Chapter 6

Physical Memory

In the next three chapters, we will turn our attention to storage. This chapter (Chapter 6) deals with physical memory, actual real memory that we can access if we know the actual address of the memory location we wish to access. Chapter 7 will deal with virtual memory, the amount of memory each process thinks it has, which is usually much, much larger than the physical memory it actually has. Most of this virtual memory actually resides on the disk so any access to one of its locations has to first check its virtual address to see if it corresponds to an actual physical memory location, or if some intervention is necessary to obtain it from the disk. Finally, Chapter 8 will deal with Cache Memory, an even smaller fraction of storage that is located on the microprocessor itself.

Our goal is to approximate an infinite capacity memory that can be accessed in a single clock cycle. Unfortunately, the laws of physics do not permit that, so the best we can do is approximate it by the effective utilization of cache memory, physical memory, and what is stored on the disk (virtual memory). Together they form what we call the storage hierarchy.

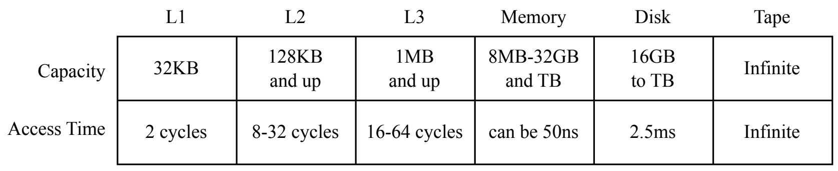
6.1 The Storage Hierarchy

The structures that make up the storage hierarchy are: registers, on-chip storage called cache memory, physical memory, and disk. The cost of each storage device is inversely proportional to its access time, Not surprisingly, we we have fewer units of devices that we can access faster. Registers are the fastest structures to access but they are also the most expensive and we have fewer of them.

In the early days of computers, an ISA had one register, which grew to 8 registers in the late 1970s with for example the Intel 8086 and Motorola 68000, and to 32 registers in the mid 1980s. Today, some processors have hundreds of registers, but this is still far less storage than the next fastest storage structure, the first level cache. Originally, processors did not have caches. By the late 1980s, most computers had two levels of cache, the first level having fewer storage locations but being much faster than the larger but slower second level cache. The rationale was to make the first level cache fast enough to approximately keep pace with the speed of the on-chip logic, and the second level cache large enough to hopefully find the needed storage location on the chip. Having a faster first level cache made it smaller. Having a larger second level cache made it slower. Such are the tradeoffs of the laws of physics. Today, three levels of cache is not uncommon with the third level being much larger and much slower than the second level.

Slower than caches is physical memory where access time can take hundreds of clock cycles to access, but have capacities of multiple megabytes or gigabytes. In fact, some computer systems having memory capacity in the terabyte range. Larger still is disk storage with capacities in the gigabyte to terabyte range and having access times of milliseconds.

Figure 6.1 gives representative numbers of size and access time of each of the members of the storage hierarchy.



	L1	L2	L3	Memory	Disk	Tape
Capacity	32KB	128KB and up	1MB and up	8MB-32GB and TB	16GB to TB	Infinite
Access Time	2 cycles	8-32 cycles	16-64 cycles	can be 50ns	2.5ms	Infinite

Figure 6.1: Capacity and Access Times of Storage Structures

6.2 Access Methods

Different storage structures use different methods of accessing storage. The most common method, used in Cache Memory and physical memory is RAM, which stands for Random Access Memory. Its defining characteristic is that the access time of the next location accessed is independent of the location of the current location being accessed. In reality, an examination of any computer program will note that if the current location accessed is N, the address of the next location access is most frequently N+1, and most microarchitectures go to a good deal of trouble to exploit such behaviors. Nonetheless, the myth persists: RAM implies the address of the next location accessed is independent of the address of the current location being accessed.

Disks are accessed by a method referred to as DASD, which stands for Direct Access Storage Device. This method behaves according to the accesses performed by the disk: a disk head is moved into position that it is hovering just above the track that is to be read. Once the head is in place, the method waits until the rotating disk puts the start of the data to be read under the track. Then a page is read, a page consisting of some number of sequential accesses.

Tape is accessed sequentially. If location N is currently being accessed, a tape will access location N+1 next. This is the slowest access method, and suffers particularly from the situation where location N+1 is not the location usually wanted next. The result: a lot of time wasted moving the tape forward or reverse. Tape is pretty much today not part of the storage hierarchy.

Finally, there are accesses wherein part of the address is contained within the data accessed. Such accesses are referred to as Content Addressable. The structure is referred to as CAM (content addressable memory). Two very widespread structures, the Translation Lookaside Buffer in a virtual memory system and the Tag Store in a cache memory both utilize CAM. We will discuss both in detail in Chapters 7 and 8.

6.3 Aligned and Unaligned Access

6.3.1 The Definition of Alignment

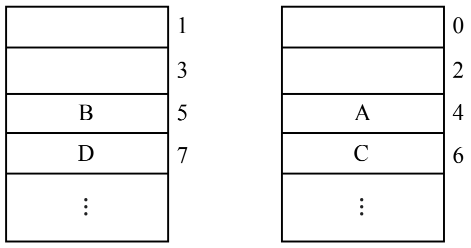
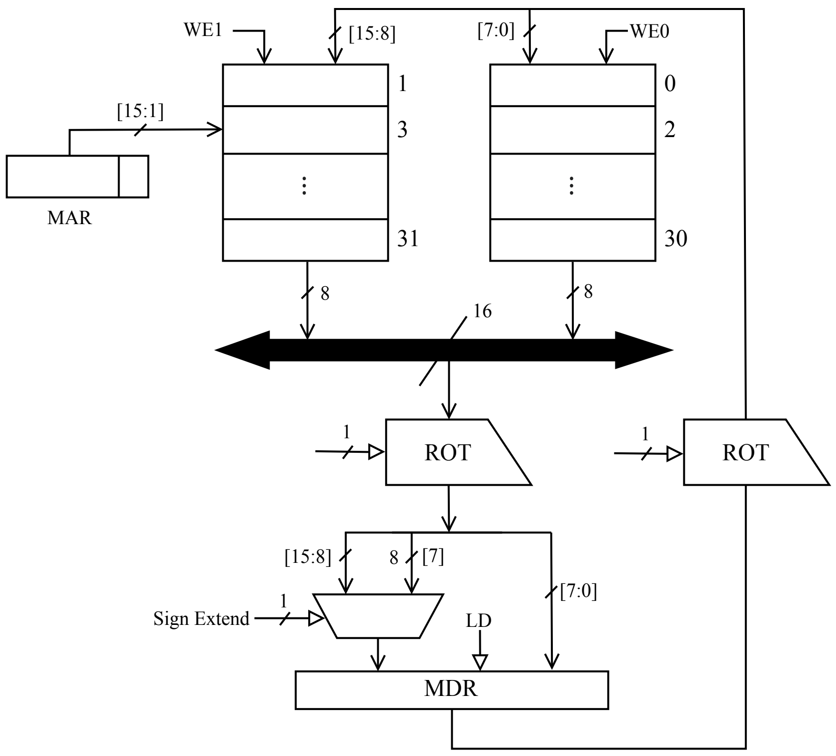


Figure 6.2: Aligned and Unaligned Information

Whether a computer system will require memory accesses to be aligned or whether unaligned accesses will be allowed is an important specification in an ISA. An aligned access is one in which all bytes of the access can be accomplished with the same address being used for all bytes required by the access.

Figure 6.2 shows a byte-addressable memory consisting of two chips, one containing the contents of all even addressed locations and the other containing the contents of all odd addressed locations. For example, a word consisting of the bytes A and B in Figure 6.2 which have addresses 000100 and 000101 is considered aligned because if you omit the low order bit of the two addresses, the resulting addresses 00010 and 00010 are identical, resulting in locations 4 and 5 both being read at the same time in a single memory access using the same memory address.

On the other hand, a word consisting of locations 000101 and 000110 does not result in an aligned access since omitting the low order bit of these two addresses results in the addresses 00010 and 00011. To access locations 5 and 6 in the same clock cycle, either we need extra logic to create the different addresses to address the two chips at the same time, which could increase the cycle time, or we could make the access of locations 5 and 6 as a single word require two separate accesses. Neither is a desirable choice, so one solution is to not allow unaligned accesses.



LD/ST	MAR[0]	W/B	ROT	Sign Extension	LD	Exception	WE1	WE0
LD	0	W	0	0	1	0	0	0
LD	0	B	0	1	1	0	0	0
LD	1	W	X	X	0	1	0	0
LD	1	B	1	1	1	0	0	0
ST	0	W	0	X	0	0	1	1
ST	0	B	0	X	0	0	0	1
ST	1	W	X	X	0	1	0	0
ST	1	B	1	X	0	0	1	0

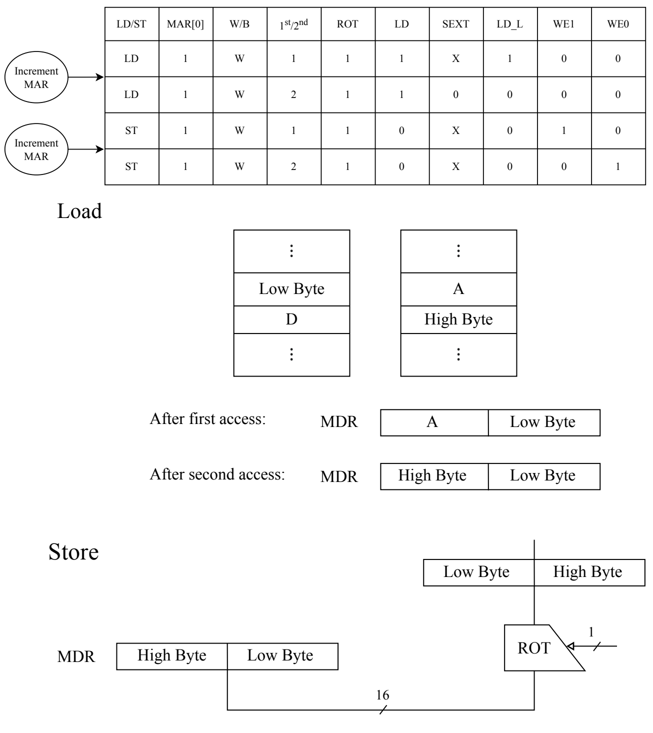
Figure 6.3: Aligned Accesses

6.3.2 Aligned access

Figure 6.3 shows all the structures needed to allow aligned accesses, and not allow unaligned accesses. The truth table shows eight input combinations, representing the eight possible situations: Load or Store, Word or Byte access, and 0 or 1, the low order bit of the address being access.

For word loads, each chip supplies a byte, if aligned (i.e., MAR[0]=0, the bytes are loaded into the MDR, the low address byte into the low byte position of the MDR and the high address byte into the high byte position of the MDR. For byte loads, both chips supply a byte, the relevant byte (MAR[0] is 0 or 1 is rotated into the low byte position, the high byte is supplied as 8 copies of bit[7] of the low byte, resulting in the byte loaded as a 16 bit word into the MDR. If the operation is a load and MAR[0]=1, an unaligned access is indicated, creating an exception.

The story for stores is pretty much the same. For word stores, with MAR[0]=0, the MDR supplies the two bytes and WE1 and WE0 signify both bytes are loaded. For byte stores, MAR[0] indicates where the low order byte of the MDR is to be stored, and the corresponding WE signal enables the load. If the operation is a word store and MAR[0]=1, an unaligned access is indicated, creating an exception.



LD/ST	MAR[0]	W/B	1st/2nd	ROT	LD	SEXT	LD_L	WE1	WE0
LD	1	W	1	1	1	X	1	0	0
LD	1	W	2	1	1	0	0	0	0
ST	1	W	1	1	0	X	0	1	0
ST	1	W	2	1	0	X	0	0	1

Figure 6.4: Unaligned Accesses

6.3.3 Unaligned Access

Figure 6.4 shows what is required in order to allow unaligned word loads and unaligned word stores. Our preference is to require two memory accesses, rather than increase the cycle time.

For an unaligned word load, the first access rotates the two bytes from memory in order to align the low byte with its correct location in the MDR. The high byte is also loaded into the MDR but that is not a problem since it will be overwritten in the second access. The memory address is then incremented to provide access to the high byte of the data being loaded. Finally, on the second access, the memory data is again rotated to align the high byte with the high byte of the MDR, and loaded into the high byte of the MDR by the LD_H control signal.

Figure 6.4 also shows what is required for unaligned word stores to be performed.

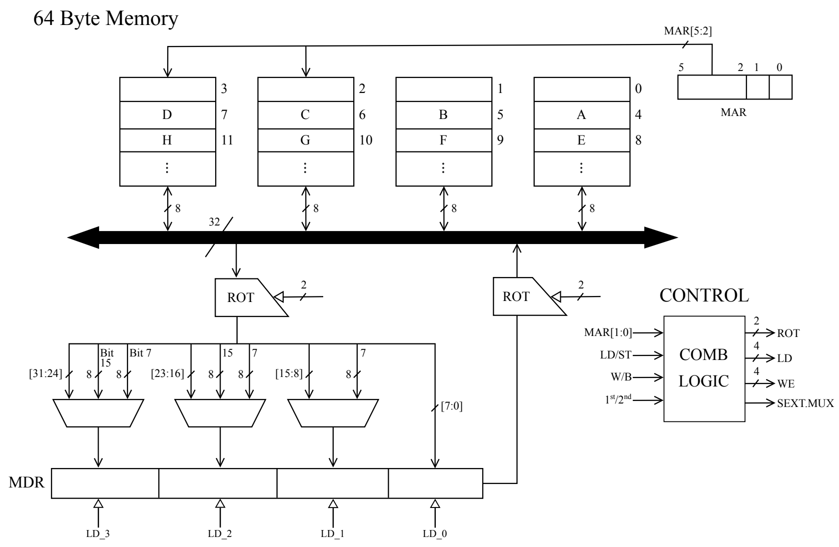


Figure 6.5:

Figure 6.5 shows the structure needed to handle unalignment if alignment is based on 32 bit boundaries. With 32 bit boundaries we need four chips. The word consisting of DCBA can be loaded or stored as an aligned access, but the word consisting of GFED requires an unaligned access. Note that in the case of DCBA, the four addresses 00100, 00101, 00110, and 0011 are identical if the two low order bits are removed from each address, but in the cae of GFED, that is not the case.

6.3.4 A final word about alignment

Like so much of computer architecture, the choice of allowing or not allowing unaligned accesses is based on a tradeoff, whether it is more important to opt for higher performance, or opt for ease of use. If you insist that all accesses need to be aligned then the programmer (or compiler writer) needs to know enough about memory organization to know what makes an access aligned or unaligned. If you allow unaligned accesses you make life Easier for the programmer at the expense of performance.

I am reminded of the products of the Digital Equipment Corporation. The PDP 11, introduced in 1970, required all memory accesses to be aligned. Their next product, the VAX (in 1977) permitted unaligned accesses. Their next product, Alpha (in 1972) required all memory accesses to be aligned. Different times, different resasons for what is necessary.

6.3.5 Interleaving

A common problem with memory operations is the fact that if a memory access is in progress, one can not access the part of memory performing the access until the current access finishes. An effective way to deal with this is called interleaving the memory, that is, partitioning the memory into units called banks, so that while one bank is processing a memory access, another bank is free to initiate a subsequent memory access.

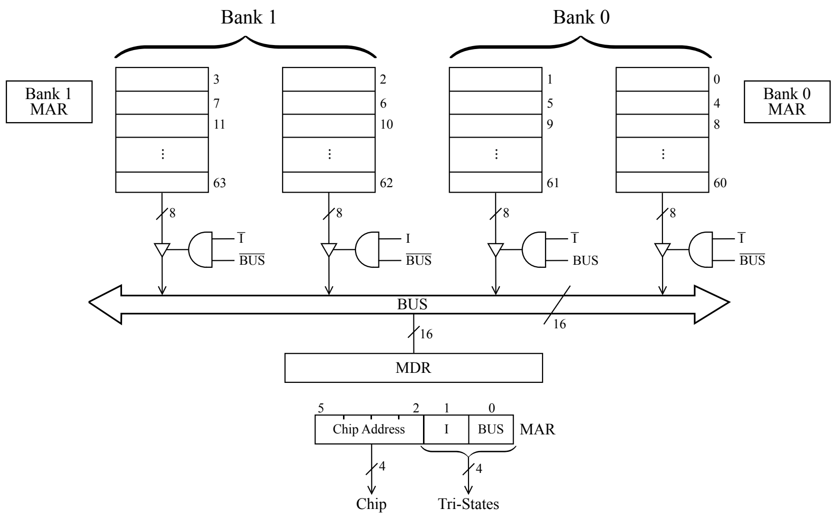
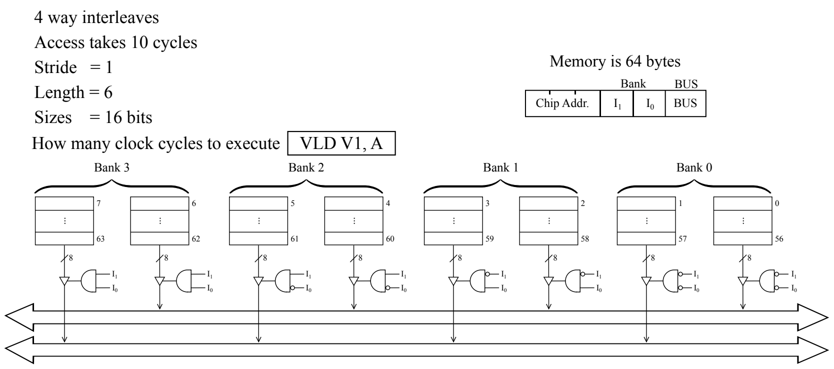


Figure 6.6: Two-way Interleaving

Figure 6.6 shows a very simple implementation, two-way interleaving. There



Six accesses

	Initiate	Finish
Bank 0	Cycle 0	Cycle 9
Bank 1	1	10
Bank 2	2	11
Bank 3	3	12
Bank 0	10	19
Bank 1	11	20

Figure 6.7: Interleaving, an important mechanism for vector instructions

are two banks. While one bank is processing an access, the other bank does not have to wait for that bank to finish, but can initiate a second memory access when it is ready to do so. Note that each bank requires its own MAR for loading the address of the memory location to be read or written, and each bank needs its own internal mechanism for accessing the data preparatory to loading the memory system's MDR on a load or obtaining the data from the processor preparatory to storing it in the memory system on a store.

Note the bus is 16 bits wide, requiring two chips to load the bus from each bank. Notice also that in a given clock cycle the bus is loaded from bank 0 or bank 1, requiring an interleave bit for tri-stating which bank accesses the bus.

Figure 6.7 shows a more complicated example, one in which a vector instruction is executed. A vector instruction is processed by loading (or storing) N values, from addresses Base, Base+k, Base+2k, ... Base+(N-1)k. Computers supporting vector instructions require vector registers, registers that can contain more than a single value. These computers also need two additional registers that need to loaded with N and k before the vector instruction is performed. In the example of Figure 6.7, the vector instruction to be performed is VLD V1,A. VLD means vector load, A is the Base address, from which values are to be loaded. V1 is vector register 1, where the values will be loaded. In this example, we are assuming k is 1 and N is 6. In computerese, k is referred to as the "stride," the distance between memory locations being accessed, and N is referred to as the "length," the number of values to be loaded.

Here we show the implementation of a vector load instruction, where (in the case shown) the vector instruction VLD V!,A is carried out by loading the contents of the six sequential memory locations, starting at location A into the first six component registers of vector register V1.

In the example of Figure 6.7, the memory is 4-way interleaved. The access time for each bank is ten clock cycles. We ask the question: once the length and stride are loaded with N and k, how many clock cycles will it take to perform the vector load instruction.

In clock cycle 0, we initiate the load of A which let's say is in Bank 0 into the first component of vector register V1. That load will complete a the end of cycle 9. In clock cycle 1, we can initiate the load of the memory location having address A+1 since that location is part of Bank 1 and so it does not have to wait for the load in Bank 0 to finish. In cycles 2 and 3, we can initiate the loads of A+2 and A+3 from Bank 2 and Bank 3. In clock cycle 3 we would like to load from Bank 0, but since it is done processing the load from A, the computer must wait until the contents of location A finishes loading, i.e., at the end of clock cycle 9. Thus the load of A+4 can not start until clock cycle 10. The load of A+1 will finish from Bank 1 at the end of cycle 10, so the load of A+5 can start in Bank 1 at the beginning of cycle 11. Since it will take ten clock cycles, the vector instruction will finish at the end of 20.

Our final example of interleaving involves the CRAY 1, a vector supercomputer that stored 64 bit values in each memory location, i.e., it had an addressibility of 64 bits. The CRAY 1 had an access time of 11 clock cycles. To optimize the time it took to do a vector load, its chief architect Seymour Cray decided to make the memory 16-way interleaved. Thus, in the clock cycle after Bank 15 was initiated, the hardware was free to initiate the load of A+16 from Bank 0 since Bank 0 had completed the load of A at the end of clock cycle 11.

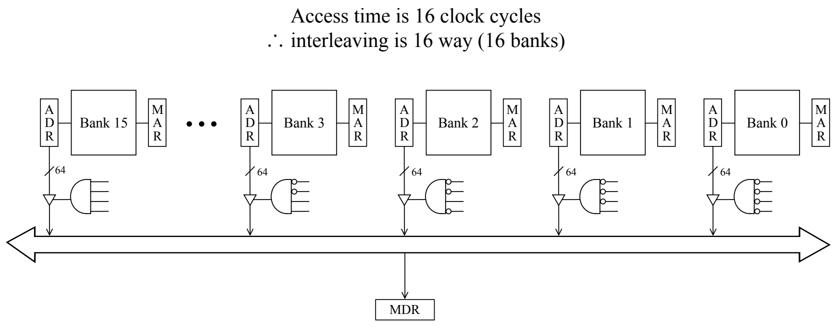

Access time is 16 clock cycles
∴ interleaving is 16 way (16 banks)
ADR
Bank 15
MAR
Bank 3
Bank 2
Bank 1
Bank 0
64
MDR

Figure 6.8: Interleaving. The CRAY I model.

Suppose the vector length was 50. That is, 50 values were to be loaded into the first 50 components of a vector register from memory. If the memory was not interleaved, this would take 50 times 11, 550 clock cycles to perform the load. With 16 way interleaving, each of 50 loads can be initiated in one of the first 50 clock cycles. The 50th value would then take another ten cycles to complete, so the instruction would take 50 + 10 clock cycles to complete, 60 cycles, not 550 cycles.

6.4 DRAM

6.4.1 DRAM, SRAM, and NVRAM

We first turn our attention to the structure of the individual memory cell, DRAM, SRAM, or NVRAM.

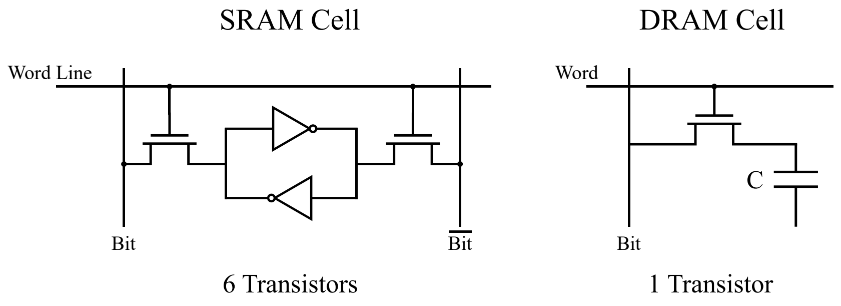


Figure 6.9: DRAM vs SRAM

It is worth noting the differences between the two most popular memory devices, DRAM and SRAM. SRAM stands for Static Random Access Memory. We will see momentarily why that name is appropriate. Figure 6.9 shows the structure of both SRAM and DRAM. To store a single bit of memory, an SRAM cell requires six transistors. The bit itself is stored in a pair of cross-coupled inverters, each requiring two transistors, one p-type and one n-type. The output of each inverter is an input to the gate of the other inverter. The output 1 of one of the inverters applied to the gate of the other inverter causes the output of the other inverter to be 0, which in turn causes the output of the first inverter to be 1. As long as power is supplied to the four-transistor circuit, the cross-coupled inverters will maintain the value stored, either 1 or 0. Ergo, the name Static RAM. The cell requires two more transistors, necessary to read the value stored, or to change the value stored. Two pass transistors serve that purpose, yielding six transistors in all.

The DRAM cell does not store its value in a transistor circuit. Instead the value 1 or 0 is stored as charge in a capacitor, charge = 1 lack of charge = 0. We do need a pass transistor to detect whether the capacitor is storing a 1 or a 0, i.e., whether it is charged or not. This is done by causing the capacitor to discharge which indicates whether a 1 or 0 was stored. This destructive sampling is then corrected by returning the capacitor to its charged or uncharged state. Unfortunately, if left alone, the capacitor will slowly discharge according to an RC time constant, after which we would have no idea whether the capacitor was charged or not. To prevent that from happening, the DRAM circuitry interrupts processing periodically i order to return the charge on the capacitor to full charge or full discharge, thereby maintaining the value of the cell to 1 or 0. This refresh step is necessary. Ergo, the name dynamic RAM.

A third device technology has surfaced over the past few decades, called NVRAM, NV standing for Non-Volatile. Recall we said that an SRAM cell retains its value as long as power was supplied to the cross-couple inverters. Remove that source of energy, and the SRAM cell no longer retains its value. We say SRAM memory is volatile because it requires s power supply to work properly. There are device technologies that are still effective even if we remove the course of power, EPROMs being among the most common. We use the term "persistence" to indicate that the nvRAM cell still retains its value even if the source of power is removed.

Figure 6.10 lists the characteristics of SRAM, DRAM, and NVRAM with respect to latency, Density, Persistence and Refresh. Only NVRAM is persistent, only DRAM requires refresh, DRAM has the highest density (one transistor per memory cell) and longest latency.

	SRAM	DRAM	NVM
Latency	Low	High	Highest
Density	Low	High	Highest
Persistence	No	No	Yes
Refresh	No	Yes	No

Figure 6.10: Characteristics of SRAM, DRAM, NVRAM

6.4.2 The DRAM array

The basic storage unit is the DRAM array. Originally the basic storage unit, a DRAM array, consisted of storage for 2^N bits of information. To identify which one of the 2^N bits of storage we wanted, we organized our 2^N bits into 2^n rows and 2^m columns, where N = n times m. Two registers, an n-bit ROW address register, and an m-bit COLUMN address register contained the address bits for the row and column of the array we wished to access.

To get there, we first loaded the n-bit ROW address register. The on-chip electronics loaded the 2^m column bits comprising the row into a structure called the ROW BUFFER. Associated with each column bit was a sense amplifier, which allowed a column bit to be output from the chip. We say a row buffer is open when it contains the 2^m bits corresponding to a row. The row buffer must be open before we can access the desired bit we want to load, or store the bit we want to store. When the row buffer is open, we can then load or store a bit of information. To do that, we load the COLUMN address register which causes the one bit corresponding to the address in the m-bit column register to be output from the DRAM chip.

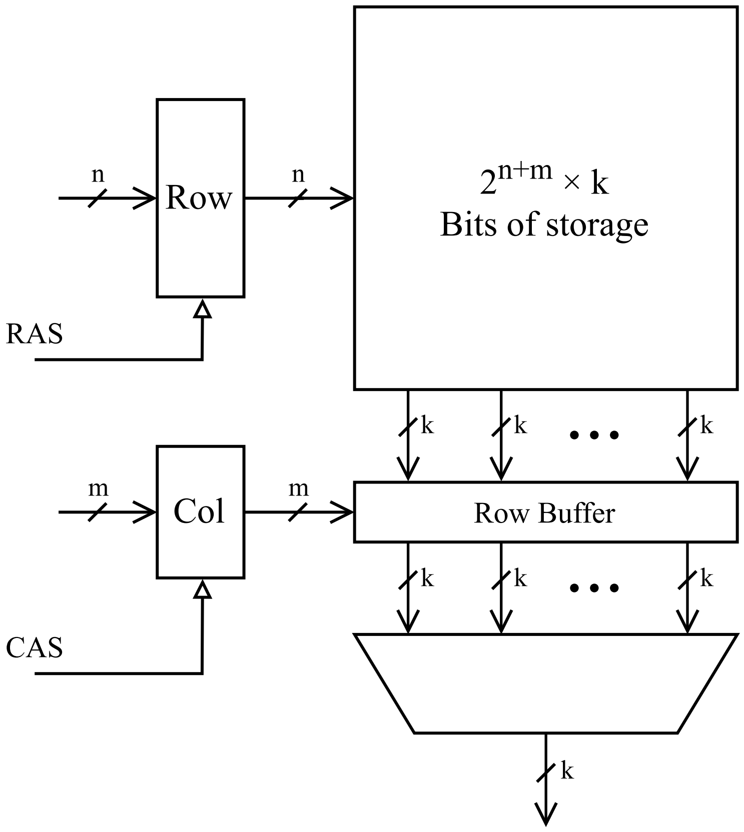


Figure 6.11: The DRAM Array

In the early days, if we wanted to access a byte of storage, we would need 8 chips, each providing one bit of the 8 bits comprising the byte of memory. Over time, device technology shrinked (the fruits of Moore's Law), allowing a single array to provide a full byte of memory.

From there, the continued benefits of Moore's Law enabled us to introduce interleaving to the DRAM chip. Instead of one array on a chip, several arrays have been made possible, each forming a different bank. As has been pointed out, each bank needs its own MAR and internal MDR. In the case of DRAM arrays, the MAR is the combination of the row address register and the column address register. The internal MDR is effectively that part of the row buffer corresponding to the column address register.

Today's DRAM chip has bits to specify what piece of a row, what piece of a column, which bank, and which byte on the bus the chip will be contributing to. A common configuration can have 3 bits to specify the byte on the bus the chip is contributing to, 17 bits of row address, 7 bits of column address, and 4 bits to specify which of 16 banks is being addressed. Together, they provide for 31 GB of DRAM memory. If more memory is needed, one can add one or more channel bits, where each channel specifies a set of pins that are used to access that 31GB of memory.

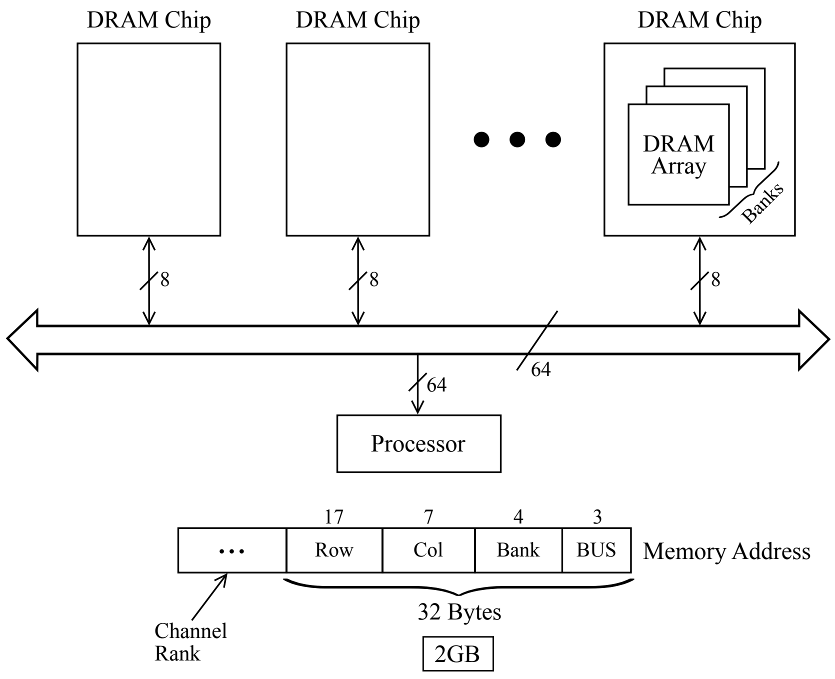


Figure 6.12: Today's DRAM Chip

6.4.3 Page Mode

Finally, before leaving DRAMs, it is important to mention Page mode, a mechanism that can save substantial access time. In order to access a byte of memory from a DRAM, as we have said, we first have to load the row address register which ends up opening the corresponding row buffer. Then the column address register needs to be loaded in order to get at the desired column. What if the address we wish to access has the same row address bits. That is, we wish to access a different column in the same row. Then we can skip the "load row address" part and immediately jump to "load column address," saving a substantial part of the access time. We refer to this as a page mode access.

6.5 The memory controller

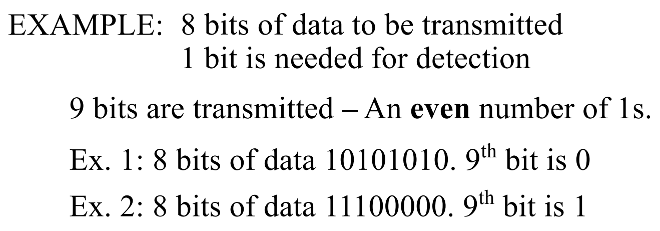
The memory controller keeps track of which memory addresses are required by the processor, and which of those to initiate next. The memory controller sits between the processor which requests the data to be read or written and the DRAMs that will participate in the transfer. To make that decision the memory controller looks at the characteristics of each request. Is the row corresponding to that request in an open row buffer? If not, the row buffer would have to be closed and the correct row put in the row buffer before we could access the required column. Is the request from the processor due to an access that is needed now in order to perform some computation, or is the access due to prefetching, a request that is not needed now, but may be needed in the near future. If the access is due to a desired prefetch request, how likely is it that the prefetch will actually be needed in the near future.

Finally, a system may have more than one memory controller. Each accesses the DRAMs via different sets of pins, and controls the accesses of those DRAMs connected to its set of pins.

6.6 Dealing with transmission errors

Not all memory accesses complete correctly, so memory systems must provide for accesses that do not complete correctly. Three common mechanisms for doing so are (1) checking parity, which will detect a single bit error so that the system can ask that the transmission be repeated, (2) using error correcting codes, which will not only detect a single bit error, but also correct it without having to retransmit, and (3) a checksum mechanism which can detect (but not correct) multiple errors that are not statistically independent.

6.6.1 Parity



EXAMPLE: 8 bits of data to be transmitted
1 bit is needed for detection

9 bits are transmitted – An **even** number of 1s.

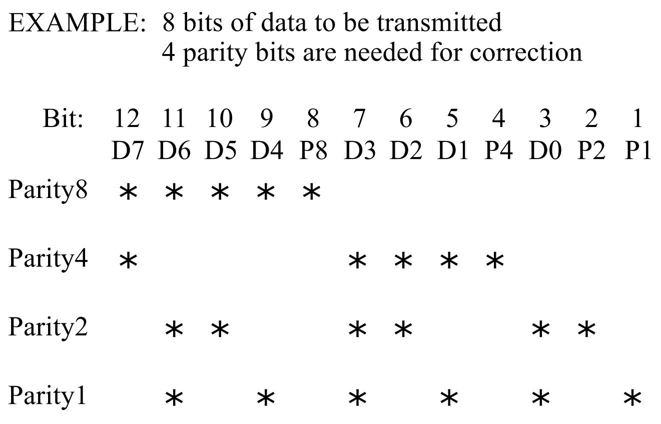
Ex. 1: 8 bits of data 10101010. 9th bit is 0

Ex. 2: 8 bits of data 11100000. 9th bit is 1

Figure 6.13: Parity Detection of errors

The simplest mechanism used to deal with transmission errors is the parity check. The probability of an error in transmission is 10^{-7}. That is one transmission error every 10,000,000 accesses. That seems pretty unlikely at first blush, but given that we may transmit at a rate approaching one every nanosecond, we see that errors do occur frequently. However, in most cases, errors across multiple bits are statistically independent. That means that two bits will be in error in the same transmission is 10^{-14}. Very remote. ...which means we must deal aggressively with single bit errors, and two bit errors not so much. Parity detection says that every transmission will consist of an even number of 1s in the transmission. Thus before we transmit, we count the number of 1's. If it is an odd number, we tack on an extra bit with value 1. If it is an even number, we tack on an extra bit with value 0. At the receiver, we count the number of 1's. If an odd number have the value 1, we must have flipped a bit, so we can detect it and ask it be retransmitted.

6.6.2 Error Correcting Codes



EXAMPLE: 8 bits of data to be transmitted
4 parity bits are needed for correction

Bit:	12	11	10	9	8	7	6	5	4	3	2	1
	D7	D6	D5	D4	P8	D3	D2	D1	P4	D0	P2	P1
Parity8	*	*	*	*	*							
Parity4	*					*	*	*	*			
Parity2		*	*			*	*			*	*	
Parity1		*		*		*		*		*		*

Figure 6.14: Error Correcting Codes to Detect and Correct errors

We can expand our parity detection scheme to not only detect a one-bit error but also to correct it, provided the bit errors are statistically independent as described above. If we wish to transmit N bits of data, we will need to augment these N bits with an additional (1 + log N) bits. Figure 6.14 shows by example how to do this for N=8 bits of data (D7, D6, D5, ...D0) with an additional four bits. The additional four bits (P8, P4, P2, P1) are parity bits that we use to generate four parity functions, each using a subset of the eight data bits.

The data bits selected for each parity function correspond to the bit number of each data bit. For example, D7 is bit 12, in binary 1100, so D7 is included in the parity function for P8 and P4. D6 is bit 11, in binary 1011, so D6 is included in the parity functions for P4, P2, and P1. Thus each data bit is included in a unique subset of the parity functions. Each P bit is set at the transmitter to 0 or 1 so that the total number of 1's in each parity function is even.

At the receiver, if a bit error occurs, the bit causing the problem will cause all parity functions containing that bit to contain an odd number of 1's. The binary bit number corresponding to the parity functions in error will identify the bit that caused the transmission error. Flipping that bit will fix the incorrect transmission.

6.6.3 Checksum (Fig 6.14)

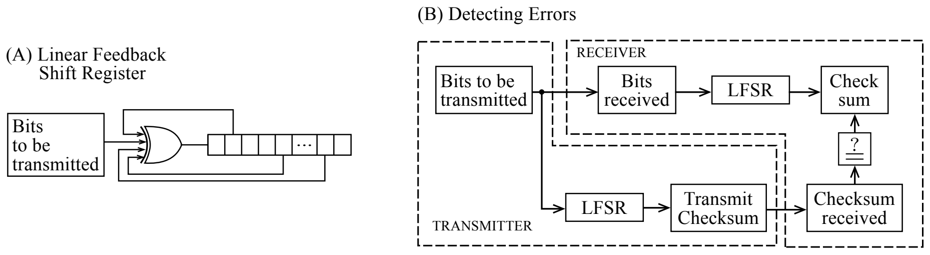


Figure 6.15: Checksum mechanism for Detecting Burst Errors

There are occasions when a problem causes many bits to be transmitted in error, often, for example, the result of a burst error. We can not correct a burst error but at least we can detect it so we can ask for it to be retransmitted. The favored method for detecting burst errors is with a linear feedback shift register since it uses all bits to be transmitted in a comprehensive way.

The method is shown in Figure 6.15. Let A be the bits to be transmitted. A is transmitted sequentially to the XOR gate at the input of the shift register. As each bit is shifted cycle by cycle through the shift register, some bits are fed back to the XOR gate. After the last bit to be transmitted is input to the shift register, the remaining bits in the shift register are recorded. They are referred to as the checksum.

The transmitter does two things with the bits of A. It transmits A to the receiver and it inputs A to the shift register, creating a checksum which it also sends to the receiver. The receiver receives A and the checksum produced at the transmitter. The receiver also inputs A to the shift register creating another copy of the checksum. The receiver compares the two checksums, the one produced by the transmitter and the one produced by the receiver. If they are identical, the transmission completed correctly. If not, then an error occurred and the receiver requests a retransmission.