2.10 Basic Input/Output Synchronisation

Hardware and software aspects of input/output synchronisation

EE302 – Real time and embedded systems

1

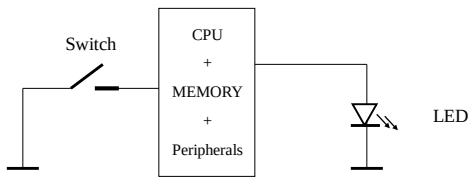
Overview

- Aims
  - Introduce the basic concept of I/O synchronization and the polling method in particular.
- Learning outcomes – you should be able to…
  - Explain I/O synchronization, why its needed, and the polling method for handling it
  - Differentiate between bus aware and non-bus aware devices
  - Explain the purpose of latches for non-bus-aware I/O
  - Write pseudocode/code to poll I/O

2

What is I/O?

3



- To be useful a computer must be able to communicate with the outside world
- Simple example
  - input from a switch (one bit of information)
  - Indicate output via an LED (also one bit of information)

3

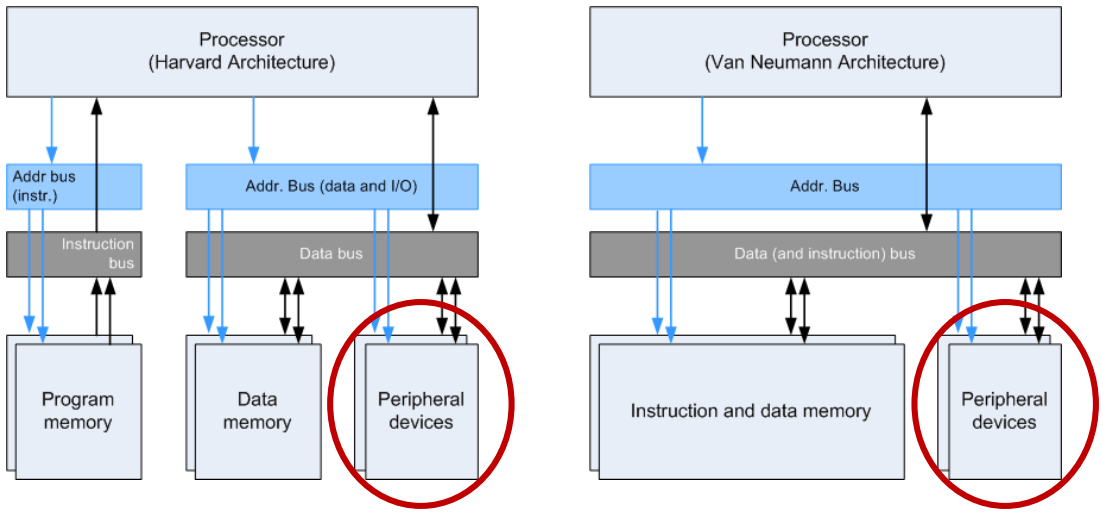
Interfacing to a CPU

4

- A CPU can only receive input **data** and send output data directly via the **data bus** CPU只能通过数据总线直接接收输入数据和发送输出数据
- You can only connect the address and data bus to things which are bus-aware (i.e. understand the bus signalling and timing), e.g.
- Therefore, the CPU's Interface to external world is mainly indirect via peripheral devices attached to the data bus

4

I/O in our computer architecture



5

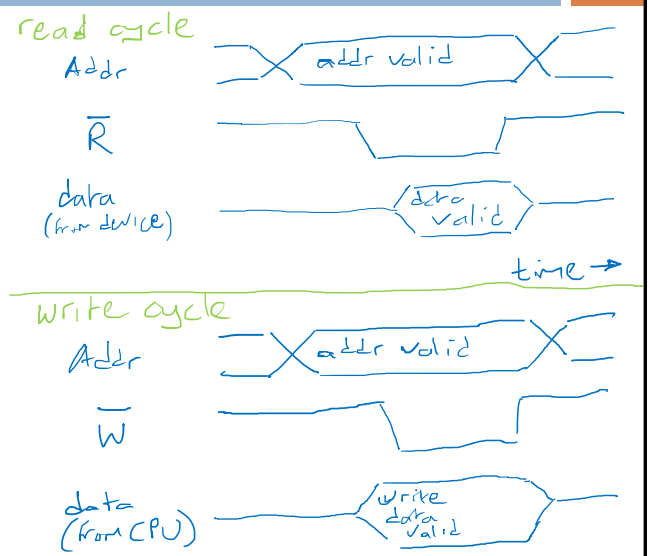
A processor's bus and control lines

- Multiple parallel data lines called the data bus
  - All connected devices should be in high impedance state (essentially disconnected) except one at most
  - The data bus must only be used while data is valid (based on control signal timing)
- Multiple parallel address lines called the address bus
  - The lines are connected to address decoder logic which will activate just one chip select line based on the address value
  - The chip select line identifies which connected device should be active – all others should be in the high impedance state
- Read and write control (multiple approaches possible), e.g.
  - A read line – active indicates that the processor is requesting data, inactive otherwise
  - A write line – active indicates that the processor is sending data, inactive otherwise
  - The read and write control lines will not be active at the same time. Active low lines are low when active and high when inactive. Active high lines are the high when active and low when inactive.

6

Representative bus signals and timing

- CPU read from device
  - Change the address lines, short delay
  - activate the read line, short delay
  - CPU reads the data line logic levels, short delay
- CPU write to device
  - Change the address lines, short delay
  - activate the write line, short delay
  - CPU sets the data line logic levels, short delay



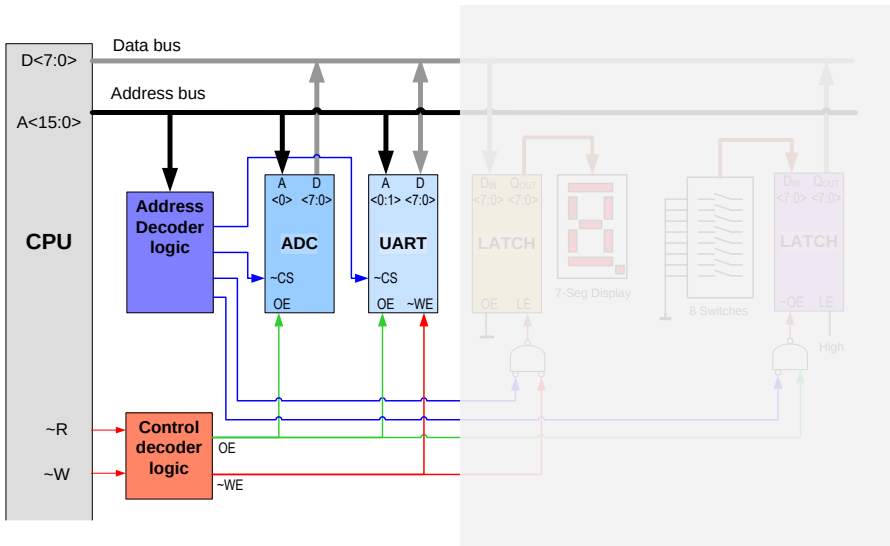
7

Bus-aware vs non-bus-aware devices

- **Bus-aware** devices
  - Must have tri-state outputs so that it can be connected directly to data and address bus of CPU
  - E.g. I/O ports, timers & clocks, communications devices, A/D and D/A convertors, etc.
  - May be internal to processor package (e.g. in a microcontroller) or external (e.g. in a GPP)
- **Non bus-aware** devices and hardware
  - Cannot be connected directly to data and address bus of CPU. Connected instead via a bus-aware peripheral, usually a digital I/O port which is largely based on a latch
  - E.g. switches and LEDs are not bus-aware

8

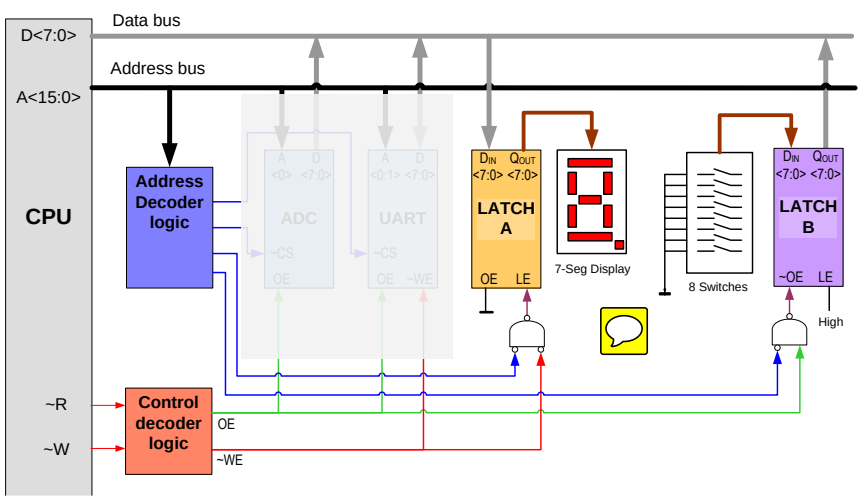
Interfacing bus-aware devices to a CPU



Bus-aware devices

9

Interfacing non-bus-aware hardware to a CPU



Non bus-aware devices

10

Interfacing devices to a CPU – hardware signalling

For the sake of example consider how the processor/CPU would read the state of the input switches (connected via LATCH B)

1. The CPU outputs a valid address on the address bus. In this case it must be the address of the LATCH B.
2. The address decoder logic decodes the address (on the address bus lines) such that at most one device is selected, in this case LATCH B, by activating the appropriate chip select (CS) line.
3. After a short time the CPU activates its read control line (R), indicating that it wants to read a value from an external device via the data bus
4. The control decoder logic maps this control input to activating the output enable (OE) line.
5. The combination of OE and CS activates the OE of LATCH B which causes the latch to pass the switch state on it's inputs (D_{IN}) to its outputs (Q_{OUT}). Thus the state of the switches now appears on the data bus lines.
6. Finally the CPU reads the current state of the data bus lines (thereby reading the switch states).

The next CPU instruction will change the address and ensure that the latch "freezes" its current values and does not output any values onto the data bus.

Self test questions

Q1. Which of the following are bus-aware devices: AND gate, op-amp, RAM chip, ADC, 555 timer, latch, joystick, serial UART, LCD display, beeper?

Q2. Why can a switch or button not be connected directly to the data bus? Explain.

Q3. How does the CPU identify which device on the bus to communicate with?

Q4. How does the CPU send information to an output peripheral such as a digital IO port?

Q5. Explain how the latch on slide 8 acts as a bus-aware connection to the 7-segment display.

I/O synchronization

13

13

I/O Synchronization

14

- Managing the time and manner in which input/output takes place is called **I/O sychronization**
- There are various synchronization methods that can be used but our initial focus will be on **polling I/O**
- Polling is appropriate for a wide range of I/O devices including digital I/O ports, analog I/O ports (via the ADC), the USART, etc.

14

Polling – an analogy

- You need a colleague to do something for you, so you request them to do it – this begins the polling operation
- Then there are a number of different possibilities for how the polling operation proceeds and completes…
  1. For immediately or very quickly completed tasks that cannot fail – no need to wait or check, so you can move on to other work immediately
  2. For slightly longer tasks with predictable duration, you might wait until the task is done – you cannot do any other work until it's done
  3. For longer tasks, you'll probably choose to check back regularly until the task is done – you can work ahead on other things between checks

15

Characteristics of Polling

- All transfers to and from I/O devices are performed by your programme at **times that you choose** (often at regular intervals)
- the software "polls" (checks) the device to initiate an I/O operation or to check if an I/O operation is possible/required
  - For simple I/O operations, polling may only be required to check the current "value" or "flag bits" in a device register
  - More complex I/O may consist of initiating some I/O and then polling until it has been completed
- Examples
  - Example 1: poll port B, bit 0, every 20ms to see if active low button is closed (=0) or open (=1)
  - Example 2: poll ADC sample conversion flag to see if next digital sample is ready. If it is, then read the sample value.

16

Polling approach

- There are two main questions to answer when designing a polling solution for some I/O:

1. When to begin an I/O operation?
2. How to complete the I/O operation?

Polling – when to initiate?

1. The possibilities for when to begin an I/O operation…

- Once off or intermittently (as needed)
  - The I/O operation is initiated only when needed
  - Often used for certain kinds of outputs (e.g. change the state of a LED in response to something else, transmit some data over serial, etc.)
- At regular periodic intervals
  - The I/O operation is initiated at regular intervals
  - This is typically used to monitor inputs such as button press, received serial communication, etc.
  - It is also often used to sample analogue inputs with a steady sample rate

Polling – how to detect completion?

2. How to detect completion of an I/O operation depends on the how quickly/reliably it can be completed

- **Poll once and forget**
  - No fail operations only. Don't wait or check for completion if not immediate – once initiated, we assume the operation will complete quickly enough. E.g. Digital port input/output.
  - Other work can continue immediately.
- **Busy wait polling (blocking)**
  - Short predictable operations generally. Wait for the operation to complete, checking (polling) constantly. E.g. waiting for ADC conversion complete once initiated.
  - No other work can be done while we wait
- **Multi-task polling (non-blocking)**
  - Longer or unpredictable duration operations. Check back (poll) at intervals (usually regular) to detect completion of the I/O operation
  - Other work can proceed between checks while waiting for the overall operation to complete

Example:

Poll once and forget


```
// General app structure
global gButtonState = OPEN

Main…

setup()
  setup button and LEDS…

loop()
    pollSwitch()
    updateLED()
    delay SUPERLOOP_TICK
```

```
pollSwitch()
  …

updateLED()
  if gButtonState is CLICKED
    // poll once and forget --
    // Note how the code doesn't wait
    // to see if the LED changed state
    // or not
    toggle(LED_PIN)
```

In this example, the updateLED function is writing to a digital I/O port pin – i.e. polling an output

Example:

Busy wait polling - ADC


```
// General app structure
global gButtonState = OPEN
global gAdcValue = 0

Main…

setup()
  setup button and ADC…

loop()
    pollSwitch()
    checkADC()
    delay SUPERLOOP_TICK
```

Note: If a programme has real time contraints, busy wait polling should only be used for short delays that cannot cause deadlines to be missed

```
pollSwitch()
  …

checkADC()
  if gButtonState is CLICKED
    // initiate ADC conversion
    set ADC_GO_DONE_BIT = GO

    // busy wait polling:
    // The code checks continuously
    // until conversion is done
    // and no other useful work can
    // be done while waiting
    while (ADC_GO_DONE_BIT is not DONE)
      doNothing

    // conversion is done and result
    // is in adc register, so copy it
    // to global variable
    set gAdcValue = adcRegister
```


Example:

Busy wait polling – transmit string


```
// General app structure
global gButtonState = OPEN

Main…

setup()
  setup button and serial…

loop()
    pollSwitch()
    transmitIfNeeded()
    // other tasks…
    delay SUPERLOOP_TICK
```

The duration that TXREG can be full is generally predictable based on the serial baud rate, but the duration might be relatively long (e.g. 1ms) so this might not be the best solution here.

```
pollSwitch()
  …

transmitIfNeeded()
  if gButtonState is CLICKED
    transmitStringCompletely("hello")

transmitStringCompletely(str)
  while not at end of str
    // busy wait polling:
    // The code checks continuously
    // until TXREG is empty
    // and no other useful work can
    // be done while waiting
    while TXREG is full
      doNothing

    // TXREG is empty, send next char
    set TXREG = next char from str
```

Example:

Multi task polling – transmit string


```
// General app structure
global gButtonState = OPEN
global gTxBuffer

Main…

setup()
  setup button and serial…

loop()
    pollSwitch()
    transmitIfNeeded()
    completeTransmission()
    // other tasks…
    delay SUPERLOOP_TICK
```

Multi-task polling means quickly checking if I/O is required or ready and returning immediately if it is not rather than waiting. This allows us to proceed with other tasks in the superloop.

```
pollSwitch()
  …

transmitIfNeeded()
  if gButtonState is CLICKED
    startTransmission("hello")

startTransmission(str)
  copy str to gTxBuffer

completeTransmission()
 // multi-task polling:
 // returns quickly whether char
 // can be transmitted or not and
 // does not wait
 if not at end of gTxBuffer
   if TXREG is full
      return

   // TXREG is empty, send next char
   set TXREG = next char from str
```


Polling features

- Advantages
  - Simple to implement
  - I/O is synchronous, taking place in the application foreground
  - Device priority is easy to change
    - Determined by the order in which software polls (checks) the devices
    - Device priority is used to decide what to handle first when several I/O devices need servicing at the same time
- Disadvantages
  - CPU time may be wasted continually polling
  - Response time/throughput is often a compromise
  - Not best choice when I/O takes place infrequently or at unpredictable intervals

Standard I/O Synchronization issues

- Some standard I/O synchronization issues and the possible solutions when using polling I/O…
  1. The CPU/app is faster than the I/O device
  2. The I/O device is (temporarily) faster than the CPU/app –
  3. The I/O device needs or supplies data at regular, predictable times
  4. The I/O device needs or supplies data at irregular, unpredictable times

contd

1. The CPU/app is faster than the I/O device
   - The app needs to buffer output data so that it can be written at the slower device rate over the course of multiple superloops
   - The app can do other work while not servicing the IO device
2. The I/O device is (temporarily) faster than the CPU/app
   - This only works if the fast IO device is only active in short bursts of activity
   - The app needs to buffer input data that is arriving quickly without doing any substantial processing work. The app must respond to device input with low latency (delay between stimulus and response) to avoid missing data
   - Later when the input burst has ended, the app can process the buffered data

contd

1. The I/O device needs or supplies data at regular, predictable times
   - Poll at regular intervals corresponding to the predictable IO times
2. The I/O device needs or supplies data at irregular, unpredictable times
   - Poll at regular intervals corresponding to the minimum latency required to handle the irregular IO times. E.g. a 10 ms minimum latency on unpredictable button presses requires polling at 10 ms (or smaller) intervals

Self test questions

Q1. With the aid of suitable examples, explain how and when each of the following approaches to polling should be used: *poll once and forget*, *busy-wait polling (blocking)*, and *multi-task polling (non-blocking)*.

Q2. For a real time system choose the appropriate polling start and completion strategies for the following (state your assumptions):

- Checking if a button is pressed
- Waiting for a currently pressed button to be released
- Lighting an LED
- Checking a window sensor in a security alarm
- Sending data via a Bluetooth sensor

Self test questions

Q. Write the pseudocode for a system which writes to a clock display (4 x 7 segment display) via a display driver. To write to the display: first start a write cycle, then wait for ACK signal from display (in 10 microsecs or less), then write each of the 4 clock numbers waiting for an ACK from the display after each number.

- *What polling strategies are you using and why?*