CS 355 - Advanced Computer Architecture
Assignment 7

Due date: see Canvas
Note: this is a paper assignment and it is due in class

1. **Timing diagram of a `WRITE` operation (30 pts)**

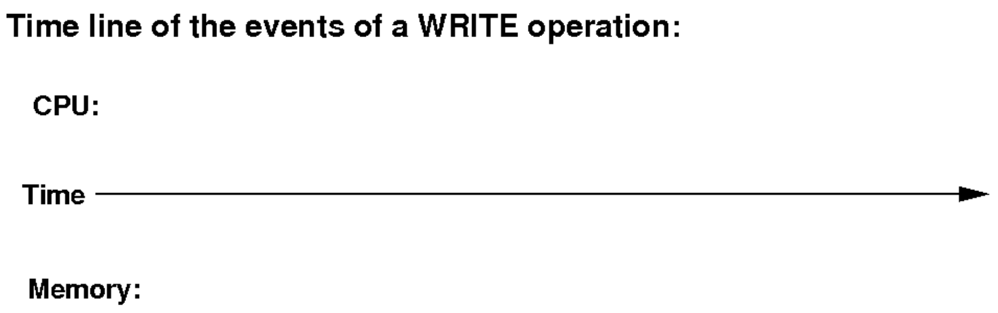
   - The following are the **events** that must occur in a `WRITE` (memory) **operation**:

     - Send out the address of memory
     - Send out the data
     - Send out the `WRITE` request
     - Write memory
     - Stop sending out the address of memory
     - Stop send out the data
     - Stop send out the `WRITE` request

   - **Place** the **events** on the following **time line** such that the `WRITE` **operation** will be **successful** (10 pts)

     The **events** that occur in the **CPU** must be placed **above** the **time line**

     The **events** that occur in the **memory** must be placed **below** the **time line**



   - Give the **timing diagram** of a **synchronous `WRITE` protocol** where the **memory** has **no `WAIT` state** (i.e., it's fast enough) and the **memory** is **written** at the **start** of the **clock period**. (10 pts)

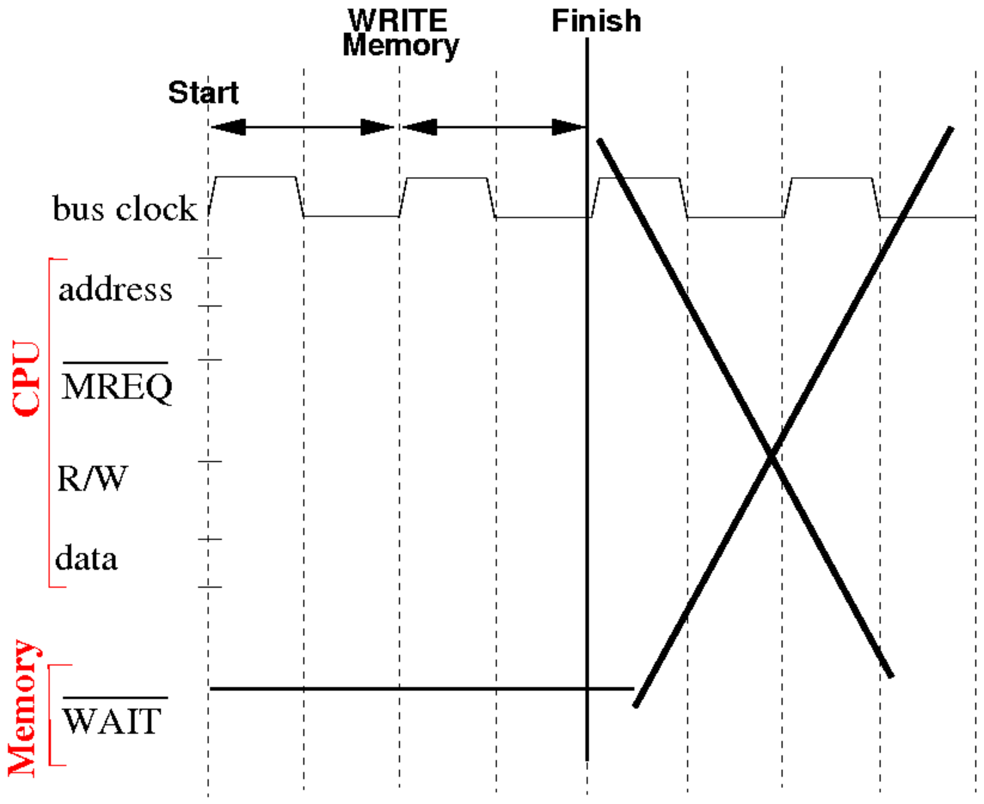
     The **value** of the $\overline{\texttt{MREQ}}$ and `R/W` signal at the **start** of the **bus cycle** are equal to **ONE (= 1)**

     The **signal** $\overline{\texttt{MREQ}}$ `= 0` is used to **indicate** the **memory** is **selected**

     To make your **answer clearer** for **grading purposes**, we will use `R/W = 0` to **indicate** the **`WRITE` operation**

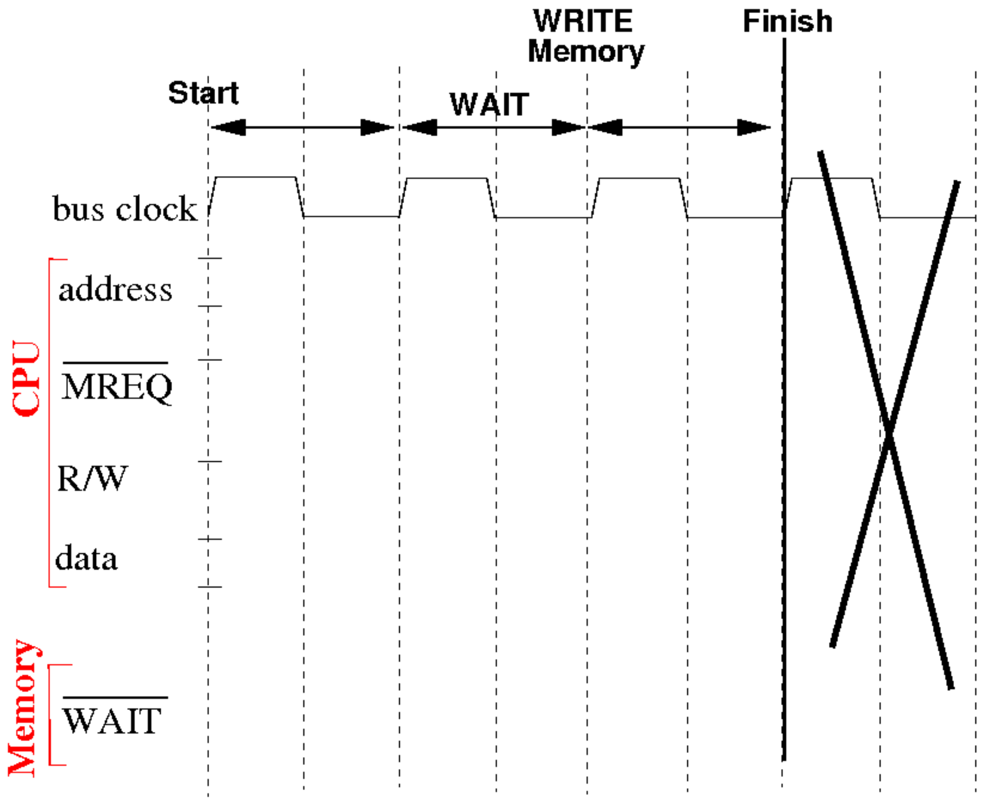
     Note: I have **drawn** in the **`WRITE` signal** in the **answer sheet** already. You just need to draw in the **other signals** in the figure (on next page)

     Note: I have **marked** the **time** that the **memory is written** in the **timing diagram** with `WRITE Memory`



- Give the **timing diagram** of a **synchronous `WRITE` protocol** where the **memory** has **1 clock period `WAIT` state** and the **memory** is **written** at the **start** of the **clock**. (10 pts)

  Note: make sure to **draw** in the **`WRITE` signal** in the **answer sheet** !!

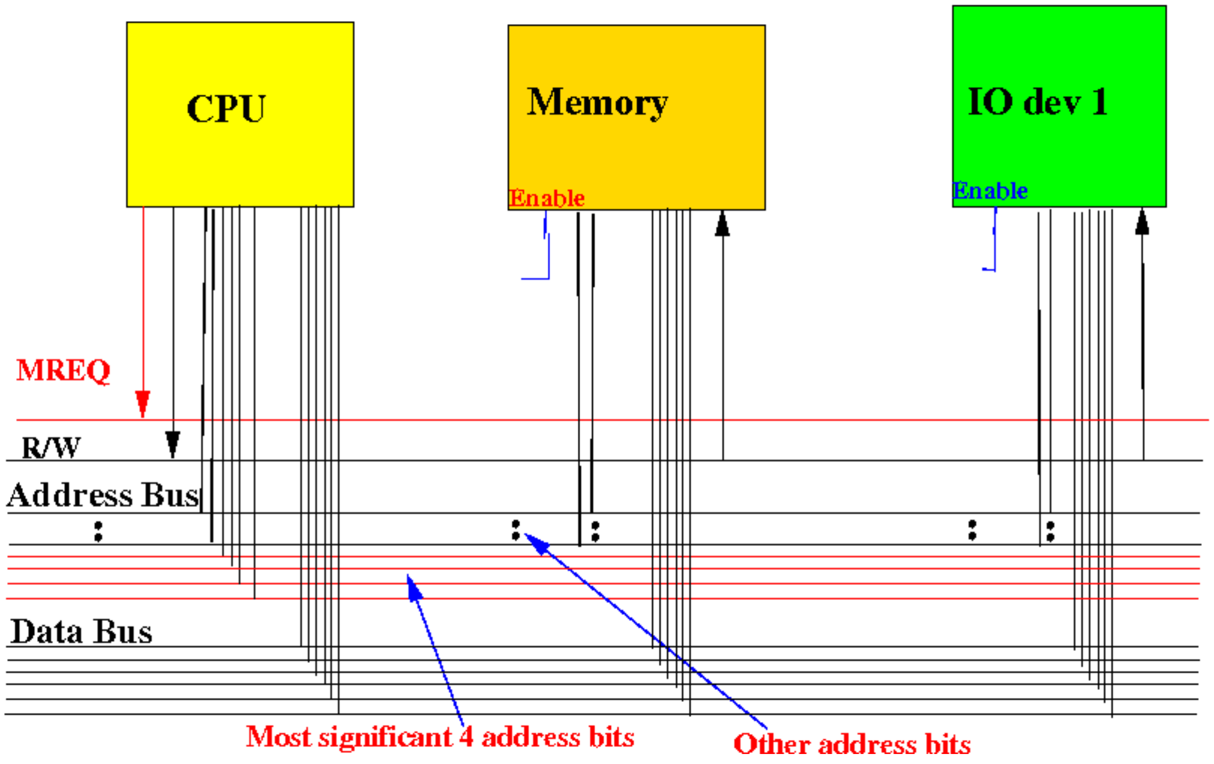


2. **Memory-mapped IO (10 pts)**

   - A **CPU** is using **memory-mapped IO** where the **I/O devices** are assigned an **address value** that **starts** with `0000`

     Address values that **do not** start with `0000` are **memory addresses**

     Provide the **circuit** that will **enable** an **IO device** and the **memory** based on the description above. (10 pts)



3. **Interrupt handling (40 pts)**

- What is the **name** of the **signal** that is **asserted** by an **IO device** to request interrupt ? (3 pts)

- What is the **name** of the **signal** that is **asserted** by the **CPU** answers the interrupt request ? (3 pts)

- Give the complete **Instruction Execution Cycle** of the **CPU** (4 pts)

- Can the computer system **guarantee** that a **computer program** will **not** use **any** of the **CPU registers** when an **interrupt** occurs (i.e., some device sends the interrupt signal) ? (2 pts)

  Circle one: Yes   /   No

- If your **answer** was **no** to the **previous question**, what **mechanism** is used to **ensure** the **CPU registers** used by a **computer program** preserved **preserved** for the **program** ? (8 pts)

  Make sure you give the **name** of the **mechanism** and the **data structure** used in the procedure.

- A **step** in the **Vector Interrupt mechanism** consists of:

  - the **transmission** of some **signal** (e.g.: **interrupt request** signal, **memory data**, **(interrupt) vector number**, **interrupt acknowledge** signal, etc., etc.) by some **device** (e.g.: **CPU**, **IO device** or **memory**)

  In response to the **signal** sent in a step, **some device** will **respond** by sending **another signal**

  This process of signal responses will end when the **CPU calls** the (**correct**) **interrupt handler**

  Starting with the **IO device sending** out an **interrupt request**, **describe** the **steps** in the **Vector Interrup mechanism** by stating: (20 pts)

  1. **Which device** will **send** the **response signal**
  2. **What** is the **response signal**

  in **each step** of the **Vector Interrupt mechanism**

  1. **IO device:**

     sends out the **interrupt request** (signal)

  2. **Action(s)** taken in **response** by the ____________________**:**

     sends out**:**

  3. **Action(s)** taken in **response** by the ____________________**:**

     sends out**:**

  4. **Action(s)** taken in **response** by the ____________________**:**

     sends out**:**

  5. **Action(s)** taken in **response** by the ____________________**:**

     sends out**:**

  6. **Action(s)** taken in **response** by the ____________________**:**

     sends out**:**

  7. **Action(s)** taken in **response** by the ____________________**:**

     sends out**:**

  Last step: Action(s) taken in **response** by the **CPU**:

  perform a **subroutine call (`bl`)** to the **interrupt handler** in **memory** at the **value** sent by the **memory**

Note: there may be **too many steps** listed in the answer above. Use as many steps as you need. (Leave the steps you don't use open).

4. **Interrupt handling routine (20 pts)**

   - Recall the timer interrupt handler implements **multi-programming** where multiple processes share the CPU. The interrupt handler discussed in class will allow each process to run for exactly **one** "time slice" (= timer interval) and switch the CPU to another process in the Ready Queue..

   - **Suppose** we want assign **priority** to **running programs (= processes)** where a **process** with a **high priority** can run for a **longer time (= use more time slices)** on the **CPU**

     To **implement** this **priority scheme**, we add the following **variables** to the **PCB**:

     - `int Priority`
       - This variable contains the **number** of **time slices** that the **process** is allowed to **used** before it must **relinguish** the **CPU** to the **next process** in the Ready Queue (so the **next process** can **run**)
       - This variable is initialized by the **Operating System** when the **process** is **started** and remains **constant (= unchanged)** throughout the execution of the process
     - `int Remaining`
       - This variable contains the **number** of **time slices remaining** that the **process** is allowed to **used** before it must **relinguish** the **CPU** for another **process** (so the **other process** can **run**)
       - This variable is initialized to the **value** in the `Priority` in the **PCB** when it **first** receive the **CPU**
       - This variable is **decremented each time** there is a **timer interrupt**
       - When this variable **becomes 0 (= zero)**, the **next process** in the **Ready Queue** will receive the **CPU**

   - **Write** the **Interrupt Handling Routine** for the **Timer device** that implements this **priority scheme**. (20 pts)

     Use these **variables** and **method calls** to write the **pseudo code** of the **Interrupt Handling Routine** for the **Timer device**:

     - Use `ReadyQueue` for the **Ready Queue**
     - `ReadyQueue.PCB` is the **first PCB** in the `ReadyQueue`
     - `PCB.Priority` and `PCB.Remaining` are the `Priority` and `Remaining` variables in a `PCB`
     - `RemoveFirst(ReadyQueue) will remove the first PCB from the ReadyQueue and return it.`
     - `InsertTail(PCB) will insert the PCB at the tail (end) of the ReadyQueue`
     - `SaveContect(ReadyQueue) saves the context information (in the CPU) into the first PCB in the ReadyQueue`
     - `RestoreContect(ReadyQueue) restores the context information in the first PCB in the ReadyQueue into the CPU`

     Answer:

Use this page if you need extra space for the answer