Patricio Bulić

Basic Components Of Computer Systems

A Textbook

November 24, 2021

Springer

Contents

Chapter 1
Main memory

Chapter goals

In this chapter, we will cover the modern memory design and operations in memory chips and modules that enable efficient data transfer between the memory controller and so called DIMMS, i.e., memory modules used in modern computer systems. To understand the organization and operation of modern memory chips fully, we need to start with some fundamental digital building blocks. Then, we gradually build memory components, arrays, operations inside the memory chips, timings and the techniques to boost the performance of memory chips. At the end of the chapter, you should fully understand modern DDR SDRAM chips, the DDR memory technology, memory timings, DIMM modules and multi-channel architecture.
From this chapter, you should gain a basic understanding of the design and operation of computer memory and storage circuits including:

- Static memory circuits using the six-transistor cell
- Dynamic memory circuits including the one-transistor cell
- Sense amplifier circuits required to detect the information stored in the memory cells
- Sense amplifier circuits required to detect the information stored in the memory cells
- Overall DRAM memory chip organization

1.1 Introduction

We are now already familiar with CPUs and caches. In this chapter, we focus on the main memory used in modern computer systems as one in Figure 1.1. Figure 1.1 illustrates the memory hierarchy in the Intel i7-860 based system. Intel i7-860 is an out-of-order execution processor that includes four cores. The L1 and L2 caches are separate for each core, while the L3 cache is shared among the cores on a chip. The L1 cache is the 32 KB, four-way set-associative cache. There are two L1 caches per core: instruction (I) and data (D) . The L2 cache is the 256 KB, eight-way set-associative cache. Finally, the L3 cache is the 8 MB, 16-way set-associative cache.

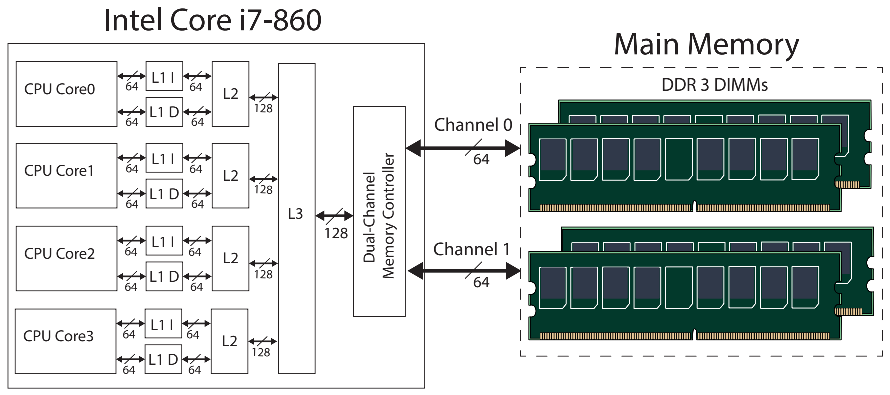


Fig. 1.1: Intel i7-860 memory hierarchy

A CPU core directly accesses only its L1 cache. If a hit in L1 occurs, the data is returned after an initial latency of 4 cycles. If the L1 cache misses, the L2 cache is accessed. If a hit in L2 occurs, the block of size 64B is returned after an initial latency of 10 cycles at a rate of 8 bytes per clock cycle. If the L2 cache misses, the L3 cache is accessed. If a hit occurs in L3, the 64-byte block is returned after an initial latency of 35 cycles at a rate of 16 bytes per clock. If L3 misses, memory access is initiated - the on-chip memory controller must get the block of size 64B from the main memory.

The main memory is implemented of DDR3 memory chips placed on the printed circuit boards called Dual In-Line Memory Module (DIMM). The memory controller on i7-800 supports two 64-bit memory channels. Each channel is used to access eight 8-bit memory chips placed on one side of DIMM (64 bits per access). Two 64-bit memory channels are used simultaneously as one 128-bit channel (since there is only one memory controller, and the same address of the missing block in L3 is sent on both channels) to fill the missing block in L3. Thus, the memory controller fills the 64-byte cache block at a rate of 16 bytes (124 bits) per memory clock cycle.

Have you struggled reading the description of the memory hierarchy in the Intel i7-860 based system? Don't worry, at the end of this chapter you should be able to understand it. Let us now begin our journey into the world of modern memory.

1.2 Basics of Digital Circuits: A Quick Review

Before looking under the hood of modern memory chips used in the computer systems, we should apprehend some basic concepts from digital electronics like MOS transistors used as logical switches and MOS inverters. The aim is to understand the operations in modern memory chips and not to fall into the physical equations of electronic circuits. Therefore, the description of the basic concepts of digital circuits will be significantly simplified.

The basic building block of all digital circuits is the MOS transistor. MOS is an acronym for Metal-Oxid-Semiconductor and indicates the manufacturing process used to make transistors. The MOS transistor has three terminals: gate (G), drain

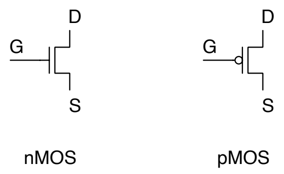


Fig. 1.2: nMOS nad pMOS transistor symbols.

(D) and source (S). The gate terminal is a control input: it controls the flow of electrical current between the source and drain terminals. There are two types of MOS transistors: nMOS and pMOS. Figure 1.2 shows the symbols of both types of MOS transistors. We will consider only the type of operation where MOS transistors act as logical switches.

1.2.1 MOS transistor as a switch

Consider first an nMOS transistor. If the gate terminal is grounded (logical 0), no current flows between drain and source. Hence, we say the transistor is OFF. If the gate voltage is high and corresponds to logic 1, a conducting path of electrons is formed from source to drain, and current can flow. We say the transistor is ON.

The reverse holds for a pMOS transistor. When the gate is at a positive voltage that corresponds to logic 1, no current flow, so the transistor is OFF. A sufficiently low gate voltage that corresponds to logic 0 forms a conducting path from source to drain, so the transistor is ON.

In summary, the gate of a MOS transistor controls the flow of current between the source and drain. Simplifying this to the extreme allows us to **view the MOS transistors as ON/OFF switches**. When the gate of an nMOS transistor is 1, the transistor is ON, and the current flow between source to drain. When the gate is 0, the nMOS transistor is OFF, and no current flows between source to drain. A pMOS

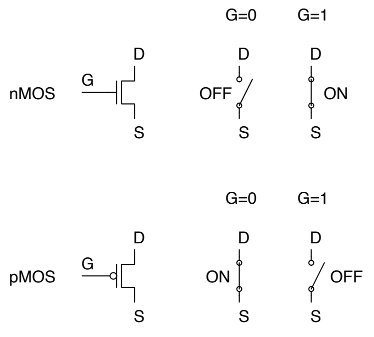


Fig. 1.3: Switch-level models of nMOS nad pMOS transistors.

transistor is just the opposite, being ON when the gate is low and OFF when the gate is high. Figure 1.3 illustrates this switch model.

1.2.2 CMOS inverter

The most straightforward logic gates that can be built using MOS transistors are an inverters. An inverter is built from two **complementary** MOS transistors, one nMOS, and one pMOS, hence the name *complementary MOS (CMOS) inverter*. Figure 1.4 shows the schematic and the switch-level model for a CMOS inverter or NOT gate using one nMOS transistor and one pMOS transistor. The bar at the top of the schematic indicates a supply voltage (Vdd), and the triangle at the bottom indicates the ground terminal (GND). The input IN connects both transistors' gates. When the input IN is 0, the nMOS transistor is OFF, and the pMOS transistor is ON. Thus, the output OUT is pulled to logic 1 because it is connected to Vdd through the pMOS transistor. Conversely, when IN is 1, the nMOS is ON, the pMOS is OFF, and OUT is pulled down to '0', because it is connected to GND through the nMOS transistor.

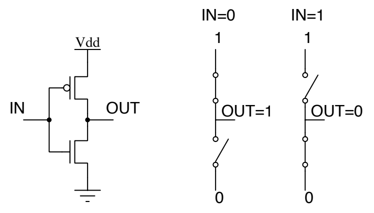


Fig. 1.4: CMOS inverter and its switch-level models.

1.2.3 Bistable element

Now, as we are familiar with MOS transistors and CMOS inverter, it is time to learn how we can store one bit of information in a MOS digital circuit, i.e., how to form a 1-bit storage (memory) cell using MOS transistors and inverters. The fundamental building block of memory is a **bistable element** - a logic element with two stable states. Figure 1.5 shows the bistable element composed of two inverters, I1 and I2. The inverters are cross-coupled, meaning that the input of I1 is the output of I2 and vice versa.

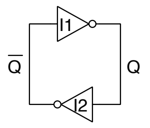


Fig. 1.5: A bistable element.

If $Q = 0$, I2 receives a FALSE input, so it produces a TRUE output on $\overline{Q}$. I1 receives a TRUE input, so it produces a FALSE output on Q. This is consistent with the original assumption that Q=0, so the circuit is in the stable state. If $Q = 1$, I2 receives a TRUE input, so it produces a FALSE output on $\overline{Q}$. I1 receives a FALSE input, so it produces a TRUE output on Q. This is consistent with the original assumption that Q=1, so the circuit is again in the stable state. Because the cross-coupled inverters have two stable states, 0 and 1, the circuit is said to be **bistable**. The state of the cross-coupled inverters is contained in one binary state variable, Q. Specifically, if $Q = 0$, it will remain 0 forever, and if $Q = 1$, it will remain 1 forever. Although the cross-coupled inverters can store a bit of information, they are not practical because the user has no inputs to control the state. So, we have to expand the bistable element with a circuitry, which provides inputs to control the value of the state variable. One such element that can accept the inputs to control the value stored in the bistable is a *static RAM cell*.

Summary: Transistors, Inverter and Bistable

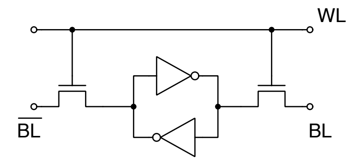
The gate of a MOS transistor controls the flow of current between the source and drain. Simplifying this to the extreme allows us to *view the MOS transistors as ON/OFF switches*.

An inverter is built from two *complementary* MOS transistors, one nMOS, and one pMOS.

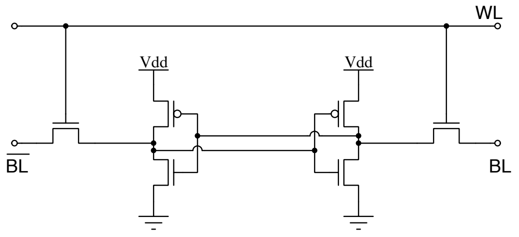
The fundamental building block of memory is a *bistable element*. It is composed of two cross-coupled inverters. It stores one bit of information.

1.3 SRAM cell

Static random-access memory (static RAM or SRAM) is a type of random-access memory (RAM) that uses a bistable element to store one bit of information. This is the type of memory used as the building block of most caches because of its superior performance over other memory structures, specifically DRAM, which we will cover later. SRAM is faster and more expensive than DRAM; it is typically used for CPU cache and registers while DRAM is used for a computer's main memory.



(a) A basic structure of a SRAM cell.



(b) 6-transistor SRAM cell.

Fig. 1.6: SRAM cell.

A typical SRAM cell is made up of six MOS transistors - two complementary pairs that form two cross-coupled inverters (bistable), and two *access* nMOS transistors that serve as a switch used to control the state of the bistable element during the read and write operations. Figure 1.6 shows an SRAM cell. Each bit in an SRAM cell is stored in the bistable element composed of four transistors that form two cross-coupled inverters. As we have already learned, this cross-coupled connection creates regenerative feedback that allows it to *store a single bit of data indefinitely* provided that power is supplied to the SRAM cell. The SRAM cell also has two bit lines that control both the input and output of the data from the cell. The first bit line (BL), holds the same value that is stored in the cell. The second bit line ($\overline{\text{BL}}$) holds the inverse of the value that is stored in the cell.

When the word line (WL) is not selected (WL=0), the cell is in standby mode. Setting the word line to a logic high enables the access nMOS transistors. This connects the cell with both bit lines and allows the cells to be read or written. The SRAM cell is read by asserting a WL and detecting the voltage difference at the bit lines BL and $\overline{\text{BL}}$. The SRAM cell is written by setting the content on the bit lines BL and $\overline{\text{BL}}$ and asserting the word line.

Due to the ability to store the information indefinitely and the high speed of SRAM cells, they are used to implement caches and registers in microprocessors. Furthermore, the main advantage of SRAM is that it uses the same fabrication process as the microprocessor core, simplifying the integration of cache and CPU registers onto the processor die. On the other hand, the main **disadvantages of SRAM cells are price, low density, and high operational power consumption**. These disadvantages prevent the usage of SDRAM cells in the main computer memory.

Since SRAM cells are not used to build the main memory, we will end up dealing with and learning about SRAM cells at this point, and we are now going to deep dive into DRAM cells. By contrast, DRAM typically uses a different process that is not optimal for logic circuits, making the integration of CPU logic and DRAM harder than the integration of CPU logic and SRAM. But DRAMs are smaller, cheaper, and consume less power, which makes them the better candidate for implementing the main memory.

Summary: SRAM cell

A *SRAM cell* uses a bistable element to store one bit of information. It is made up of a bistable and two access nMOS transistors that serve as a switch used to control the state of the bistable element during the read and write operations.

Due to the ability to store the information indefinitely and the high speed of SRAM cells, they are used to implement caches and registers in microprocessors.

1.4 DRAM cell

Dynamic Random Access Memory (DRAM) is the main memory used for all computers. To pack more bits per chip, a DRAM cell consists only of a single MOS transistor (T) and a storage capacitor (C) as shown in Figure 1.7. The data in the

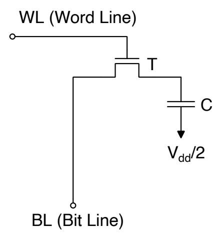


Fig. 1.7: A DRAM cell.

cell can be read or written through the bit line (BL) terminal. In contrast to SRAMs, DRAMs store their contents as a charge on a capacitor C. This way, the DRAM cell is substantially smaller than the SRAM cell. The transistor T acts as a switch between the storage capacitor and the bit line. The word line (WL) terminal is used to switch on/off the transistor T. Reading the bit from the DRAM cell discharges the capacitor and thus destroys the information. Even if we do not read the DRAM cell, the charge leaks from the capacitor because the cell transistor does not entirely disconnect the storage capacitor from the bit line. Even though the transistor is switched off, a tiny current flows from the capacitor to the bit line and discharges the capacitor. Therefore, the charge (information) must be refreshed several times each second. Hence the name *dynamic*.

1.4.1 Basic operation of DRAM

The transistor T acts as a switch between the storage capacitor C and the bit line BL. One node of the capacitor is connected to $V_{dd}/2$. The voltage across the capacitor is either $+V_{dd}/2$, if the capacitor stores "1", or $-V_{dd}/2$, if the capacitor stores "0". The charge stored in a capacitor is equal to capacitance times voltage across the capacitor:

$$Q = C \times V_{dd}/2 \; . \tag{1.1}$$

In a 90 nm DRAM process technology, the capacitance of a DRAM storage cell is 30 fF. If we assume $V_{dd} = 3.3V$, then

$$Q = 30fF \times 3.3V/2 = 34.5fC \; .$$

As you may recall from physics class, one electron equals to a charge of $1.6 \cdot 10^{-19}C$, thus the storage capacitor stores only 210000 electrons! Even though the transistor has a very high resistance when switched-off, the charge on the capacitor leaks away through switched off transistor in tens to hundreds of milliseconds. Storage cells should be regularly refreshed to avoid loss of data.

The data is written into a memory cell by placing the "1" or "0" charge into the storage capacitor. To write data into a cell, we first set the bit line to Vdd ("1") or to GND ("0") and assert the word line to connect the capacitor to the bit line. The storage capacitor then retains the stored charge after the word line is de-asserted, and the transistor is turned off. The electric charge on the storage capacitor slowly leaks off, so without intervention, the data on the chip would soon be lost. This capacitor will be accessed for either a new write, a read, or a refresh.

To read data from the cell, the bit line is first precharged to $V_{dd}/2$. The word line is then driven high to connect a cell's storage capacitor to its bit line. This causes the transistor to conduct, transferring charge from the storage cell to the connected bit line (if the stored value is "1") or from the connected bit-line to the storage cell (if the stored value is "0"). This process is depicted in Figure 1.14. In both cases, information stored in the DRAM cell is lost. Thus, reading from

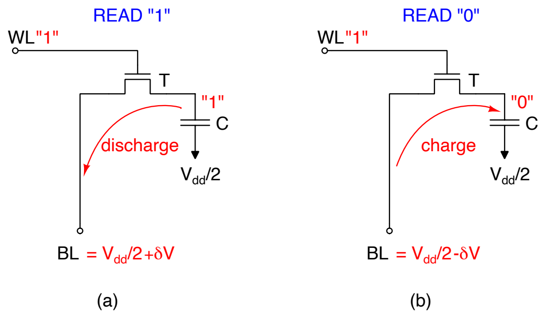


Fig. 1.8: Reading from a DRAM cell. (a) Reading "1" from a DRAM cell discharges the storage capacitor and slightly increases the voltage of the bit line. (b) Reading "0" from a DRAM cell charges the storage capacitor and slightly decreases the voltage of the bit line. In both cases, information is lost.

DRAM is a **destructive operation**. The bit lines are relatively long because they

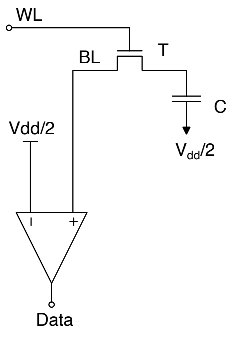


Fig. 1.9: A DRAM cell with a sense amplifier.

connect storage cells in all memory words, and they act as a capacitor with relatively high capacitance (the capacitance of the bit lines is ten times the capacitance of the storage capacitor). According to the charge-sharing equation (capacitive voltage divider), the voltage swing (the magnitude of a voltage difference) δV on the bit line during readout is

$$\delta V = \frac{V_{dd}}{2} \frac{C}{C + C_{BL}} , \tag{1.2}$$

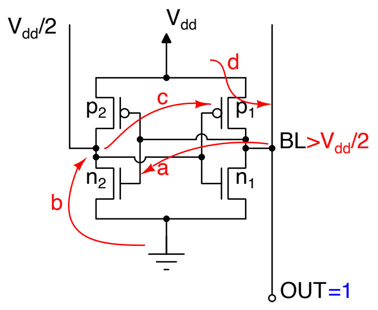
where C is the capacitance of the storage capacitor and C_{BL} is the capacitance of the bit line. If the capacitance of the bit line is ten times the capacitance of the storage capacitor and $V_{dd} = 3.3V$, the voltage difference δV on the bit line during the read operation is only 150 mV! When dealing with such tiny voltage swing,

correctly detecting the bit value is quite a challenge. Thus, we need a special circuit to sense this small voltage swing. Sensing is necessary to read the cell data properly. A special circuit used to detect the voltage swing and read the data is a **sense amplifier**.

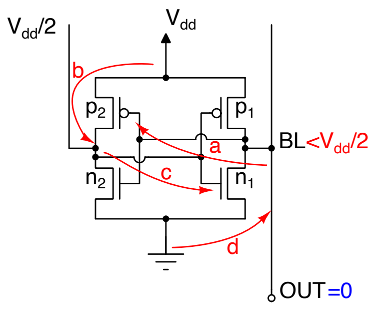
To sense the voltage swing on the bit line, a sense amplifier is used, as presented in Fig 1.9. A sense amplifier has two inputs. One input is connected to the bit line, and the other input is tied to $V_{dd}/2$. The sense amplifier detects the voltage difference at its inputs and outputs 0 at the Data terminal if the voltage on the bit line is less than $V_{dd}/2$, or 1 otherwise.

1.4.2 Basic operation of sense amplifiers

A sense amplifier is a simple circuit made up of two cross-coupled CMOS inverters - so it is a SRAM cell. Figure 1.10 shows a sense amplifier built from cross-coupled CMOS inverters. Initially, the bit line (BL) is precharged to $V_{dd}/2$. During a read, the bit line changes its voltage by a small amount, δV. If the voltage of the bit line is higher than $V_{dd}/2$ (Figure 1.10a), the `n2` nMOS transistor begins to conduct and pulls the precharged line down to "0". This, in turn, causes the `p1` pMOS transistor to conduct. After a small delay, BL is pulled high, and OUT=1. On the other hand, if the voltage of the bit line is lower than $V_{dd}/2$ (Figure 1.10b), the `(p2)` pMOS transistor begins to conduct and pulls the precharged line up to "1". This, in turn, causes the `n1` nMOS transistor to conduct. After a small delay, BL is pulled down to "0", and OUT=0. The feedback that occurs from the cross-connected inverters



(a) Sensing "1".



(b) Sensing "0".

Fig. 1.10: A simplified structure and operation of a sense amplifier. (a) Sensing "1". (b) Sensing "0".

thereby amplifies the small voltage difference between the BL and the precharged input reference until the bit line is entirely at the lowes or the highest voltage.

We have just learned that the main function of sense amplifiers is to sense the tiny voltage swing on the bit lines that occurs when an access transistor is turned on and a storage capacitor places its charge on the bit line. The second function of sense amplifiers is to restore the value of cells after the voltage on the bit lines is sensed. Recall that turning on the access transistor allows a storage capacitor to share its stored charge with the bit line. However, the process of sharing the charge from a storage cell discharges that storage cell. Thus, the information in the cell is lost and cannot be read again. But this information is stored in the sense amplifier, as the sense amplifier is a bistable circuit made up of two cross-coupled inverters. As such, it can store information as long the supply voltage is present. Consequently, after sensing, the sense amplifier is used to write back the the bit value to the storage cell. This operation is referred to as **(row) precharge**.

Summary: DRAM cell

Dynamic Random Access Memory (DRAM) is the main memory used for all computers. DRAMs store their contents as a charge on a capacitor. A DRAM cell consists only of a storage capacitor and a single nMOS transistor that acts as a switch between the storage capacitor and the bit line.

Reading from a DRAM cell is a *destructive operation*. Besides, the charge on the capacitor leaks away through switched off transistor in tens to hundreds of milliseconds. Thus DRAMs should be regularly *refreshed*.

A sense amplifier is a special circuit used to detect the tiny voltage swing on the bit line and read the data. The sense amplifier is also used to write back the bit value to the storage cell. This operation is referred to as *precharge*.

1.5 DRAM Arrays and DRAM Banks

DRAM is usually arranged in a rectangular **memory array** of storage cells organized into rows and columns. Figure 1.11 shows a simplified basic structure of a DRAM cell array containing R-by-C cells. **DRAM arrays usually contain many hundreds or thousands of cells in height and width**. The cells of a DRAM are accessed by a **row address** and a **column address**. The rows address lines (i.e., the word lines) are connected to the gates of the nMOS transistors, and the column lines are connected to the sense amplifiers.

The array size represents a trade-off between density and performance. Larger arrays contain more bits of information, but they also require longer word lines and bit lines. Longer word and bit lines have a higher capacitance. An array that contains thousands of cells in height and width has an order of magnitude higher capacitance on the bit line than in the cell, so the bit line voltage swing δV during a read is tiny,

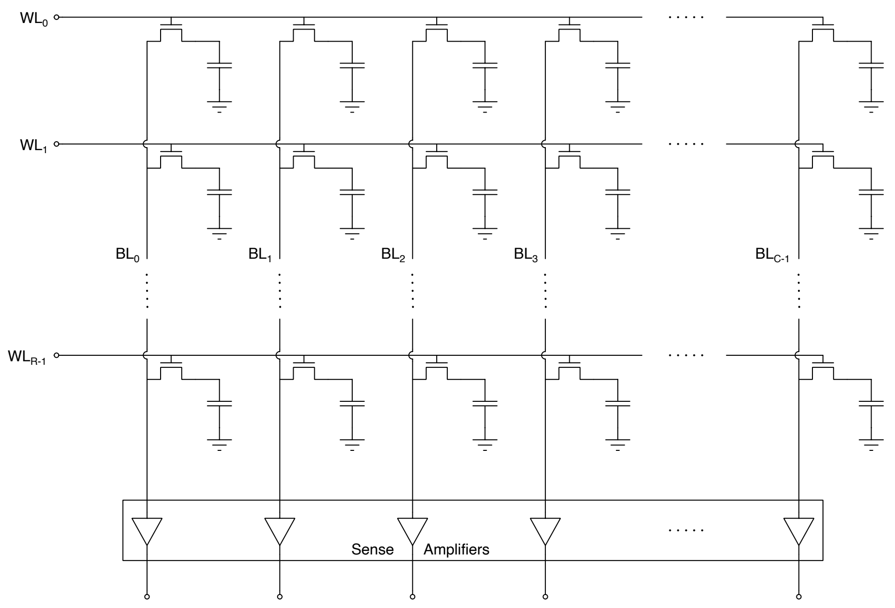


Fig. 1.11: A simplified structure of a DRAM array.

which is hard to detect. Besides, due to a higher capacitance, larger arrays are slow. A typical array size in a recent DRAM is 32K words (rows) by 1024 bits (columns).

A DRAM memory chip can have 4-16 DRAM arrays that are accessed simultaneously, and transmits or receives a number of bits equal to the number of arrays each time the memory controller accesses the DRAM. Each array provides a single bit to the output pin. DRAM chips are described as xN, where N refers to the number of memory arrays and output pins. For example, in a simple organization, a x8 DRAM (pronounced "by eight") indicates that the DRAM has at least eight memory arrays and that a column width is 8 bits (each column read or write access transmits 8 bits of data). This means that the DRAM transmits or receives eight bits each time the memory controller accesses the DRAM. A set of memory arrays accessed simultaneously is referred to as a **bank**.

Summary: DRAM Arrays and DRAM Banks

DRAM is arranged in a rectangular *memory array* of storage cells organized into rows and columns.

The cells of a DRAM are accessed by a *row address* and a *column address*.

A **bank** is a set of N memory arrays accessed simultaneously, forming an N-bit width column. Usually, there are 4, 8, or 16 DRAM arrays in a bank.

1.6 DRAM Chips

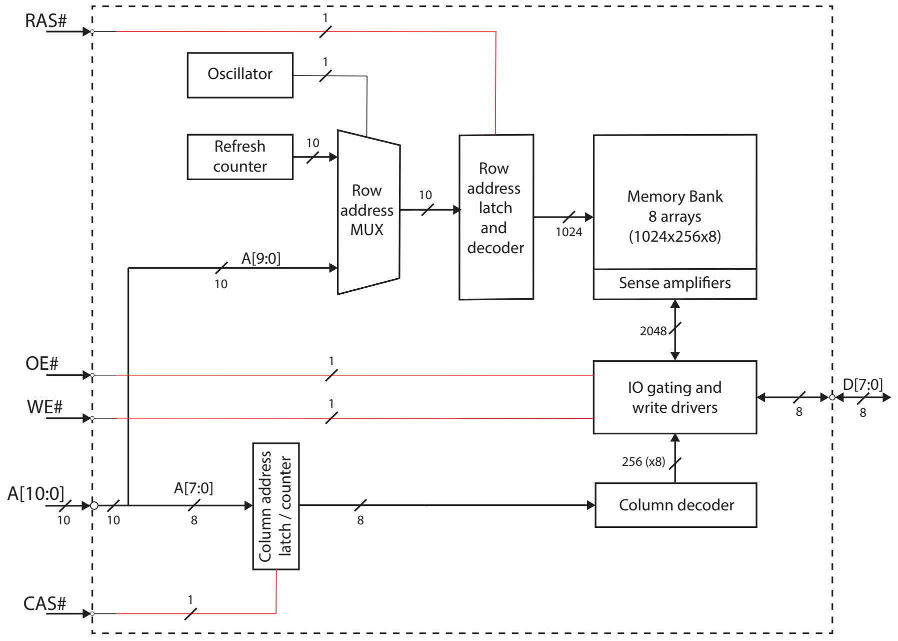


Fig. 1.12: Simplified structure of a 256K × 8-bit DRAM chip.

Figure 1.12 presents the basic structure of a DRAM chip. As we have learned, the DRAM memory is organized as a rectangular matrix of rows and columns. The DRAM chip in Figure 1.12 contains a bank of 8 arrays. Each array has 1024-by-256 storage cells. All arrays in a bank are accessed at the same time, so the DRAM chip in Figure 1.12 reads or transmits eight bits in a single access (D0 to D7). The components identifying the row and column are referred to as the **row address decoder** and the **column selector**. The row address decoder is used to activate the appropriate word line from the given row address. The column selector is used to select the appropriate column from the given column address.

As the capacity of DRAMs is large, the DRAM chips would require a large number of address lines to address a row and a column. For example, to address a cell in a 32256-by-1024 array, we need 15 bits to select a word and 10 bits to select a column. Such a large number of address bits could be an issue. The solution is to **multiplex the address lines**. Firstly, the row address is applied to the address lines, then the column address follows. In such a way, the number of address pins is cut almost in half. The same holds for DRAM in Figure 1.12. Instead of having 18 address bits (10 for the row and 8 for the column), only 10 address bits are used. To indicate which of two addresses is currently on the bus, we need **two additional control signals**: the **row access strobe (RAS)** and the **column access strobe (CAS)**. When the RAS signal is activated, the address bits A0 to A9 are latched into the **row address latch**. Similarly, when the CAS signal is activated, the address bits A0 to A7 are latched into the **column address latch**.

Two more control signals are required to appropriate transfer data into and from a DRAM chip. The **write enable (WE)** signal is used to choose a read operation or a write operation. A low voltage level signifies that a write operation is desired; a high voltage level is used to choose a read operation. During a read operation, the **output enable (OE)** signal is used to prevent data from appearing at the output until needed. When OE is low, data appears at the data outputs as soon as it is available. OE is kept high during a write operation. Figure 1.13 illustrates a pinout diagram of a 256K $\times$ 8-bit DRAM from Figure 1.12.

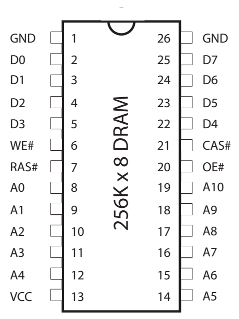


Fig. 1.13: 256K $\times$ 8-bit DRAM chip pinout.

Summary: DRAM Chips

DRAM chips contain at least one memory bank. The *row address decoder* is used to activate the appropriate word line from the given row address. The *column selector* is used to select the proper column from the given column address.

As the number of address bits required to select rows and columns can be quite large, the *address lines are multiplexed*. To indicate which of two addresses is currently on the bus, we need two additional control signals: the *row access strobe (RAS)* and the *column access strobe (CAS)*.

The *write enable (WE)* signal is used to choose a read or a write operation. During a read operation, the *output enable (OE)* signal is used to prevent data from appearing at the output until needed.

FALLACY: Memories (DRAMs) are physically organized as a liner vector of memory words.

It is a common and erroneous belief that memory is physically organized as a vector of memory words (and not as a rectangular array of rows and columns). Such an organization of memory would otherwise be ideal. A memory array would be just one long vector of memory cells, and there would be only one memory cell in a word. All memory cells would then be connected to the same bit-line. In that case, a DRAM array would contain R-by-1 memory cells. The memory with 8-bit words would then be composed of eight parallel R-by-1 memory arrays. In this case, the row address would already be the column address, because there would be only one column in a row. The memory addresses would not be multiplexed, and we would not need the RAS and CAS signals. Wouldn't that be great? However, it is physically impossible to make such memory because, in such memory, the bit lines would be extremely long and would have huge capacitance. The capacitance of such long bit lines would probably be several thousand times greater than the capacitance of the memory cells, and it would be impossible to detect a tiny voltage swing.

1.7 Basic DRAM operations and timings

The most challenging aspect when working with DRAMs is resolving the timing requirements. **DRAMs are generally asynchronous, responding to input signals whenever they occur**. As long as the signals are applied in the proper sequence, with signal durations and delays between signals that meet the specified limits, the DRAM works properly. The following signals control the DRAM operations:

1. Row Address Strobe (RAS). RAS is active low. To enable RAS, a transition from a high voltage to a low voltage, is required. The voltage must remain low until RAS is no longer needed. During a complete memory cycle, there is a minimum amount of time that RAS must be active (t_{RAS}). There is a minimum amount of time that RAS must be inactive before activating it again, called the RAS precharge time (t_{RP}). t_{RP} tells us how fast the row can be precharged before we can engage another RAS.

2. Column Address Strobe (CAS). CAS is used to latch the column address and to initiate the read or write operation. It is active low. The memory specification lists the minimum amount of time CAS must remain active (t_{CAS}). For most memory operations, there is also a minimum amount of time that CAS must be inactive before activating it again, called the CAS precharge time (t_{CP}).

3. Write Enable (WE). The write enable signal is used to choose a read operation or a write operation. It is active low.

4. Output Enable (OE). It is active low. When OE is low during a read operation, data appears at the data outputs as soon as it is available. During a write operation, OE should be high.
5. Address. The addresses are used to select a memory location on the chip. The address pins on a memory device are used for both row and column selection (multiplexing).
6. Data In or Out. The data pins on the DRAM memory device are used for data input and output. During a write operation, data at data pins are stored in the selected memory cells. During a read operation, data from the selected memory cells appear at the data once access is complete, and OE is low.

1.7.1 *Reading data from DRAM memory*

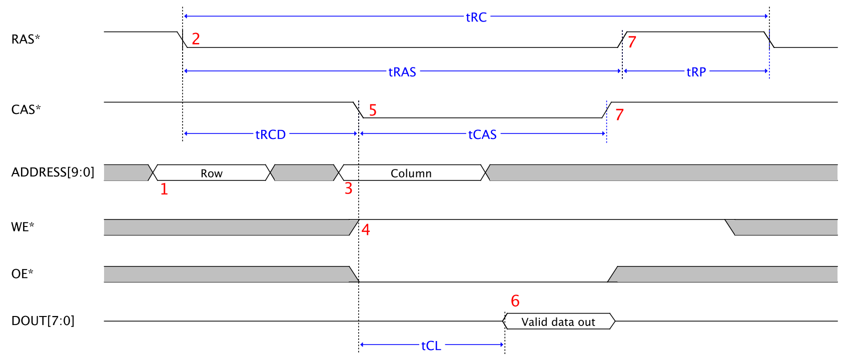


Fig. 1.14: Simplified DRAM read cycle.

To read the data from a DRAM memory cell, we must select the DRAM memory cell by applying its row and column addresses to the address input pins. The charge on the selected DRAM cell must then be sensed by the sense amplifier and sent to the data output (pins). In terms of timing, the following steps must occur:

1. The row address must be applied to the address input pins on the memory device before RAS goes low.
2. RAS must go from high to low and remain low for the prescribed amount of time (t_{RAS}). When RAS goes low, the **memory row** addressed by the row address **is open**, and the charge from the cells in the selected row starts to flow to the bit lines.

3. The column address must be applied to the address input pins on the memory device before CAS goes low.
4. WE must be set high for a read operation to occur before the transition of CAS, and remain high after the transition of CAS.
5. Only after the prescribed amount of time (t_{RCD}), CAS must go from high to low and remain low for the prescribed amount of time (t_{CAS}). RAS-to-CAS delay (t_{RCD}) time ensures that the charge from the selected cells is on the bit lines and properly sensed by the sense amplifiers.
6. Data appears at the data output pins of the memory device. The time at which the data appears is called CAS latency (t_{CL}).
7. Before the read cycle can be considered complete, CAS and RAS must return to their inactive states. A new read or write access can start only after the prescribed amount of time (t_{RP}- Row Precharge).

The read access lasts for a **row cycle time** (t_{RC}):

$$t_{RC} = t_{RAS} + t_{RP} \ . \tag{1.3}$$

The row cycle time, t_{RC}, determines the minimum time a memory row takes to complete a full cycle, from row activation up to the precharging of the active row. This is an interval between accesses to different rows in a given set of DRAM arrays.

1.7.2 Writing data to DRAM memory

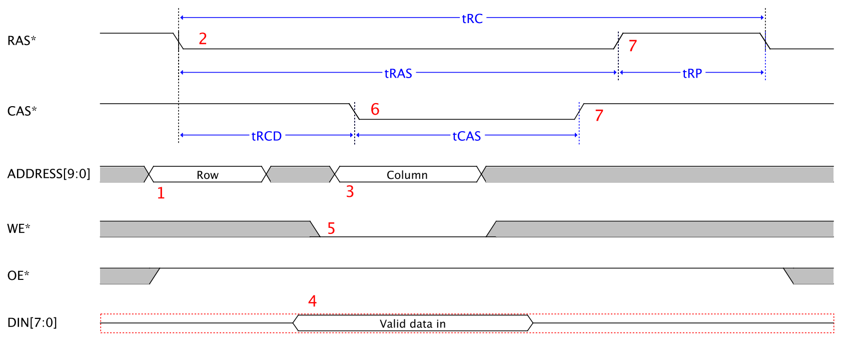


Fig. 1.15: Simplified DRAM write cycle.

To write to a DRAM memory cell, the row and column address for the DRAM cell must be selected, and data must be presented at the data input pins. The sense amplifier either charge the memory cell's capacitor or discharges it, depending on whether a 1 or 0 is to be stored. In terms of timing, the following steps must occur:

1. The row address must be applied to the address input pins on the memory device before RAS goes low.
2. RAS must go from high to low and remain low for the prescribed amount of time (t_{RAS}). When RAS goes low, the **memory row** addressed by the row address is open.
3. Data must be applied to the data input pins before CAS goes low.
4. The column address must be applied to the address input pins on the memory device after RAS goes low and before CAS goes low.
5. WE must be set low for a write operation to occur.
6. Only after the prescribed amount of time (t_{RCD}), CAS must switch from high to low and remain low for a prescribed amount of time (t_{CAS}).
7. Before the write cycle can be considered complete, CAS and RAS must return to their inactive states. A new read or write access can start only after the prescribed amount of time (t_{RP}).

The write access also lasts for a **row cycle time** (t_{RC}).

1.7.3 Refreshing the DRAM memory

Since DRAM memory cells are capacitors, the charge they contain can leak away over time. If the charge is lost, the data is lost! To prevent the loss of data, DRAMs must be refreshed, i.e., the charge on the individual memory cells must be restored. **DRAMs are refreshed one row at a time**. The frequency of refresh depends on the silicon technology used to manufacture the memory chip and the design of the memory cells. **Most of today's DRAMs require a refresh to occur every 64 ms**.

Reading or writing a memory cell has the effect of refreshing the selected cell because after read/write the entire row is precharged. Unfortunately, not all cells are read or written within 64 ms time frame. Hence, each row in the array must be accessed and restored during the refresh interval. The refresh cycles are distributed across the entire refresh interval of 64 ms in such a way that all rows are refreshed within the required interval. If, for example, a DRAM array has 4096 rows, every 15.6 microseconds a new row must be refreshed. At the end of the 64 ms interval, the process begins again.

DRAMs use an **internal oscillator** to determine the refresh frequency and a **counter** to keep track of which row is to be refreshed, and initiate the refresh periodically. Such an auto-initiated refresh is referred to as **self refresh**. To refresh one row of the memory array, the so-called **CAS-before-RAS refresh** is used. The following steps form the CAS-before-RAS refresh:

1. CAS must switch from high to low, while the WE signal remains in a high state (equivalent to read).
2. After the prescribed delay, RAS must switch from high to low.
3. The internal counter determines which row is to be refreshed and applies the row address at the address pins
4. After the required delay, CAS returns to a high level.
5. After the necessary delay, RAS returns to a high level.

Summary: DRAM Operations and Timings

DRAMs are asynchronous systems, responding to input signals whenever they occur. The DRAM will work properly, as long as the input signals are applied in the proper sequence, with signal durations and delays between signals that meet the specified limits.

Typical operations in DRAMs are: read, write, and refresh. All these operations are initiated and controlled by the prescribed sequence of input signals.

The read and write accesses last for a *row cycle time* (t_{RC}):

$$t_{RC} = t_{RAS} + t_{RP} \ .$$

DRAMs must be refreshed in order to prevent the loss of data. DRAMs are refreshed one row at a time. DRAMs use an *internal oscillator* to determine the refresh frequency and initiate a refresh and a *counter* to keep track of row to be refreshed. Such an auto-initiated refresh is referred to as *self refresh*. Self-refresh uses the so-called CAS-before-RAS sequence.

Summary: Important timings in DRAMs.

Name	Symbol	Description
Row Active Time	t_{RAS}	The minimum amount of time RAS is required to be active (low) to read or write to a memory location.
CAS latency	t_{CL}	This is the time interval it takes to read the first bit of memory from a DRAM with the correct row already open.
Row Address to Column Address Delay	t_{RCD}	The minimum time required between activating RAS and activating CAS . It is the time interval between row access and data ready at sense amplifiers.
Random Access Time	t_{RAC}	This is the time required to read any random memory cell. It is the time to read the first bit of memory from an DRAM without an active row. t_{RAC}= t_{RCD}+ t_{CL}.
Row Precharge Time	t_{RP}	After a successful data retrieval from the memory, the row that was used to access the data needs to be closed. This is the minimum amount of time that RAS must be inactive.
Row Cycle Time	t_{RC}	This is the time associated with single rad or write cycle. t_{RC}= t_{RAS}+ t_{RP}

1.8 Improving the performance of a DRAM chip

As mentioned earlier, one DRAM access is divided into row access and column access. Let's first look at how we read two consecutive columns from the same row in classic DRAMs. A timing diagram for reading two consecutive columns, A and B, in the same row X is shown in the Figure 1.16. Although both columns are in the same row X, we have to repeat the entire reading cycle from Figure 1.14 to read each column. Wouldn't it be better to keep the entire row 'open' once the amplifiers

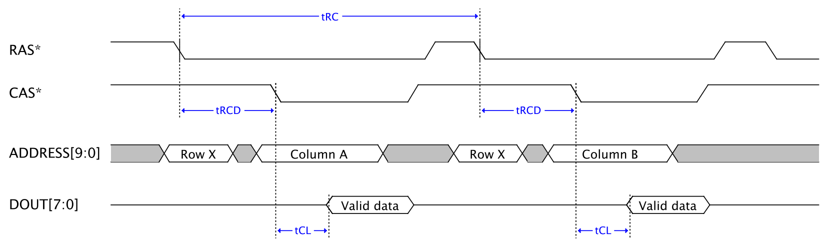


Fig. 1.16: Simplified read timing for two columns in the same row for conventional DRAM.

sensed all bits in that row? Actually, the sense amplifiers can act like a row buffer to keep the row data. That way, we don't have to access the row every time and then close it after reading each column. Exactly this solution was used for one of the first performance enhancements in DRAM memories. But wait, how often do we access two or more consecutive columns from the same row? Very often, indeed, due to **temporal and spatial locality**. All methods used to improve the performance of a DRAM chip and to decrease the access time rely on the ability to access all of the data stored in a row without having to initiate a completely new memory cycle.

1.8.1 Fast Page Mode DRAM

Fast Page Mode DRAM is a minor modification to the first-generation DRAMs that allows faster access to data in the same row. The performance of read and write accesses to a row was improved by avoiding the inefficiency of opening and precharging the same row repeatedly to access different columns in the same row. Fast Page Mode DRAM eliminates the need for a row address if data is located in the row previously accessed. In the Fast Page Mode DRAM, after a row has been opened by holding RAS low, the row bits are kept by the sense amplifiers, and multiple reads or writes could be performed to any of the columns in the open row. Each column access is initiated by asserting CAS and presenting a column address.

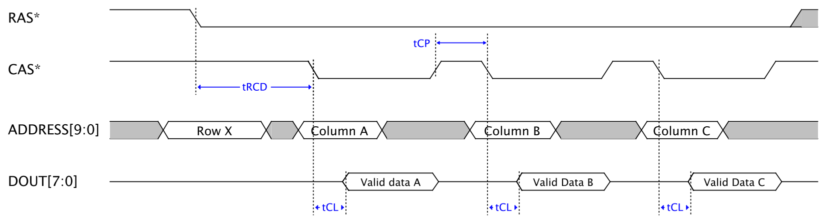


Fig. 1.17: Simplified read timing for two columns in the same row for conventional DRAM.

To read data using Fast Page Mode, we start a regular read operation by addressing the row (same steps 1 through 6 as in Figure 1.14). Once the row data is valid, we switch CAS high but leave RAS low. There is a minimum amount of time that CAS must be inactive, called the CAS precharge time (t_{CP}). When CAS has been inactive (high) for the required amount of time (t_{CP}), we repeat steps 3 through 6 of the read operation from Figure 1.14. We can continue in this way until a new row address is required or the chip needs to be refreshed. Figure 1.17 is a simplified timing diagram that illustrates a Fast Page Mode read cycle.

Let's use an example to illustrate how fast page mode impacts the system's performance. In this example, we compare two scenarios: 4 memory accesses in the same row without fast page mode, and 4 memory accesses in the same row with fast page mode. We are assuming that t_{RC} is 70 ns, t_{RCD} is 20 ns, and t_{CL} is 15 ns. In the first scenario, the data from the fourth column will be available after $3 \cdot t_{RC} + t_{RCD} + t_{CL} = 245ns$. In the second scenario, we are also assuming that CAS should remain high for 5 ns before going down again (t_{CP} is 5 ns), and that data is kept valid for 20 ns. Now, the data from the fourth column will be available after $t_{RCD} + 3 \cdot (t_{CL} + 20 + t_{CP}) = 140ns$.

1.8.2 Extended Data Output DRAM

The second change to improve the performance is Extended Data Out (EDO) DRAM. EDO is very similar to FPM. The primary advantage of EDO DRAMs over FPM DRAMs is that the data outputs are not disabled when CAS goes high on the EDO DRAM, allowing the data from the current read cycle to be present at the outputs while the next read cycle begins, i.e., data is still present on the output pins, while CAS is changing and a new column address is latched. This allows a certain amount of overlap in operation (pipelining), resulting in faster access (cycle) time. Figure 1.18 is a complete timing diagram that illustrates an EDO mode read cycle. Let's now illustrate how EDO impacts the system's performance using the same ex-

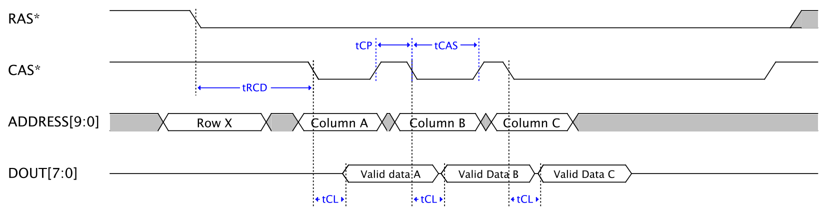


Fig. 1.18: Simplified read timing for two columns in the same row for conventional DRAM.

ample as before, i.e., 4 memory accesses in the same row with EDO. Assuming, that we keep the data valid for 20 ns, the data from the fourth column becomes available after $t_{RCD} + t_{CL} + 3 \cdot 20 = 95ns$.

Summary: FPM and EDO DRAMs

Due to *temporal and spatial locality*, we often access two or more consecutive columns from the same row.

All methods used to improve the performance of a DRAM chip and to decrease the access time rely on the ability to access all of the data stored in a row without having to initiate a completely new memory cycle.

Fast Page Mode DRAM eliminates the need for a row address if data is located in the row previously accessed.

In *EDO DRAMs*, data is still present on the output pins, while CAS is changing, and a new column address is latched. This allows a certain amount of overlap in operation (pipelining), resulting in faster access time.

1.9 Synchronous DRAM

Originally, DRAMs that we have just covered and were produced from the early 1970s to early 1990s had an asynchronous interface, in which input control signals have a direct effect on internal functions. The **synchronous DRAM (SDRAM)** device represents a significant improvement over the DRAM devices. In particular, SDRAM devices differ from previous generations of DRAM devices in two significant ways:

1. the **clock signal** was added to the SDRAM device; hence the SDRAM device has a synchronous device interface, where commands instead of signals are used to control internal latches, and
2. SDRAM devices contain **multiple independent banks**.

Besides, SDRAMs typically also have a programmable **mode register** to hold the number of bytes requested, and hence can send many bytes over several cycles per request without sending any new addresses. This type of transfer is referred to as **burst mode**.

SDRAMs have the clock signal and all internal actions occur on its negative edge. As we have seen, in DRAM devices, the RAS, CAS, and WE signals from the memory controller directly control internal latches and input/output buffers, and these signals can arrive at the DRAM device's pins at any time. The DRAM devices then respond to the RAS, CAS, and WE signals as soon as possible. Contrary, in SDRAM devices, the RAS, CAS, and WE signals do not directly control internal latches and buffers. In SDRAM devices these signals form a command bus used to transmit **commands** to the internal state machine, which **executes the commands at the falling edge of the clock signal**. In this way, the control of internal latches and input/output buffers moved from the external memory controller into the state machine in the SDRAM device's control logic. The RAS, CAS and WE names were retained for signals on the command bus that transmits commands, although these specific signals no longer control latches and buffers that are internal to the SDRAM device.

The second feature that significantly differentiates the SDRAM device from the DRAM devices is that the SDRAM devices contain multiple banks. The presence of multiple, independent banks in each SDRAM device means that while one bank is busy with a row activation command or a precharge command, the memory controller can send a new command to a different bank. Multiple banks now enable the interleaving of memory requests to different banks in a single SDRAM device. SDRAM devices contain either 2, 4, or 8 independent banks. One to three bank address inputs (BA0, BA1, and BA2) determine which bank the command refers to.

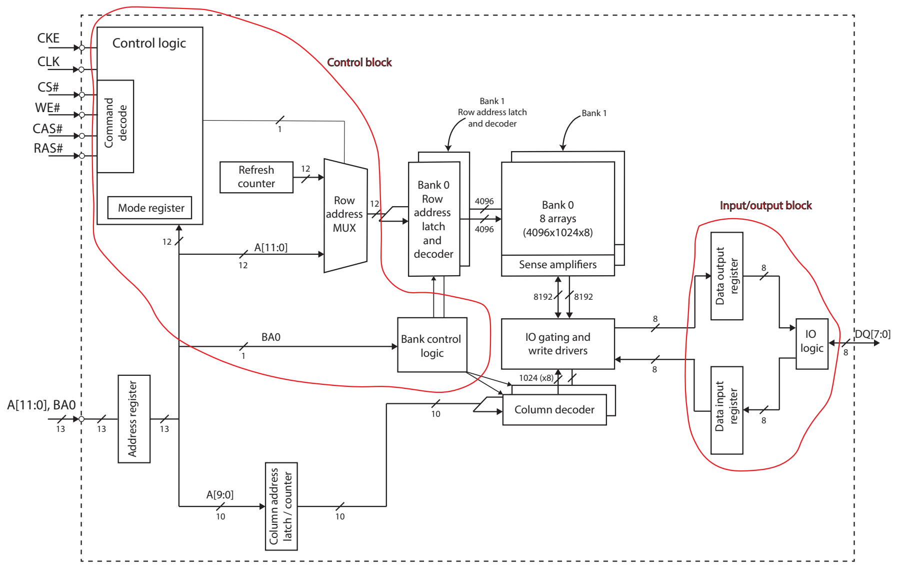


Fig. 1.19: Simplified block diagram of a SDRAM device with two banks.

1.9.1 Functional description

Figure 1.19 shows the simplified block diagram of an SDRAM device with two independent banks. The hash (#) beside a signal name denotes that the signal is active low. Each bank has its row address latch and decoder, its column decoder, and its sense amplifiers. Each bank in the SDRAM device in Figure 1.19 consists of eight DRAM arrays of size 4096-by-1024 bits. The address now consists of bank number (BA0), row address (A[11:0]), and a column address (A[9:0]).

In an SDRAM device, commands are decoded on the rising edge of the clock signal (CLK) and executed on the falling edge of CLK if the chip-select signal (CS) is active. The command is asserted on the command bus by the external memory controller . The command bus consists of WE, CAS, and RAS signals. All these signals are active low. Although the signal lines retain the function-specific names from DRAMs, they only form a command bus. Table 1.1 shows the command set of the SDRAM device and the input signal combinations on the command bus that designate the commands. The table also shows that as long as CS is not active, the SDRAM device ignores the signals on the command bus.

The control block in Figure 1.19 consists of control logic, a multiplexor to select a row address, a refresh counter and bank control logic. The refresh counter keeps track of the row to be refreshed. The multiplexor is used to select a row address to be transferred into the row address latch and decoder. The address is either an address coming from the refresh counter (in case the control logic performs a refresh cycle) or an address from the external address bus coming from the DRAM

Table 1.1: SDRAM commands.

Command	**CS#**	**RAS#**	**CAS#**	**WE#**	**Address**
COMMAND INHIBIT	H	X	X	X	X
NO OPERATION (NOP)	L	H	H	H	X
ACTIVE (select bank and activate row)	L	L	H	H	Bank/row
READ (select bank and column, and start READ burst)	L	H	L	H	Bank/col
WRITE (select bank and column, and start WRITE burst)	L	H	L	L	Bank/col
PRECHARGE (deactivate row in bank)	L	L	H	L	Bank/row
AUTO REFRESH	L	L	L	H	X
LOAD MODE REGISTER	L	L	L	L	Code

controller. Control logic contains a command decoder, a finite state machine that executes commands, and the mode register. The mode register is a programmable 10-bit register whose individual bits determine:

- CAS latency (CL). CL is t_{CL} rounded-up to the nearest number of clock cycles,
- the length of the burst transfer,
- and the order of memory words in the burst transfer.

The control logic receives a command from the command bus. Then, depending on the type of command and values contained in the respective fields of the mode register, the control logic performs specific sequences of operations to execute the command. These operations are performed by the internal state machine on successive clock cycles without requiring clock-by-clock control from the memory controller. Figure 1.20 illustrates a simplified state diagram of the internal state machine. After the initialization of the mode register, the internal state machine is in the Idle state with all banks and rows precharged. If no command is issued to SDRAM, the SDRAM chip will regularly perform the self-refresh. The internal counter drives the self-refresh operation. To start memory access, the memory controller should first issue the ACTIVE command. This will eventually open a row/bank, and the internal state machine waits in the Active state for additional commands. To read data, the memory controller should issue the READ command, and to write data into memory, the memory controller should issue the WRITE command. Then, the internal state machine enters the Read or Write state, and uses the column address and generates the appropriate internal signals to access the column. The READ or WRITE commands can be followed by any number of READ or WRITE commands or the PRECHARGE command can be issued to restore the data and close the open bank/row. After the precharge operation has been executed, the internal state machine will wait in the IDLE state.

For example, in the case of the ACTIVE command, the state machine passes the row address to the row address latch and decoder through the multiplexor. The address bit BA0 determines the bank, which will be accessed. The bank control block, which acts as a decoder, selects the appropriate row address latch and decoder, and the appropriate column decoder based on the BA0 bit. The selected row is then

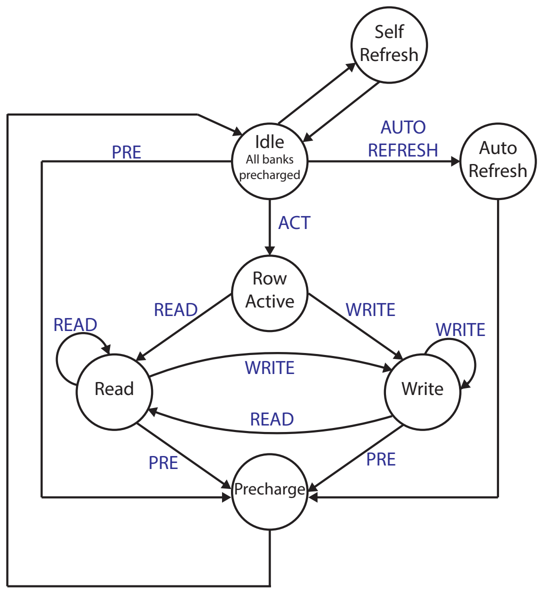


Fig. 1.20: Simplified state diagram of the internal state machine.

opened and its content is transferred into the sense amplifiers. In the case the memory controller asserts a READ command, the internal state machine drives the bank control logic, which selects the appropriate column decoder, based on the BA0 bit. The column decoder then selects the word from the sense amplifiers of the chosen bank. Each bank has its own column decoder - this feature is especially useful when interleaving transfers from two (or more) active banks. The SDRAM device in Figure 1.19 provides for two rows of the DRAM to be opened simultaneously. Memory accesses between two opened banks can be interleaved to hide RAS-to-CAS delay and row precharge time. When an address is firstly sent that designates a new bank, the row in that bank must be opened. But when subsequent access specifies the same row in an already open bank, the access can happen quickly, sending only the column address. This feature requires that each bank has its own row address latch, sense amplifiers and a column decoder. For example, while one row is accessed, the memory controller can send an ACTIVE command to a different bank and, in such a way, transfer a new row into the sense amplifiers. This row can than be read or written to without waiting for t_{RCD}. Later, we will learn how data is transferred to/from SDRAM chip and how the burst transfers and bank interleaving can speed up memory transactions.

Summary: SDRAMs

SDRAM devices have a synchronous device interface, where commands, instead of signals, are used to control internal latches.

In SDRAM devices, signals CAS, RAS, WE and CS form a *command bus* used to transmit *commands* to the *internal state machine*.

SDRAM devices contain multiple independent banks.

SDRAMs can transfer many columns over several cycles per request without sending any new addresses. This type of transfer is referred to as *burst mode*.

1.9.2 Basic operations and timings

Now that we are familiar with the basic functionality of SDRAMs, we are going to present four basic operations in SDRAMs: ACTIVE, READ, WRITE, and PRECHARGE.

1.9.2.1 Activate (open) row

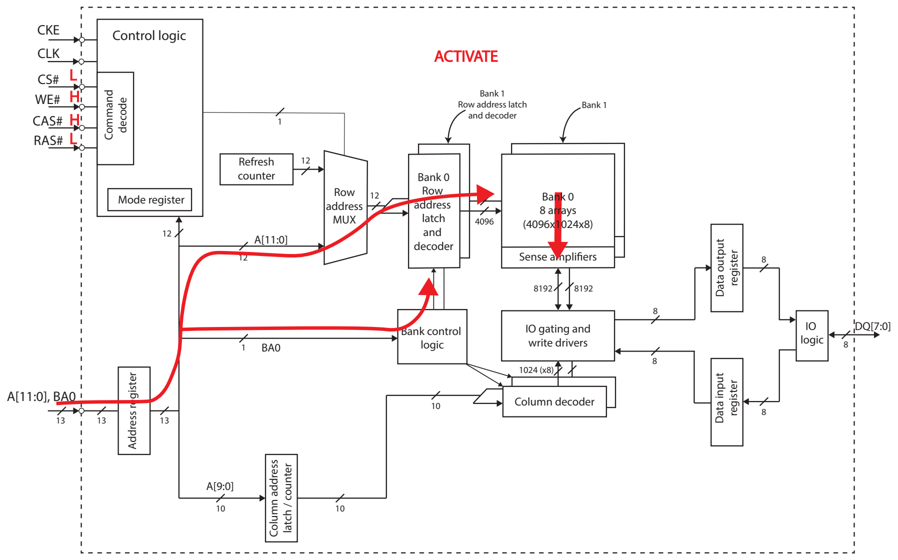


Fig. 1.21: The progression of the ACTIVE command.

Before any READ or WRITE commands can be issued to a bank within the SDRAM, a row in that bank must be opened. This is accomplished via the ACTIVE command. The purpose of the ACTIVE command is to open (activate) a row in a selected bank and move data from the DRAM arrays to the sense amplifiers of the open bank. Figure 1.21 illustrates the progression of the ACTIVE command. The address A11-A0 from the address bus is stored into the row address latch and decoder of the selected bank. The address bit BA0 selects the bank and its row address latch and decoder. Then, the entire row of data is read into the sense amplifiers. Similarly to DRAMs, two timings are associated with the ACTIVE command: *Row Address to Column Address Delay* (t_{RCD}) and *Row Active Time* t_{RAS}. t_{RCD} is the time it takes for the ACTIVE command to move data from the DRAM cell arrays to the sense amplifiers that hold the entire row of data. After t_{RCD}, a column read or write access commands can be issued to move data between the sense amplifiers and the memory controller through the input/output block and data bus (Figure 1.22). Row

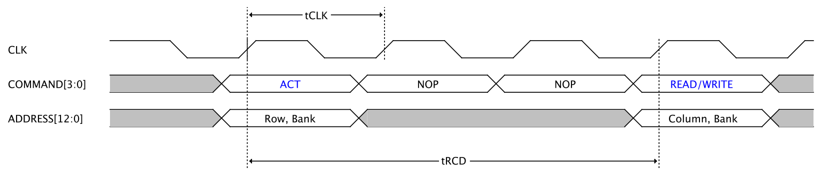


Fig. 1.22: Meeting t_{RCD}.

address to column address delay, t_{RCD}, should be divided by the clock period and rounded up to the nearest whole number to determine the earliest clock edge after the ACTIVE command on which a READ or WRITE command can be issued. For example, a t_{RCD} of 20ns with a 125 MHz clock (8ns period) results in 2.5 clock periods, rounded to 3. A subsequent ACTIVE command to a different row in the same bank can only be issued after the previous active row has been precharged.

Row active time, t_{RAS}, is the minimum amount of time that must elapse before the PRECHARGE command can be issued to the open row. t_{RAS} is also referred to as ACTIVE-to-PRECHARGE time.

1.9.2.2 Read

Figure 1.23 illustrates the progression of a column read command. A column read command moves data from the sense amplifiers of a selected bank to the memory controller through IO gating and write drivers and data output register. The address A[9:0] from the address bus is stored into the column address latch and column decoder of the selected bank. The address bit BA0 selects the bank and its column

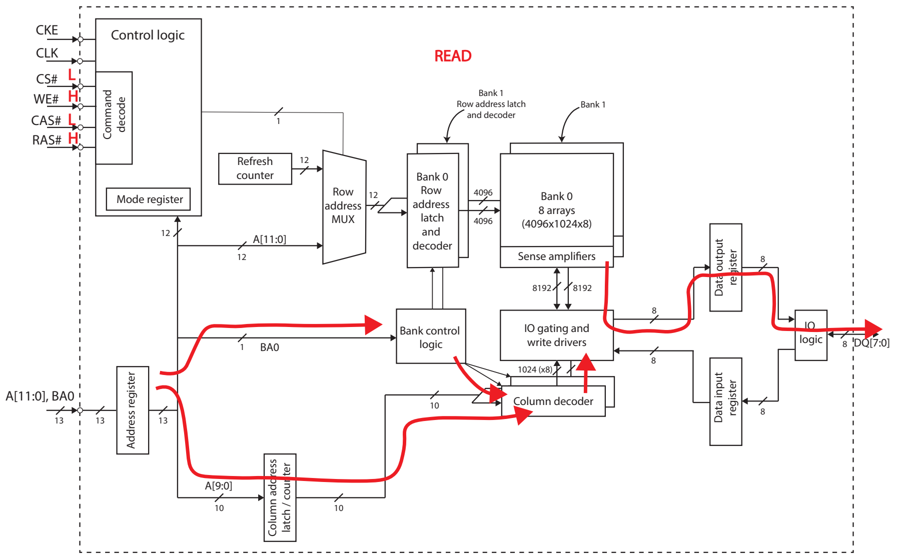


Fig. 1.23: The progression of the READ command.

decoder and sense amplifiers. Then, the selected 8-bit data is read from the sense amplifiers and output to DQ pins. There are two (timing) parameters associated with a column read command: CAS latency (CL) and burst length (BL).

CL is the time it takes for the SDRAM device to move the requested data from the sense amplifiers through IO gating and output register onto the data DQ bus. For SDRAMs, the **CAS latency (CL) is the delay, in clock cycles**, between the registration of a READ command and the availability of the output data. In modern SDRAMs, the CAS latency can be set to two or three clocks. If a READ command is registered at clock edge n, and the CL is m clocks, the data will be available by clock edge $n+m$. Now, we can combine the timing parameters, t_{RCD} and CL, to

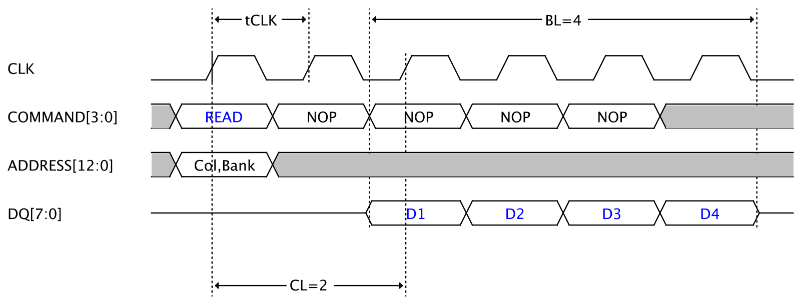


Fig. 1.24: The READ burst with CL=2 and BL=4.

form a **random access time** (t_{RAC}).

$$t_{RAC} = t_{RCD} + CL \tag{1.4}$$

Random access time, t_{RAC}, denotes the speed at which the SDRAM device can move data from the DRAM arrays into the memory controller.

Modern memory systems move data in relatively short bursts, and the burst length (BL) is programmable. The burst length determines the maximum number of column locations that can be accessed for a given READ or WRITE command. Typically, BL is 2, 4, or 8. Read bursts are initiated with a READ command, as shown in Figure 1.24. The starting column and bank addresses are provided with the READ command. During READ bursts, the valid data from the starting column address is available following the CAS latency after the READ command. Each subsequent data will be valid by the next positive clock edge. Upon completion of a burst, assuming no other commands have been initiated, the DQ signals will go to High-Z.

Data from a fixed-length READ burst can be followed immediately by data from a new READ or WRITE command. In such a way, a continuous flow of data can be maintained. SDRAM devices use a pipelined architecture, and therefore, a READ command can be initiated on any clock cycle following a READ command. The

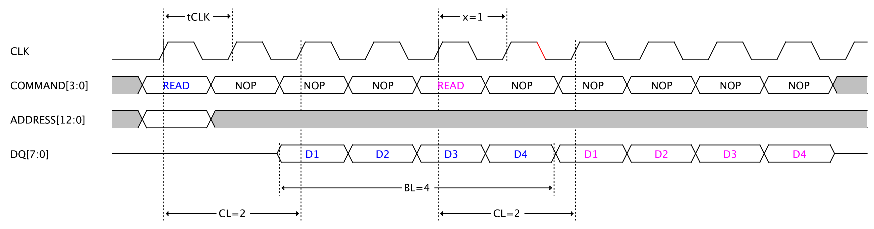


Fig. 1.25: Two consecutive READ bursts with CL=2 and BL=4.

new READ command should be issued x cycles before the clock edge at which the last desired data element is valid, where $x = CL - 1$. This is shown in Figure 1.25 for CL=2 and BL=4. Full-speed random read accesses can be performed to the same bank, or each subsequent READ can be performed to a different open bank (bank interleaving).

1.9.2.3 Write

Figure 1.26 illustrates the progression of the WRITE command. The WRITE command moves data from the DQs pins through IO gating and write drivers and data input register to the sense amplifiers of a selected bank. The column address A[9:0] from the address bus is stored into the column address latch and column decoder of

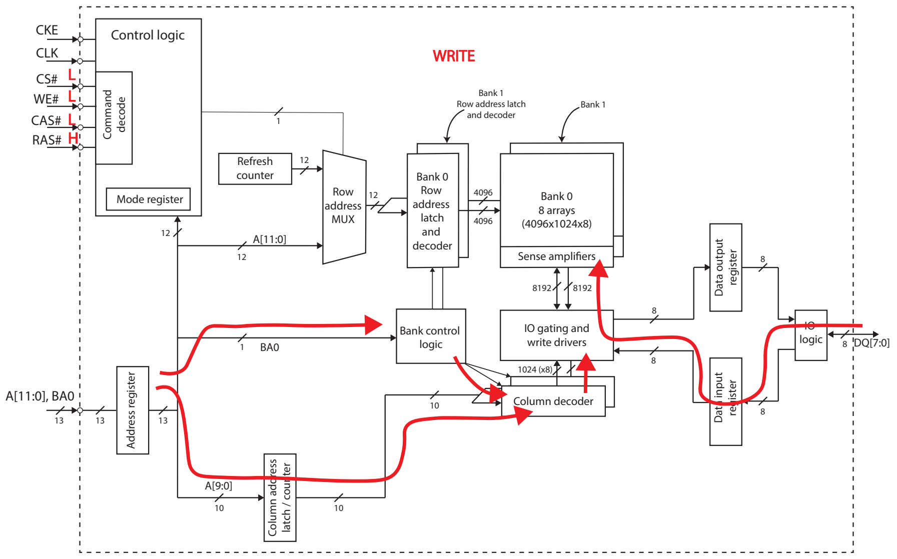


Fig. 1.26: The progression of the WRITE command.

the selected bank. The address bit BA0 selects the bank and its column decoder and sense amplifiers.

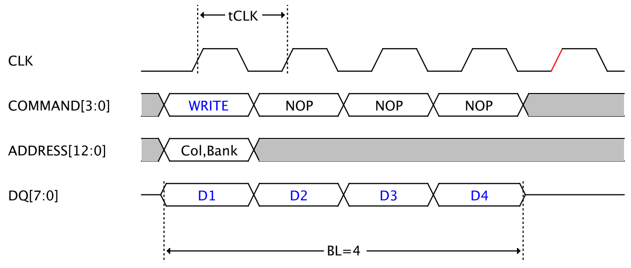


Fig. 1.27: The WRITE burst with BL=4.

Figure 1.27 shows a write burst with BL=4. The starting column and bank addresses are provided with the WRITE command, which initiates write bursts. During write bursts, the first valid data is registered coincident with the WRITE command. Subsequent data are registered on each successive positive clock edge. Upon completion of a fixed-length burst, assuming no other commands have been initiated, the DQ pins remain at High-Z, and any additional input data is ignored.

Data from a fixed-length WRITE burst can be followed immediately by data from a new READ or WRITE command. In such a way, a continuous flow of data can be maintained. Figure 1.28 shows two consecutive write bursts with BL=2.

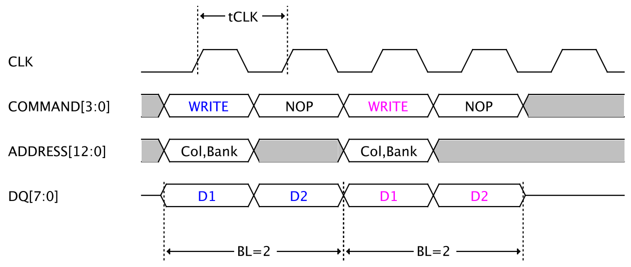


Fig. 1.28: Two consecutive WRITE bursts with BL=2.

1.9.2.4 Precharge

So far, we have seen that accessing data on a SDRAM device is a two-step process. First, the ACTIVE command opens a row in a selected bank and moves data from the DRAM cells in that row to the sense amplifiers. The data then remains in the sense amplifiers and can be transferred to or from SDRAM using the READ and WRITE commands. The PRECHARGE command is used to deactivate the open row in a particular bank or the open row in all banks - it restores data in the row, resets the sense amplifiers and the bit lines, and prepares the sense amplifiers for another row access. Figure 1.29 illustrates the progression of the PRECHARGE command. The address A[11:0] from the address bus is stored into the row address latch and decoder of the selected bank. The address bit BA0 selects the bank and its row address latch and decoder. Then the selected bank is precharged.

The timing parameter associated with the (row) PRECHARGE command is row precharge time,t_{RP}. The bank(s) will be available for a subsequent access row precharge time (t_{RP}) after the PRECHARGE command is issued. Recall that t_{RAS} is the minimum amount of time that the row should remain open before issuing the PRECHARGE command (i.e., ACTIVE-to-PRECHARGE time). Now, we can combine the timing parameters, t_{RP} and t_{RAS}, to form a **row cycle time** (t_{RC}):

$$t_{RC} = t_{RAS} + t_{RP} \tag{1.5}$$

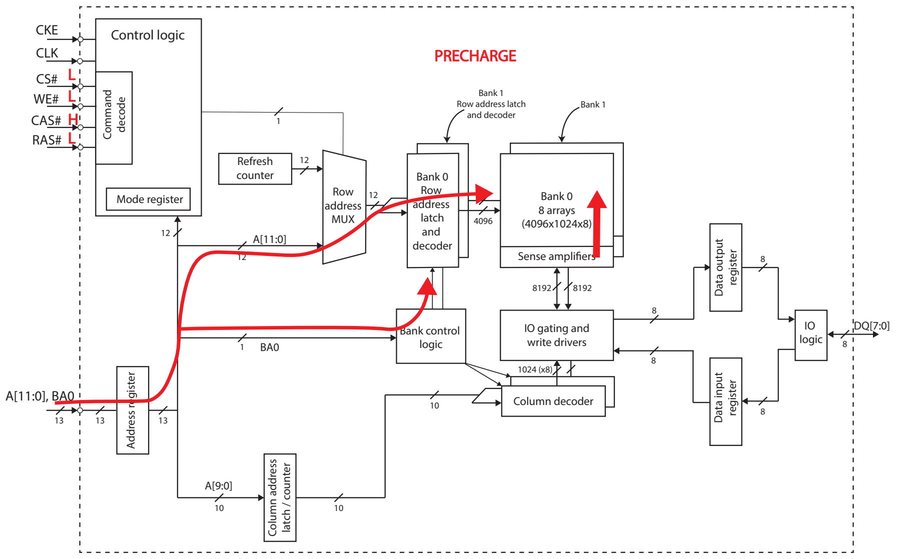


Fig. 1.29: The progression of the PRECHARGE command.

Row cycle time, t_{RC}, denotes the speed at which the SDRAM device can bring data from the DRAM arrays into the sense amplifiers, restore the data to the DRAM cells, and be ready for another ACTIVE command. t_{RC} is the fundamental limitation to the speed at which data may be retrieved from different rows within the same SDRAM bank.

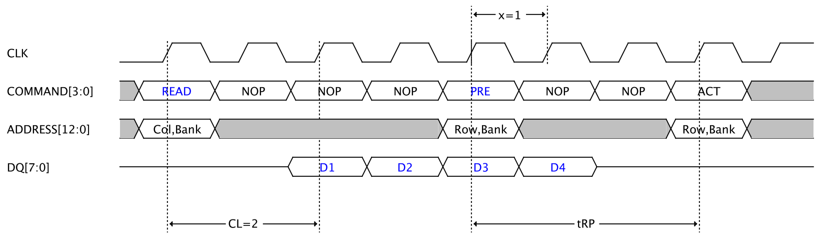


Fig. 1.30: READ to PRECHARGE.

A PRECHARGE command may follow a READ or WRITE burst to the same bank. In the case of PRECHARGE after READ, the PRECHARGE command should be issued $x = CL - 1$ cycles before the clock edge at which the last data element in a burst is valid. This is shown in Figure 1.30 for CL = 2. In the case of PRECHARGE after WRITE, the PRECHARGE command should be issued at least one clock period after the positive clock edge at which the last input data is registered, regardless of frequency (Figure 1.31).

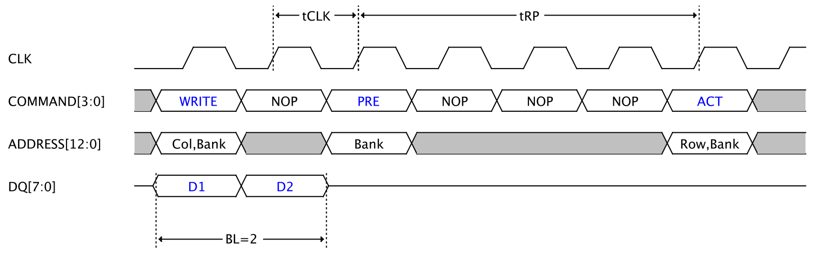


Fig. 1.31: WRITE to PRECHARGE.

Following the PRECHARGE command, a subsequent command to the same bank cannot be issued until t_{RP} is met. The disadvantage of the PRECHARGE command is that it requires that the command and address buses be available at the appropriate time to issue the command.

1.10 Double Data Rate SDRAM

How can we further speed-up memory transfers? The solution is to access two adjacent columns simultaneously with one READ/WRITE command. So, instead of reading/writing one 8-bit memory word (column), we can read/write two adjacent 8-bit memory words (columns). But with that solution, a new challenge arises. How to transfer two 8-bit words in the same amount of time as one 8-bit word? One solution would be to have a twice wider bus. Thus, instead of the 8-bit data bus (DQ[7:0]), the SDRAM device would have had a 16-bit data bus (DQ[5:0]). But this could be challenging because more wires mean more noise on the data bus and worse data/signal integrity. The second solution would be to have a twice faster bus. But this is also challenging because higher frequency means worse data/signal integrity and higher power consumption. The better solution is to **transfer data at both clock edges to double data bus bandwidth without a corresponding increase in clock frequency or in data bus width**.

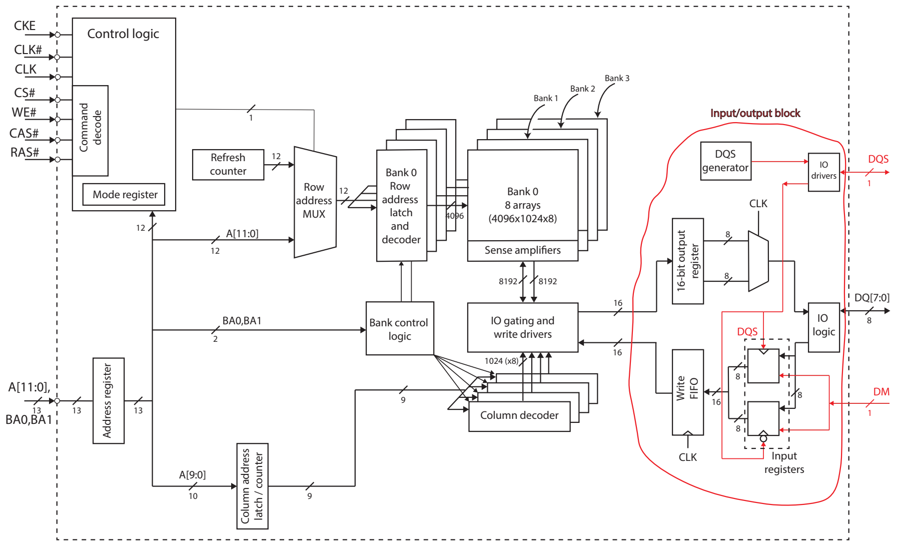


Fig. 1.32: Simplified block diagram of a DDR SDRAM device with four banks.

In SDRAM devices, each time a column read command is issued, the control logic determines the duration of the data burst, and each column is moved separately from the sense amplifiers through the I/O logic to the external data bus. However, the separate control of each column limits the operating data rate of the SDRAM device. **In Double Data Rate (DDR) SDRAM devices, two adjacent columns are moved in parallel from the sense amplifiers to the output data register**, and the data is then pipelined through a multiplexor to the external data bus. The feature to access two columns at a time is referred to as **2N-prefetch**. Figure 1.32 illustrates the simplified block diagram of a DDR SDRAM device with four independent banks. We can see that the internal structure is similar to the internal structure of an SDRAM device except for the IO block. The memory arrays and banks used in DDR SDRAMs are the same as in SDRAMs. The name "double data rate" refers to the fact that a DDR SDRAM with a certain clock frequency achieves nearly twice the bandwidth of an SDRAM running at the same clock frequency, due to this double pumping. Double data rate SDRAM is a significant improvement of SDRAM. DDR SDRAMs have been used in computer systems' memory since 2001.

The main difference in the internal organization of DDR SDRAM over SDRAMs is an improved I/O block. The I/O block of an 8-bit DDR SDRAM device from Figure 1.32 now consists of a 16-bit output register, a 2/1 multiplexor, a DQS generator, two 8-bit input registers, a write FIFO and IO logic. Figure 1.32 shows that, in the case of the READ access, given the width of the external data bus (DQ) as 8-bit, 16 bits are moved from the sense amplifiers to the output register, and the 16 bits are then pipelined through the multiplexor to the external data pins. The clock signal controls the select input of the multiplexor. In the case of the WRITE access, two 8-bit data are stored successively (one after the other) in two 8-bit input registers and then transferred together into a 16-bit write FIFO. From there, data is transferred to the sense amplifiers through IO gating and write drivers. Besides, DDR SDRAMs have two new control signals: data strobe (DQS) and data mask (DM). In the following subsections, we are going to describe the operation of the IO block during the READ and WRITE accesses, and the role of DQS and DM in more detail.

The downside of the 2N-prefetch architecture means that short column bursts are no longer possible. In DDR SDRAM devices, a minimum burst length of 2 columns of data is accessed per column read command.

1.10.1 Functional description

The DDR SDRAM uses a double data rate architecture to achieve high-speed operation. The double data rate architecture is essentially a 2N-prefetch architecture with an I/O block designed to transfer two data words per clock cycle at the I/O pins. A single read or write access for the DDR SDRAM effectively consists of a single 2N-bit-wide, one clock cycle data transfer at the internal DRAM core, and two corresponding N-bit-wide, one-half clock cycle data transfers at the I/O pins.

The DDR SDRAM operates from a **differential clock** . Differential clock employs two complementary clock signals, CLK and CLK#. In general, a clock signal can be regarded as a binary signal whose duty cycle is nominally 50%. As we know, the clock signal is used to synchronize and capture data at its rising or falling edges. In DDR SDRAMs, data are synchronized and captured at both clock edges. But clocks are notoriously bad at having 50% duty cycles at high frequencies. As a rule of thumb, high frequency is generally considered to be above 100MHz. So, the reason for having two separate clocks is to allow for more precise alignment of the rising edges of the clock with the data. The crossing of CLK going HIGH and CLK# going LOW is referred to as the positive edge of CLK. Commands (address and control signals) are registered at every positive edge of CLK.

Read and write accesses to the DDR SDRAM are burst oriented. Accesses start at a selected location and continue for the BL number of locations in a sequence. Similarly to SDRAMs, accesses begin with the registration of an ACTIVE command, which may then be followed by a READ or WRITE command. The address bits registered coincident with the ACTIVE command are used to select the bank and row to be accessed. The address bits registered coincident with the READ or WRITE command are used to select the bank and the starting column location for the burst access. The DDR SDRAM provides for programmable READ or WRITE burst lengths of 2, 4, or 8 locations.

1.10.1.1 Read

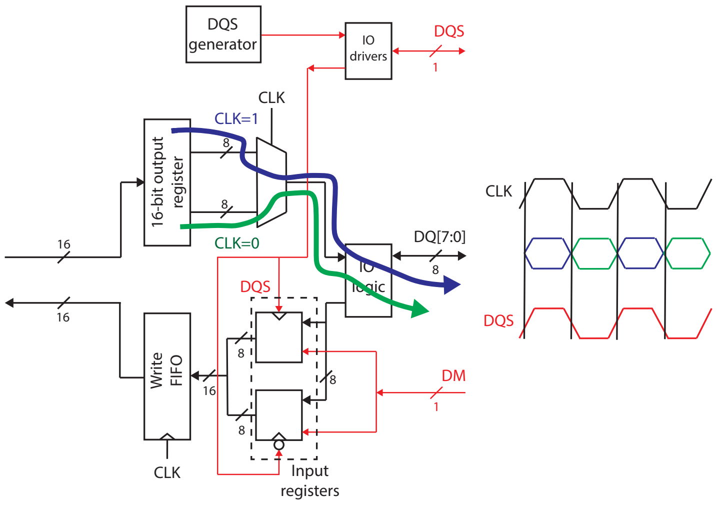


Fig. 1.33: Operation of the IO block during READ.

Figure 1.33 illustrates the operation of the I/O block during the READ access to an 8-bit DDR SDRAM. First, 16 bits (two adjacent 8-bit columns) are transferred from the sense amplifiers to the 16-bit output register as the consequence of the READ command. Then, when CLK is HIGH, the first 8-bit word is transferred through the multiplexor onto the I/O pins; when the CLK signal is LOW, the second 8-bit word is transferred through the multiplexor onto the IO pins. In such a way, two 8-bit words from the DRAM array are transferred in one clock cycle. A bidirectional data strobe (DQS) signal is transmitted, along with data, for use in data capture at the memory controller. The DQS generator generates the DQS signal and synchronizes it with the memory controller's global clock. The **DQS signal is edge-aligned with data for READs**.

1.10.1.2 Write

Figure 1.34 illustrates the operation of the I/O block during the WRITE access to an 8-bit DDR SDRAM. Two 8-bit words are successively transferred from the data bus into the input registers.Two input registers form a DDR input pair. A bidirectional data strobe **(DQS) signal is now transmitted by the memory controller**, along with data, for use in data capture at DDR SDRAM. The first 8-bit word is captured into the first data input register at the positive edge od DQS, while the second 8-bit word is captured into the second input register at the negative edge of DQS. Hence, **input data is registered on both edges of DQS**, and **DQS signal is center-aligned with data for WRITEs**. Then, the 16-bit data is transferred into the write FIFO at the positive edge od the CLK signal and written to the sense amplifiers and the DRAM array during the PRECHARGE command.

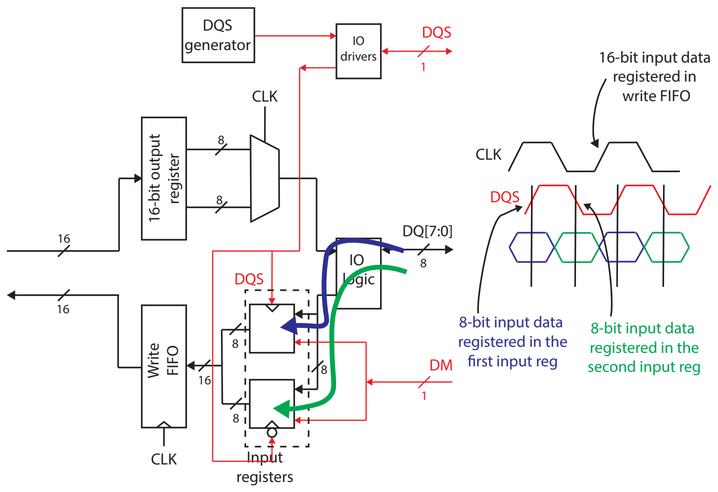

DQS generator
IO drivers
DQS
1
16-bit input data registered in write FIFO
CLK
16-bit output register
8
8
CLK
DQS
16
DQ[7:0]
8
IO logic
16
DQS
Write FIFO
16
8
8
8
DM
1
CLK
Input registers
8-bit input data registered in the first input reg
8-bit input data registered in the second input reg

Fig. 1.34: Operation of the IO block during WRITE.

1.10.2 DDR SDRAM timing diagrams

1.10.2.1 Read bursts

Figure 1.35 shows the timing for a read burst with CL=2 and BL=4. During READ bursts, the valid data-out element from the starting column address is available following the CL after the READ command. Each subsequent data-out element is valid at the next positive or negative clock edge (i.e., at the next crossing of CLK and CLK#). DQS is driven by the DDR SDRAM along with output data. The initial LOW state on DQS is known as the *read preamble*; the LOW state coincident with the last data-out element is known as the *read postamble*. Upon completion of a read burst, assuming no other commands have been initiated, the DQ will go High-Z.

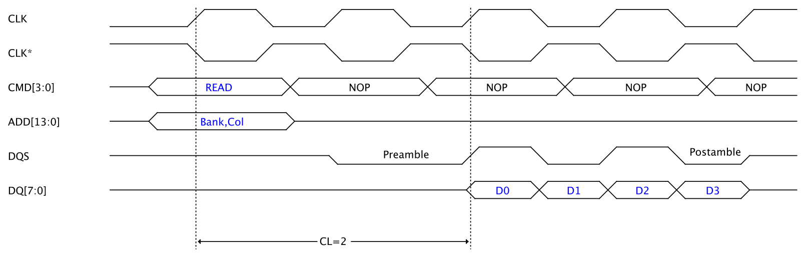


Fig. 1.35: The DDR READ burst with CL=2 and BL=4.

Data from any READ burst may be concatenated with data from a subsequent READ command. In such a way, a continuous flow of data can be maintained. The first data element from the new burst will follow the last element of a completed burst if the new READ command is issued x cycles after the first READ command, where x equals the number of desired data element pairs (pairs are required by the 2N-prefetch architecture). This is shown in Figure 1.35.

A PRECHARGE command may follow a READ burst to the same bank. The PRECHARGE command should be issued x cycles after the READ command, where x equals the number of desired data element pairs ($x = BL/2$). This is shown in Figure 1.37. Following the PRECHARGE command, a subsequent command to the same bank cannot be issued until both t_{RAS} and t_{RP} have been met.

1.10.2.2 Write bursts

Figure 1.38 shows the timing for a WRITE burst with BL=4. Input data appearing on the DQ is written to the memory array subject to the data mask (DM) input coin-

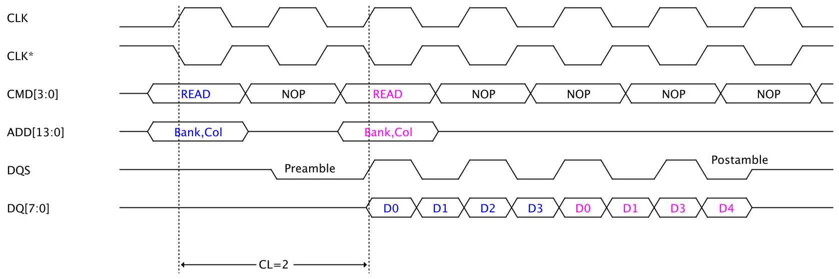


Fig. 1.36: Two consecutive DDR READ bursts with CL=2 and BL=4.

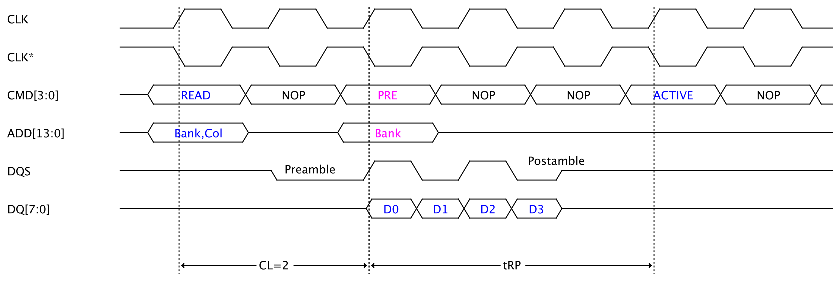


Fig. 1.37: DDR READ to PRECHARGE.

cident with the data. The DQS and DM signals are now transmitted by the memory controller, along with data. If the DM signal is registered LOW, the corresponding input data is written to memory. If the DM signal is registered HIGH, the corresponding input data is ignored, and a WRITE is not executed to that column location. During WRITE bursts, the first valid input data element is registered on the first rising edge of DQS following the WRITE command. Subsequent data elements are registered on the successive edges of DQS. The LOW state on DQS between the WRITE command and the first rising edge is known as the *write preamble*, and the LOW state on DQS following the last input data element is known as the *write postamble*. The first input data element following the WRITE command, along with its DQS, should be valid on the data bus one clock period after the WRITE command. Actually, most modern DDR SDRAMs specify this time between the WRITE command and the first corresponding rising edge of DQS from 75% to 125% of one clock cycle. In all of the WRITE diagrams, this time is one clock cycle.

Data for any WRITE burst may be concatenated with a subsequent WRITE command. The new WRITE command should be issued *x* cycles after the first WRITE

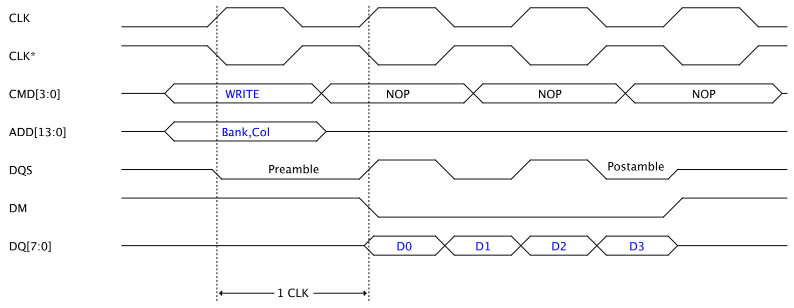


Fig. 1.38: The DDR WRITE burst with BL=4.

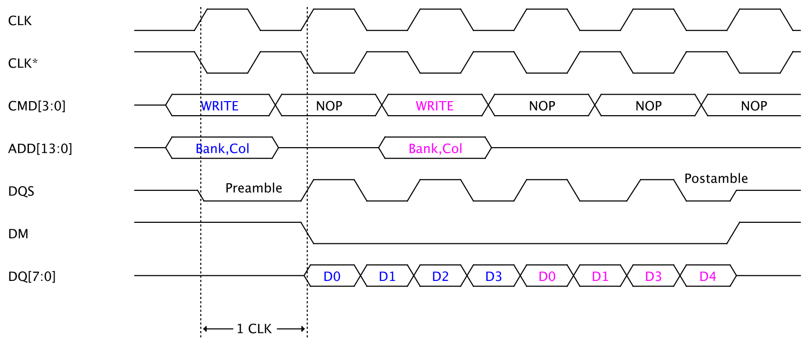


Fig. 1.39: Two DDR WRITE bursts with BL=4.

command, where x equals the number of desired data element pairs. Figure 1.39 illustrates two concatenated bursts with BL=4.

A PRECHARGE command to the same bank may follow a WRITE burst, as shown in Figure 1.40. There is a time period, **write recovery time** (t_{WR}), associated with the WRITE-to-PRECHARGE command sequence. Only the data-in pairs registered prior to the t_{WR} period are written to the internal array. After the PRECHARGE command, a subsequent command to the same bank cannot be issued until t_{RP} is met.

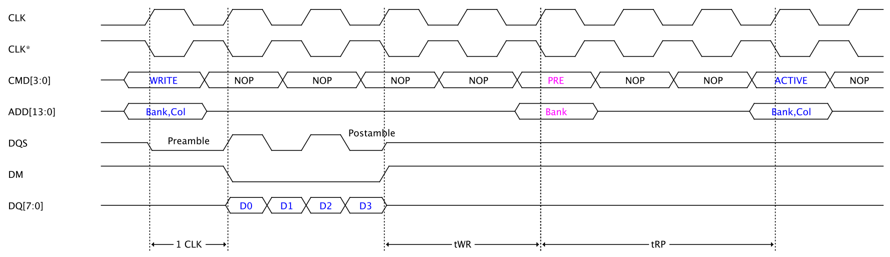


Fig. 1.40: DDR WRITE to PRECHARGE.

1.10.3 Address Mapping

Now that we are familiar with the basic operations in SDRAMs, we can move forward and see how an address from the CPU should be mapped into SDRAM's bank, row, and column address. The memory controller performs the address mapping. Let us suppose we are addressing a DDR SDRAM chip that consists of 8 banks, and each bank has eight DRAM arrays of size 4096 rows by 1024 columns. To address such a DDR SDRAM chip, we need 12 bits for the row address, three bits for the bank address, and 10 bits for the column address.

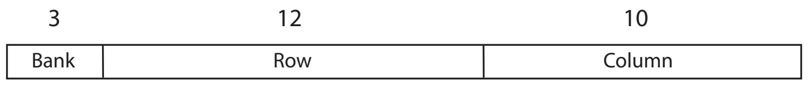


Fig. 1.41: Naive address mapping.

Figure 1.41 shows the naive way of an address mapping, where the top address bits are used to address the bank, the middle 14 bits are used to address the row, and the last 10 bits select the column. The main problem of such naive address mapping is that consecutive rows are in the same bank; hence, there is no bank interleaving. In the case of consecutive memory transfers consisting of more than one row, the currently open row should first be precharged before the new row is open.

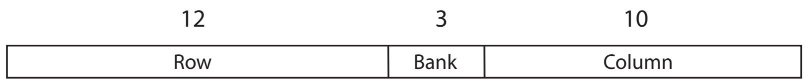


Fig. 1.42: Bank interleaving.

The better way of an address mapping would be to take advantage of bank interleaving, such that consecutive rows are in different banks. In this way, we can open a new row before the currently accessed row is precharged. We say that the precharge time is masked. Figure 1.42 shows the address mapping, where **bank interleaving** is used. Now, the top address bits select the row, while the middle address bits select the bank. Each time the end of a row is reached, the same row in a different bank is accessed.

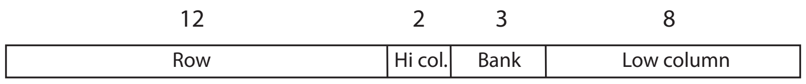


Fig. 1.43: Cache block interleaving.

The third way of an address mapping would be to take into account the cache memory. Typically, the cache block is of size 64 bytes. In reality, memory reads or writes are rarely random due to locality of reference. If a cache is used to support the locality of references, the CPU will access consecutive cache blocks. Hence, the cache misses will occur on the consecutive 64 bytes in memory. For example, if a cache block is stored in the last 64 bytes of a row, the cache miss on the next cache block would require to precharge the row and open a new one. In the case were consecutive cache blocks are stored in different banks, a row precharge would not be required. Thus it would be better to put consecutive cache blocks into different banks - this is **called cache block interleaving**. Figure shows the address mapping, where cache block interleaving is used. Now the column bits are split into two parts. Low column bits select the word within the cache block. The remaining hi column bits address the cache block in different banks.

1.10.4 Memory timings: a summary

So far, we have learned that each memory operation is associated with one or more memory timings that should be met in order to perform these operations correctly. Table 1.2 summarizes the most important memory timings.

CL, t_{RCD}, and $t_{RP}$are for most modern SDRAMs, typically around 13 ns, and have not changed significantly since the SDRAMs were first introduced. Actually, the DRAM cell and array process technologies have not significantly changed over the decades, and only the techniques to speed-up memory transfers have been (e.g. synchronous interface, bank interleaving, etc.). The next subsection covers the techniques to speed-up memory transfers in DDR SDRAMs.

Table 1.2: Summary of important timings in SDRAMs.

Name	Symbol	Description
CAS latency	CL	The number of cycles between sending a column address to the memory and the beginning of the data in response to a READ command. This is the number of cycles it takes to read the first bit of memory from a DRAM with the correct row already open. CL is an exact number that must be agreed on between the memory controller and the memory.
Row Address to Column Address Delay	t_{RCD}	The minimum number of clock cycles required between opening a row and issuing a READ/WRITE command. The time to read the first bit of memory from an SDRAM without an active row is t_{RCD}+ CL.
Row Precharge Time	t_{RP}	The minimum number of clock cycles required between issuing the precharge command and opening the next row. The time to read the first bit of memory from an SDRAM with the wrong row open is t_{RP}+ t_{RCD}+ CL.
Row Active Time	t_{RAS}	The minimum number of clock cycles required between a row active command and issuing the precharge command. This is the time needed to internally refresh the row, and overlaps with t_{RCD}. In SDRAM modules, it is usually t_{RCD}+ CL.

1.10.5 DDR Versions

To bust the performance od DDR SDRAMs, DDR SDRAMs have been further improved. Due to its nature (data is stored as a charge) and the process technology used to implement DRAM cells, the DRAM core (DRAM arrays) has not changed significantly over the decades, and its speed of operation remains relatively low. In SDRAMs, the clock rate used to transfer data on the data bus equals the clock rate used to transfer data between internal latches, sense amplifiers, and input/output data registers. The following improvements aim to speed-up memory transfers by employing larger prefetch or by increasing the frequency on the data bus (and not the frequency of the SDRAM core). These subsequent improved versions of DDR SDRAM are numbered sequentially: DDR2, DDR3, and DDR4.

DDR SDRAMs have 2N-prefetch, and the typical frequencies of the SDRAM core and the data bus are 133, 167, and 200 Mhz. In DDR2 SDRAM devices, the number of columns prefetched is 4. Hence, **DDR2 employs 4N-prefetch**. Besides, DDR2 internal clock runs at half the DDR2 external bus clock rate. DDR2 offers data bus clock rates of 266 MHz, 333 MHz, and 400 MHz. DDR2 also lowers power by dropping the voltage from 2.5 volts (DDR) to 1.8 volts. **DDR3 increased the prefetch to 8N**. DDR3 bus clock rate is 4 times faster than DDR3 internal clock rate. DDR3 also drops the voltage to 1.5 volts and has a maximum data-bus clock speed of 800 MHz. **DDR4 also employs 8N-prefetch** but drops the voltage to 1 to 1.2 volts and has a maximum data-bus clock rate of 1600 MHz. DDR4 bus clock rate is 4 times faster than DDR4 internal clock rate.

1.11 DIMM Modules

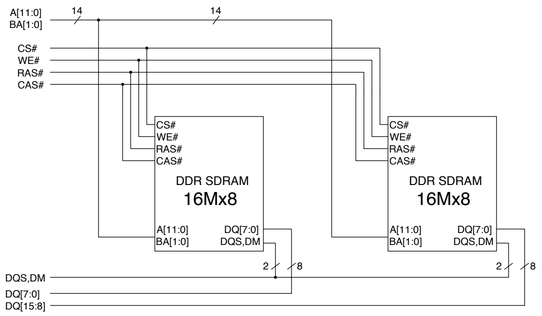


Fig. 1.44: A rank composed of two DRAM 16Mx8 chips.

The capacities of one DDR3 SDRAM chip are 1, 2, 4, and 8 Gbits, while the capacities of one DDR4 SDRAM chip are 4, 8, 16, and 32 Gbits. To increase memory capacity and bandwidth, we can connect two or more chips together, as illustrated in Figure 1.44. Each chip in Figure 1.44 is the DDR SDRAM chip from Figure 1.32, containing four banks, each of size 4096x1024x8 bits. Hence, one DDR SDRAM chip is of size 16Mx8 bits. Both chips in Figure 1.44 share the memory, the control (DQS and DM), and the command bus (CS#, WE#, RAS#, and CAS#); hence, both chips are accessed simultaneously. A set of DRAM chips connected to the same chip select (CS#) signal, which are therefore accessed simultaneously, is referred to as **a rank**. The chips in figure form a DDR SDRAM of size 16MX16 bits. Hence, connecting two DRAM chips as in Figure 1.44 we have increased the capacity and the data bus bandwidth, as now 16 data bits are transferred simultaneously.

We can further increase the size and the bandwidth od DRAM by connecting more than two chips in one rank. Figure 1.45 illustrates a rank composed of four DDR SDRAM chips of size 16Mx8 bits. Again, all four DDR SDRAM chips share the same CS# signal and are accessed simultaneously. The rank is of size 16Mx32 bits, as now 32 data bits are transferred simultaneously.

We can even form two independent ranks. In such a way, we can interleave the accesses to both ranks (similarly to bank interleaving) and mask latencies: while accessing one rank, we can activate a row in another rank or refresh another rank. Figure 1.46 illustrates two independent ranks, Ran0, and Rank1. For each rank, there is a separate CS# signal: CS0# for Rank 0, and CS0# for Rank 0. Now both ranks share the same data bus, as only one rank can be read or written at the same time.

In modern computer systems, DRAM chips are combined on a printed circuit board designed for use in personal computers, workstations, and servers. The memory chips are placed on both sides of the printed circuit board. Typically, there are eight (8) memory chips placed on one side of the printed circuit boards. A printed

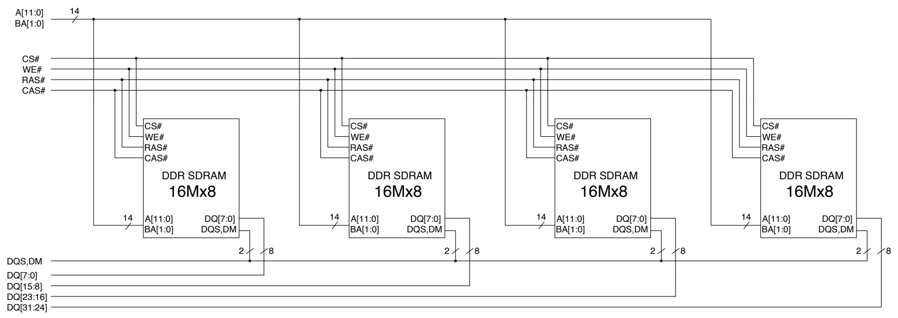


Fig. 1.45: A rank composed of four DRAM 16Mx8 chips.

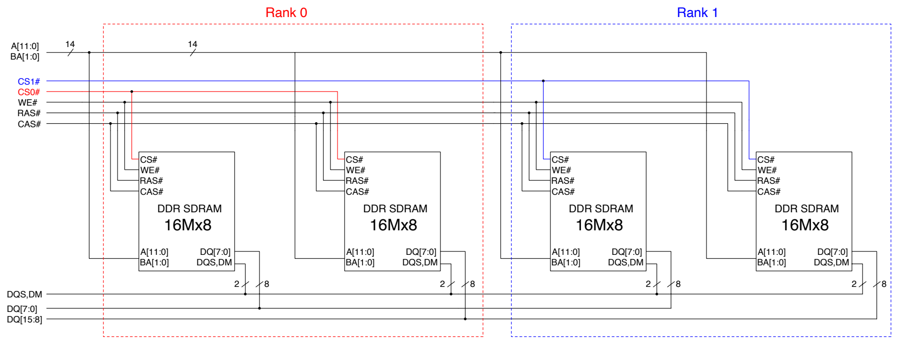


Fig. 1.46: Two ranks each containing two DRAM 16Mx8 chips.

circuit board containing memory chips on both sides is referred to as **dual in-line memory module (DIMM).** For instance, the 64-bit data bus for DIMM requires eight 8-bit chips, addressed in parallel. The DRAM chips on one side of the DIMM module form one rank: they share the same chip select (CS#) signal and are therefore accessed simultaneously. Figure 1.47 illustrates a DIMM module and its two ranks, Rank 0 and Rank 1. In practice, all DRAM chips on DIMM share all of the other command and control signals, and only the chip select pins for each rank are separate. Each side of a DIMM, containing eight 8-bit DRAM chips is one rank, and each rank has a 64-bit-wide data bus.

Manufacturers use the rather confusing labeling of SDRAM chips and DIMM modules. When DDR SDRAMs are packaged as DIMMs, they are confusingly labeled by the peak DIMM bandwidth. For example, when DDR SDRAMs with a clock frequency of 133 MHz are packed as a DIMM, the DIMM name becomes PC2100. The name comes from 133MHz x 2(DDR) x 8 bytes (eight 8-bit DRAM chips in a rank) equals 2100 MB/sec. Also, confusing names are used to label the DRAM chips. DRAM chips are labeled with the number of bits per second rather than their clock rate, so a 133 MHz DDR SDRAM chip is called a DDR266. Table

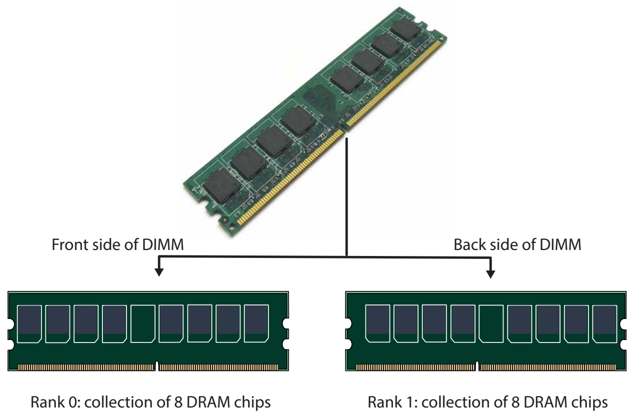

Front side of DIMM
Back side of DIMM
Rank 0: collection of 8 DRAM chips
Rank 1: collection of 8 DRAM chips

Fig. 1.47: A DIMM module.

Table 1.3: Comparison of DDR SDRAM generations and DIMMs.

Generation		**Chip**		**Data bus**		**Timings**		**DIMM**		
DRAM	**DRAM name**	**Clock (Mhz)**	**Prefetch**	**Clock (MHz)**	**MT/s**	**CL-t_{RCD}-t_{RP}**	**t_{CL} (ns)**	**MB/s**	**Voltage**	**DIMM name**
DDR	DDR-266	133		133	266	2.5-3-3	18.8	2128		PC-2100
DDR	DDR-300	150	2N	150	300			2400	2.5	PC-2400
DDR	DDR-400	200		200	400	3-3-3	15	3200		PC-3200
DDR2	DDR2-533	133		266	533	4-4-4		4264		PC2-4300
DDR2	DDR2-667	166	4N	333	667	5-5-5	15	5336	1.8	PC2-5300
DDR2	DDR2-800	200		400	800	6-6-6		6400		PC2-6400
DDR3	DDR3-1066	133		533	1066	7-7-7	13.12	8528		PC3-8500
DDR3	DDR3-1333	166	8N	666	1333	9-9-9	13.5	10664	1.5	PC3-10700
DDR3	DDR3-1600	200		800	1600	11-11-11	13.75	12800		PC3-12800
DDR4	DDR4-2400	300	8N	1200	2400	18-18-18	13.5	19200	1.2	PC4-19200
DDR4	DDR4-2666	333		1333	2666	20-20-20	13.6	21333		PC4-21333
DDR4	DDR4-3200	400		1600	3200	22-22-22	13.75	25600		PC4-25600

1.3 shows the relationships among internal and data-bus clock rates, prefetch, transfers per second per chip, chip names, DIMM bandwidth, DIMM supply voltage and and DIMM names.

DDR, DDR2, DDR3 and DDR4 memories are classified according to the maximum speed at which they can work, as well as their timings. The important memory timings of commercial memory chips are usually given as triple:

$$CL - t_{RCD} - t_{RP} \ ,$$

where CL, t_{RCD}, and t_{RP} are given in data-bus clock cycles. For example, a DDR3-1333 chip can be described as 9-9-9, meaning that CL equals nine bus clock cycles,

t_{RCD} equals nine bus clock cycles, and t_{RP} equals nine bus clock cycles. As the bus clock rate of a DDR3-1333 chip is 667MHz, all timings equal 13.5 ns.

1.11.1 Micron DDR4 DIMM module

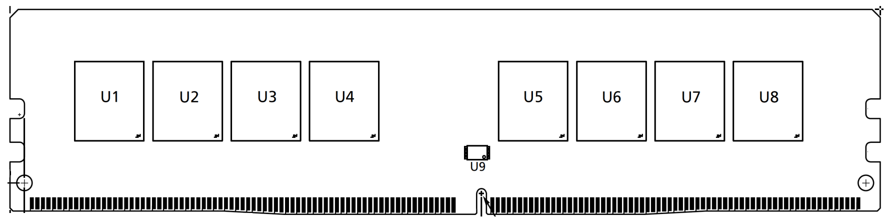


Fig. 1.48: 288-Pin Micron (1G x 64 bit) DDR4 SDRAM DIMM - Front side.

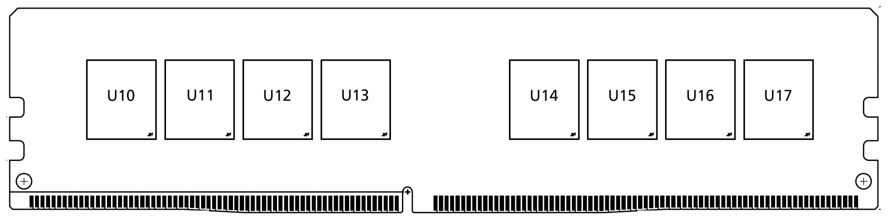


Fig. 1.49: 288-Pin Micron (1G x 64 bit) DDR4 SDRAM DIMM - Back side.

1.12 Memory channels

We have learned that multiple banks and multiple ranks enable concurrent DRAM accesses. Multiple ranks can be further used to form a **channel**, but only one rank can be activated at a time. **Multiple independent channels** serve the same purpose as multiple banks or ranks, but they are even better because they **have separate data buses**. In such a way, bus bandwidth is increased. The advantage of running two or four channels is that they will provide the same capacity as a larger single-channel, while at the same time doubling and quadrupling the amount of memory bandwidth. Of course, multiple channels bring a few disadvantages: more board wires and more

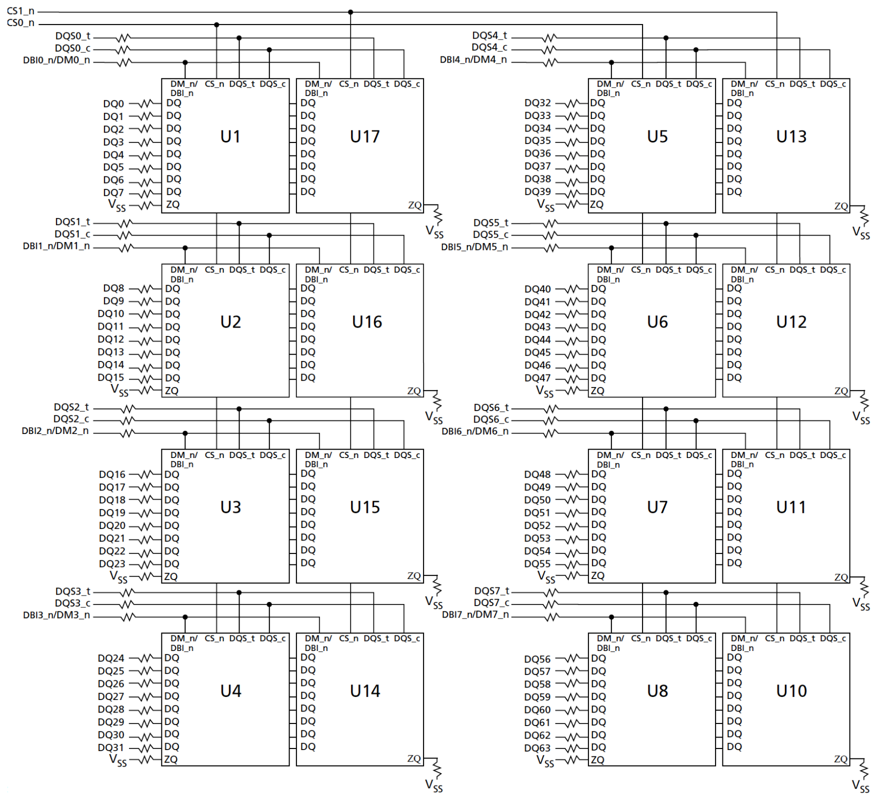


Fig. 1.50: 88-Pin Micron (1G x 64 bit) DDR4 SDRAM DIMM functional block diagram.

pins (on memory controller) are required. Multiple-channel architecture is a technology implemented on motherboards by the motherboard manufacturer and does not apply to memory modules. Also, a memory controller (which is a part of chipset) must support multiple-channel architecture. Theoretically, dual-channel configurations double the memory bandwidth when compared to single-channel configurations.

Figure 1.51 illustrates one channel formed from two ranks on the same DIMM module. Indeed, in multi-channel architectures, one channel is formed from at least one DIMM module. In today's desktop computers, up to two DIMM modules can be used to form one channel.

Most of today's computer systems support the dual channel configuration. Dual-channel-enabled memory controllers in a PC system architecture use two 64-bit data channels. For example, the Intel Core i7-800 series supported dual-channel configuration, as illustrated in Figure 1.52.

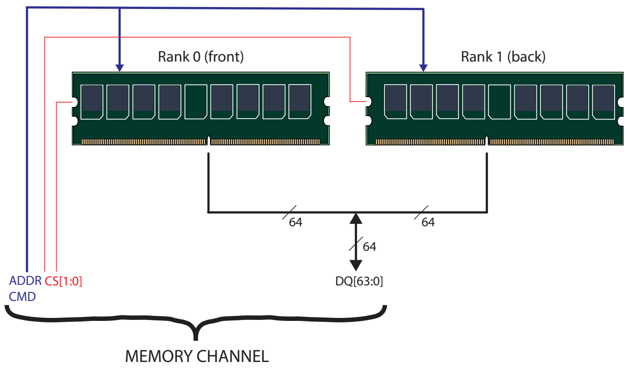


Fig. 1.51: A memory channel.

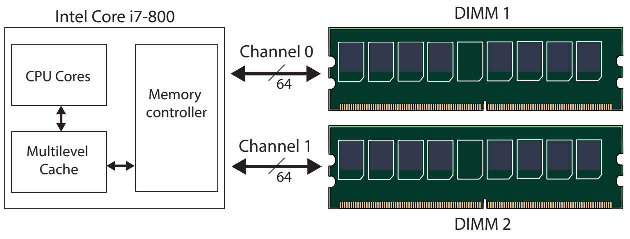


Fig. 1.52: A dual channel configuration supported by Intel Core i7-800. DIMM 1 and DIMM 2 should be identical in capacity, speed and CAS latency.

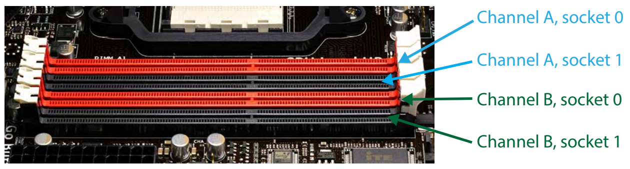


Fig. 1.53: Color codes of channels on PC motherboard.

Figure 1.53 shows a part of a motherboard that supports two memory channels. The motherboard has four DIMM sockets. To distinguish the channel's sockets on the motherboard, the sockets are color-coded. The motherboards use two colors. The colored pair of sockets is a dual channel set. A **matching pair of DIMMs** are two DIMMs that **are identical in capacity, speed, and CAS latency**. A matching pair should be used in both memory channels, i.e., a matching pair od DIMMs should be installed on the same color sockets. Another matching pair then goes in the remaining two sockets. Figure 1.54 shows two identical DIMM modules (a matching pair) inserted into the same-color sockets (red) forming two identical memory channels A and B. Ideally, all DIMM modules should be identical in a system, or else we may end up with some memory being potentially downclocked to the lowest common denominator.

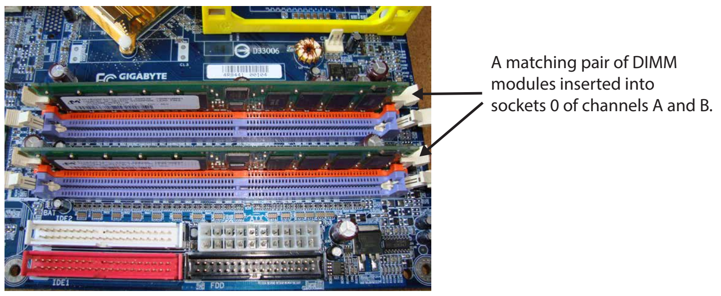

A matching pair of DIMM modules inserted into sockets 0 of channels A and B.

Fig. 1.54: A matching pair of DIMMs form two channels.

Intel Core i7-900 series DDR3 uses a triple-channel architecture, while modern high-end processors like the Intel Core i9 and AMD Ryzen Threadripper series support quad-channel memory. The quad-channel architecture can be used only when all four DIMM memory modules (or a multiple of four) are identical in capacity and speed and are placed in the same-color quad-channel sockets. When two DIMM memory modules are installed, the architecture will operate in a dual-channel mode; when three memory modules are installed, the architecture will operate in a triple-channel mode. On motherboards supporting quad-channel configuration, a similar color-coding scheme is used for dual-channel DIMM sockets. A same-color quadruple is a quad-channel set. A matching DIMM module quadruple (i.e., four DIMMs that are identical in capacity, speed, and CAS latency) should be installed on the same color sockets.

1.12.1 Case study: Intel i7-860 memory

At the beginning of the chapter, we have introduced the i7-860 and its memory hierarchy. This system is again illustrated in Figure 1.55. Now, we are going to describe the system with its real memory components and the case of an L3 miss.

The i7-860 supports up to two 64-bit memory channels, each consisting of a separate set of DDR3 1066/1333 DIMMs, and each of which can transfer in parallel. The i7-860 supports up to two DIMMs per channel and a total of up to 16 GB of memory.

In the case of L3 miss, both 64-bit memory channels are used simultaneously as one 128-bit channel (since there is only one memory controller, and the same address of the missing block in L3 is sent on both channels) to fill the missing block in L3. Using DDR3-1333 (DIMM PC3-10700), the i7-860 has a peak memory

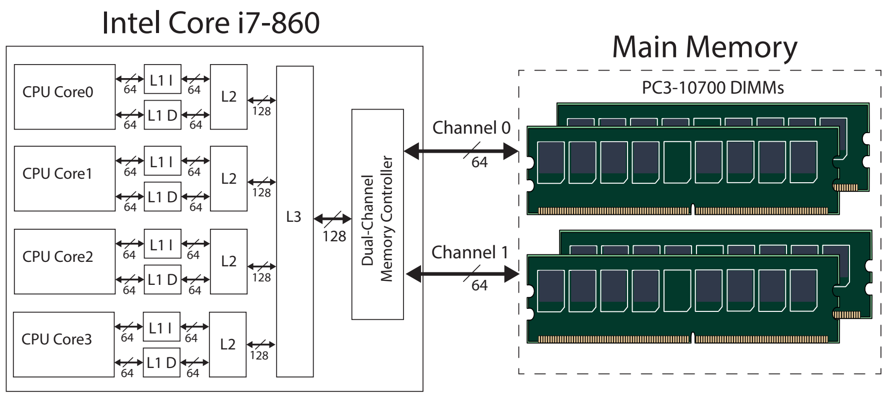


Fig. 1.55: Intel i7-860 memory.

bandwidth of just over 21 GB/sec. Thus, the memory controller fills the 64-byte cache block at a rate of 16 bytes (124 bits) per memory clock cycle.

If we assume the peak memory bandwidth, a 64-byte block is transferred at the rate of 21GB/s, which equals to 3 ns. Of course, we cannot assume that the missing block is transferred at the peak memory bandwidth. At best, we can assume that the row in SDRAMs, containing the missing block, is open. Thus, we have to add the CAS latency (CL), which equals 13.5 ns for DDR3-1333 chips. Thus, the missing block in L3 can be filled in 16.5 ns. The i7-860 runs at 2.8 GHz, which means that one CPU cycle equals 0.36 ns. Thus, the missing block in L3 will be available no prior than in 47 CPU cycles. In the case that the row containing the missing block is not open and all rows in that bank are precharged, we should add at least t_{RCD} to the above block access time. As t_{RCD} also equals 13.5 ns, the block is transferred in 29 ns or 81 CPU cycles. And finally, if we have to precharge a row before opening the row containing the missing block, the block will be transferred in 42.5 ns or 119 CPU cycles.

1.12.2 Case study: i9-9900K memory

Figure 1.56 illustrates the Intel i9-9900K system. Intel i7-9900K is an out-of-order execution processor that includes eight cores. The L1 and L2 caches are separate for each core, while the L3 cache is shared among the cores on a chip. The L1 cache is the 32 KB, eight-way set-associative cache. The L2 cache is the 256 KB, four-way set-associative cache. Finally, the L3 cache is the 16 MB, 16-way set-associative cache. The i9-9900K supports up to four 64-bit memory channels, each consisting of a separate set of DDR4-2666 DIMMs (PC4-21333), and each of which can transfer in parallel, thus the peak memory bandwidth is 41.6 GB/s. The i9-9900K supports up to two DIMMs per channel and a total of up to 128 GB of memory.

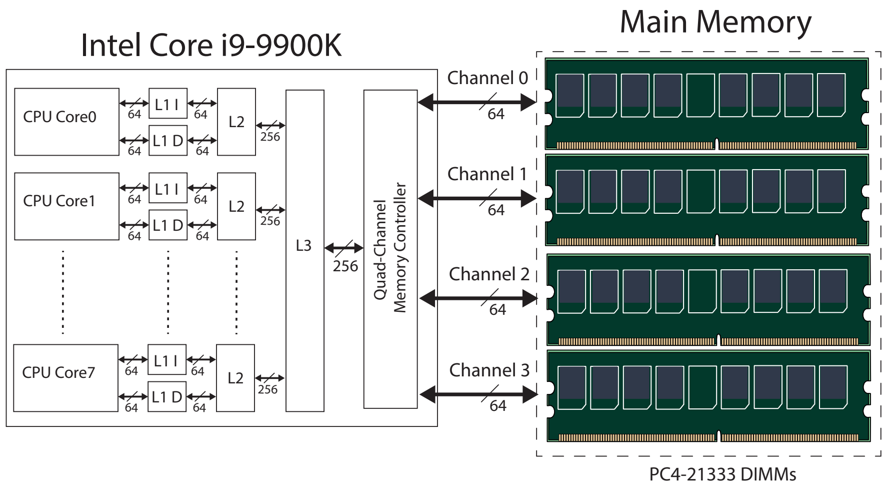


Fig. 1.56: Intel i9-9900K memory.

1.13 Bibliographical notes

TODO: The primary source of information including all details of DRAMs is the application note "Understanding DRAM Operation" [1] where basic asynchronous DRAM operation, including some of the most commonly used features for improving DRAM performance, is described.

the complete desreference guide is available.

Chapter 2
Buses

CHAPTER GOALS

Have you ever wondered how instructions and data travel between the CPU and memory or between memory and I/O devices? In this chapter, we provide a detailed explanation of computer buses. We start with a straightforward computer bus, which allows us to explain the basic concepts of computer buses and data transfers over a bus. Then we give a detailed description of some real computer buses, such as PCI, AMBA, and PCIe.
Upon completion of this chapter, you will be able to:

- Explain the basic operations on a computer bus during a data or instruction transfer.
- Explain the bus signals involved in a transfer.
- Distinguish between parallel and serial buses.
- Explain the function of the PCI bus.
- Explain the function of the AMBA bus.
- Explain the function of the PCIe bus.

2.1 Introduction

2.1.1 Parallel vs. Serial Bus

Parallel and serial data transmission are the most widely used data transfer techniques. In parallel transmission, N bits are transferred simultaneously with one clock pulse rate. Parallel transmission transmits data quickly as it utilizes several lines for sending the data. In serial transmission, data is sent bit by bit over one line. Hence, in serial transmission, only one bit is with one clock pulse rate. Intuitively, we would think that parallel data transmission should be faster than serial data transmission;

in parallel, we are transferring many bits at the same time, whereas in serial, we are transferring one bit at a time. If we simplify this to extremes, we may say that in parallel transmission, N bits are sent in the time of one clock pulse, while in serial transmission, N bits are sent over N clock pulses. Hence, one may conclude that the parallel transmission is superior in terms of speed, but this is not always true. We are going to describe the advantages and disadvantages of both data transfer methods and learn why in modern computer systems, memory busses are parallel, while I/O buses are serial. In general, modern computer busses use parallel transmissions for shorter distances (few centimeters) between CPUs and main memory, where parallel transmission provides greater speed. On the other hand, serial transmissions are reliable for transferring data over longer distances, where they can provide greater speed. Hence, modern computer busses use serial transmission for connecting I/O devices.

In parallel data transmissions, two physical phenomenons limit the speed and reliability of data transfer: **signal skew** and **crosstalk**. Parallel transmissions require at least N physical transmission channels (wires or lanes on the printed circuit board) to transmit N bits. N bits leave the transmitter at the same time, but may not be received at the receiver at the same time, as some may reach later than others because each bit travels over its own wire (lane). This difference in arrival times between signals in a group is referred to as signal skew. The traditional parallel bus has a single clock for all signals. It assumes that the clock and signals all arrive at the same time. To overcome the signal skew, the receiving end has to wait until all the bits are received. The greater the skew, the greater would be the delay. Thus, signal skew has a negative impact on the maximal frequency of the parallel bus (i.e., the speed of data transfer). We cannot increase the signal frequency for a parallel transmission without limit, because, all signals from the transmitter need to arrive at the receiver at the same time. This means that a wide parallel connection is limited in speed: we have to sen data slowly enough that all bits arrive within the same time window and are not overwritten by the next bit coming along. This cannot be guaranteed for high frequencies, as we cannot guarantee that the signal transit time is equal for all signal lines (think of different paths and geometries on the mainboard). The higher the frequency, the more tiny differences matter. Hence, signal skew could make high speed parallel communication over long transmission lines impossible. Propagation delay could be matched by tightly controlling the wire lengths and geometries, which is problematic for really long wires. This is why parallel data transmission is used in modern systems only to transfer data between main memory and CPUs. We will see in Chapter 1 that we must transfer 64 bits at every clock edge to/from memory modules. This is only possible with a parallel data bus. Hence, the memory modules must be installed very close to the CPUs.

Another problem associated with the parallel transmission is crosstalk. Crosstalk is a phenomenon where a signal transmitted on one wire creates unwanted voltage spikes in a neighboring bus wire. Crosstalk is caused by the capacitive and the inductive coupling between two wires. Hence, a switching event in one wire can result in the injection of current into the adjacent wire, causing an electrical spike. The capacitive coupling increases with the length and the number of adjacent wires. A large coupling capacitance can cause a large spike in adjacent wires that may cause a spurious switching event, potentially leading to transmission errors. The combination of higher bit rates and tightly spaced circuit traces in parallel busses leads to an increased amount of crosstalk distortion between signals. As a result, error in transmission grows significantly. Hence, the probability of a corrupted word increases and the need to retransmit it. Crosstalk can be solved by shielding the wires from each other, further driving up the cost.

Serial transmission is slower than parallel transmission, given the same signal frequency. But, by sending data in serial (bit-by-bit), we no longer need to worry about signal skew. Serial transmission thus solves synchronization issues with some signals traveling faster than others. Besides, if we use only one signal line, we are free to switch it as fast as the hardware allows, as we do not have to worry about crosstalk. We can use two serial data lines to transmit and receive signals simultaneously. This is almost impossible with the parallel transmission, as this would require several hundred signal lines. Hence, we say that serial communication is full-duplex, whereas parallel communication is half-duplex.

2.2 A simple bus

2.3 PCI bus

2.4 AMBA bus

2.5 PCIe bus

Chapter 3
Interrupts and interrupt handling

CHAPTER GOALS

Have you ever wonder how computer components demand and get attention from the CPU? How do they tell the CPU or operating system that something important has just happened in the computer system, which requires an immediate response from the CPU, e.g., new data has just arrived at an I/O interface and should be processed immediately? This is done using so-called interrupts. In this chapter, we will cover the theory and practice of interrupts and their handling. An interrupt is a hardware-initiated procedure that interrupts whatever program (CPU) is currently executing and requests that the CPU immediately start running another program that is written to service the particular interrupt request.
Upon completion of this chapter, you will be able to:

- Distinguish between interrupts and exceptions..
- Explain the operation of the interrupt signals.
- Explain the interrupt and exception handling.
- Explain the function of interrupt vectors and vector tabels.
- Explain the function of an interrupt controller.
- Explain the interrupts and interrupt handling in Intel and ARM family of processors.

3.1 Introduction

During my childhood, there were two powerful military blocs in Europe and the world: the Eastern (Soviet) Block and the Western (USA) Block. That was a period of geopolitical tension between the Soviet Union and the United States and their respective allies, the Eastern Bloc and the Western Bloc. The country where I grew up, former Yugoslavia, was not part of any of these military blocks, though politically, it was closer to the eastern block. In the 1970s, former Yugoslav air force purchased a number of Soviet MIG-21 fighter aircraft from the USSR. The MIG-21 aircraft sold to Yugoslav air force had virtually no modern electronic devices, and the military of Yugoslavia wanted to install missile sensors in the planes. However, the USA and its allies have imposed an embargo on the purchase of electronic and computer components against Yugoslavia. Among all the universities in Yugoslavia, only the University of Ljubljana was allowed to purchase a few pieces (up to 20) of each chip that would be used only in the educational process. That's why the Yugoslav Army approached the University of Ljubljana to buy all the necessary electronic and computer components and develop a system that would be installed on the aircraft and would detect missiles. The system at the time had to be based on the modern Motorola 6800 microprocessors from the US. At its core, the system had a microcomputer built on the Motorola 6800 processor and a missile sensor. In addition to detecting missiles, the microcomputer had to do other things, also. If the missile sensor detected a rocket, the computer system had to immediately stop whatever it was currently doing and alert the pilot to the approaching missile. But how would a missile sensor be able to communicate to the CPU if the CPU could do nothing but fetch and execute instructions from memory? Remember that the CPU fetches and executes instructions every clock cycle. That's all it is able to do. So there must be some mechanism by which the CPU can be immediately interrupted and required to start another program. In our case, the CPU would run another program (e.g., display the current altitude and speed of the aircraft). In the event that the sensor detects a missile, it must, in some way, immediately suspend the currently running program and require the CPU to execute a program to flash the warning lights and alert the pilot. So, the CPU must have some mechanism in place to immediately stop the execution of one program and start another program. This mechanism is called **interrupts**, and the program that the CPU starts running in the response to an is called **interrupt service program (ISP)** or **interrupt handler**.

Interrupts and interrupt handling must be **transparent**. This means that the stopped (interrupted) program must not know that it has been stopped and must continue after the termination of the interrupt service program as if it had not been interrupted at all.

In the following chapters, we will learn about the interrupt mechanism and interrupt handling.

3.2 Interrupts

As we said in the Introduction, we want to have to ability to service external interrupts. This is useful if a device external to the processor needs attention. Figure 3.1 illustrates a simplified system with a CPU and a peripheral device. To be able to respond to interrupt requests from a peripheral device, a CPU usually has at least one interrupt request (IRQ) pin and one interrupt acknowledge (INTA) pin. The IRQ pin is the input used by a peripheral device to interrupt the processor (i.e., to interrupt the normal program flow in the CPU.). Since the CPU should finish executing the current instruction(s) before servicing any external interrupts, the peripheral device may have to wait for several clock cycles before the CPU responds to the interrupt request. The INTA pin is the output used to signal the peripheral device, which has requested an interrupt via the IRQ signal, that the CPU has started servicing the interrupt request and that the IRQ signal can be deactivated. Both pins in Figure 3.1, IRQ and INTA, are active low. Two resistors are used to establish a logic one on both signals IRQ and INTA (i.e., both signals are deactivated) when no one drives them.

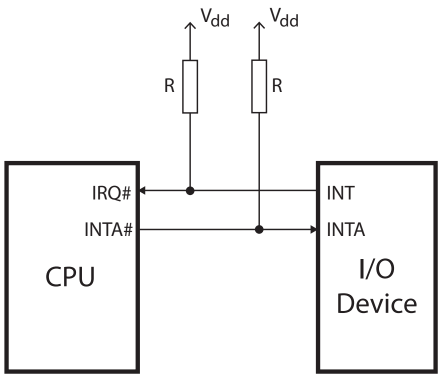


Fig. 3.1: A simplified block diagram of a computer system with interrupt controlling signals.

In general, CPUs can respond to interrupts in two different ways: in either an **edge-sensitive** or **level-sensitive** manner. In an edge-sensitive manner, the interrupt signal input is designed to be triggered by a particular signal edge (level transition): either a falling edge (high to low) or a rising edge (low to high). In a level-sensitive manner, the interrupt signal input is designed to be triggered by a logic signal level. A peripheral device invokes a level-triggered interrupt by driving the signal to and holding it at the active level. We refer to this operation as **asserting the signal**. It de-asserts the signal when the processor signals it to do so. One advantage of level-triggered interrupt inputs is that they allow multiple devices to share a common interrupt signal. Most often, the active level of an interrupt input signal is LOW. In such a case, the interrupt signal is tied to the HIGH voltage level using a pull-up resistor. When multiple peripheral devices share one level-triggered interrupt input signal, the device that wants to assert the interrupt request simply connects the signal to the ground (pulls thew signal LOW). The system in Figure 3.1 uses level-sensitive interrupt signals.

Summary: Assering and de-asserting a signal

Some signals are active high, and some signals are active low. To avoid the problem of high vs. low and the fact that for some signals, active means high and for some signals active means low, we just say asserted (activated) vs. de-asserted (deactivated).

When the device needs the attention from the CPU, it activates (asserts) the IRQ pin on the CPU. During the normal flow of execution through a program, the program counter increases sequentially through the address space, with branches to nearby labels or branches and links to subroutines. The CPU checks the status of the IRQ pin every time before a new instruction pointed to by the program counter is fetched from memory. When a peripheral device requests the interrupt, it is necessary to preserve the previous processor status while handling the interrupt, so that execution of the program that was running when the interrupt request occurred can resume when the appropriate interrupt handler has completed. We say that the interrupts must be 100% transparent. So, when an interrupt request occurs, the CPU

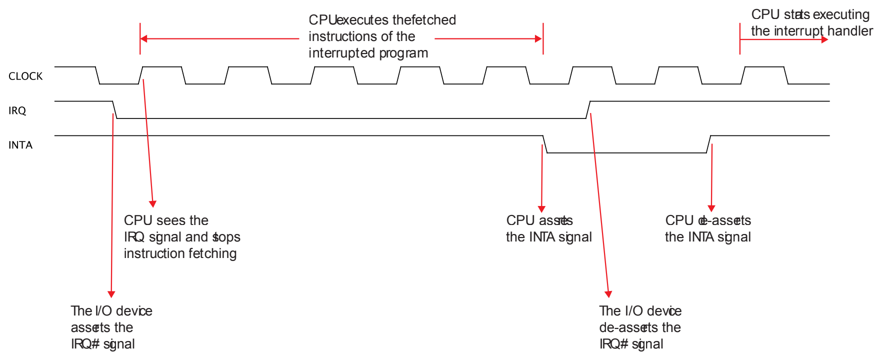

CPU executes the fetched instructions of the interrupted program
CPU starts executing the interrupt handler
CLOCK
IRQ
INTA
CPU sees the IRQ signal and stops instruction fetching
CPU asserts the INTA signal
CPU de-asserts the INTA signal
The I/O device asserts the IRQ# signal
The I/O device de-asserts the IRQ# signal

Fig. 3.2: A timing diagram for an external interrupt request.

completes the current instruction and asserts the INTA signal. When a peripheral device sees the INTA signal, it de-asserts the IRQ signal. Figure 3.2 shows the timing diagram for an external interrupt request for the simple system from Figure 3.1.

Then the CPU saves the part of the context of the interrupted program in the stack. A context is a state of the program counter, status register, stack pointer, and all other program-visible CPU registers. Some CPUs save the whole context in the stack, while others save only a part of the context in the stack. Since interrupts can happen at any time, there is no way for the active programs to prepare for the interrupt (e.g., by saving registers that the interrupt handler might write to). It is important to note that calling conventions do not apply when handling interrupts: the interrupt is not being "called" by the active program; it is interrupting the active program. Thus, the interrupt handler code must preserve the content ensure that it does not overwrite any registers that the program may be using before their content is saved. After the CPU has saved the context, the CPU automatically loads the address of the interrupt handler into the program counter. The interrupt handler is a program written by the user and depends on the peripheral device's functionality. Depending on how much of the context is automatically saved by the CPU, the interrupt handler must first save every register it intends to use in the stack or somewhere in memory.

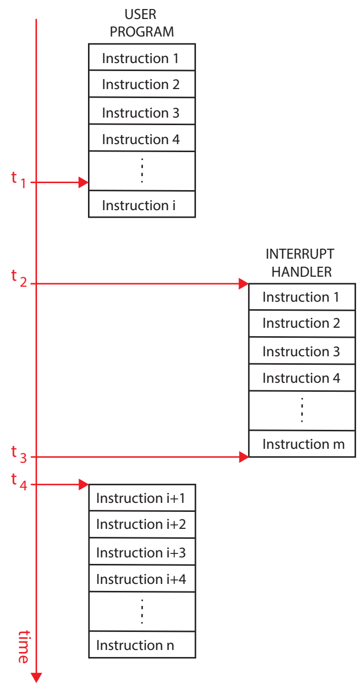


Fig. 3.3: The procedure involved in interrupts.

Figure 3.3 shows the procedure involved in interrupts: the CPU executes the sequence of instructions from a user program until an interrupt request occurs at the time t_1. When the IRQ signal is asserted, the CPU stops executing the user code and starts executing the interrupt handler. But before executing the interrupt handler at time t_2, the CPU must finish the execution of already fetched instructions, save the (part of) context, and obtain the address of the interrupt handler. The time $t_2 - t_1$ required for this procedure is called **interrupt latency**. In general, interrupt latency is the time that elapses from when the IRQ signal is asserted to when the CPU starts to execute the interrupt handler. Interrupt latency duration is usually not predetermined and depends on how many instructions are already in the CPU's pipeline, on how CPU saves the context and on whether any new interrupt requests are temporarily disabled. Once the CPU completes the execution of the interrupt handler at time t_3, it returns back to the execution of the user code at time t_4. Before returning to user code, the CPU must automatically restore the previously saved context.

3.2.1 *Types of interrupts*

There are typically three types of interrupts regarding the source of the interrupt: external interrupts (or simply interrupts), traps or exceptions, and software interrupts. External interrupts are triggered by an external device by activating the interrupt request pin on the CPU. Traps or exceptions are activated internally in the CPU, usually as a result of some exceptional condition caused by instruction. For example, traps are caused when illegal or undefined instruction is fetched, or when the CPU attempts to execute an instruction that was not fetched because the address was illegal. A special instruction triggers software interrupts. Such instructions function similarly to subroutine calls, but the subroutine, in this case, the interrupt handler, is not being "called", but an interrupt-like sequence occurs. These software-interrupt instructions are useful when the user program does not know or is not allowed to know the address of the routine which it would like to "call", e.g., they are usually used for requesting operating system services and routines.

External interrupts are divided into two types: maskable and non-maskable interrupts. Maskable interrupts can be enabled or disabled by setting a bit in the CPU's control register or by executing a special instruction. For example, Intel has the CLI instruction to mask the interrupts, and ARM has CPSID instruction for this purpose. Non-maskable interrupts have a higher priority than maskable interrupts. That means that if both maskable and non-maskable interrupts are activated at the same time, the CPU will service the non-maskable interrupt first.

3.2.2 *Handling interrupts*

In a situation where multiple types of interrupts and exceptions can occur, there must be a mechanism in place where different handler code can be executed for different types of events. In general, there are two methods for handling this problem: polled interrupts and vectored interrupts.

In polled interrupts, the processor branches to a specific address that begins a sequence of instructions that check the cause of the interrupt or exception and branch to handler code for the type of interrupt/exception encountered. This is also called polled interrupt/exception handling.

In vectored interrupts, the processor branches to a different address for each type of interrupt or exception. Each exception address is separated by only one word, and these addresses form a table called **interrupt vector table**. Each entry of the interrupt vector table is called **interrupt vector**, and it is the address of an interrupt handler. Hence, the vector table contains the start addresses, called interrupt vectors, for all exception handlers. This method is called **vectored interrupt handling**. This concept is common across many processor architectures, although interrupt vector tables may be implemented in other architecture-specific fashions. For example, another common concept is to place a jump instruction (instead of vectors) at each entry in the table. Each of these jump instructions forces the processor to jump to the handler code for each type of interrupt/exception. In this case, the address of each table entry is considered as an interrupt vector.

3.2.3 ARM 9 interrupts

The ARM9 supports the following six types of interrupts and exceptions:

- Fast interrupt Request,
- Interrupt Request,
- Data and Prefetched abort exceptions,
- Undefined instruction exception, and
- Software interrupt, and
- Reset.

The interrupt instruction SWI raises the software interrupts. The software interrupts allow a program running in the user mode to request privileged operations such as OS functions. The Prefetch abort exception occurs when the CPU fetches an instruction from an illegal address. The Data abort exception occurs when a data transfer instruction attempts to load or store data at an illegal address. The Undefined instruction exception occurs when the processor cannot recognize the currently fetched instruction. The Interrupt request occurs when the processor's external interrupt request pin (IRQ) is asserted (LOW), and the interrupt mask bit (I) in the current program status register (CPSR) is cleared (interrupts enabled). The Fast interrupt request occurs when the processor's external fast interrupt request pin (FIQ) is asserted (LOW), and the interrupt mask bit (F) in the current program status register (CPSR) is cleared (fast interrupts enabled). The Reset interrupt occurs when the processor's reset pin is asserted.

3.2.3.1 Vector table and interrupt priorities

ARM9 processors use the vectored interrupt handling method. Each interrupt/exception has its own entry in the vector table. Each entry in the vector table has only 32 bits, which is not enough to contain the full code for a handler; hence, each entry commonly contains a branch instruction or load pc instruction to the actual handler. Table 3.1 shows the interrupt/exception, its address in the vector table, and its priority. As interrupts/exceptions can coincide, the CPU has to use a priority mechanism to handle the most important interrupt/exception. For example, the Reset interrupt

Table 3.1: ARM9 vector table.

Interrupt/Exception	**Vector Table Address**	**Priority (1-High, 6-Low)**
Reset	0x00000000	1
Undefined Instruction	0x00000004	6
Software Interrupt	0x00000008	6
Prefetch Abort	0x0000000C	5
Data Abort	0x00000010	2
Interrupt Request	0x00000018	4
Fast Interrupt Request	0x0000001C	3

has the highest priority, and it takes precedence over all other interrupts/exceptions. All interrupts/exceptions disable further interrupts/exceptions by setting the I bit in the CPSR register. The Reset and Fast Interrupt Request also set the F bit in the CPSR register and thus mask the Fast interrupt request. Listing 3.1 shows a typical method of implementing a vector table for ARM9 processors.

```
        .org 0x00000000
Vector_Table:
        b Reset_Handler
        b Undefined_Handler
        b SWI_Handler
        b Prefetch_Handler
        b Abort_Handler
        nop                     // never used
        b IRQ_Handler
        b FIQ_HAndler


Reset_Handler:
        <handler instructions>
Undefined_Handler:
        <handler instructions>
SWI_Handler:
        <handler instructions>
Prefetch_Handler:
        <handler instructions>
Abort_Handler:
        <handler instructions>
IRQ_Handler:
        <handler instructions>
FIQ_Handler:
        <handler instructions>
```

Listing 3.1: ARM vector table and interrupt handlers.

Listing 3.1 shows a typical method of implementing a vector table for ARM9 processors. The vector table starts at the address 0x00000000. Each entry in the vector table is 32 bits long and contains a branch instruction (B) to the interrupt handler. When, for example, a Data Abort exception occurs, the CPU stops the execution of the current running program, saves the program context, and moves the vector

`0x00000010` into the program counter. This way, the `b Abort_Handler` instruction is fetched, and the CPU jumps to `Abort_Handler`.

As we already said, the Reset interrupt is the highest priority interrupt and is always taken whenever the Reset pin is asserted. The reset handler is responsible for initializing the system and other interrupt sources, and to set the stack pointer. So the Reset interrupt masks automatically all other interrupts before their sources are initialized. Only then the reset handler enables other interrupts. Hence, during the first few instructions of the reset handler, we should avoid SWI, undefined instructions, and memory accesses that can cause the Data and Prefetch aborts.

The Fast Interrupt Request (FIQ) occurs when a peripheral asserts the processor's FIQ pin. The peripheral device mus hold the FIQ input low until the processor acknowledges the interrupt request. As a response to FIQ, the CPU disables both Interrupt and Fast Interrupt requests. Hence, no external device can interrupt the CPU unless the IRQ and FIQ interrupts are re-enabled by software. The Fast Interrupt Request reduces the execution time of the exception handler relative to a normal interrupt by removing the requirement for register saving (minimizing the overhead of context switching).

The Interrupt Request (IRQ) is a normal interrupt that occurs when a peripheral device asserts the IRQ pin. The peripheral device mus hold the IRQ input pin low until the processor acknowledges the interrupt request. An IRQ has a lower priority than the FIQ and Data Abort and is masked on entry to an FIQ or Data Abort sequence. On entry to the IRQ handler, the further IRQ interrupts are disabled and should remain disabled until the current interrupt source has been acknowledged, and the IRQ pin has been de-asserted.

We can notice from Table 3.1 that both Software Interrupt and Undefined Instruction have the same level of a priority since they cannot occur at the same time.

3.2.3.2 ARM9 interrupt handling

ARM9 processors are 5-stage pipelined machines with Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM) and Write-Back (WB) stages. In a pipelined machine, an instruction is executed step by step and is not completed for several clock cycles. An external interrupt can occur at any time during the execution of an instruction. Also, other instructions in the pipeline can raise exceptions that may force the machine to abort the instructions in the pipeline before they have been completed. One of the problems with interrupts in the pipelined CPUs is when to halt instruction in the pipeline. In the case of external interrupts, one possible solution would be to execute all fetched instructions before handling the interrupt request. But the problem with this approach would be a long interrupt latency. The other solution would be to halt the execution of all fetched instructions and fetch them again upon returning from the interrupt handler. This way, we would have minimal interrupt latency. Obviously, this is not a good idea because some instructions, such as STORE instructions, can modify the content in memory and should not be stoped and executed again. Also, arithmetic instructions might have already changed the content of the status register (usually in the Execution stage), and should not be dismissed. The most common solution to the problem is to execute all instructions that have been issued into the execution stage. In the case of an external interrupt in ARM9, the CPU executes all instructions in the stages EX, MEM nad WB, while dismissing two instructions in the stages IF and ID.

To resume, in the case of an external interrupt, the CPU has to let all instructions that were issued for execution complete and flush all succeeding instructions from the pipeline. In the case of an exception caused by an instruction, the CPU should stop executing the offending instruction, let all preceding instructions complete and flush all succeeding instructions from the pipeline. Only then can CPU start saving the context and fetching the instruction pointed by the interrupt vector (the first instruction in the interrupt handler).

Let us now look at how ARM9 handles the IRQ interrupts. When an IRQ interrupt occurs, the ARM 9 processor executes the three instructions that are issued for execution and will flush the last two fetched instructions. The last two fetched instructions are from the addresses PC (the instruction currently in the IF stage) and PC-4 (the instruction currently in the ID stage). The instruction in the EX stage is from the address PC-8. This is very important to notice because the last executed instruction before entering the interrupt handler was from the address PC-8, but the program counter contains the address PC. The first instruction to execute upon returning from the interrupt handler is one that was in the ID stage when the interrupt request occurs. Hence, the address of the instruction that should be fetched upon returning from the interrupt handler is PC-4.

When an IRQ interrupt occurs, the ARM9 processor executes the instructions that are issued for execution. Then, the following hardware procedure is executed:

- the CPU saves the Current Program Status (CPSR) register into the Saved Program Status (SPSR) register; hence the processor automatically saves the status of the interrupted program. The CPSR register is a special purpose register in ARM9 processors that contains arithmetic flags and interrupt masks,
- the CPU automatically disables interrupts by setting the I bit in the CPSR register,
- the CPU saves the current program counter (PC) into the link register (LR). This way, the LR register holds the return address. It is important to note that the CPU saves the address of the last fetched instruction and does not automatically correct this value to point to the instruction that was in the ID stage when the interrupt occurs. Hence, it is the programmer's responsibility to adjust the value in PC upon returning from the interrupt handler, and
- the CPU fetches the instruction from the interrupt vector 0x00000018.

Now, the interrupt handler starts. The above procedure is hard-wired in the CPU and does not involve any instruction fetch and execution. When an interrupt handler has completed, it must move both the return value in the LR register minus 4 to the PC and the SPSR to the CPSR. This action restores both the PC and the CPSR and returns to the interrupted program. Listing 3.2 shows a typical method of returning from an IRQ interrupt handler.

```
IRQ_Handler:
        <handler instructions>
        ...
        ...
        subs pc, lr, #4     // pc <- lr-4
```

Listing 3.2: A typical IRQ interrupt handler

Many instructions in ARM9 can have an "s" suffix. The "s" suffix ensures that when the program counter is the destination register, the CPSR register is automatically restored from the SPSR register. The same holds for the `subs` instruction in Listing 3.2. Hence, the instruction `subs pc,lr,#4` firstly saves the LR-4 into the program counter (remember that the programmer is responsible to correctly restore the return address into the program counter upon returning from the handler) and then restores CPSR from SPSR.

It is important to stress that not all interrupt/exception handlers use the same instruction to return. For example, the Data abort exception occurs in the MEM stage. Hence, only the instruction in WB stage is executed, while the instructions from IF, ID, and EX stages are flushed. When the Data abort exception occurs, the instruction in the EX stage is from the address PC-8. Thus, the Data abort handler uses `subs pc,lr,#8` to return:

```
IRQ_Handler:
        <handler instructions>
        ...
        ...
        subs pc, lr, #8     // pc <- lr-8  !!!!
```

Listing 3.3: A typical Datta abort exception handler

3.2.4 *Intel interrupts*

Intel processors have two external pins for external interrupts:

- INTR pin - it is used to signal for normal (maskable) interrupts.
- NMI pin - it is used to signal nonmaskable interrupts

Besides interrupts, Intel processors can detect exceptions from two sources:

- Processor exception - triggered form processor as a result of some exceptional conditions within the processor (e.g., divide by zero). These exceptions are further classified as faults, traps, and aborts.
- Software interrupts - triggered with the processor instruction INT.

Exceptions are classified as:

- Faults are either detected before the instruction begins to execute or during the execution of the instruction. A fault is an exception that can generally be corrected, and that, once corrected, allows the program to be restarted with no loss

of continuity. The return address for the fault handler points to the faulting instruction, rather than to the instruction following the faulting instruction.
- A trap is an exception that is reported immediately following the execution of the instruction INT. Traps allow the execution of a program or task to be continued without loss of program continuity. The return address for the trap handler points to the instruction to be executed after the trapping instruction.
- An abort is an exception that does not allow a restart of the program or task that caused the exception. Aborts are used to report severe errors.

The Intel processor services interrupts and exceptions only between the end of one instruction and the beginning of the next. This is referred to as the instruction boundary. Certain conditions and flag settings cause the processor to inhibit certain interrupts and exceptions at instruction boundaries. The IF (interrupt-enable flag) bit in the FLAGS register (this is the status register in Intel x86 microprocessors that contains the current state of the processor.) controls the acceptance of external interrupts signaled via the INTR pin. When IF=0, INTR interrupts are masked; when IF=1, INTR interrupts are enabled. The Intel processor instructions CLI (Clear Interrupt-Enable Flag) and STI (Set Interrupt-Enable Flag) are used to clear/set the IF flag.

If more than one interrupt or exception is pending at an instruction boundary, the processor services one of them at a time according to their priority. In general, aborts have the highest priority, followed by traps, NMI, and INTR. The faults have the lowest priority.

Each architecturally defined exception and interrupt in Intel processors is assigned a unique identification number, called a **vector number**. The processor uses the vector number assigned to an interrupt as an index into the interrupt vector table. The allowable range for vector numbers is 0 to 255. The Intel architecture reserves vector numbers in the range 0 through 31 for architecture-defined exceptions and interrupts. Vector numbers in the range 32 to 255 are designated as user-defined interrupts and are assigned to external I/O devices to enable those devices to send interrupts. One characteristic of Intel processors, which distinguish them from ARM processors, is that the peripheral device that caused an interrupt must provide the vector number to the CPU. Table 3.2 shows vector number assignments and exception types for architecturally defined exceptions and interrupts.

In the older Intel processors (before 80386), the interrupt table is called IVT (interrupt vector table). The IVT is an array of 32-bit interrupt vectors stored consecutively in memory and indexed by an interrupt vector. The IVT always resides at the same location in memory, ranging from 0x0000 to 0x03ff, and consists of 256 four-byte interrupt vectors (i.e. pointers to the interrupt/exception handles). When responding to an exception or interrupt, the processor multiplies the vector number by four to form the address of the entry in the IVT.

In modern Intel processors, the interrupt table is called IDT (interrupt descriptor table). The IDT is an array of 8-byte descriptors stored consecutively in memory and indexed by an interrupt vector. Each descriptor holds the information that describes how to access the interrupt/exception handler. The IDT may reside anywhere in physical memory. The processor has a special register (IDTR) to store both the

Table 3.2: Intel Exceptions and Interrupts. Only a few exceptions and interrupts are shown.

Vector Number	Description	Type
0	Division by zero	Fault
1	Debug	Fault
2	NMI	Interrupt
3	Breakpoint	Trap
...	...	...
	...	...
14	Page Fault	Fault
...	...	...
32-255	External interrupts on INTR	Interrupt

physical base address and the length in bytes of the IDT. When an interrupt occurs, the processor multiplies the interrupt vector by eight and adds the result to the IDT base address. With the help of the IDT length, the resulting memory address is then verified to be within the table; if it is too large, an exception is generated. If everything is okay, the 8-byte descriptor stored at the calculated memory location is loaded, and actions are taken according to the descriptor's contents. As said, the interrupt descriptor table (IDT) associates each vector number with a descriptor for the instructions that service the associated event. Because there are only 256 vector numbers, the IDT contains up to 256 descriptors. It can contain fewer than 256 entries; entries are required only for vector numbers that are actually used.

The interrupt handling procedure in the Intel processor is rather complicated. Here, we omit all the details and give only the basic concepts. When responding to an exception or interrupt, the processor first saves the current state of the interrupted program or task (the status FLAGS register and the program counter) on the stack. Each entry in the IDT (or IVT) holds the start address of the interrupt handler. The processor thus reads the start address of the handler from the IDT (or IVT) into the program counter and starts the execution of the handler. To return from an exception- or interrupt-handler handler, the handler uses the IRET instruction. The IRET instruction is similar to the RET instruction used to return from normal procedures except that it restores the saved status register FLAGS.

3.2.5 Interrupt handlers in C

Interrupt handlers can be written in assembler or in a high-level language like C. Usually, we want to avoid the assembly language as much as possible and to program in our favorite high-level language. Remember that an interrupt handler is called directly by the CPU, and the protocol for calling an interrupt handler differs from calling a C function. Most importantly, an ISR has to end with some "interrupt return" opcode, whereas usual C functions end with ordinary "return" opcode. We have seen previously that the ARM interrupt handlers should return with SUBS opcode, which is used to restores the PC from LR-4 and CPSR from SPSR. In the case of an ordinary subroutine, the return opcode for ARM would be MOV PC, LR (restores PC from LR). A programmer could be tempted to write an interrupt handler like this:

```
/* How NOT to write an interrupt handler */
void my_interrupt_handler(void)
{
    /* do something */
}
```

Listing 3.4: How not to write an interrupt handler.

This simply cannot work. The compiler doesn't understand that this is to be an interrupt handler and that the SUBS PC,LR,#4 instruction should be the last instruction used to return. The compiler will simply use the MOV PC, LR instruction to return.

Some compilers, such as GCC, Clang, and ARMCC, to name a few, have directives like #pragma or special function attributes, allowing you to declare a routine interrupt. For example, the *interrupt* function attribute in GCC indicates that the specified function is an interrupt handler. The compiler then generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

The correct (GCC) way of implementing an interrupt handler in C is:

```
/* GCC style interrupt handler */
__attribute__((interrupt)) void my_interrupt_handler()
{
    /* do something */
}
```

Listing 3.5: GCC style interrupt handler.

The ARMCC compiler offers the `__irq` function declaration keyword to write C interrupt handlers. The `__irq` keyword preserves all registers used by the interrupt handler and exits the handler by setting the PC to (LR–4) and restoring the CPSR to its original value from SPSR. Also, if the kernel calls a subroutine, `__irq` preserves the link register (LR), which is corrupted by the subroutine call.

```
/* GCC style interrupt handler */
__irq void my_interrupt_handler()
{
    /* do something */
}
```

Listing 3.6: ARMCC style interrupt handler.

But it is not only the directive or function qualifier that designates the interrupt handlers. Often, compilers require that the handler declaration contains a special function argument, which specifies the kind of interrupt (for example, IRQ or

Abort). The compiler uses this special argument to restore the PC from LR correctly (for example, LR-4 for IRQ or LR-8 for Data abort). All these attributes and arguments defined and used by a particular compiler prevent the handler code from being portable.

3.3 Interrupt controllers

We have seen that the interrupt line from a peripheral device should be connected to the CPU's interrupt input signal. In such a way, a peripheral device can interrupt the CPU and require its attention. The CPU will sense this interrupt input signal at every instruction fetch and know that the peripheral device needs attention. But what should we do if there is more than one peripheral device that would like to interrupt the CPU? What if there are tens of external peripheral devices, which is often the case in real computer systems? Should we add an interrupt input pin to the CPU for every external peripheral device? A large number of interrupt input pins on the CPU for every external device would make the CPU interfacing very complicated and increase the error probability.

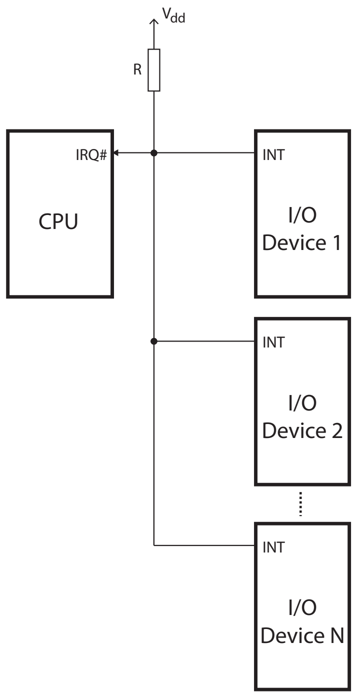


Fig. 3.4: Example system with several I/O devices sharing one interrupt input signal.

One possible solution to solve this problem would be to have one level-sensitive interrupt input pin (IRQ) on the CPU shared by all external peripheral devices. This solution is illustrated in Figure 3.4. Whenever an interrupt is asserted, the CPU branches to the interrupt handler associated with the IRQ pin. This interrupt handler would poll each and every I/O peripheral device to determine which device asserted the interrupt line. So, the CPU will handle the interrupt request on the IRQ pin as vectored interrupts, but will also, within the interrupt handler, use the polled interrupts method to check the interrupt's cause. Every modern I/O peripheral device has an addressable (memory-mapped) status register. There is usually one bit in this status register, referred to as an interrupt-pending bit, which is set internally by the interrupting device when the device asserts the interrupt line. This interrupt-pending bit in the status register can be read by the CPU to determine the interrupting device. If the interrupt-pending bit is set, the processor branches to a specific device-service routine. Within this device-specific routine, the interrupt handling bit is cleared and the interrupt request is serviced. Listing 3.7 shows pseudocode for the IRQ interrupt handler that uses polling to determine which device has requested the interrupt. The downside to this technique is that it is time-consuming.

```
/*
* Polling Handler
*/
__attribute__((interrupt)) void polling_IRQ_handler () {

    /* Check the interrupt pending bit in the I/O device 1 */
    if (IO1_status_reg & (1<<INT_PEND_BIT)) {
    /* I/O Device 1 Code */
    IO1_status_reg &= ~(1<<INT_PEND_BIT); // clear int pending bit

    /* Do something */

    }

    /* Check the interrupt pending bit in the I/O device 2 */
    if (IO2_status_reg & (1<<INT_PEND_BIT)) {
    /* I/O Device 2 Code */
    IO2_status_reg &= ~(1<<INT_PEND_BIT); // clear int pending bit

    /* Do something */

    }

    ...

    /* Check the interrupt pending bit in the I/O device N */
    if (ION_status_reg & (1<<INT_PEND_BIT)) {
    /* I/O Device N Code */
    ION_status_reg &= ~(1<<INT_PEND_BIT); // clear int pending bit

    /* Do something */

    }
}
```

Listing 3.7: IRQ interrupt handler with polling.

A better solution to this problem would be to use a special chip — an **interrupt controller**. The interrupt controller is a special device, which:

- combine all external interrupt requests onto one CPU IRQ line,
- prioritizes them (decides which interrupt request will be routed to the CPU when more than one I/O device has requested the interrupt),
- routes the selected interrupt request to the CPU's IRQ input signal, and
- most importantly, it provides the CPU with the information which device has requested the interrupt. Commonly, it provides the CPU with the interrupt vector; thus, the CPU does not have to poll I/O peripheral devices.

Figure 3.5 illustrates the structure of a system that uses an interrupt controller. All potential external interrupt sources are routed through the interrupt controller. In the case of one or more interrupt requests, the interrupt controller prioritizes the interrupt inputs, it transfers the interrupt request with the highest priority to the CPU, along with interrupt vector. This sequence is performed by hardware in the interrupt controller and not by software in the CPU, hence the interrupt controller provides a much faster response to an interrupt request.

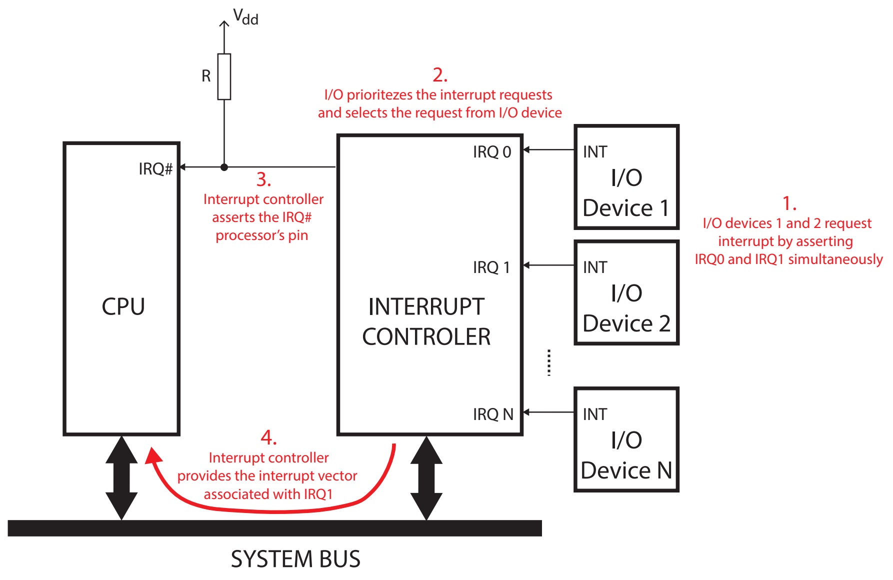


Fig. 3.5: A system with an interrupt controller.

Although the operations in an interrupt controller are performed by hardware, interrupt controllers are programmable. It means that they typically have a common set of addressable (memory-mapped) registers, which enable the system programmer to set the priorities and interrupt vectors for each interrupt source before the interrupt controller is being used. In the following sections, we will cover a few real-world interrupt controllers used with ARM and Intel processors.

3.3.1 ARM Advanced Interrupt Controller

The Advanced Interrupt Controller (AIC) is an 8-level priority vectored interrupt controller, providing handling of up to thirty-two interrupt sources. It is used with ARM9 processors. Figure 3.6 illustrates the block diagram of an ARM9 based system with AIC. The AIC drives the FIQ# (fast interrupt request) and the IRQ# (standard interrupt request) inputs of an ARM9 processor. Inputs of the AIC are external interrupts coming from the peripheral I/O devices.

The Interrupt Source 0 (IS 0) is always connected to the FIQ processor's input. The interrupt sources 1 to 31 (IS 1 to IS 31) can be connected to the interrupt outputs of peripheral devices. An 8-level priority controller drives the IRQ line of the processor. Each interrupt source has a programmable priority level of 7 (the highest priority) to 0 (the lowest priority).

As soon as an interrupt request occurs on an interrupt source, the IRQ# line is asserted. If several interrupt sources have asserted the interrupt request, the priority controller determines the interrupt source with the highest priority, which will be serviced. If several interrupt sources of equal priority are pending, the interrupt with the lowest interrupt source number is serviced first. If an interrupt request happens during the interrupt service in progress, it is delayed until the software indicates to the AIC the end of the current service. Figure 3.6 illustrates the simplified internal structure of AIC. AIC employs an interrupt vectoring scheme. The interrupt handler

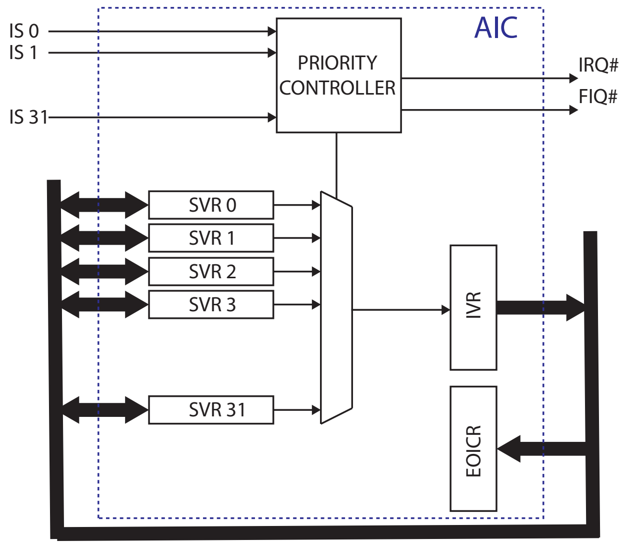


Fig. 3.6: Simplified internal structure of AIC.

addresses (interrupt vectors) corresponding to each interrupt source can be stored in the AIC's registers SVR1 to SVR31 (Source Vector Register 1 to 31). When one or more interrupt requests occur, the content of the SVR corresponding to the interrupt source with the highest priority is automatically transferred to the Interrupt Vector Register (IVR). To obtain the start address of the interrupt handler, the CPU must read the IVR register. In the ARM9 based systems, the IVR register is always mapped at the absolute address 0xFFFFF100. Remember that the interrupt vector for IRQ interrupt is 0x00000018. Hence, the IVR accessible from the ARM interrupt vector at address 0x00000018 through the following instruction:

```
ldr pc,[pc,#-0XF20]
```

When the processor executes this instruction, it loads the value in IVR into its program counter, thus branching the execution on the correct interrupt handler. Besides, reading the IVR also de-asserts the IRQ# line on the processor. But from where the value -0xF20 comes in the above instruction? Recall that the instructions are executed in the EX stage. By the time the above instruction is issued into the EX stage, the PC has already been increased by 8 and is equal to 0x00000020. This is because the CPU has fetched two more instructions. Hence, we have to subtract 0x2F0 from 0x00000020 to obtain 0xFFFF F100.

Before returning, the interrupt handler must indicate to the AIC the end of the current service by dummy writing to the EOICR register (End Of Interrupt Command Register). This will re-enable the further interrupts in AIC. The return from the interrupt handler is, as we have already learned, performed by the `subs pc,lr,#4` instruction. This has the effect of returning from the interrupt to whatever was being executed before, and of restoring the CPSR from the SPSR.

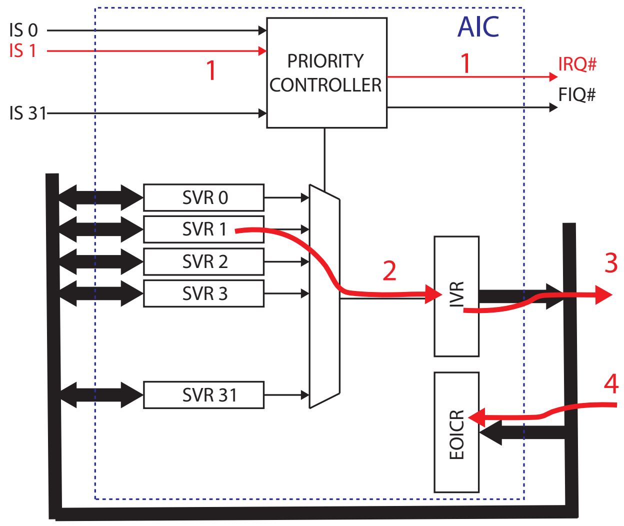


Fig. 3.7: Simplified internal operation of AIC.

An example of the procedure of obtaining the interrupt vector is in Figure 3.7. Let us assume that a peripheral device asserts the interrupt request at the IS 1 line of AIC (step 1). Assuming that no other IS line has been asserted and that the CPU services no interrupt, the priority controller in AIC immediately asserts the CPU's IRQ# signal (step 1). Then, the priority controller selects the SVR1 register, and its content is transferred into the IVR register (step 2). The CPU detects that IRQ# has been asserted, stops the instruction execution, and saves the context of the interrupted program. It then fetches the instruction `ldr pc,[pc,#-0XF20]` from the IRQ interrupt vector (0x00000018). This instruction moves the content of the IVR register into the program counter (step 3) and CPU branches on the IRQ handler. Before returning from the IRQ handler, the CPU dummy writes into the EOICR (step 4).

As we have seen, when AIC is used to route external interrupt requests from peripheral devices to the CPU, the instruction at the interrupt vector 0x00000018 is not a branch instruction (B) to the interrupt handler, but the instruction that loads the IVR into PC (which also acts as a branch). The same holds for the FIQ vector. Hence, we should change the interrupt vector table from Listing 3.1, accordingly. Also, the interrupt handlers for interrupt sources IS1 to IS31 should dummy write to EOICR before returning. Listing 3.8 shows the updated interrupt vector table and pseudocode for an ISx interrupt handler.

```
        .org 0x00000000
Vector_Table:
        b Reset_Handler
        b Undefined_Handler
        b SWI_Handler
        b Prefetch_Handler
        b Abort_Handler
        nop
        lr pc, [pc, #-0xF20]     // load IVR into PC
        lr pc, [pc, #-0xF20]     // load IVR into PC


ISx_Handler:
        <handler instructions>
        ...
        <write to EOICR>
        subs pc,lr,#4
```

Listing 3.8: ARM vector table and ISx handler when AIC is present in the system.

3.3.2 *Intel 8259A Programmable Interrupt Controler*

Intel processors also have only a single interrupt input. As a personal computer has several peripheral devices that can raise interrupts, the Intel Programmable Interrupt Controller (PIC) 8259A is used to manage them. The 8259A PIC is a special interrupt controller designed particularly for Intel processors. It is connected between the interrupt requesting peripheral device and the Intel processor. This means that the interrupt requests from peripheral devices are first transferred to the PIC, which in turn asserts the processor's interrupt input. Figure 3.8 illustrates the system with the 8259A PIC.

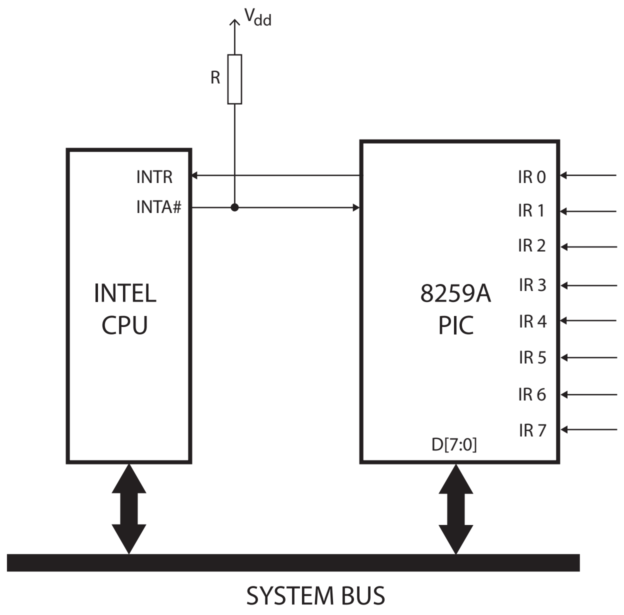


Fig. 3.8: A system with the 8259A PIC.

The 8259A was introduced in the early 1980s and was used in personal computers until the 1990s. It is still used in some Intel-based embedded systems. While not anymore a separate chip, the 8259A interface is still provided by the chipset on modern x86 motherboards. Although someone could say it is obsolete, its functioning will help us to understand the evolution of interrupt controllers in the Intel-based computer systems.

The Intel 8259A Programmable Interrupt Controller handles up to eight vectored priority interrupts for the CPU. It is cascadable for up to 64 vectored priority interrupts without additional circuitry. The interrupt inputs have fixed priority based on its number, and the interrupts on its inputs may be either edge-triggered or level-triggered.

The 8592A PIC has the following set of registers: Interrupt Request Register (IRR), In-Service Register (ISR), and Interrupt Mask Register (IMR). The IRR register specifies which interrupts are pending. The ISR register specifies which interrupts have been acknowledged, and the IMR specifies which interrupts are to be ignored and not acknowledged. Figure 3.9 illustrates the simplified internal structure of the 8259A PIC.

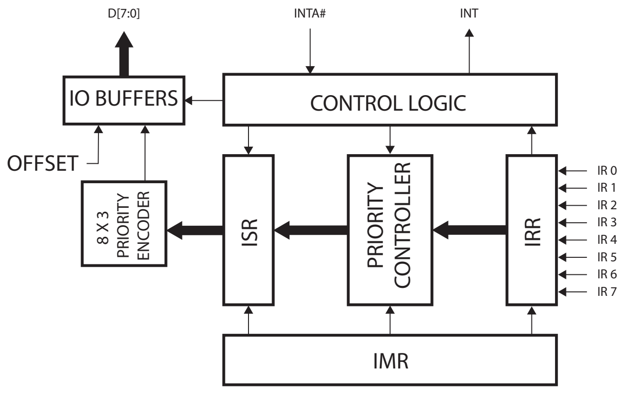


Fig. 3.9: Simplified internal structure of the 8259A PIC.

The peripheral device that wishes to issue an interrupt request asserts one of the pins IR0 to IR7. If the interrupt is not masked in the IMR register, the 8259A PIC will set the corresponding bit in the interrupt request register (IRR). The IRR register remembers all the pending interrupt requests. As more peripheral devices can issue the interrupt request simultaneously, several bits may be set in the IRR register at the same time. At the same time, the 8259A sends an INT to the CPU. When the 8259A PIC asserts the interrupt request on the processor's INTR input, the processor recognizes this request on the next instruction fetch. It then stops the instruction fetch and automatically saves the program context onto the stack. The CPU then starts the so-called **interrupt-acknowledge cycle**.

Interrupt-acknowledge cycles are special bus cycles that enable the PIC to output an interrupt vector onto the data bus. This vector is fetched by the CPU and transferred into the program counter during the interrupt-acknowledge cycle. The value read during the interrupt-acknowledge cycle is then multiplied by 4 and used to load an interrupt vector from this address in memory. The Intel processors perform two back-to-back interrupt-acknowledge cycles in response to an active INTR input:

1. Firstly, the CPU responds by asserting the first INTA pulse. Upon receiving an INTA from the CPU, the priority controller in the 8259 passes the highest priority bit from IRR to the In-Service Register (ISR), and the corresponding IRR bit is reset. The set bit in the ISR indicates which interrupt request is being serviced.
2. Secondly, the processor asserts the second INTA pulse to instruct the 8259A to release an 8-bit interrupt number onto the Data Bus (D0-D7). This ends the interrupt-acknowledge sequence.

The CPU now reads the 8-bit interrupt number (n) and multiplies it by 4. This value represents the address of the memory location that holds the start address of the interrupt handler. Hence, the CPU executes in hardware the following operation:

```
PC <-  Mem[n x 4]
```

The structure of the 8-bit vector number returned from the 8259A PIC is shown in Figure 3.10. The lower three bits are the binary-coded number of the bit that was set in the ISR. The higher five bits are the offset that can be programmed during the 8259A PIC initialization. Recall, that the Intel stores its IVT table at the address 0x0000 and that the interrupt numbers 32 through 255 are reserved for external interrupts signaled on the INTR pin. If we want to map the IRQ interrupts from 8259A PIC at the address 0x0080 (=32x4) in the IVT, the offset returned in the 8-bit vector number should be 00100. In the case of the IR0 interrupt, the returned vector number is 0x20, which maps to 0x0080; in the case of the IR1 interrupt, the returned vector number is 0x21, which maps to 0x0084, etc.

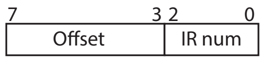


Fig. 3.10: The 8-bit vector number returned by the 8259A PIC. The lower three bits are the binary-coded number of the bit that was set in the ISR. The higher five bits are the offset that can be programmed during the 8259A PIC initialization.

To reset the bit in the ISR register, the interrupt handler should issue an End-Of-Interrupt (EOI) command to the 8259A PIC. The set bit is therefore deleted manually. The 8259A is now ready to process the next pending hardware interrupt request in IRR. The priority controller passes the highest priority bit form IRR to ISR, and the above sequence is repeated. Figure 3.11 illustrates the operation of the 8259A PIC when IR2 and IR4 are issued simultaneously, and IR2 has a higher priority than IR4.

Summary: AIC vs. 8259A

Besides being designed for different processors, the main difference between the ARM AIC and Intel 8259A PIC is how the interrupt vector is obtained. In ARM AIC, the CPU reads the interrupt vector from an AIC's memory-mapped register using a LOAD instruction, while in 8259A PIC, the CPU reads the vector from the data bus, without executing any instruction.

The former is considered faster (recall that instructions in ARM9 are executed in 5 clock cycles) but requires additional signaling between interrupt controller and CPU (INTA) and a special interrupt-acknowledge cycle.

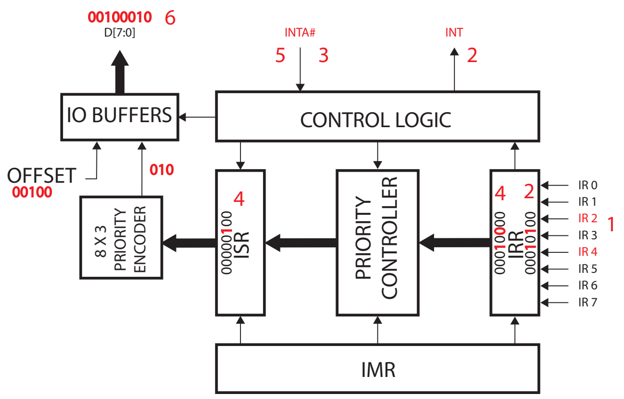


Fig. 3.11: the operation of the 8259A PIC when IR2 and IR4 are issued simultaneously, and IR2 has a higher priority than IR4. (1) Two peripheral devices assert the pins IR2 and IR4. (2) Assuming the interrupts are not masked in the IMR register, the 8259A PIC will set the corresponding bits in the interrupt request register (IRR). At the same time, the 8259A sends an INT to the CPU. (3) The CPU responds by asserting the first INTA pulse. (4) Upon receiving an INTA from the CPU, the priority controller in the 8259 passes the highest priority bit from IRR to ISR, and the corresponding IRR bit is reset. (5) The processor asserts the second INTA pulse. (6) The 8259A controller releases an 8-bit interrupt number onto the data bus (D0-D7).

3.3.3 8259A PIC Cascading

With one 8259A PIC, eight interrupt sources could be managed. But soon, eight interrupt lines weren't enough. The 8259A PIC has the capability for **two-level cascading**. The first level is made up of one **master** PIC, and the second level is formed from up to eight **slave** PICs. Such a configuration can manage up to 64 peripheral interrupt requests. But in practice, only two PCs are used: one is the master, and the other is the slave. The PCs included two 8259A PICs chained together, and this setup became de facto standard for the x86 platform. This scheme is referred to as the dual-PIC system. Figure 3.12 illustrates the dual-PIC system.

In the dual-PIC system, the slave's INT output is connected to the master's IR2 input. PC documentation established the following naming convention: IRQs 0 through IRQ7 are processed with the first Intel 8259 PIC (master), and IRQs from 8 to 15 are processed with the second Intel 8259 PIC (slave). Therefore, the slave's INT output is connected to the master's IRQ2 input. Only the master's INT output is connected to the CPU's INTR input and can signal about the incoming interrupts. INTA and D0 through D7 signals of both PICs are connected to the CPU and data bus as in the single-PIC configuration. Note that IRQ 2 is not available for device interrupts, and there are only fifteen interrupt inputs available for peripheral device interrupts.

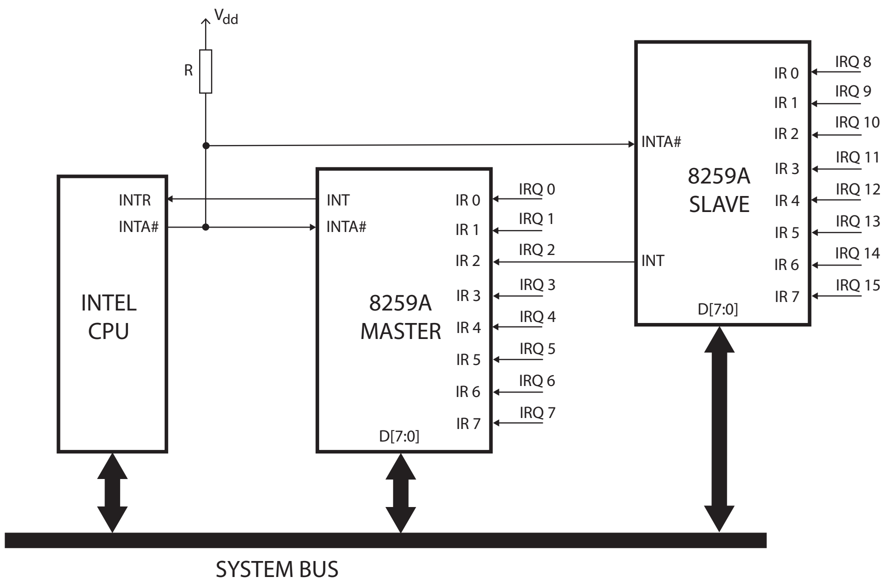


Fig. 3.12: A dual-PIC system.

The way in which an interrupt request is processed depends on whether the request is asserted on slave's or master's IRQ inputs. If the request is asserted on IRQs form 0 to 7 (master), it is processed in the same way as in the single-PIC configuration. Otherwise, the following steps are required: when an interrupt request is placed on lines IRQ 8 through 15, the corresponding bit is set in the slave's IRR register. The slave asserts its INT output and signals the IRQ interrupt to the master PIC, when master PIC receives the interrupt request on IRQ2, it sets the bit 2 in its IRR and asserts its INT output to signal the INTR request to the processor, the CPE starts the interrupt-acknowledge sequence and sends the first INTA pulse. Upon receipt of the first INTA pulse, the highest priority bits in the master's and slave's IRRs are cleared, and the corresponding bits in both ISRs are set, the CPE outputs the second INTA pulse and causes the slave PIC to output its 8-bit vector number. The interrupt handler must send two EOI commands to clear both ISR bits.

But wait! How can we set two different vector numbers for interrupts signaled at the master's IR3 (IRQ3) input and at the slave's IR3 input (IRQ11)? Recall, that the 8-bit vector number returned from the 8259A PIC contains a programmable offset in its higher five bits. Hence, the master and slave PICs should be initialized with different offsets. For example, we can set the master's offset to 00100 and the slave's offset to 00110. In this case, the interrupts from the master PIC will be mapped to the IVT addresses 0x0080-x009F, and the slave's interrupts will be mapped to the IVT addresses 0x00C0-0x00DF.

The original PCs used the ISA bus for its I/O devices. The interrupts on the ISA bus are edge-triggered. An I/O device asserts an interrupt by raising the signal from low to high. Edge-triggered interrupts inhibit the sharing of ISA interrupts by multiple devices, so each ISA device requires a dedicated interrupt input on the 8259As. A typical interrupt configuration at that time is presented in Table 3.3.

Table 3.3: ISA Interrupt Assignments.

IRQ	**Assignment**
0	System timer
1	Keyboard controller
2	interrupt from slave controller
3	Serial ports COM 2 / COM 4
4	Serial ports COM 1 / COM 3
5	Sound card
6	Floppy disk controller
7	Parallel port 1 (Printer)
8	Real-time clock
9	ACPI
12	PS/2 mouse controller
13	Math (floating point) co-processor
14	ATA channel 1 (Primary IDE)
15	ATA channel 1 (Secondary IDE)

The Intel 8259 PIC has several limitations to interrupt servicing in modern computer systems:

1. a limited number of interrupt lines necessitates the sharing of interrupts. Shared interrupts require the OS to poll multiple IO devices to determine who actually generated the interrupt,
2. interrupt priority is fixed based on IR number,
3. PIC does not support multiple CPUs.

At the time when the main bus for external devices was the ISA bus, this 8929A based architecture was sufficient. It was only necessary that different peripheral devices did not connect to the same IRQ inputs since ISA uses the edge-triggered interrupts, which are not shareable. But the PCI bus later replaced the ISA bus, and interrupts in the PCI bus can be shared. Also, the PCI bus has been replaced by the serial message-based PCI Express (PCIe) bus, and more CPU cores are added to the computer system. The following sections will cover the evolution of the interrupt controller used in the Intel-based computer systems, and we will learn how to handle the interrupts on the PCI bus where the number of peripheral devices exceeds the number 15, how to share the interrupt lines on the PCI bus, and finally how to handle interrupts in the modern multi-core PCIe-based systems.

3.3.4 Intel Advanced Programmable Interrupt Controler

By nature, the 8259A PIC can only send interrupts to one CPU, and in a multiprocessor system, it is desired to load CPUs in a balanced way. The solution to this problem was the new APIC (Advanced PIC) architecture. This architecture addressed many of the limitations of the older PIC-based architecture. The most apparent is the support for multiple CPUs.

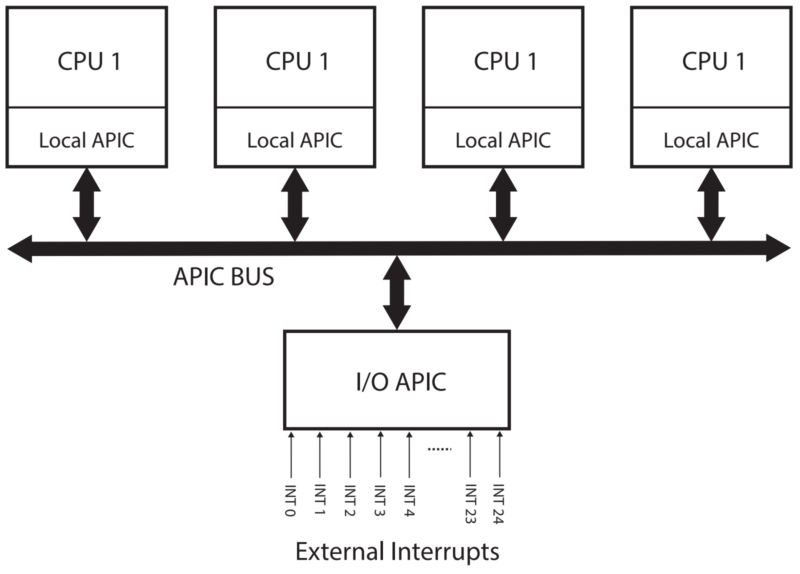


Fig. 3.13: An APIC based computer system.

At the system level, APIC consists of two parts (Figure 3.13). One part resides in the I/O subsystem and is called the I/O APIC. It is responsible for routing interrupts from external devices to the other part, the Local APIC (LAPIC), which resides in each CPU. The local APIC and the I/O APIC communicate over a dedicated 3-wire serial APIC bus. The IOAPIC bus interface consists of two bi-directional data signals (APICD[1:0]) and a clock input (APICCLK). The modern systems may use a standard system bus instead of a separate APIC bus for this task. It is worth noting that it is possible to have several I/O APIC controllers in the system. For example, one for 24 interrupts in a southbridge, and the other one for 32 interrupts in a northbridge. The CPU's Local APIC contains the necessary intelligence to determine whether or not its processor should accept interrupts broadcast on the APIC bus. The Local APIC also provides local pending of interrupts and handles all interactions with its local processor (e.g., the interrupt acknowledge sequence). Additionally, each I/O APIC has 24 interrupt lines and allows the priority of each interrupt to be set independently. The I/O APIC sends an interrupt vector to the local APIC, and, as a result, the OS does not have to interact with the I/O APIC until it sends the end of interrupt notification.

Summary: APIC

In the APIC-based systems, each CPU includes a local APIC which receives interrupt messages and uses them to assert interrupts on the CPU. The chipset includes one or more I/O APICs which are responsible for converting device interrupt signals into messages that are delivered to one or more local APICs.

3.3.4.1 Local APIC

The Local Advanced Programmable Interrupt Controller (LAPIC) was introduced into the Pentium processor and is included in more recent Intel processor families. The local APIC performs two primary functions for the processor: it receives interrupts from the processor's interrupt pins and an external I/O APIC. It sends these to the processor core for handling. In multiple-processor systems, it sends and receives interprocessor interrupt (IPI) messages to and from other processors on the system bus. IPI messages are used to distribute interrupts among the processors in the system. When a local APIC sends an interrupt to its processor core (by asserting the processor's INTR line) for handling, the processor uses the interrupt and exception handling mechanism described in Section 3.2.4.

The LAPIC receives interrupts from several sources:

- Locally connected I/O devices: these interrupts are asserted by an I/O device connected directly to the processor's local interrupt pins (LINT0 and LINT1).
- Inter-processor interrupts (IPIs): an Intel processor can use the IPI messages to interrupt another processor or group of processors on the system bus.
- Externally connected I/O devices: these interrupts are asserted by an I/O device connected to the interrupt input pins of an I/O APIC. Interrupts are sent as IPI messages from the I/O APIC to one or more LAPICs in the system.

Figure 3.14 illustrates a simplified internal structure of a Local APIC. The heart of the LAPIC is very similar to the 8259A: it contains the IRR and ISR registers and a priority controller. Besides, it contains two registers that form the local vector table LVT and the interrupt command register (ICR). In fact, the LAPI is a rather complicated device containing a large set of addressable registers, timers, and other control logic. Figure 3.14 shows only its vital parts that are necessary to understand its interrupt handling. The Protocol Transition Logic block receives the IRI messages from the APIC bus. If the LAPIC is the destination, the Protocol Transition Logic block forwards the destination mode and the vector number from the IRI message to the Acceptance Logic block, which decodes the 8-bit vector number and forwards the bit from the decoded 256-bit word into the IRR register. The rest of the internal logic is described for each interrupt source in the text below.

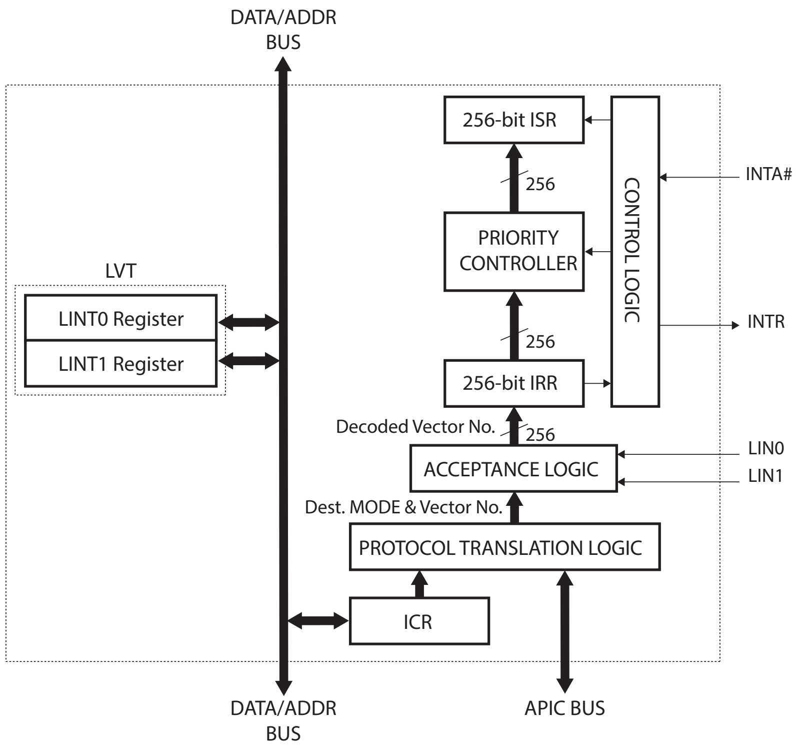


Fig. 3.14: Simplified internal structure of a Local APIC.

Interrupts from locally connected I/O devices. Upon receiving a signal from the processor's LINT0 and LINT1 pins, the local APIC delivers the interrupt to the processor core using a group of APIC registers called the **local vector table**. A separate entry (i.e., a separate register) is provided in the local vector table for each local interrupt pin (LINT0 and LINT1). For example, if the LINT1 pin is going to be used as an NMI pin, the LINT1 entry in the local vector table can be set up to deliver an interrupt with vector number 2 (NMI interrupt in Table 3.2) to the processor core. The LVT consists of two 32-bit registers: LINT0 Register (specifies the interrupt number when an interrupt is signaled at the LINT0 pin) and LINT1 Register (specifies interrupt number when an interrupt is signaled at the LINT0 pin). An interrupt number is an 8-bit number stored in the bits 0 through 7 in each LIN register.

Inter-processor interrupts (IPIs). A processor generates IPIs by writing to a special LAPIC register called the interrupt command register (ICR) in its local APIC. The act of writing to the ICR causes an IPI message to be generated and issued on the system bus or the APIC bus. An IPI message includes the processor destination number, the vector number, and trigger mode (edge or level). When the target processor receives an IPI message, its local APIC handles the interrupt request automatically using information included in the message such as vector number and trigger mode. The IPI mechanism is used in multi-processor systems to send or forward interrupts for a specific vector number. For example, a local APIC can use an IPI to forward an interrupt to another processor for servicing. Also, the IPI mechanism is used by I/O APIC to send an interrupt for a specific vector number that originates from an I/O device connected to I/O APIC. The interrupt command register (ICR) is

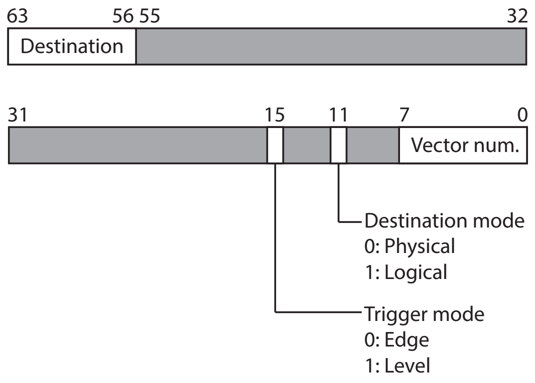


Fig. 3.15: The LAPIC ICR register.

a 64-bit local APIC register that allows software running on the processor to specify and send interprocessor interrupts (IPIs) to other processors in the system. The act of writing to the low 32 bits of the ICR causes the IPI message to be sent. Figure 3.15 illustrates the ICR register (only the bits that are important for understanding are specified/show). The 8-bit Destination field specifies the target processor. The Destination Mode bit further specifies if the destination is either physical (0) or logical (1) processor. The Trigger Mode bit selects the trigger mode: edge (0) or level (1).

Externally connected I/O devices. The local APIC can also receive interrupts from externally connected devices through the I/O APIC (see Figure 3.13). The I/O APIC is responsible for receiving interrupts generated by system hardware and I/O devices and forwarding them to the local APIC as IPI messages. Each individual pin on the I/O APIC can be programmed to generate a specific interrupt vector when asserted. This vector is then sent to LAPIC as a part of an IPI message.

The local APIC handles the interrupts as follows:

1. if it receives a message on the APIC bus, it determines if it is the specified destination. If it is the specified destination, it accepts the message; otherwise, it discards the message,

2. if the local APIC determines that it is the designated destination for the interrupt, the local APIC sets the appropriate bit in the IRR,
3. when interrupts are pending in the IRR register, the local APIC sends them to the processor one at a time, based on their priority, similarly as in the 8259A. The processor responds with the interrupt acknowledge sequence. During the first INTA pulse, the LAPIC moves the highest priority bit form the IRR to ISR. During the second INTA pulse, the LAPIC puts the interrupt vector on the data bus. If the interrupt request comes from a locally connected I/O device (at the LINT0 or LINT1 pins), the interrupt number is stored in the corresponding LVT entry in the LAPIC. If the interrupt request comes from a message, the interrupt number is contained in the the message.

Completing the handler routine is indicated by instruction in the instruction handler code that writes to the end-of-interrupt (EOI) register in the local APIC. The act of writing to the EOI register causes the local APIC to delete the interrupt from its ISR.

3.3.4.2 I/O APIC

The I/O Advanced Programmable Interrupt Controller (IOAPIC) provides multiprocessor interrupt management and incorporates interrupt distribution across all processors. In systems with multiple I/O subsystems, each subsystem can have its own set of interrupts. Each interrupt pin is individually programmable as either edge or level triggered. The interrupt vector can be specified per interrupt input.

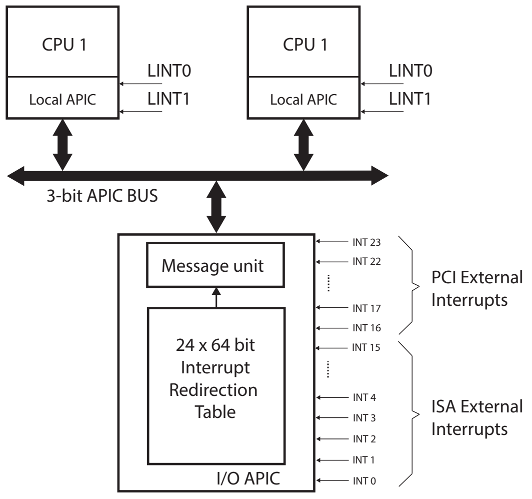


Fig. 3.16: The LAPIC ICR register.

The I/O APIC (Figure 3.16) consists of 24 interrupt input lines, a 24-entry Interrupt Redirection Table (IRT) with 64-bit entries, programmable registers, and a message unit for sending and receiving messages over the APIC bus. I/O devices signals interrupt requests by asserting one of the interrupt lines to the I/O APIC. The I/O APIC selects the corresponding entry in the IRT and uses the information in that entry to format an interrupt request message. Each entry in the IRT contains:

- a bit that indicates edge/level sensitive interrupt,
- the interrupt vector and priority, and
- the destination processor.

The information in the IRT entry is used to form and transmit an IRI message to other LAPICs via the APIC bus.

When an external interrupt request is signaled on the I/O APIC interrupt input, the I/O APIC controller will send an interrupt message to the LAPIC of one of the system CPUs. In this way, the I/O APIC controller helps balance interrupt load between processors.

APIC messages come in several formats and different lengths. Here we present only two types of APIC messages: EOI Message and the so-called Short Message. Local APICs use EOI messages to send an end-of-interrupt (EOI) occurring for a level-triggered interrupt to an I/O APIC. This message is needed, so the I/O APIC knows when an interrupt has been serviced. In this way, the I/O APIC can differentiate between a new interrupt on the interrupt line versus the same interrupt on the interrupt line. I/O APICs use Short Messages for the delivery of external interrupts to local APICS.

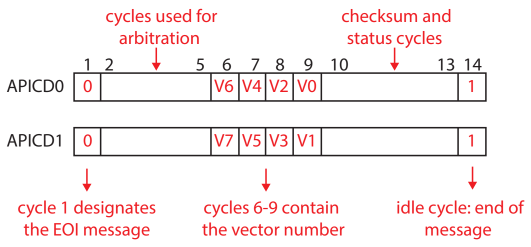

cycles used for arbitration
checksum and status cycles
1 2 5 6 7 8 9 10 13 14
APICD0
0
V6
V4
V2
V0
1
APICD1
0
V7
V5
V3
V1
1
cycle 1 designates the EOI message
cycles 6-9 contain the vector number
idle cycle: end of message

Fig. 3.17: The format of the EOI message.

The format of the EOI message is presented in Figure 3.17. All EOI messages are 14 bits long and take 14 cycles on the APIC bus to transmit. Local APICs send 14-cycle EOI messages to the I/O APIC to indicate that a level triggered interrupt has been accepted by the processor. This message is a result of software writing into the EOI register of the local APIC. The first cycle is used to designate an EOI message. The vector number is sent in cycles 9 through 12. The local APIC gives the target of the EOI message by transmitting the interrupt vector number (V7 through V0). When this message is received, the I/O APIC resets the IRR bit for that interrupt. If the interrupt signal is still active after the IRR bit is reset, the I/O APIC treats it as a new interrupt. The last cycle in which both data lines are set high is used to signal end-of-message.

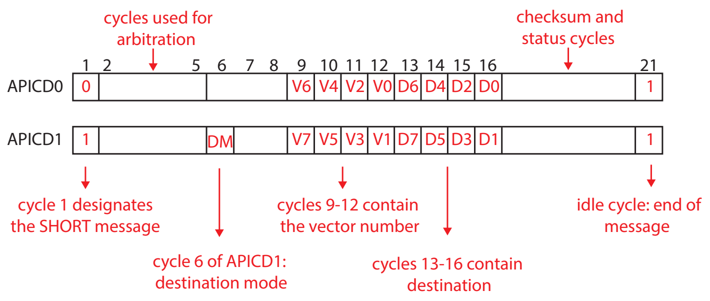

cycles used for arbitration
checksum and status cycles
1 2 5 6 7 8 9 10 11 12 13 14 15 16 21
APICD0
0
V6 V4 V2 V0 D6 D4 D2 D0
1
APICD1
1
DM
V7 V5 V3 V1 D7 D5 D3 D1
1
cycle 1 designates the SHORT message
cycles 9-12 contain the vector number
idle cycle: end of message
cycle 6 of APICD1: destination mode
cycles 13-16 contain destination

Fig. 3.18: The format of the SHORT message.

The format of a SHORT message is presented in Figure 3.18. All SHORT messages are 21 bits long and take 21 cycles on the APIC bus to transmit. Short messages are used by I/O APICS for sending external interrupts to local APICs. The first cycle is used to designate a SHORT message. The vector number is sent in cycles 9 through 12. If Destination Mode (DM) is 0, then cycles 15 and 16 are the local APIC ID, and cycles 13 and 14 are sent as 1. In this case, the message is sent to a physical processor. If DM is 1, then cycles 13 through 16 are the 8-bit Destination field that selects the logical processor. The last cycle in which both data lines are set high is used to signal end-of-message.

The I/O APIC with 24 input interrupt lines was used in the systems with the PCI bus. The APIC architecture could support up to 16 CPUs. The I/O APIC provided backward compatibility with the older 8259A PIC-based systems. Interrupts 0-15 were used for old ISA interrupts for compatibility with older systems, and interrupts 16-23 were meant for all the PCI devices. With this delimitation, all conflicts between ISA edge-triggered and PCI level-triggered interrupts could be easily avoided. This assignment of interrupts 0-15 provided only eight additional interrupts, which forced the sharing of PCI interrupts - two ore more devices on the PCI bus were forced to share the same I/O APIC's input interrupt line. We will cover the PCI interrupts sharing and routine in the following sections.

One of the biggest differences between the 8259A PICs and I/O APICs is that the pins on I/O APICs are completely independent. With the 8259A PICs, the eight input pins are mapped to eight consecutive vectors in IDT (or IVT), and all of the interrupts are sent to the same CPU. In I/O APICs, on the other hand, each pin is programmed independently. Each pin is assigned its own vector by the operating system and can be mapped to one or more CPUs.

3.4 PCI interrupts

The PCI bus was added to the PCs in the mid-1990s. In the first years, both buses, PCI and ISA, coexisted in the systems. In PCI, the term **device** refers to a piece of hardware that is plugged-in into the PCI slot and contains from one to eight **functions**. A multi-function PCI device is a physical PCI expansion board that embodies between two and eight PCI functions. For example, a single PCI device may include several USB controllers as functions. Another example would be a PCI card with a high-speed communications port and a parallel port. Hence, a PCI expansion card inserted in the PCI expansion slot is a PCI device, and a single expansion card may contain up to eight functions. However, from the operating system's perspective, each function on a PCI device is a logical operating device.

PCI allows devices to assert interrupts in two different ways:

1. The first way uses dedicated interrupt signals (lines) and is known as **Legacy INTx interrupts**.
2. The second way uses special memory writes that are sent over the data bus, just like APIC messages, and is known as **Message Signaled Interrupts (MSI)**. First, we will cover Legacy INTx interrupts. Later, we are going to cover the MSI interrupts.

3.4.1 PCI Legacy interrupts

A PCI card in a slot may have up to eight functions on it, but there are only 4 PCI interrupt pins: INTA#, INTB#, INTC#, and INTD#. PCI legacy interrupts are level-triggered; hence, they may be shared by multiple functions. Each function within the device is only permitted to use one of these interrupt pins to generate requests. If a device contains only one function that uses only one interrupt pin, it must be called INTA#. If a device contains more than one function, all functions within a device may be bonded to the same pin, INTA#, or each may be bonded to a dedicated pin (this would be true for a device with up to four functions embodied within it). Also, a group of functions within the package may share the same interrupt pin. In the most simple (and most common) case, a PCI device has only one function with its interrupt going to the lane INTA#.

Figure 3.19 illustrates a simple interrupt model with two peripheral devices on the PCI bus; hence, both are required to INTA# . Each peripheral device embodies only one function that generates PCI interrupts. The first device is an ethernet card in the PCI slot 0 that generates interrupts on INT#A line, while the second device is a sound card in the PCI slot 3 that also generates interrupts on the INT#A line. Both devices share the same interrupt request signal trace on the system board, which is routed to the IRQ5 input on the Intel 8259A programmable interrupt controller (PIC). Indeed, the PCI standard does not limit the interrupt controller used to route

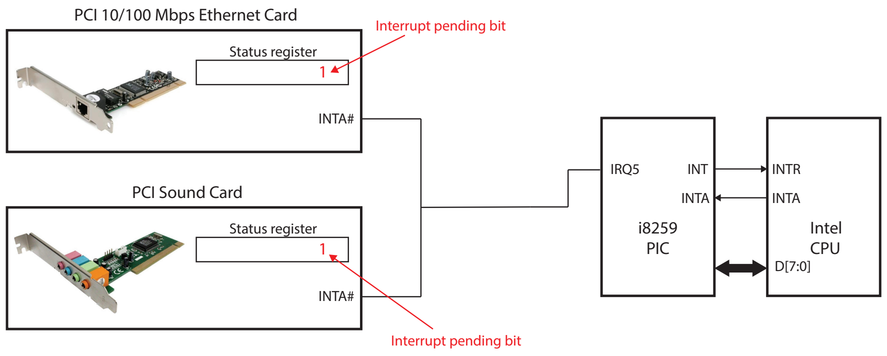


Fig. 3.19: Example system for shared interrupts on the PCI bus.

PCI interrupts to the CPU as long it supports level-triggered interrupts; hence, in this example, we assume that the 8259A PIC is used for this purpose.

Let us assume that both devices assert the interrupt request and that the sound card asserted the interrupt first. The interrupts are asserted by driving the interrupt line LOW. In addition to asserting the interrupt request, both devices set an interrupt pending bit in their memory-mapped status registers so that interrupt handlers can access both status registers. Let us also assume that no other device in the system had asserted an interrupt request before the sound card and ethernet card.

When the Intel 8259A PIC detects the interrupt request on its IRQ5 input, it asserts the interrupt request on the processor's INTR input. The processor will recognize this request on the next instruction fetch. It then stops the instruction fetch and automatically saves the program context onto the stack. The CPU then starts the interrupt acknowledge sequence: the CPU responds by asserting the first INTA pulse, the 8259 PIC prioritizes the pending interrupt requests by setting the bit 5 in its ISR register and clearing the bit 5 in its IRR register, the CPU outputs the second INTA pulse to instruct the 8259A PIC to release the 8-bit pointer onto the data bus (D0 to D7) where it is read by the CPU. Now the processor has received the interrupt vector number associated with IRQ5. Let assume that the interrupt vector number is `0x07`. The processor multiplies this value by 4, which yields the address of the interrupt vector entry, `0x0000001C`. The processor now reads the content of the memory location `0x0000001C` to obtain the start address of the interrupt handler.

As both devices, the ethernet card and the sound card, share the same interrupt line IRQ5, the interrupt handler should contain the code to handle the interrupt requests from both devices. Let us assume that the interrupt handler contains both codes, the "ethernet" handler, and the "sound card" handler. The simplified structure of the IRQ5 interrupt handler is presented in Listing 3.9. The interrupt handler first checks which device has asserted the interrupt request by checking the interrupt pending bit in the corresponding status register.

```c
/*
* IRQ5 Handler
*/
__attribute__((interrupt)) void irq5handler () {

    /* Check the interrupt pending bit in the Ethernet status reg */
    if (eth_status_reg & (1<<INT_PEND_BIT)) {
    /* Ethernet Handler Code */
    eth_status_reg &= ~(1<<INT_PEND_BIT); // clear int pending bit
    ...
    ...
    ...
    }

    /* Check the interrupt pending bit in the Sound card status reg */
    if (snd_status_reg & (1<<INT_PEND_BIT)) {
    /* Sound Card Handler Code */
    snd_status_reg &= ~(1<<INT_PEND_BIT); // clear int pending bit
    ...
    ...
    ...
    }

    /* return from interrupt */
}
```

Listing 3.9: IRQ5 interrupt handler

Listing 3.9 shows that the interrupt handler checks the pending bit in the ethernet card's status register first. As this bit is asserted, the ethernet handler is executed first. The ethernet handler clears the pending bit in the status register and processes the interrupt request. Then the IRQ5 handler proceeds with the sound card handler. This handler checks the sound card's interrupt pending bit to determine if it requires servicing. Since the pending bit is set, the main body of the sound card handler is executed. It clears the interrupt pending bit and services the interrupt request. As both devices have their pending bit clear, the interrupt line is de-asserted.

Hence, the system in Figure 3.19 relies on the vectored interrupt handling to determine which interrupt request input in the PIC 8259A has been asserted, but use the interrupt polling to determine which PCI device has asserted the interrupt request. The sequence of polling determines the interrupt priority. This is why the ethernet card is serviced first, although the sound card had asserted the interrupt request a few moments before the ethernet card.

3.4.2 PCI interrupts routing

Each PCI device (function) that needs an interrupt comes with a fixed PCI interrupt that can't be changed. But this PCI legacy interrupts signal (lane) can be mapped (routed or redirected) to any APIC interrupt input. Thus, at one end, we have a PCI legacy interrupt lines (INTA# through INTD#) being signaled by a PCI function that needs attention. At the other end, we have a CPU receiving an IDT vector. In the middle is an interrupt controller (most commonly, this would be an APIC pair: I/O APIC and LAPIC). Whenever any of the PCI legacy interrupt pins is asserted, the

I/O APIC module supplies the vector associated with that input to the processor's embedded local APIC module. We have already learned that IR16-IR23 input I/O APIC pins are devoted to PCI. The upper four (IR20-IR23) are devoted to PCI functions embedded in the chipset, while the lower four are devoted to PCI legacy interrupt pins. The I/O APIC inputs IR20 through IR23 are called PIRQA through PIRQH when used for PCI interrupts. The acronym PIRQ stands for PCI interrupt request. We have also learned that several PCI devices or functions can use the same PCI legacy interrupt signal to assert interrupts. So, the question is, how the PCI legacy interrupt signals INTA# through INTD# are routed to the I/O APIC inputs PIRQA through PIRQD? The best scenario is pictured in Figure 3.20 where each of the individual PCI interrupt lines is routed to an interrupt controller as a separate input. But such a solution is possible only if there are only up to four PCI devices because the I/O APIC has only four available interrupt inputs.

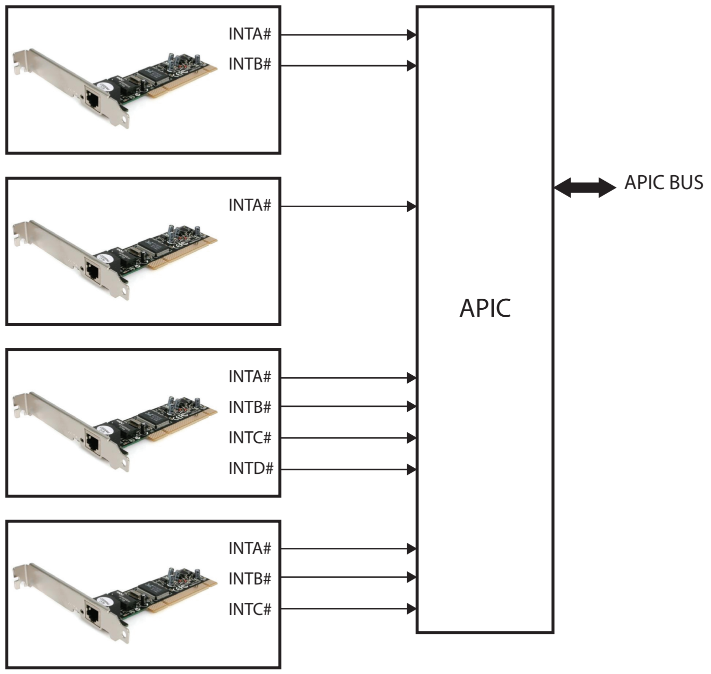


Fig. 3.20: Ideal routing of PCI interrupt lanes.

PCI interrupt signals can be routed to I/O APIC interrupt pins (PIRQs) in several different ways. One simple method of connecting (hardwiring) these lines from PCI devices to the PIRQs would be to connect all INTA# interrupts to PIRQA, all INTB# interrupts to PIRQB, all INTC# interrupts to PIRQC, and all INTD# interrupts to PIRQD. Figure 3.21 illustrates this method of PCI interrupts routing. As we've already said above, the most common case is when a PCI device has only

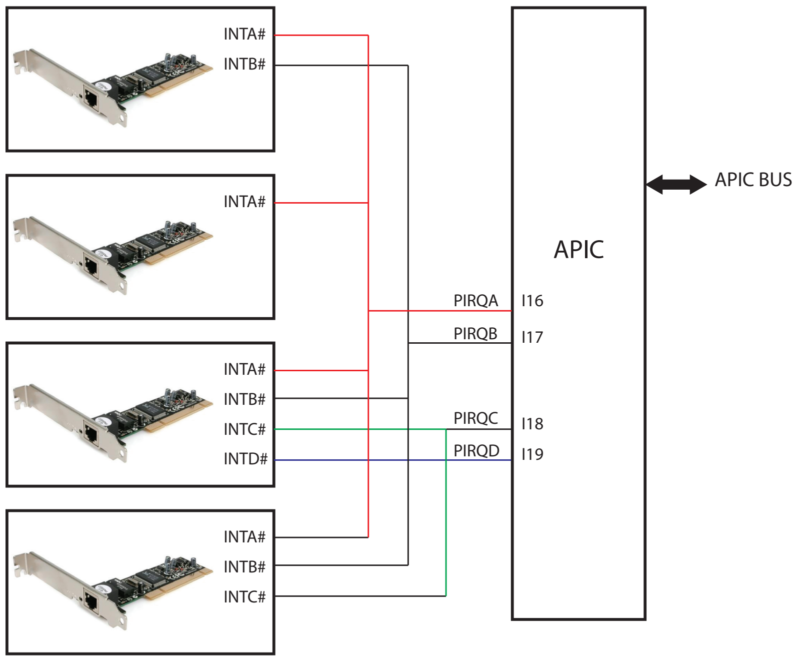


Fig. 3.21: Unbalanced routing of PCI interrupt lanes.

one function, and its interrupt must be connected to the INTA# pin. Therefore if we decided to route all PCI interrupt lanes as we've written, almost all of the devices in a system would share interrupt input PIRQA. As you can see in Figure 3.21 the PIRQA request line is heavily-weighted (four PCI devices). Suppose this lane is connected to the IRQ16 of the APIC. This way, every time a processor has a signal that there is an interrupt on the IRQ16 input, it has to poll all of the device drivers of the PCI devices connected to that IRQ16 line (PIRQA) if they have asserted an interrupt. If there are many of those devices, it will surely decrease system response to the interrupt. And in this case, lanes PIRQB-PIRQD would stand idle most of the time.

The optimal way of PCI interrupts routing should take into account that each PIRQ should have fairly the same number of PCI functions connected to it. We should also consider that some functions trigger interrupts very rarely and some almost constantly (e.g., Ethernet controller). Hence, we may connect the PIRQs in a more random way so that each of them will share about the same number of actual PCI legacy interrupts.

One method of doing this is illustrated in Figure 3.22. This illustration shows how legacy interrupt traces are physically routed across the PCI slots. Although the physical interrupt lines wire each PIRQ to every slot, each PIRQ connects differently to the pins in each slot. Here, wire PIRQA share interrupts INTA# in the PCI slot

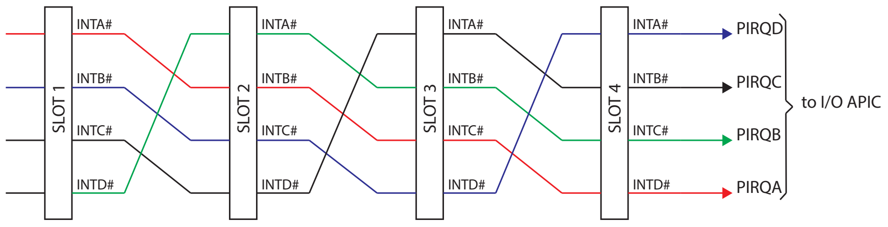


Fig. 3.22: Round-robin routing of PCI legacy interrupt traces (lanes).

1, INTB# in the PCI slot 2, INTC# in the PCI slot 3, and INTD# in the PCI slot 4. Likewise, wire PIRQB share interrupts INTB# in the PCI slot 1, INTC# in the PCI slot 2, INTD# in the PCI slot 3, and INTA# in the PCI slot 4, etc.

Figure 3.23 shows a 4-slot PCI system with the following cards installed:

1. Card 1 installed in Slot 1. This card includes two PCI functions of which one generates interrupts on IRQA#, and the other generates interrupts on IRQD#. The IRQA# pin of this card connects to PIRQA.
2. Card 2 installed in Slot 2. This card includes four PCI functions, thus generating interrupts on IRQA# through IRQD#. The IRQA# pin of this card connects to PIRQB.
3. Card 3 installed in Slot 3. This card includes three PCI functions, which generate interrupts on IRQA# through IRQC#. The IRQA# pin of this card connects to PIRQC.
4. Card 4 installed in Slot 4This card includes four PCI functions, thus generating interrupts on IRQA# through IRQD#. The IRQA# pin of this card connects to PIRQD.

A practical use for this is that one may change the interrupt routing of a PCI card by inserting it in a different slot. In the above example, INTA# of a PCI card will be connected to wire PIRQA if the card is inserted into slot 1, but INTA# will be connected to wire PIRQB when inserted into slot 4.

3.4.3 Message Signaled Interrupts

As we described in the previous section, a PCI device can use up to four dedicated interrupt pins to signal an interrupt request to the I/O APIC. This method is referred to as Legacy INTx interrupts. Each PCI device can have up to four PCI legacy interrupts. After the I/O APIC has received an interrupt request, it forwards it to LAPICs by the mean of the APIC messages. But why not implement this "interrupt message" functionality into a PCI device itself? This way, we could eliminate the need for interrupt traces and interrupt sharing. This method is referred to as **Message Signaled Interrupts (MSI)** and is described in this section. Message Signaled In-

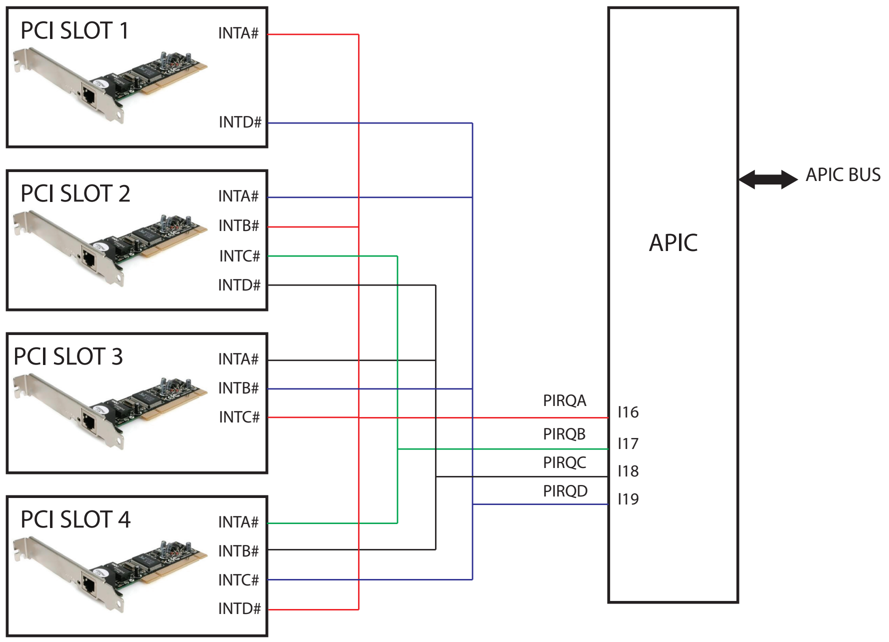


Fig. 3.23: Common round-robin routing of PCI interrupt lanes.

terrupts are special memory writes that are sent over the system bus. In essence, the MSI interrupts do not differ from ordinary PCI memory write transactions. But they are recognized by LAPICs from the address they write to. The data sent in these transactions contain an interrupt vector.

Using a separate signal for PCI INTx interrupts raises several issues. First, on many x86 systems, this requires separate physical traces on the motherboard to connect the signals to interrupt controller input pins. Second, the interrupt signals should be routed in a clever way, so that interrupt requests are evenly distributed across the PIRQ lines. But the largest issue can arise when a device writes data to memory and raises a pin-based interrupt to signal the CPU that the data has been written. It is possible that the interrupt arrives before all the data has arrived in memory. In order to ensure that all the data has arrived in memory, the interrupt handler must read a special register on the PCI device, which raised the interrupt. This read will not be completed until any pending transactions in-between the CPU and the PCI device complete. PCI transaction ordering rules require that all the data arrive in memory before the value may be returned from this special register. Thus, this dummy read guarantees that all the effects of the event that triggered the interrupt will be visible to the CPU. But this dummy read adds extra latency and work to the interrupt handler. Using message signaled interrupts avoids this problem as the interrupt message is a memory write transaction. Hence, it cannot pass the data writes, so when the interrupt is raised, the interrupt handler knows that all the data has been successfully written into memory.

To summarize, the advantages of MSI interrupts versus the legacy interrupts are:

- the MSI interrupts eliminate the need for interrupt traces between PCI devices and the I/O APIC,
- the MSI interrupts eliminate multiple PCI functions sharing the same PIRQ,
- the MSI interrupts eliminate the need for device polling in the interrupt handlers,
- the MSI interrupts eliminate the need to perform a read from a device's register o force all posted memory writes to be flushed to memory.

When a PCI function supports MSI, it generates an interrupt request to the processor's LAPIC by writing a predefined data item to a predefined memory address in the LAPIC. This PCI write transaction that contains a predefined data and a predefined address is referred to as an interrupt message. MSI was introduced in revision 2.2 of the PCI spec in 1999 as an optional component. However, with the introduction of the PCIe specification in 2004, implementation of MSI became mandatory from a hardware standpoint. It is worthwhile to notice that MSI interrupts can't work without LAPIC, but MSI eliminates the devices' need to use the IO-APIC, allowing every device to write directly to the CPU's LAPIC.

Each PCI function that generates MSI must contain two addressable registers (Message Data register and Message Address register) where BIOS or OS stores this predefined data and addresses during the initialization. When a PCI function asserts an interrupt using MSI, it performs a PCI write operation that writes the content of the Message Data register to the address specified in the Message Address register.

The following is the sequence for MSI delivery and servicing:

1. A device needing servicing from the CPU generates an MSI, writing the interrupt vector number directly into the Local-APIC of the CPU servicing it.
2. The interrupted CPU begins running the handler associated with the interrupt vector number it received. The device is serviced without any need to check and clear an IRQ pending bit

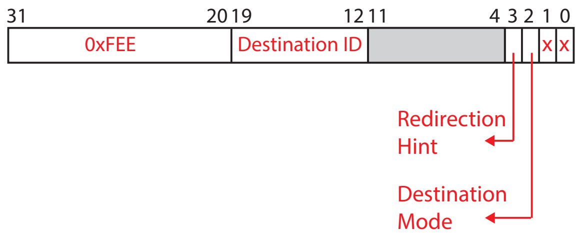


Fig. 3.24: Layout of the MSI Memory Address register.

The format of the Message Address register is presented in Figure 3.24. Fields in the Message Address Register are as follows:

- Bits 31-20 contain a fixed value for interrupt messages (0xFEE). This value locates interrupts at the 1-MByte area with a base address of 0xFEE00000. All accesses to this region are directed as interrupt messages.
- Destination ID — This field contains an 8-bit destination ID. It identifies the target LAPIC.
- Redirection hint (RH) — When this bit is set, the message is directed to the processor with the lowest interrupt priority among processors that can receive the interrupt. When this bit is reset, the interrupt is directed to the processor listed in the Destination ID field.
- Destination mode (DM) — This bit indicates whether the Destination ID field should be interpreted as logical or physical LAPIC for delivery of the lowest priority interrupt. If RH is 0, then the DM bit is ignored. If RH is 1 and DM is 0, only the processor listed in the Destination ID field is considered for delivery of that interrupt (this means no redirection). If RH is 1 and DM is 1, the redirection is limited to only those processors that are part of the logical group of processors based on the Destination ID field in the message.

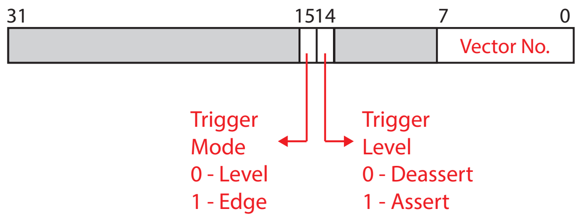


Fig. 3.25: Layout of the MSI Memory Data register.

Figure 3.25 illustrates the format of the Message Data register. The fields in the Message Data Register are:

- Vector number. This 8-bit field contains the interrupt vector number associated with the message. Values range from 0x10 to 0xFE. The software must guarantee that the field is not programmed with vector 0x00 to 0x0F as this are reserved for non-external interrupts and exceptions.
- Trigger Mode. If this bit is 0, the interrupt is edge-triggered. If this bit is 1, the interrupt is level-triggered.
- Level. For edge-triggered interrupts this field is ignored. For level-triggered interrupts, this bit reflects the active state of the interrupt input.

Chapter 4
Direct memory access

CHAPTER GOALS

Have you ever wondered how information travels between input-output devices and main memory in a computer system? In this chapter, we provide a detailed explanation of the Direct Memory access (DMA) I/O technique used in modern computer systems, including those using the Intel and ARM family of microprocessors. This chapter also aims to demystify the DMA controller internals and its programming with various peripherals. Upon completion of this chapter, you will be able to:

- Distinguish between programmed IO, interrupt-driven IO, and DMA transfers.
- Explain the operation of the signals used in direct memory access controllers.
- Explain the function of the Intel 8237 DMA controller when used for DMA transfers.
- Explain the function of the DMA controller used in STM Cortex-M based systems.
- Explain the function of bus-mastering (also referred to as first-party DMA).

4.1 Introduction

4.2 Programmed Input/Output

The programmed I/O was the most straightforward type of I/O technique for the exchanges of data between I/O devices and memory. This data transfer method requires the least amount of hardware. With programmed I/O, data transfers between

I/O devices and memory are accomplished by the central processing unit (CPU). In the case of programmed I/O, *the I/O device does not have direct access to the main memory*. The I/O devices have memory-mapped registers. This means that the CPU accesses the I/O device's registers using LOAD/STORE instructions.

A transfer from an I/O device to the main memory (or vice versa) requires the execution of several instructions by the CPU. This includes a LOAD instruction to transfer the data from the I/O device's data register(s) to the CPU and STORE instruction to transfer the data from CPU to the main memory. Besides, the CPU must continuously sense the I/O device's status. When the CPU issues a command to the I/O device, it must wait until the I/O operation is complete or new data is available. For example, before reading the data from the I/O device with the LOAD instruction, the CPU must first read the status register (also with a LOAD instruction) of the I/O device to check if the I/O device has new data. Similarly, before writing the data to the I/O device, the CPU must first read the status register (also with a LOAD instruction) of the I/O device to check if the I/O device is prepared to accept new data. As the CPU is faster than the I/O module, the problem with programmed I/O is that the CPU has to wait a long time for the I/O device to be ready for either reception or transmission of data. The CPU stays in the program loop until the I/O unit indicates that it is ready for data transfer. This process of waiting and checking the status of the I/O device is known as **polling or busy waiting**. As a result, polling severely degrades the level of the performance of the entire system. This situation can be avoided by using an interrupt-driven I/O, which we discuss in the next section.

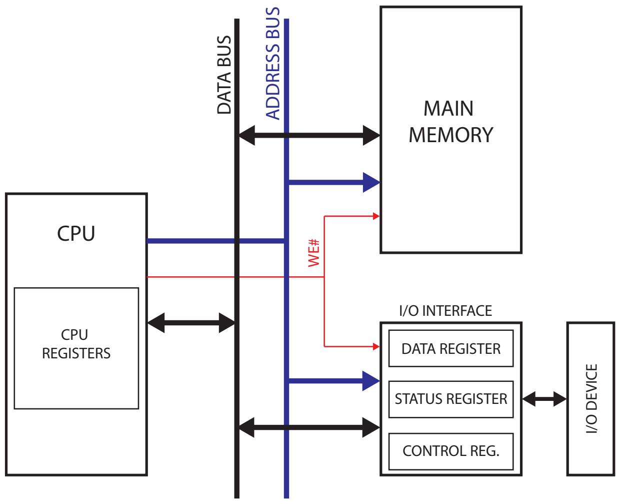


Fig. 4.1: A simplified block diagram of a computer system with programmed I/O.

Let's take a look at how programmed I/O would work if you were copying information from an I/O device to the main memory. Figure 4.1 illustrates a simplified block diagram of a computer system using programmed I/O. The I/O device shares the data, address, and control bus with the main memory. Although modern computer systems have more buses organized hierarchically, we can still simplify this discussion by assuming that there is only one bus in the system. The I/O devices in modern computer systems are memory-mapped, meaning that the CPU accesses these devices through a well-defined I/O interface. The I/O interface of an I/O device contains a set of registers, each of them having its unique address from the global address space. The CPU reads and writes to these I/O registers in the same way as it reads or writes to the main memory: using the LOAD and STORE instructions.

The I/O device in Figure 4.1 has three memory-mapped registers: a control register, a status register, and a data register. The control register is used to program the I/O device, e.g., set the data rate, parity check, etc. The status register reflects the status of the I/O device, e.g., the I/O device is ready to accept new data, or the I/O device has new data, etc. The data register is used to transfer data to/from the I/O device. In programmed I/O mode, the CPU would constantly check the status register to see if new data is available. Thus, the CPU would read the status register with the LOAD instruction and check a particular bit, which flags that the I/O device has new data. The CPU would perform the polling operation inside a program loop. In the case new data is available, the CPU would first transfer data from the data register into an internal register with the LOAD instruction. Then, the CPU would transfer data from the internal register into the memory with a STORE instruction. Listing 4.1 illustrates the programmed I/O transfer from the I/O device to the main memory:

```

                ; wait for new data
busy_wait:      lw r1, status_reg;
                beq r1, r0, busy_wait;

                ; CPU transfers data
transfer:       lw r2, data_reg
                sw rw, mem_addr


```

Listing 4.1: Programmed I/O data transfer

While not in use anymore, programmed I/O mode transfers were used in older hard drives back a few decades ago when so-called DMA transfers didn't exist. For example, programmed I/O was used by the Western Digital WD1003, the hard disk controller used by the first PCs. Programmed I/O is still used now in some low-end and embedded computer systems. Also, the Intel 80286, 80386, and 80486 microprocessors used in personal computers were well suited to programmed I0 since they can move blocks of data with a single *String Move* instruction. This data move instruction allowed programmed I/O transfers to reach speeds of about 2.5 Mbytes/s. In an embedded system where the CPU has nothing else to do, busy waiting is reasonable. However, in a more sophisticated computer system where the CPU has to do other things, polling is inefficient, and a better I/O transfer method is needed.

4.3 Interrupt-driven I/O

A disadvantage of polling is that the CPU must continuously sense the I/O device's status a loop. The waiting may significantly slow down the system capability of executing other instructions and processing other data. The so-called interrupt-driven I/O could be more efficient. In interrupt-driven I/O, the I/O device, when ready for a new transfer, initiates the data transfer by interrupting the CPU. The CPU then executes the interrupt service program that transfers the data. Similarly, as in programmed I/O, the transfer from an I/O device to the main memory (or vice versa) requires the execution of a LOAD instruction to transfer the data from the I/O device's data register(s) to the CPU and STORE instruction to transfer the data from CPU to the main memory. But now, there is no need to wait in a loop and read the I/O device's status. The interrupt-driven I/O technique requires more complex hardware but makes far more efficient use of CPU time and capacities.

For the transfer from an I/O device to memory, the device interrupts the CPU when new data has arrived and is ready to be retrieved by the system processor. As most of the I/O devices have memory-mapped registers, the interrupt service program will then read the device's data register into a CPU register and store the data from the CPU register to the memory location.

For the transfer from a memory location to an I/O device, the device delivers an interrupt either when it is ready to accept new data or to acknowledge a successful previous data transfer. The interrupt service program will then read the memory location into a CPU register and store the data from the CPU register to the device's data register.

Hence, in the interrupt-driven I/O, the CPU continuously works on given tasks. When the I/O device is ready for the data transfer, such as when someone types a key on the keyboard or a serial communication interface is ready to transmit a new byte, it interrupts the CPU from its work to take care of the data transfer. The CPU can work continuously on a task without checking the input devices, allowing the devices themselves to interrupt it as necessary.

The interrupt-driven I/O is adequate for simple computer systems, but there are situations in modern computing that complicate the picture. For example, what if the CPU is executing som critical task that should not be interrupted? What if the CPU executes an interrupt service program corresponding to an interrupt with the priority higher than the priority of the current interrupt request? In such a case, the handling of the current interrupt request from an I/O device should be deferred.

4.4 Direct Memory Access

We have seen two different methods used to transfer data between I/O devices and the main memory: polling and interrupt-driven I/O. Both techniques work well with low-bandwidth devices and some low-end computer systems, and both methods use the CPU to move data. While moving data, the CPU can not perform any other operations, making both methods inappropriate for modern, high-speed computer systems.

An alternative mechanism is to offload the CPU and to have **another device transfer data directly to or from the main memory** - without involving the CPU at all. This mechanism is called **direct memory access (DMA)**. DMA is a feature that allows systems to access the main memory without any help from the processor. The special device that performs the DMA transfer is a **DMA controller**. A DMA controller offloads the CPU tremendously as it fulfills a memory transfer without intervention from the processor. When the transfer is finished, it signals the CPU with an interrupt.

The DMA controller transfers data between the main memory and an I/O device independent of the CPU. The CPU only initializes the DMA controller. A DMA transfer is fulfilled in the following steps:

1. The CPU initializes the DMA controller: it provides the source and destination addresses of the data to be transferred, the number of bytes to be transferred, and the type of transfer to perform (we will discuss these types later).
2. When data is available, **the I/O device requests the DMA transfer** from the DMA controller. The DMA controller then requests the bus from the CPU and becomes the master of the bus and starts the transfer. During the transfer, the DMA controller supplies the memory addresses and the control signals needed to complete the transfer. If the request form the I/O device requires more than one transfer, the DMA controller will automatically generate the next memory address(es) and will complete the entire DMA transfer of hundreds of thousands of bytes without involving the CPU. The modern DMA controllers usually contain FIFO buffers that help them deal with different timings and delays during a transfer.
3. When the DMA transfer is complete, the DMA controller interrupts the CPU. The CPU can then decide if more transfers are required and reinitialize the DMA controller for new DMA transfers.

I'm sure you are now wondering how the CPU accesses the main memory during a DMA transfer. Well, (usually) it does not. But wait, how does the CPU fetch instructions and data from the main memory if the DMA controller occupies the memory bus? Remember, that CPU never directly access the main memory - it always accesses the L1, L2, and L3 caches first, and only if there is a miss in the L3 cache, the memory controller transfers the cache line to/from the main memory. Thus, again we rely on the temporal and spatial data locality and assume that there is a very high probability that instructions and operands, needed by the CPU, are already in the cache(s). So, the DMA transfer usually does not prevent the CPU from fetching instructions and data. By using caches, the CPU leaves most of the memory bandwidth free for use by a DMA controller. In the case of the cache miss, the modern systems rely on multitasking: in that case, the OS would perform a task switch another (ready) task.

Summary: Direct Memory Access

Direct memory access (DMA) is a mechanism that allows us to offload the CPU and to have a DMA controller transfer data directly between a peripheral device and the main memory.

A DMA transfer starts with a peripheral device placing a DMA request to the DMA controller. The DMA controller then requests the bus from the CPU and starts the transfer. When the DMA transfer is complete, the DMA controller interrupts the CPU.

Because of the use of cache and memory hierarchy in modern computer systems, a DMA transfer does not prevent the CPU from fetching instructions and data.

Let's take a look at how a DMA controller transfers data between the main memory and an I/O device. Figure 4.2 illustrates a simplified block diagram of a system with a DMA controller. The DMA controller is connected to the data, address, and control buses. It also has six control signals: DREQ, DACK, HOLD, HLDA, END, and WE#. Two control signals, DREQ (DMA request) and DACK (DMA acknowledge), namely, are used between the DMA controller and an I/O device to request and acknowledge a DMA transfer. Another two control signals, HOLD (Hold request) and HLDA (Hold acknowledge), are used between the DMA controller and the CPU to request and acknowledge a DMA transfer. The WE# signal selects between the memory read or memory write operations, and the END signals the CPE that the DMA transfer is finished and data is ready for further processing. The DMA controller has two registers: the **address register** and the **count register**. The address register holds the address of the memory location from/to which the data is to be transferred. The count register holds the number of data words to be transferred. Both registers are memory-mapped, and the CPU is responsible for their initialization.

Before any data transfer takes place, the CPU should initialize the DMA controller's address and count registers. The CPU writes the address of the memory location to/from which the data is to be transferred into the address register, and the number of data words to be transferred into the count register. During the transfer, the DMA controller will decrement the count register after each data word is transferred. It will also increment or decrement the address register, depending on the mode of operation, and automatically store/read the data to/from consecutive memory locations. When the count register signals that there is no more data to be transferred, the DMA controller will activate the END signal. This signal is usu-

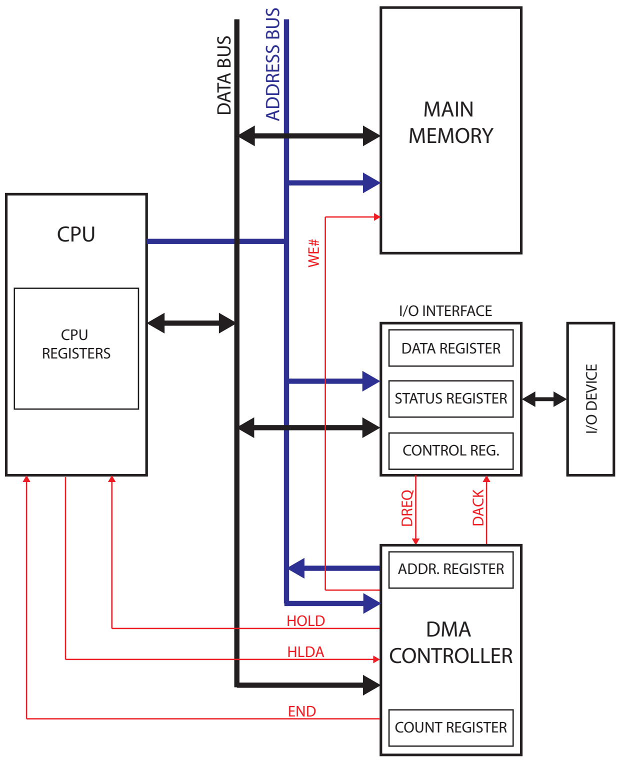


Fig. 4.2: A simplified block diagram of a computer system with a DMA controller.

ally connected to a CPU interrupt input and rises an interrupt when a transfer is completed.

Let's suppose the data transfer of one data word from the main memory to the I/O device. Firstly, the CPU writes the address of the memory location that holds the data into the address register, and the value of 1 into the count register, indicating that only one data word is to be transferred. The following steps are then required to accomplish the DMA transfer (Figure 4.3):

1. The I/O device is ready to receive data, so it asserts the DREQ signal.
2. The DMA controller requests the bus (requests the DMA transfer) from the CPU by asserting the HOLD signal.
3. The CPU relinquishes the control of the main memory. It voluntarily places all its bus signals at a high-impedance state and asserts the HLDA signal to indicate the bus is granted.
4. The DMA controller places the memory address from the address register on the address bus and puts the WE# signal into the high state to indicate the read access.
5. The main memory places the requested data onto the data bus.

6. The DMA controller asserts the DACK signal. This is to indicate that the I/O device can fetch data from the data bus. The DMA controller also decrements the count register.
7. The I/O device latches data from the data bus into its data register.
8. As the count register now indicates that there is no data left to transfer, the DMA controller de-asserts the HOLD signal to return the control over the bus to the CPU and activates the END signal to raise a CPU interrupt.
9. The CPU de-asserts the HLDA signal and eventually starts to service the interrupt request.

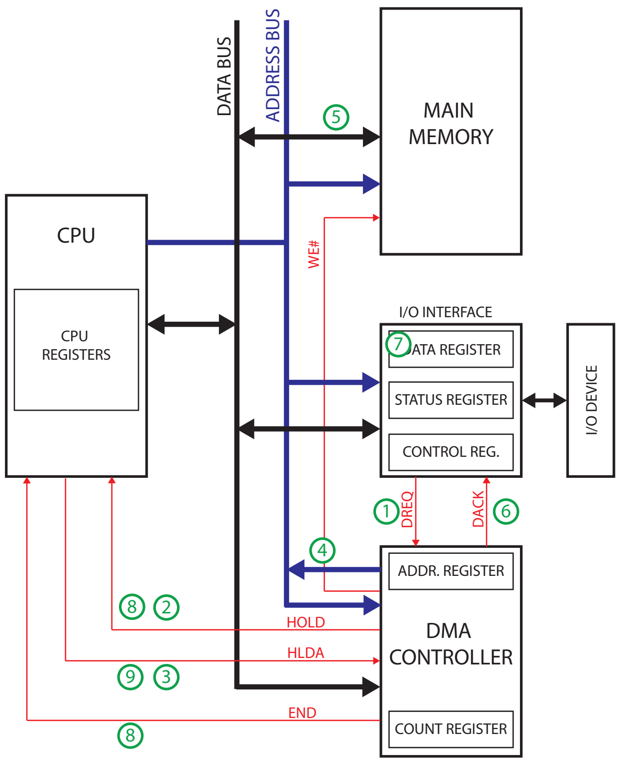


Fig. 4.3: A DMA transfer.

There is usually only one DMA controller in the computer system that is used for DMA transfers to/from several I/O devices. In that case, **the DMA controller has a separate pair of DREQ and DACK signals for each I/O device**. This separate pair (DREQ, DACK) is called **DMA channel**.

The DMA transfer described above is referred to as **"Fly-by" DMA**. This means that the data, which is transferred between an I/O device and memory, does not pass through the DMA controller. "Fly-by" DMA refers to the DMA transfer between an I/O device and memory in which the data flows into/out of the memory using address lines for only one side of the transfer (the memory side). The other side (the I/O device) is "addressed" by the DACK signal, i.e., the DACK signal selects the I/O device involved in the DMA transfer, which should then latch data from the bus or place data onto the bus.

The way that the DMA function is implemented varies between computer architectures, and there is also another type of DMA transfer referred to as **"Fly-through" DMA**. In "Fly-through" DMA, both source and destination address need to be specified. The data flows through the DMA controller, which now has a FIFO buffer to store the data temporarily. The "Fly-through" DMA controller first places the source address onto the address bus, reads the data from the source into its internal FIFO, then places the destination address onto the address bus and writes the data from its FIFO into the destination.

"Fly-by" DMA is much faster because "Fly-through" DMA results in two bus transfers: one from the source to the internal FIFO and the other from the internal FIFO to the destination. But on the other hand, "Fly-through" DMA enables the memory-to-memory DMA transfers, which are not possible with "Fly-by" DMA controllers.

Summary: DMA controllers

Each DMA transfer is driven by at least the DMA controller's internal two registers: the address register and the count register.

A DMA channel is a pair of two control signals between a peripheral device and the DMA controller: DMA request (DREQ) and DMA acknowledge (DACK).

In "Fly-by" DMA transfers, the data, which is transferred between an I/O device and memory, does not pass through the DMA controller. Only the memory address needs to be specified, while the peripheral device is selected by the DACK signal. Only one memory transaction is needed to accomplish a DMA transfer.

In "Fly-through" DMA, both source and destination address need to be specified. The data flows through the DMA controller, which has a FIFO buffer to store the data temporarily. The "Fly-through" DMA controller first places the source address onto the address bus, reads the data from the source into its internal FIFO, then places the destination address onto the address bus and writes the data from its FIFO into the destination. Two memory transactions are required to accomplish one DMA transfer.

4.5 Real-world DMA Controllers

So far, we have learned that a DMA controller is a special device used to transfer data between an I/O device and the main memory without involving the CPU. Modern DMA controllers also support memory-to-memory DMA transfers, thus allowing efficient data transfers between two memory regions. For example, on most modern computer systems, the C library function `memcpy()` is implemented using the DMA transfer. In this section, we are going to describe two real-world DMA controllers and their functionality: the Intel 8237A DMA "fly-by" controller used in the older Intel PCs, and the "fly-through" DMA controller used in modern ARM Cortex-M based systems.

4.5.1 Intel 8237A DMA controller

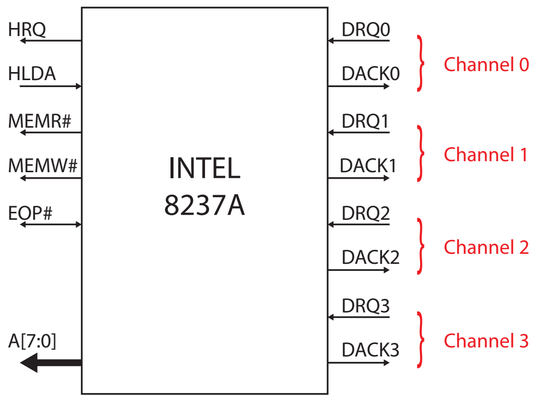


Fig. 4.4: Intel 8237A DMA controller. The signals used to initialize the DMA controller are not shown.

The Intel PC DMA subsystem is based on the Intel 8237A DMA controller. The Intel 8237A contains four DMA channels that can be programmed independently, and any of the channels may be active at any moment. These channels are numbered as 0, 1, 2, and 3. The Intel 8237A DMA controller moves one byte in each transfer and is very similar to the DMA controller described in Figure 4.2.

The Intel 8237A is depicted in Figure 4.4. It has two electrical signals for each channel, named DRQ (DMA request) and DACK (DMA acknowledge). There are additional signals with the names HRQ (Hold Request), HLDA (Hold Acknowledge), EOP# (End of Process), and the bus control signals MEMR# (Memory Read), and MEMW# (Memory Write). Table 4.1 provides full 8237A signals description.

Table 4.1: Intel 8237A signals description.

Signal name	Signal description
HRQ	Hold request is an output used to request the bus.
HLDA	Hold acknowladge is an input that signals that the CPU has granted the bus.
DREQ[3:0]	DMA request inputs are used to request a DMA transfer for each of the four DMA channels.
DACK[3:0]	DMA acknowledge outputs acknowledge the a channel DMA request and select the I/O device during the DMA transfer.
A[7:0]	These pins are outputs and are used to provide the DMA transfer memory address.
EOP#	End-of-process is a bidirectional active-low signal used used as an input to terminate a DMA transfer or as an output to signal the end of a DMA transfer.
MEMR#	Memory read is an active-low output used to read data from the selected memory location during a DMA transfer
MEMW#	Memory write is an active-low output used to write data to the selected memory location during a DMA transfer

The Intel 8237A DMA controller is a "fly-by" DMA controller. Subsequently, the DMA can only transfer data between an I/O device and a memory, but not between two I/O devices or two memory locations. Actually, the Intel 8237A controller does allow two channels to be connected to allow memory-to-memory DMA operations, but nobody in the PC industry used this DMA controller this way since it is faster to move data between memory locations using the CPU. Each DMA channel is activated only when an I/O device connected to that DMA channel requests a transfer by asserting the DRQ line.

Each channel in The 8237A DMA controller has two internal registers that control the transfer: the **count register** and the **address register**. Both registers are programmable by the CPU. The count register holds the number of bytes to be transferred, while the address register holds the (initial) memory address. When a byte of data is transferred, the address register is decremented or incremented, depending on how it is programmed. The count register is decremented after each transfer. When the value in the count register goes from zero to 0xFFFF, the EOP# output signal is activated.

The 8237A is designed to operate in two major cycles. These are called Idle and Active cycles. When no channel is requesting DMA transfer, the 8237A controller enters the Idle cycle. In this cycle, the 8237A samples the DREQ lines every clock cycle to determine if any channel is requesting a DMA transfer. When a channel requests a DMA service by asserting its DREQ signal, the 8237A asserts the HRQ signal to the microprocessor requesting the bus and enters the Active cycle. It is in the Active cycle that the DMA transfer will take place.

There are three modes of operation: single-mode, block mode, and demand mode. In single-mode, the device is programmed to make one transfer only. Single-mode transfer releases the HOLD signal after each byte is transferred. In this mode, DRQ must be held active until DACK becomes active. If the DRQ is held active, the 8237A again requests the bus with the HOLD signal. Upon receipt of a new HLDA, another single transfer will be performed.

Block mode automatically transfers the number of bytes indicated by the count register. The count register will be decremented, and the address register decremented or incremented following each transfer. In block mode, the DMA controller is activated by DREQ to continue making transfers until the count register goes from 0 to FFFFH, or an external EOP# is activated. DREQ need only be held active until DACK becomes active.

In demand mode, the DMA controller transfers data until an external EOP# is asserted or until DREQ goes inactive. This mode is used when there is a block of data to be transferred, but the I/O device has not a high data capacity, and the transfer should be paused until the I/O device is ready again. During the time when the transfer is paused, the CPU is allowed to use the bus, and the intermediate values of address and word count are preserved. When the I/O is ready to continue the transfer, the DMA transfer is re-established by activation of the DREQ signal.

Intel 8237A was also used for a DRAM refresh. To refresh one row in DRAM, the 8237A DMA controller reads data from memory onto the data bus. During this dummy read, sense amplifiers in the memory chips are enabled. This automatically leads to the refresh of one memory cell row. But the data is not fetched by an I/O device, as no device has issued a DRQ, and the DMA controller does not assert a DACK.

Summary: Intel 8237A DMA Controller

The Intel 8237A controller is a "fly-by" DMA controller. Subsequently, the DMA can only transfer data between an I/O device and a memory. Each DMA transfer requires only one memory transaction. It was used in Intel-based PC systems.

It contains four DMA channels, and any of the channels may be active at any moment. Each channel in The 8237A DMA controller has two internal registers that control the transfer: the count register and the address register. Both registers are programmable by the CPU.

4.5.2 *STM32Fx series DMA controller*

This subsection describes the direct memory access (DMA) controller available in the STM32Fx Arm Cortex-Mx core-based series of systems-on-chips. The STM32Fx series DMA controller allows data transfers to take place in the background, without the intervention of the Cortex-Mx processor. During this operation, the main processor can execute other tasks, and it is only interrupted when a whole data block is transferred and available for processing. The STM32Fx series DMA controller is a fly-through DMA controller and supports the transfer of large amounts of data with no significant impact on system performance. The DMA controller can do automated memory to memory data transfers, also do peripheral to memory and peripheral to peripheral.

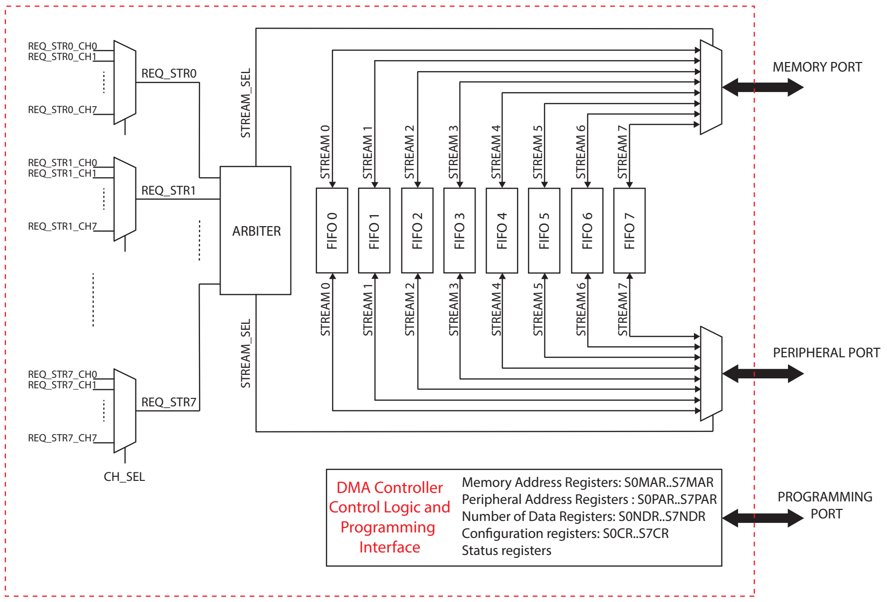


Fig. 4.5: Simplified block diagram of a STM32Fx series DMA controller.

Figure 4.5 illustrates the simplified block diagram of the STM32Fx series DMA controller. The STM32Fx series DMA controller features three ports: a programming port for DMA programming and two ports (peripheral and memory ports) that allow the DMA to initiate data transfers between different I/O devices and memory. A **port** is a connection to the data, address, and control bus. Thus, one port comprises data lines, address lines, and control signals, which are not depicted in Figure 4.5 due to simplicity.

Each STM32Fx series DMA controller supports up to eight streams. A **stream** is an active DMA transfer between a peripheral device and memory, two peripheral devices, or between two memory blocks. Each stream has eight selectable channels (requests). This selection is software-configurable and allows several peripherals to initiate DMA requests. Each channel is associated with a peripheral device that can trigger a data transfer request when ready. Thus, the DMA controller can be used by up to 64 I/O devices (64 channels) and can manage up to eight interleaved DMA transfers (streams). More than one enabled DMA stream must not serve the same peripheral request.

The DMA controller contains an arbiter for handling the priority between DMA streams. Stream priority is software-configurable. The arbiter selects the stream with the highest priority. If two or more DMA streams have the same software priority level, the lowest stream number gets priority. Also, the DMA channels can be assigned one of four priority levels: very high, high, medium, and low. Channel priority is also software-configurable. And if two same priority channels assert a DMA request at the same time – the lowest channel number gets priority.

When a peripheral is ready, it sends a DMA request to the DMA controller asserting a DMA request signal. The DMA controller will then serve the DMA request depending on the stream priority. As this is a "fly-through" DMA controller, the data flows through the DMA controller, which has a FIFO buffer associated with each stream. The FIFO buffer is used to store the data temporarily and to amortize the difference in transmission speeds of two peripheral devices. Standard block transfer is accomplished by the DMA controller performing a sequence of memory transfers. Each transfer involves a load operation from a source address into the FIFO, followed by a store operation from the FIFO to a destination address.

The DMA controller's control logic and programming interface are accessed through the programming port. The programming interface comprises a set of registers per stream. Each stream is characterized by four registers: Memory Address Register (SxMAR), Peripheral Address Register (SxPAR), Number of Data Register (SxNDR), and Configuration Register (SxCR). All these registers memory-mapped and will be discussed in the following subsections.

STM32Fx devices embed two DMA controllers (Figure 4.6), offering up to 16 streams in total (eight per controller), each dedicated to managing memory access requests from one or more peripherals. Data transfer direction of both DMA controllers can be from peripheral to memory, from memory to peripheral and from memory to memory (only the second DMA controller). The peripheral port of the second DMA controller can also be connected to memories in order to allow memory to memory transfers. The DMA Contoller 1 peripheral port is not connected to the bus like DMA controller 2. As a result, only DMA Controller 2 streams are able to perform memory-to-memory transfers.

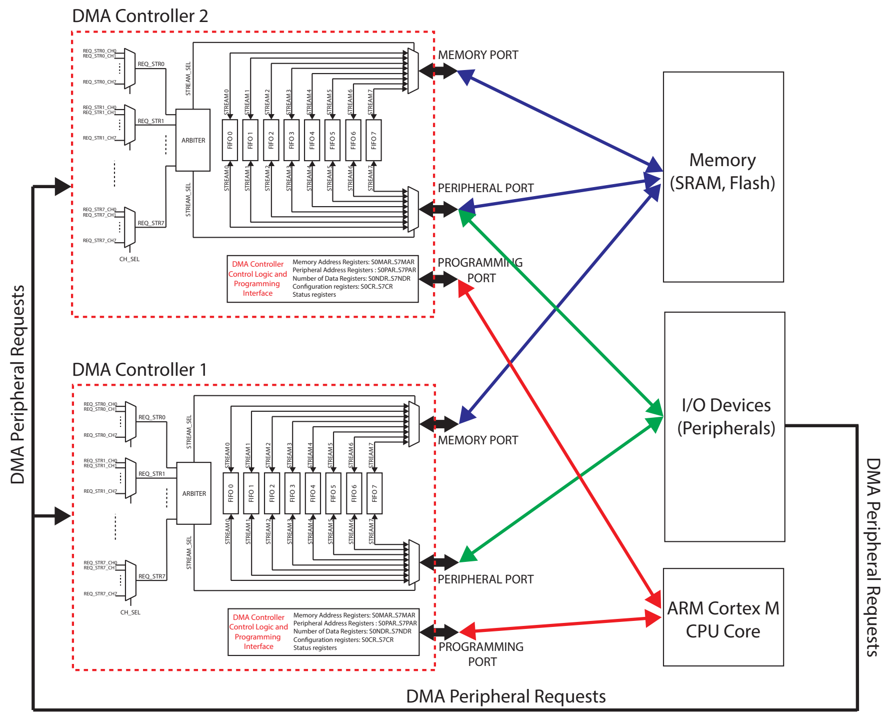

DMA Controller 2
DMA Controller 1
MEMORY PORT
PERIPHERAL PORT
PROGRAMMING PORT
ARBITER
DMA Controller Control Logic and Programming Interface
Memory Address Registers: S0MAR..S7MAR
Peripheral Address Registers : S0PAR..S7PAR
Number of Data Registers: S0NDR..S7NDR
Configuration registers: S0CR..S7CR
Status registers
Memory (SRAM, Flash)
I/O Devices (Peripherals)
ARM Cortex M CPU Core
DMA Peripheral Requests

Fig. 4.6: STM32Fx devices embed two DMA controllers. Data transfer direction of the DMA controller 1 can be from memory to peripheral and from peripheral to memory. Data transfer direction of the DMA controller 2 can be from memory to peripheral, from peripheral to memory and from memory to memory.

Summary: STM32Fx series DMA controller

The STM32Fx series DMA controller is a "fly-through" DMA controller used in the STM32Fx Arm Cortex-M based systems.

The STM32Fx series DMA controller features three ports: a programming port for DMA programming and initialization, and two ports (peripheral and memory ports) that allow the DMA to initiate data transfers between different I/O devices and memory.

Each STM32Fx series DMA controller supports up to eight *streams*. A stream is an active DMA transfer between a peripheral device and memory, two peripheral devices, or between two memory blocks. Each stream has eight selectable channels.

As this is a "fly-through" DMA controller, the data flows through the DMA controller, which has a FIFO buffer associated with each stream.

STM32Fx devices embed two DMA controllers. Data transfer direction of the DMA controller 1 can be from memory to peripheral and from peripheral to memory. Data transfer direction of the DMA controller 2 can be from memory to peripheral, from peripheral to memory and from memory to memory.

4.5.2.1 Peripheral and memory addresses

Each DMA transfer is defined by a source address and a destination address. Both addresses should be aligned to transfer size. The transfer size value defines the volume of data to be transferred from source to destination. Each stream has a pair of registers to store these addresses: Peripheral Address Register (SxPAR - *Stream x Peripheral Address Register*) and Memory Address Register (SxMAR - *Stream x Memory Address Register*). Before each transfer, the CPU should initialize both registers with the valid addresses. It is possible to configure the DMA to automatically increment the source and/or destination address after each data transfer.

4.5.2.2 Transfer size, type and mode

Each DMA transfer is defined by the transfer size and the transfer mode. The transfer size is a value that defines the volume of data to be transferred from source to destination. This value is stored in the so-called Number of Data Register (NDR). Each stream has its Number of Data Register, labeled as SxNDTR. Each SxNDTR is a 16-bit register, and the number of data items to be transferred is software programmable from 1 to 65535. After each transfer, the value in SxNDTR is decreased by the amount of the transferred data; thus, SxNDTR contains the number of data transfers still to be performed.

The STM32Fx series DMA controller can perform two transfer types: normal type and circular type. In normal type, once the SxNDTR register reaches zero (the transfer has completed), the stream is disabled. This means that the CPU should reinitialize the DMA controller in order to activate the stream again. In circular type, the DMA controller can handle circular buffers and continuous data flow. In this type, the SxNDTR register is reloaded automatically with the previously programmed value when a transfer has completed.

Each STM32Fx series DMA controller is capable of performing three different transfer modes:

1. peripheral to memory,
2. memory to peripheral,
3. memory to memory (only the second DMA controller is able to do such transfer; in this mode, the circular type is not allowed).

4.5.2.3 FIFOs and burst transfers

Each stream has a 4x32 bits FIFO that is used to temporarily store data coming from the source before transmitting them to the destination. The DMA FIFOs help to reduce memory access and to do burst transactions which optimize the transfer bandwidth. They also allow independent source and destination transfer width (byte, half-word, word): when the data widths of the source and destination are not equal, the DMA automatically packs/unpacks the necessary transfers to optimize the bandwidth. For example, the data from the source can be transferred into FIFO as bytes or 16-bit half-words and then transferred to the destination from FIFO as bytes, 16-bit half-words, or 32-bit words.

Because of the internal FIFOs, the DMA controller is capable of burst transfers of length 4x, 8x, or 16x data units. A data unit can be a byte, a 16-bit half-word, or a 32-bit word. The burst size on the DMA peripheral port must be set according to the peripheral needs/capabilities. The size of the burst is software-configurable, usually equal to half the FIFO size of the peripheral.

4.5.2.4 Programming and using the STM32Fx series DMA controller

Programming and using the STM32Fx series DMA controller is relatively easy. Each stream is controlled using four memory-mapped registers: memory address register (SxMAR), peripheral address register (SxPAR), number of data register (SxNDTR), and configuration register (SxCR). Once set, the DMA controller takes care of data transfers and memory address increment without disturbing CPU. To configure the DMA controller and a DMA stream, the following procedure should be applied:

1. If the stream is enabled, disable it by resetting stream enable bit in the SxCR register.
2. Set the peripheral port register address in the SxPAR register. The data will be moved from/to this address to/from the peripheral port after the peripheral DMA request.
3. Set the memory address in the SxMAR register. The data will be written to or read from this memory after the peripheral DMA request.
4. Configure the total number of data items to be transferred in the SxNDTR register. After each (burst) transfer, this value is decremented accordingly.
5. Select the DMA channel (request) and configure the stream priority, the data transfer direction, single or burst transactions, peripheral and memory data widths, circular/normal type, interrupts in the SxCR register.
6. Activate the stream by setting the stream enable bit in the SxCR register.

As soon as the stream is enabled, it can serve any DMA request from the peripheral connected to the stream and DMA transactions using the stream can be performed.

4.5.2.5 DMA Transactions

After a peripheral is ready for DMA transfer, the peripheral sends a request signal to the DMA controller. The DMA controller serves the request depending on the channel priorities. A DMA transaction consists of a sequence of a given number of data transfers. The number of data items to be transferred and their width (8-bit, 16-bit, or 32-bit) are software programmable. Each DMA transfer consists of three operations:

1. A **loading** from the peripheral data register or a location in memory, addressed through the SxPAR or SxMAR register. The data is loaded into the stream's FIFO. If the load is from a location in memory, the SxMAR register is incremented or decremented.
2. A **storage** of the data from the stream's FIFO to the peripheral data register or a location in memory addressed through the SxPAR or SxMAR register. If the store is to a location in memory, the SxMAR register is incremented or decremented.
3. A post-decrement of theSxNDTR register, which contains the number of transactions that still have to be performed.

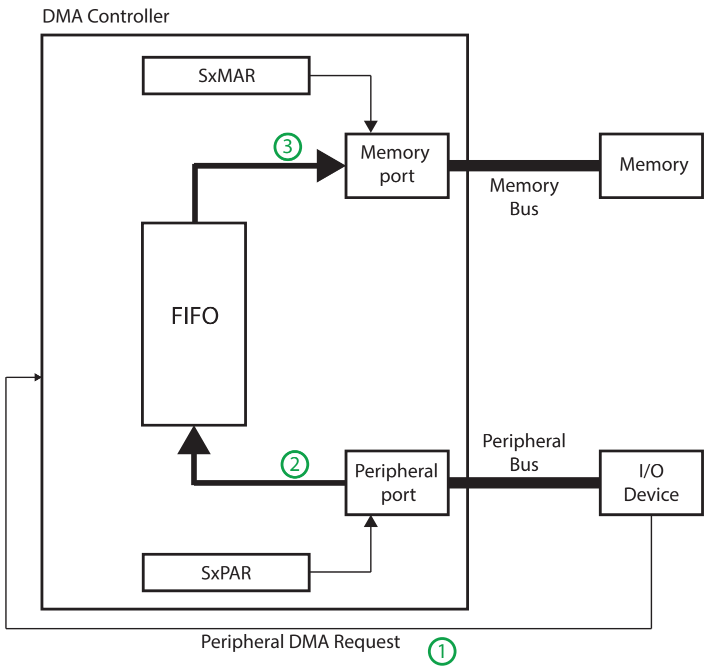


Fig. 4.7: A peripheral-to-memory DMA transaction.

Figure 4.7 illustrates a peripheral-to-memory DMA transaction. Each time a peripheral request occurs, the stream initiates a transfer from the source (address is in SxPAR) to fill the FIFO. Then, the contents of the FIFO are drained and stored in the destination (address is in the SxMAR). The transfer stops once the SxNDTR register reaches zero, or when the enable bit in the SxCR register is cleared by software (stream disabled).

Figure 4.8 illustrates a memory-to-peripheral DMA transaction. In this mode, the stream immediately initiates transfers from the source (address is in SxMAR) to en-

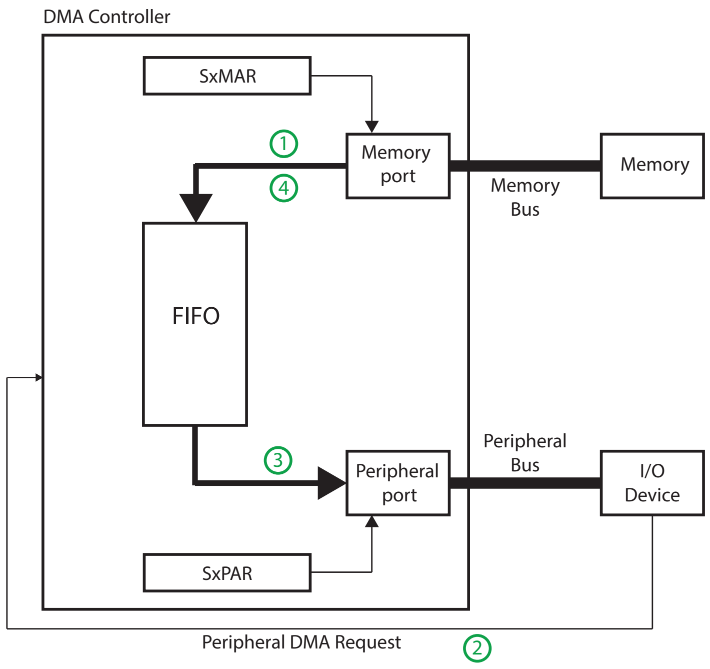


Fig. 4.8: A memory-to-peripheral DMA transaction. In this mode, the stream immediately initiates transfers from the memory to entirely fill the FIFO. When a peripheral request occurs, the contents of the FIFO are stored in the peripheral device.

tirely fill the FIFO, and the SxMAR register is incremented/decremented. The DMA controller does not wait for DMA request from a peripheral device to read from memory. When a peripheral request occurs, the contents of the FIFO are drained and stored in the destination (address is in the SxPAR). The DMA controller then reloads the empty internal FIFO again with the next data to be transferred from memory (address is in SxMAR). The transfer stops once the SxNDTR register reaches zero, or when the enable bit in the SxCR register is cleared by software (stream disabled).

Summary: STM32Fx series DMA transfers

Each DMA transfer is defined by a source address and a destination address, and each stream has a pair of registers to store these addresses: Peripheral Address Register (SxPAR -Streamx Peripheral Address Register) and Memory Address Register (SxMAR -Stream x Memory Address Register)

Each DMA transfer is defined by the transfer size and the transfer mode. Each stream has its Number of Data Register (SxNDR), which stores the transfer size.

The STM32Fx series DMA controller can perform two transfer types: normal type and circular type.

FIFOs allow independent source and destination transfer width and burst transfers.

Each DMA transfer consists of two transactions on the bus: loading from the peripheral data register or a location in memory, and storage of the data to the peripheral data register or a location in memory.

4.6 Bus Mastering DMA

So far, we have learned that we can use a special piece of hardware, a DMA controller, namely, to transfer large amounts of data between a peripheral device and memory. This approach is sometimes referred to as **third-party DMA**. Third-party DMA requires an independent DMA controller, which is built into motherboard chipsets, to move data between a peripheral device (referred to as the *first party*) and system RAM (referred to as the *second party*). Here, the DMA controller is shared by multiple peripheral devices, which is why it is viewed as the third party DMA. As we have learned previously, each "fly-through" DMA transfer (fly-through is the type of the DMA used in the majority of today's computer systems) requires two memory transactions: one to load the data from the source, and one to store the data to the destination.

The better approach to DMA transfers would be to have only one memory transaction per DMA transfer, but still avoiding third-party "fly-by" DMA controllers. This is possible with the latest I/O devices built in the modern computer systems, where each I/O device can act as a *bus master*, i.e., each device can directly access any other I/O device or memory on the bus. Indeed, each modern I/O device now contains its own, integrated, DMA controller, which is not shared by other I/O devices. This highest performing DMA type is called first-party DMA or **Bus Mastering DMA**. Peripheral devices, which support the Bus Mastering technology, have the ability to move data to and from system memory without the intervention of the CPU or a third party DMA controller.

Bus Mastering allows data to be transferred much faster than third party DMA. This is because half as many bus cycles are needed. The third-party DMA requires the DMA controller to alternately read a segment of data from one device (this can be a peripheral device or system memory) and write it to the other device. Each data segment requires at least one bus cycle to be read and one bus cycle to be written. Bus mastering devices only require bus cycles when accessing system memory, so half as many bus cycles are needed. Because of this, devices that support Bus Mastering can move data many times faster than third party DMA. While bus mastering theoretically allows one peripheral device to directly communicate with another, in practice almost all peripherals master the bus exclusively to perform peripheral-to-memory and memory-to-peripheral transfers.

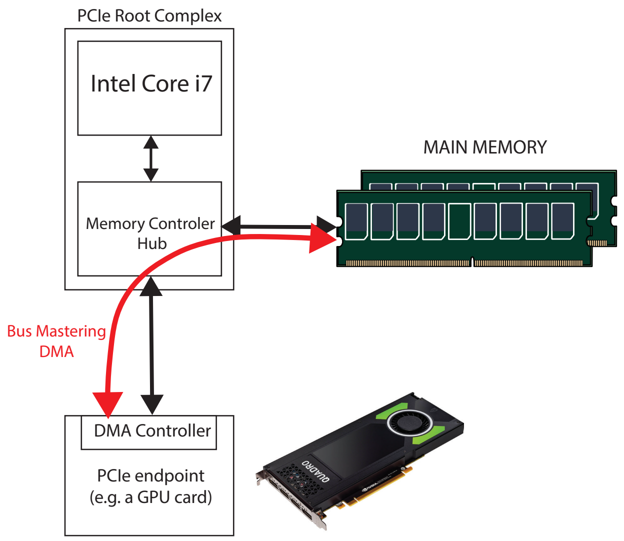


Fig. 4.9: Bus Maserting in an Intel based system. Bus Mastering is the feature integrated into PCIe endpoint devices. A DMA transfer either transfers data from an endpoint device into system memory or from system memory into the endpoint device on the PCI Express bus. The DMA request is always initiated by the integrated DMA controller in the endpoint device after being initialized from the application driver (i.e., receiving parameters that define DMA transaction and memory buffer address).

Bus Mastering is used in the computer systems with a PCI Express (PCIe) bus. A Bus Mastering DMA implementation is by far the most common type of DMA found in systems based on PCI Express and resides within the peripheral device, which is called Bus Master because it initiates the movement of data to and from system memory. Figure 4.9 shows a typical Intel system architecture. The system includes the CPU core(s) and a memory controller hub—these two form the so-called PCIe root complex. The system in Figure 4.9 also contains the main memory and one PCIe peripheral device (e.g. a GPU card). The peripheral device is connected to the PCIe bus. PCIe peripheral devices are called PCI **endpoint devices**. The memory controller hub also acts as a bridge between the PCIe bus, the CPU bus, and the main memory bus. A DMA transfer either transfers data from an endpoint device into system memory or from system memory into the endpoint device on the PCI Express bus. The DMA request is always initiated by the integrated DMA controller in the endpoint device after being initialized from the application driver (i.e., receiving parameters that define DMA transaction and memory buffer address).

Summary: Bus Mastering

Bus Mastering DMA is the highest performing DMA type. Peripheral devices, which support the Bus Mastering technology, have the ability to move data to and from system memory without the intervention of the CPU or a third party DMA controller.

Bus Mastering is used in the computer systems with a PCI Express (PCIe) bus.

PCIe peripheral devices are called PCI **endpoint devices**. Endpoint devices have their own integrated DMA controller, which is not shared by other endpoint devices.

References

1. IBM: Understanding DRAM Operation (1996). URL https://compas.cs.stonybrook.edu/{~}nhonarmand/courses/sp15/cse502/res/dramop.pdf