4. Addressing modes

The topic of this chapter are the addressing modes, the different ways the address of an operand in memory is specified and calculated. Although the computer's world offers a large variety of addressing modes, we will discuss only about the basic ones, those that are used the heaviest in programs.

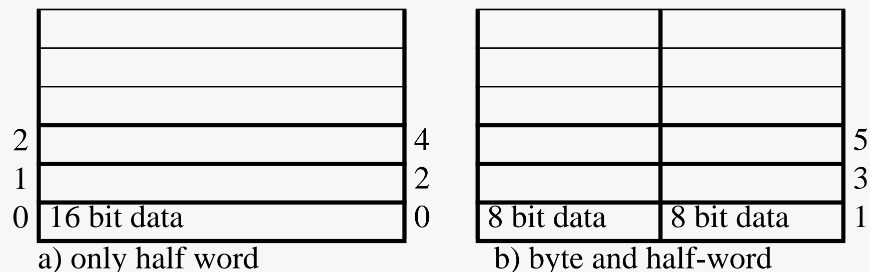
We begin with a discussion about memory addressing: our special concern in this section is how is data (especially if we allow different data sizes) retrieved from memory and what problems appear.

4.1. Interpreting memory addresses

A major decision a designer faces is how is the memory to be addressed.

Example 4.1 MEMORY ACCESS:

You have to design a 16 bit architecture: show how you can access the memory if you allow byte and half-word addressing or only half-word.
Answer:



a) only half word b) byte and half-word

If we have a n-bit wide architecture, and we always read/write from/to memory only n-bit wide data (in our case 16-bit architecture and read/write only half-words as in example 4.1a) then there are not very many hardware problems; an address always means a half-word. However, a problem appears when we try to manipulate characters (a character nicely fits a byte):

- use a whole half-word for one character. In this case we waste a byte for every char;

- use a half-word to accommodate two bytes (packed). In this case we must write code to pack and unpack chars; access to memory is very simple, but the speed will be affected by the extra software manipulation.

In the case we should have a **byte-addressable** memory, as in (4.1b), then each address will specify a **byte** in the memory, and it is a matter of organization if we want to have two banks of memory, 8 bit each, or only one, 16 bit wide, from which we select the proper byte.

The problem of multiple data representation is unavoidable in a general-purpose computer:

- characters are represented as bytes

- integers require at least 16 bits:

  - **16** bits for **short**
  - 16 or **32** for **int**
  - 32 or **64** for **long int**

- floats are represented with at least 32 bits (IEEE 754):

  - 32 bits for single precision;
  - 64 bits for double precision;
  - 80 bits for extended precision.

The above list doesn't include other data types like pointers, etc.

Efficient access to data of various sizes is so important that modern machines (32 bit machines) provide access to byte (8 bits), half-word (16 bits), word (32 bits), and some provide also access to double-words (64 bits).

Accessing data of various sizes rises some problems in memory organization. If the memory organization only allows byte-access, as in Figure 4.1, then one byte requires one memory access, a half-word (2

bytes) requires two memory accesses and a word (4 bytes) requires four memory accesses.

An example of such an organization was the IBM PC/XT which used the 8088 microprocessor 16 bit architecture (like 8086), with the difference that it handles byte instead of half-words through the external data bus.

I/O	B	L/H	Operation	Comment
0	0	0	DB15-DB0 → D15-D0	write half word
0	0	1	DB15-DB0 → D15-D0	write half word
0	1	0	DB7-DB0 → D7-D0	write low byte
0	1	1	DB7-DB0 → D15-D0	write high byte
1	0	0	D15-D0 → DB15-DB0	read half word
1	0	1	D15-D0 → DB15-DB0	read half word
1	1	0	D7-D0 → DB7-DB0;DB15-DB8=0	read low byte
1	1	1	D15-D8 → DB7-DB0;DB15-DB8=0	read high byte

Table 4.1 The truth table for the alignment network in Figure 4.3.

A memory organization that supports both byte accesses and half-words, as in Figure 4.2, must be provided with an extra control signal, B/H in our case which indicates if a byte is to be read/written (B/H = 1) or a half-word (B/H = 0). The internal organization of such a memory module could look like in Figure 4.3. As it can be seen the access to memory (to write) or from memory is done through an **alignment network** which is a combinatorial circuit described by the Table 4.1.

- the I/O input controls if data is flowing from the data bus to memory (I/O = 0) or from memory to data bus (I/O = 1)

- the B input controls the data to be transferred: byte (B = 1) or half-word (B = 0)

- the I/H input controls if the byte to be transferred is the low (I/H = 0) or the high one (I/H = 1), i.e. from memory bank 1.

There is also another combinatorial circuit (denoted by CLC_0 in Figure 4.3) that controls which bank is selected during a byte write. With the memory organization in Figure 4.3 a byte written at an even address will end up in memory bank 0, while bytes at odd addresses reside in memory bank 1.

Reading/writing 16 bit data is more complicated. If the 16 bit data starts at an even address like in Figure 4.4.a, then there is no problem at all to read/writer data from/into the memory presented in Figure 4.3: the low bit will be read/written from/into memory bank 0 and the high byte (that which is

at a higher address) will be located in memory bank 1. The address bits AD_{n-1} - AD_1 are the same for the two bytes; hence a single access suffices to get both bytes.

When the 16 bit data starts at an odd address (like in Figure 4.4.b), then **two accesses** are necessary to read/write it: the addresses of the low and high bytes differ in more than the least significant bit (AD0) so that the memory banks must get different addresses to retrieve the proper data.

Example 4.2 DATA AND ADDRESS:

Suppose you have the memory from Figure 4.3. and the data configuration in Figure 4.4.b. What data will be read if the address is 15 and B/H = 0? But if the address is 16?

Answer:
When the address is 15 and B/H = 0 the read data will be the half word starting at address 14. As can be seen in figure 4.3, during a read, the least significant bit of the address, AD0 is used only to select the proper byte if B/H=1. The two banks receive as address the binary configuration 0..00111, which are the bits ADn-1-AD1 of the address.

When the address is 16 and B/H=0 then the read data will be the half word starting at address 16. Both memory banks receive the same binary configuration as address 0..01000

By now it should be clear why many systems require data to be aligned: in our case 16 bit values must be stored at even addresses. If data is aligned in memory then it is possible to access that data with a simple memory design and without wasting time.

Note that the alignment network in Figure 4.3 increases the response time of the memory (accesses to memory take longer). Instead of using an alignment network some machines prefer to always read whole words (for 32 bit machines) and then to shift data inside the CPU in the cases alignment is needed; nonword accesses are slower but the common case (word access) is not affected and can be optimized to work at maximum speed. As we shall see in next chapter the alignment problem is really critical in the case an alignment network must be placed between the cache and the CPU.

More generally, if a data object needs up to 2^N bytes to be represented, then aligning it means placing it in the memory starting with an address that has the least significant N bits zero. Another problem related to memory addressing is the byte ordering inside a data that is longer than a byte (half word, word, etc.).

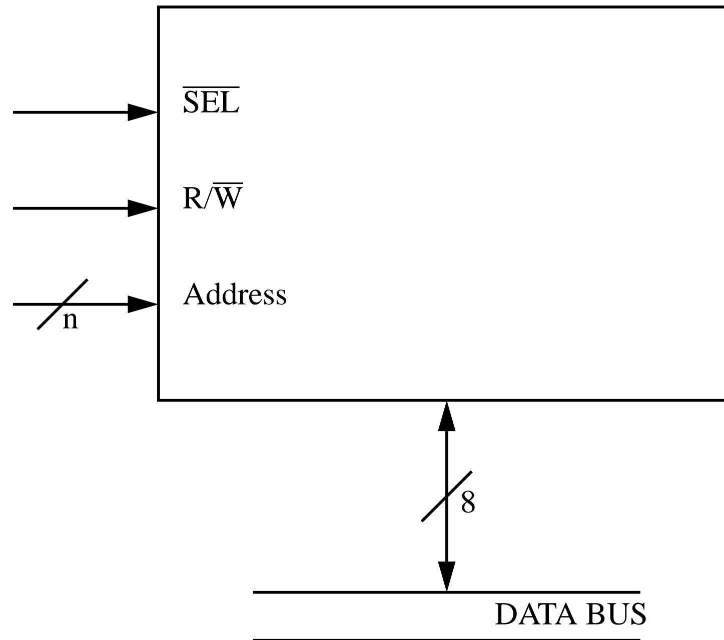
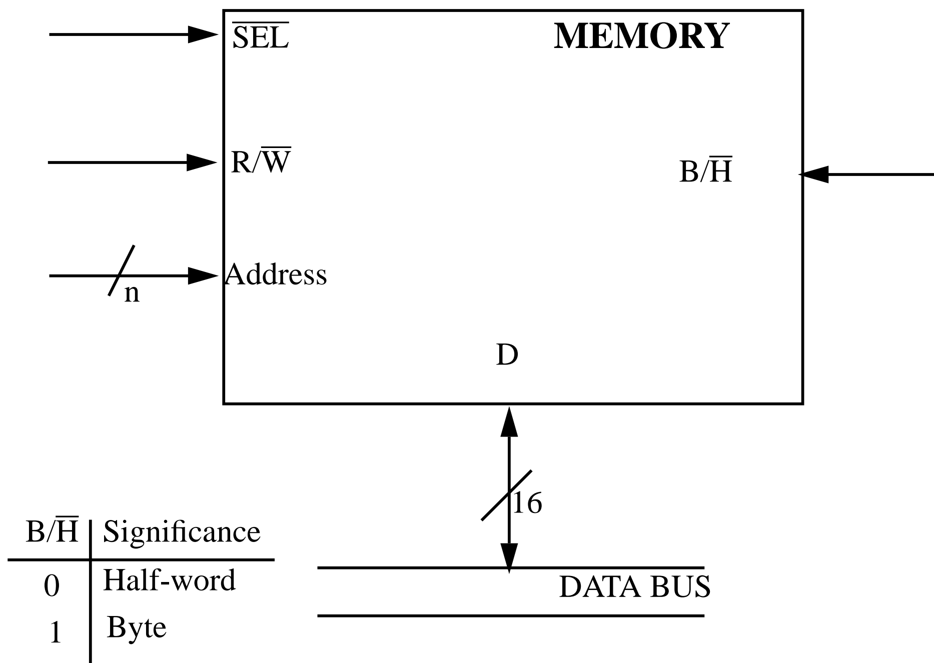


FIGURE 4.1 A byte-organized memory



B/$\overline{H}$	Significance
0	Half-word
1	Byte

FIGURE 4.2 A memory that can be byte and half-word accessed.

Example 4.3 LABELING:

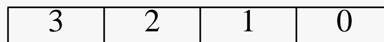
Label the following boxes:

a) as an array of four elements;
b) as the bits of a four bit binary number.

Answer:
There are to ways to do so and even if the problem seem to be stupid, there is a lot of debate around.

a) People tend to label from left to right probably because this is the way we write, from left to right; this is also the way many of us label the elements of an array:

b) In this case the tendency is to label boxes from right to left, as we associate each box with a power of two:



The inconsistency in labeling, pointed out by the previous example is at the heart of many problems, as the following example shows.

Example 4.4 PROBLEMS DUE TO INCONSISTENCY IN LABELING:

A word (32 bit) is stored at address 0 in a memory that is byte addressable. Which byte of the word is stored at address 0?

Answer:
Depends how we label the memory's bytes. We all agree that the most significant part of a data object is at the left and the least significant is at the right, as it happens with the numbers that have the most significant digit at the left most side. If we use a left to right numbering of memory addresses, then we have:

In this case the most significant byte of the word (Byte3) is stored at address 0, while the least significant byte in word (Byte0) is stored at address 3. When we use a right to left numbering of memory addresses then we have:

In this case the least significant byte of the address is stored at address zero.

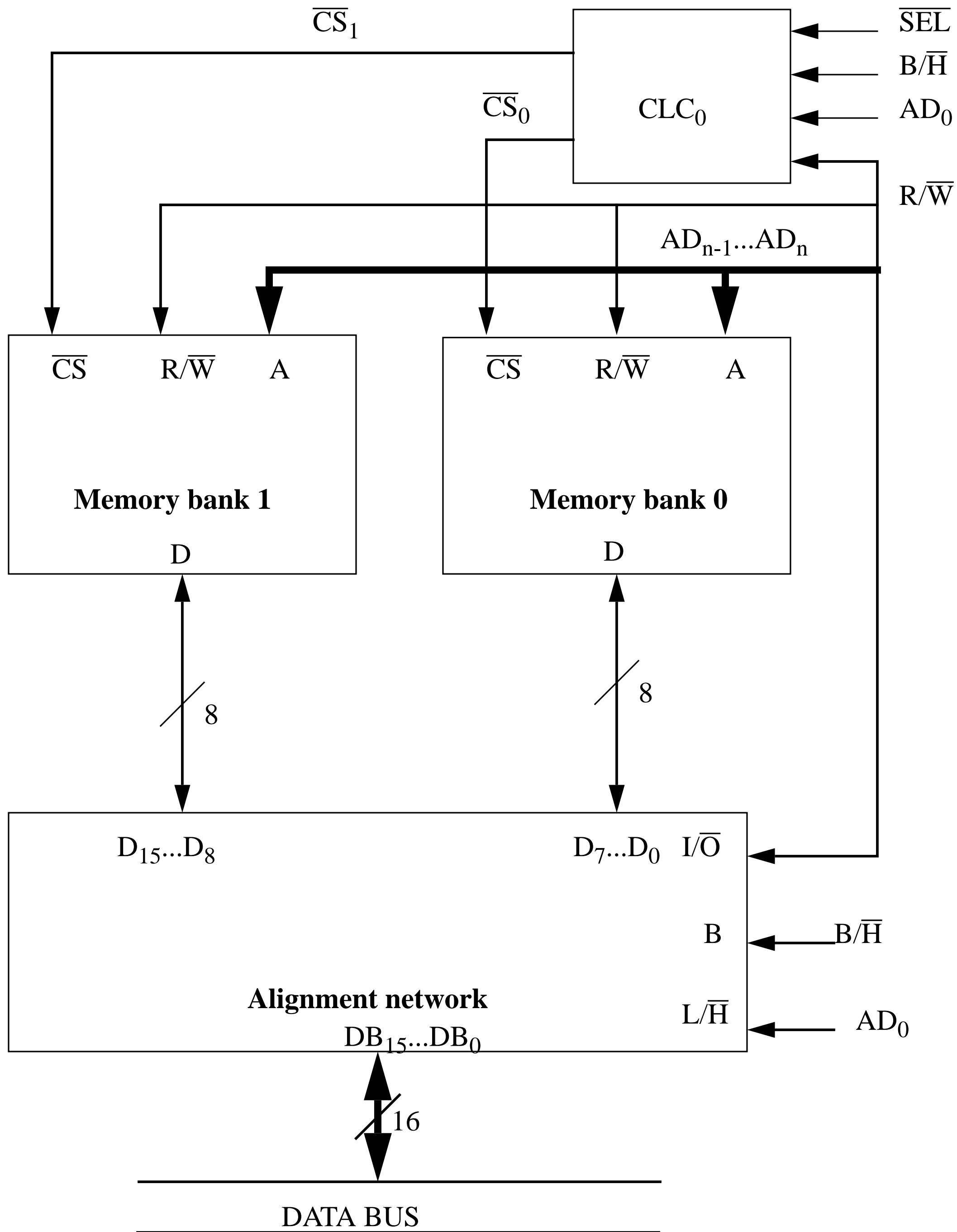


FIGURE 4.3 Internal organization of the memory in figure 4.2

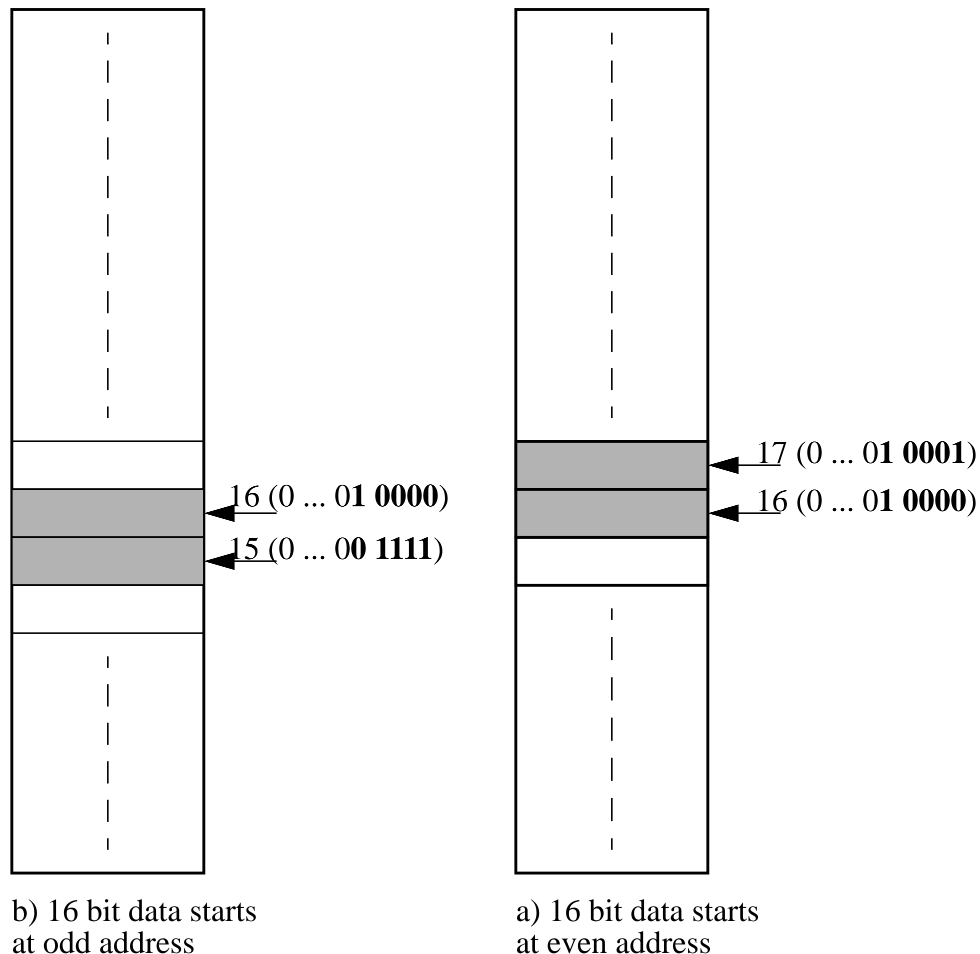


FIGURE 4.4 Aligned and not aligned 16 bit data.

The first convention presented in example 4.4 is called **Big Endian** (most significant byte stored at the lowest address in the word); the second convention presented is called **Little Endian** (the most significant byte of the object is stored at the highest address for that object).

Is there any of the two approaches better than the other? Unfortunately no, and this is the reason a debate between the advocates of two possibilities has been around for many time. The names *Big Endian* and *Little Endian* come from a paper of Cohen[1981], which draws an analogy between the arguments over how to number bytes and each end of the egg should be broken as described in *Gulliver's Travels*.

As long as you have to deal only with one machine there is no problem in adopting its convention.; however if you have to deal with machines using different conventions or you have to transfer data between machines, then the problem is no longer so simple and it remembers somehow the troubles in transferring files from the IBM machines that use the EBCDIC code for characters, to other machines that use the ASCII representation for characters.

We face a similar problem, though causing less trouble, when discussing about bit numbering (labeling) inside a binary number. If we label the bits in this way:

$^{MSB}b_0 b_1 ... b_{n-1}{}^{LSB}$

then we use the Big Endian convention (the most significant bit is labeled with the index 0). On the other hand, if we label the bits in the following way:

$^{MSB}b_{n-1} b_{n-2} ... b_0{}^{LSB}$

then we are using the Little Endian convention, which happens to be very easy to use in this case because labels (indexes) represent the powers of two the bits are multiplying when we convert an unsigned binary to decimal.

Depending upon the convention used for byte and bit ordering machines can be classified as in the table below

Name	Byte Ordering	Bit Ordering	Examples
Consistent LE	LE	LE	Intel 80x86, DEC VAX
Consistent BE	BE	BE	TI9900, IBM-360/370
Inconsistent LE	LE	BE	-
Inconsistent BE	BE	LE	Motorola 68000

Many new CPUs allow the user to use either Little Endian or Big Endian addressing convention: MIPS R2000 and Intel i860 are only two examples.

4.2. A classification of addressing modes

We can now determine, given an address, what byte(s) in memory correspond to that address. In the rest of this chapter, we'll be discussing about addressing-modes, in other words, we'll discuss about how the architectures specify the address of an object in the memory.

In general-purpose-register (GPR) machines, the ones we are concerned at most, an addressing mode may specify:

- a constant;
- a register;
- a memory location.

Before considering in some detail the basic addressing modes, let us recall that we have already discussed about this topic (even though not in terms of addressing modes) when we approached the n-address machine subject.

Using examples, we saw that the operand can be in a register (we call this to be the **register addressing mode**) or fully included in the instruction (we call this **direct addressing mode**), and we tried to figure out the impact over the code length and execution speed. We also discussed about the **PC-relative addressing** and the use of displacements.

The most used addressing modes are presented below; a left arrow means assignment, and M stands for memory. We use an array notation for memory because we can view the memory as an array of bytes (or half-words or words whichever you prefer, but the significance of the notation must be very clear).

Register

ex: `ADD r1, r2, r3`
means: `r1 ← r2 + r3`
comment:used when a value is in a register.

Immediate (or literal)

ex: `ADD r1, r2, 1`
means: `r1 ← r2 + 1`
comment:used when a constant is needed.

Direct
ex: `ADD r1, r2, (100)`
means: `r1 ← r2 + M[100]`
comment: used to access static data; the address of the operand is include in the instruction; space must be provided to accommodate a whole address.

Register indirect (or register deferred)

ex: `ADD r1, r2, (r3)`
means: `r1 ← r2 + M[r3]`
comment: the register (r3 in this example) contains the address of a memory location.

Displacement

ex: `ADD r1, r2, 100(r3)`
means: `r1 ◄— r2 + M[r3 + 100]`
comment: the address is the sum of the content of the register (the base) and a constant from the instruction (the displacement); used to access local variables on a stack or data structures. If the displacement is zero, then it is the same as register indirect.

Memory indirect (or memory deferred)

ex: `ADD r1, r2, @r3`
means: `r1 ◄— r2 + M[M[r3]]`
comment: used in pointer addressing; if r3 contains the address of a pointer p, then M[M[r3]] yields *p.

Indexed

ex: `ADD r1, r2, (r3)[r4]`
means: `r1 ◄— r2 + M[r3 + size*r4]`
comment: two registers are added to get a memory address, used in array addressing with one register the base address of the array and the other one the offset from the base to the desired element in the array.

Autoincrement

ex: `ADD r1, r2,(r3)+`
means: `r1 ◄— r2 + M[r3]`
`r3 ◄— r3 + s`
comment: used to step through arrays, the first time it is used r3 points to the beginning of the array; each access increments r2 with the size s of an array's element (s = 1 for byte, s = 2 for half-word, etc.)

Autodecrement

ex: `ADD r1, r2, -(r3)`
means: `r3 ◄— r3 - s`
`r1 ◄— r2 + M[r3]`
comment: can be used like autoincrement, but to step through arrays in reverse order. Together with the autoincrement mode it can be used to implement a stack.

This list is far from being complete and you may wonder:

- why so many addressing modes?
- which are the minimum necessary addressing modes?

The answer to the first question is that addressing modes significantly reduce instruction count of programs (keep in mind that this comes with the increased hardware complexity). As for the second question the answer depends upon the architecture; for a load-store architecture **register**, **immediate**, and **register deferred** are sufficient. From this three we can create the equivalent of other addressing modes although somehow clumsily.

Example 4.5 ADDRESSING MODES:

Rewrite the following sequence of code using the register and register deferred addressing modes:

```
/* a sequence that does nothing */
LOAD r1, (200)
ADD  r3, r2, 10(r1)
AND  r3, r2, @r2
...
...
...
```

Answer:

```
LOAD r1, 200# immediate;
LOAD r1, (r1)# register deferred
ADD  r4, r2, 10# these two instructions replace the
ADD  r3, r2, r4# ADD in the above sequence
LOAD r4, (r2)# and the two simulate the memory
AND  r3, r2,(r4)# deferred addressing;
```

Observe that the IC is greater, and we need more registers; r4 is used to calculate the displacement from r1 because r1 has not been destroyed; the same remark holds for r2.

4.3 Immediate addressing:

Immediate values are used in:

- arithmetic operations:

```
ADD r1, 1        # increment register r
```

- comparisons, mostly in branches (although it depends upon the branching method the architecture is using):

```
BEQ addr, r1, 2 #branch to addr if contents of r1 is 2;
```

- moves (immediate loads) to bring a constant into a register:

```
LOAD r1, 3
```

- In this case the immediate can be:

- a constant written in the code (as in A + B + 4). The values are usually small
- an address constant which can be large.

Again there is a tradeoff in deciding the maximum size immediate operands should have in the instruction set:

- if the instruction size is fixed, a word (32 bit) for instance, then we must compromise between the space we allocate for immediate operands and the space we have to use for other fields (opcode, registers, etc.)

- if the instruction size is variable, then an immediate could easily accommodate an address.

4.4 Displacement Addressing:

Displacements are used in two ways:

- to access data in memory. The most common examples are accessing the local variables in an activation record (AR), and accessing the fields of a structure or record;

- to compute the target address for branches, jumps, calls in PC-relative addressing.

Measurements show a wide distribution of displacements for the first category, while for the second one most displacements are short (4 - 8 bits), mainly due to the fact that usually the target of a branch (jump) is close to the actual instruction. The tradeoffs in deciding the length of the displacement are similar to those in deciding the length of immediate operands.

Example 4.6 ADDRESSING MODES:

An array of two integers (an integer is 32 bits wide) is placed in memory starting with address 100. Show how to increment the elements of the array using displacement addressing. The memory is byte addressable.

Answer:

```
LOAD  r1, 100# in r1 base
LOAD  r2, 0(r1)# load the first element of array
ADD   r2, r2, 1
STORE 0(r1), r2
LOADr2, 4(r1)# second word starts at 104
ADDr2, r2, 1
STORE 4(r1),r2
```

4.5 Indirect addressing (memory indirect)

In indirect addressing an address is considered to be the address of an address, rather than the address of a value. Suppose we have the following declarations in C:

```
int   num,  *ptr;
```

which declares num to be of type integer and ptr to be of type pointer to integer. The following statement:

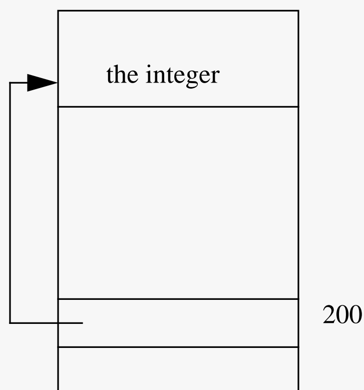
```
ptr    =   &num;
```

assigns the address of num to ptr. The logical structure created by the above assignment is:

Example 4.7 ADDRESSING MODES:

An integer is stored somewhere in the memory; a pointer to this integer is at address 200. Use memory indirect addressing to increment the number.

Answer:
The relation between the pointer and the number is shown below:



If the machine supports the base address (200) to be in memory we have:

```
LOAD  r2, 1
ADD   r2, r2, @(200)
STORE@(200), r2
```

If the base address is in a register, as most of the machines require (machines that support this addressing mode):

```
LOAD  r1, 200
LOAD  r2, 1
ADD   r2, r2, @(r1)
STORE@(r1), r2
```

This addressing mode is at the base of all virtual memory systems. It allows to relocate programs without changing addresses in programs that use them. The problem is that if we want to accept instructions that take much more time to execute than their counterparts using simpler addressing modes.

Example 4.8 MEMORY ACCESS:

How many memory accesses require the following instructions?

```
ADD r1, r2, r3
ADD r1, r2, (r3)
ADD r1, r2, @r3
```

Suppose every instruction is one word long, as well as every address.

Answer:

`ADD r1, r2, r3`
require only one memory access, reading the instruction;
`ADD r1, r2, (r3)`
require two memory accesses, the first to read the instruction and the other one to read the value from memory location(s), whose address is in r3;
`ADD r1, r2, @(r3)`
three memory accesses are made in this case, the first to read the instruction, the second to get M[r3], and the third one to get M[M[r3]].

While the implementation of this addressing mode has nothing difficult per se, it raises some problems when we try to realize an **efficient implementation** of the CPU: efficient pipelining require all instructions to complete in the same number of clock cycles, which is difficult if we allow instructions with different running times. Sure we could make all instructions execute in the same number of clock cycles as the longest running instruction, by simply inserting idle clock cycles. What an idea! It’s wrong because it happens that the longest running instructions **are not** the common cases. And it is precisely the common case that we want to optimize.

4.6 Indexed addressing

Indexed addressing is provided in many machines for the convenience of accessing arrays.

Example 4.9 ADDRESSING MODES

An array of two integers (each integer = 32 bits) is placed in memory starting with address 100. Show how to increment the elements of the array using indexed addressing.

Answer:

```
LOAD  r1, 100     # in r1 base;
LOAD  r2, 0       # the first index in array;
LOAD  r3, 1
ADD   r4, r3, (r1)[r2]
STORE (r1)[r2]
ADD   r2, r2, 1   #index = index + 1
ADD   r4, r3, (r1)[r2]
STORE (r1)[r2]
```

What is the price we have to pay to include this addressing mode in the instruction set? Again we take into account how the addressing mode effects the instruction length and the execution time as compared with other addressing modes. We shall consider the instruction encoding and the execution time as the parameters for this discussion.

Instruction encoding

For a three address machine (our examples so far are for such a machine) we need four fields to encode the registers to be used:

- destination
- first source operand
- base
- index

For a 32 bit machine with fixed instruction size (one word), and a set of 32 registers, the necessary fields to encode the four registers require 4*5 = 20 bits. If we allow 6 bits for the opcode, then we still have 32 - (20+6) = 6 bits in the instruction. Do we need them? Of course we need them because we have to encode two more things in the instruction:

- what kind of addressing mode are we using;
- what is the dimension *size* with which the index has to be multiplied.

As long as we are concerned with the first problem we face yet another

problem, we didn't discuss so far:

- do we allow all possible addressing modes we choose for our instruction set apply for any operand?

  If the answer is *yes,* then we have to specify, besides the addressing mode, to which operand does this addressing mode apply (we discuss about a register-memory machine, not about a memory-memory machine, which allows **all** operands to have their own addressing mode, an even more complicated problem). For a three address machine, this requires two more bits in the instruction.

  In the case of a *no* answer, we have to decide to which operand the addressing mode applies (in our examples, it was second source operand); in this case we do not need extra encoding space, because the operand to which the addressing mode applies is *implicitly* specified, and thus can be hardwired.

To conclude this, we have to encode the addressing mode(s), and possibly to which operand(s) it applies.

When we finally come to the second problem, specifying the size, we realize that we need two bits because arrays can be formed with the following data sizes:

- byte;
- half-word;
- word;
- double-word.

Certainly these two bits are needed, it would be much too restrictive not to allow certain arrays of the basic data types. By now the following bits are available:

$$32 - (20 + 6) - 2 = 4 \text{ bits}$$

four bits with which we can try to encode the addressing mode and, possibly, the operand to which it applies. A possible solution could be to use two bits for encoding the addressing mode, thus allowing four addressing modes, and the other two to specify to which of the three operands applies. Should we allow more addressing modes, so we restrict the number of operands to which that mode applies.

A final note: if we decide on fixed size instructions, then we have to drop some addressing modes; for instance, the **direct** mode which requires a whole absolute address to be provided in the instruction (unless the instruction is wider than an address).

Execution time

Executing an instruction which allows one operand to be accessed using indexing means that we have to compute the address as follows:

address = base_address + size * index

One integer multiplication and one integer addition have to be performed; while the addition is not a problem, we have the ALU, the multiplication requires special hardware, which can be included in the CPU, and this hardware must be very fast. When we say hardware support we do not necessarily mean a multiplier; the reason for this is that multiplication by one (byte), two (half-word), four (word), eight (double-word) can be easily done using left shifting by zero, one, two, and three positions respectively.

Example 4.10 SHIFT OPERATIONS:

A 16 bit register contains the binary configuration:

0000 0000 0001 0010

that is the register contains 18_{10} in an unsigned representation. Show the register's content after left shifting by one, two, and three positions respectively, and the decimal representation of the number.

Answer:

one bit left shift:0000 0000 0010 0100
is the binary representation of 36:
36 = 18 * 2

two bits left shift:0000 0000 0100 1000
is the binary representation of 72:
72 = 18 * 4

three bits left shift:0000 0000 1001 0000
is the binary representation of 144:
144 = 18 * 8

While the shift hardware is simpler than that required for a full integer multiplier, it still is an expensive resource (in terms of silicon area), and will probably be introduced in the CPU only if the shift operation is a must. Note that though most CPUs introduced in the recent years have shifters as a part of the CPU, shift instructions are common in the instruction sets.

Computing an address of an operand using the addressing mode requires at least two clock-cycles: one for multiplication (or shift) and another one for addition. We should consider the adoption of this addressing mode carefully, especially if we consider having instructions that execute in the

same number of clock-cycles.

Obviously this addressing mode is more expensive in clock-cycles than a simpler addressing mode, like register deferred; indexed addressing can be simulated using simpler addressing modes as the example 4.11 shows:

Example 4.11 ADDRESSING MODES

An array of two integers (each integer = 32 bits) is placed in memory starting with address 100. Show how to increment each element of the array using register deferred addressing mode.

Answer:

```
LOAD  r1, 100# the base
LOAD  r2, 1 # 1 will be used for increment
ADD   r3, r2, (r1)
STORE (r1), r3
ADD   r1, r1, 4# the next array element is at 104
ADD   r3, r2, (r1)
STORE (r1), r3
```

Compare this example with example 4.9.

We conclude this session by underlining again the basic design principle: **make the common case fast**. If this addressing mode is not very often used (and indeed it is not, at least in common programs), it is probably wise to dispose it, and try to optimize the common case(s) (which seems to be immediate, displacement, register deferred).

Exercises

4.1 Design a memory module with an 1 Mx16 organization, to be accessed by half word, using 1Mx1 DRAM.

4.2 Design a memory module with an 1Mx32 organization, to be accessed by byte, half word and word. You will need two control lines to specify the type of access:

B1	B0	Significance
0	0	not used
0	1	byte
1	0	half word
1	1	word

4.3 Design in detail the alignment described in Figure 4.3.