ECE 255 Introduction to Computer Architecture Sample Quiz #3 (October 24, 2025, 9:30-10:15)

Name: _____ ID: _____ Total Marks: 30.

CLOSED Book/Notes/Phone; NO calculator; make appropriate assumptions.

You must justify your answers, that is, explain how you get the answer.

A. $2^0 = 1$; $2^1 = 2$; $2^2 = 2^1 \text{ X } 2^1$; $2^3 = 2^1 \text{ X } 2^2$; … $2^n = 2^1 \text{ X } 2^{(n-1)}$

B. $2^{10} = 1K$; $2^{20} = 1M$; $2^{30} = 1G$; $2^{40} = 1T$

C. Given an *n*-bit vector **B** = $b_{n-1} \ldots b_1 b_0$; its decimal value is = $b_{n-1} \text{ X } 2^{n-1} + \ldots + b_1 \text{ X } 2^1 + b_0 \text{ X } 2^0$

D. Decimals 0, ..9, 10, ..15 can be represented in hexadecimal as 0, ..9, A, ..F or 0, ..9, a, ..f

E. Addressing Modes: EA= Effective Address

Immediate	#value	operand=value
Register	R*i*	EA= R*i*
Absolute (Direct)	LOC	EA=LOC
Register Indirect	(R*i*)	EA=[R*i*]
Index	X(R*i*) (R*i*,R*j*)	EA=[R*i*]+X EA=[R*i*]+[R*j*]

F. Marking Scheme:

Q	Marks	Q	Marks			
1a	/ 4	4a	/ 4			
1b	/ 4	4b	/ 2			
2	/ 6	5a	/ 2			
3	/ 6	5b	/ 2	Total	/ 30	Bonus:

---

1. In ARM architecture, the four status bits are negative, zero, carry, and overflow. I want to check the condition of 'positive', but none of the status bits indicate positive (or greater than). What should I do?

Check changes in two bits: not negative and not zero

2. There are three types of Loops: While, Do While, and For Loop. Indicate the significant difference among the three.

DO WHILE: check at the end of the loop or the task probably changes the condition; execute the task at least once

WHILE: check at the beginning of the loop, or the condition must be set to the desired value before entering; may not execute the task at all

For: we know precisely how many times to go through the loop

3. Before writing an assembly program for an application, do you prefer to define problem data structures (such as variables, arrays, etc.) first or program data structures first (such as counters, etc.)? Explain.

Program data structures first, so that we can write instructions that access the operands correctly (correct effective address).

4. Elapsed time: the wall clock time to execute a program. Processor time: the time periods that the processor is executing a program. Which time is more useful to the user/programmer? Explain.

A user cares about the actual time it takes to run the program, which includes the processor time or actual time the program is run by the CPU, as well as the system load (other programs) at the time.

5. In the laboratory, we use the ARM board and run assembly programs. Do you expect a big difference between the CPU time and the Wall Clock time when running an assembly program? Explain.

No, CPU time and Wall Clock time should not have a big difference here because the board is dedicated to running a single user program, without other programs running on the CPU, which would increase the Wall Clock time.

6. An integer can be represented in its binary equivalent using hint C above. How do we perform a "multiply by 2" operation on a positive integer? Show the process using +9.

+9 = 1001

10010 [Shift each bit by one position to its left, or left shift by 1 bit]

7. For 4-bit 2's complement integers (range is {-8,-7…0,+1,+2..,+7}), which additions with 2 negative integers would not result in overflow if one of the two is -8? Explain

None; -8 added to any negative integer within the range results in an overflow

8. Comment on the validity of this statement: "In 2's complement arithmetic, overflow (i.e., the result is outside of the range the number of digits can represent) can occur only when adding two numbers that have different signs."

No, when adding two numbers that have different signs, the result is always within range.

9. Show whether we should use pre-decrement, pre-increment, post-decrement, or post-increment, for a POP instruction (move item from stack memory to a register, e.g., Move ±(SP)±, R1) if

9a. Stack growth is ascending with a full-stack stack pointer. Explain.

Post-decrement or (SP)-

Explain: SP points to a valid item (last stacked item), so it is the one that we want to pop, then the SP moves towards the bottom since the stack shrinks, and it's from higher to lower address, therefore decrement.

9b. Stack growth is descending with an empty-stack stack pointer. Explain.

Pre-increment or +(SP)

Explain: SP points to a space above the last stacked item, so the SP must be moved first to point to the top element (pre), then pop the item. This movement is from higher to lower within a stack, and since stack growth is descending, we must increment.

10. Show whether we should use pre-decrement, pre-increment, post-decrement, or post-increment for a POP instruction if

10a. Stack growth is ascending with an empty-stack stack pointer.

pre-decrement

10b. Stack growth is descending with a full-stack stack pointer.

post-increment

11. In assembly programming, we need to initialize the stack pointer to point to the bottom of the stack (in the case of an empty stack, where the SP is pointing to the next available space). What else do we need to initialize? Explain? Explain.

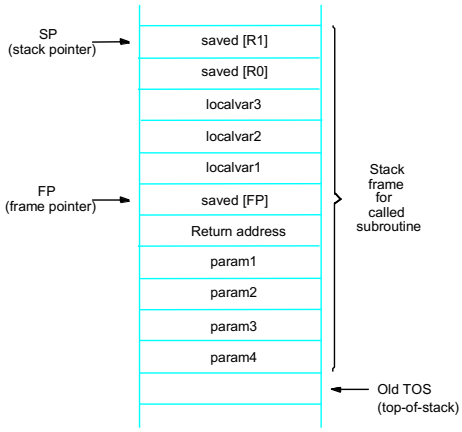
The size of the stack ensures that it won't consume all the allocated memory with many pushes.

12. To push an item on top of the stack, the instruction Move NEWITEM, (SP)- is used. Circle the correct answers:

12a. This stack grows from (lower / higher) address to (lower/higher) address in memory.

12b. This stack's pointer SP is pointing to a (valid / useless) item on the stack.

13. Given the following stack frame layout as shown



13a. What is the minimum number of parameters that the Caller could pass? And why?

Zero

Why: when the Subroutine does not need any parameters

13b. What is the minimum number of local variables that the Subroutine should create? And why

Zero

Why: when the Subroutine is not using local variables

13c. What is the minimum number of registers that the Subroutine could use? And why?

Zero

Why: when the Subroutine is not using general-purpose registers

14. Who is responsible for storing each of these items on the stack? (Hint: Caller, Subroutine, and Processor):

11a. Parameters passed **caller/subroutine**

11b. Return address **processor**

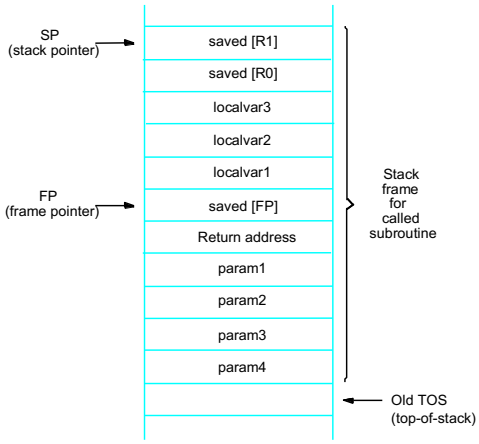
11c. Local variables **subroutine**

11d. Saved registers **subroutine**

15. A single subroutine call is made, resulting in a stack frame on the stack, with 4 parameters passed, 2 registers saved, and space for 3 local variables allocated, as shown in the figure below. Assuming there are four general-purpose registers available in this processor: R0, R1, R2, and R3

Can we use R1 as an FP? Explain.

No, the original value of R1 is saved at the location pointed to by the FP, and then R1's new value is saved on the stack, but if we start using and changing the value of R1 in the subroutine, then the FP would not be pointing to the stack frame correctly.



16. There is some overhead associated with a subroutine call, such as the 'Call' and 'Return' instructions, stack frame creation, and other related tasks. Give two situations in which we should not use a subroutine.

Subroutine consists of a few instructions.

Subroutine is called only a few times.

17. A caller is responsible for putting the parameters onto the stack before calling a subroutine. How would the caller know how many parameters to pass?

Either one of these or similar:

From specifications (e.g., commonly used routines from the library)

The design team typically implements the main program and the subroutine; obviously, they communicate with each other, so what needs to be passed is known

The user writes the main and the subroutine

18. What is the minimum size of (how many items in) a stack frame?

2 , Return Address and saved Frame Pointer

19. Upon completing a subroutine's execution, where should the callee (subroutine) store the local variables used, before control returns to the caller? Explain.

No need to store the local variables.

Explain: When the subroutine is done and before returning to the caller, it no longer needs the local variables and therefore they don't have to be stored anywhere.

20. In an application program, by the time we are in the third (3) nested subroutine, we have (Indicate the correct answer and explain):

(i) 3 stack pointers used and 3 frame pointers created;

(ii) 1 stack pointer used and 1 frame pointer created;

(iii) 3 stack pointers used and 1 frame pointer created;

(iv) 1 stack pointer used and 3 frame pointers created

Correct answer: (iv) 1 stack pointer used and 3 frame pointers created

Explain:

3 nested subroutines => 3 frame pointers

Always 1 stack pointer only

END