimizing** the number of basic operations for these special values of A and B. Report the number of operations that you have used, as well.

Solution: $34 \times A = (2^5 + 2^1) \times 0000\ 0011 = 0110\ 0000 + 0000\ 0110 = 0110\ 0110$
$\Rightarrow (34 \times A) + B = 0110\ 0110 + 0001\ 1001 = 0111\ 1111 = Y$ (say)
So, this needs two left-shift logical operations (one by 5 bits, one by 1 bit), and two additions.
In order to compute $\lfloor Y/7 \rfloor$, we can do the following:
It is easy to verify that Y/7 is equal to the infinite series: [{(Y + Y/8)/8} + Y]/8 + …..;
For approximation, we keep up to three terms as above.
Division by 8 can be performed by right shifting logically three bits (padding 0's on the left).
Thus, Y/8 = 0000 1111; Y + Y/8 = 0111 1111 + 0000 1111 = 1000 1110 (unsigned addition, overflow ignored); (Y + Y/8)/8 = 0001 0001; [{(Y + Y/8)/8} + Y] = 0001 0001 + 0111 1111 = 1001 0000 (unsigned addition, overflow ignored); [{(Y + Y/8)/8} + Y]/8 = 0001 0010;
Note that the actual answer is $\lfloor 127/7 \rfloor = 18$; After three stages of iterations, we achieve the exact value of X; note that a right-shift operation computes the floor function implicitly.
Thus, $X = \lfloor ((34 \times A) + B)/7 \rfloor$ = 0001 0010 = 18. The division process needs three logical right-shifts (each by 3 bits) and two additions. Hence, in total, we need *five shifts* and *four addition* operations in order to compute the value of X correctly without using multiplication/division/subtraction operations.

3 (**10 points**). Consider that two numbers, M and N represented in 2's complement notation, where M = 0xABCDEF76 and N = 0x543765DF. We need to multiply them using the Booth's Multiplication Algorithm, using *fewest* addition/subtraction operations.
(a) State, with justification, how many times addition, subtraction, shift, and bit-pair test operations (whether two consecutive bit-pairs are 00, 01, 10 or 11) need to be performed in order to complete the task.

(b) In the multiplier circuit, the delays of hardware modules are as follows: Adder/subtractor: 0.25 *ns*; Combinational Shifter: 0.05 *ns*; Bit-pair testing: 0.10 *ns*. Ignore initialization time required for loading M and N.
(i) In one implementation of the multiplier, each of these operations needs one clock cycle. What is the CPU-time needed to finish the above multiplication?
(ii) To speed up the operation of the multiplier, in another implementation each operation is allowed to be completed in multiple clock-cycles. How much CPU-time is needed to finish the above multiplication for this case? Your approach should attempt to minimize CPU-time.
(6 + (2 + 2))

Solution:
M = 0xABCDEF76 = 1010 1011 1100 1101 1110 1111 0111 0110
N = 0x543765DF = 0101 0100 0011 0111 0110 0101 1101 1111

Both M and N have nine runs of "1"; however, M has fewer 0 ←1 transition (one less). Therefore, we select M as the multiplier and N as the multiplicand.

(a) During the execution of Booth's algorithm, we will need 9 subtractions, 8 additions, 32 bit-pair testing (including the one with initial 0), and 32 shifts.

(b) (i) Each operation needs one clock cycle. Therefore, the clock-cycle time has to be fixed at 0.25 *ns*, because addition/subtraction is the *slowest* among all. The CPU-time needed for multiplications is therefore = (9 + 8 + 32 + 32) × 0.25 *ns* = 81 × 0.25 *ns* = 20.25 *ns*.

(ii) In the second case, where multi-cycle realization is allowed, we set the clock-cycle time to just cover the delay of the *fastest* operation (shifting), that is, 0.05 *ns*. Hence, each shift would take *one* clock cycle, bit-pair testing - *two* clock cycles, adder/subtractor - *five* clock cycles.
Thus, total # clock cycles needed = (17 × 5) + (32 × 2) + (32 × 1) = 181 cycles;
Therefore, CPU-time = 181 × 0.05 = 9.05 *ns*.

4. (**10 points**) Consider the following two normalized floating-point (FP) numbers A and B in IEEE 754 single-precision format, which we want to add:

A: 0 1111 1110 1111 0000 0000 0000 0000 001

B: 0 1111 1011 1100 0000 0000 0000 0001 001

(a) What will be the values of guard, round, and sticky bits?
(b) What will be the result of (A + B) in 32-bit IEEE 754 format? Show your work and justify your answers. (3 + (4 + 3))

Solution:
(a) The FP-exponent field in A = 1111 1110 = 254; hence the actual exponent is 254 – 127 = 127;
The FP-exponent field in B = 1111 1011 = 251; hence the actual exponent is 251 – 127 = 124;
For adding A and B, the significand of the lower exponent field (i.e., of B's) should be right-shifted to make the exponent fields equal. Thus, three bits 0, 0, 1 should be shifted out from B. Hence, Guard-bit = 0, Round-bit = 0, and Sticky-bit = 1 (logical OR-ing of all bits following Round-bit).

(b) While performing A + B, the exponent field of the sum becomes 2^{128} after normalization. Therefore, it would create positive overflow. Hence, the sum would be set to + ∞, i.e.,
A+ B = 0 1111 1111 0000 0000 0000 0000 0000 000

5. (**10 points**) Let N be a normalized FP-number in 32-bit IEEE 754 format. Both the exponent field (E) and mantissa (M) are extracted and stored in two 32-bit registers $t1 and $t2, respectively, in right-justified fashion, i.e., with all blank bits on the left filled with 0's. Sketch an algorithm in the light of MIPS which can check whether or not N is exactly an integer expressible as 32-bit 2-complement number. You do not need to write any MIPS code, just describe what conditions need to be checked so that they can be implemented using MIPS instructions.

Solution: Let *E* denote the FP-exponent, *e*: the actual exponent, *M*: mantissa, and *s*: sign.
A normalized number *N* in 32-bit IEEE 754 format is evaluated as:

$$N = (-1)^s \, 1.M \times 2^e; \; E = e + 127$$

Note that M is 23-bit binary string. Let a ($0 \leq a \leq 23$) denote the number of bits in M counted from the MSB-side until we find the "last" 1 in M. In other words, all bits towards the right starting from (a + 1)-place in M are 0. If N is an integer, then clearly, $a \leq e$. Thus, e cannot be negative.
Note that in the significand of N, there is a leading 1 (hidden bit). In order to accommodate the leading 1 and the sign bit in 32-bit 2's complement integer format, one must have, $e \leq 30$, in the FP-representation of N. Hence, N can be represented as a 32-bit 2's complement number only if $a \leq e \leq 30$. Note that the converse may not be true. For example, a number which is representable in 32-bit 2'complement may not be exactly representable in 32-bit floating point, because the latter has only 23-bit long mantissa!

These conditions can easily be checked by writing MIPS code that involves logical shifting and comparisons.

6. (**10 points**) Consider a special floating-point format, which is an 8-bit normalized format, with 1 sign bit, 4 exponent bits, and 3 mantissa bits. Except for being only 8-bit wide, this scheme is analogous to the standard IEEE-754 32-bit or 64-bit format in terms of the meaning of fields, bias, and special encodings.

(a) Represent $(0.00110111)_2$ in this 8-bit special format with the nearest-even rounding.
(b) Represent $(16.0)_{10}$ in this 8-bit special format. (5 + 5)

Solution:
(a) $N = (0.00110111)_2 = 1.10111 \times 2^{-3} \Rightarrow$ since we have 4-bit exponent, it should be excess-biased with 7. Hence, in the 8-bit special format, its representation will be after even rounding:

$N = 0\ \ 0100\ \ 110$

(b) $N = (16.0)_{10} = (10000.00)_2$ in $= 1.0 \times 2^4 \Rightarrow E = 4 + 7 = 11 = 1011$; hence, in special 8-bit format, $N = 0\ \ 1011\ \ 000$

7. (**5 points**)

(a) What is the maximum difference between two consecutive normalized FP-numbers (ignoring $\pm \infty$) that are representable in 32-bit IEEE 754 format?
(b) A and B are two 32-bit FP-number, where A(B) is a normalized (denormal) number. Given A = 0 1111 1110 1111 0000 0000 0000 0000 001, find B such that (A – B) is maximum. Write your result in 32-bit IEEE 754 FP-format.
(c) In FP-arithmetic, while performing addition, the normalization step may require either left or right shift of the sum of two significands. On the other hand, during multiplication, the normalization step may require *only right shift* of the product of two significands. Justify the reason. (2 + 2 + 1)

Solution:
(a) Maximum difference between two consecutive 32-bit normalized FP-numbers
$= 2^{-23} \times 2^{127} = 2^{104}$.

(b) Since A is a positive normalized number and B has to be a denormal number, the value of B for which A – B is maximized when B is the least denormal number on the real line. Hence,
$B = -(1-2^{-23}) \times 2^{-126}$. In FP-format, B = 1 0000 0000 1111 1111 1111 1111 1111 111.

(c) During multiplication of two significands, the number of significant bits preceding the binary point can only increase. Hence, during normalization only right-shifts of significands may be required. This is not always true while performing addition, because after exponent equalization and addition, shifts in both directions may be necessary during normalization.

8. (**10 points**)
Let M denote hardware realization of a 32-bit array multiplier for integers operands. M uses AND gates for computing partial products and 1-bit FA-blocks for additions. The architecture deploys a *standard* carry-save-adder for adding partial products (not Wallace-tree type). The last row of adders is implemented with a 4-stage carry-select-adder, each stage comprising an 8-bit ripple carry adder (RCA). Assume the delay of each 1-bit FA block is 1 *ns*. Ignore delays of all AND-gates MUX-es, other logic, if any, and interconnecting wires.

(a) What is the worst-case delay of M?
(b) How many 1-bit FA-blocks do you need? (5 + 5)

Solution:
Different kinds of array-multipliers with carry-save additions are possible. The last row is being implemented with 4-stage carry-select adders, where each stage is an 8-bit RCA. Therefore, the number of FA's needed for the last row would be $8 + (3 \times (8 + 8)) = 56$; delay along the last row = 8 *ns* for the first RCA stage; rest is pre-computed for both carry-bit 0 and 1, which may be treated as instantaneous because the delays of associated logic, MUXes are ignored. For the rest of the array, different realizations are possible: #FA's = (32×32) or (32×31) or (31×31). Thus, the total number of FA's, after adding 56, may be 1080, 1048, or 1017;

The total delay may also be $(32 + 8) = 40$ *ns*, or $(31 + 8) = 39$ *ns*.

Note to TAs: For different CSA architectures, the number of required FA's and worst-case delay may be different, so use your judgement to grade.

9. (**10 points**)
(a) Two (nk)-bit numbers $A = a_{nk}\, a_{nk-1}\ldots a_1$ and $B = b_{nk}\, b_{nk-1}\ldots b_1$ are being added using the following scheme: The bits are partitioned into n groups, each group consisting of k bits. For each group of k bits, a PPA (Brent-Kung Parallel Prefix Adder) is employed to compute the sum. These PPA's are then serially cascaded as in ripple-carry adders. Estimate the cost and delay of the proposed adder in terms of n and k.

(b) We want to multiply two integers A = 57 and B = 58, using Karatsuba multiplication algorithm (KMA). In *decimal space*, show the steps that lead to the correct result for $(A \times B)$ with reduced computational effort. Go down only up to *one level* of recursion for illustration. In general, what is the complexity of KMA for multiplying two n-bit binary integers?
((2 + 3) + (3 + 2))

Solution:
(a) Cost = $O(n\, k \log k)$; Delay = $O(n \log k)$;

(b) A = 57 and B = 58; In Karatsuba algorithm, the strings are partitioned into halves and multiplication is performed so that in each step, the number of multiplications reduces from 4 to 3. This process is then executed recursively. For this particular example, we proceed as follows:

Let X = 5, Y = 7; P = 5, Q = 8

We can write: A = 10 X + Y; B = 10 P + Q
Therefore, A × B = (10 X + Y) × (10 P + Q) = 10^2 XP + 10(XQ + YP) + YQ;
XP = 5 × 5 = 25; YQ = 7 × 8 = 56;
Karatsuba's technique: (XQ + YP) = (X + Y)(P + Q) – XP – YQ = (12 × 13) – 25 – 56 = 75;
(*two new* multiplications reduce to few additions and *one new* multiplication);

Hence, A × B = 10^2 XP + 10(XQ + YP) + YQ = (10^2 × 25) + (10 × 75) + 56 = 2500 + 750 + 56 = 3306, which is the correct result.

In general, for multiplying two n-bit binary numbers, the recurrence $T(n) \leq 3T(n/2) + O(n)$ solves to $T(n) = O(n^{lg\ 3}) = O(n^{1.585})$, because shifts and additions in each step take $O(n)$ time.
This improves the conventional quadratic complexity for multiplication.