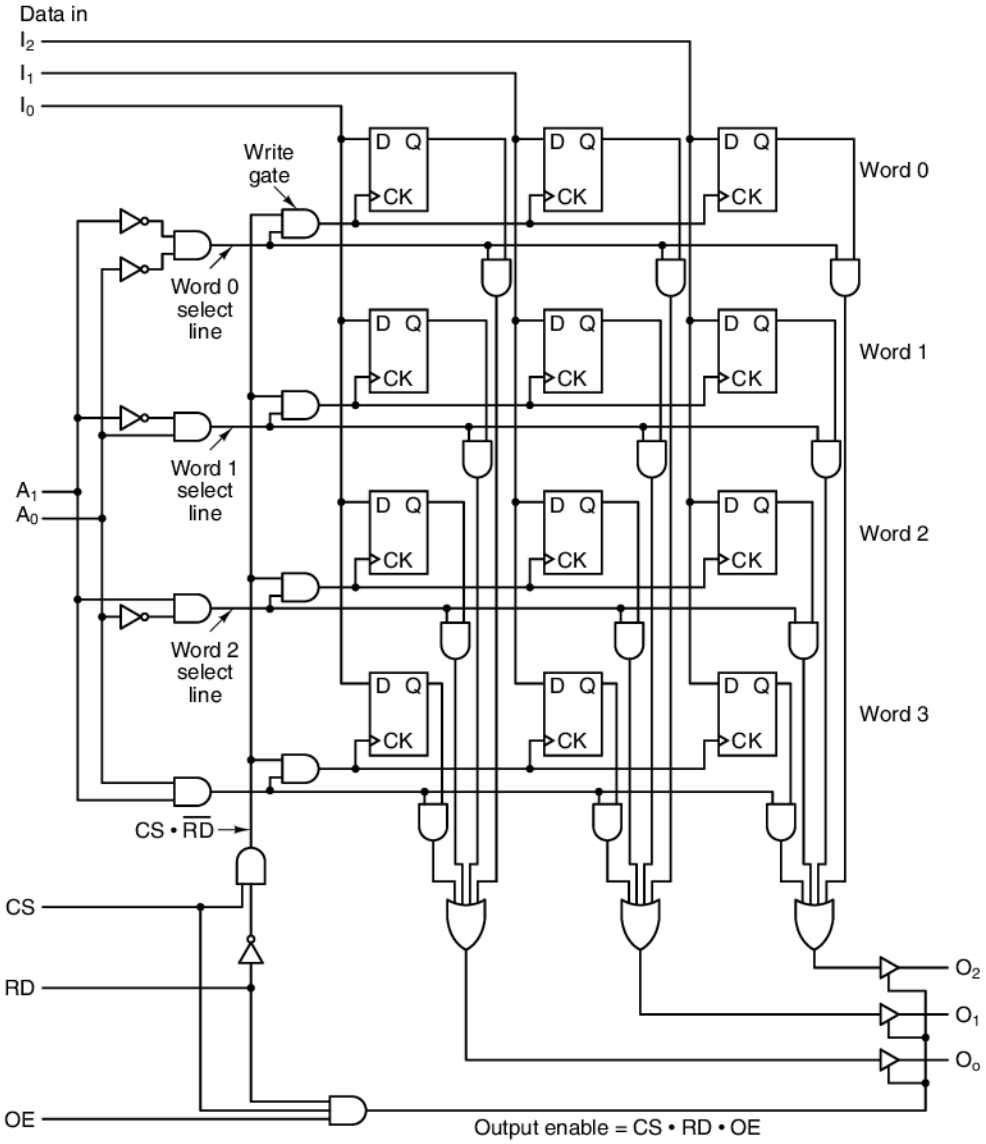


Figure 3-28. Logic diagram for a 4 × 3 memory. Each row is one of the four 3-bit words. A read or write operation always reads or writes a complete word.

enabling one of the four write gates, depending on which word select line is high. The output of the write gate drives all the CK signals for the selected word, loading the input data into the flip-flops for that word. A write is done only if CS is high and RD is low, and even then only the word selected by A_0 and A_1 is written; the other words are not changed at all.

Read is similar to write. The address decoding is exactly the same as for write. But now the CS · $\overline{\text{RD}}$ line is low, so all the write gates are disabled and none of the flip-flops is modified. Instead, the word select line that is chosen enables the AND gates tied to the Q bits of the selected word. Thus, the selected word outputs its data into the four-input OR gates at the bottom of the figure, while the other three words output 0s. Consequently, the output of the OR gates is identical to the value stored in the word selected. The three words not selected make no contribution to the output.

Although we could have designed a circuit in which the three OR gates were just fed into the three output data lines, doing so sometimes causes problems. In particular, we have shown the data input lines and the data output lines as being different, but in actual memories the same lines are used. If we had tied the OR gates to the data output lines, the chip would try to output data, that is, force each line to a specific value, even on writes, thus interfering with the input data. For this reason, it is desirable to have a way to connect the OR gates to the data output lines on reads but disconnect them completely on writes. What we need is an electronic switch that can make or break a connection in a fraction of a nanosecond.

Fortunately, such switches exist. Figure 3-29(a) shows the symbol for what is called a **noninverting buffer**. It has a data input, a data output, and a control input. When the control input is high, the buffer acts like a wire, as shown in Fig. 3-29(b). When the control input is low, the buffer acts like an open circuit, as shown in Fig. 3-29(c); it is as though someone detached the data output from the rest of the circuit with a wirecutter. However, in contrast to the wirecutter analogy the connection can be subsequently restored in a fraction of a nanosecond by just making the control signal high again.

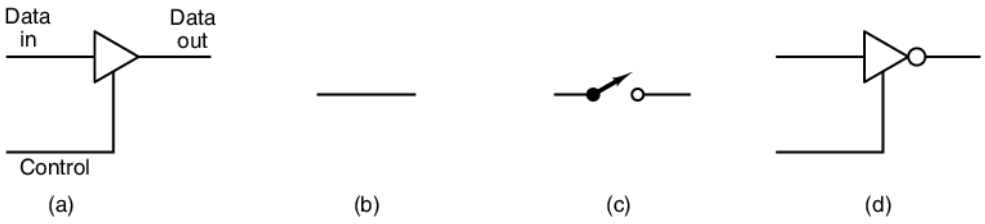


Figure 3-29. (a) A noninverting buffer. (b) Effect of (a) when control is high. (c) Effect of (a) when control is low. (d) An inverting buffer.

Figure 3-29(d) shows an **inverting buffer**, which acts like a normal inverter when control is high and disconnects the output from the circuit when control is low. Both kinds of buffers are **tri-state devices**, because they can output 0, 1, or none of the above (open circuit). Buffers also amplify signals, so they can drive many inputs simultaneously. They are sometimes used in circuits for this reason, even when their switching properties are not needed.

Getting back to the memory circuit, it should now be clear what the three noninverting buffers on the data output lines are for. When CS, RD, and OE are all high, the output enable signal is also high, enabling the buffers and putting a word onto the output lines. When any one of CS, RD, or OE is low, the data outputs are disconnected from the rest of the circuit.

3.3.5 Memory Chips

The nice thing about the memory of Fig. 3-28 is that it extends easily to larger sizes. As we drew it, the memory is 4×3, that is, four words of 3 bits each. To extend it to 4×8 we need only add five more columns of four flip-flops each, as well as five more input lines and five more output lines. To go from 4×3 to 8×3 we must add four more rows of three flip-flops each, as well as an address line A_2. With this kind of structure, the number of words in the memory should be a power of 2 for maximum efficiency, but the number of bits in a word can be anything.

Because integrated-circuit technology is well suited to making chips whose internal structure is a repetitive two-dimensional pattern, memory chips are an ideal application for it. As the technology improves, the number of bits that can be put on a chip keeps increasing, typically by a factor of two every 18 months (Moore's law). The larger chips do not always render the smaller ones obsolete due to different trade-offs in capacity, speed, power, price, and interfacing convenience. Commonly, the largest chips currently available sell at a premium and thus are more expensive per bit than older, smaller ones.

For any given memory size, there are various ways of organizing the chip. Figure 3-30 shows two possible organizations for an older memory chip of size 4 Mbit: $512\text{K} \times 8$ and $4096\text{K} \times 1$. (As an aside, memory-chip sizes are usually quoted in bits, rather than in bytes, so we will stick to that convention here.) In Fig. 3-30(a), 19 address lines are needed to address one of the 2^{19} bytes, and eight data lines are needed for loading or storing the byte selected.

A note on terminology is in order here. On some pins, the high voltage causes an action to happen. On others, the low voltage causes the action. To avoid confusion, we will consistently say that a signal is **asserted** (rather than saying it goes high or goes low) to mean that it is set to cause some action. Thus, for some pins, asserting it means setting it high. For others, it means setting the pin low. Pins that are asserted low are given signal names containing an overbar. Thus, a signal named CS is asserted high, but one named $\overline{\text{CS}}$ is asserted low. The opposite of asserted is **negated**. When nothing special is happening, pins are negated.

Now let us get back to our memory chip. Since a computer normally has many memory chips, a signal is needed to select the chip that is currently needed so that it responds and all the others do not. The $\overline{\text{CS}}$ (Chip Select) signal is provided for this purpose. It is asserted to enable the chip. Also, a way is needed to distinguish reads from writes. The $\overline{\text{WE}}$ signal (Write Enable) is used to indicate that data are

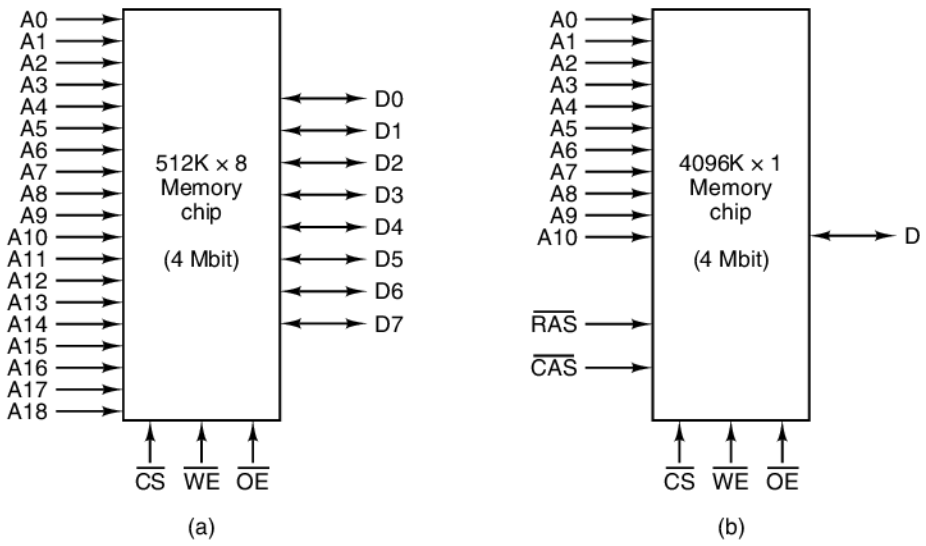


Figure 3-30. Two ways of organizing a 4-Mbit memory chip.

being written rather than being read. Finally, the $\overline{\text{OE}}$ (Output Enable) signal is asserted to drive the output signals. When it is not asserted, the chip output is disconnected from the circuit.

In Fig. 3-30(b), a different addressing scheme is used. Internally, this chip is organized as a 2048 × 2048 matrix of 1-bit cells, which gives 4 Mbits. To address the chip, first a row is selected by putting its 11-bit number on the address pins. Then the $\overline{\text{RAS}}$ (Row Address Strobe) is asserted. After that, a column number is put on the address pins and $\overline{\text{CAS}}$ (Column Address Strobe) is asserted. The chip responds by accepting or outputting one data bit.

Large memory chips are often constructed as $n \times n$ matrices that are addressed by row and column. This organization reduces the number of pins required but also makes addressing the chip slower, since two addressing cycles are needed, one for the row and one for the column. To win back some of the speed lost by this design, some memory chips can be given a row address followed by a sequence of column addresses to access consecutive bits in a row.

Years ago, the largest memory chips were often organized like Fig. 3-30(b). As memory words have grown from 8 bits to 32 bits and beyond, 1-bit-wide chips began to be inconvenient. To build a memory with a 32-bit word from 4096K × 1 chips requires 32 chips in parallel. These 32 chips have a total capacity of at least 16 MB, whereas using 512K × 8 chips requires only four chips in parallel and allows memories as small as 2 MB. To avoid having 32 chips for memory, most chip manufacturers now have chip families with 4-, 8-, and 16-bit widths. And the situation with 64-bit words is even worse, of course.

Two examples of 512-Mbit chips are given in Fig. 3-31. These chips have four internal memory banks of 128 Mbit each, requiring two bank select lines to choose a bank. The design of Fig. 3-31(a) is a 32M × 16 design, with 13 lines for the $\overline{\text{RAS}}$ signal, 10 lines for the $\overline{\text{CAS}}$ signal, and 2 lines for the bank select. Together, these 25 signals allow each of the 2^{25} internal 16-bit cells to be addressed. In contrast, Fig. 3-31(b) is a 128M × 4 design, with 13 lines for the $\overline{\text{RAS}}$ signal, 12 lines for the $\overline{\text{CAS}}$ signal, and 2 lines for the bank select. Here, 27 signals can select any of the 2^{27} internal 4-bit cells to be addressed. The decision about how many rows and how many columns a chip has is made for engineering reasons. The matrix need not be square.

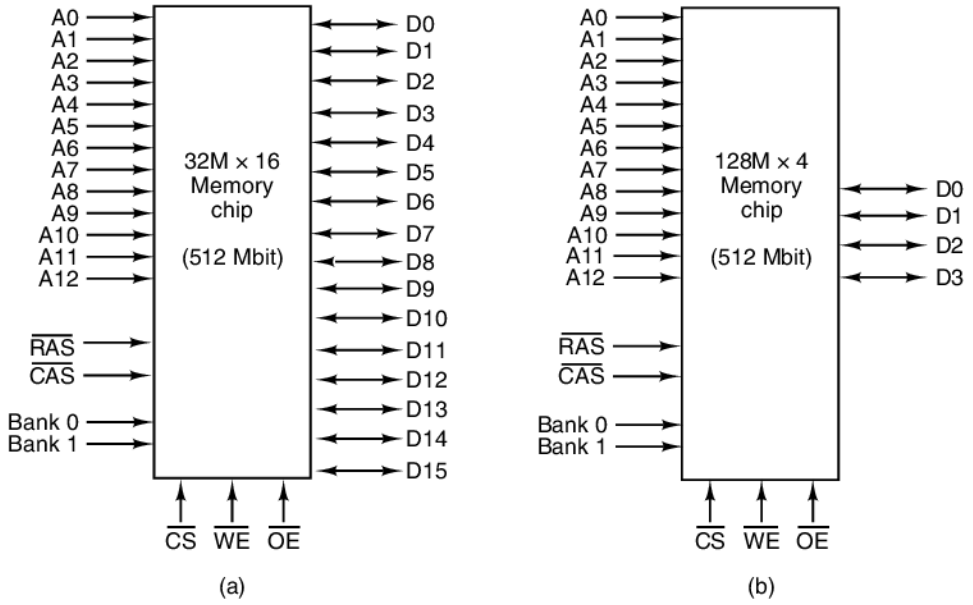


Figure 3-31. Two ways of organizing a 512-Mbit memory chip.

These examples demonstrate two separate and independent issues for memory-chip design. First is the output width (in bits): does the chip deliver 1, 4, 8, 16, or some other number of bits at once? Second, are all the address bits presented on separate pins at once or are the rows and columns presented sequentially as in the examples of Fig. 3-31? A memory-chip designer has to answer both questions before starting the chip design.

3.3.6 RAMs and ROMs

The memories we have studied so far can all be read and written. Such memories are called **RAM**s (Random Access Memories), which is a misnomer because all memory chips are randomly accessible, but the term is too well established to

get rid of now. RAMs come in two varieties, static and dynamic. **Static RAM**s (**SRAM**s) are constructed internally using circuits similar to our basic D flip-flop. These memories have the property that their contents are retained as long as the power is kept on: seconds, minutes, hours, even days. Static RAMs are very fast. A typical access time is on the order of a nanosecond or less. For this reason, static RAMS are popular as cache memory.

Dynamic RAMs (**DRAM**s), in contrast, do not use flip-flops. Instead, a dynamic RAM is an array of cells, each cell containing one transistor and a tiny capacitor. The capacitors can be charged or discharged, allowing 0s and 1s to be stored. Because the electric charge tends to leak out, each bit in a dynamic RAM must be **refreshed** (reloaded) every few milliseconds to prevent the data from leaking away. Because external logic must take care of the refreshing, dynamic RAMs require more complex interfacing than static ones, although in many applications this disadvantage is compensated for by their larger capacities.

Since dynamic RAMs need only one transistor and one capacitor per bit (vs. six transistors per bit for the best static RAM), dynamic RAMs have a very high density (many bits per chip). For this reason, main memories are nearly always built out of dynamic RAMs. However, this large capacity has a price: dynamic RAMs are slow (tens of nanoseconds). Thus, the combination of a static RAM cache and a dynamic RAM main memory attempts to combine the good properties of each.

Several types of dynamic RAM chips exist. The oldest type still around (in elderly computers) is **FPM** (**Fast Page Mode**) DRAM. Internally it is organized as a matrix of bits and it works by having the hardware present a row address and then step through the column addresses, as we described with $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ in the context of Fig. 3-30. Explicit signals tell the memory when it is time to respond, so the memory runs asynchronously from the main system clock.

FPM DRAM was replaced with **EDO** (**Extended Data Output**) DRAM, which allows a second memory reference to begin before the previous memory reference has been completed. This simple pipelining did not make a single memory reference go faster but did improve the memory bandwidth, giving more words per second.

FPM and EDO worked reasonably well when memory chips had cycle times of 12 nsec and slower. When processors got so fast that faster memories were really needed, FPM and EDO were replaced by **SDRAM** (**Synchronous DRAM**), which is a hybrid of static and dynamic RAM and is driven by the main system clock. The big advantage of SDRAM is that the clock eliminates the need for control signals to tell the memory chip when to respond. Instead, the CPU tells the memory how many cycles it should run, then starts it. On each subsequent cycle, the memory outputs 4, 8, or 16 bits, depending on how many output lines it has. Eliminating the need for control signals increases the data rate between CPU and memory.

The next improvement over SDRAM was **DDR** (**Double Data Rate**) SDRAM. With this kind of memory, the memory chip produces output on both the rising

edge of the clock and the falling edge, doubling the data rate. Thus, an 8-bit-wide DDR chip running at 200 MHz outputs two 8-bit values 200 million times a second (for a short interval, of course), giving a theoretical burst rate of 3.2 Gbps. The DDR2 and DDR3 memory interfaces provide additional performance over DDR by increasing the memory-bus speeds to 533 MHz and 1067 MHz, respectively. At the time this book went to press, the fastest DDR3 chips could output data at 17.067 GB/sec.

Nonvolatile Memory Chips

RAMs are not the only kind of memory chips. In many applications, such as toys, appliances, and cars, the program and some of the data must remain stored even when the power is turned off. Furthermore, once installed, neither the program nor the data are ever changed. These requirements have led to the development of **ROM**s (Read-Only Memories), which cannot be changed or erased, intentionally or otherwise. The data in a ROM are inserted during its manufacture, essentially by exposing a photosensitive material through a mask containing the desired bit pattern and then etching away the exposed (or unexposed) surface. The only way to change the program in a ROM is to replace the entire chip.

ROMs are much cheaper than RAMs when ordered in large enough volumes to defray the cost of making the mask. However, they are inflexible, because they cannot be changed after manufacture, and the turnaround time between placing an order and receiving the ROMs may be weeks. To make it easier for companies to develop new ROM-based products, the **PROM** (Programmable ROM) was invented. A PROM is like a ROM, except that it can be programmed (once) in the field, eliminating the turnaround time. Many PROMs contain an array of tiny fuses inside. A specific fuse can be blown out by selecting its row and column and then applying a high voltage to a special pin on the chip.

The next development in this line was the **EPROM** (Erasable PROM), which can be not only field-programmed but also field-erased. When the quartz window in an EPROM is exposed to a strong ultraviolet light for 15 minutes, all the bits are set to 1. If many changes are expected during the design cycle, EPROMs are far more economical than PROMs because they can be reused. EPROMs usually have the same organization as static RAMs. The 4-Mbit 27C040 EPROM, for example, uses the organization of Fig. 3-31(a), which is typical of a static RAM. What is interesting is that ancient chips like this one do not die off. They just become cheaper and find their way into lower-end products that are highly cost sensitive. A 27C040 can now be bought retail for under $3 and much less in large volumes.

Even better than the EPROM is the **EEPROM** which can be erased by applying pulses to it instead of putting it in a special chamber for exposure to ultraviolet light. In addition, an EEPROM can be reprogrammed in place, whereas an EPROM has to be inserted in a special EPROM programming device to be programmed. On the minus side, the biggest EEPROMs are typically only 1/64 as

large as common EPROMs and they are only half as fast. EEPROMs cannot compete with DRAMs or SRAMs because they are 10 times slower, 100 times smaller in capacity, and much more expensive. They are used only in situations where their nonvolatility is crucial.

A more recent kind of EEPROM is **flash memory**. Unlike EPROM, which is erased by exposure to ultraviolet light, and EEPROM, which is byte erasable, flash memory is block erasable and rewritable. Like EEPROM, flash memory can be erased without removing it from the circuit. Various manufacturers produce small printed-circuit cards with up to 64 GB of flash memory on them for use as "film" for storing pictures in digital cameras and many other purposes. As we discussed in Chap. 2, flash memory is now starting to replace mechanical disks. As a disk, flash memory provides faster access times at lower power, but with a much higher cost per bit. A summary of the various kinds of memory is given in Fig. 3-32.

Type	Category	Erasure	Byte alterable	Volatile	Typical use
SRAM	Read/write	Electrical	Yes	Yes	Level 2 cache
DRAM	Read/write	Electrical	Yes	Yes	Main memory (old)
SDRAM	Read/write	Electrical	Yes	Yes	Main memory (new)
ROM	Read-only	Not possible	No	No	Large-volume appliances
PROM	Read-only	Not possible	No	No	Small-volume equipment
EPROM	Read-mostly	UV light	No	No	Device prototyping
EEPROM	Read-mostly	Electrical	Yes	No	Device prototyping
Flash	Read/write	Electrical	No	No	Film for digital camera

Figure 3-32. A comparison of various memory types.

Field-Programmable Gate Arrays

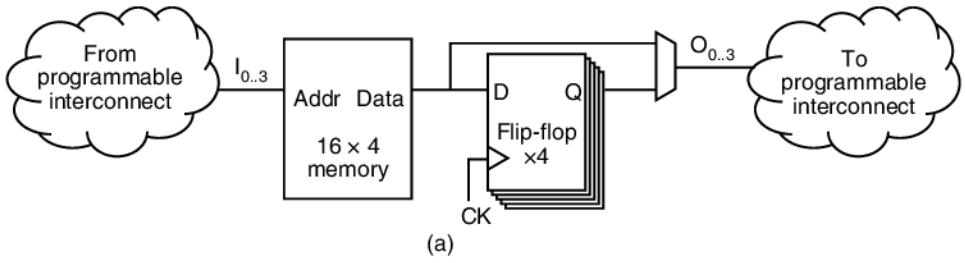
As we saw in Chap. 1, **field-programmable gate arrays** (**FPGA**s) are chips which contain programmable logic such that we can form arbitrary logic circuit by simply loading the FPGA with appropriate configuration data. The main advantage of FPGAs is that new hardware circuits can be built in hours, rather than the months it takes to fabricate ICs. Integrated circuits are not going the way of the dodo, however, as they still hold a significant cost advantage over FPGAs for high-volume applications, and they run faster and use much less power. Because of their design-time advantages, however, FPGAs are often used for design prototyping and low-volume applications.

Let's now look inside an FPGA and understand how it can be used to implement a wide range of logic circuits. The FPGA chip contains two primary components that are replicated many times: **LUT**s (**LookUp Tables**) and **programmable interconnects**. Let us now examine how they are used.

A LUT, shown in Fig. 3-33(a), is a small programmable memory that produces a signal output optionally to a register, which is then output to the programmable interconnect. The programmable memory is used to create an arbitrary logic function. The LUT in the figure has a 16×4 memory, which can emulate any logic circuit with 4 bits of input and 4 bits of output. Programming the LUT requires loading the memory with the appropriate responses of the combinational logic being emulated. In other words, if the combinational logic produces the value Y when given the input X, the value Y would be written into the LUT at index X.

The example design in Fig. 3-32(b) shows how a single 4-input LUT could implement a 3-bit counter with reset. The example counter continually counts up by adding one (modulo 4) to the current value of the counter, unless the reset signal *CLR* is asserted, in which case the counter resets its value to zero.

To implement the example counter, the upper four entries of the LUT are all zero. These entries output the value zero when the counter is reset. Thus, the most significant bit of the LUT input (I_3) represents the reset input (*CLR*) which is asserted with a logic 1. For the remaining LUT entries, the value at index $I_{0..3}$ of the LUT contains the value $(I + 1)$ modulo 4. To complete the design, the output signal $O_{0..3}$ must be connected, using the programmable interconnect to the internal input signal $I_{0..3}$.



(a)

Signal Assignment

FPGA	Counter
I_3	CLR
$O_{2..0}$	$O_{2..0}$
CK	CK

Addr	Data
0	1
1	2
2	3
3	0

Addr	Data
4	0
5	0
6	0
7	0

(b)

Figure 3-33. (a) A field-programmable logic array lookup table (LUT). (b) The LUT configuration to create a 3-bit clearable counter.

To better understand the FPGA-based counter with reset, let's consider its operation. If, for example, the current state of the counter is 2 and the reset (*CLR*) signal is not asserted, the input address to the LUT will be 2, which will produce an output to the flip-flops of 3. If the reset signal (*CLR*) were asserted for the same state, the input to the LUT would be 6, which would produce the next state of 0.

All in all, this may seem like an arcane way to build a counter with reset, and in fact a fully custom design with an incrementer circuit and reset signals to the flip-flops would be smaller, faster, and use less power. The main advantage of the FPGA-based design is that you can craft it in an hour at home, whereas the more efficient fully custom design must be fabricated from silicon, which could take a month or more.

To use an FPGA, the design must be described using a circuit description or a hardware description language (i.e., a programming language used to describe hardware structures). The design is then processed by a synthesizer, which maps the circuit to a specific FPGA architecture. One challenge of using FPGAs is that the design you want to map never seems to fit. FPGAs are manufactured with varying number of LUTs, with larger quantities costing more. In general, if your design does not fit, you need to simplify or throw away some functionality, or purchase a larger (and more expensive) FPGA. Very large designs may not fit into the largest FPGAs, which will require the designer to map the design into multiple FPGAs; this task is definitely more difficult, but still a walk in the park compared to designing a complete custom integrated circuit.

3.4 CPU CHIPS AND BUSES

Armed with information about integrated circuits, clocks, and memory chips, we can now start to put all the pieces together to look at complete systems. In this section, we will first look at some general aspects of CPUs as viewed from the digital logic level, including **pinout** (what the signals on the various pins mean). Since CPUs are so closely intertwined with the design of the buses they use, we will also provide an introduction to bus design in this section. In subsequent sections we will give detailed examples of both CPUs and their buses and how they are interfaced.

3.4.1 CPU Chips

All modern CPUs are contained on a single chip. This makes their interaction with the rest of the system well defined. Each CPU chip has a set of pins, through which all its communication with the outside world must take place. Some pins output signals from the CPU to the outside world; others accept signals from the outside world; some can do both. By understanding the function of all the pins, we can learn how the CPU interacts with the memory and I/O devices at the digital logic level.

The pins on a CPU chip can be divided into three types: address, data, and control. These pins are connected to similar pins on the memory and I/O chips via a collection of parallel wires called a bus. To fetch an instruction, the CPU first puts the memory address of that instruction on its address pins. Then it asserts one or

more control lines to inform the memory that it wants to read (for example) a word. The memory replies by putting the requested word on the CPU's data pins and asserting a signal saying that it is done. When the CPU sees this signal, it accepts the word and carries out the instruction.

The instruction may require reading or writing data words, in which case the whole process is repeated for each additional word. We will go into the detail of how reading and writing works below. For the time being, the important thing to understand is that the CPU communicates with the memory and I/O devices by presenting signals on its pins and accepting signals on its pins. No other communication is possible.

Two of the key parameters that determine the performance of a CPU are the number of address pins and the number of data pins. A chip with m address pins can address up to 2^m memory locations. Common values of m are 16, 32, and 64. Similarly, a chip with n data pins can read or write an n-bit word in a single operation. Common values of n are 8, 32, and 64. A CPU with 8 data pins will take four operations to read a 32-bit word, whereas one with 32 data pins can do the same job in one operation. Thus, the chip with 32 data pins is much faster but is invariably more expensive as well.

In addition to address and data pins, each CPU has some control pins. They regulate the flow and timing of data to and from the CPU and have other miscellaneous uses. All CPUs have pins for power (usually +1.2 to +1.5 volts), ground, and a clock signal (a square wave at some well-defined frequency), but the other pins vary greatly from chip to chip. Nevertheless, the control pins can be roughly grouped into the following major categories:

1. Bus control.
2. Interrupts.
3. Bus arbitration.
4. Coprocessor signaling.
5. Status.
6. Miscellaneous.

We will briefly describe each of these categories below. When we look at the Intel Core i7, TI OMAP4430, and Atmel ATmega168 chips later, we will provide more detail. A generic CPU chip using these signal groups is shown in Fig. 3-34.

The bus control pins are mostly outputs from the CPU to the bus (thus inputs to the memory and I/O chips) telling whether the CPU wants to read or write memory or do something else. The CPU uses these pins to control the rest of the system and tell it what it wants to do.

The interrupt pins are inputs from I/O devices to the CPU. In most systems, the CPU can tell an I/O device to start an operation and then go off and do some

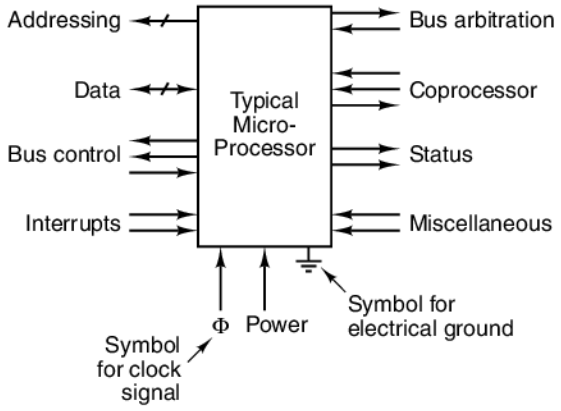


Figure 3-34. The logical pinout of a generic CPU. The arrows indicate input signals and output signals. The short diagonal lines indicate that multiple pins are used. For a specific CPU, a number will be given to tell how many.

other activity, while the I/O device is doing its work. When the I/O has been completed, the I/O controller chip asserts a signal on one of these pins to interrupt the CPU and have it service the I/O device, for example to check whether if I/O errors occurred. Some CPUs have an output pin to acknowledge the interrupt signal.

The bus arbitration pins are needed to regulate traffic on the bus, in order to prevent two devices from trying to use it at the same time. For arbitration purposes, the CPU counts as a device and has to request the bus like any other device.

Some CPU chips are designed to operate with coprocessors such as floating-point chips, but sometimes graphics or other chips as well. To facilitate communication between CPU and coprocessor, special pins are provided for making and granting various requests.

In addition to these signals, there are various miscellaneous pins that some CPUs have. Some of these provide or accept status information, others are useful for debugging or resetting the computer, and still others are present to assure compatibility with older I/O chips.

3.4.2 Computer Buses

A **bus** is a common electrical pathway between multiple devices. Buses can be categorized by their function. They can be used internal to the CPU to transport data to and from the ALU, or external to the CPU to connect it to memory or to I/O devices. Each type of bus has its own requirements and properties. In this section and the following ones, we will focus on buses that connect the CPU to the memory and I/O devices. In the next chapter we will examine more closely the buses inside the CPU.

Early personal computers had a single external bus or **system bus**. It consisted of 50 to 100 parallel copper wires etched onto the motherboard, with connectors spaced at regular intervals for plugging in memory and I/O boards. Modern personal computers generally have a special-purpose bus between the CPU and memory and (at least) one other bus for the I/O devices. A minimal system, with one memory bus and one I/O bus, is illustrated in Fig. 3-35.

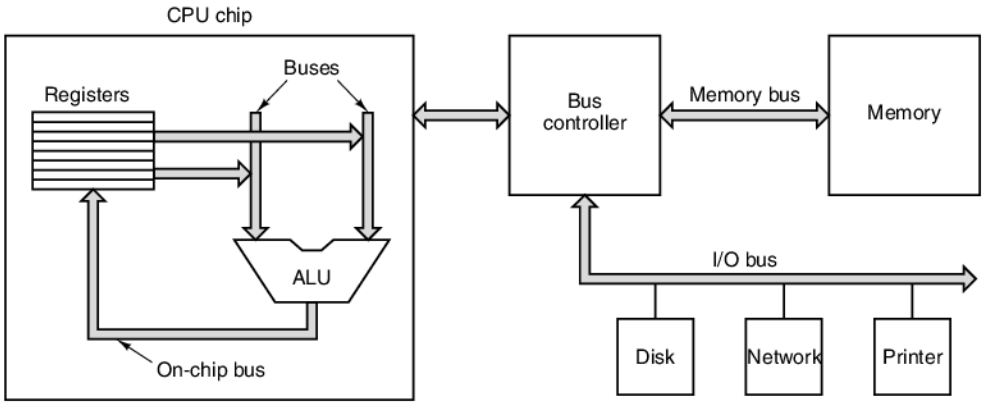


Figure 3-35. A computer system with multiple buses.

In the literature, buses are sometimes drawn as "fat" arrows, as in this figure. The distinction between a fat arrow and a single line with a diagonal line through it and a bit count next to it is subtle. When all the bits are of the same type, say, all address bits or all data bits, then the short-diagonal-line approach is commonly used. When address, data, and control lines are involved, a fat arrow is more common.

While the designers of the CPU are free to use any kind of bus they want inside the chip, in order to make it possible for boards designed by third parties to attach to the system bus, there must be well-defined rules about how the external bus works, which all devices attached to it must obey. These rules are called the **bus protocol**. In addition, there must be mechanical and electrical specifications, so that third-party boards will fit in the card cage and have connectors that match those on the motherboard mechanically and in terms of voltages, timing, etc. Still other buses do not have mechanical specifications because they are designed to be used only within an integrated circuit, for example, to connect components together within a system-on-a-chip (SoC).

A number of buses are in widespread use in the computer world. A few of the better-known ones, historical and current (with examples), are the Omnibus (PDP-8), Unibus (PDP-11), Multibus (8086), VME bus (physics lab equipment), IBM PC bus (PC/XT), ISA bus (PC/AT), EISA bus (80386), Microchannel (PS/2), Nubus (Macintosh), PCI bus (many PCs), SCSI bus (many PCs and workstations),

Universal Serial Bus (modern PCs), and FireWire (consumer electronics). The world would probably be a better place if all but one would suddenly vanish from the face of the earth (well, all right, how about all but two?). Unfortunately, standardization in this area seems very unlikely, as there is already too much invested in all these incompatible systems.

As an aside, there is another interconnect, PCI Express, that is widely referred to as a bus but is not a bus at all. We will study it later in this chapter.

Let us now begin our study of how buses work. Some devices that attach to a bus are active and can initiate bus transfers, whereas others are passive and wait for requests. The active ones are called **masters**; the passive ones are called **slaves**. When the CPU orders a disk controller to read or write a block, the CPU is acting as a master and the disk controller is acting as a slave. However, later on, the disk controller may act as a master when it commands the memory to accept the words it is reading from the disk drive. Several typical combinations of master and slave are listed in Fig. 3-36. Under no circumstances can memory ever be a master.

Master	Slave	Example
CPU	Memory	Fetching instructions and data
CPU	I/O device	Initiating data transfer
CPU	Coprocessor	CPU handing instruction off to coprocessor
I/O device	Memory	DMA (Direct Memory Access)
Coprocessor	CPU	Coprocessor fetching operands from CPU

Figure 3-36. Examples of bus masters and slaves.

The binary signals that computer devices output are frequently too weak to power a bus, especially if it is relatively long or has many devices on it. For this reason, most bus masters are connected to the bus by circuitry called a **bus driver**, which is essentially a digital amplifier. Similarly, most slaves are connected to the bus by a **bus receiver**. For devices that can act as both master and slave, a combined circuit called a **bus transceiver** is used. These bus interfaces are often tri-state devices, to allow them to float (disconnect) when they are not needed, or are hooked up in a somewhat different way, called **open collector**, that achieves a similar effect. When two or more devices on an open-collector line assert the line at the same time, the result is the Boolean OR of all the signals. This arrangement is often called **wired-OR**. On most buses, some of the lines are tri-state and others, which need the wired-OR property, are open collector.

Like a CPU, a bus also has address, data, and control lines. However, there is not necessarily a one-to-one mapping between the CPU pins and the bus signals. For example, some CPUs have three pins that encode whether the CPU is doing a memory read, memory write, I/O read, I/O write, or some other operation. A typical bus might have one line for memory read, a second for memory write, a third for I/O read, a fourth for I/O write, and so on. A decoder circuit would then be

needed between the CPU and such a bus to match the two sides up, that is, to convert the 3-bit encoded signal into separate signals that can drive the bus lines.

Bus design and operation are sufficiently complex subjects that a number of entire books have been written about them (Anderson et al., 2004, Solari and Willse, 2004). The principal bus design issues are bus width, bus clocking, bus arbitration, and bus operations. Each of these issues has a substantial impact on the speed and bandwidth of the bus. We will now examine each of these in the next four sections.

3.4.3 Bus Width

Bus width is the most obvious design parameter. The more address lines a bus has, the more memory the CPU can address directly. If a bus has n address lines, then a CPU can use it to address 2^n different memory locations. To allow large memories, buses need many address lines. That sounds simple enough.

The problem is that wide buses need more wires than narrow ones. They also take up more physical space (e.g., on the motherboard) and need bigger connectors. All of these factors make the bus more expensive. Thus, there is a trade-off between maximum memory size and system cost. A system with a 64-line address bus and 2^{32} bytes of memory will cost more than one with 32 address lines and the same 2^{32} bytes of memory. The possibility of expansion later is not free.

The result of this observation is that many system designers tend to be short-sighted, with unfortunate consequences later. The original IBM PC contained an 8088 CPU and a 20-bit address bus, as shown in Fig. 3-37(a). Having 20 bits allowed the PC to address 1 MB of memory.

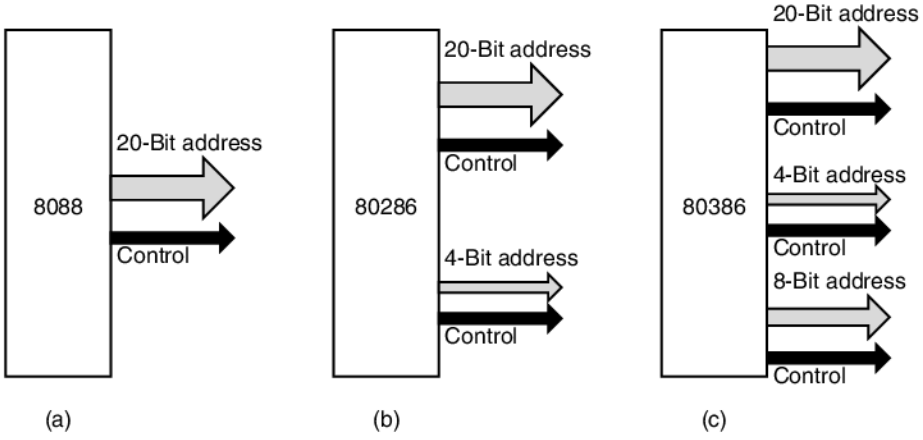


Figure 3-37. Growth of an address bus over time.

When the next CPU chip (the 80286) came out, Intel flet it had to increase the address space to 16 MB, so four more bus lines were added (without disturbing the

original 20, for reasons of backward compatibility), as illustrated in Fig. 3-37(b). Unfortunately, more control lines had to be added to deal with the new address lines. When the 80386 came out, another eight address lines were added, along with still more control lines, as shown in Fig. 3-37(c). The resulting design (the EISA bus) is much messier than it would have been had the bus been given 32 lines at the start.

Not only does the number of address lines tend to grow over time, but so does the number of data lines, albeit for a different reason. There are two ways to increase the bandwidth of a bus: decrease the bus cycle time (more transfers/sec) or increase the data bus width (more bits/transfer). Speeding the bus up is possible (but difficult) because the signals on different lines travel at slightly different speeds, a problem known as **bus skew**. The faster the bus, the more the skew.

Another problem with speeding up the bus is it will not be backward compatible. Old boards designed for the slower bus will not work with the new one. Invalidating old boards makes both the owners and manufacturers of the old boards unhappy. Therefore the usual approach to improving performance is to add more data lines, analogous to Fig. 3-37. As you might expect, however, this incremental growth does not lead to a clean design in the end. The IBM PC and its successors, for example, went from 8 data lines to 16 and then 32 on essentially the same bus.

To get around the problem of very wide buses, sometimes designers opt for a **multiplexed bus**. In this design, instead of the address and data lines being separate, there are, say, 32 lines for address and data together. At the start of a bus operation, the lines are used for the address. Later on, they are used for data. For a write to memory, for example, this means that the address lines must be set up and propagated to the memory before the data can be put on the bus. With separate lines, the address and data can be put on together. Multiplexing the lines reduces bus width (and cost) but results in a slower system. Bus designers have to carefully weigh all these options when making choices.

3.4.4 Bus Clocking

Buses can be divided into two distinct categories depending on their clocking. A **synchronous bus** has a line driven by a crystal oscillator. The signal on this line consists of a square wave with a frequency generally between 5 and 133 MHz. All bus activities take an integral number of these cycles, called **bus cycles**. The other kind of bus, the **asynchronous bus**, does not have a master clock. Bus cycles can be of any length required and need not be the same between all pairs of devices. Below we will examine each bus type.

Synchronous Buses

As an example of how a synchronous bus works, consider the timing of Fig. 3-38(a). In this example, we will use a 100-MHz clock, which gives a bus cycle of 10 nsec. While this may seem a bit slow compared to CPU speeds of 3

GHz and more, few existing PC buses are much faster. For example, the popular PCI bus usually runs at either 33 or 66 MHz, and the upgraded (but now defunct) PCI-X bus ran at a speed of up to 133 MHz. The reasons current buses are slow were given above: technical design problems such as bus skew and the need for backward compatibility.

In our example, we will further assume that reading from memory takes 15 nsec from the time the address is stable. As we will see shortly, with these parameters, it will take three bus cycles to read a word. The first cycle starts at the rising edge of T_1 and the third one ends at the rising edge of T_4, as shown in the figure. Note that none of the rising or falling edges has been drawn vertically, because no electrical signal can change its value in zero time. In this example we will assume that it takes 1 nsec for a signal to change. The clock, ADDRESS, DATA, $\overline{\text{MREQ}}$, $\overline{\text{RD}}$, and $\overline{\text{WAIT}}$ lines are all shown on the same time scale.

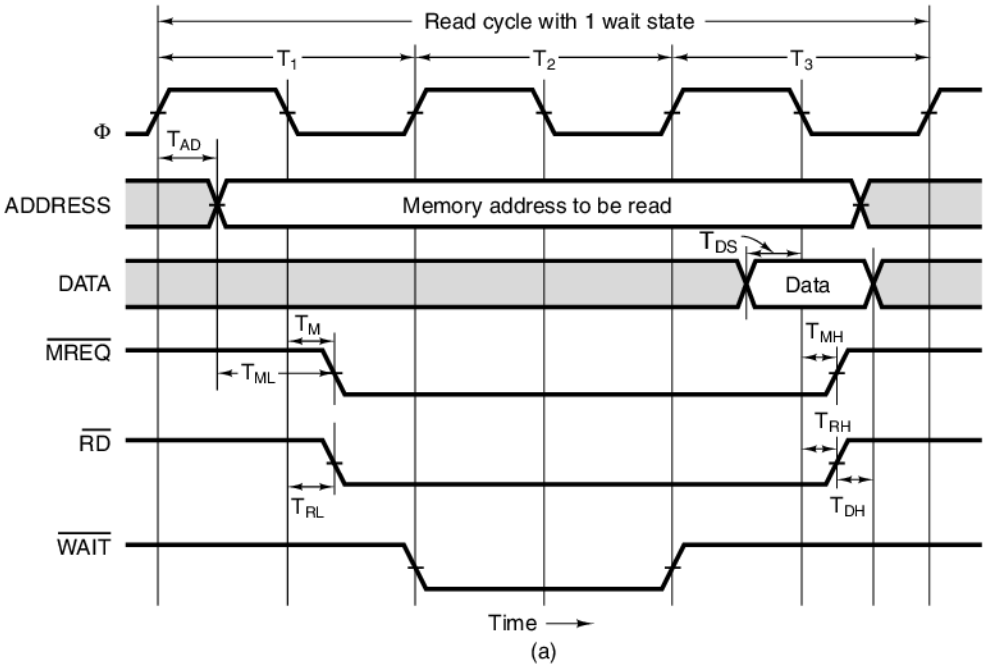
The start of T_1 is defined by the rising edge of the clock. Partway through T_1 the CPU puts the address of the word it wants on the address lines. Because the address is not a single value, like the clock, we cannot show it as a single line in the figure; instead, it is shown as two lines, with a crossing at the time that the address changes. Furthermore, the shading prior to the crossing indicates that the shaded value is not important. Using the same shading convention, we see that the contents of the data lines are not significant until well into T_3.

After the address lines have had a chance to settle down to their new values, $\overline{\text{MREQ}}$ and $\overline{\text{RD}}$ are asserted. The former indicates that memory (as opposed to an I/O device) is being accessed, and the latter is asserted for reads and negated for writes. Since the memory takes 15 nsec after the address is stable (partway into the first clock cycle), it cannot provide the requested data during T_2. To tell the CPU not to expect it, the memory asserts the $\overline{\text{WAIT}}$ line at the start of T_2. This action will insert **wait states** (extra bus cycles) until the memory is finished and negates $\overline{\text{WAIT}}$. In our example, one wait state (T_2) has been inserted because the memory is too slow. At the start of T_3, when it is sure it will have the data during the current cycle, the memory negates $\overline{\text{WAIT}}$.

During the first half of T_3, the memory puts the data onto the data lines. At the falling edge of T_3 the CPU strobes (i.e., reads) the data lines, latching (i.e., storing) the value in an internal register. Having read the data, the CPU negates $\overline{\text{MREQ}}$ and $\overline{\text{RD}}$. If need be, another memory cycle can begin at the next rising edge of the clock. This sequence can be repeatedly indefinitely.

In the timing specification of Fig. 3-38(b), eight symbols that occur in the timing diagram are further clarified. T_{AD}, for example, is the time interval between the rising edge of the T_1 clock and the address lines being set. According to the timing specification, $T_{\text{AD}} \leq 4$ nsec. This means that the CPU manufacturer guarantees that during any read cycle, the CPU will output the address to be read within 4 nsec of the midpoint of the rising edge of T_1.

The timing specifications also require that the data be available on the data lines at least T_{DS} (2 nsec) before the falling edge of T_3, to give it time to settle



Symbol	Parameter	Min	Max	Unit
T_{AD}	Address output delay		4	nsec
T_{ML}	Address stable prior to $\overline{\text{MREQ}}$	2		nsec
T_M	$\overline{\text{MREQ}}$ delay from falling edge of Φ in T_1		3	nsec
T_{RL}	RD delay from falling edge of Φ in T_1		3	nsec
T_{DS}	Data setup time prior to falling edge of Φ	2		nsec
T_{MH}	$\overline{\text{MREQ}}$ delay from falling edge of Φ in T_3		3	nsec
T_{RH}	$\overline{\text{RD}}$ delay from falling edge of Φ in T_3		3	nsec
T_{DH}	Data hold time from negation of $\overline{\text{RD}}$	0		nsec

(b)

Figure 3-38. (a) Read timing on a synchronous bus. (b) Specification of some critical times.

down before the CPU strobes it in. The combination of constraints on T_{AD} and T_{DS} means that, in the worst case, the memory will have only $25 - 4 - 2 = 19$ nsec from the time the address appears until it must produce the data. Because 10 nsec is enough, even in the worst case, a 10-nsec memory can always respond during T_3. A 20-nsec memory, however, would just miss and have to insert a second wait state and respond during T_4.

The timing specification further guarantees that the address will be set up at least 2 nsec prior to $\overline{\text{MREQ}}$ being asserted. This time can be important if $\overline{\text{MREQ}}$ drives chip select on the memory chip because some memories require an address setup time prior to chip select. Clearly, the system designer should not choose a memory chip that needs a 3-nsec setup time.

The constraints on T_M and T_{RL} mean that $\overline{\text{MREQ}}$ and $\overline{\text{RD}}$ will both be asserted within 3 nsec from the T_1 falling clock. In the worst case, the memory chip will have only 10 + 10 − 3 − 2 = 15 nsec after the assertion of $\overline{\text{MREQ}}$ and $\overline{\text{RD}}$ to get its data onto the bus. This constraint is in addition to (and independent of) the 15-nsec interval needed after the address is stable.

T_{MH} and T_{RH} tell how long it takes $\overline{\text{MREQ}}$ and $\overline{\text{RD}}$ to be negated after the data have been strobed in. Finally, T_{DH} tells how long the memory must hold the data on the bus after $\overline{\text{RD}}$ has been negated. As far as our example CPU is concerned, the memory can remove the data from the bus as soon as $\overline{\text{RD}}$ has been negated. On some actual CPUs, however, the data must be kept stable a little longer.

We would like to point out that Fig. 3-38 is a highly simplified version of real timing constraints. In reality, many more critical times are always specified. Nevertheless, it gives a good flavor for how a synchronous bus works.

A last point worth making is that control signals can be asserted high or low. It is up to the bus designers to determine which is more convenient, but the choice is essentially arbitrary. One can regard it as the hardware equivalent of a programmer's choice to represent free disk blocks in a bit map as 0s vs. 1s.

Asynchronous Buses

Although synchronous buses are easy to work with due to their discrete time intervals, they also have some problems. For example, everything works in multiples of the bus clock. If a CPU and memory are able to complete a transfer in 3.1 cycles, they have to stretch it to 4.0 because fractional cycles are forbidden.

Worse yet, once a bus cycle has been chosen, and memory and I/O cards have been built for it, it is difficult to take advantage of future improvements in technology. For example, suppose a few years after the system of Fig. 3-38 was built, new memories became available with access times of 8 nsec instead of 15 nsec. These would get rid of the wait state, speeding up the machine. Then suppose 4-nsec memories became available. There would be no further gain in performance because the minimum time for a read is two cycles with this design.

Putting this in slightly different terms, if a synchronous bus has a heterogeneous collection of devices, some fast and some slow, the bus has to be geared to the slowest one and the fast ones cannot use their full potential.

Mixed technology can be handled by going to an asynchronous bus, that is, one with no master clock, as shown in Fig. 3-39. Instead of tying everything to the clock, when the bus master has asserted the address, $\overline{\text{MREQ}}$, $\overline{\text{RD}}$, and anything else

it needs to, it then asserts a special signal that we will call $\overline{\text{MSYN}}$ (Master SYNchronization). When the slave sees this, it performs the work as fast as it can. When it is done, it asserts $\overline{\text{SSYN}}$ (Slave SYNchronization).

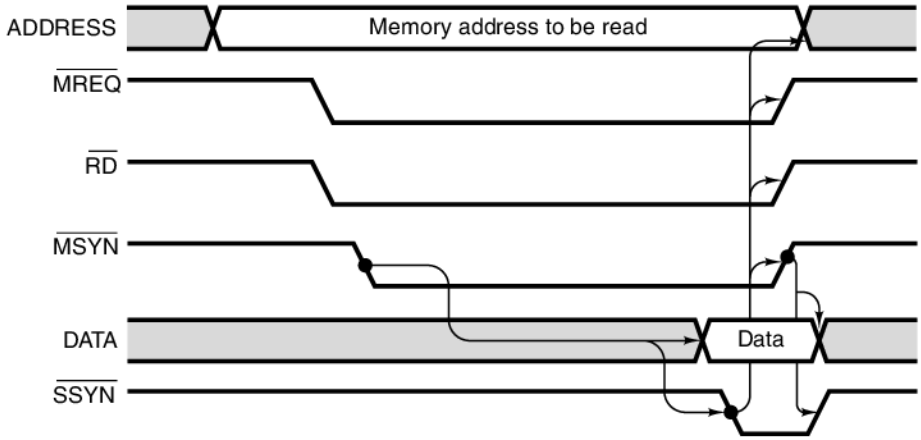


Figure 3-39. Operation of an asynchronous bus.

As soon as the master sees $\overline{\text{SSYN}}$ asserted, it knows that the data are available, so it latches them and then negates the address lines, along with $\overline{\text{MREQ}}$, $\overline{\text{RD}}$, and $\overline{\text{MSYN}}$. When the slave sees the negation of $\overline{\text{MSYN}}$, it knows that the cycle has been completed, so it negates $\overline{\text{SSYN}}$, and we are back in the original situation, with all signals negated, waiting for the next master.

Timing diagrams of asynchronous buses (and sometimes synchronous buses as well) use arrows to show cause and effect, as in Fig. 3-39. The assertion of $\overline{\text{MSYN}}$ causes the data lines to be asserted and also causes the slave to assert $\overline{\text{SSYN}}$. The assertion of $\overline{\text{SSYN}}$, in turn, causes the negation of the address lines, $\overline{\text{MREQ}}$, $\overline{\text{RD}}$, and $\overline{\text{MSYN}}$. Finally, the negation of $\overline{\text{MSYN}}$ causes the negation of $\overline{\text{SSYN}}$, which ends the read and returns the system to its original state.

A set of signals that interlocks this way is called a **full handshake**. The essential part consists of four events:

1. $\overline{\text{MSYN}}$ is asserted.
2. $\overline{\text{SSYN}}$ is asserted in response to $\overline{\text{MSYN}}$.
3. $\overline{\text{MSYN}}$ is negated in response to $\overline{\text{SSYN}}$.
4. $\overline{\text{SSYN}}$ is negated in response to the negation of $\overline{\text{MSYN}}$.

It should be clear that full handshakes are timing independent. Each event is caused by a prior event, not by a clock pulse. If a particular master/slave pair is slow, that in no way affects a subsequent master/slave pair that is much faster.

The advantage of an asynchronous bus should now be clear, but the fact is that most buses are synchronous. The reason is that it is easier to build a synchronous system. The CPU just asserts its signals, and the memory just reacts. There is no feedback (cause and effect), but if the components have been chosen properly, everything will work without handshaking. Also, there is a lot of investment in synchronous bus technology.

3.4.5 Bus Arbitration

Up until now, we have tacitly assumed that there is only one bus master, the CPU. In reality, I/O chips have to become bus master to read and write memory, and also to cause interrupts. Coprocessors may also need to become bus master. The question then arises: "What happens if two or more devices all want to become bus master at the same time?" The answer is that some **bus arbitration** mechanism is needed to prevent chaos.

Arbitration mechanisms can be centralized or decentralized. Let us first consider centralized arbitration. One particularly simple form of this is shown in Fig. 3-40(a). In this scheme, a single bus arbiter determines who goes next. Many CPUs have the arbiter built into the CPU chip, but sometimes a separate chip is needed. The bus contains a single wired-OR request line that can be asserted by one or more devices at any time. There is no way for the arbiter to tell how many devices have requested the bus. The only categories it can distinguish are some requests and no requests.

When the arbiter sees a bus request, it issues a grant by asserting the bus grant line. This line is wired through all the I/O devices in series, like a cheap string of Christmas tree lamps. When the device physically closest to the arbiter sees the grant, it checks to see if it has made a request. If so, it takes over the bus but does not propagate the grant further down the line. If it has not made a request, it propagates the grant to the next device in line, which behaves the same way, and so on until some device accepts the grant and takes the bus. This scheme is called **daisy chaining**. It has the property that devices are effectively assigned priorities depending on how close to the arbiter they are. The closest device wins.

To get around the implicit priorities based on distance from the arbiter, many buses have multiple priority levels. For each priority level there is a bus request line and a bus grant line. The one of Fig. 3-40(b) has two levels, 1 and 2 (real buses often have 4, 8, or 16 levels). Each device attaches to one of the bus request levels, with more time-critical devices attaching to the higher-priority ones. In Fig. 3-40(b) devices, 1, 2, and 4 use priority 1 while devices 3 and 5 use priority 2.

If multiple priority levels are requested at the same time, the arbiter issues a grant only on the highest-priority one. Among devices of the same priority, daisy chaining is used. In Fig. 3-40(b), in the event of conflicts, device 2 beats device 4, which beats 3. Device 5 has the lowest-priority because it is at the end of the lowest priority daisy chain.

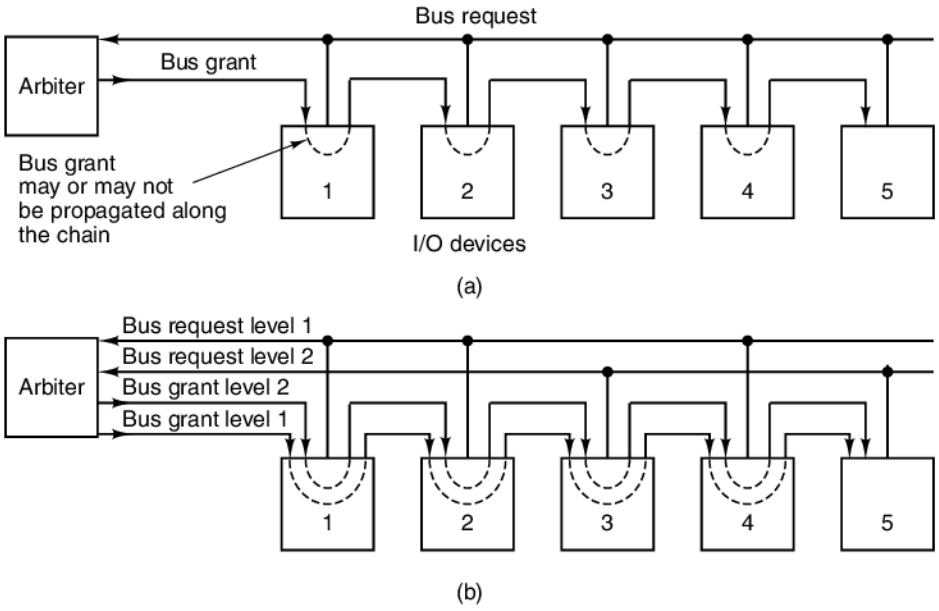


Figure 3-40. (a) A centralized one-level bus arbiter using daisy chaining. (b) The same arbiter, but with two levels.

As an aside, it is not technically necessary to wire the level 2 bus grant line serially through devices 1 and 2, since they cannot make requests on it. However, as an implementation convenience, it is easier to wire all the grant lines through all the devices, rather than making special wiring that depends on which device has which priority.

Some arbiters have a third line that a device asserts when it has accepted a grant and seized the bus. As soon as it has asserted this acknowledgement line, the request and grant lines can be negated. As a result, other devices can request the bus while the first device is using the bus. By the time the current transfer is finished, the next bus master will have already been selected. It can start as soon as the acknowledgement line has been negated, at which time the following round of arbitration can begin. This scheme requires an extra bus line and more logic in each device, but it makes better use of bus cycles.

In systems in which memory is on the main bus, the CPU must compete with all the I/O devices for the bus on nearly every cycle. One common solution for this situation is to give the CPU the lowest priority, so it gets the bus only when nobody else wants it. The idea here is that the CPU can always wait, but I/O devices frequently must acquire the bus quickly or lose incoming data. Disks rotating at high speed cannot wait. This problem is avoided in many modern computer systems by

putting the memory on a separate bus from the I/O devices so they do not have to compete for access to the bus.

Decentralized bus arbitration is also possible. For example, a computer could have 16 prioritized bus request lines. When a device wants to use the bus, it asserts its request line. All devices monitor all the request lines, so at the end of each bus cycle, each device knows whether it was the highest-priority requester, and thus whether it is permitted to use the bus during the next cycle. Compared to centralized arbitration, this arbitration method requires more bus lines but avoids the potential cost of the arbiter. It also limits the number of devices to the number of request lines.

Another kind of decentralized bus arbitration, shown in Fig. 3-41, uses only three lines, no matter how many devices are present. The first bus line is a wired-OR line for requesting the bus. The second bus line is called BUSY and is asserted by the current bus master. The third line is used to arbitrate the bus. It is daisy chained through all the devices. The head of this chain is held asserted by tying it to the power supply.

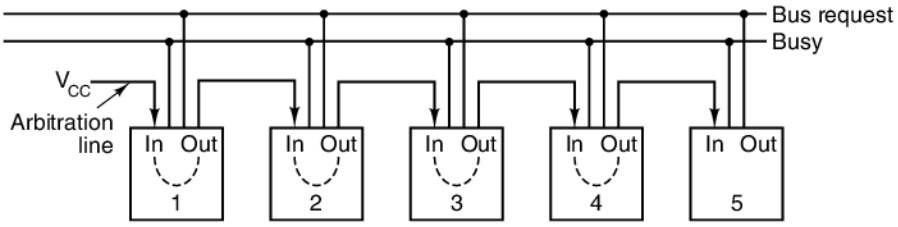


Figure 3-41. Decentralized bus arbitration.

When no device wants the bus, the asserted arbitration line is propagated through to all devices. To acquire the bus, a device first checks to see if the bus is idle and the arbitration signal it is receiving, IN, is asserted. If IN is negated, it may not become bus master, and it negates OUT. If IN is asserted, however, and the device wants the bus, the device negates OUT, which causes its downstream neighbor to see IN negated and to negate its OUT. Then all downstream devices all see IN negated and correspondingly negate OUT. When the dust settles, only one device will have IN asserted and OUT negated. This device becomes bus master, asserts BUSY and OUT, and begins its transfer.

Some thought will reveal that the leftmost device that wants the bus gets it. Thus, this scheme is similar to the original daisy chain arbitration, except without having the arbiter, so it is cheaper, faster, and not subject to arbiter failure.

3.4.6 Bus Operations

Up until now, we have discussed only ordinary bus cycles, with a master (typically the CPU) reading from a slave (typically the memory) or writing to one. In fact, several other kinds of bus cycles exist. We will now look at some of these.

Normally, one word at a time is transferred. However, when caching is used, it is desirable to fetch an entire cache line (e.g., 8 consecutive 64-bit words) at once. Often block transfers can be made more efficient than successive individual transfers. When a block read is started, the bus master tells the slave how many words are to be transferred, for example, by putting the word count on the data lines during T_1. Instead of just returning one word, the slave outputs one word during each cycle until the count has been exhausted. Figure 3-42 shows a modified version of Fig. 3-38(a), but now with an extra signal $\overline{\text{BLOCK}}$ that is asserted to indicate that a block transfer is requested. In this example, a block read of 4 words takes 6 cycles instead of 12.

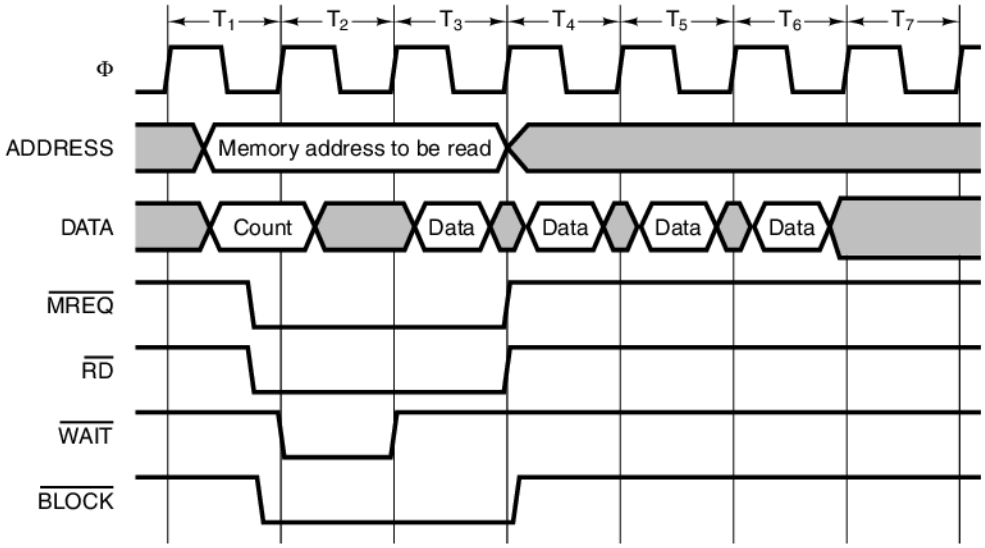


Figure 3-42. A block transfer.

Other kinds of bus cycles also exist. For example, on a multiprocessor system with two or more CPUs on the same bus, it is often necessary to make sure that only one CPU at a time uses some critical data structure in memory. A typical way to arrange this is to have a variable in memory that is 0 when no CPU is using the data structure and 1 when it is in use. If a CPU wants to gain access to the data structure, it must read the variable, and if it is 0, set it to 1. The trouble is, with some bad luck, two CPUs might read it on consecutive bus cycles. If each one sees that the variable is 0, then each one sets it to 1 and thinks that it is the only CPU using the data structure. This sequence of events leads to chaos.

To prevent this situation, multiprocessor systems often have a special read-modify-write bus cycle that allows any CPU to read a word from memory, inspect and modify it, and write it back to memory, all without releasing the bus.

This type of cycle prevents competing CPUs from being able to use the bus and thus interfere with the first CPU's operation.

Another important kind of bus cycle is for handling interrupts. When the CPU commands an I/O device to do something, it usually expects an interrupt when the work is done. The interrupt signaling requires the bus.

Since multiple devices may want to cause an interrupt simultaneously, the same kind of arbitration problems are present here that we had with ordinary bus cycles. The usual solution is to assign priorities to devices and use a centralized arbiter to give priority to the most time-critical devices. Standard interrupt interfaces exist and are widely used. In Intel-processor-based PCs, the chipset incorporates an 8259A interrupt controller, illustrated in Fig. 3-43.

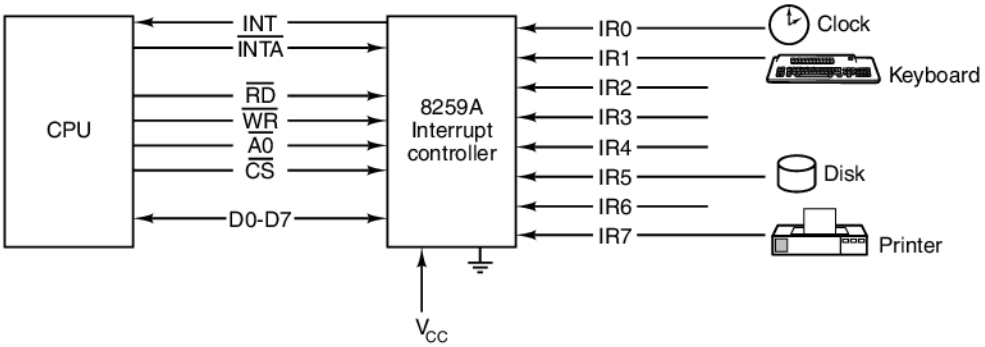


Figure 3-43. Use of the 8259A interrupt controller.

Up to eight 8259A I/O controllers can be directly connected to the eight IRx (Interrupt Request) inputs to the 8259A. When any of these devices wants to cause an interrupt, it asserts its input line. When one or more inputs are asserted, the 8259A asserts INT (INTerrupt), which directly drives the interrupt pin on the CPU. When the CPU is able to handle the interrupt, it sends a pulse back to the 8259A on INTA (INTerrupt Acknowledge). At that point the 8259A must specify which input caused the interrupt by outputting that input's number on the data bus. This operation requires a special bus cycle. The CPU hardware then uses that number to index into a table of pointers, called **interrupt vectors**, to find the address of the procedure to run to service the interrupt.

The 8259A has several registers inside that the CPU can read and write using ordinary bus cycles and the $\overline{\text{RD}}$ (ReaD), $\overline{\text{WR}}$ (WRite), $\overline{\text{CS}}$ (Chip Select), and $\overline{\text{A0}}$ pins. When the software has handled the interrupt and is ready to take the next one, it writes a special code into one of the registers, which causes the 8259A to negate INT, unless it has another interrupt pending. These registers can also be written to put the 8259A in one of several modes, mask out a set of interrupts, and enable other features.

When more than eight I/O devices are present, the 8259As can be cascaded. In the most extreme case, all eight inputs can be connected to the outputs of eight

more 8259As, allowing for up to 64 I/O devices in a two-stage interrupt network. The Intel ICH10 I/O controller hub, one of the of the chips in the Core i7 chipset, incorporates two 8259A interrupt controllers. This gives the ICH10 15 external interrupts, one less than 16 interrupts on the two 8259A controllers because one of the interrupts is used to cascade the second 8259A onto the first one. The 8259A has a few pins devoted to handling this cascading, which we have omitted for the sake of simplicity. Nowadays, the "8259A" is really part of another chip.

While we have by no means exhausted the subject of bus design, the material above should give enough background to understand the essentials of how a bus works, and how CPUs and buses interact. Let us now move from the general to the specific and look at some examples of actual CPUs and their buses.

3.5 EXAMPLE CPU CHIPS

In this section we will examine the Intel Core i7, TI OMAP4430, and Atmel ATmega168 chips in some detail at the hardware level.

3.5.1 The Intel Core i7

The Core i7 is a direct descendant of the 8088 CPU used in the original IBM PC. The first Core i7 was introduced in November 2008 as a four-processor 731-million transistor CPU running up to 3.2 GHz with a line width of 45 nanometers. The line width is how wide the wires between transistors are (as well as being a measure of the size of the transistors themselves). The narrower the line width, the more transistors can fit on the chip. Moore's law is fundamentally about the ability of process engineers to keep reducing the line widths. For comparison purposes, human hairs range from 20,000 to 100,000 nanometers in diameter, with blonde hair being finer than black hair.

The initial release of the Core i7 architecture was based on the "Nahalem" architecture; however, the newest versions of the Core i7 are built on the newer "Sandy Bridge" architecture. The architecture in this context represents the internal organization of the CPU, which is often given a code name. Despite being generally serious people, computer architects will sometimes come up with very clever code names for their projects. One of particular note was the AMD K-series architectures, which were designed to break Intel's seeming invulnerable hold on the desktop CPU market. The K-series processors' code name was "Kryptonite," a reference to the only substance that could hurt Superman, and a clever jab at the dominant Intel.

The new Sandy-Bridge-based Core i7 has evolved to having 1.16 billion transistors and running at speeds up to 3.5 GHz with line widths of 32 nanometers. Although the Core i7 is a far cry from the 29,000-transistor 8088, it is fully backward

compatible with the 8088 and can run unmodified 8088 binary programs (not to mention programs for all the intermediate processors as well).

From a software point of view, the Core i7 is a full 64-bit machine. It has all the same user-level ISA features as the 80386, 80486, Pentium, Pentium II, Pentium Pro, Pentium III, and Pentium 4 including the same registers, same instructions, and a full on-chip implementation of the IEEE 754 floating-point standard. In addition, it has some new instructions intended primarily for cryptographic operations.

The Core i7 processor is a multicore CPU, thus the silicon die contains multiple processors. The CPU is sold with a varying number of processors, ranging from 2 to 6 with more planned for the near future. If programmers write a parallel program, using threads and locks, it is possible to gain significant program speedups by exploiting parallelism on multiple processors. In addition, the individual CPUs are "hyperthreaded" such that multiple hardware threads can be active simultaneously. Hyperthreading (more typically called "simultaneous multithreading" by computer architects) allows very short latencies, such as cache misses, to be tolerated with hardware thread switches. Software-based threading can tolerate only very long latencies, such as page faults, due to the hundreds of cycles needed to implement software-based thread switches.

Internally, at the microarchitecture level, the Core i7 is a very capable design. It is based on the architecture of its predecessors, the Core 2 and Core 2 Duo. The Core i7 processor can carry out up to four instructions at once, making it a 4-wide superscalar machine. We will examine the microarchitecture in Chap. 4.

The Core i7 processors all have three levels of cache. Each processor in a Core i7 processor has a 32-KB level 1 (L1) data cache and a 32-KB level 1 instruction cache. Each core also has its own 256-KB level 2 (L2) cache. The second-level cache is unified, which means that it can hold a mixture of instructions and data. All cores share a single level 3 (L3) unified cache, the size of which varies from 4 to 15 MB depending on the processor model. Having three levels of cache significantly improves processor performance but at a great cost in silicon area, as Core i7 CPUs can have as much as 17 MB total cache on a single silicon die.

Since all Core i7 chips have multiple processors with private data caches, a problem arises when a processor modifies a word in this private cache that is contained in another processor's cache. If the other processor tries to read that word from memory, it will get a stale value, since modified cache words are not written back to memory immediately. To maintain memory consistency, each CPU in a multiprocessor system **snoops** on the memory bus looking for references to words it has cached. When it sees such a reference, it jumps in and supplies the required data before the memory gets a chance to do so. We will study snooping in Chap. 8.

Two primary external buses are used in Core i7 systems, both of them synchronous. A DDR3 memory bus is used to access the main memory DRAM, and a PCI Express bus connects the processor to I/O devices. High-end versions of the Core i7 include multiple memory and PCI Express buses, and they also include a

Quick Path Interconnect (QPI) port. The QPI port connects the processor to an external multiprocessor interconnect, allowing systems with more than six processors to be built. The QPI port sends and receives cache coherency requests, plus a variety of other multiprocessor management messages such as interprocessor interrupts.

A problem with the Core i7 as well as with most other modern desktop-class CPUs, is the power it consumes and the heat it generates. To prevent damaging the silicon, the heat must be moved away from the processor die soon after it is produced. The Core i7 consumes between 17 and 150 watts, depending on the frequency and model. Consequently, Intel is constantly searching for ways to manage the heat produced by its CPU chips. Cooling technologies and heat-conductive packaging are vital to protecting the silicon from burning up.

The Core i7 comes in a square LGA package 37.5 mm on edge. It contains 1155 pads on the bottom, of which 286 are for power and 360 are grounded to reduce noise. The pads are arranged roughly as a 40×40 square, with the middle 17×25 missing. In addition, 20 more pads are missing at the perimeter in an asymmetric pattern, to prevent the chip from being inserted incorrectly in its socket. The physical pinout is shown in Fig. 3-44.

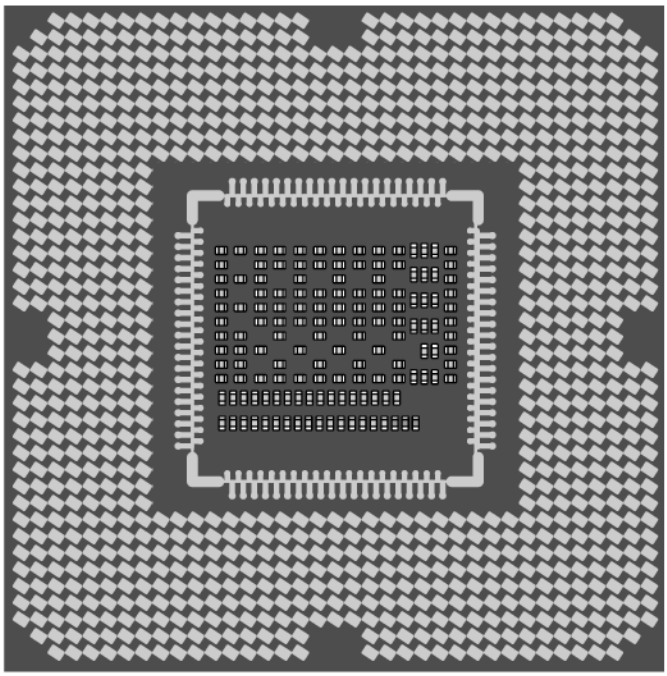


Figure 3-44. The Core i7 physical pinout.

The chip is outfitted with a mounting plate for a heat sink to distribute the heat and a fan to cool it. To get some idea of how large the power problem is, turn on a

150-watt incandescent light bulb, let it warm up, and then put your hands around it (but do not touch it). This amount of heat must be dissipated continuously by a high-end Core i7 processor. Consequently, when a Core i7 has outlived its usefulness as a CPU, it can always be used as a camp stove.

According to the laws of physics, anything that puts out a lot of heat must suck in a lot of energy. In a portable computer with a limited battery charge, using a lot of energy is not desirable because it drains the battery quickly. To address this issue, Intel has provided a way to put the CPU to sleep when it is idle and to put it into a deep sleep when it is likely to be that way for a while. Five states are provided, ranging from fully active to deep sleep. In the intermediate states, some functionality (such as cache snooping and interrupt handling) is enabled, but other functions are disabled. When in deep sleep state, the register values are preserved, but the caches are flushed and turned off. When in deep sleep, a hardware signal is required to wake it up. It is not known whether a Core i7 can dream when it is in deep sleep.

The Core i7's Logical Pinout

The 1155 pins on the Core i7 are used for 447 signals, 286 power connections (at several different voltages), 360 grounds, and 62 spares for future use. Some of the logical signals use two or more pins (such as the memory-address requested), so there are only 131 different signals. A somewhat simplified logical pinout is given in Fig. 3-45. On the left side of the figure are five major groups of bus signals; on the right side are various miscellaneous signals.

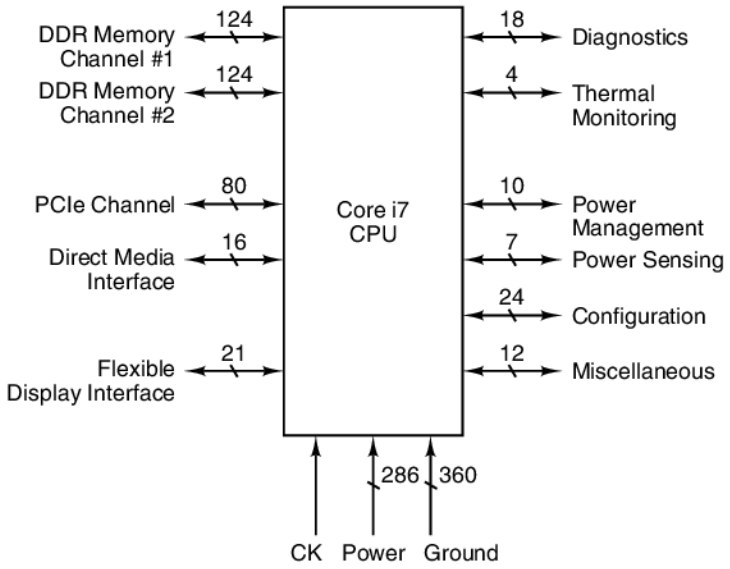


Figure 3-45. Logical pinout of the Core i7.

Let us examine the signals, starting with the bus signals. The first two bus signals are used to interface to DDR3 compatible DRAM. This group of signals provides address, data, control and clock to the DRAM bank. The Core i7 supports two independent DDR3 DRAM channels, running with a 666-MHz bus clock that transfers on both edges to allow 1333 million transactions per second. The DDR3 interface is 64 bits wide, thus, the two DDR3 interfaces work in tandem to provide memory-hungry programs up to 20 gigabytes of data each second.

The third bus group is the PCI Express interface, which is used to connect peripherals directly to the Core i7 CPU. The PCI Express interface is a high-speed serial interface, with each single serial link forming a "lane" of communication with peripherals. The Core i7 link is an x16 interface, which means that it can utilize 16 lanes simultaneously for an aggregate bandwidth of 16 GB/sec. Despite its being a serial channel, a rich set of commands travel over PCI Express links, including device reads, writes, interrupts, and configuration setup commands.

The next bus group is the Direct Media Interface (DMI), which is used to connect the Core i7 CPU to its companion **chipset**. The DMI interface is similar to the PCI Express interface, although it runs at about half the speed since four lanes can provide only up to 2.5-GB-per-second data transfer rates. A CPU's chipset contains a rich set of additional peripheral interface support, typically required for higher-end system with many I/O devices. The Core i7 chipset is composed of the P67 and ICH10 chips. The P67 chip is the Swiss Army knife of chips, providing SATA, USB, Audio, PCIe, and Flash memory interfaces. The ICH10 chip provides legacy interface support, including a PCI interface and the 8259A interrupt control functionality. Additionally, the ICH10 contains a handful of other circuits, such as real-time clocks, event timers, and direct memory access (DMA) controllers. Having chips like these greatly simplifies construction of a full PC.

The Core i7 can be configured to use interrupts the same way as on the 8088 (for purposes of backward compatibility), or it can also use a new interrupt system using a device called an **APIC** (**Advanced Programmable Interrupt Controller**).

The Core i7 can run at any one of several predefined voltages, but it has to know which one. The power-management signals are used for automatic power-supply voltage selection, telling the CPU that power is stable, and other power-related matters. Managing the various sleep states is also done here since sleeping is done for reasons of power management.

Despite sophisticated power management, the Core i7 can get very hot. To protect the silicon, each Core i7 processor contains multiple internal heat sensors that detect when the chip is about to overheat. The thermal monitoring group deals with thermal management, allowing the CPU to indicate to its environment that it is in danger of overheating. One of the pins is asserted by the CPU if its internal temperature reaches 130°C (266°F). If a CPU ever hits this temperature, it is probably dreaming about retirement and becoming a camp stove.

Even at camp-stove temperatures, you need not worry about the safety of the Core i7. If the internal sensors detect that the processor is about to overheat, it will

initiate **thermal throttling**, which is a technique that quickly reduces heat generation by running the processor only every *N*th clock cycle. The larger the value of *N*, the more the processor is throttled down, and the more quickly it will cool. Of course, the cost of this throttling is a decrease in system performance. Prior to the invention of thermal throttling, CPUs would burn up if their cooling system failed. Evidence of these dark times of CPU thermal management can be found by searching for "exploding CPU" on YouTube. The video is fake but the problem is not.

The Clock signal provides the system clock to the processor, which internally is used to generate variety of clocks based on a multiple or fraction of the system clock. Yes, it is possible to generate a multiple of the system clock frequency, using a very clever device called a delay-locked loop, or DLL.

The Diagnostics group contains signals for testing and debugging systems in conformance with the IEEE 1149.1 JTAG (Joint Test Action Group) test standard. Finally, the miscellaneous group is a hodge-podge of other signals that have various special purposes.

Pipelining on the Core i7's DDR3 Memory Bus

Modern CPUs like the Core i7 place heavy demands on DRAM memories. Individual processors can produce access requests much faster than a slow DRAM can produce values, and this problem is compounded when multiple processors are making simultaneous requests. To keep the CPUs from starving for lack of data, it is essential to get the maximum possible throughput from the memory. For this reason, the Core i7 DDR3 memory bus can be operated in a pipelined manner, with as many as four simultaneous memory transactions going on at the same time. We saw the concept of pipelining in Chap. 2 in the context of a pipelined CPU (see Fig. 2-4), but memories can also be pipelined.

To allow pipelining, Core i7 memory requests have three steps:

1. The memory ACTIVATE phase, which "opens" a DRAM memory row, making it ready for subsequent memory accesses.
2. The memory READ or WRITE phase, where multiple accesses can be made to individual words within the currently open DRAM row or to multiple sequential words within the current DRAM row using a burst mode.
3. The PRECHARGE phase, which "closes" the current DRAM memory row, and prepares the DRAM memory for the next ACTIVATE command.

The secret to the Core i7's pipelined memory accesses is that DDR3 DRAMs are organized with multiple **banks** within the DRAM chip. A bank is a block of DRAM memory, which may be accessed in parallel with other DRAM memory

banks, even if they are contained in the same chip. A typical DDR3 DRAM chip will have as many as 8 banks of DRAM. However, the DDR3 interface specification allows only up to four concurrent accesses on a single DDR3 channel. The timing diagram in Fig. 3-46 illustrates the Core i7 making 4 memory accesses to three distinct DRAM banks. The accesses are fully overlapped, such that the DRAM reads occur in parallel within the DRAM chip. The figure shows which commands lead to later operations through the use of arrows in the timing diagram.

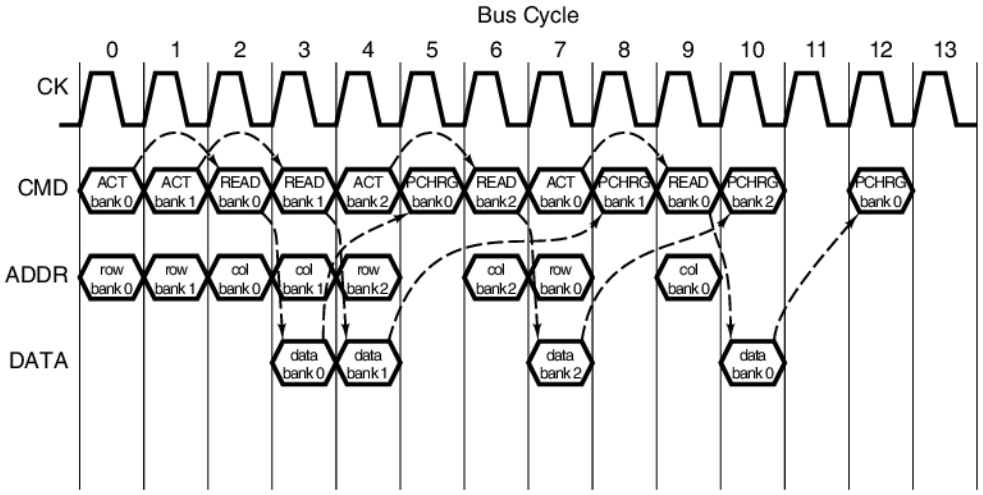


Figure 3-46. Pipelining memory requests on the Core i7's DDR3 interface.

As shown in Fig. 3-46, the DDR3 memory interface has four primary signal paths: bus clock (CK), bus command (CMD), address (ADDR), and data (DATA). The bus clock signal CK orchestrates all bus activity. The bus command CMD indicates what activity is requested of the connect DRAM. The ACTIVATE command specifies the address of the DRAM row to open via the ADDR signal. When a READ is executed, the DRAM column address is given via the ADDR signals, and the DRAM produces the read value a fixed time later on the DATA signals. Finally, the PRECHARGE command indicates the bank to precharge via the ADDR signals. For the purpose of the example, the ACTIVATE command must precede the first READ to the same bank by two DDR3 bus cycles, and data are produced one bus cycle after the READ command. Additionally, the PRECHARGE operation must occur at least two bus cycles after the last READ operation to the same DRAM bank.

The parallelism in the memory requests can be seen in the overlapping of the READ requests to the different DRAM banks. The first two READ accesses to banks 0 and 1 are completely overlapped, producing results in bus cycles 3 and 4, respectively. The access to bank 2 partially overlaps with the first access of bank 1, and finally the second read of bank 0 partially overlaps with the access to bank 2.

You might be wondering how the Core i7 knows when to expect its READ command data to return, and when it can make a new memory request. The answer is that it knows when to receive and initiate requests because it fully models the internal activities of each attached DDR3 DRAM. Thus, it will anticipate the return of data in the correct cycle, and it will know to avoid initiating a precharge operation until two cycles after its last read operation. The Core i7 can anticipate all of these activities because the DDR3 memory interface is a **synchronous memory interface**. Thus, all activities take a well-known number of DDR3 bus cycles. Even with all of this knowledge, building a high-performance fully pipelined DDR3 memory interface is a nontrivial task, requiring many internal timers and conflict detectors to implement efficient DRAM request handling.

3.5.2 The Texas Instruments OMAP4430 System-on-a-Chip

As our second example of a CPU chip, we will now examine the Texas Instruments (TI) OMAP4430 **system-on-a-chip** (**SoC**). The OMAP4430 implements the ARM instruction set, and it is targeted at mobile and embedded applications such as smartphones, tablets, and Internet appliances. Aptly named, a system-on-a-chip incorporates a wide range of devices such that the SoC combined with physical peripherals (touchscreen, flash memory, etc.) implements a complete computing device.

The OMAP4430 SoC includes two ARM A9 cores, additional accelerators, and a wide range of peripheral interfaces. The internal organization of the OMAP4430 is shown in Fig. 3-47. The ARM A9 cores are 2-wide superscalar microarchitectures. In addition, there are three more accelerator processors on the OMAP4430 die: a POWERVR SGX540 graphics processor, an image signal processor (ISP), and an IVA3 video processor. The SGX540 provides efficient programmable 3D rendering, similar to the GPUs found in desktop PCs, albeit smaller and slower. The ISP is a programmable processor designed for efficient image manipulation, for the type of operations that would be required in a high-end digital camera. The IVA3 implements efficient video encoding and decoding, with enough performance to support 3D applications like those found in handheld game consoles. Also included in the OMAP4430 SoC is a wide range of peripheral interfaces, including a touchscreen and keypad controllers, DRAM and flash interfaces, USB, and HDMI. Texas Instruments has detailed the roadmap for the OMAP series of CPUs. Future designs will have more of everything—more ARM cores, more GPUs, and more diverse peripherals.

The OMAP4430 SoC was introduced in early 2011 with two ARM A9 cores running at 1 GHz using a 45-nanometer silicon implementation. A key aspect of the OMAP4430 design is that it performs significant amounts of computation with very little power, since it is targeted to mobile applications that are powered by a battery. In battery-powered mobile applications, the more efficiently the design operates, the longer the user can go between battery charges.

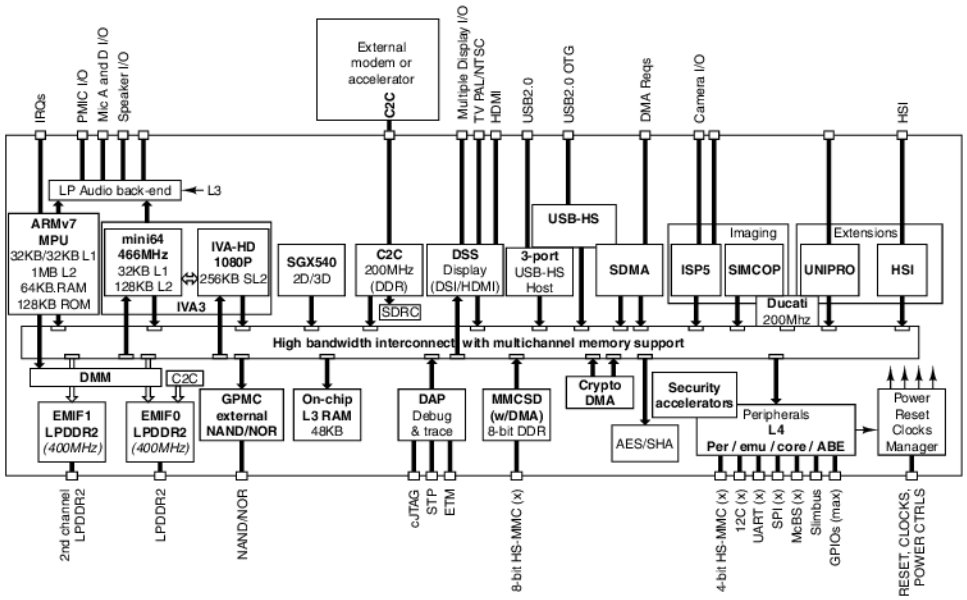


Figure 3-47. The internal organization of the OMAP4430 system-on-a-chip.

The many processors of the OMAP4430 are incorporated specifically to support its mission of low-power operation. The graphics processor, ISP, and IVA3 are all programmable accelerators that provide efficient computation capabilities at significantly less energy compared to the same tasks running on the ARM A9 CPUs alone. Fully powered, the OMAP4430 SoC draws only 600 mW of power. Compared to a high-end Core i7, the OMAP4430 uses about 1/250 as much power. The OMAP4430 also implements a very efficient sleep mode; when all components are asleep, the design draws only 100 μW. Efficient sleep modes are crucial to mobile applications with long periods of standby time, such as a cell phone. The less energy used in sleep mode, the longer the cell phone can stay in standby mode.

To further reduce power demands of the OMAP4430, the design incorporates a variety of power-management facilities, including **dynamic voltage scaling** and **power gating**. Dynamic voltage scaling allows components to run slower at a lower voltage, which significantly reduces power requirements. If you do not need the CPU's most blazing speed for computation, the voltage of the design can be lowered to run the CPU at a slower speed and much energy will be saved. Power gating is an even more aggressive power-management technique where a component is powered down completely when it is not in use, thereby eliminating its power draw. For example in a tablet application, if the user is not watching a movie, the IVA3 video processor is completely powered down and draws no power. Conversely, when the user is watching a movie, the IVA3 video processor is speeding through its video decoding tasks, while the two ARM A9 CPUs are asleep.

Despite its bent for joule-frugal operation, the ARM A9 cores utilize a very capable microarchitecture. They can decode and execute up to two instructions each cycle. As we will learn in Chap. 4, this execution rate represents the maximum throughput of the microarchitecture. But do not expect it to execute this many instructions each cycle. Rather, think of this rate as the manufacturer's guaranteed maximum performance, a level that the processor will never exceed, no matter what. In many cycles, fewer than two instructions will execute due to the myriad of "hazards" that can stall instructions, leading to lower execution throughput. To address many of these throughput limiters, the ARM A9 incorporates a powerful branch predictor, out-of-order instruction scheduling, and a highly optimized memory system.

The OMAP4430's memory system has two main internal L1 caches for each ARM A9 processor: a 32-KB cache for instructions and a 32-KB cache for data. Like the Core i7, it also uses an on-chip level 2 (L2) cache, but unlike the Core i7, it is a relatively tiny 1 MB in size, and it is shared by both ARM A9 cores. The caches are fed with dual LPDDR2 low-power DRAM channels. LPDDR2 is derived from the DDR2 memory interface standard, but changed to require fewer wires and to operate at more power-efficient voltages. Additionally, the memory controller incorporates a number of memory-access optimizations, such as tiled memory prefetching and in-memory rotation support.

While we will discuss caching in detail in Chap. 4, a few words about it here will be useful. All of main memory is divided up into cache lines (blocks) of 32 bytes. The 1024 most heavily used instruction lines and the 1024 most heavily used data lines are in the level 1 cache. Cache lines that are heavily used but which do not fit in the level 1 cache are kept in the level 2 cache. This cache contains both data lines and instruction lines from both ARM A9 CPUs mixed at random. The level 2 cache contains the most recently touched 32,768 lines in main memory.

On a level 1 cache miss, the CPU sends the identifier of the line it is looking for (Tag address) to the level 2 cache. The reply (Tag data) provides the information for the CPU to tell whether the line is in the level 2 cache, and if so, what state it is in. If the line is cached there, the CPU goes and gets it. Getting a value out of the level 2 cache takes 19 cycles. This is a long time to wait for data, so clever programmers will optimize their programs to use less data, making it more likely to find data in the fast level 1 cache.

If the cache line is not in the level 2 cache, it must be fetched from main memory via the LPDDR2 memory interface. The OMAP4430 LPDDR2 interface is implemented on-chip such that LPDDR2 DRAM can be connected directly to the OMAP4430. To access memory, the CPU must first send the upper portion of the DRAM address to the DRAM chip, using the 13 address lines. This operation, called an *ACTIVATE*, loads an entire row of memory within the DRAM into a row buffer. Subsequently, the CPU can issue multiple READ or $\overline{\text{WRITE}}$ commands, sending the remainder of the address on the same 13 address lines, and sending (or receiving) the data for the operation on the 32 data lines.

While waiting for the results, the CPU may well be able to continue with other work. For example, a cache miss while prefetching an instruction does not inhibit the execution of one or more instructions already fetched, each of which may refer to data not in any cache. Thus, multiple transactions on the two LPDDR2 interfaces may be outstanding at once, even for the same processor. It is up to the memory controller to keep track of all this and to make actual memory requests in the most efficient order.

When data finally arrives from the memory, it can come in 4 bytes at a time. A memory operation may utilize a burst mode read or write, which will allow multiple contiguous addresses within the same DRAM row to be read or written. This mode is particularly efficient for reading or writing cache blocks. Just for the record, the description of the OMAP4430 given above, like that of the Core i7 before it, has been highly simplified, but the essence of its operation has been described.

The OMAP4430 comes in a 547-pin **ball grid array** (PBGA), as shown in Fig. 3-48. A ball grid array is similar to a land grid array except that the connections on the chip are small metal balls, rather than the square pads used in the LGA. The two packages are not compatible, providing further evidence that you cannot stick a square peg into a round hole. The OMAP4430's package consists of a rectangular array of 28×26 balls, with two inner rings of balls missing, plus two asymmetric half rows and columns of balls missing to prevent the chip from being inserted incorrectly in the BGA socket.

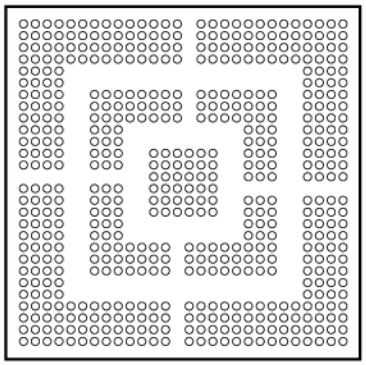


Figure 3-48. The OMAP4430 system-on-a-chip pinout.

It is difficult to compare a CISC chip (like the Core i7) and a RISC chip (like the OMAP4430) based on clock speed alone. For example, the two ARM A9 cores in the OMAP4430 have a peak execution speed of four instructions per clock cycle, giving it almost the same execution rate as that of the Core i7's 4-wide superscalar processors. The Core i7 achieves faster program execution, however, since it has up to six processors running with a clock speed 3.5 times faster (3.5 GHz) than the OMAP4430. The OMAP4430 may seem like a turtle running next

to the Core i7 rabbit, but the turtle uses much less power, and the turtle may finish first, especially if the rabbit's battery is not very big.

3.5.3 The Atmel ATmega168 Microcontroller

Both the Core i7 and the OMAP4430 are examples of high-performance computing platforms designed for building highly capable computing devices, with the Core i7 focusing on desktop applications while the OMAP4430 focuses on mobile applications. When many people think about computers, systems like these come to mind. However, another whole world of computers exists that is actually far more pervasive: embedded systems. In this section we will take a brief look at that world.

It is probably only a slight exaggeration to say that every electrical device costing more than $100 has a computer in it. Certainly televisions, cell phones, electronic personal organizers, microwave ovens, camcorders, VCRs, laser printers, burglar alarms, hearing aids, electronic games, and other devices too numerous to mention are all computer controlled these days. The computers inside these things tend to be optimized for low price rather than for high performance, which leads to different trade-offs than the high-end CPUs we have been studying so far.

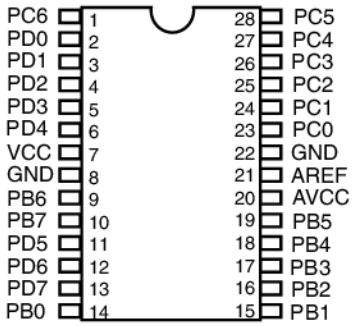


Figure 3-49. Physical pinout of the ATmega168.

As we mentioned in Chap. 1, the Atmel ATmega168 microcontroller is widely used, mostly due to its very low cost (about $1). As we will see shortly, it is also a versatile chip, which makes interfacing to it simple and inexpensive. So let us now examine this chip, whose physical pinout is shown in Fig. 3-49.

As can be seen from the figure, the ATmega168 normally comes in a standard 28-pin package (although other packages are available). At first glance, you probably noticed that the pinout on this chip is a bit strange compared to the previous two designs we examined. In particular, this chip has no address and data lines. This is because the chip is not designed to be connected to memory, only to devices. All of the memory, SRAM and flash, is contained within the processor, obviating the need for any address and data pins as shown in Fig. 3-50.

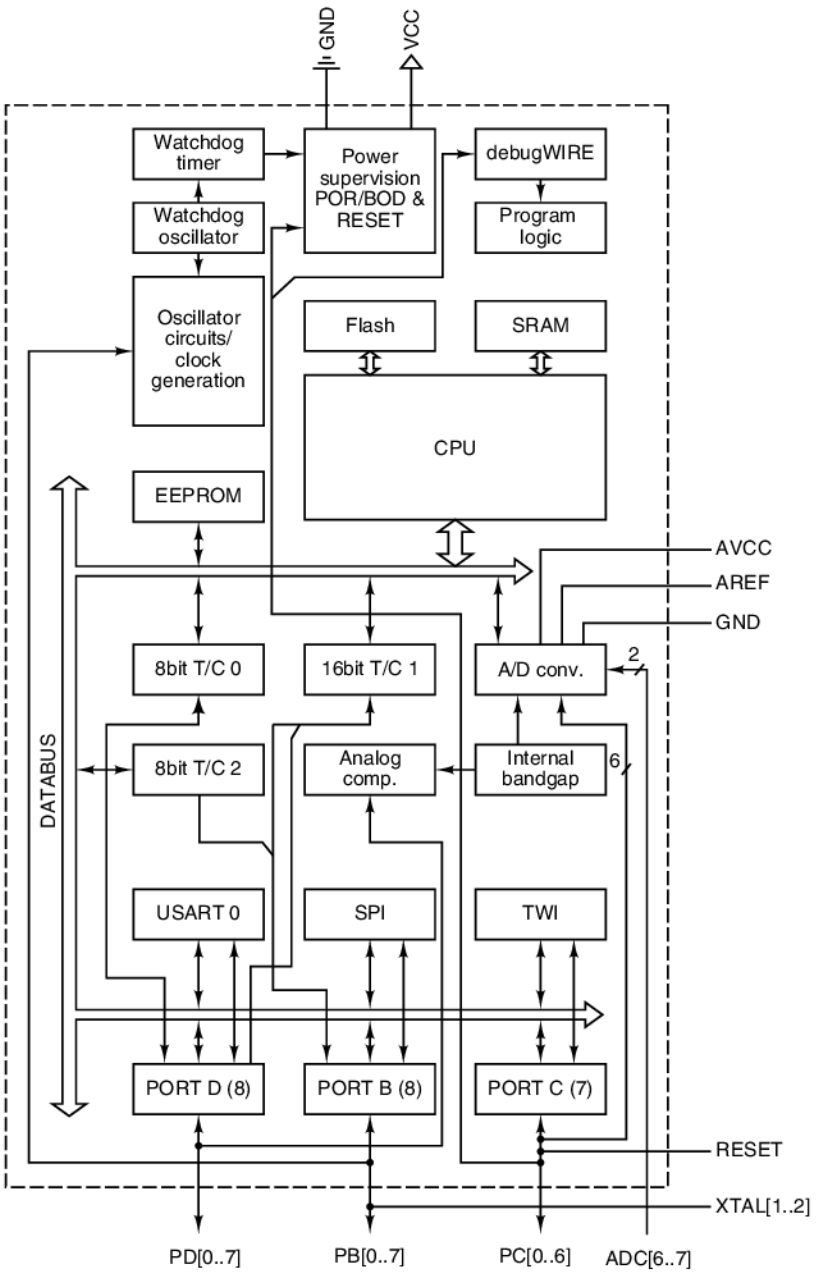


Figure 3-50. The internal architecture and logical pinout of the ATmega168.

Instead of address and data pins, the ATmega168 has 27 digital I/O ports, 8 lines in port B and D, and 7 lines in port C. These digital I/O lines are designed to be connected to I/O peripherals, and each line can be internally configured by startup software to be an input or an output. For example, when used in a microwave oven, one digital I/O line would be an input from the "door open" sensor. Another digital I/O line would be an output used to turn the microwave generator on and off. Software in the ATmega168 would check that the door was closed before turning on the microwave generator. If the door is suddenly opened, the software must kill the power. In practice, hardware interlocks are always present, too.

Optionally, six of the inputs from port C can be configured to be analog I/O. Analog I/O pins can read the voltage level of an input or set the voltage level of an output. Extending our microwave oven example, some ovens have a sensor that allows the user to heat food to a given temperature. The temperature sensor would be connected to a C port input, and software could read the voltage of the sensor and then convert it to a temperature using a sensor-specific translation function. The remaining pins on the ATmega168 are the power input (VCC), two ground pins (GND), and two pins to configure the analog I/O circuitry (AREF, AVCC).

The internal architecture of the ATmega168, like that of the OMAP4430, is a system-on-a-chip with a rich array of internal devices and memory. The ATmega168 comes with up to 16 KB of internal flash memory, for storage of infrequently changing nonvolatile information such as program instructions. It also includes up to 1 KB of EEPROM, which is nonvolatile memory that can be written by software. The EEPROM stores system-configuration data. Again, using our microwave example, the EEPROM would store a bit indicating whether the microwave displayed time in 12- or 24-hour format. The ATmega168 also incorporates up to 1 KB of internal SRAM, where software can store temporary variables.

The internal processor runs the AVR instruction set, which is composed of 131 instructions, each 16 bits in length. The processor is an 8-bit processor, which means that it operates on 8-bit data values, and internally its registers are 8 bits in size. The instruction set incorporates special instructions that allow the 8-bit processor to efficiently operate on larger data types. For example, to perform 16-bit or larger additions, the processor provides the "add-with-carry" instruction, which adds two values plus the carry out of the previous addition. The remaining internal components include the real-time clock and a variety of interface logic, including support for serial links, pulse-width-modulated (PWM) links, I2C (Inter-IC bus) link, and analog and digital I/O controllers.

3.6 EXAMPLE BUSES

Buses are the glue that hold computer systems together. In this section we will take a close look at some popular buses: the PCI bus and the Universal Serial Bus. The PCI bus is the primary I/O peripheral bus used today in PCs. It comes in two

forms, the older PCI bus, and the new and much faster PCI Express (PCIe) bus. The Universal Serial Bus is an increasingly popular I/O bus for low-speed peripherals such as mice and keyboards. A second and third version of the USB bus run at much higher speeds. In the following sections, we will look at each of these buses in turn to see how they work.

3.6.1 The PCI Bus

On the original IBM PC, most applications were text based. Gradually, with the introduction of Windows, graphical user interfaces came into use. None of these applications put much strain on early system buses such as the ISA bus. However, as time went on and many applications, especially multimedia games, began to use computers to display full-screen, full-motion video, the situation changed radically.

Let us make a simple calculation. Consider a 1024×768 color video with 3 bytes/pixel. One frame contains 2.25 MB of data. For smooth motion, at least 30 screens/sec are needed, for a data rate of 67.5 MB/sec. In fact, it is worse than this, since to display a video from a hard disk, CD-ROM, or DVD, the data must pass from the disk drive over the bus to the memory. Then for the display, the data must travel over the bus again to the graphics adapter. Thus, we need a bus bandwidth of 135 MB/sec for the video alone, not counting the bandwidth that the CPU and other devices need.

The PCI bus' predecessor, the ISA bus, ran at a maximum rate of 8.33 MHz and could transfer 2 bytes per cycle, for a maximum bandwidth of 16.7 MB/sec. The enhanced ISA bus, called the EISA bus, could move 4 bytes per cycle, to achieve 33.3 MB/sec. Clearly, neither of these approached what is needed for full-screen video.

With modern full HD video the situation is even worse. It requires 1920×1080 frames at 30 frames/sec for a data rate of 155 MB/sec (or 310 MB/sec if the data have to traverse the bus twice). Clearly, the EISA bus does not even come close to handling this.

In 1990, Intel saw this coming and designed a new bus with a far higher bandwidth than the EISA bus. It was called the **PCI bus** (**Peripheral Component Interconnect bus**). To encourage its use, Intel patented the PCI bus and then put all the patents into the public domain, so any company could build peripherals for it without having to pay royalties. Intel also formed an industry consortium, the PCI Special Interest Group, to manage the future of the PCI bus. As a result, the PCI bus became extremely popular. Virtually every Intel-based computer since the Pentium has a PCI bus, and many other computers do, too. The PCI bus is covered in gory detail in Shanley and Anderson (1999) and Solari and Willse (2004).

The original PCI bus transferred 32 bits per cycle and ran at 33 MHz (30-nsec cycle time) for a total bandwidth of 133 MB/sec. In 1993, PCI 2.0 was introduced,

and in 1995, PCI 2.1 came out. PCI 2.2 has features for mobile computers (mostly for saving battery power). The PCI bus runs at up to 66 MHz and can handle 64-bit transfers, for a total bandwidth of 528 MB/sec. With this kind of capacity, full-screen, full-motion video is doable (assuming the disk and the rest of the system are up to the job). In any event, the PCI bus will not be the bottleneck.

Even though 528 MB/sec sounds pretty fast, the PCI bus still had two problems. First, it was not good enough for a memory bus. Second, it was not compatible with all those old ISA cards out there. The solution Intel thought of was to design computers with three or more buses, as shown in Fig. 3-51. Here we see that the CPU can talk to the main memory on a special memory bus, and that an ISA bus can be connected to the PCI bus. This arrangement met all requirements, and as a consequence it was widely used in the 1990s.

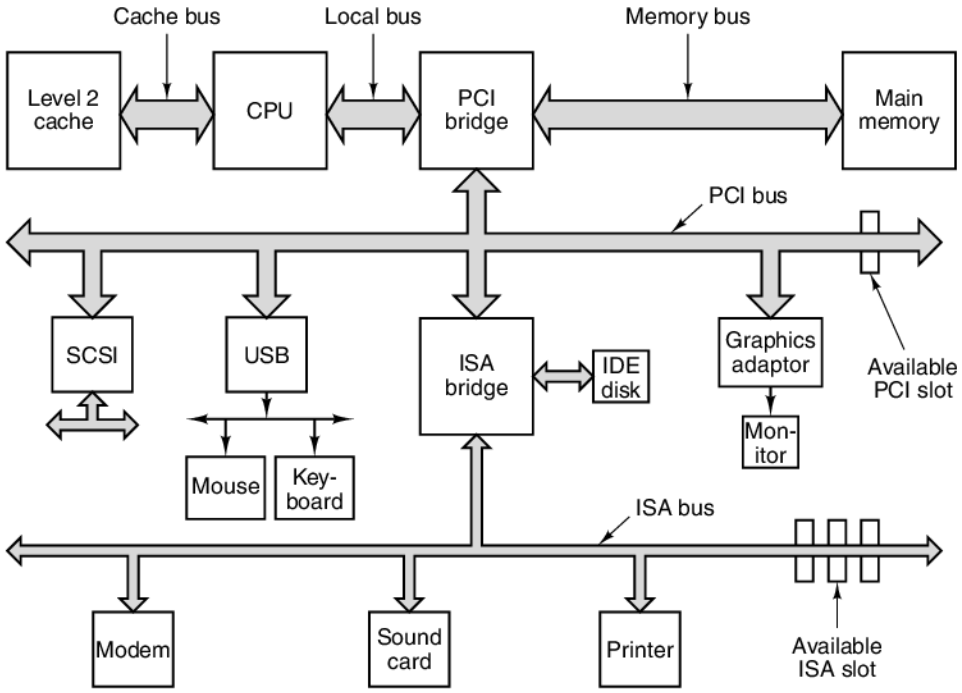


Figure 3-51. Architecture of an early Pentium system. The thicker buses have more bandwidth than the thinner ones but the figure is not to scale.

Two key components in this architecture are the two bridge chips (which Intel manufactures—hence its interest in this whole project). The PCI bridge connects the CPU, memory, and PCI bus. The ISA bridge connects the PCI bus to the ISA bus and also supports one or two IDE disks. Nearly all PC systems using this architecture would have one or more free PCI slots for adding new high-speed peripherals, and one or more ISA slots for adding low-speed peripherals.

The big advantage of the design of Fig. 3-51 is that the CPU has an extremely high bandwidth to memory using a proprietary memory bus; the PCI bus offers high bandwidth for fast peripherals such as SCSI disks, graphics adaptors, etc.; and old ISA cards can still be used. The USB box in the figure refers to the Universal Serial Bus, which will be discussed later in this chapter.

It would have been nice had there been only one kind of PCI card. Unfortunately, such is not the case. Options are provided for voltage, width, and timing. Older computers often use 5 volts and newer ones tend to use 3.3 volts, so the PCI bus supports both. The connectors are the same except for two bits of plastic that are there to prevent people from inserting a 5-volt card in a 3.3-volt PCI bus or vice versa. Fortunately, universal cards exist that support both voltages and can plug into either kind of slot. In addition to the voltage option, cards come in 32-bit and 64-bit versions. The 32-bit cards have 120 pins; the 64-bit cards have the same 120 pins plus an additional 64. A PCI bus system that supports 64-bit cards can also take 32-bit cards, but the reverse is not true. Finally, PCI buses and cards can run at either 33 or 66 MHz. The choice is made by having one pin wired either to the power supply or to ground. The connectors are identical for both speeds.

By the late 1990s, pretty much everyone agreed that the ISA bus was dead, so new designs excluded it. By then, however, monitor resolution had increased in some cases to 1600×1200 and the demand for full-screen full motion video had also increased, especially in the context of highly interactive games, so Intel added yet another bus just to drive the graphics card. This bus was called the **AGP bus** (**Accelerated Graphics Port bus**). The initial version, AGP 1.0, ran at 264 MB/sec, which was defined as 1x. While slower than the PCI bus, it was dedicated to driving the graphics card. Over the years, newer versions came out, with AGP 3.0 running at 2.1 GB/sec (8x). Today, even the high-performance AGP 3.0 bus has been usurped by even faster upstarts, in particular, the PCI Express bus, which can pump an amazing 16 GB/sec of data over high-speed serial bus links. A modern Core i7 system is illustrated in Fig. 3-52.

In a modern Core i7 based system, a number of interfaces have been integrated directly into the CPU chip. The two DDR3 memory channels, running at 1333 transactions/sec, connect to main memory and provide an aggregate bandwidth of 10 GB/sec per channel. Also integrated into the CPU is a 16-lane PCI Express channel, which optimally can be configured into a single 16-bit PCI Express bus or dual independent 8-bit PCI Express buses. The 16 lanes together provide a bandwidth of 16 GB/sec to I/O devices.

The CPU connects to the primary bridge chip, the P67, via the 20-Gb/sec (2.5 GB/sec) serial direct media interface (DMI). The P67 provides interfaces to a number of modern high-performance I/O interfaces. Eight additional PCI Express lanes are provided, plus SATA disk interfaces. The P67 also implements 14 USB 2.0 interfaces, 10G Ethernet and an audio interface.

The ICH10 chip provides legacy interface support for old devices. It is connected to the P67 via a slower DMI interface. The ICH10 implements the PCI bus,

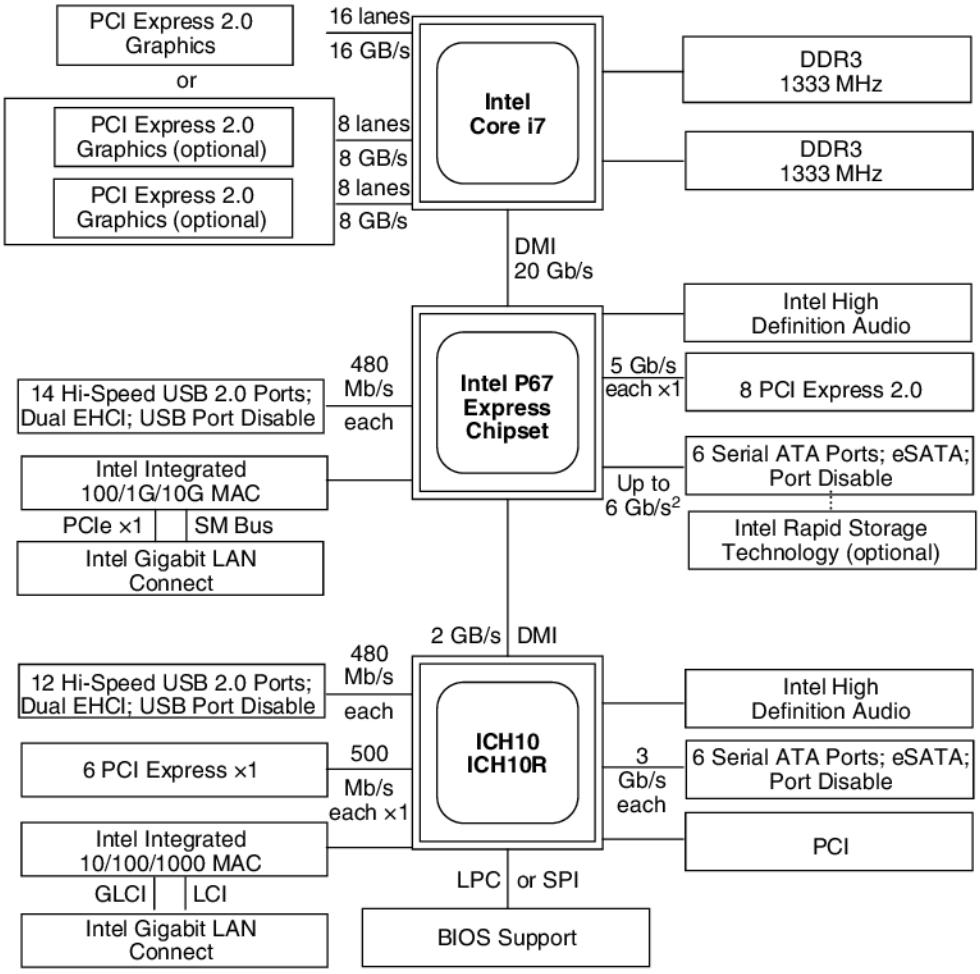


Figure 3-52. The bus structure of a modern Core i7 system.

1G Ethernet, USB ports, and old-style PCI Express and SATA interfaces. Newer systems may not incorporate the ICH10; it is required only if the system needs to support legacy interfaces.

PCI Bus Operation

Like all PC buses going back to the original IBM PC, the PCI bus is synchronous. All transactions on the PCI bus are between a master, officially called the **initiator**, and a slave, officially called the **target**. To keep the PCI pin count

down, the address and data lines are multiplexed. In this way, only 64 pins are needed on PCI cards for address plus data signals, even though PCI supports 64-bit addresses and 64-bit data.

The multiplexed address and data pins work as follows. On a read operation, during cycle 1, the master puts the address onto the bus. On cycle 2, the master removes the address and the bus is turned around so the slave can use it. On cycle 3, the slave outputs the data requested. On write operations, the bus does not have to be turned around because the master puts on both the address and the data. Nevertheless, the minimum transaction is still three cycles. If the slave is not able to respond in three cycles, it can insert wait states. Block transfers of unlimited size are also allowed, as well as several other kinds of bus cycles.

PCI Bus Arbitration

To use the PCI bus, a device must first acquire it. PCI bus arbitration uses a centralized bus arbiter, as shown in Fig. 3-53. In most designs, the bus arbiter is built into one of the bridge chips. Every PCI device has two dedicated lines running from it to the arbiter. One line, REQ#, is used to request the bus. The other line, GNT#, is used to receive bus grants. Note: REQ# is PCI-speak for $\overline{\text{REQ}}$.

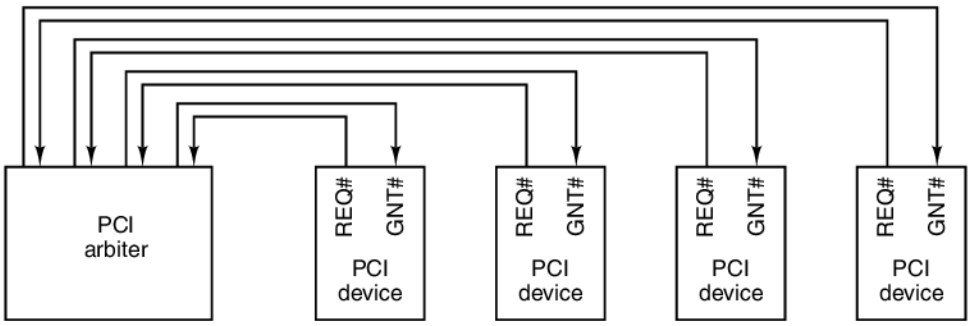


Figure 3-53. The PCI bus uses a centralized bus arbiter.

To request the bus, a PCI device (including the CPU) asserts REQ# and waits until it sees its GNT# line asserted by the arbiter. When that event happens, the device can use the bus on the next cycle. The algorithm used by the arbiter is not defined by the PCI specification. Round-robin arbitration, priority arbitration, and other schemes are all allowed. Clearly, a good arbiter will be fair, so as not to let some devices wait forever.

A bus grant is for only one transaction, although the length of this transaction is theoretically unbounded. If a device wants to run a second transaction and no other device is requesting the bus, it can go again, although often one idle cycle between transactions has to be inserted. However, under special circumstances, in

the absence of competition for the bus, a device can make back-to-back transactions without having to insert an idle cycle. If a bus master is making a very long transfer and some other device has requested the bus, the arbiter can negate the GNT# line. The current bus master is expected to monitor the GNT# line, so when it sees the negation, it must release the bus on the next cycle. This scheme allows very long transfers (which are efficient) when there is only one candidate bus master but still gives fast response to competing devices.

PCI Bus Signals

The PCI bus has a number of mandatory signals, shown in Fig. 3-54(a), and a number of optional signals, shown in Fig. 3-54(b). The remainder of the 120 or 184 pins are used for power, ground, and related miscellaneous functions and are not listed here. The *Master* (initiator) and *Slave* (target) columns tell who asserts the signal on a normal transaction. If the signal is asserted by a different device (e.g., CLK), both columns are left blank.

Let us now look briefly at each of the PCI bus signals. We will start with the mandatory (32-bit) signals, then move on to the optional (64-bit) signals. The CLK signal drives the bus. Most of the other signals are synchronous with it. A PCI bus transaction begins at the falling edge of CLK, which is in the middle of the cycle.

The 32 AD signals are for the address and data (for 32-bit transactions). Generally, during cycle 1 the address is asserted and during cycle 3 the data are asserted. The PAR signal is a parity bit for AD. The C/BE# signal is used for two different things. On cycle 1, it contains the bus command (read 1 word, block read, etc.). On cycle 2 it contains a bit map of 4 bits, telling which bytes of the 32-bit word are valid. Using C/BE# it is possible to read or write any 1, 2, or 3 bytes, as well as an entire word.

The FRAME# signal is asserted by the master to start a bus transaction. It tells the slave that the address and bus commands are now valid. On a read, usually IRDY# is asserted at the same time as FRAME#. It says the master is ready to accept incoming data. On a write, IRDY# is asserted later, when the data are on the bus.

The IDSEL signal relates to the fact that every PCI device must have a 256-byte configuration space that other devices can read (by asserting IDSEL). This configuration space contains properties of the device. The plug-and-play feature of some operating systems uses the configuration space to find out what devices are on the bus.

Now we come to signals asserted by the slave. The first of these, DEVSEL#, announces that the slave has detected its address on the AD lines and is prepared to engage in the transaction. If DEVSEL# is not asserted within a certain time limit, the master times out and assumes the device addressed is either absent or broken.

The second slave signal is TRDY#, which the slave asserts on reads to announce that the data are on the AD lines and on writes to announce that it is prepared to accept data.

Signal	Lines	Master	Slave	Description
CLK	1			Clock (33 or 66 MHz)
AD	32	×	×	Multiplexed address and data lines
PAR	1	×		Address or data parity bit
C/BE	4	×		Bus command/bit map for bytes enabled
FRAME#	1	×		Indicates that AD and C/BE are asserted
IRDY#	1	×		Read: master will accept; write: data present
IDSEL	1	×		Select configuration space instead of memory
DEVSEL#	1		×	Slave has decoded its address and is listening
TRDY#	1		×	Read: data present; write: slave will accept
STOP#	1		×	Slave wants to stop transaction immediately
PERR#	1			Data parity error detected by receiver
SERR#	1			Address parity error or system error detected
REQ#	1			Bus arbitration: request for bus ownership
GNT#	1			Bus arbitration: grant of bus ownership
RST#	1			Reset the system and all devices

(a)

Signal	Lines	Master	Slave	Description
REQ64#	1	×		Request to run a 64-bit transaction
ACK64#	1		×	Permission is granted for a 64-bit transaction
AD	32	×		Additional 32 bits of address or data
PAR64	1	×		Parity for the extra 32 address/data bits
C/BE#	4	×		Additional 4 bits for byte enables
LOCK	1	×		Lock the bus to allow multiple transactions
SBO#	1			Hit on a remote cache (for a multiprocessor)
SDONE	1			Snooping done (for a multiprocessor)
INTx	4			Request an interrupt
JTAG	5			IEEE 1149.1 JTAG test signals
M66EN	1			Wired to power or ground (66 MHz or 33 MHz)

(b)

Figure 3-54. (a) Mandatory PCI bus signals. (b) Optional PCI bus signals.

The next three signals are for error reporting. The first of these is STOP#, which the slave asserts if something disastrous happens and it wants to abort the current transaction. The next one, PERR#, is used to report a data parity error on the previous cycle. For a read, it is asserted by the master; for a write it is asserted by the slave. It is up to the receiver to take the appropriate action. Finally, SERR# is for reporting address errors and system errors.

The REQ# and GNT# signals are for doing bus arbitration. These are not asserted by the current bus master, but rather by a device that wants to become bus master. The last mandatory signal is RST#, used for resetting the system, either due to the user pushing the RESET button or some system device noticing a fatal error. Asserting this signal resets all devices and reboots the computer.

Now we come to the optional signals, most of which relate to the expansion from 32 bits to 64 bits. The REQ64# and ACK64# signals allow the master to ask permission to conduct a 64-bit transaction and allow the slave to accept, respectively. The AD, PAR64, and C/BE# signals are just extensions of the corresponding 32-bit signals.

The next three signals are not related to 32 bits vs. 64 bits, but to multiprocessor systems, something that PCI boards are not required to support. The LOCK signal allows the bus to be locked for multiple transactions. The next two relate to bus snooping to maintain cache coherence.

The INTx signals are for requesting interrupts. A PCI card can have up to four separate logical devices on it, and each one can have its own interrupt request line. The JTAG signals are for the IEEE 1149.1 JTAG testing procedure. Finally, the M66EN signal is either wired high or wired low, to set the clock speed. It must not change during system operation.

PCI Bus Transactions

The PCI bus is really very simple (as buses go). To get a better feel for it, consider the timing diagram of Fig. 3-55. Here we see a read transaction, followed by an idle cycle, followed by a write transaction by the same bus master.

When the falling edge of the clock happens during T_1, the master puts the memory address on AD and the bus command on C/BE#. It then asserts FRAME# to start the bus transaction.

During T_2, the master floats the address bus to let it turn around in preparation for the slave to drive it during T_3. The master also changes C/BE# to indicate which bytes in the word addressed it wants to enable (i.e., read in).

In T_3, the slave asserts DEVSEL# so the master knows it got the address and is planning to respond. It also puts the data on the AD lines and asserts TRDY# to tell the master that it has done so. If the slave were not able to respond so quickly, it would still assert DEVSEL# to announce its presence but keep TRDY# negated until it could get the data out there. This procedure would introduce one or more wait states.

In this example (and often in reality), the next cycle is idle. Starting in T_5 we see the same master initiating a write. It starts out by putting the address and command onto the bus, as usual. Only now, in the second cycle it asserts the data. Since the same device is driving the AD lines, there is no need for a turnaround cycle. In T_7, the memory accepts the data.

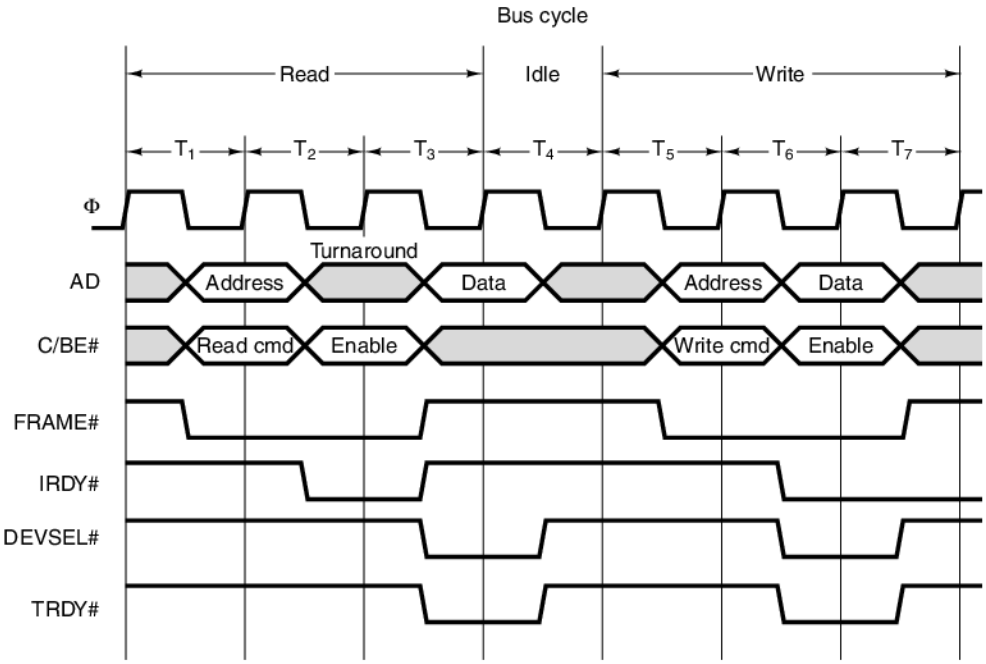


Figure 3-55. Examples of 32-bit PCI bus transactions. The first three cycles are used for a read operation, then an idle cycle, and then three cycles for a write operation.

3.6.2 PCI Express

Although the PCI bus works adequately for most current applications, the need for greater I/O bandwidth is making a mess of the once-clean internal PC architecture. In Fig. 3-52, it is clear that the PCI bus is no longer the central element that holds the parts of the PC together. The bridge chip has taken over part of that role.

The essence of the problem is that increasingly many I/O devices are too fast for the PCI bus. Cranking up the clock frequency on the bus is not a good solution because then problems with bus skew, crosstalk between the wires, and capacitance effects just get worse. Every time an I/O device gets too fast for the PCI bus (like the graphics card, hard disk, network, etc.), Intel adds a new special port to the bridge chip to allow that device to bypass the PCI bus. Clearly, this is not a long-term solution either.

Another problem with the PCI bus is that the cards are quite large. Standard PCI cards are generally 17.5 cm by 10.7 cm and low-profile cards are 12.0 cm by 3.6 cm. Neither of these fit well in laptop computers and and certainly not in mobile devices. Manufacturers would like to produce even smaller devices. Also,

some companies like to repartition the PC, with the CPU and memory in a tiny sealed box and the hard disk inside the monitor. With PCI cards, doing this is impossible.

Several solutions have been proposed, but the one that won a place in all modern PCs today (in no small part because Intel was behind it) is called **PCI Express**. It has little to do with the PCI bus and in fact is not a bus at all, but the marketing folks did not like letting go of the well-known PCI name. PCs containing it are now the standard. Let us now see how it works.

The PCI Express Architecture

The heart of the PCI Express solution (often abbreviated PCIe) is to get rid of the parallel bus with its many masters and slaves and go to a design based on high-speed point-to-point serial connections. This solution represents a radical break with the ISA/EISA/PCI bus tradition, borrowing many ideas from the world of local area networking, especially switched Ethernet. The basic idea comes down to this: deep inside, a PC is a collection of CPU, memory, and I/O controller chips that need to be interconnected. What PCI Express does is provide a general-purpose switch for connecting chips using serial links. A typical configuration is illustrated in Fig. 3-56.

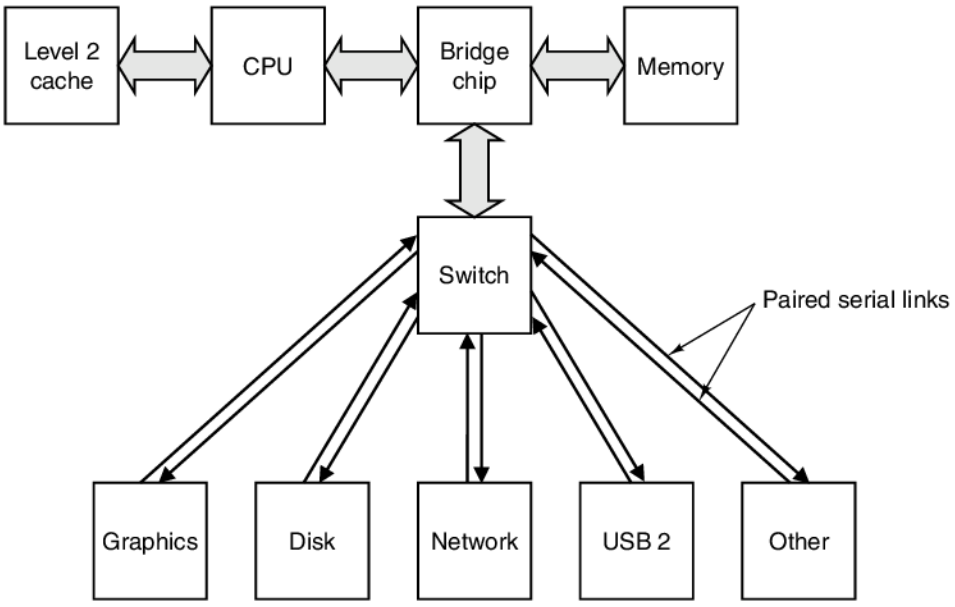


Figure 3-56. A typical PCI Express system.

As illustrated in Fig. 3-56, the CPU, memory, and cache are connected to the bridge chip in the traditional way. What is new here is a switch connected to the

bridge (possibly part of the bridge chip itself or integrated directly into the processor). Each I/O chip has a dedicated point-to-point connection to the switch. Each connection consists of a pair of unidirectional channels, one to the switch and one from it. Each channel is made up of two wires, one for the signal and one for ground, to provide high noise immunity during high-speed transmission. This architecture has replaced the old one with a much more uniform model, in which all devices are treated equally.

The PCI Express architecture differs from the old PCI bus architecture in three key ways. We have already seen two of them: a centralized switch vs. a multidrop bus and a the use of narrow serial point-to-point connections vs. a wide parallel bus. The third difference is more subtle. The conceptual model behind the PCI bus is that of a bus master issuing a command to a slave to read a word or a block of words. The PCI Express model is that of a device sending a data packet to another device. The concept of a **packet**, which consists of a header and a payload, is taken from the networking world. The **header** contains control information, thus eliminating the need for the many control signals present on the PCI bus. The **payload** contains the data to be transferred. In effect, a PC with PCI Express is a miniature packet-switching network.

In addition to these three major breaks with the past, there are also several minor differences as well. Fourth, an error-detecting code is used on the packets, providing a higher degree of reliability than on the PCI bus. Fifth, the connection between a chip and the switch is longer than it was, up to 50 cm, to allow system partitioning. Sixth, the system is expandable because a device may actually be another switch, allowing a tree of switches. Seventh, devices are hot pluggable, meaning that they can be added or removed from the system while it is running. Finally, since the serial connectors are much smaller than the old PCI connectors, devices and computers can be made much smaller. All in all, PCI Express is a major departure from the PCI bus.

The PCI Express Protocol Stack

In keeping with the model of a packet-switching network, the PCI Express system has a layered protocol stack. A **protocol** is a set of rules governing the conversation between two parties. A protocol stack is a hierarchy of protocols that deal with different issues at different layers. For example, consider a business letter. It has certain conventions about the placement and content of the letterhead, the recipient's address, the date, the salutation, the body, the signature, and so on. This might be thought of as the letter protocol. In addition, there is another set of conventions about the envelope, such as its size, where the sender's address goes and its format, where the receiver's address goes and its format, where the stamp goes, and so on. These two layers and their protocols are independent. For example, it is possible to reformat the letter but use the same envelope or vice versa. Layered