CMPT 250 : Weeks 12-13 (Nov 21 to 30)

1. HIERARCHICAL MEMORY

In choosing between SRAM and DRAM for implementing memories, the trade-offs are expensive vs cheap, and fast vs slow. To approach the performance of SRAM at the cost of DRAM, it is possible to combine the two types of components and in many circumstances achieve a performance approaching SRAM at the cost of DRAM.

That is, improve the efficiency of the memory cycle using faster devices to provide only a small part of the physical memory.

If only a part of physical memory is implemented with fast components, then the average memory access time can be used to compare the performance of different hierarchical memory architectures:

$$t_{avg} = p \times t_{fast\ mem} + (1-p) \times t_{slow\ mem}$$

where p is the probability that the address specified maps to a physical address in the fast memory. Obviously the closer that p is to 1, the closer the access time is to that achieved if only fast memory components were used. On the other hand, for economic reasons, as well as physical constraints, we would like to make the ratio of fast memory to slow memory as small as possible.

Locality of reference is the property possessed by a program if the memory addresses used to fetch and execute each instruction "for the most part" are near to each other in space or time. *Space locality* refers to instructions that occupy the same local area of memory. Two instructions exhibit *Time locality* of reference if one will be required shortly after the other.

The degree to which programs possess the property of locality of reference determines how successful a hierachical memory structure will be in providing a shorter memory access than that expected of slow memory alone.

1.1. CACHE HIERARCHIES

A *cache memory architecture* is a hierarchical memory where the fast memory is provided by bipolar semiconductor components (flip-flops) and slow memory is provided by some non-bipolar technology (eg DRAM). The "fast memory" component will be referred to as the *cache*, and the slow memory as the "main memory". The contents of cache initially is usually a copy of some part of main memory.

The probability *p* that a virtual address maps to a cache location is called the *hit ratio* of the cache memory architecture.

In the following examples of different cache architectures, assume that the cache (ie fast memory component) has been divided up into k blocks of m words per block. Thus the cache consists of $m \times k$ words of fast memory.

Let main memory (ie slow memory) be n times larger than the cache. Then main memory consists of $n \times m \times k$ words of slow memory.

Assume each block of m words in the cache is a copy of some block of m words in main memory. Now the cache can only hold a copy of 1/n of the total contents of main memory, so each word of cache will be extended by an additional field to identify from which block of main memory the word has been copied. This field is called the *tag* for the word (or block).

Thus a cache memory architecture can be constructed from two storage components: the cache memory with data words extended to include a tag, and the main memory itself. The components required for the memory management unit will depend on which blocks from main memory can actually reside in the cache simultaneously. Different strategies for mapping a block of main memory to cache are possible, and these strategies affect the definition of the address mapping logic that is required to map a virtual address to a physical address in the cache memory architecture.

2. DIRECT MAPPING CACHE

Memory Mapping Strategy

1. From the physical parameters defining the size of the storage components, the main memory consists of a total of $n \times k$ blocks of m words. Therefore we can divide them up into k sets of n blocks, $S_0, S_1, \ldots, S_{k-1}$.
2. Since there are k cache blocks, associate a different one of the k sets of main memory blocks with each cache block. A main memory block belonging to the set S_i can only be copied to cache block *i*.
3. Since there are n members of S_i, associate a unique identifier, called a *tag*, with each block in the set to distinguish it from the others in the same set. Store the tag of the block from set S_i currently occupying cache block *i*, in the most significant bits of each word in cache block *i*. Thus each word in the cache will have two fields: a **`data`** field and a **`tag`** field.

Address Mapping Function

1. Partition an address referencing main memory into three fields:

a. Field B - set identifier: identifies which set the block belongs to, and therefore which cache block is the one where the word to be accessed might be found.

b. Field T - tag identifier: identifies which main memory block among those that are associated with cache block B contains the word being accessed.

c. Field W - offset field: identifies which of the m words within the block is to be accessed.

2. If the T field matches the tag part of any word in cache block S then the location can be retrieved from the cache. Otherwise the word must be retrieved from main memory. In that case, the main memory block containing the word addressed is brought into cache before the value is returned. The figure below illustrates the mapping. Note the extra bit **v** of each cache word. This is used to determine if there is valid data stored at that location.

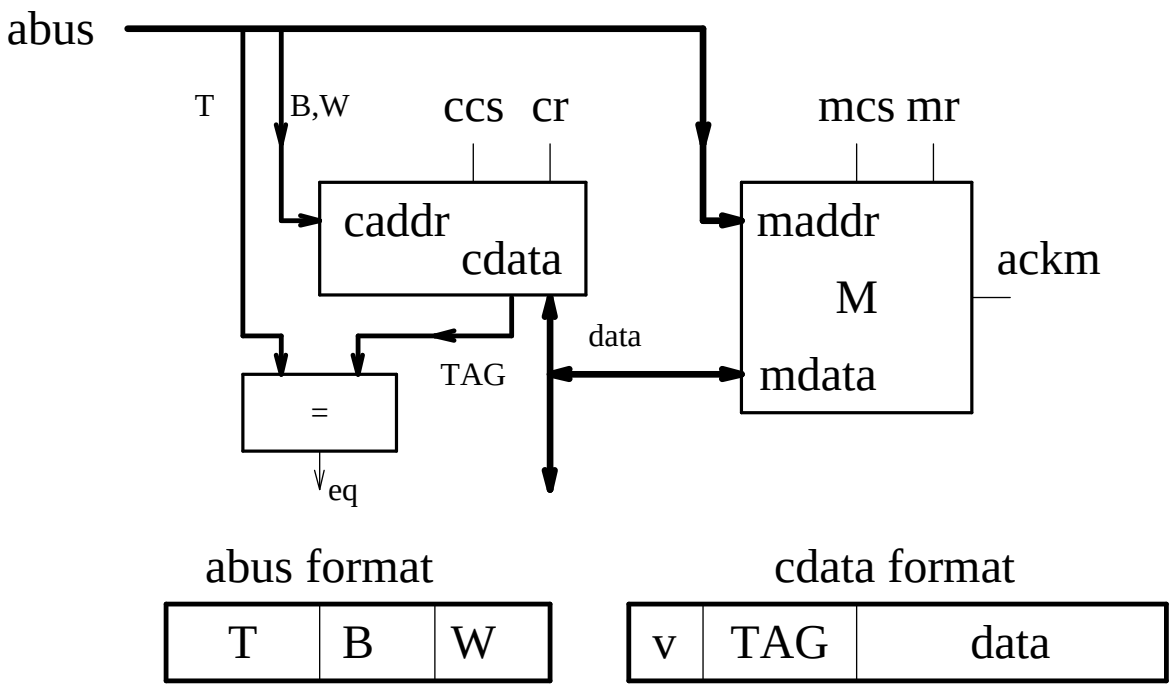

Figure 2-1: Address Logic for Direct Mapping

EXAMPLE: A cache memory architecture employs a main memory of sixteen 32-bit words partitioned into blocks of size 2 words. The cache is 25% the size of main memory.

Therefore n = 4, m = 2, and k = 2, so the address will be partitioned into a B field of 1 bit, a W field of 1 bit, and a T field of 2 bits.

The size of the required memory components are:

1. Main Memory: 16 words, each 32 bits long;

2. Cache Memory: 4 words, each 35 bits long;

The *memory mapping strategy* defines which main memory blocks are allocated to which cache blocks. From the problem specification, main memory is partitioned into 2 groups of 4 blocks.

One possible partition is to place the first four blocks in group 0, and the second four blocks in group 1. That is, if a word belongs to one of the first four blocks, it can be found only in cache block 0, provided a copy of the block to which the word belongs currently resides in that block of cache.

Thus only one of the first four blocks can be in cache at any one time. For this partition, addr(B) = addr(MSB), and addr(W) = addr(LSB) with addr(T) = addr(2,1).

A second possible partition is to place alternate blocks in the same group. In this case blocks 0, 2, 4, and 6 are placed in group 0, with blocks 1, 3, 5, and 7 being placed in group 1. In thisd case addr(B) = addr(1), addr(W) = addr(LSB) and addr(T) = addr(3,2). (Bits are labelled from right to left, with LSB = 0 MSB = 3).

The choice of partition influences the efficiency of cache memory. For programs which are stored in contiguous locations in memory, the second partition results in more consecutive locations of a program being in cache at the same time. With the first partition, no two consecutive blocks (except for block 3 and block 4) can be in cache simultaneously. In other words it takes advantage of spatial locality, since it allows for more instructions that will follow the current one to be in cache at the same time.

3. ASSOCIATIVE MAPPING CACHE

The disadvantage of direct mapping is that only one of n blocks can be placed in a given cache block position at any time. If we relax this constraint, then any one of the $n \times k$ blocks of main memory could occupy any cache block position. This, of course, would require a larger block identifier, and would require that the T field of the address be compared with all tags in cache. A search based on sequential or binary comparisons would take too long. What is required is the logic to compare simultaneously all the tags of the blocks in cache with the T field of the address.

This requires a content addressable memory be used for storing the tags associated with each cache block

A *content addressable memory (CAM)* is any memory device where the value of the data is used to access memory. That access generates an output that determines if the value is in memory, and if so where.

When the tag array is implemented with a CAM, then the T field of an address can be compared simultaneously with all tags in the tag array. The output is a binary vector with a 1 in position i if and only if the T field matches the value in CAM[i]. If the output is non-zero, there will be a 1 in exactly one output bit. The address i of the location of the tag in the CAM can be used to construct the cache address by concatenating it with the W field of the `abus` address.

The requirements of a CAM (or *associative memory*) are met by the following architecture. Each row of ‘‘cells’’ defines a word of the CAM. The associated row of XNOR gates provides the comparison logic permitting the input **`tag`** to be compared with the corresponding word of CAM. The select logic allows a particular row to be enabled for input, in order to store a new value in that word of CAM:

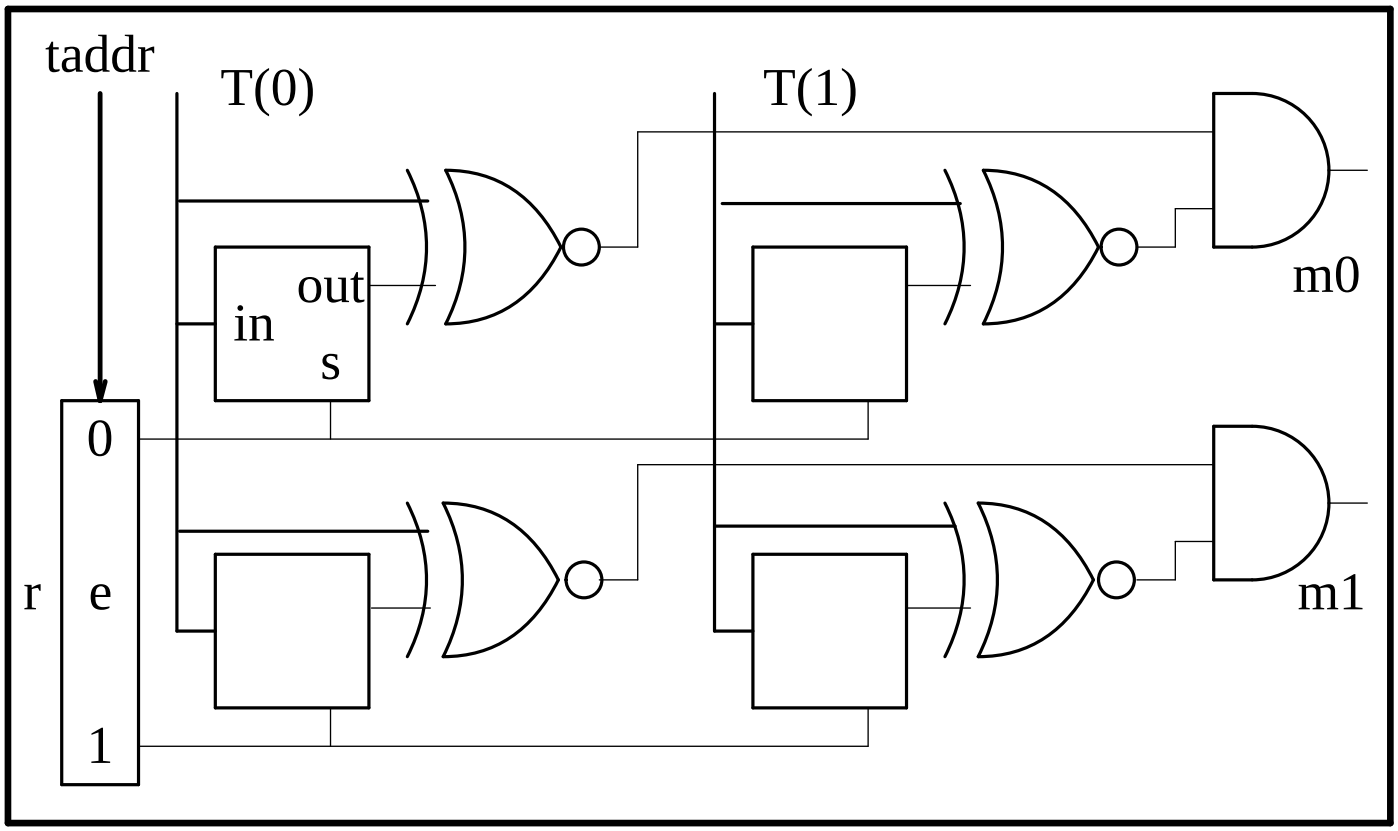

Figure 3-1: 2x2 CAM Organization

Using a content addressable memory, the addressing logic for retrieving a value from a location in cache would be organized as follows:

1. The T-field is compared with all tags stored in the TAG array.
2. The match vector is decoded by the code converter. Its output is the binary representation of the location in the match vector where a bit has been set (if any). This output provides the block address to the cache memory. The W field of the address is used to complete the address to cache.

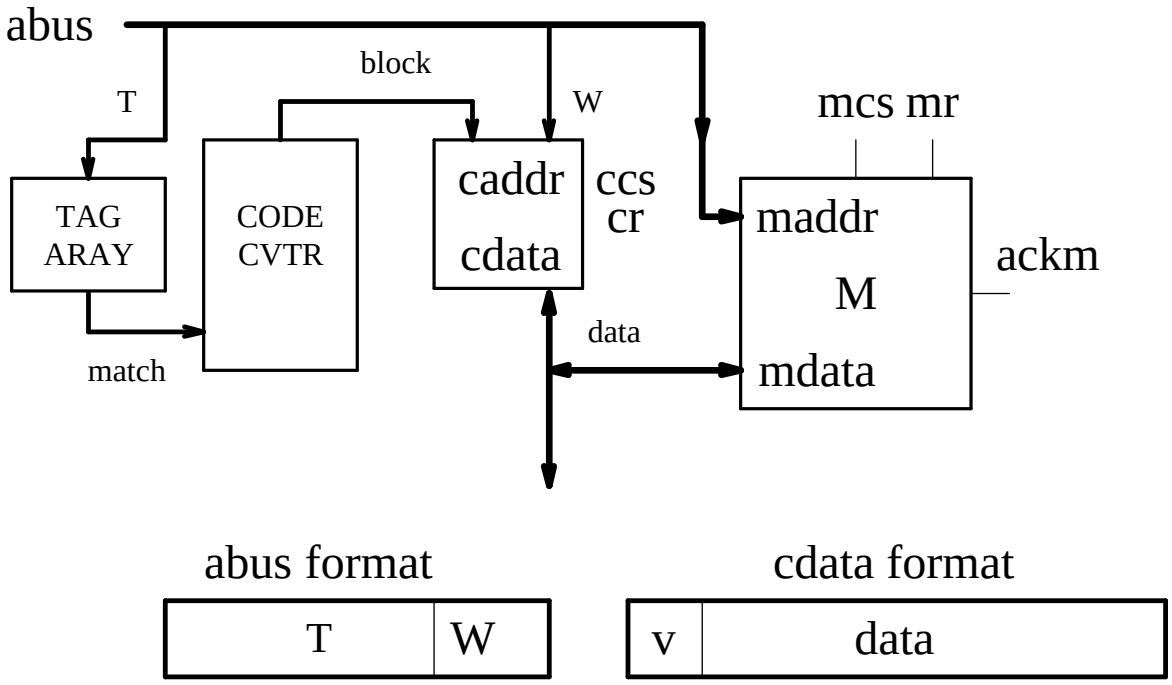

Figure 3-2: Address Logic for Associative Mapping

3.1. REPLACEMENT STRATEGIES

An examination of the associative mapping strategy reveals that when there are no further free cache blocks for transfering a block from memory, then a decision must be made as to which occupied cache block to replace.

Possible strategies include:

- **Random**: The selection of the replacement block is based on some randomized selection procedure and does not make use of any knowledge of the application.
- **FIFO**: The oldest cache block loaded is overwritten. Such algorithms

employ a First-In-First-Out (FIFO) cache block replacement strategy. This strategy is reflected in the ASM diagram for associative cache given in last week's notes.

- **LRU**: Although random and FIFO are the simplest replacement strategies to implement, they do not take into account the possibility that "other" blocks may be more suitable replacement candidates, such as those that are seldom referenced. The Least-Recently-Used (LRU) replacement strategy attempts to address this issue. By keeping track of the time since last reference for each cache block it is possible to choose the "most stale" cache block; that is, the one least recently referenced.

4. WRITING TO CACHE MEMORIES

With retrieval, the contents of memory remain unchanged and therefore each block of cache is an exact copy of some block of main memory. However, with the storage of a value two options exist as to how to proceed:

1. **Write-Through**

   With this technique, every update of cache automatically updates main memory. The advantage of this approach is that memory is always up-to-date. However no benefit of the memory architecture occurs since every storage access by definition accesses main memory. The performance of the cache memory architecture is determined by the hit ratio for retrievals and the ratio of retrieval accesses to storage accesses

2. **Write-Back**

   With this technique, only the cache is updated. A flag associated with the updated cache block is set and if and when it becomes necessary to overwrite that block, it is written back to main memory if its flag is set. This strategy improves the performance of cache memory particularly in programs where the same location is referenced frequently. Additional hardware is required for the flags and for making it possible to perform a block transfer from cache to main memory.

The write-back method takes advantage of cache memory, but with additional cost and complexity of the extra hardware requirements. This plus the fact that "typical" programs have a significantly greater number of retrieval requests than storage requests, have made the write-through technique a more common solution in the past to the problem of storing in cache memories.

Recently however, with the development of new memory technologies including

synchronized RAM, it is possible to improve the efficiency of block transfers and consequently write-back systems are becoming more common.

5. INPUT/OUTPUT INTERFACES

The logic required for communication between the distinct physical modules of a computer system (Processor, memory, I/O devices) is collectively referred to as the *intrasystem communications* logic. The communications logic needed to transfer data between different computer systems is referred to as the *intersystem communications* logic. Intrasystems interface design addresses two problems:

1. The propagation delay caused by the physical separation of devices.
2. The variation in device interface requirements.
3. The limitation imposed by the available inputs and outputs on the CPU chip.

5.1. BUS ORGANIZATION

Any collection of signal lines between devices constitutes a *communications bus*. The purpose of a bus determines its type:

- **processor-memory bus**: Addresses data traffic between a processor and memory. It is typically short, to minimize delay, and possesses a wide data path (at least one word).
- **input-output bus**: Provides the communications link between input/output devices and the processor.
- **backplane bus**: A "hybrid" bus permitting both memory and input/output devices to share the same bus.

Bus adaptors are used to permit two different buses to be connected. Input and output devices are not usually connected directly to a bus but are connected via an **Interface Unit** that matches the device specific signal lines to the standard reflected by the common processor-memory or backplane bus.

To design an Interface Unit requires a determination of the communication requirements for the Processor to fully interact with the device. These can be classified as follows:

- **Command Decoding**: The Processor must be able to request a particular function or activity of the device;
- **Data**: The Processor must be able to transmit/receive data according to the type of function request;

- **Status Reporting**: The external device may have a different clock period from that of the Processor; hence, it must be able to report task completion to the Processor (or error detection);
- **Address recognition**: The Processor must be able to specify uniquely which of several alternate devices the I/O function request is addressed to.

These requirements motivate the structural organization of an interface unit as defined in the a "behavioral" logic diagram on the next page.

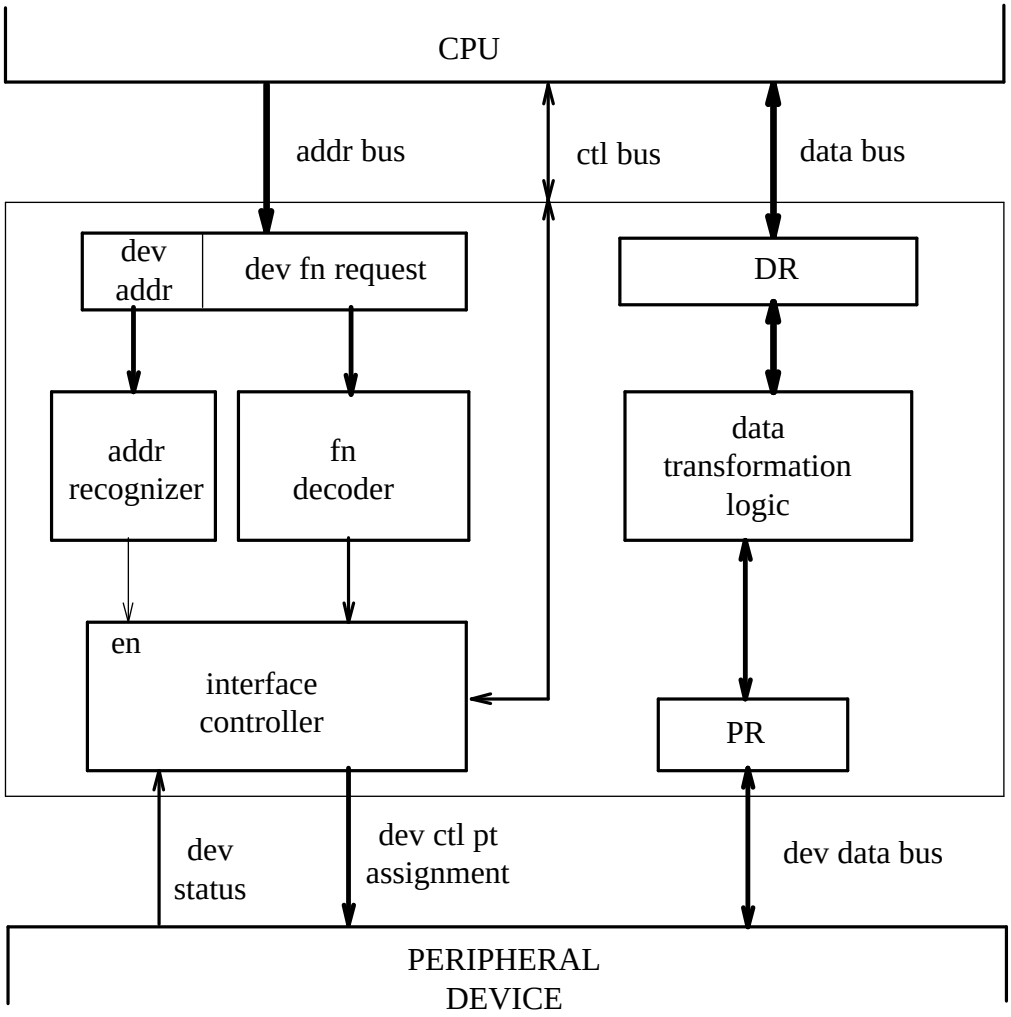

Figure 5-1: Basic I/O Interface Organization

The types of function that can be requested by the Processor via the address bus are:

- **Control**: Activate the device in a particular way;

- **Test**: Monitor some aspect of the device status;
- **Data Output (from Processor)**: Transfer one data item from I/O bus to the DATA Register (DR);
- **Data Input (to Processor)**: Transfer one data item from device to Peripheral Register (PR).

6. BUS ARBITRATION

In a common bus system only two devices can be using the bus at a time; one for sending (the source), and one for receiving (the destination). All other devices must be prevented from accessing the bus at inappropriate times. The problem of who gains control of the common bus is called the *bus arbitration problem*. Devices sharing a bus can be designed to provide one or both of the following roles:

- **bus master**: Any device which can control a bus is a potential bus master. Otherwise it is referred to as a *slave*.
- **bus arbiter**: A device which controls which bus master will be given access to the bus. There is at most one bus arbiter associated with a common bus.

A *One bus-master system* is an architecture where only one device can send data over a bus. In such a case no arbiter is required. Otherwise the communications architecture will employ a *mulitple bus-master system* and require a bus arbiter. In such situations the following control lines are required:

- Bus Request: (*breq*) Used to initiate a request for bus control.
- Bus Grant: (*bgrant*) Used to acknowledge a bus request.
- Bus Ready: (*brel*) Used to advise the bus arbiter when the bus is free.

The bus arbiter employs a protocol to address the following questions:

1. How does the arbiter identify which Interface Unit issued a bus request?
2. If several devices issue requests, which one is serviced?

There are several ways to define a protocol with hardware:

1. **Daisy Chain Arbitration**
    a. Device Interface Unit places request for bus access. **`breq <- 1`**.

b. Arbiter holds request until **`brel = 1`** (i.e., bus free) and then asserts **`bgrant`**.

c. **`bgrant`** nis passed from one interface unit to next until received by the module initiating the request;

d. Initiating module receives **`bgrant`** and takes control of the bus, thus becoming the bus master. **`brel`** is set to 0 to indicate the bus is now busy.

e. When finished the bus master frees the bus by asserting **`brel`**.

The basic architecture implied by this strategy is to form a "daisy chain" of I/O interfaces; that is, each interface is connected to the next in such a manner that information can be transmitted in one direction along the chain of interfaces. One possible way to define such a chain is as follows:

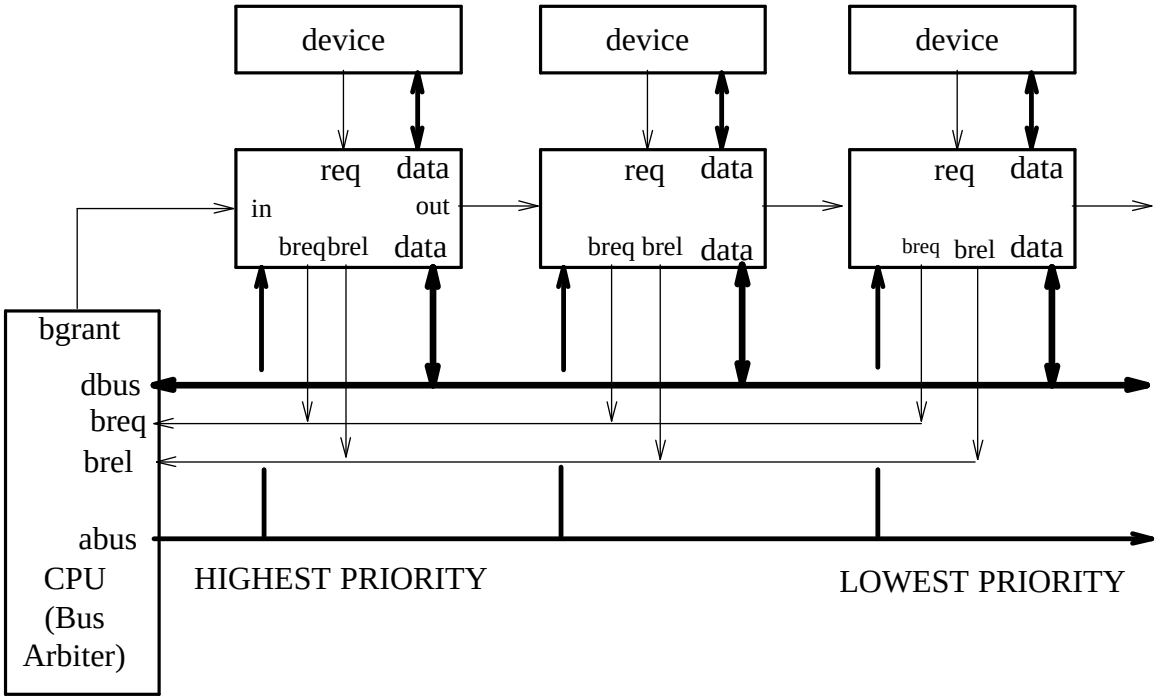

Figure 6-1: Daisy Chain Arbitration

2. **Parallel Arbitration**

Daisy-chaining provides a simple way of imposing a priority on the

order in which interrupts from devices are handled: the "closer" the interface is to the front of the chain, the higher the priority. For the example given, the front of the chain is that interface that first receives the interrupt acknowledge from the Processor.

By providing a separate bus arbiter or "device selector" to process bus requests from the interfaces (or device requests directly from the peripheral units themselves), a more flexible priority management capability can be realized. In particular, the ability to program the behavior of the device selector from the Processor allows software to be used to define the handling of bus requests. The following diagram illustrates the basic architecture:

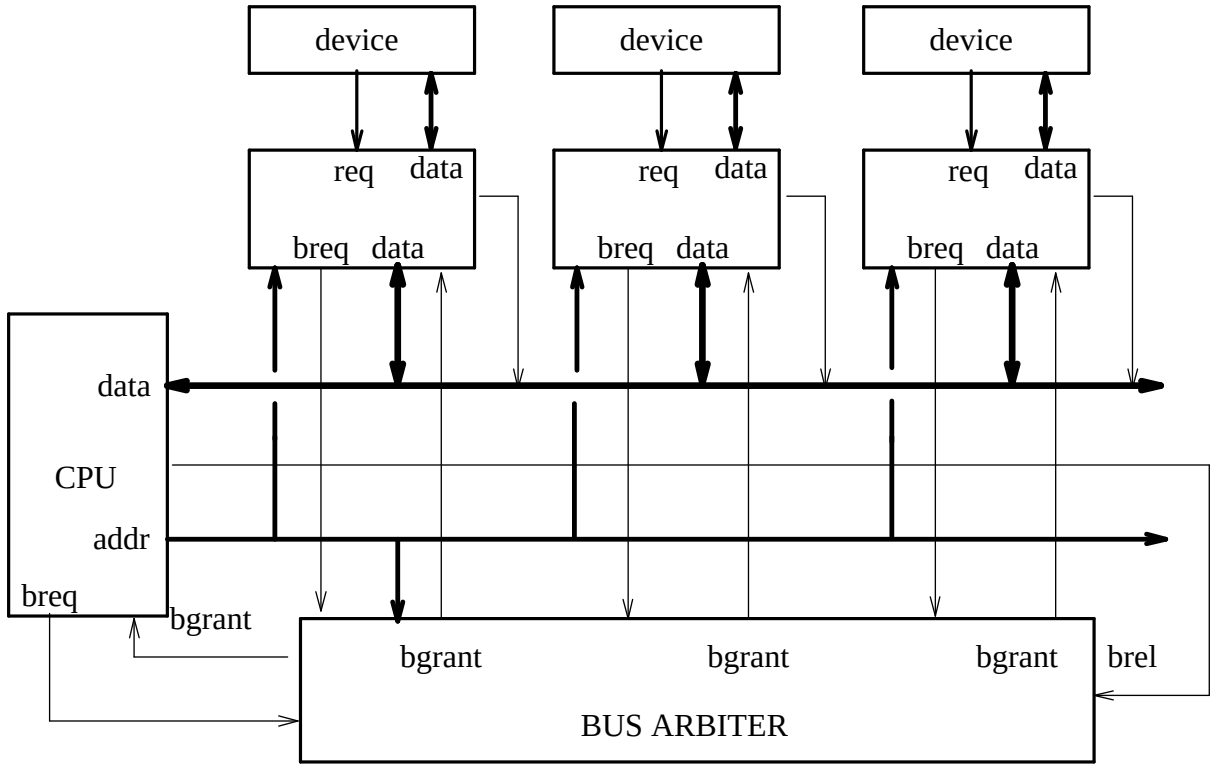

Figure 6-2: Parallel Arbitration

The advantages of this approach are:

a. The priority is not determined by the physical position of the I/O module;

b. A masking facility for bus requests can be provided;

c. The priority of request handling can be modified after hardware construction.

3. **Distributed Arbitration**

Rather than employ special hardware or require that devices be connected in a particular order, it is possible to incorporate the priority selection into the Interface Units themselves:

- **Self-Selection**: Each interface unit can write its identity to part of the data bus. Since it can read all of the bus, it can identify who wants access to the bus; in particular, it can determine by knowing who else wants the bus whether it has highest priority. If so it seizes the bus.
- **Collision Detection**: It is possible through the use of a carrier signal for a device to transmit data on the bus, and at the same time to listen and see if the contents of the bus are what it is transmitting. If not, then a collision has occurred, and both devices stop transmitting, with each retrying after a short but different amount of time has elapsed.