CS31007: Computer Organization and Architecture

Model Solutions of Exam-1 (Autumn 2021)

Full Marks = 50; Credit = 30%

1. (**6 points**) In an assembly-language program *P*, 20% instructions require sequential execution because of data dependence. Two machines M1 and M2 are given: M1 comprises one core, i.e., a single processor; M2 is a multi-core machine with 80 copies of M1 on-chip that is capable of performing parallel processing. Both of them operate with the same clock frequency.
   (i) Calculate the maximum achievable speed-up with respect to the execution time needed by the new machine M2 compared to that of M1 for the program *P*, following Amdahl's law. **(2 points)**
   (ii) Next, compute the same following Gustavson-Barsis law. If the estimated speed-up values differ in these two cases, explain the reason behind them. **(4 points)**

   For machine M2, you may ignore inter-processor communication time and other overheads.

   Solution:

   (i) Let T denote the execution time which comprises sequential components (t_s) and parallelizable component (t_p) on a single-core machine; Thus, $T = t_s + t_p$. When N cores are available, the speed-up by Amdahl's Law is:
   $= (t_s + t_p)/ (t_s + (t_p/N)) = T/(0.2T + (0.8T/N))$ since 20% of tasks are sequential. Putting N = 80, we get, Speed-up ≅ **4.76**

   (ii) Following Gustavson's Law, we can write Speed-up as follows. Assume, t_s denotes the time spent on the sequential component on an N-core machine, and t_p denotes the time to execute the parallelizable component on the same multi-core machine; thus, $t_s + t_p = 1$ (100%).
   Therefore, Speed-up = $(t_s + N \times t_p)/(t_s + t_p) = 0.2 + (80 \times 0.8) =$ **64.2**

   The reason behind such discrepancy between speed-up values obtained by the above two methods is the following: Amdahl's law assumes that even with the availability of multiple cores, the job size does not change, and hence, it provides a pessimistic view of speed-up that is limited mostly by the fraction not-enhanced. On the other hand, Gustavson's Law is more realistic as it assumes that a multi-core machine can execute a larger-size job compared to a single-core machine at the expense of same computation time, thus providing an optimistic view of achievable speed-up.

2. **(6 points)** Consider the following MIPS code segment being executed on machine M:
   lw $t1, 10($t2)
   srl $t1, $t1, 16
   lui $t3 7FFF (hex)

```
        ori $t3 FFFF (hex)
Loop:   xor $t4, $t1, $t3
        add $t1, $t1, $t4
        sll $t1, $t1, 16
        bgt $t1, $zero, Loop
        sw $t1, 1000($t3)
```

Assume M has a clock frequency is 3.5 GHz. Also, assume that *lw* needs 5 clock cycles, *sw* 4 clock cycles, *bne* needs 2 clock cycles, and all other instructions 3 clock cycles to execute. Calculate the total amount of CPU-time (in nanoseconds) required to execute the above code.

Solution: The branch instruction is not taken and hence, all instructions are executed exactly once. Assuming branch takes 2 clock cycles, we need a total of 29 clock cycles. Clock cycle time = (1/3.5) nanosec. Thus, CPU-time = 29/3.5 ≅ 8.286 nanosec.

Note to TA: Some students may assume that *bgt* takes 3 cycles (because of a typo in the question paper), or bgt takes 5 cycles (assuming slt plus branch); in those cases, the total clock cycles 30 cycles or 32 cycles. Hence, CPU-time may be computed as 8.57 ns or 9.14 ns. Please give then full credit as well.

3.(a) **(2 points)** f = x ? y : z ;

Note to TA: if a student has used if…else construct and the solution is correct, give 1 (out of 2).

(b) **(4 points)**
```
add $t0, $t3, $zero # greedily set f = z ; move $t0, $t3 would also work
beq $t1, $zero, L1 # if x == 0, we are done, go to L1
add $t0, $t2, $zero # correct the value and set f = y; move $t0, $t2 would also work
L1: ……
```

Note to TA: other three instruction solutions, and four instruction solutions (possibly including unconditional jump) are also possible. Give full credit in each case if the solution is correct. However, if the solution is correct but contains more than four instructions, give 2 (out of 4).

4. **(2 points)** The correct answer is **(d).**

5. (5 points)

Let the content of register `$t1` represent a 2's complement number N. Write a MIPS code fragment such that the register `$t2` stores the absolute value of N. In other words, register `$t2` has a copy of register `$t1` if N is positive, and `$t2` should hold the negative of N, if N is negative. (Hint: you are encouraged to attempt it using at most three instructions, however you can write longer code if necessary). [5]

Solution:

There are many possible solutions, TA may check students' solution.

We assume that as output t2 will hold the unsigned result, i.e., the absolute value.

sra $t2, $t1, 31
xor $t1, $t2, $t2
subu $t2, $t1, $t2;

Another solution may be:

bge $t1, $zero, exit
subu $t1, $zero, $t1
addi $t2, $t1, $zero;

6. (3 points) (a) The program calculates the GCD of the two values present in registers $a0 and $a1.

   (b) (1 points) The output of the program would be: "The value is: 3"

7. (2 + 3 = 5 points) The *addi* instruction performs 2's complement addition. Also, this instruction interprets its 16-bit constant operand to be a 2's complement number. As a result, *addi* automatically sign-extends the 16-bit immediate constant operand while performing the addition. In tis case, the constant operand 0xdcba will be sign extended to 0xffffdcba (since the most significant bit of 0xdcba is 1), which is a -ve number in 2's complement notation.

   The result of this addition (the carry-out will be discarded) would be 0x1233dcba and would be found in the $t0 register, which is not what the student wanted.

8. (4 points) $(71234)_{10}$ is the same as 0x0001_1642. Hence, the student can perform the following:

   lui $t1, 0x0001
   ori $t1, $t1, 0x1642

9. (2 + 5 = 7 points)
   The output Y would become 0 only when all four inputs feeding it assume logic zero.
   The two outputs of the full-adder block would become 0 only when A = B = C = 0.
   The output of the NAND gate would be 0 only when D = E = F = 1. The output of NOR gate would be zero only for three combinations: G, H = (0, 1), (1, 0), (1, 1). Hence, out of 256 combinations, there are exactly $(1 \times 1 \times 3) = 3$ combinations for which output Y will be 0. Thus, for the remaining 253 combinations of input variables, output Y will assume logic 1.

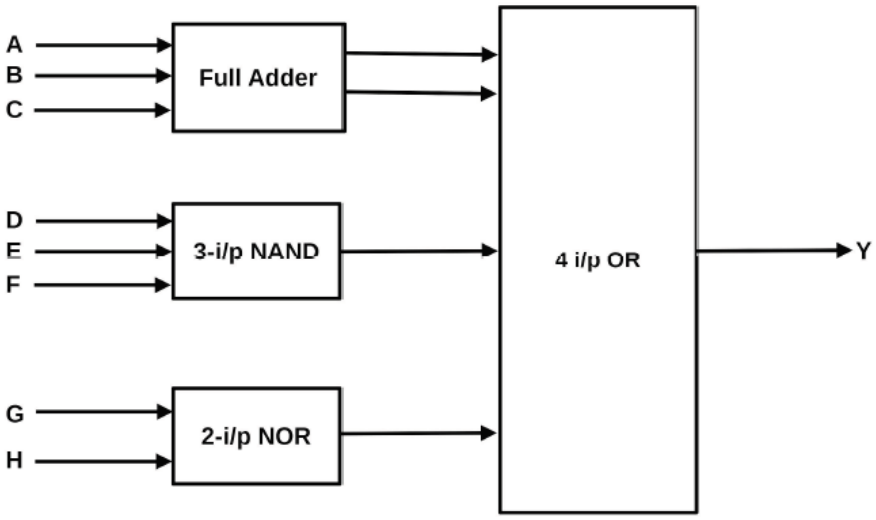

Figure 1: A simple eight-input combinational logic circuit.

Consider the logic circuit shown in Fig. 1, with eight Boolean inputs: A through H, and one Boolean output Y. The functionalities of the individual modules in the given circuit are self-evident. Out of 256 possible input combinations, how many would cause the output Y to assume logic value 1? Give brief justification in support of your answer. Note that no credit would be given for the first part of the question, of the explanation provided for the second part is incorrect.
(Hint: Please **do not** try to construct the truth-table for an eight variables Boolean function, or simplify the Boolean expressions for Y, as those approaches would be time-consuming.) $[\mathbf{2 + 5 = 7}]$

10. (5 points) **See solution uploaded on Moodle.**

Note to TA: only the *toupper* procedure body has to be described in the solution, the student need not write the rest of the program in his/her solution. If the solution does not cover the case of a null-string passed to the procedure as input argument correctly, deduct 1 mark.