Direct Memory Access (DMA)

Slater Section 5.7
Heath pp. 134-145

DMA refers to the ability of an I/O device to transfer data directly to and from memory without going through the CPU. Contrast DMA with *programmed I/O* where the CPU explicitly copies data using loads & stores.

(diagram)

Why use DMA?

- Higher transfer bandwidth (one bus op instead of two)
- CPU can do other things during transfer
- faster, more consistent response times (no interrupt or polling overhead)

Example: simple disk transfer. Need to seek to correct cylinder, wait for sector to come under head, then must immediately transfer data at disk rate else lose bytes.

Example 2: network transfer. IP packet can be up to 4K bytes. Once hardware matches address, must immediately pull data in at network rate or lose bytes.

Why not use DMA?

- Hardware more complicated: I/O device must know how to become bus master, issue transactions, etc. (all while still acting as a slave too)
- Software more complicated (will be evident soon)
- DMA transfer takes bus away from CPU (can limit ability of CPU to do other stuff)
- Overhead to set up transfer: need to transfer many bytes to overcome

Basic DMA Example

DMA-capable I/O device will have control registers for:

- starting address of memory buffer
- number of bytes to transfer
- transfer direction (to/from device, read/write on bus)
- device-specific control (e.g., head, cylinder, sector for disk access)

DMA transfer algorithm:

1. Program makes I/O request to device
2. CPU does *initiation routine* (usu. part of device driver)
   - use programmed I/O (stores) to set up control regs
     - device-specific parameters
     - DMA parameters (buffer address/length)
     - last write: enable (start) bit
3. I/O device interface does transfer
   1. arbitrate for bus mastership
   2. put address on bus, assert control signals to do read/write (looks just like CPU to memory)
      - likely use burst transfer if available
      - size/type may be programmable via control reg
   3. I/O device supplies/consumes data
   4. increment address, decrement byte count
   5. if byte count > 0, repeat
      - usu. release bus to give other devices a chance
   6. if byte count == 0, set completion bit in status reg, generate interrupt
4. CPU ISR runs *completion routine* (also part of driver)
   - check for errors, retry if necessary
   - notify program that transfer is done (e.g., set flag variable in memory)
   - set up next transfer if appropriate (call initiation routine)
5. Program notices that request is complete

- main program may do other things during steps 2, 3, 4
  - on multitasking OS, another program may be run
- on DMA write to I/O device, program must be careful not to re-use buffer until it's notified that the write is complete

DMA Controllers

In modern systems, devices that need DMA typically have their own DMA control engine built in. In older/low-end systems (like the IBM PC), the logic required was/is a significant burden, so often a stand-alone, sharable "DMA controller" device is provided that lets several simpler I/O devices do DMA transfers.

The MPC823 has a couple of DMA controllers that are shared among the on-chip peripherals and can also be used by simple off-chip peripherals.

Basic idea (single I/O device):

- DMAC has address, direction, count registers
- A pair of request/acknowledge signals go directly to & from actual device (diagram)
- Initiation routine initializes both DMAC and device
- For each data transfer on bus:
  - Device uses DMA req line to tell DMAC it's ready for transfer
  - DMAC arbitrates for bus, puts out address, uses ack line to tell device when to put data on/take data from bus
  - note that device is doing a read or write under control of the DMAC without looking at bus address (which actually corresponds to memory location)
- DMAC interrupts CPU when done

Multiple devices:

- DMAC has a pair of req/ack lines for each device (say 4)
- Also has a set of control regs (address, dir, count) for each device
- Software can treat like 4 separate DMACs, initialize devices & DMAC reg sets independently
- DMAC looks at all req lines, on mult requests chooses one device to do transfer (prioritized or round-robin)

Each of these sets of registers, req/ack lines is called a DMA "channel"

IBM PC had 4, used 0 for DRAM refresh (timer, highest pri.) PC/AT extended to 7.

This is called a "one-cycle" or "fly-by" DMA transfer (same as device w/built-in). Stand-alone DMACs are usually capable of two-cycle transfers as well, where initialization routine specifies two addresses, etc. (for devices too dumb to deal with DMA req/ack lines). Note that this is not much faster than programmed I/O (but still may have a faster response time and frees the CPU). This feature can also be used for mem-mem copies (but rarely is).

Advanced DMA

For fast devices, we may be limited by the time it takes the CPU to run initiation & completion routines. Need a way to let device do multiple requests back-to-back w/o waiting on CPU.

Simple answer: “buffer chaining”: have multiple sets of regs per channel (equiv to mult channels per device), DMA moves on to next one immediately after prev. completes

Answer: device is now smart enough to read/write memory, let it get parameters from there. Build array in memory of request parameters: address, length, direction, device parameters, etc.

- device control regs now hold starting address, length of parameter array (aka descriptor array)
- status flag indicates valid/done
- managed as circular buffer
- may interrupt after each request, or only when out of requests/request buffer full etc.