**Question 2. Please explain how the stack is used for subroutine call and return (Chapter 7 “Stack Instructions” (Maximum 1 point)**

**Answer:**

**In computer science, a call stack is a stack data structure that stores information about the active subroutines of a computer program. This kind of stack is also known as an execution stack, program stack, control stack, run-time stack, or machine stack, and is often shortened to just "the stack". Although maintenance of the call stack is important for the proper functioning of most software, the details are normally hidden and automatic in high-level programming languages. Many computer instruction sets provide special instructions for manipulating stacks.**

**A call stack is used for several related purposes, but the main reason for having one is to keep track of the point to which each active subroutine should return control when it finishes executing. An active subroutine is one that has been called but is yet to complete execution after which control should be handed back to the point of call. Such activations of subroutines may be nested to any level (recursive as a special case), hence the stack structure. If, for example, a subroutine DrawSquare calls a subroutine DrawLine from four different places, DrawLine must know where to return when its execution completes. To accomplish this, the address following the instruction that jumps to DrawLine, the return address, is pushed onto the call stack with each call.**

**Citation:** Call stack. (2019, May 24). Retrieved from <https://en.wikipedia.org/wiki/Call_stack>.

**Question 3: Explain a computer's register-level architecture, including: (Maximum 1 point)**

1. **CPU-memory interface:**

CPU-Memory interface includes the data bus, address bus and some control signals including Read, Write, and Memory-Function-Complete (MFC). The CPU is interfaced to the data bus and address bus through the MDR and MAR registers, respectively.

In addition to this interface, there is a need for a CPU-Memory interface circuitry to manage their interaction. When the CPU wants to perform a read or write operation, it asserts either the Read or Write signal and puts the address to be read from or written to in the MAR register. Then, the CPU waits for the memory to finish the requested transfer operation. It is required that the CPU keeps the Read or Write signal set until the memory finishes the requested operation. The memory activates the MFC signal when the requested operation is completed. One the MFC is set to 1, and then the Read or Write signal can be set to 0. This interaction process between the CPU and memory is called handshaking.

A versatile CPU can communicate with main memory modules of different speeds

A fast memory can be accessed within a single clock cycle

Slower memory may require several clock cycles

The CPU-memory interface circuit should handle both fast and slow memories. Recall that internal signals generated by the control unit are active for one clock cycle during a given control step. Slow memory has to see those signals for more than one clock cycle. So, the CPU-memory interface circuit has to keep the Read or Write signals set to 1 until the MFC signal becomes 1. This is because the CPU will set them to 1 for one clock cycle and then go to the next control step when they may become 0.

It is assumed here that the memory is falling-edge triggered i.e., MFC signal will change value on the falling-edge of the clock. However, the control unit is rising-edge triggered and changes values of control signals on the rising edge of the clock.

**Citation:** El-Maleh, A. H. (n.d.). CPU-Memory Interface Circuit . Retrieved from <https://faculty.kfupm.edu.sa/COE/aimane/assembly/pagegen-172.aspx.htm>.

1. **Special-Use Registers**:

A Special Function Register (or Special Purpose Register, or simply Special Register) is a register within a microprocessor, which controls or monitors various aspects of the microprocessor's function. Depending on the processor architecture, this can include, but is not limited to:

* I/O and peripheral control (such as serial ports or general-purpose IOs)

timers

* stack pointer
* stack limit (to prevent overflows)
* program counter
* subroutine return address
* processor status (servicing an interrupt, running in protected mode, etc.)
* condition codes (result of previous comparisons)

Because special registers are closely tied to some special function or status of the processor, they might not be directly writeable by normal instructions (such as adds, moves, etc.). Instead, some special registers in some processor architectures require special instructions to modify them. For example, the program counter is not directly writeable in many processor architectures. Instead, the programmer uses instructions such as return from subroutine, jump, or branch to modify the program counter. For another example, the condition code register might not directly writable, instead being updated only by compare instructions.

**Citation:** Special function register. (2019, January 28). Retrieved from <https://en.wikipedia.org/wiki/Special_function_register>.

1. **Addressing modes:**

Addressing modes are an aspect of the instruction set architecture in most central processing unit (CPU) designs. The various addressing modes that are defined in a given instruction set architecture define how machine language instructions in that architecture identify the operand(s) of each instruction. An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction or elsewhere.

In computer programming, addressing modes are primarily of interest to those who write in assembly languages and to compiler writers.

There are 7 types of addressing modes:

1. **Immediate Mode:** The operand is an immediate value is stored explicitly in the instruction

Example: SPIM ( opcode dest, source)

1. **Index Mode:** The address of the operand is obtained by adding to the contents of the general register (called index register) a constant value. The number of the index register and the constant value are included in the instruction code. Index Mode is used to access an array whose elements are in successive memory locations. The content of the instruction code, represents the starting address of the array and the value of the index register, and the index value of the current element. By incrementing or decrementing index register different element of the array can be accessed.

Example: SPIM/SAL - Accessing Arrays

1. **Indirect Mode:** The effective address of the operand is the contents of a register or main memory location, location whose address appears in the instruction. Indirection is noted by placing the name of the register or the memory address given in the instruction in parentheses. The register or memory location that contains the address of the operand is a pointer. When an execution takes place in such mode, instruction may be told to go to a specific address. Once it's there, instead of finding an operand, it finds an address where the operand is located.

NOTE:

Two memory accesses are required in order to obtain the value of the operand (fetch operand address and fetch operand value).

Example: (textbook) ADD (A), R0

(address A is embedded in the instruction code and (A) is the operand address = pointer variable)

Example: SPIM - simulating pointers and indirect register addressing, SPIM/SAL - - array pointers and indirect register addressing

1. **Absolute (Direct) Mode:** The address of the operand is embedded in the instruction code.

Example (SPIM)

1. **Register Mode:** The name (the number) of the CPU register is embedded in the instruction. The register contains the value of the operand. The number of bits used to specify the register depends on the total number of registers from the processor set.

Example (SPIM)

1. **Displacement Mode:** Similar to index mode, except instead of a index register a base register will be used. Base register contains a pointer to a memory location. An integer (constant) is also referred to as a displacement. The address of the operand is obtained by adding the contents of the base register plus the constant. The difference between index mode and displacement mode is in the number of bits used to represent the constant. When the constant is represented a number of bits to access the memory, then we have index mode. Index mode is more appropriate for array accessing; displacement mode is more appropriate for structure (records) accessing.

Example: SPIM/SAL - Accessing fields in structures

1. **Autoincrement / Autodecrement Mode:** A special case of indirect register mode. The register whose number is included in the instruction code, contains the address of the operand. Autoincrement Mode = after operand addressing , the contents of the register is incremented. Decrement Mode = before operand addressing, the contents of the register is decrement.

Example: SPIM/SAL - - simulating autoincrement/autodecrement addressing mode

**Citation:** (n.d.). Retrieved from <http://www.cs.iit.edu/~cs561/cs350/addressing/addsclm.html>.

Addressing mode. (2019, September 30). Retrieved from <https://en.wikipedia.org/wiki/Addressing_mode>.