HW6 – Efficient Multiplication using PAL

The goal of this assignment is to practice implementing algorithms in PAL.

Deliverables: ONE (and only one) partner should submit your final solution. Submit a zip folder named user1_user2 containing 2 .pal program files named part1.pal and part2.pal with your solutions for each of the 2 problems described below. Make sure you include both partners' names in comments at the top of **each** .pal file. Also that you submit your .pal files, **not** your assembled .rim files, those are unreadable without loading them into the emulator and don't include labels, comments, etc.

- If you are not familiar with creating zip folders on a mac, just put both of your .pal files (and nothing else) into a new empty folder named user1_user2. Note it's important you name the folder this way **before** zipping it, as if you simply rename the zip file after creating it, the name will change back to its original value when the grader unzips it. Once you've created your folder with the correct name, right click on it and choose the "Compress folder_name" option. A file named folder_name.zip will appear in the same place as the original folder. This is the file you should submit, Moodle will only accept a single file, so you have to create this zip file with both of your solutions.

Part 1 – Multiplication with 12-bit result (15/20 pts)

Write a PAL program in part1.pal that multiplies 2 ***unsigned integers*** using the same efficient multiplication algorithm you programmed in MIPS for HW3 – though the efficiency will be O(12) instead of O(32) because PAL numbers are at most 12 bits.

Remember everything is binary to the computer, but while working with PAL all numbers (whether a memory address, the data value stored at a particular address, the value in the AC, anything!) **will be shown in octal** to make them more human-readable, rather than displaying all 12 bits. It's easy to convert octal to decimal, each octal digit corresponds to 3 bits, so a 4 digit octal value represents a 12 bit value. This applies to anywhere you see PAL-related values displayed, e.g. in your .pal code file, in the memory displayed in the simulator, and in this assignment description itself. Octal looks a lot like decimal, so it's important you don't misinterpret these values as if they were typical integers in base 10! For example, we will put our data values at addresses 174-177, because those are the 4 slots just before the program code starts at the next slot with address 200. If you try to do something at address 178 you will get an error, because 8 is not a valid octal digit! Rather 177 + 1 = 200 in octal.

The numbers to multiply (*operands*) will be stored at memory locations 174 and 175. Store your multiplication result (the *product*) in memory location 176. Memory location 177 will be used to signify overflow, if overflow occurs set this value to 1, otherwise set it to 0. The product in memory 176 will be meaningless if overflow occurs, it does not matter what partial result you leave in there, only that you set the overflow bit. You may use any of the other addresses between 0 and 174 to hold temporary/intermediate values as you work through the algorithm, but the grader will edit the values in 174 and 175 to test your program, and check 176 and 177 for the result and/or overflow, so you must use those precisely as specified. You should not assume that a value in memory will be 0 initially unless you expressly set it to 0 in your code.

As always use good coding style with clear comments, and for full credit make your solution as efficient as possible (see hints below).

Hints:

- You are working with ***unsigned*** 12-bit integers, so the range of possible values is 0 - 7777 (octal). Overflow occurs if the multiplication result becomes larger than 7777. For example, 2501 * 3 = 7703 with no overflow, but 2501 * 4 causes overflow because the result 12404 would require 13 bits, and 2501 * 401 obviously causes overflow because the result 1243101 would require 19 bits.
- If you find yourself counting the number of bits in one/both of the operands in order to determine the number of times to loop, you should think about how you can avoid that. What check can you make instead to determine when to quit the loop? Stated another way – you should use a *while* style loop, not a *for* style loop.
- You should only need 1 loop, if you find yourself using a subloop to for example shift a variable number of times, think about an easier method that does not require a subloop.
- You should quit the loop as early as possible, either when the multiplication is complete or as soon as you detect overflow.
- The instructions to shift in PAL are RAR and RAL (rotate accumulator right and rotate accumulator left). These shifts incorporate the link bit as well, so when you shift right, the last bit of the accumulator gets put in the link, and the value that was in the link gets put in the first bit of the accumulator. How might you use this to check the value of the last bit in the AC instead of masking it out with logic or multiple shifts?
- Remember to clear the current link bit before shifting if you do not want its value to end up in the first or last bit of the accumulator! Conversely make sure you do not clear it if you do want it to end up in the AC.

Part 2 – Multiplication with 24-bit result (5/20 pts).

Copy your program from part 1 **to a new file** part2.pal; you will turn in **both** solutions in separate files. You will use a slightly modified version of our efficient multiplication algorithm to again multiply 2 ***unsigned*** 12-bit integers, but now you will compute the correct product regardless of how large the operand values are, by using 2 memory slots, or 24 bits total, to store the product. Given two 12-bit operands, the product cannot be larger than 24 bits, so overflow can no longer occur.

The operands will be stored at the same memory addresses 174 and 175, but now use both addresses 176 and 177 to store your 24-bit product, with the upper (most significant) 12 bits of the product in address 176, and the lower (least significant) 12 bits in address 177. Concatenating the values in 176 and 177 will then produce the entire 24-bit product. For example, when computing 2501 * 401 = 1243101 which caused overflow in Part 1, you should now end up with the value 0124 in address 176, and 3101 in address 177.

The specific algorithm you will use is described below, using X and Y to label the operands in memory addresses 174 and 175, and FIRST and LAST to refer to the 2 product memory addresses 176 and 177. Please **use those same labels**, not actual memory addresses, in your .pal program file in order to make your code readable. As in part 1 try to make your solution as efficient as possible, though any working solution that follows the algorithm will receive full credit (within reason). Remember you **cannot** quit the loop early anymore, but you can minimize the number of instructions inside the loop.

1. initialize any necessary values (set mem addresses 174/175 to operand values, 176/177 to 0, any other mem used for temp data to appropriate values, clear AC & L, etc.)
2. repeat next 3 steps **exactly 12 times** (you can't quit early in this algorithm or your product won't be shifted far enough right)
    a. if last bit of Y is 1
    add X to **upper** half of product (FIRST)
    b. shift Y right 1
    c. shift entire product right 1 – remember product is ALL 24 bits of FIRST : LAST (: refers to concatenation) – so this means the last bit of FIRST gets shifted into the first bit of LAST. The last bit of LAST is thrown out. Try and do this 24-bit shift efficiently by using the specific way PAL's rotate instructions work as you rotate each 12-bit half-product. You can do the entire shift in <= 6 instructions, using no "if" type conditional branches (so no skip instructions at all).

Final 24-bit product is now just FIRST : LAST

Here's an example of running the algorithm on our sample calculation 2501 * 401 = 01243101

1. PRODUCT will be the 24 bits of FIRST : LAST, initially 00000000 octal, or 24 0's in binary
   X = 2501 octal, or 010 101 000 001 binary
   Y = 0401 octal, or 000 100 000 001 binary
2. 1st time through loop
    a. last bit of Y is 1, so add X to FIRST, FIRST = 0000 + 2501 = 2501
    PRODUCT = 25010000 octal, or 010 101 000 001 000 000 000 000 binary
    b. shift Y right 1 **BIT** (NOT 1 octal digit, that would actually be shifting by 3!)
    Y = 000 010 000 000 binary, or 0200 octal
    c. shift product right 1 **BIT**
    PRODUCT = 001 010 100 000 100 000 000 000 binary, or 12404000 octal
    specifically we did this by shifting FIRST right 1 bit, so FIRST now = 1240, but the last bit of FIRST that we shifted off of the end gets shifted onto the first bit of LAST as part of also shifting LAST right 1, and that bit was a 1, so LAST now = 4000.
3. 2nd time through loop
    a. last bit of Y is 0, so no add
    b. shift Y right 1 bit, Y = 0100 octal
    c. shift product right 1 bit, PRODUCT = 05202000 octal
4. 3rd loop
    a. no add
    b. Y = 0040 octal
    c. PRODUCT = 02501000 octal
5. 4th loop
    a. no add
    b. Y = 0020 octal
    c. PRODUCT = 01240400 octal
6. 5th loop
    a. no add
    b. Y = 0010 octal
    c. PRODUCT = 00520200 octal

7. 6th loop
    a. no add
    b. Y = 0004 octal
    c. PRODUCT = 00250100 octal
8. 7th loop
    a. no add
    b. Y = 0002 octal
    c. PRODUCT = 00124040 octal
9. 8th loop
    a. no add
    b. Y = 0001 octal
    c. PRODUCT = 00052020 octal
10. 9th loop
    a. add X (2501) to FIRST (0005)! FIRST = 2506 octal
       PRODUCT = 25062020 octal, 010 101 000 110 010 000 010 000 bin
    b. Y = 0000 octal
    c. PRODUCT = 12431010 octal, 001 010 100 011 001 000 001 000 bin
11. 10th loop
    a. no add
    b. Y = 0000 octal
    c. PRODUCT = 05214404 octal, 000 101 010 001 100 100 000 100 bin
12. 11th loop
    a. no add
    b. Y = 0000 octal
    c. PRODUCT = 02506202 octal, 000 010 101 000 110 010 000 010 bin
13. 12th loop
    a. no add
    b. Y = 0000 octal
    c. PRODUCT = 01243101 octal

DONE!!!
- Note that Y became 0 several loop iterations before we quit, but we still had to continue all the way through the 12th to get the final PRODUCT shifted right far enough.