Addressing Modes

Clearly the purpose of computing is to transform data – this implies that we need to reference *sources* of data and be able to send transformed data to some specified *destination*. There are basically three ways of referencing the source or destination of data:

- Register
- Immediate
- Memory

1 Register Addressing

Accessing data in registers is fast. For example, the very common (assembler) instruction:

```
MOV A,B
```

will move the contents of register `A` to register `B`, or in symbolic terms, $A \rightarrow B$. In fact "move" is a poor naming convention (yet again!) since the contents of `A` are copied to register `B`, the original contents of which are over-written. But the contents of `A` remain unchanged by this operation. So "copy" would be a better name for this instruction.

Some times the destination register is implicit, as in:

```
ADD B
```

As we saw in discussing CPUs, the second operand would typically come from the `A` register and the result would be placed back in the `A` register.

2 Immediate Addressing

This would be used for the incorporation constants in a program. For example:

```
LI A,42
```

would "load immediate" (`LI`) the value 42 into the `A` register. As we have seen, the value of 42 would comprise the operand of the instruction and would be placed in memory immediately after the op code for "load immediate value into register `A`".

3 Memory Addressing

Here, the operand which follows the op code in program memory would lead to the address (in data memory) of the datum - either the source or the destination.

There are actually four sub-modes of memory addressing:

- Direct memory addressing
- Indirect memory addressing
- Based memory addressing
- Register indirect addressing

3.1 Direct Memory Addressing

Here the instructions operand directly specifies the address of the datum to be accessed. Notice that to execute such an instruction, the processor would need to read the operand into its address register (AR) and then read or write the actual memory location. Hence it would require *two* memory accesses per piece of data.

Where would such a programming facility be used? When a C compiler encounters a variable declaration it would add the name of the symbol (“`x`”) together with the allocated address of that variable to a *symbol table*. Whenever the compiler encounters an instance of the symbol in the program, it will replace the symbol with the *address* of that symbol. Hence the program symbol will be stored in data memory.

3.2 Indirect Memory Addressing

Initially this looks like direct addressing in that the op code is followed in memory by an address. In direct memory addressing, this is the address of the variable. In indirect addressing this operand is the address of the address of the datum, hence the name, “indirect”. Diagramatically, in Figure 1 :

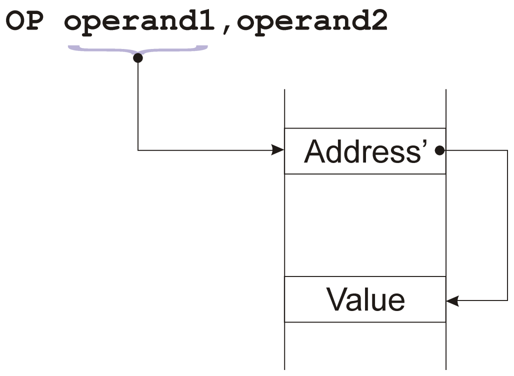

Figure 1: Illustration of indirect memory addressing

Notice accessing the final value takes *three* memory accesses here: One to get the operand, one to get Address’ and place it in the Address Register (AR), and one final memory access to get the datum in question.

What indirect addressing facilitates is pointers (in C). A pointer, recall, is just another variable although it actually contains the *address* of another quantity rather than a quantity itself. Pointers turn-out to be very valuable, particularly for data objects whose size is not known in advance.

3.3 Based Memory Addressing

Based memory addressing is typically used for addressing the elements of an array. Here the operand is an offset from the first array element and the address of the required element is obtained by adding the operand (the offset) to the contents of a special purpose *base register* in the CPU (which, of course holds the address of the first element).

3.4 Register Indirect Addressing

In the final addressing mode of this memory family is register indirect addressing. Here the address of the variable of interest is held in a register, often a special register of double length to accommodate, say, a 16-bit address of an 8-bit microprocessor.

Since the address is held in a register, access to it is very fast. (Compare this to direct memory addressing above where the address has to be fetched from external program memory).

In practice, ‘real’ processors often have proprietary addressing modes that are minor variants of the above.