EEL 4744C: Microprocessor Applications

Lecture 6

Part 1

Computer Buses & Parallel I/O

Reading Assignment

- Software and Hardware Engineering (new version): page 17-20

OR

- Microcontrollers and Microcomputers: Chapter 7

Design Statement

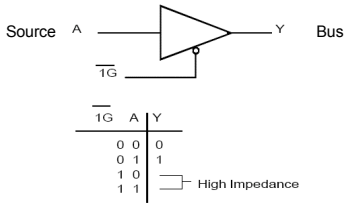
- We need to transfer information, in parallel or in serial, in or out of the CPU, i.e. I/O
- I/O requires a hardware interface between the I/O devices and the computer bus
- Design the hardware interface to transfer code and data from multiple sources to the CPU, and from the CPU to multiple destinations, using a computer's buses

Computer Bus

- A bus is a parallel, bidirectional, and binary information pathway with multiple sources and multiple destinations
- CPU is interconnected to memory and I/O devices through 3 kinds of buses: data, address, control
- Component-level bus: defined by the signals on the microprocessor chip, e.g. READ/WRITE
- System-level bus: defined by the signals on the backplane (system board), e.g. MEMRD, IORD

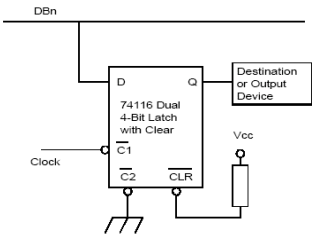
The Input Interface

- A parallel 8-bit input interface can be constructed with 8 tri-state buffers whose enable (1G) lines are connected together, e.g. 74LS244 octal line driver
- Signal 1G must be asserted (= 0) to activate output



The Output Interface

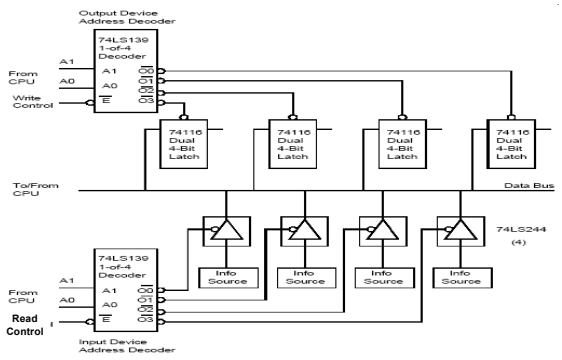
- A latch is the interface between the data bus and the output device. Control signals for this latch are generated from the sequence controller, e.g. clock



Multiple Sources & Destinations

- The interface must let CPU select from one of many sources and destinations of I/O. We can use decoders for selecting these
- READ_CONTROL is used to select the input source to read from
- WRITE_CONTROL is used to select the output destination to write to
- A0, A1, READ_CONTROL and WRITE_CONTROL are generated by the CPU (sequence controller)

Address Decoding for Sources and Destinations



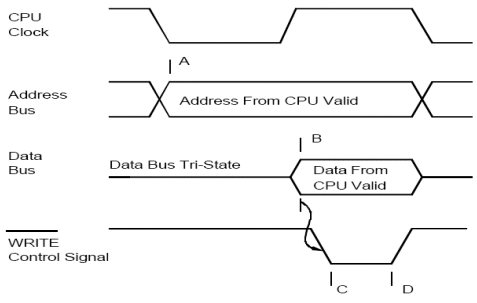
Multiple Sources & Destinations

- The CPU must provide timing and synchronization so that the transfer of information occurs at the right time
- This means that data can only be taken from or placed onto the bus at the correct time
- Write cycle: transfer of data from a register to an output data latch
- Read cycle: transfer of data from an external source to the CPU (a register)

Write Cycle

- CPU places the address on the address bus at point A. CPU timing is controlled by the clock
- Data bits are placed by CPU onto the data bus at point B
- The WRITE signal is asserted by CPU shortly after at point C
- The WRITE signal stays asserted long enough until point D, to let the data bits be latched

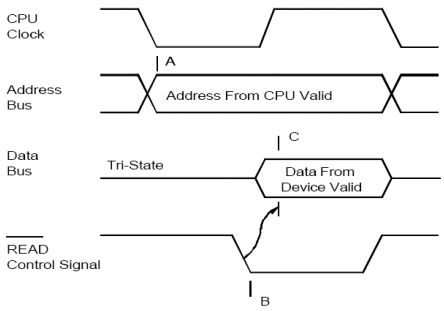
Write Cycle



Read Cycle

- CPU places the address on the address bus at point A. CPU timing is controlled by the clock
- The READ signal is asserted at point B to let the input device know that CPU is ready for data
- CPU begins taking data bits from the data bus at point C
- If the input device is not ready at point C, then need to have I/O synchronization

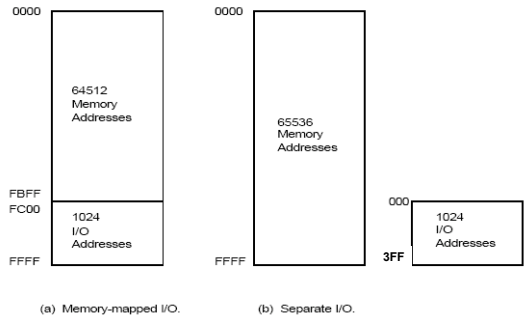
Read Cycle



I/O Addressing

- Address bus is used by both the memory and I/O devices (through the I/O interface)
- How does the hardware differentiate between memory reads/writes and I/O reads/writes?!
- 1st way: memory-mapped I/O, any instruction that reads/writes memory ALSO reads/writes I/O
- 2nd way: separate I/O, use separate I/O instructions for I/O reads/writes

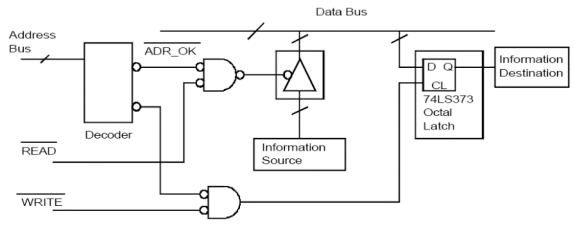
I/O Addressing



Memory Mapped I/O

- The entire address space is divided into memory address space and I/O address space
- Popular in early minicomputers and today's many microcontrollers
- Pros: simpler CPU design; allows any memory reference instruction to access any I/O device
- Cons: reduced amount of application memory; requires full address bus be decoded to avoid confusion between memory and I/O addresses

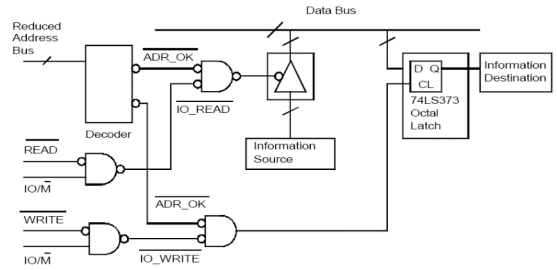
I/O Interface for Memory Mapped I/O



Separate I/O

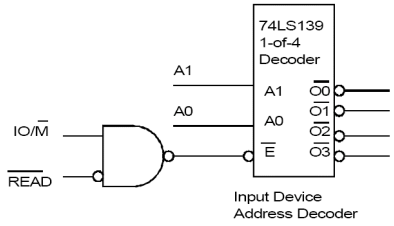
- **Separate memory map and I/O map. Since far fewer I/O devices are needed than memory, I/O map is much smaller than memory map⇒ fewer I/O address bits and cheaper address decoders!**
- **Need additional hardware and a new signal IO/M to prevent simultaneous data bus access by memory and I/O devices. IO/M is usually HI for I/O access and LO for memory ⇒ impact sequence controller!**
- **Need I/O instructions different from the memory reference instructions ⇒ impact instruction decoder!**

I/O Interface for Separate I/O



Separate I/O

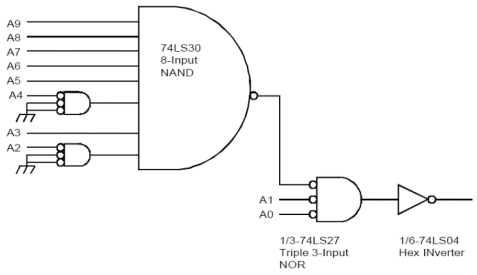
- **How do we use READ and IO/M signals to enable the output of a 74LS139 dual 1-of-4 decoder?**



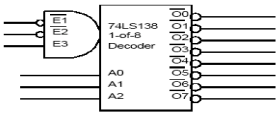
Address Decoding

- Full address decoding: requires all address bits to be decoded, for systems with many I/O devices
- Incomplete address decoding: used when a system does not need all the I/O address space
  - Reduced address decoding: only decode higher-order address bits, lower-order bits are treated as don't-cares
  - Linear select decoding: for small systems with few I/O devices, each address bit can select an I/O device!
- Need I/O instructions different from the memory reference instructions ⇒ impact instruction decoder!

Decoding using Discrete Logic Circuit



74LS138 Decoder

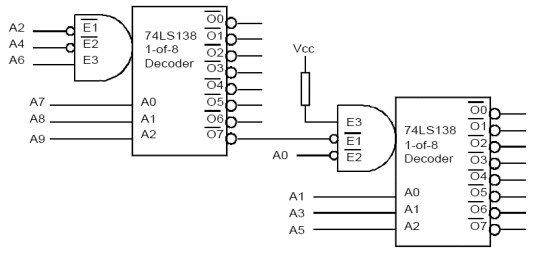


INPUTS						OUTPUTS							
$\overline{E1}$	$\overline{E2}$	E3	A2	A1	A0	$\overline{O0}$	$\overline{O1}$	$\overline{O2}$	$\overline{O3}$	$\overline{O4}$	$\overline{O5}$	$\overline{O6}$	$\overline{O7}$
H	X	X	X	X	X	H	H	H	H	H	H	H	H
X	H	X	X	X	X	H	H	H	H	H	H	H	H
X	X	L	X	X	X	H	H	H	H	H	H	H	H
L	L	H	L	L	L	L	H	H	H	H	H	H	H
L	L	H	L	L	H	H	L	H	H	H	H	H	H
L	L	H	L	H	L	H	H	L	H	H	H	H	H
L	L	H	L	H	H	H	H	H	L	H	H	H	H
L	L	H	H	L	L	H	H	H	H	L	H	H	H
L	L	H	H	L	H	H	H	H	H	H	L	H	H
L	L	H	H	H	L	H	H	H	H	H	H	L	H
L	L	H	H	H	H	H	H	H	H	H	H	H	L

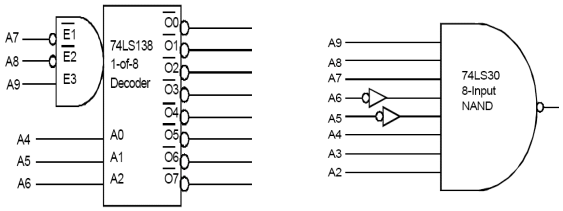
Decoding using Decoder

A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	Add.	Output
1	1	1	1	0	0	0	0	0	0	3C0	O0
				0		0		1		3C2	O1
				0		1		0		3C8	O2
				0		1		1		3CA	O3
				1		0		0		3E0	O4
				1		0		1		3E2	O5
				1		1		0		3E8	O6
				1		1		1		3EA	O7

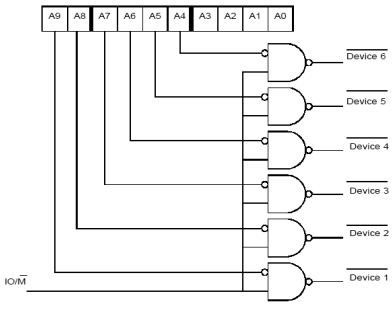
Decoding using Decoder



Incomplete Address Decoding



Linear Select Addressing



I/O Synchronization

- **We need to synchronize I/O and the CPU when:**
  - **CPU is faster than the I/O device. Example: the read cycle discussed previously**
  - **I/O device needs to transfer data at unpredictable intervals. Can be solved with interrupts**
  - **I/O device is faster than CPU. This problem is often solved with direct memory access (DMA), bypassing the CPU!**

Software I/O Synchronization

- First method is called *real-time synchronization*
- Uses software delay, e.g. loop, to make CPU wait for the I/O device
- Example: write a simple delay loop to wait for the output to be done
- Cons: the delay loop, i.e. # of iterations, has to be changed if the CPU clock speed changes; also CPU cannot do anything while waiting

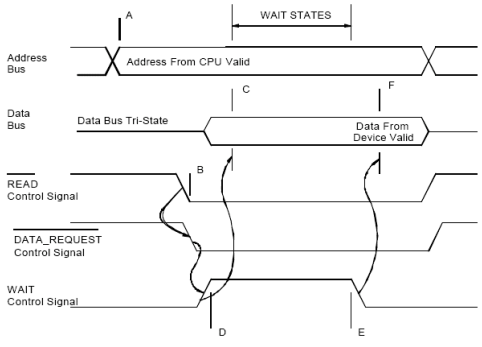
Software I/O Synchronization (2)

- Second method is called *polled I/O*
- Uses a status register with a DATA_READY bit with the input device
- Software will check the DATA_READY bit. If the device is not ready, the software keeps looping
- Otherwise, start reading the information
- Can set up the same scheme for output devices
- Allows CPU to do other things while waiting

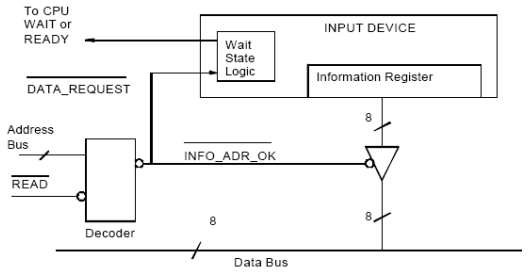
Handshaking I/O

- A hardware method using READY and WAIT
- When WAIT signal is asserted by an external device, the sequence controller "spins its wheels" waiting, until WAIT is de-asserted, and when READY signal is asserted
- CPU then proceeds with the I/O operation
- The finite state machine will have the WAIT state added to support the hardware design

Read Cycle with WAIT State



Input Handshaking I/O HW

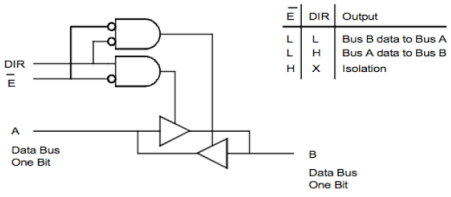


Multiplexed Bus

- Reason: not enough pins on a CPU chip to provide all the desired signals (all at the same time)
- Idea: time multiplexing. Use of a pin may change as a function of time
- Example: multiplexed address bus. For a 16-bit address, CPU supplies 8 bits at a time, thus saving 8 pins (from the chip) for other signals
- Signal ADDRESS_STROBE enables latching of the higher 8 address bits

Bidirectional Bus Transceiver

- Lets data flow into and out of CPU. Signal E enables the tri-state buffers. Signal DIR controls direction of data flow



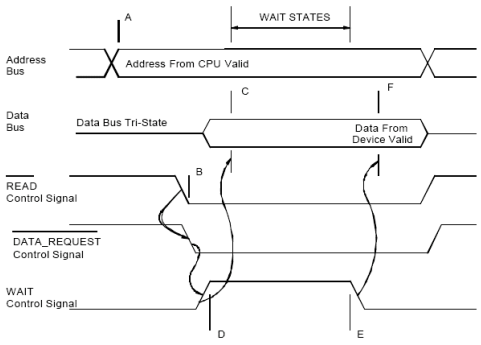
$\overline{E}$	DIR	Output
L	L	Bus B data to Bus A
L	H	Bus A data to Bus B
H	X	Isolation

One section of the 74LS245 octal bus transceiver

More Buses

- Synchronous bus: transfer of data on and off the bus is done in 1 CPU clock cycle, and ALL involved devices must respond within this time frame
- Problem with synchronous bus: clock frequency has to be based on the SLOWEST device in the system!
- Semi-synchronous bus: use 2 more signals, WAIT and DATA_REQUEST, to control the interface between CPU and the device
- Asynchronous bus requires all devices to respond with a WAIT signal, or signaling the absence of it

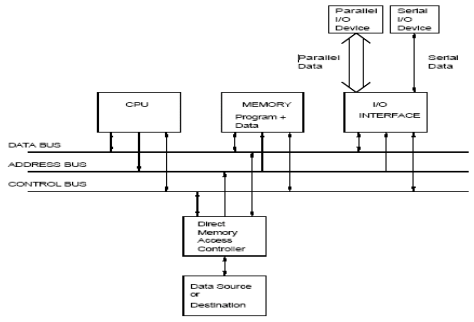
Asynchronous Bus Timing



Bus Masters and Slaves

- Normally CPU is only bus master, all the other devices, including memory, are bus slaves because they act based on control signals given by CPU
- But in the case of direct memory access (DMA), the DMA controller generates addresses and control signals to transfer data from I/O devices directly to memory, bypassing the CPU
- In fact, the DMA controller can suspend the CPU when data transfer is in progress. So the DMA controller is the second bus master!

DMA



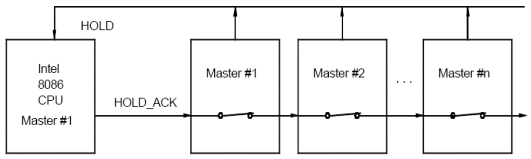
Bus Arbitration

- But what happens when multiple bus masters request the bus at the same time?
- The "Motorola model": a master asserts the signal BUS_REQUEST, the CPU responds by asserting BUS_GRANT, then the requesting master asserts BUS_GRANT_ACKNOWLEDGE
- But still, what should we do if >2 masters assert the BUS_REQUEST signal?
- Need a bus arbitration scheme

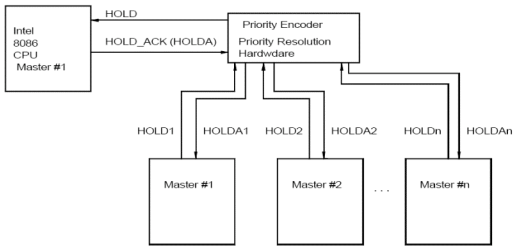
Bus Arbitration

- **Daisy chain arbitration:**
  - **A requesting master asserts the HOLD signal and opens its switch in the HOLD_ACK signal line**
  - **When CPU asserts the HOLD_ACK signal, it passes through all the non-requesting masters until reaching the requesting one. The closer a master is to CPU, the sooner it will have control of the bus. The farther masters must wait for the closer ones to get done and closes their HOLD_ACK switches to receive the HOLD_ACK signal**

Daisy Chain Bus Arbitration

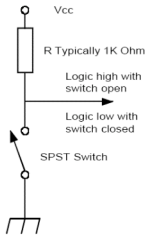


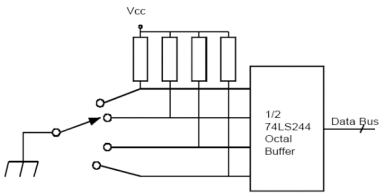
Hardware Priority Bus Arbitration



Priority encoder resolves the conflict: highest priority device gets it!

I/O Devices: Input Switches





SPST switch **Multiple-pole rotary switch**

Switch Bounces

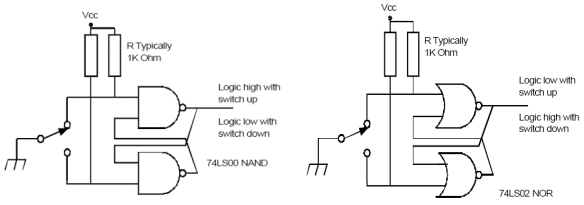
- Transient behavior of a switch can cause erroneous counting in software. Thus we need to "debounce" the switch. Switches usually bounce for 5~10 ms!
- Software debouncing:
  - Method 1: "wait and see". Switch bouncing usually lasts for about 5~10 ms. So, if the software detects a logic low (the switch closed), the software can wait for >10 ms
  - Method 2: "integrating debouncer". Initialize a counter with a value of 10. After the first detection of a logic low, poll the switch every 1 ms. Decrement the counter if a low is polled, increment the counter if a high is polled. When the counter reaches 0 ⇒ switch has been closed for at least 10 ms. If the counter reaches 20 ⇒ switch has been open for at least 10 ms

Software Debouncer

- Method 2 (is the switch closed or open?)

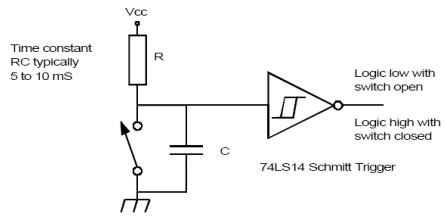
```
IF switch changed state INITIALIZE Count = 10
WHILE Count > 0 and < 20
    Delay 1 ms
    Poll the switch
    IF switch is closed decrement Count
    ELSE increment Count
ENDWHILE
IF Count = 0
    Switch is closed
ELSE
    Switch is open
```


Hardware Debouncers



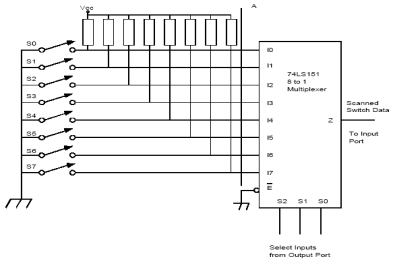
NAND latch debouncer NOR latch debouncer

Hardware Debouncers



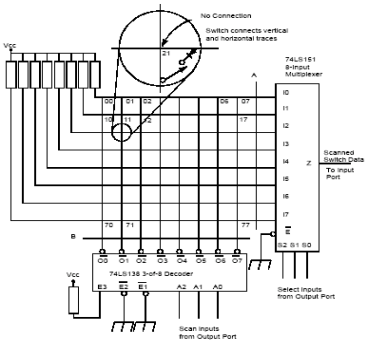
Schmitt trigger requires an input voltage threshold before switching. RC = 5~10 ms. C requires switch to be closed/open for sufficient time before logic state changes, otherwise the Schmitt gate is in hysteresis.

Linear Array of Switches



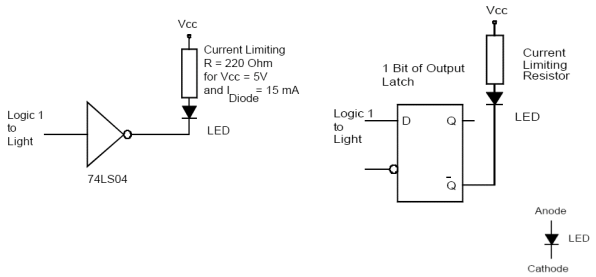
(S2,S1,S0) sequences from 000 to 111, selected switch input is scanned by software one bit at a time, taking into account switch debouncing.

2-D Array of Switches



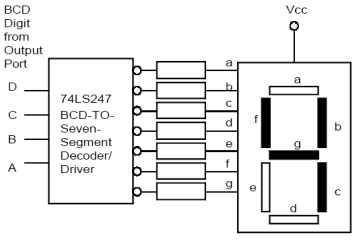
64 switches in total. (A2,A1,A0) selects a switch's A position. (S2,S1,S0) selects its B position.

LED Displays



Single LED driver circuits. LED lights up when a current of 10~20 mA passes through it forward

LED Displays



A common anode LED display

Programmable I/O Devices

- Programmable devices with internal registers that can be initialized by software
- Example: Motorola 6821 Peripheral Interface Adaptor. See M&M Fig. 7-30 and Tbl. 7-5~7-7
- For direct memory access (DMA), see Intel i8257 or Motorola MC68450

Buffered I/O

- Refers to the temporary storage of data between the I/O device and the CPU -- *data buffering*
- Also refers to the conversion between different electrical characteristics of the CPU, data buses, and I/O devices -- *electronic buffering*
- Data buffering is also one way of I/O synchronization. It uses latches to keep data from I/O devices. Handshaking I/O hardware is usually included
- Electronic buffering deals with *V* and *I* translations among different logic families e.g. CMOS and TTL