4.20 Advanced I/O synchronization

Interrupt driven I/O

Other I/O synchronization methods

Overview

- Aims
  - To introduce alternatives to polling I/O methods, particularly interrupt driven I/O
- Learning outcomes – you should be able to…
  - Explain the handling of an interrupt using polled interrupts or vectored interrupts
  - Explain/show how to implement interrupt priority (when multiple interrupts occur simultaneously) using polled interrupt software, interrupt daisy chaining, and a programmable interrupt controller
  - Convert between single interrupt service routine (ISR) implementing polled interrupt priority vs. multiple ISRs (and associated hardware) implementing vectored interrupt priority

Recall - I/O Synchronization

- Why refer to it as I/O synchronisation?
- Some specific cases…
  1. The CPU is faster than the I/O device
  2. The I/O device is faster than the CPU
  3. The I/O device needs to transfer data at regular, predictable times
  4. The I/O device needs to transfer data at irregular, unpredictable times

Interrupt driven I/O

I/O devices “interrupt” CPU via an “interrupt request” (IRQ) line/pin as follows…

1. When an I/O device wants service it causes a transition on the CPU’s IRQ pin (or internal IRQ line when it is an internal peripheral)
2. CPU immediately(*) pauses main program execution
3. CPU executes Interrupt Service Routine (ISR) and ISR code services the device (e.g. by reading or writing data)
4. When ISR completes, CPU resumes program execution

NOTE: see 4.10 Timer notes (where interrupts were introduced) for more detail

Example – interrupt driven timer

```
main:…

setup():
  set TMR0 = TMR0_START_VALUE

loop():
    while gTimerExpired is FALSE, do nothing // wait
    set gTimerExpired = FALSE

    doWork…
    // NOTE: no delay or sleep required

doWork():
  …

isr(): // Interrupt Service Routine implementing polled interrupts
  if TMR0IF interrupt
    set TMR0IF = 0 // clear interrupt flag since we’ve responded
    set TMR0 = TMR0_START_VALUE // reset timer
    set gTimerExpired = TRUE // signal superloop
```

Example – interrupt driven UART

```
// NOTE: this pseudocode reads commands from the UART but does
// nothing useful with them.
main:…

setup():
  configure initial device/peripheral settings
  unmask Rx interrupt and enable interrupts

loop():
    processSerialCommand()
    …
    delay or sleep

processSerialCommand():
  if gCommandComplete:
    process command…

isr(): // Interrupt Service Routine implementing polled interrupts
  if (UART Rx interrupt)
    buffer[i] = value of UART Rx register
    if buffer[i] is END_OF_COMMAND // e.g. a newline
      set gCommandComplete = TRUE
    increment i (if buffer still has space)
```

Interrupt synchronisation features

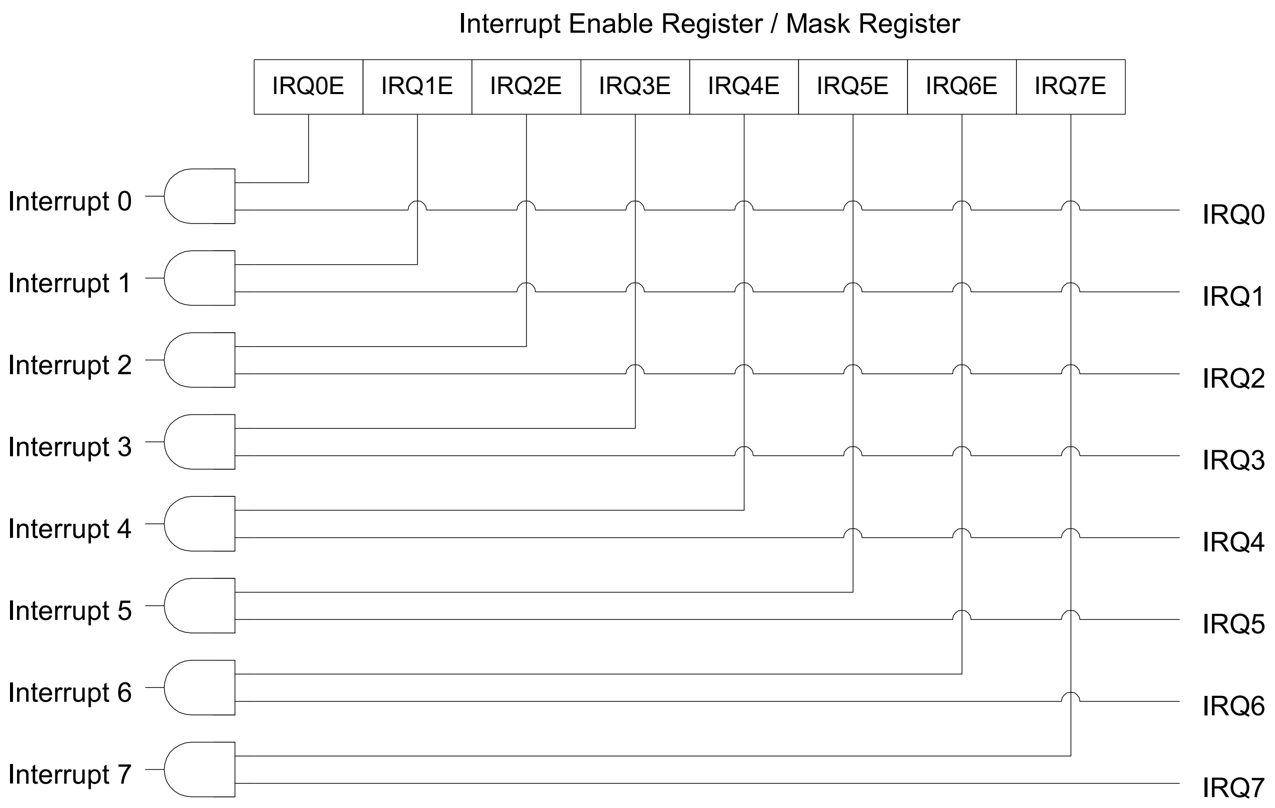
- Best for
  - I/O which occurs at infrequent or unpredictable intervals
  - Asynchronous I/O which takes place “in the background”
- Advantages
  - Software doesn’t “waste” time polling I/O devices
- Disadvantages
  - Software can be complex

Some interrupt issues

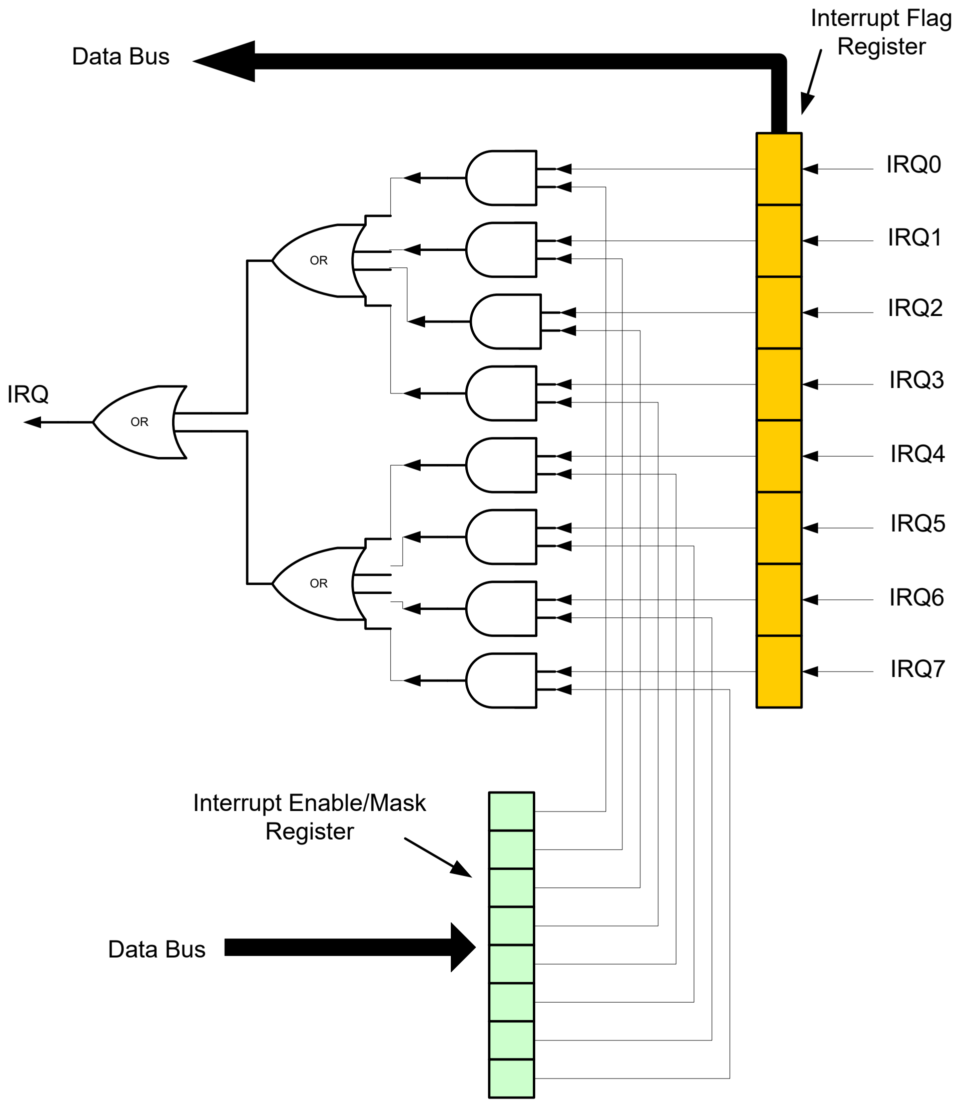
- If there are many I/O devices...
  - Which one caused the interrupt (and which ISR should be executed)?
  - What happens if 2 or more devices try to interrupt at once?
- Solutions...
  - Polled interrupts use software based priority
  - Vectored interrupts with Daisy chained priority or Interrupt controller priority
- Additional considerations
  - Masking and disabling interrupts
  - Multiplexing a single interrupt line

Masking and disabling interrupts

- With multiple interrupt systems there has to be a way to enable and disable all or indivdual interrupts.



Multiplexing a single interrupt line



This figure shows how 8 I/O devices may be connected to a single IRQ pin.

Each of the 8 interrupts is maskable (mask register).

Software priority (polled interrupts) is made possible with the inclusion of an Interrupt Flag Register.

Polled Interrupts

- Used when there are multiple devices connected to a single IRQ pin and no additional information transmitted during interrupt
- There is just one ISR
- ISR code polls (checks) some register(s) to determine which interrupt occurred
- Interrupt priority is determined by order in which the code polls (checks) the set of possible interrupts
- Sometimes called Software Polled Interrupts
- Polled interrupts is the method used in PIC microcontroller

```
isr()
  if SSP interrupt flag set
    handle SSP Interrupt
  else if TMR0 interrupt flag set
    handle Timer0 Interrupt
  else if ADC interrupt flag set
    handle ADC Interrupt

// Note: often not a good idea
// to call functions from isr
// - instead embed handler code
// directly in isr
```

Vectored Interrupts

- Using vectored interrupts, there are multiple ISRs, one per interrupt source
- Each I/O Device is configured with a unique number called a vector, which can be either
  - the address (or part thereof) of its particular ISR
  - an index into a table (array) of ISR addresses, called the interrupt vector table or interrupt dispatch table
- Hardware signalling is used to get the vector from the device which generated interrupt
- Compared to (software) polled interrupts
  - Faster to identify interrupt source
  - Requires hardware to implement interrupt priority

Vectored interrupt ISR code

```
// Separate ISRs for each possible interrupt.
// No checking of flags required to identify
// which interrupt occurred – the vector which
// causes particular ISR to be invoked is sufficient.

ssp_isr()
  handle SSP Interrupt

tmr0_isr
  handle Timer0 Interrupt

adc_isr
  handle ADC Interrupt
```

Compare with polled interrupts example

Basic vectored interrupt hardware (each device)

① device needs service, asserts IRQ line

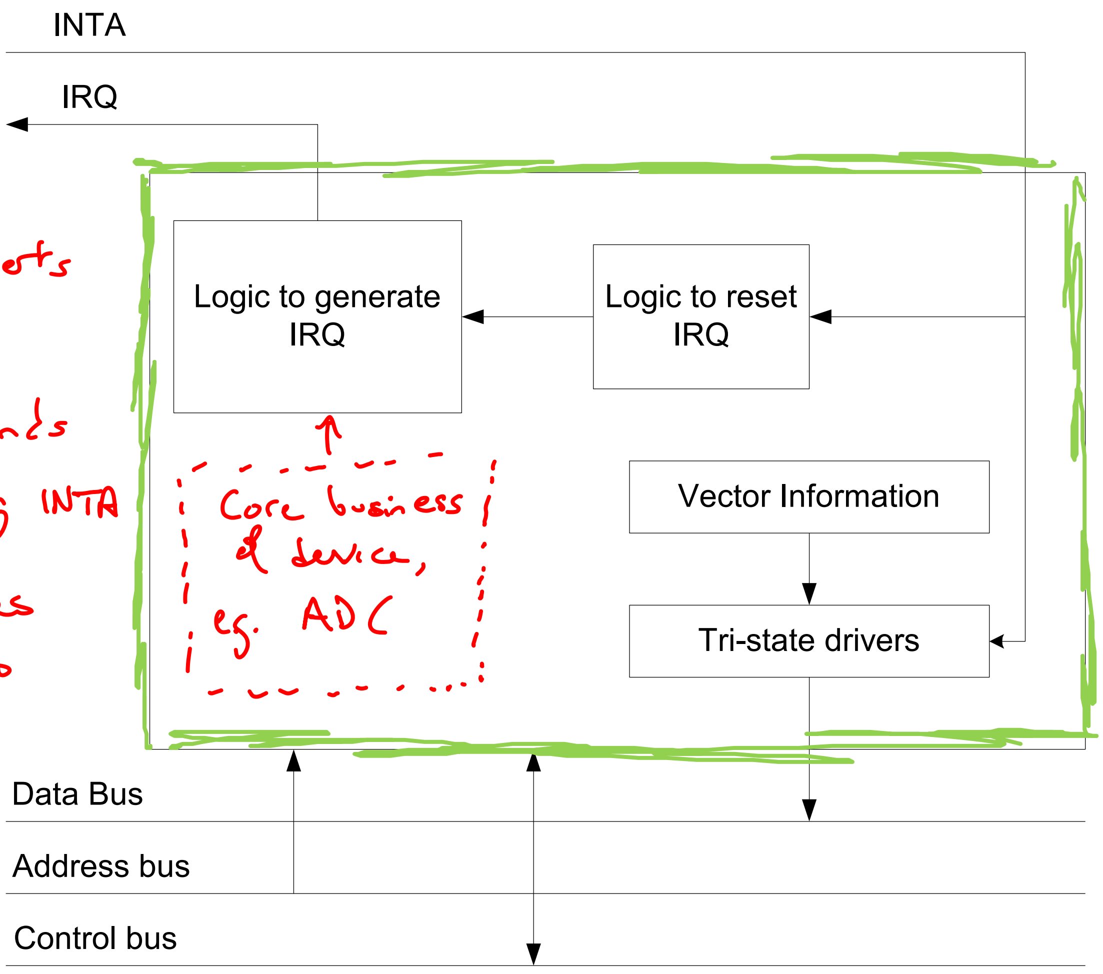
② CPU responds by asserting INTA

③ Device places vector onto databus

⑤ CPU reads vector from databus

etc...

④ Device resets the IRQ line

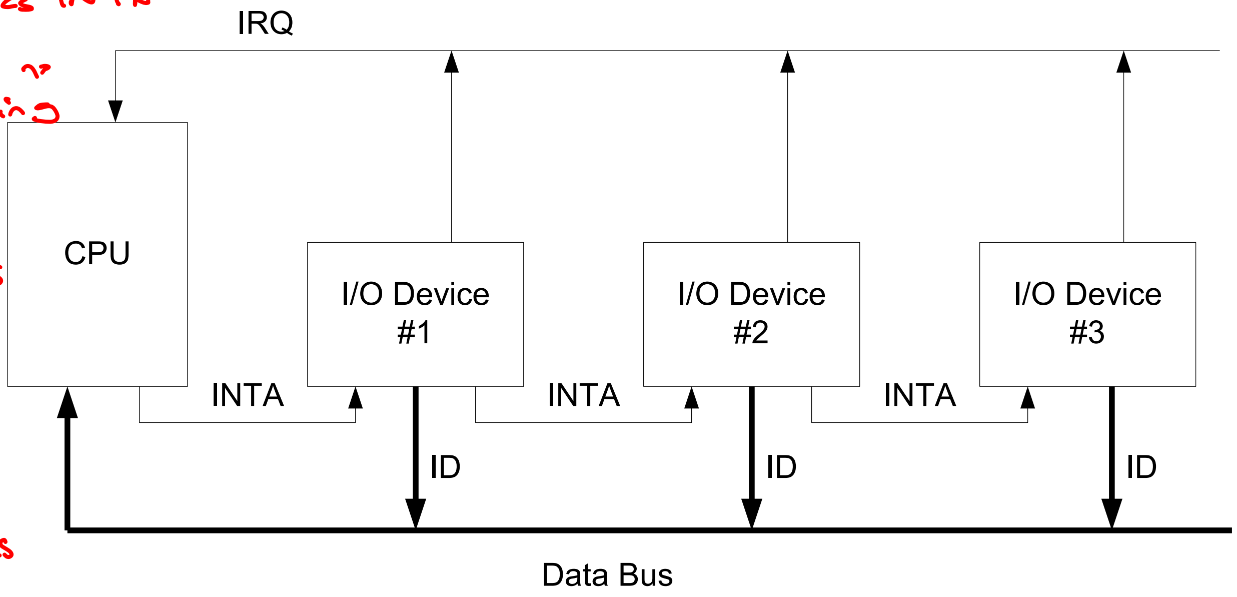


Vectored interrupting device hardware

Vectored Interrupt Priority – Daisy Chain

Suppose device 2 and device 3 generate simultaneous interrupts

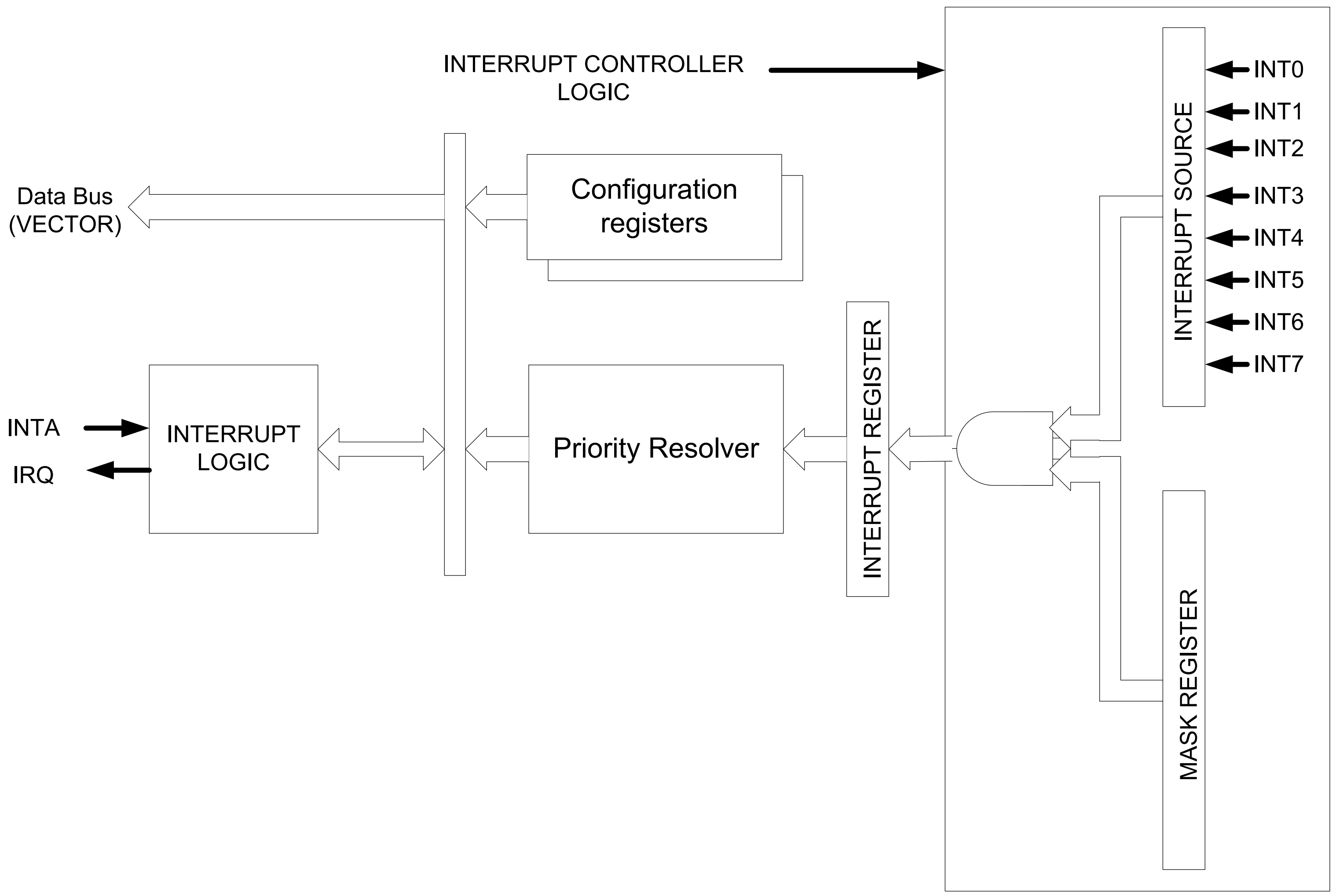
1. IRQ line asserted by device 2,3
2. CPU responds INTA
3. Device 1 has no interrupt pending so forwards INTA
4. Device 2 has interrupt pending so responds with ID (vector)
5. CPU handles interrupt
6. IRQ still asserted (because of device 3) so do it again ....



Daisy Chained Interrupts for Prioritization

Fast but inflexible

Vectored Interrupt Priority – Programmable Interrupt Controller



Synchronization methods

There are four synchronization methods used to manage I/O data transfers

1. Polling
2. Handshaking
3. Interrupts
4. Direct Memory Access – DMA

Note 1: Different methods may be used for different I/O devices (in a single embedded system)

Note 2: Combinations of methods may be used with a single I/O device

1. Polling features (looked at previously)

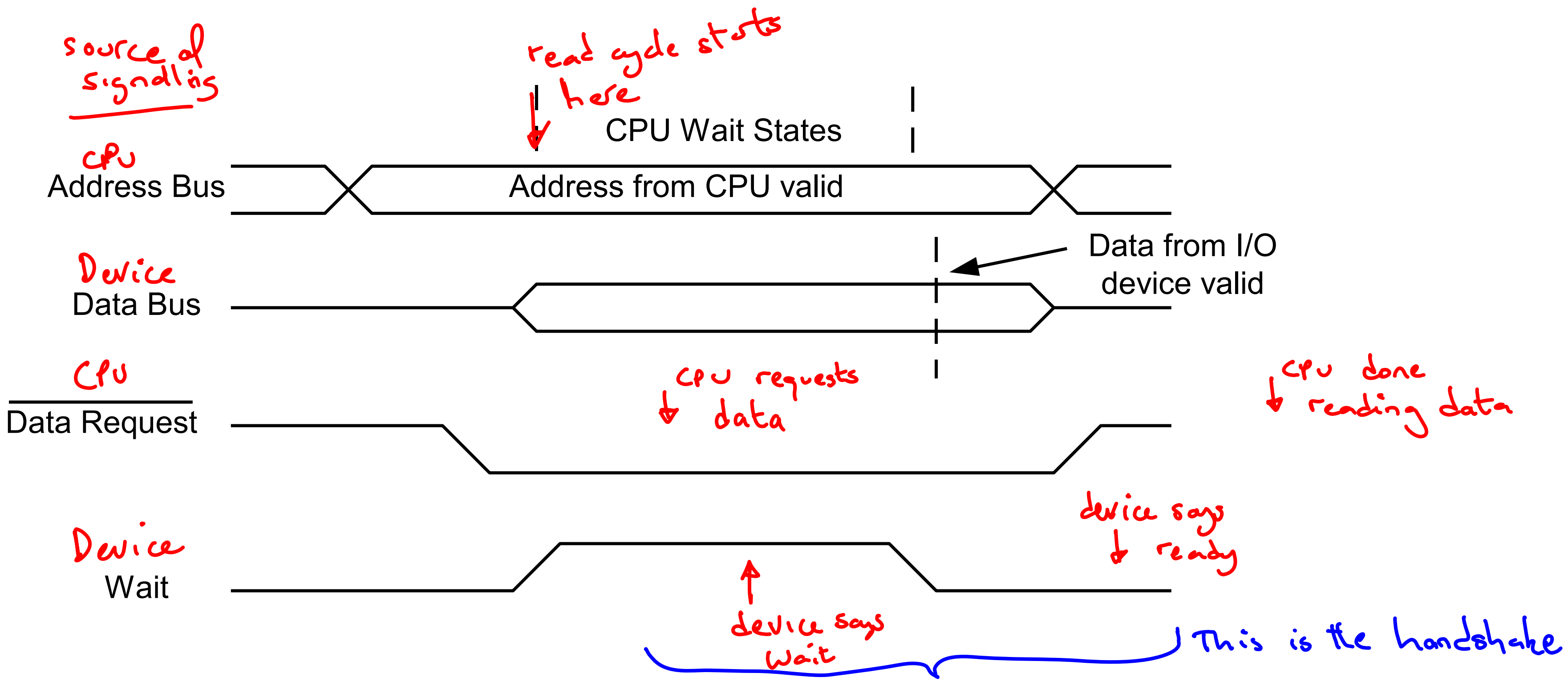
- Advantages
  - Simple to implement
  - I/O is synchronous, taking place in the foreground
  - Device priority is easy to change
    - Determined by the order in which software polls (checks) the devices
- Disadvantages
  - CPU time may be wasted continually polling
  - Response time/throughput is often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

2. Handshaking I/O

- A hardware method of synchronizing with a slow(ish) I/O device
  - Not strictly an alternative to polling and invisible to software
  - Can use with other synchronisation methods
  - I/O device “handshakes” with CPU to say when ready
  - Often used to interface to “slow” memory

- Advantages
  - Simpler for programmer (than software polling)
  - Potentially faster (than software polling)
- Disadvantages
  - Requires additional pins/lines from the CPU, e.g. READY/WAIT
  - When CPU is in the wait state it can not do anything else

Handshaking example



Hardware handshaking - Read cycle with wait states

3. Interrupt driven I/O

- Best for
  - I/O which occurs at infrequent or unpredictable intervals
  - Asynchronous I/O which takes place “in the background”

- Advantages
  - Software doesn’t “waste” time polling I/O devices when no service is required
- Disadvantages
  - Software can be complex

4. Direct Memory Access - DMA

- A DMA Controller (DMAC) is a single purpose processor designed to perform high speed data transfers between memory and I/O devices.
- A DMAC does not require the CPU to accomplish the data transfer
  - The DMAC takes control of the Address and Data buses to effect the data transfer
    - May suspend or halt the CPU (temporarily)
    - May steal some memory cycles (Cycle Stealing)
    - Often I/O is slow-ish so DMAC doesn’t interfere much
  - Used for data intensive I/O such as hard disk or memory card transfers (e.g. in a digicam)

Exam: 2009-2010 Q3 (c)

- Briefly describe the terms *interrupt vector* and *interrupt priority*.

Consider a system with 3 peripheral devices (A, B, and C) which may generate interrupts. The priority order from highest to lowest is A, B, C. The system uses the daisy chain implementation of vectored interrupts.

- With the aid of a diagram, describe in detail how the scheme would operate if device A and device C both required service simultaneously.

Exam: 2011-2012 Q4 (a)

- List the key differences between software polling, polled interrupts and vectored interrupts.
- For a system using polled interrupts, show (using pseudocode) how you would prioritize interrupts from the following devices in the order given: Timer, ADC, and UART.

```
ISr ( ):
    if (Timer intr)
        handleTimerIntr ( )
    else if ( ADC intr )
        handleAdcIntr()
    else if (UART intr)
        handleUartIntr()
```

Exam: 2011-2012 Q4 (b) - extract

Consider a polled interrupt routine that must use the timer interrupt (TMR0IF) to determine when to start an ADC conversion and use the end of ADC conversion interrupt (ADIF) to copy the ADC sample into set the variable **gSample** and then set the variable **gSampleToProcess** true.

- Write the pseudocode implementation of the polled interrupt routine. Assume that the Timer0 register is TMR0, that the period between timeouts is ADC_SAMPLE_TIMEOUT, that ADC conversion is started by setting the ADGO bit, and that we are only interested in 8-bit ADC samples from register ADRESH.

```
isr ( ) :
    if (TMR0IF is set)
        set ADGO = 1
        set TMR0 = 256 - ADC_SAMPLE_TIMEOUT
        clear TMR0IF
    else if (ADIF is set)
        set gSample = ADRESH
        set gSampleToProcess = TRUE
        clear ADIF
```