PassQueen
a.k.a dumpsqueen.com

ARM Accredited Engineer

ARM EN0-001

Version Demo

Total Demo Questions: 24

Total Premium Questions: 240

Buy Premium PDF

https://passqueen.com

support@passqueen.com

Topic Break Down

Topic	No. of Questions
Topic 1, Arm Architecture	71
Topic 2, Arm Instruction Set Architecture	50
Topic 3, Arm Cortex Processors	36
Topic 4, Arm Development Tools	59
Topic 5, Arm System Design	24
Total	240

QUESTION NO: 1

In ARMv7-A Thumb state, which TWO statements about an IT instruction are true? (Choose two)

A. It specifies the condition under which following Thumb instructions execute

B. It can define a conditional-execution block of up to four instructions

C. It changes the processor from Thumb state to ARM state

D. It invalidates entries in the instruction cache

E. It can only be used before a load or store instruction

ANSWER: A B

Explanation:

An **IT instruction specifies the condition under which following Thumb instructions execute**. It establishes an If-Then conditional-execution block, allowing a short sequence of Thumb instructions to execute conditionally without requiring a branch for every instruction.

An IT block can contain **up to four instructions, including the IT instruction**. The condition encoded by the IT instruction controls whether each instruction in the block is executed according to the specified Then or Else pattern. The individual instructions in the block must follow the architectural restrictions for IT blocks; in particular, branch instructions have special restrictions and are not generally usable at arbitrary positions in the block. See the ARM Architecture Reference Manual for ARMv7-A and ARMv7-R.

QUESTION NO: 2

When setting the initial location of the stack pointer and the base address of the heap, the ARM EABI requires that the:

A. Base address of the heap must be the same as the initial stack pointer.

B. Stack pointer must be 8-byte aligned.

C. Heap must be in external RAM.

D. Initial stack pointer must be the lowest addressable memory location.

ANSWER: B

Explanation:

The requirement that the **stack pointer must be 8-byte aligned** is correct. The ARM procedure-call standard specifies a universal stack constraint: whenever the stack pointer is used to access memory, it must be word aligned, and at a public interface—such as a function entry or exit—it must be aligned to an 8-byte boundary. Consequently, startup code must establish an initial stack-pointer value whose address is divisible by eight, and code must preserve that alignment when making calls. This rule enables interoperable object code and supports data types and instructions that require or benefit from doubleword alignment, including 64-bit values and floating-point data. It also allows independently compiled routines, libraries, and exception-related code to rely on a consistent stack layout without imposing extra realignment work at every interface. The heap location is a linker, runtime, or platform memory-layout decision; the EABI stack-alignment rule does not require a particular relationship between the heap base and the initial stack-pointer address. See the ARM Procedure Call Standard for the Arm Architecture, especially its stack constraints, in the official ABI for the Arm Architecture releases and the current AAPCS32 specification source.

QUESTION NO: 3

In general, when programming in C, stack accesses will be reduced by:

A. Disabling inlining.

B. Never passing more than four parameters in function calls.

C. Declaring automatic variables as "packed".

D. Configuring the compiler to optimize for space.

ANSWER: B

Explanation:

Never passing more than four parameters in function calls is correct because the standard ARM Procedure Call Standard uses the first four core registers, r0 through r3, to pass argument values at a function-call boundary. When a call requires additional arguments, those values are normally placed in the caller's stack argument area. Keeping calls to four or fewer suitably sized core-register arguments therefore avoids those extra stack stores by the caller and corresponding stack loads by the callee. This can reduce memory traffic as well as code required to prepare and consume the call frame.

This guideline is general rather than absolute: argument types, structure passing, floating-point calling conventions, compiler optimization choices, and register pressure can affect the generated code. Nevertheless, designing frequently called C interfaces around no more than four arguments that are eligible for core-register passing aligns with the ARM ABI's primary argument-passing mechanism and commonly minimizes stack use. The AAPCS32 specification defines r0–r3 as the argument/result/scratch registers and specifies that remaining arguments are copied to memory for a call. See the ARM Procedure Call Standard for the Arm Architecture and ARM's AAPCS32 documentation page.

QUESTION NO: 4

The Q-flag in the program status register (PSR) indicates which of the following?

A. Arithmetic overflow has occurred

B. Processor is in Thumb execution state

C. Imprecise data aborts are currently disabled

D. Saturation has occurred after execution of a saturated arithmetic instruction

ANSWER: D

Explanation:

The Q flag is the cumulative, or "sticky," saturation flag in the Application Program Status Register. It is set when a saturating arithmetic operation has had to clamp its mathematical result to the largest or smallest value representable by the destination operand size. For example, signed saturating add, subtract, and certain DSP-style arithmetic instructions can set Q when their unbounded result exceeds the signed range that can be stored. The flag provides software with a way to detect that saturation occurred without requiring a separate range test after every operation.

"Saturation has occurred after execution of a saturated arithmetic instruction" is therefore correct. The key point is that Q records the occurrence of saturation, rather than representing the ordinary signed-overflow condition for a single conventional arithmetic operation. It is cumulative: once set, it normally remains set until software explicitly clears it using an architecturally supported mechanism. This behavior is particularly useful in fixed-point signal-processing code, where saturation is often intentional but the application may still need to know whether any result reached a numeric limit. ARM documents Q as the cumulative saturation bit in the PSR/APSR and specify which saturating instructions update it. See the ARM Architecture Reference Manual and ARM's Cortex-M instruction set documentation.

QUESTION NO: 5

Which THREE of the following items should be preserved by software when entering dormant mode? (Choose three)

A. Current Program Status Register (CPSR)

B. Contents of the Level 2 data cache

C. The Floating Point Status and Control Register (FPSCR)

D. All User mode general-purpose registers

E. The CP15 Multiprocessor Affinity Register

F. Contents of the Level 1 data cache

ANSWER: A C D

Explanation:

Dormant mode is a power-management state in which processor context is not assumed to remain available to software after the processor has been powered down and later resumed. Therefore, software must save the execution context required to continue the interrupted program correctly. The **Current Program Status Register (CPSR)** must be preserved because it records essential processor state, including condition flags, interrupt masks, execution-state information, and processor mode. Restoring it ensures execution resumes with the same architectural status as before entry to dormant mode.

The Floating Point Status and Control Register (FPSCR) must also be preserved when floating-point or SIMD execution is in use. FPSCR contains floating-point control settings, exception status, and rounding behavior, all of which affect the correctness and repeatability of subsequent floating-point instructions.

All User mode general-purpose registers form the principal working context of the executing program. Saving these registers preserves program variables, pointers, stack-related values, and intermediate results that are held in registers when dormant mode is entered. Together, these saved values allow low-level resume code to reconstruct the program-visible CPU context before returning to normal execution. ARM power-management guidance describes dormant entry and restoration as software-managed context handling; see ARM Power Management Software and the ARM Trusted Firmware design documentation.

QUESTION NO: 6

Which of these processors is only available as a single core configuration?

A. Cortex-A5

B. Cortex-A8

C. Cortex-A9

D. Cortex-A15

ANSWER: B

Explanation:

Cortex-A8 is correct because it was designed and licensed as a single-core application processor. It implements the ARMv7-A architecture and uses a superscalar, in-order pipeline intended to provide substantial performance for smartphones, consumer electronics, and similar embedded applications of its era. Although a system designer can use multiple Cortex-A8 processors in a broader system design, Cortex-A8 itself was not offered as an ARM MPCore processor configuration with multiple CPU cores integrated into one processor cluster. Therefore, its architectural product configuration is single core.

This distinguishes Cortex-A8 from ARM processor families that were specifically designed with MPCore implementations, where one processor cluster can contain multiple cores and share resources such as an L2 cache and coherency logic. The Cortex-A8 product information identifies it as a single-core processor, while ARM's Cortex-A8 technical documentation describes the processor's implementation and interfaces in that single-core context. See ARM's Cortex-A8 product page and the Cortex-A8 Technical Reference Manual for product and implementation details.

QUESTION NO: 7

A processor prepares a cacheable memory buffer for a non-coherent DMA device to read. Which TWO actions are required before the device is started? (Choose two)

A. Clean the data cache lines containing the buffer to the Point of Coherency

B. Execute a Data Synchronization Barrier before starting the DMA operation

C. Invalidate the data cache lines containing the buffer before starting the DMA operation

D. Invalidate the instruction cache lines containing the buffer

E. Disable the MMU while the DMA transfer is active

ANSWER: A B

Explanation:

The data cache lines containing the buffer must be **cleaned to the Point of Coherency**. A processor write to a write-back cacheable buffer can remain only in a dirty data-cache line. Cleaning the relevant lines writes the updated contents to the point at which the DMA device can observe them.

Software must also execute a **Data Synchronization Barrier after the cache clean and before starting the DMA operation**. The DSB ensures that the cache-maintenance operation has completed to the required point before software programs the device to read the buffer. Invalidating the data cache before a device reads the buffer could discard processor-written dirty data, and instruction-cache maintenance is unrelated to DMA data transfers. See the Arm Architecture Reference Manual for A-profile architecture and the Arm Cortex-A Series Programmer's Guide.

QUESTION NO: 8

When a linker is removing unused sections during a static link (for example, -remove or -gc-sections), it finds the sections to keep by following all relocations starting from:

A. The entry point(s).

B. The function named 'main'.

C. All local functions and variables.

D. The reset vector.

ANSWER: A

Explanation:

The entry point(s). is correct. Section garbage collection works by identifying a set of root sections that must be present in the linked image, then marking every section reachable from those roots through relocation references. The program entry point is necessarily a root because execution begins there; its containing section must remain, as must sections referenced by relocations from it. This reachability process continues transitively, retaining the code and data needed by the startup and application path while allowing unreferenced input sections to be discarded. Embedded images can use an explicitly selected entry symbol or linker-script entry definition, and an image may have additional linker-defined roots depending on the link configuration. The essential principle tested here is that retention begins from the image entry point or points, rather than from a source-language convention. In particular, the linker operates on symbols, sections, and relocations after compilation; it does not require a C-specific application model to determine the initial root. ARM linker documentation describes removal of unused sections as a link-time optimization, while the GNU linker documentation provides the general relocation-reachability model used for section garbage collection. See Arm Compiler for Embedded documentation and GNU ld input-section garbage-collection documentation.

QUESTION NO: 9

A loop performs the same arithmetic operation independently on a large array of elements. Which TWO of the following are potential benefits of using Advanced SIMD instructions for this loop? (Choose two)

A. Multiple data elements can be processed by a single instruction

B. The number of arithmetic instructions can be reduced

C. All loops execute faster when vectorized

D. Memory bandwidth can no longer limit performance

E. Data dependencies between loop iterations are removed automatically

ANSWER: A B

Explanation:

Multiple data elements can be processed by a single instruction and **the number of arithmetic instructions can be reduced** are correct. Advanced SIMD registers contain multiple lanes, allowing one vector arithmetic instruction to apply the same operation to several independent elements. For suitable data-parallel code, this can increase throughput and reduce loop overhead.

Vectorization is not automatically beneficial for every loop. Dependencies between iterations, irregular control flow, data alignment, conversion overhead, and memory bandwidth can limit or remove the expected improvement. Advanced SIMD also does not eliminate the need to load input data and store results through the memory system. See the Arm ACLE Advanced SIMD intrinsics reference.

QUESTION NO: 10

A development board is supplied with a Board Support Package (BSP) for a particular operating system. Which TWO of these items would you expect to find in the BSP? (Choose two)

A. Power supply and electrical cables

B. Debugging hardware and software solution

C. System on chip peripheral driver source code

D. Boundary scan protocol definition

E. Boot code for board-specific devices

ANSWER: C E

Explanation:

A Board Support Package provides the software layer that adapts an operating system and its boot flow to a specific hardware board. **System on chip peripheral driver source code** is therefore expected: drivers configure and expose board-relevant SoC hardware, such as serial interfaces, storage controllers, network interfaces, interrupt controllers, timers, and GPIO, so that the operating system can use those peripherals correctly. BSP content commonly also includes the associated board configuration and device-description material needed to select and configure those drivers.

Boot code for board-specific devices is also a core BSP component. Before the operating system can start, platform boot software must perform board initialization appropriate to the hardware design. This can include setting up clocks, pin multiplexing, memory, console output, and board-specific storage or boot-device access, then loading and transferring control to the next firmware or operating-system stage. Arm's Trusted Firmware-A porting guidance describes platform-specific boot and initialization responsibilities, while the Yocto Project BSP guide identifies bootloader configuration and hardware-specific software support as BSP concerns. See Trusted Firmware-A Porting Guide and Yocto Project Board Support Package Guide.

QUESTION NO: 11

How is data written into NOR flash memory?

A. Data can only be written once, when the flash device is being manufactured

B. Writing data to the memory locations using store instruction, as you would with RAM

C. Reading and writing specific registers following a device-specific procedure

D. Using an external programming device, which utilizes an ultra-violet lamp to alter the data stored on the device

ANSWER: C

Explanation:

Reading and writing specific registers following a device-specific procedure is correct because NOR flash is a non-volatile memory technology that requires a defined program-command protocol rather than ordinary RAM-style writes. A processor normally accesses a NOR device through a memory-mapped interface, but it must first issue the command sequences specified by that device's manufacturer. These sequences select program mode, identify the target address or buffer, supply the data to be programmed, and then allow the device to signal completion through status polling or a ready/busy indication. The flash internally applies the required programming voltages and verifies the programmed cells. Erasure is also controlled separately, commonly at sector or block granularity, before locations can be reprogrammed as required by the device's erase/program rules. Consequently, software such as a bootloader or flash driver follows the particular NOR flash command set and timing requirements documented for the connected part. This controlled procedure is fundamental to NOR flash operation and is why systems use device-aware flash drivers rather than treating flash as directly writable RAM. See the Micron NOR Flash product information and Infineon NOR Flash documentation for NOR flash device families and their programming-command documentation.

QUESTION NO: 12

When an ARMv7-A MPCore system is in SMP mode, which of the following TWO operations can the processor handle automatically? (Choose two)

A. Coherency management between all L1 data caches

B. Broadcast of some inner-shared cache and TLB maintenance operations

C. Broadcast of some outer-shared cache and TLB maintenance operations

D. Coherency management between all L1 instruction caches

E. Coherency management between all external caches

ANSWER: A B

Explanation:

In an ARMv7-A MPCore implementation operating in SMP mode, the Snoop Control Unit provides hardware cache coherency for the participating processors' Level 1 data caches. Consequently, when one processor writes a cacheable shareable memory location, the coherency mechanism can snoop the other processors' data caches and ensure that they do not continue to use stale copies. This hardware support is the basis for coherent shared-memory multiprocessing, although software still needs appropriate memory ordering and synchronization instructions when sharing data.

The architecture also supports broadcast of applicable cache-maintenance and TLB-maintenance operations when they are issued with inner-shareable scope. This allows maintenance relevant to the inner-shareable domain to reach the processors that share that domain, rather than requiring software to individually invoke the operation on every processor. The exact maintenance operation and its required scope remain important: software must select operations appropriate to the memory attributes and shareability domain in use. ARM describes the SMP cache-coherency role of the Snoop Control Unit in the Cortex-A9 Technical Reference Manual; the architectural treatment of shareability, cache maintenance, and TLB maintenance is specified in the ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition.

QUESTION NO: 13

In which TWO of the following locations would a compiler typically place local variables? (Choose two)

A. ROM

B. Heap

C. Cache

D. Registers

E. Stack

ANSWER: D E

Explanation:

Registers and **Stack** are the normal implementation locations for automatic local variables in compiled ARM code. When a local value is short-lived, frequently used, and does not need a stable memory address, an optimizing compiler can keep it in a general-purpose or floating-point register. This avoids memory loads and stores and is typically the fastest representation for values such as loop counters, temporary arithmetic results, and simple scalar locals.

The stack provides memory storage for locals that need an address, are too numerous to fit in available registers, must remain live across calls, are aggregate objects, or are spilled by register allocation. A function establishes a stack frame as needed, and the compiler accesses stack-resident locals at fixed offsets relative to the stack pointer or frame pointer. The ARM Procedure Call Standard defines the stack conventions that allow independently compiled functions to interoperate, including the required alignment of the stack at public interfaces. Compiler optimization level, debug settings, variable type, lifetime, and whether a variable's address is taken all affect the final placement, so a single source-level local variable can be represented differently in different builds.

See the ARM Procedure Call Standard and Arm's Arm Compiler documentation for ABI and compiler implementation context.

QUESTION NO: 14

In an ARMv7-A system, the following C function calculates a simple checksum for an input data packet of variable length. The checksum is defined to be the sum of all of the 16-bit data items in the packet modulo 65536. The parameter data_items contains the number of 2-byte data items in the packet, and it cannot be zero by design.

When using an ARM compiler, which TWO of the following optimizations could improve the performance of this code? (Choose two)

A. Use a do/while loop instead of a for loop

B. Change the type of sum to be an unsigned short

C. Change the type of i to be an unsigned int

D. Use signed variables instead of unsigned variables

E. Declare sum as a global variable

ANSWER: A C

Explanation:

Use a do/while loop instead of a for loop can improve this loop because the packet is guaranteed to contain at least one item. A do/while construct expresses that invariant directly: the loop body executes before its terminating condition is tested. This can allow an ARM compiler to generate a loop with less initial control-flow overhead and, where appropriate, use an efficient count-down loop form. The actual improvement depends on compiler version, optimization level, and surrounding code, but this is a valid source-level optimization for a nonempty data set.

Change the type of i to be an unsigned int is also appropriate when it is compared with an unsigned item-count parameter. Matching the induction variable's unsigned type and natural machine word size avoids unnecessary signedness

handling and makes the loop's range semantics clear to the compiler. On ARMv7-A, an unsigned integer induction variable is naturally handled in general-purpose registers and can support efficient loop comparison and increment code generation. ARM compiler optimization guidance emphasizes writing loops with clear, consistent types and bounds so that the optimizer can recognize and transform them effectively. See the Arm Compiler for Embedded User Guide and the ARM Architecture Reference Manual for ARMv7-A and ARMv7-R.

QUESTION NO: 15

In the VFPv4-D32 architecture, which of the following best describes the arrangement of the registers?

A. D0..D31 and S0..S31 are separate register banks

B. D0..D31 overlap with S0..S63

C. D0..D15 overlap with S0..S31, and D16..D31 do not overlap with any single-precision registers

D. D0 overlaps with S0, D1 with S1 etc. up to D31 and S31

ANSWER: B

Explanation:

VFPv4-D32 provides a floating-point register file of thirty-two 64-bit doubleword registers, named D0 through D31. The same physical storage is also exposed as sixty-four 32-bit single-precision registers, S0 through S63. Each doubleword register is composed of a consecutive pair of single-precision registers: D0 contains S0 and S1, D1 contains S2 and S3, continuing in that pattern through D31, which contains S62 and S63. Consequently, the doubleword and single-precision names are overlapping views of one register bank, rather than independent banks. This aliasing is fundamental to VFP register access: an instruction operating on a D register accesses the same underlying bits that can be accessed through its two corresponding S-register names. The "D32" designation specifically identifies the implementation with the full set of thirty-two D registers and therefore the full S0–S63 single-precision register view. ARM's architecture documentation describes these VFP register naming and overlap rules in its floating-point register model. See the Arm Architecture Reference Manual and Arm's floating-point register documentation.

QUESTION NO: 16

In an ARMv7-A processor, which TWO registers are banked when the processor enters IRQ mode? (Choose two)

A. R8

B. R12

C. R13

D. R14

E. R15

ANSWER: C D

Explanation:

R13 and **R14** are banked in IRQ mode. IRQ mode has its own stack pointer, R13_irq, allowing an interrupt handler to use a dedicated exception stack without overwriting the interrupted mode's stack pointer. It also has its own link register, R14_irq, which holds the return address associated with IRQ exception entry.

R0 through R12 are normally shared between User, System, IRQ, Supervisor, Abort, and Undefined modes. FIQ mode is the exception because it additionally has banked R8 through R12, supporting low-latency interrupt handling. See the ARM Architecture Reference Manual for ARMv7-A and ARMv7-R.

QUESTION NO: 17

For Cortex-A series cores, what instruction(s) are recommended to implement a mutex or semaphore?

A. SWP and SWPB

B. DSB and ISB

C. LDREX and STREX

D. DMB

ANSWER: C

Explanation:

LDREX and STREX are the recommended instruction pair for implementing a mutex or semaphore on Cortex-A processors. They provide an exclusive load/store mechanism suitable for constructing atomic read-modify-write operations in shared memory. LDREX loads a value while marking the relevant memory location in the processor's exclusive monitor. Software can then inspect or modify that value, and STREX attempts to store the updated value only if no intervening write has invalidated the exclusive state. STREX reports whether the store succeeded, allowing software to retry the sequence when contention occurs.

This retry-based pattern is the fundamental building block for lock acquisition and release algorithms: a thread observes an unlocked state, attempts to change it atomically to locked, and repeats the attempt if another observer modified the location first. The exclusive-access instructions avoid reliance on the legacy swap instruction and are designed to work with multiprocessor cache-coherent systems. Correct lock implementations also place appropriate memory-ordering operations around acquisition and release as required by the algorithm, but the atomic mutual-exclusion primitive itself is formed by the LDREX/STREX exclusive-access sequence. ARM documents the exclusive monitors and exclusive accesses in the Arm Architecture Reference Manual; Cortex-A programming guidance is available from ARM documentation.

QUESTION NO: 18

Which TWO of the following statements about common ELF sections are correct? (Choose two)

A. The .text section normally contains executable code

B. The .bss section reserves storage for zero-initialized data

C. The .data section contains only read-only constants

D. The .rodata section contains the process stack

E. The .bss section stores initialized string constants in the executable file

ANSWER: A B

Explanation:

The .text section normally contains executable code and **the .bss section reserves storage for zero-initialized data** are correct. The .text section contains the machine instructions generated for program functions and is normally mapped with execute permission. The .bss section represents uninitialized static-duration objects that are initialized to zero before program execution begins.

Unlike .data, .bss normally does not need to occupy bytes in the executable file for each zero-initialized object. The loader or startup code reserves and clears the required memory. Initialized writable objects are normally placed in .data, while read-only constants are commonly placed in .rodata. See the System V Application Binary Interface: ELF specification.

QUESTION NO: 19

Cortex-A series processors contain event counting hardware which can be used to profile and benchmark code. The counters for these are programmed using:

A. Memory-mapped registers.

B. Generic Interrupt Controller (GIC) registers.

C. Debug Coprocessor Registers (CPU).

D. System Control Coprocessor Registers (CP15).

ANSWER: D

Explanation:

System Control Coprocessor Registers (CP15). is correct for the ARMv7-A Cortex-A processors to which this terminology applies. Their Performance Monitoring Unit (PMU) provides a cycle counter and programmable event counters that can count architecturally defined events, such as retired instructions, cache activity, and branch behavior. Software configures the PMU through its CP15 performance-monitor registers, principally in coprocessor register space c9. These registers control PMU enablement, select an event counter, assign the event type to count, reset or read counters, and enable counter-overflow interrupts where implemented.

This CP15 programming model allows privileged software, including an operating system, firmware, or a suitably authorized profiler, to establish a measurement interval and obtain hardware-derived execution statistics. The exact event set is processor-dependent, so software should consult the relevant Cortex-A technical reference manual when selecting event codes. ARM's ARMv7-A architecture reference material documents the performance-monitor register interface and its CP15 encodings: ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition. ARM also provides PMU guidance and event information in its Cortex-A processor documentation.

QUESTION NO: 20

In the Generic Interrupt Controller (GIC), when an interrupt is requested, but is not yet being handled, it is in which of the following states?

A. Inactive

B. Active

C. Pending

D. Edge-triggered

ANSWER: C

Explanation:

Pending is correct. In the GIC interrupt state model, an interrupt becomes pending when the interrupt source asserts a request, or when software sets its pending state. This records that service is required, but it does not mean the processor has started executing the interrupt handler. The GIC considers the interrupt's configuration, enable state, priority, target routing, priority mask, and interrupt-group/security settings to determine whether that pending interrupt can be signaled to a processing element. When the processor acknowledges the interrupt through the appropriate GIC CPU interface or system-register interface, the interrupt transitions to active as its handler begins execution. Thus, "requested, but not yet being handled" precisely describes the pending state. ARM documentation also recognizes a combined active-and-pending state when a further request arrives while an earlier instance of the same interrupt remains active; however, that is distinct from the condition described here, where the requested interrupt has not yet begun handling. See ARM's GIC Architecture Specification and the ARM GICv3/v4 Software Overview for the GIC interrupt lifecycle and state definitions.

QUESTION NO: 21

Which TWO of the following options can the ARM Compiler (armcc) directive__packed be used for? (Choose two)

A. To tell the compiler to use only Thumb code

B. To tell the compiler to produce code of minimum size

C. To tell the compiler to use the v6 SIMD pack/unpack instructions

D. To tell the compiler that an object can be on an unaligned address

E. To tell the compiler not to perform padding inside structures

ANSWER: D E

Explanation:

The ARM Compiler `__packed` qualifier is used to describe data whose alignment requirements are reduced to one byte. Therefore, it tells the compiler that an object can be on an unaligned address. This is particularly important when software accesses externally defined binary layouts, such as protocol headers, storage records, or hardware data structures, where fields might not begin at their naturally aligned addresses. The compiler can then generate the accesses necessary to safely handle that packed object according to the target architecture and compiler settings.

It can also be used to tell the compiler not to perform padding inside structures. Normally, a compiler inserts padding between structure members and at the end of a structure to meet each member's alignment requirements and to improve access efficiency. Applying `__packed` makes the structure layout compact by reducing member alignment, which permits the layout to match an exact byte-oriented format. ARM's CMSIS documentation describes packed structures as having alignment of one byte and identifies their role in controlling memory layout. See the CMSIS compiler-control documentation and the Arm Compiler reference for __packed.

QUESTION NO: 22

In an experiment, the time taken for an application to complete a given task is measured using a stopwatch. Which THREE of the following make up the total time? (Choose three)

A. The time spent waiting for I/O operations

B. The time taken to download the program via the debugger

C. The time taken for memory accesses

D. The time taken for the CPU to execute instructions

E. The time taken to compile the source code

F. The time taken to perform instruction tracing

ANSWER: A C D

Explanation:

A stopwatch measures elapsed, or wall-clock, time from the start of the application task until it completes. Consequently, this measurement includes *The time spent waiting for I/O operations*, because the application has not yet finished while it is waiting for storage, peripheral, network, or other input/output activity. It also includes *The time taken for memory accesses*. Instruction execution depends on fetching instructions and reading or writing data, and memory-system latency can contribute directly to the observed duration of the task. Finally, *The time taken for the CPU to execute instructions* is necessarily part of the elapsed time, since the processor must execute the application's instructions to perform the work and reach completion.

This distinction is important when interpreting performance results: elapsed time represents the end-to-end duration experienced for the workload, rather than only processor execution time. ARM performance analysis guidance distinguishes between time spent executing and time lost or delayed through effects such as memory-system behavior; measuring the complete task with an external timer captures both kinds of contribution. ARM's performance-analysis resources provide further context on measuring and optimizing software performance: Arm Performance Studio and Arm Performance Reports User Guide.

QUESTION NO: 23

On an ARM processor that does not implement Security Extensions, which one of the following can be the starting address of the exception vector table?

A. 0xFFFFFFFF

B. 0xFFFFFFF0

C. 0xFFFF0000

D. 0x0000FFFF

ANSWER: C

Explanation:

`0xFFFF0000` can be the starting address of the exception vector table. In the ARM architecture without the Security Extensions, exceptions use the vector base determined by the V bit in the System Control Register (SCTLR). With this control bit clear, the vector table begins at the low-vector base address, `0x00000000`. With the V bit set, it begins at the high-vector base address, `0xFFFF0000`. The individual exception vectors are then located at fixed offsets from that base, typically in 4-byte increments for the reset, undefined-instruction, supervisor-call, prefetch-abort, data-abort, IRQ, and FIQ entries. Thus, `0xFFFF0000` is an architecturally defined vector-table base, rather than merely an address within a possible vector region. This high-vector arrangement is useful where the low address range is reserved for other mappings or software requirements. The ARM Architecture Reference Manual documents the SCTLR V control and the low- and high-vector base locations; see the ARM Architecture Reference Manual and ARM's exception handling overview.

QUESTION NO: 24

A function written in C has the prototype:

void my_function(float a. double b, float c);

The function is built and linked into an application using hard floating-point linkage. What registers are used to pass arguments to the function?

A. a->s0; b->d0; c->s1

B. a->s0; b->d1; c->s1

C. a->d0; b->d1; c->d2

D. a->s0; b->d1; c-> s2

E. a->s0; b->d1; c->s4

ANSWER: E

Explanation:

With the hard floating-point procedure-call standard (the VFP variant of AAPCS), floating-point arguments are allocated to VFP registers in argument order. The first `float`, `a`, is passed in `s0`. A `double` occupies two consecutive single-precision registers and must start at an even-numbered `s` register. Since `s0` has already been allocated and the next available register would be `s1`, the allocation is rounded up to `s2`. The `double` argument `b` therefore occupies `s2` and `s3`, which are collectively named `d1`. Allocation then continues after both halves of that double, so the final `float`, `c`, is passed in `s4`. Thus the required mapping is `a->s0; b->d1; c->s4`. This alignment rule ensures that double-precision values are placed in an appropriately aligned VFP register pair and is part of the ARM Architecture Procedure Call Standard's VFP register-candidate allocation process. See the AAPCS32 specification and ARM's AAPCS32 source specification.