EECS 373 Midterm 1
Winter 2022

10 February 2022

No calculators or reference material.

Name

UM Uniqname

Sign below to acknowledge the Engineering Honor Code: "I have neither given nor received aid on this examination, nor have I concealed a violation of the Honor Code."

Signature

1 ABI (7 pts.)

```
void foo(int i, int adjustment, char z)

foo:
  push {r3, r4, r6, LR}
  mov r6, r1
  add r4, r0, #5
  mov r5, r2
loop:
  cmp r0, r4
  beq end
  push {r0}
  sub r2, r5, r6
  mov r0, r2
  bl print
  pop {r0}
  add r0, #1
  b loop
end:
  pop {r3, r4, r6, PC}
```

You are conducting a code review of a function, "foo", from Team Apple, one of two development teams you manage. It calls an ABI compliant "print" from Team B. The function "foo" has arguments, two integers and a character, and has no outputs. It prints the character, adjusted by the second integer argument, a total of 5 times. The function "print" takes a single character as an input and has no outputs. The "print" function has already been validated, and while Team Apple assures you that their function is functionally correct, they have a history of introducing defects into shipped products. Your job is to determine whether the "foo" function is a valid ABI-compliant function and to justify your answer.

Is it a valid, ABI-compliant function?

◯ Yes. ◯ No.

Justify in one terse sentence.

"No. The function is not ABI compliant because it uses a callee-saved register, r5, without preserving its contents."

Note that the function correctly restores the LR when it pops the PC at the end of the function. The function pushes caller-saved r3 onto the stack and pops it at the end; while unconventional, this is consistent with the ABI.

2 Assembly and memory layout (10 pts.)

The Table 1 represent two sections of memory in a Cortex-M processor. For convenience the memory values in the address 0x08000XXX block have been decoded into instructions. Label locations are indicated in italics. Assume that the Program Counter (PC) is 0x08000104 and the Stack Pointer is 0x20000000. Determine the value of R0, R1, R2, and the values of the memory addresses in the right table (i.e., 0x20000000 to 0x1FFFFFD0) when the PC = 0x08000114. Leave any unknown memory values blank.

The stack pointer starts at 0x20000000, so the next push will put the register contents at 0x1FFFFFFC.

Table 1: Memory and Register Contents

Address	**Instruction**	**Address / Register**	**Value**
0x08000104	MOV R0, #3	0x20000000	
0x08000108	MOV R1, #7	0x1FFFFFFC	**0x08000114**
0x0800010C	MOV R2, #3	0x1FFFFFF8	**0x7**
0x08000110	BL funA	0x1FFFFFF4	**0x3**
0x08000114	*done:* B done	0x1FFFFFF0	**0x6**
0x08000118	*funA:* PUSH {R0,R1,R2,LR}	0x1FFFFFEC	**0x2**
0x0800011C	*loop:* CMP R0, #0	0x1FFFFFE8	**0x5**
0x08000120	BEQ next	0x1FFFFFE4	**0x1**
0x08000124	POP {R0}	0x1FFFFFE0	**0x3**
0x08000128	BL funB	0x1FFFFFDC	**0x4**
0x0800012C	PUSH {R0}	0x1FFFFFD8	**0x0**
0x08000130	B loop	0x1FFFFFD4	
0x08000134	*next:* MOV R0, #6	0x1FFFFFD0	
0x08000138	*loop2:* CMP R0, #0		
0x0800013C	BEQ endA	R0	**6**
0x08000140	POP {R1}	R1	**3**
0x08000144	SUB R0, #1	R2	**7**
0x08000148	B loop2		
0x0800014C	*endA:* POP {R0,R1,R2,PC}		
0x08000150	*funB:* POP {R1,R2}		
0x08000154	PUSH {R1}		
0x08000158	PUSH {R2}		
0x0800015C	PUSH {R0}		
0x08000160	SUB R0, R1, #1		
0x08000164	BX LR		

3 Interrupts (10 pts.)

1. Use at most two sentences to describe the main purpose of the Nested Vector Interrupt Controller (NVIC).

   The NVIC controls the proper handling of interrupt requests, such as resolving priorities, handling interrupts that occur while other interrupts are being executed, and restoring code to its previous state once the interrupts have been handled. Other functions handled by the NVIC are tail-chaining and stack management (when transitioning states). The NVIC also reduces latency because it can perform operations at the same time as the main processor.

Many answers described the role of the Interrupt Vector Table (IVT): a map that stores the addresses of the interrupt routines and provide a way to know what function should be executed given an interrupt request. This does not completely answer the question though: the IVT is a part of the NVIC, but the IVT simply stores values while the NVIC actively coordinates execution of interrupt requests. Partial credit was given to students who gave this answer.

2. Use at most two sentences to explain why it is good practice for interrupt handlers to terminate quickly.

   Answer: If an interrupt handler has a large latency, it can interfere with time-sensitive tasks like data collection and serial communication. Depending on how time-sensitive the interrupted tasks are, this can make the resulting data incorrect/unusable or result in a catastrophic error in the code's operation.

   The answer required students to specify that a task was "time-sensitive" rather than "important" or "a main task", because there are many situations in which it is perfectly acceptable for tasks to take a long time to execute when there is no time sensitivity.

   An answer that received slight penalty was that longer execution times waste energy. This is true, but not the focus of the question or the main reason for interrupt handlers to be quick.

   Another acceptable answer was that high latency increases the likelihood that two tasks access the same data and make it unusable. The standard way to resolve this type of issue is to use "semaphores". If you are curious, more information on semaphores can be found here:
   https://www.baeldung.com/cs/semaphore

3. On a Cortex M4 processor there are two IRQ handling routines, named IRQ A and IRQ B. IRQ A is set to trigger when signal X transitions from low to high, and has an execution time of 1 clock cycle. IRQ B is set to trigger when signal Y transitions from low to high, and has an execution time of 2 clock cycles. Each of these interrupt handlers sets GPIO A and B, respectively, to HIGH at the start of execution and LOW and the end of execution. Given the following waveforms for X and Y, draw the waveforms for GPIO A and GPIO B given the following preemption priority assignments, where a lower number indicates a higher priority. Assume the processor uses tail chaining. Appended are some excerpts from the ARMv7 Architecture Reference Manual that may be useful.

   Figure 1: The main point was that subpriorities break ties, but cannot preempt other tasks of equal or higher preemption priority.

   Figures 2 and 3: This question could have been answered in two ways depending on whether the exception handlers were synchronous or asynchronous; we took both answers. If the exception handler were synchronous, the interrupt request at t=3 would be ignored, while if it was asynchronous handler B would take on the "active & pending" state at t=3. Although the two functions were tail chained together, the GPIO would still have a small dip during the chaining because inside of the handler it must still pass over the "set GPIO to low" and "set GPIO to high" at the end and start of execution. However, a mistake on this part could easily be due to slight misinterpretation of the question so very few points were deducted for not having a dip. One aspect of the question that caused confusion was whether GPIO's signal should go low while GPIO A preempted it. Because handler B only sets the GPIO to low at the end of execution, and preemption interrupts it in the middle of execution, there is no code that can tell GPIO B to go low during this process.

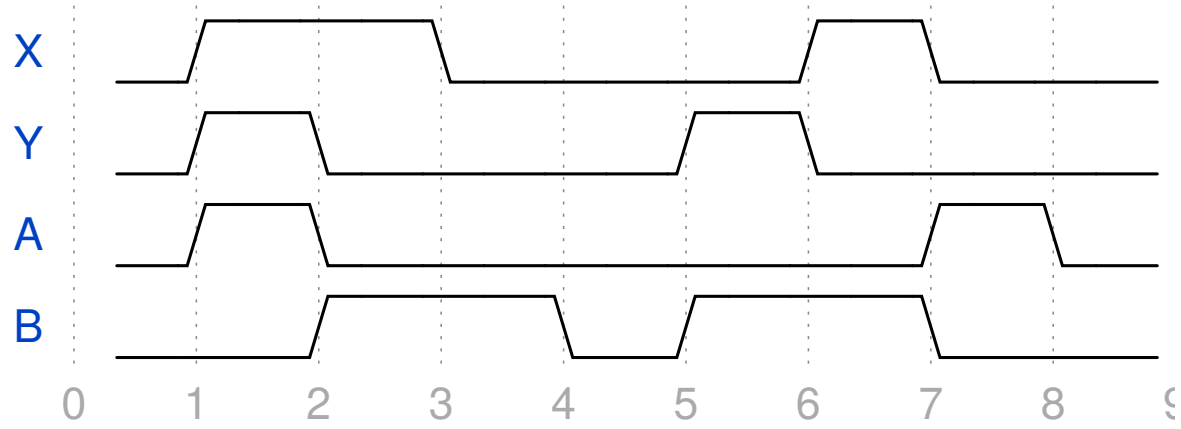

Figure 1: Timing diagram in which GPIO A has a preempt priority of 0 and a subpriority of 0 and GPIO B has a preempt priority of 0 and a subpriority of 1.

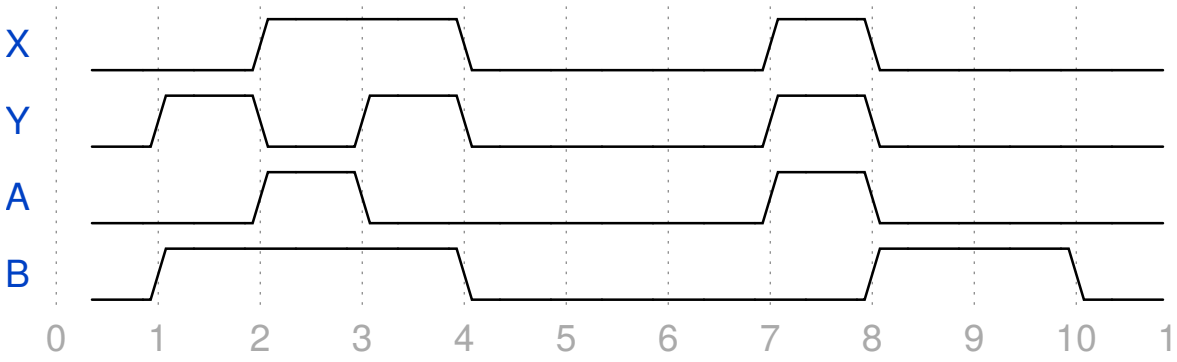

Figure 2: Synchronous version of timing diagram in which GPIO A has a preempt priority of 0 and a subpriority of 1 and GPIO B has a preempt priority of 1 and a subpriority of 0.

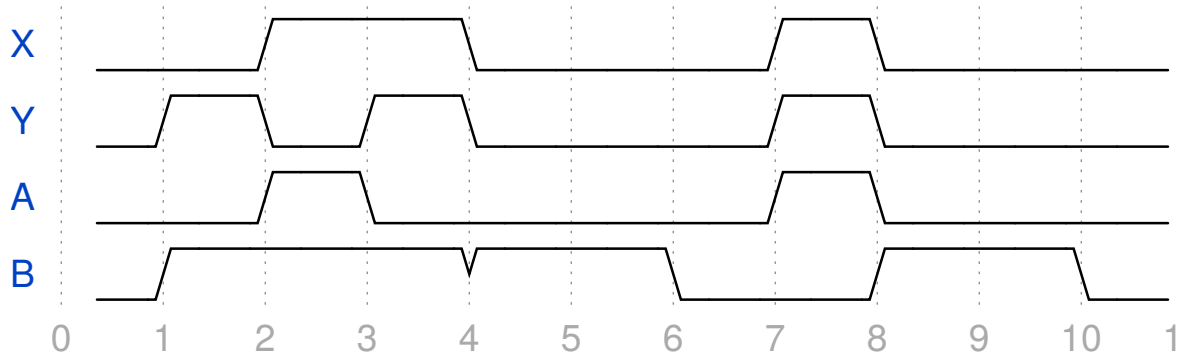

Figure 3: Asynchronous version of timing diagram in which GPIO A has a preempt priority of 0 and a subpriority of 1 and GPIO B has a preempt priority of 1 and a subpriority of 0.

4 Build process (8 pts.)

Add directed edges between nodes in Figure 4 to illustrate the flow of information, i.e., data or metadata such as file modification times, in the standard embedded system build process. Don't include edges for command executions, e.g., "Make" should not have an arc to "Linker". We have added a few correct edges to help you get started.

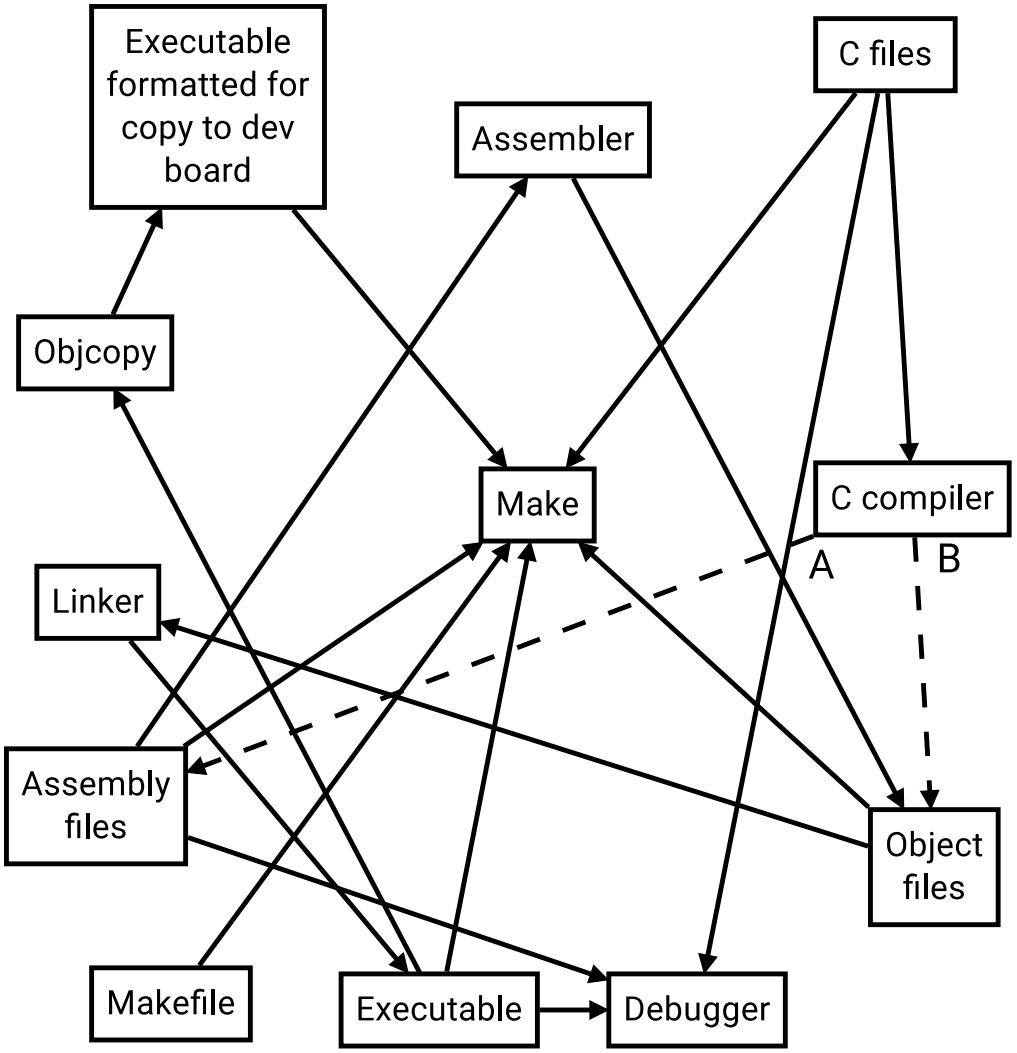

Figure 4: Incompletely specified embedded development board build process.

Information flow implies data or metadata flowing from one file or application to another. For example, the Make application reads the Makefile but the Makefile does not read or accept information from the Make application. Directionality is crucial for this question, so undirected edges were not given credit. If the distinction between directed and undirected graphs is not clear, please see me in office hours: I can explain the different types of graphs that engineers commonly encounter. I accepted answers indicating that the C compiler produces an assembly file (A option in the figure) or an object file (B option in the figure). The second is possible as an optimization for commonly encountered build systems. To better understand this material, I recommend reviewing the in-class example files posted to the website, the manpages for the relevant commands, e.g., make, and the associated lecture videos. I can also explain the process during office hours.

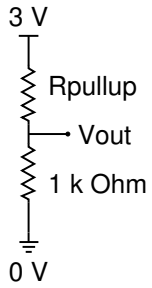

Figure 5: Circuit diagram for open collector bus line.

5 Debugging (4 pts.)

You have implemented an ISR to handle a button press interrupt. The ISR uses a persistent (static) variable in memory to track the state of an LED; it inverts (!=) that variable and writes the new value to an LED interface on the APB, turning on or off the LED. However, on boot the LED is off and pressing the button never turns the LED on, or off. What is the first thing you would do with a debugger to determine whether the ISR is executing? Use at most one sentence.

Many solutions were accepted. Setting a breakpoint at the ISR is what I would expect most engineers to try first, but there were other legitimate answers, which I gave credit for.

6 Open-collector style buses and voltage division (5 pts.)

You have implemented an open-collector style bus. In your system, V_{DD} is 3 V. By convention, voltages below 1 V signify activity and voltages above 2 V signify inactivity. Several components are connected to a bus line, each controlling a transistor that electrically connects the line to ground when activated. However, by using a multimeter, you find that when a single component's transistor is turned on, the line voltage decreases to 1.5 V and never drops below 1 V. Your component interface drivers have output resistances of 1 kΩ.

1. What is the pull-up resistance on the bus line?
2. What pull-up resistance would enable an active device to reduce the voltage to 0.5 V.

   There are two parts to this problem: understanding the high-level problem and doing the basic circuit analysis necessary to solve it.

   At a high level, it is necessary to understand what an open-collector bus style is and thus understand that the circuit diagram in Figure 5 is under consideration. Please review the lectures and lab material on open-collector bus design style to better understand this. I would also be very happy to help review it in office hours.

   The key detailed concept required to answer this question is an understanding of voltage dividers, i.e., determining how voltage is divided by series resistors. The general expression (assuming a grounded node) follows.

$$V_{out} = \frac{V_{top} \cdot R_{bottom}}{R_{bottom} + R_{top}}. \tag{1}$$

   For this problem, $V_{top} = 3\,\text{V}$ and $R_{bottom} = 1\,\text{k}\Omega$ allowing us to simplify Equation 1 as follows:

$$V_{out} = \frac{3\,\text{V} \cdot 1\,\text{k}\Omega}{1\,\text{k}\Omega + R_{pullup}}. \tag{2}$$

We can solve the first sub-problem by fixing V_{out} to 1.5 V and solving for R_{pullup}.

$$1.5\,\mathrm{V} = \frac{3\,\mathrm{V} \cdot 1\,\mathrm{k\Omega}}{1\,\mathrm{k\Omega} + R_{pullup}}, \tag{3}$$

$$1\,\mathrm{k\Omega} + R_{pullup} = \frac{3\,\mathrm{V} \cdot 1\,\mathrm{k\Omega}}{1.5\,\mathrm{V}}, \tag{4}$$

$$R_{pullup} = \frac{3\,\mathrm{V} \cdot 1\,\mathrm{k\Omega}}{1.5\,\mathrm{V}} - 1\,\mathrm{k\Omega}, \textbf{ and} \tag{5}$$

$$R_{pullup} = 1\,\mathrm{k\Omega}. \tag{6}$$

The second sub-problem can be answered by solving Equation 2 for R_{pullup} with $V_{out} = 0.5\,\mathrm{V}$.

$$1\,\mathrm{k\Omega} + R_{pullup} = \frac{3\,\mathrm{V} \cdot 1\,\mathrm{k\Omega}}{0.5\,\mathrm{V}}, \tag{7}$$

$$R_{pullup} = \frac{3\,\mathrm{V} \cdot 1\,\mathrm{k\Omega}}{0.5\,\mathrm{V}} - 1\,\mathrm{k\Omega}, \tag{8}$$

$$R_{pullup} = 6\,\mathrm{k\Omega} - 1\,\mathrm{k\Omega}, \textbf{ and} \tag{9}$$

$$R_{pullup} = 5\,\mathrm{k\Omega}. \tag{10}$$

If this wasn't clear, the concept can be reviewed in an introductory physics textbook. I would also be very happy to explain it in detail in office hours.

7 MMIO and logic (4 pts.)

You are designing an APB interface for an ultra-low-power device that supports reads from the following MMIO addresses: 0x0, 0x5, 0x6, 0x7, and 0xd. For each address, indicate the **minimal** number of bits that must be used as inputs to an AND gate used for detecting an access to that address. Do not consider sharing AND gate logic: each address gets its own AND gate. Do not assume that only the lowest-order bits are used: you may skip bits. You needn't consider aliasing with other devices because PSEL can handle that. Consider aliasing among the device's own addresses. You must show your work to receive credit.

1. 0x0
   ◯ 0 ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 8
2. 0x5
   ◯ 0 ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 8
3. 0x6
   ◯ 0 ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 8
4. 0x7
   ◯ 0 ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 8
5. 0xd
   ◯ 0 ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 8

There are two parts to this problem: understanding the high-level problem and doing the discrete math necessary to solve it.

The high-level problem is to distinguish between each of the addresses and all the alternatives. Fortunately, we have PSEL to rule out aliasing of addresses outside the range for this peripheral so we only need to distinguish each device address from the four other device addresses. This also assumes that invalid addresses won't be used, a reasonable assumption in this case.

The low-level problem is distinguishing numbers. Let's rewrite them in binary for the sake of illustration.

(0x0)	**0000**	**1**
(0x5)	**0101**	**3**
(0x6)	**0110**	**2**
(0x7)	**0111**	**2**
(0xd)	**1101**	**1**

Considering 0000 column by column (0–3) , we see that its entry is unique in column 2: no other number has a 0 in this column, i.e., X0XX, where X indicates a don't-care condition. Therefore, we need a one-input AND gate implementing function I_2'. For 0101, we see that no single column of the number is unique, and therefore consider two-column cases. It is not unique for any combination of two specified columns so we move on to three columns, finding that 101X distinguishes it from other numbers. There may also be other answers, but we have already demonstrated that no answer using fewer than three inputs is sufficient and that there exists an answer using three inputs. Similar reasoning can be used for the remaining rows.

If this didn't make sense to you, you can review my entry in the Encyclopedia of Algorithms on "Optimal Two-Level Boolean Minimization". It covers the terminology and foundational concepts needed to understand specification and minimization of Boolean functions. I took great pains to keep it terse, formal, and precise so it shouldn't take much reading to finish, although it may take some thinking to digest.

8 Cat (1 pt.)

Indicate the concept Figure 6 represents.

- ◯ Non-volatile memory.
- ◯ The customer's view of me.
- ◯ The customer's view of my early-stage product idea.
- ◯ My lab partner.
- ◯ The APB.

"The customer's view of my early-stage product idea." The problem I was trying to illustrate in lecture is that potential customers may see substantial flaws in the product definition but be unwilling to share them because doing so might seem impolite to the engineer, who they will generally have a favorable view of if they agreed to an interview. Your job is to structure the interview to avoid this source of biased answers to learn the cold, hard truth about actual customer needs.

Figure 6: An illustration of...

Overflow space. We won't look at this space unless you tell us to after the relevant question.

NVIC operation

Armv7-M supports level-sensitive and pulse-sensitive interrupt behavior. This means that both level-sensitive and pulse-sensitive interrupts can be handled. Pulse interrupt sources must be held long enough to be sampled reliably by the processor clock to ensure they are latched and become pending. A subsequent pulse can add the pending state to an active interrupt, making the status of the interrupt active and pending. However, multiple pulses that occur during the active period only register as a single event for interrupt scheduling.

B1.3.2 Exceptions

Each exception has:

- An exception number.
- A priority level.
- A vector in memory that defines the entry point for execution on taking the exception. The value held in a vector is the address of the entry point of the exception handler, or *Interrupt Service Routine* (ISR), for the corresponding exception.

An exception, other than reset, has the following possible states:

Inactive An exception that is not pending or active.

Pending An exception that has been generated, but that the processor has not yet started processing. An exception is generated when the corresponding exception event occurs.

Active An exception for which the processor has started executing a corresponding exception handler, but has not returned from that handler. The handler for an active exception is either running or preempted by the handler for a higher priority exception.

Active and pending

One instance of the exception is active, and a second instance of the exception is pending.

Only asynchronous exceptions can be active and pending. Any synchronous exception is either inactive, pending, or active.

Exception return

The processor executes the exception handler in Handler mode, and returns from the handler. On exception return:

- If the exception state is active and pending:
  - — If the exception has sufficient priority, it becomes active and the processor reenters the exception handler.
  - — Otherwise, it becomes pending.
- If the exception state is active it becomes inactive.
- The processor restores the information that it stacked on exception entry.
- If the code that was preempted by the exception handler was running in Thread mode the processor changes to Thread mode.
- The processor resumes execution of the code that was preempted by the exception handler.

The Exception Return Link, a value stored in the link register on exception entry, determines the target of the exception return.

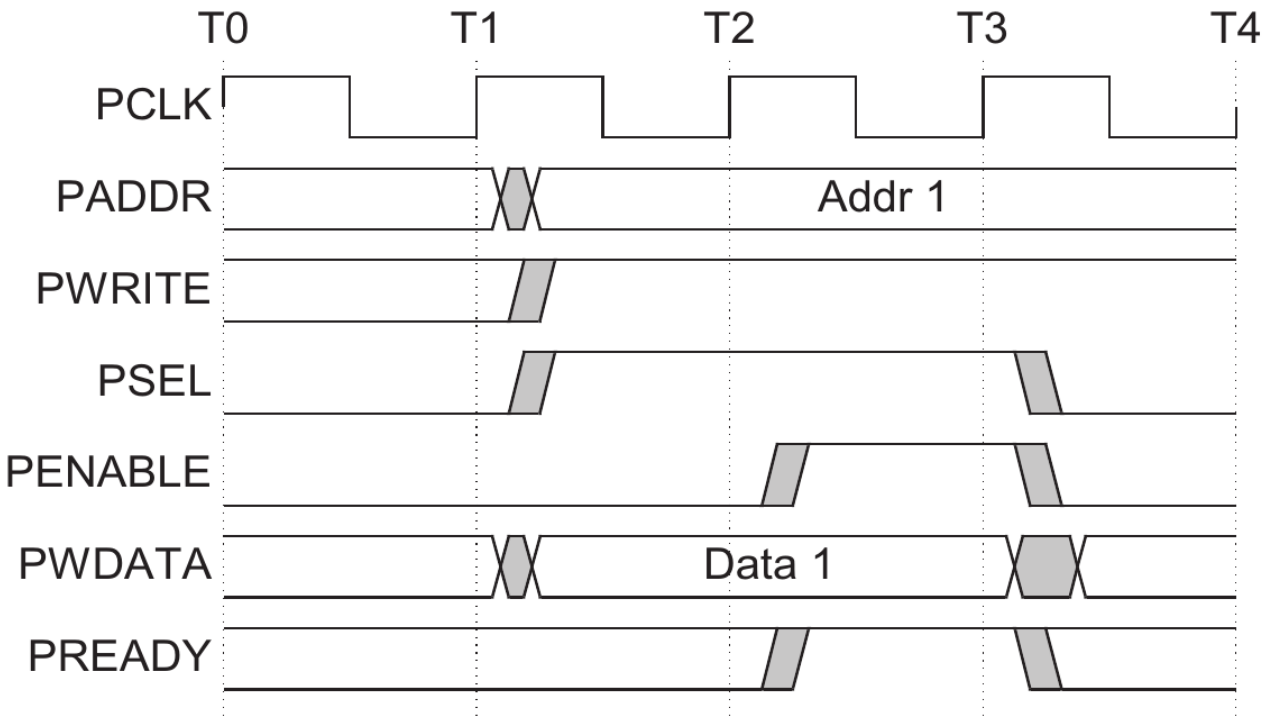

Figure 3-1 Write transfer with no wait states