1 Why use a bus?

The purpose of a bus is data transport from one location to another. A higher performance, lower latency system can almost certainly be built using a fully interconnected network, i.e. a cross-bar switch where every device has a direct, dedicated connection to every other device. However this is very expensive in practice. Buses (shared communication highways) are used as a cheaper (in terms of wire/area and control complexity) alternative. It should be noted that a bus approach to interconnect probably consumes more power than its equivalent crossbar system due to the high load incurred by the bus. There are three categories of bus that are in common use:

- serial bus, e.g. RS232, ethernet - typically low bandwidth communications systems
- multiplexed address/data parallel bus, e.g. PCI, SCSI - generally backplane buses
- separate channel address and data parallel bus, e.g. MARBLE(async), AMBA(sync), PI-BUS(sync) - usually on-chip buses / processor buses with high throughput requirements.

This lecture is primarily concerned with the latter category, although the principles discussed are generally applicable to all three.

A typical system based around an on-chip bus is shown in Figure 1. This system represents the AMULET3i chip, a fully asynchronous VLSI system under construction by the AMULET group here at Manchester. The bus, MARBLE (The Manchester AsynchRonous Bus for Low Energy) in this system has 4 initiators (CPU-Fetch, CPU-Data, DMA, Test) and 6̃ Targets (Local-Bus-Ram,DMA-config, External Memory, ROM, Sync-bridge, control registers) - typical of the requirements of an embedded system.

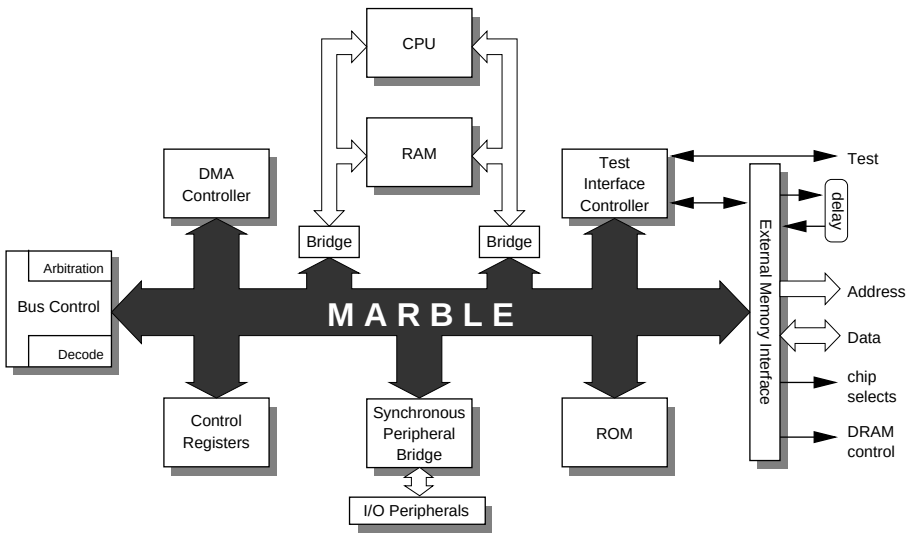


Figure 1: AMULET3i System

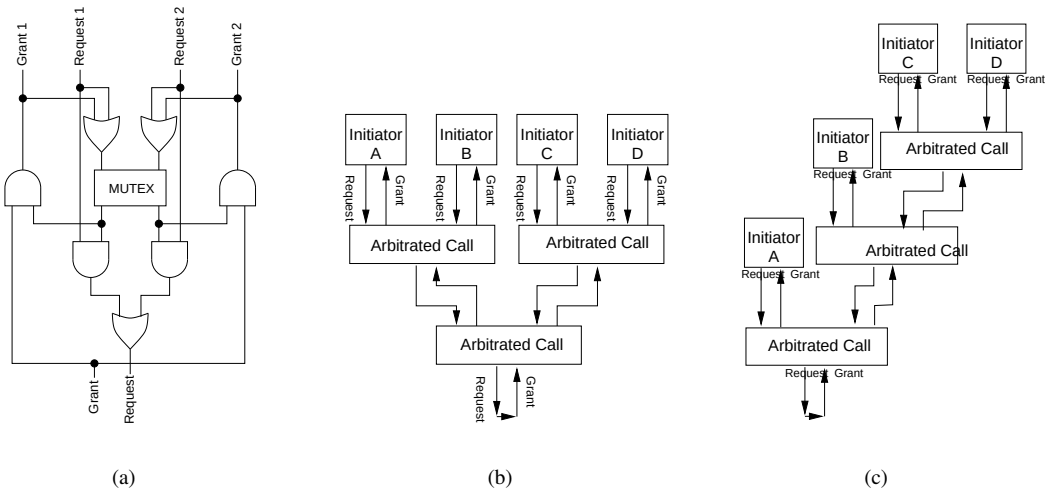
2 Bus Control Basics

2.1 Single Master Bus

A simple, single-master bus requires nothing more than an address-decoder to activate the appropriate target at the correct time. An address decoder is a simple piece of combinatorial logic that decodes the address to generate a select line. There are two options here, centralized versus distributed. No real difficulty with either approach. Centralized uses more wire, distributed uses more gates. Distributed is more modular and as such was more appropriate to backplane buses which could have more devices connected in expansion slots. This is not really an issue in on-chip buses.

2.2 Multi-Master Bus

When multiple masters can compete for use of the bus, an arbitration system is also required. This must ensure that only one master uses the bus at any one time. Methods such as collision detection (as in ethernet) and priority-algorithms based on sampling after a specified delay or clock period (as in SCSI, VMEbus, AMBA, PI-Bus...) are not really suitable to asynchronous VLSI. Here it is much better to make use of the MUTEX element. By adding a wrapper around the MUTEX as shown in Figure 2(a), an arbitrated call block can be created. An n-way arbiter can then be constructed from multiple arbitrated calls as shown in Figure 2(b)



(a) (b) (c)

Figure 2: Arbitration: Tree Arbiters constructed from Arbitrated Call Blocks

The arbiter tree must be constructed using arbitrating calls, rather than simply from MUTEXes so that every arbitration 'propagates to the root of the tree'. Otherwise it would be possible for starvation of some clients to occur. Different tree structures allow bandwidth to be crudely apportioned. (c.f. priority in a synchronous bus arbiter) - the tree in Figure 2(b) gives equal share ($\frac{1}{4}$) to all the initiators whereas the tree in Figure 2(c) gives $\frac{1}{2}$ the bandwidth to initiator A, $\frac{1}{4}$ to initiator B and $\frac{1}{8}$ to each of C and D. Note that the root-node arbitrated-call in these trees can be simplified to just a MUTEX.

Most operating systems use semaphores of some form. The implementation of a semaphore requires that an uninterrupted sequence of memory transfers can be performed, typically a read-modify-write operation on a given memory address. Clearly a bus-arbiter must allow an initiator to claim the bus for as many uninterrupted cycles as it requires, i.e. lock the bus so other initiators cannot use it. This is implemented as part of the bus arbitration system, and in async. VLSI it is easily achieved by an initiator not lowering the arbitration-request to the bus arbiter until its atomic transactions have been performed.

2.3 Pipelining to Improve Performance

Pipelining is used in buses for the same reasons as it is used in processor design, improved throughput. The main application of pipelining in buses is in the overlap of the arbitration for the next cycle with the execution of the current cycle. Also, the address cycle of a transfer can be overlapped with the data cycle of the preceeding transfer. As with microprocessors, the pipelining in asynchronous systems is much more flexible/elastic than in synchronous systems.

However, adding pipeline latches to allow the address and data cycles of a transfer to be staggered / overlapped introduces a new problem: it is possible for data cycles to occur in a different order from the address cycles, hence sending the wrong data to the wrong device.

Consider the system shown in Figure 3. The pipeline latches have been added at the targets on the address ports of the interfaces, i.e. at each point where packets are taken off the address-channel at the targets. The latches within the interfaces, between the address and data units are to store any control information required, e.g. whether this is a read or write transfer (so that the data unit knows what to do). If say, I0 sends an address to T0 for a read transaction, this address is stored in the latches at T0, releasing

the address channel so that it can be used by another transfer, (i.e. pipelining the address cycle of the next cycle with the data cycle of this transfer). So, the next address cycle can be started, say I0 sends an address for a read to target 1 (T1). Now, what happens if T1 is faster than T0, then in this case, the read data can be returned to I0 in the wrong order. Also, since it is possible for data to come in any order at any time, the data channel clearly needs to be the subject of an arbitration and there is the problem of how to route the data packets to the correct place, e.g. if I0 send an address to T0 for a read, and I1 sends an address to T1 for a read, how do we ensure that the correct read-data goes to the correct initiator. Similar problems exist for write data.

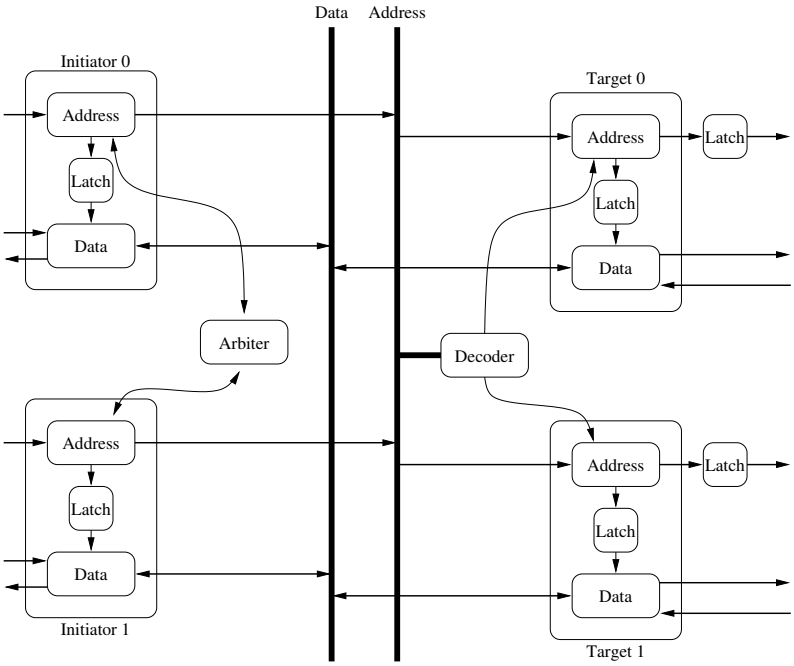


Figure 3: Adding Pipeline latches to a Bus System

To maintain the pipelining but ensure that the bus still functions correctly, an ordering constraint must be applied somewhere in the system. Buses can be categorized by the ordering constraint that they use which is typically a variant of one of two approaches - Interlocking, or Split (tagged) Transfer. These approaches apply equally well to multiplexed address-data, or separate channel buses.

2.3.1 Address-Data Interlock Methods - In-order buses

After address cycle n has been performed, ensure that address cycle n+1 cannot begin until after data transfer n has started. By always doing the sequence address-cycle, data-cycle, address-cycle, data-cycle ... there can never be any confusion as to which data packet corresponds to which address packet. (PCI is a synchronous example of such a bus). The initiator thus arbitrates for both the address, and the data cycles when it obtains access to the bus. Also, between the address and data cycles (which may be a long time for a read from a slow device) the bus is stalled waiting for the transfer to complete.

2.3.2 Split Transfer with Sequence Colours - Any-order buses

Each cycle of a transfer carries two additional pieces of information, an initiator ID-tag (the tag) and a sequence indicator (a ‘colour’). All address and data cycles are arbitrated for separately. The tag is used to route the data cycle to the correct initiator for a read, or to the correct target for a write. The colour is necessary because data cycles can occur out of order when compared to the address cycles. E.g. a read from a fast device may complete before a read from a slow device that was started before it. The colour is thus used, in conjunction with a re-order buffer to ensure that data is presented to a device in the correct order. SCSI has a mode where it supports such split transfers.

2.4 Example: MARBLE Flow Control

MARBLE has separate 32-bit address and data channels and uses a reduced/simplified split transfer system with target-started data cycles and single-outstanding-issue. The split transfer allows transfers between different devices to be interleaved on the bus, rather than stalling the bus between the start and finish of a transfer. Because the bus is split transfer, an initiator tag must be transmitted with all address and data cycles (2-bits here for 4 initiators), and separate arbiters must be used for the address and data channels.

Only allowing targets to start data cycles means that a target will never receive data out of order. This also reduces the arbitration requirements (only targets arbitrate for the data channel) when compared to a full split-transfer system (targets and initiators arbitrate for the data channel). The single-outstanding issue means that an initiator will never receive data in the wrong order, since it waits for the start of the data cycle before proceeding with the next address cycle (i.e. the initiator locally behaves as if it were on an interlocked A-D bus).

3 Implementation Issues and Channel Signalling for Async. VLSI Buses

3.1 Surely we want to use Delay Insensitive Dual Rail

Consider the fundamental component of a CMOS bus, the tristate driver. It is very difficult to construct a delay-insensitive low power CMOS system based around such a component. (Why?) Further, since a bus has one sender and (possibly) many receivers of any comunication, it is necessary to make assuptions about isochronic forks in the wires of the bus. Some degree of timing verification by simulation is thus going to be necessary for any asynchronous (and sychronous also) bus. Further, since an on-chip bus is used to connect macrocells (i.e. pre-laid-out blocks of logic), then the wrinig of the bus is going to be routed close together, simplifiying the timing verification of the bundling constraints in a single-reail system. The need for such timing assumptions and the cost involved in dual rail design lead one to the conclusion that a single-rail protocol is probably the most appropriate for a bus.

3.2 2-Phase v 4-Phase. What protocol do we want to use?

There are many issues to consider here:

- All tristate-able bus lines should have bus keepers (including signalling lines if they are tristateable) to ensure they do not float/oscillate (power implications).
- Wired-OR signalling as used in backplane buses and early NMOS VLSI buses cannot be used in CMOS technology when zero idle power is a design objective.
- Gated (centralized) signalling using a wide OR gate to combine the individual requests from each interface to give the bus request line, and likewise for the acknowledge is possible when using 4-phase signalling, but the 2-phase equivalent results in a large, complex circuit.
- Tristate signalling - relying on the charge retention to hold the level of the signalling line when not driven, and then any module who drives the line does so through a tristate driver - gives a more modular system
- 2-phase tristate signalling drive handover is difficult because the new participant(s) in the transfer must be able to assume the current state of the bus signalling lines before commencing the transfer.
- 4-phase tristate signalling handover is relatively easy since the bus idle state is always the same (e.g. req=ack=0)
- Tristate signalling relies on charge retention to hold values of the lines - poor noise immunity
- 2-phase has less edges on heavily loaded bus signal lines (hence lower power)
- 2-phase requires bundled lines to be not-driven by time acknowledge event occurs, so that next initiator can start driving them immediately. i.e. must rely on charge retention to hold value of bundled lines

- 4-phase has the return to zero phase, and so bus cycles will take longer (hence occupying the shared bus for longer) than 2-phase bus cycles.
- 4-phase can be used with early data-validity protocols so that the bundled lines are driven throughout the period in which their value is used/latched by the receiver

As an aside, it is worth noting that the tristate drive handover problem is usually solved in synchronous buses by either having a null/idle cycle or switching tristates on/off just after a clock edge.

3.3 Example: MARBLE Signalling

MARBLE uses 4-phase signalling with the bundling as shown in Figure 4. The version being built uses centralized OR gates to merge local requests/acknowledges into the global bus-address-request, bus-data-request and corresponding acknowledges. The address request and data acknowledge are always controlled by the initiator, and the address acknowledge and data request are always controlled by the target.

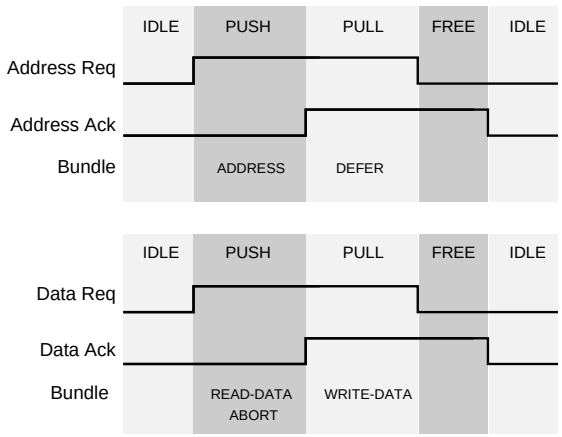


Figure 4: MARBLE Signalling and Bundling

The address is always pushed from initiator to target (via the address decoder). Read data is pushed from target to initiator (using the tag to route it to the correct initiator), and write data is pulled from the initiator when the target is ready for it.

4 Advanced Issues

The principles covered thus far are sufficient for most systems. Other more advanced features/issues discussed here are required in some circumstances.

4.1 Bus Errors / Aborts

The main error that can occur in a bus-based system is that the address does not decode to a valid target. This could be detected either by the bus-decoder or through a time-out mechanism. Other errors which may be detected by the bus control logic, or by the target device include invalid operations, e.g. write to a read-only device and privilege violations, e.g. user mode access to a supervisor-mode-only I/O device. All of these must be signalled back to the (correct) initiator. This may be done by globally broadcasting an interrupt line, or as in the ARM/AMULET systems using an abort, a precise interrupt bundled as an extra signal in the data cycle.

4.2 Defer

Support for a hardware-retry (defer) primitive is often useful if not essential in a bus. Its inclusion in the protocol as an (address-time response) allows for more advanced features:

- It allows the bus to be released after starting a transfer to a device that takes a long time to complete, so other initiators can use the bus whilst deferring the completion of this transfer. This provides reduced latency to these devices and improves overall system performance.
- When a device is busy (possibly in a recovery cycle) and cannot accept another transaction, in a system without defer any attempt to access this device across the bus would result in the bus being stalled with the attempted transfer sat on it. Defer allows the initiator to be told to retry the command later, i.e. defer starting the transfer.
- Bridges between multimaster buses require that one or both of the buses connected to the bridge supports deferral. To illustrate the necessity of this feature, consider the case where device A on bus 0 tries to communicate with device C on bus 1 at the same time as device D on bus 1 tries to communicate with device B on bus 0. Both transfers require the use of the bridge and the two buses, so the bridge must reply with a defer command to one initiator to allow the other transfer to complete, otherwise the system would simply deadlock. Note here that maintaining the atomicity of an atomic series of transfers across a bridge is difficult, requiring added complexity and bus support.

The first two points here are irrelevant in a split-transfer system. The last point here is really the most significant. In general, any bus that has a device (such as a bridge), that can behave as both an initiator and a target requires a defer mechanism if it is possible for the device to get into a state where it must be able to perform a bus transfer as an initiator before it can act as the target of a transfer.

5 Further Improving the System Performance

5.1 Sequential Optimisations

Some target devices and memory systems, notably DRAM, can operate with reduced cycle times for all but the first access in a series of accesses to the same or sequential addresses. In DRAM, this is known as paged mode access, where the first access requires both a RAS and CAS precharge, but subsequent accesses to the same page require only a CAS precharge which is typically $\frac{1}{3}$ or $\frac{1}{2}$ of the total RAS-CAS access time. To exploit this behaviour in a macrocell-based system requires additional bus support to convey the spatial locality properties of the addresses. It is preferable to communicate this information in advance of the sequential access when possible, since detection of sequential addresses when they arrive at the target is both expensive and too late to be of much use. MARBLE uses a two-bit code transmitted as part of the address packet to indicate the three situations where these types of optimization can be beneficial, as shown in Table 1.

Table 1: SEQ and PRED Spatial Locality Coding

PRED	SEQ	Meaning	Typical Use
0	0	Non sequential	Random access
0	1	Sequential to previous address	Instruction Fetch
1	0	Next address will be in same 2^n page	Cache Line Fetch wrap-around
1	1	Next address will be sequential and in same 2^n page	DMA transfer, Cache Line Fetch, Block Move

5.2 Caches

The final twist to the bus story is when devices on a multimaster bus include caches. The problem is that a cache must be kept consistent, and so, for example if your core has a cache which contains the contents of memory at address x, and the DMA controller subsequently writes to address x, steps must be taken to ensure that the cache does not supply invalid data to the processor. Approaches to this problem include:

- Don't cache areas of the address-space that can be accessed by multiple initiators.
- The cache could snoop the bus and invalidate the line containing address x when it sees this address transferred on the bus

- The cache could snoop the bus and update the stored value for address x

These solutions work satisfactorily for reads, and for writes in a write-through cache. However, for a write-back cache there is the added problem of what happens if another initiator reads from the address of a cached write that has not been written back. Solutions to this problem include:

- Don't cache areas of the address-space that can be accessed by multiple initiators.
- The cache could snoop the bus and defer or abort the transfer so that it can write to the target before the initiator retries the transfer
- The cache could interact with the address decoder, and respond as the target supplying the correct data

In all cases when using a cache, consistency is impossible to maintain if caching addresses that lie on a second bus which is the other side of a bridge.