FIT1047 - Week 4

Memory and Indirect Addressing

The topics for this week are related to how the CPU interacts with the two main components outside the CPU: memory and input/output devices.

First of all, we'll take a look at **memory**.

Overview

- Memory organisation
  - how to address different locations
- Instructions for accessing memory
  - indirect addressing
  - subroutines

Memory

As a computer user, you're probably aware that e.g. your laptop has a certain amount of main memory (RAM), such as 4GB. Perhaps you have opened a tool like the Windows Task Manager, the Mac OS Activity Monitor, or the Gnome System Monitor on Linux, which show you how much memory each application is currently using.

But from a user's point of view, memory stays quite opaque. Let's take a look inside.

From the programmer's and system architect's point of view, memory can be characterised as follows:

- A sequence of *locations*
- Each location has an *address*
  - an unsigned integer, starting from 0
- Each location stores one *value*
  - each value has a fixed *width* (number of bits)
- We can *read* and *change* the value stored at a location

What does it store?

This is a representation of a small section of memory in table form. It shows the first six memory locations.

Address	Hex Value	Integer	Bit pattern	Instruction
000	1004	4100	0001000000010100	Load 004
001	3005	12293	0011000000000101	Add 005
002	2006	8198	0010000000000110	Store 006
003	7000	28672	0111000000000000	Halt
004	008E	142	0000000010001110	JnS 08E
005	0D80	3456	0000110110000000	JnS D80
006	0000	0	0000000000000000	JnS 000

The first column shows the *address* of that location. The following columns show different *interpretations* of the current contents of the memory cell at that address. E.g., address 003 holds the value 7000 when interpreted as a hexadecimal number, or 28672 as a decimal number. But we can also interpret it as a MARIE instruction, in which case it means *Halt*. After all, the memory at address 003 simply stores the bits 0111000000000000, and they can mean different things depending on the context that they are used in.

Addressing

An address is simply an unsigned integer that references one unique memory location. Addresses are usually consecutive numbers starting at 0 and ranging up to the number of distinct memory locations (minus one).

In most architectures,

one memory location stores one byte.

Consequently, each location, each byte, has its own address.

This is called **byte-addressable**.

In MARIE,

one memory location stores one word (two bytes, or 16 bits).

An address therefore references a whole word.

Let's now compute the *number* of addresses we need in order to use a certain size memory. For n locations, we clearly need n different, consecutive unsigned integer numbers. And in order to *represent* the numbers $0 \ldots n-1$, we need to use binary numbers of length $\log_2 n$.

In most cases we're dealing with memory whose size is a power of 2, e.g. 2 gigabyte is 2×2^{30} bytes. In a byte-addressable architecture, we therefore need $2 \times 2^{30} = 2^{31}$ different addresses, ranging from $0 \ldots 2^{31} - 1$. Using powers of two makes it really easy to compute the number of bits needed: $\log_2 2^{31} = 31$.

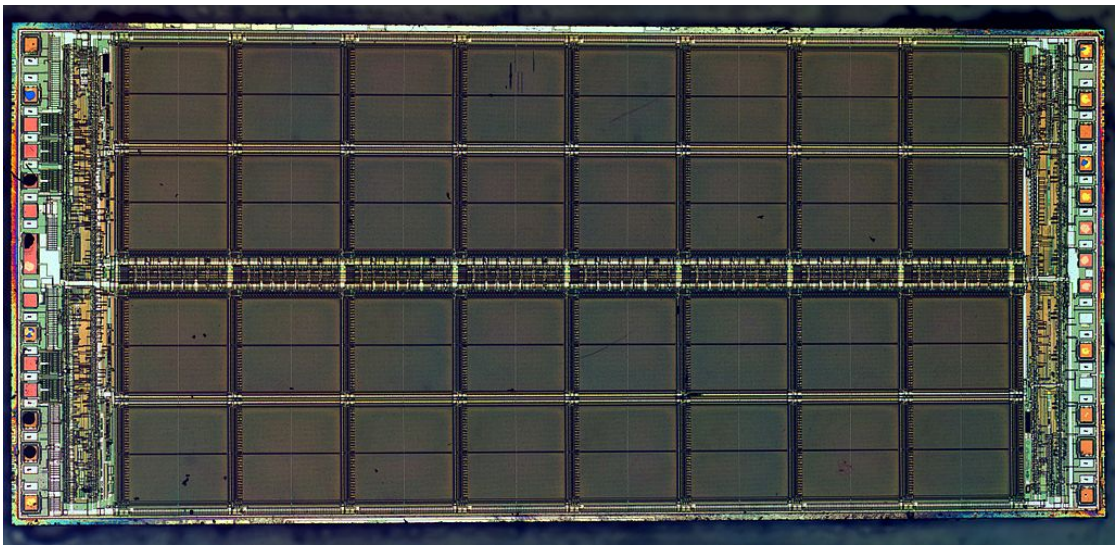
Let's now assume a word-addressable architecture where the word size is 16. We still have 2×2^{30} bytes of memory, but each memory location now holds *two* bytes (16 bits). So the number of memory locations is only $\frac{2 \times 2^{30}}{2} = 2^{30}$. Accordingly, we only need 30 bits to address each location.

So to summarise, **in order to address 2^n memory locations, we always need n bits for the addresses**. In byte-addressed systems, the number of locations is the number of bytes. In word-addressed systems, it's the number of words, which is half the number of bytes if the word size is 16, a quarter if the word size is 32, or an eighth if the word size is 64.

	#bytes	*#addresses*	*#bits*
Byte-addressable	2^n	2^n	n
16-bit word-addressable	2^n	$2^n/2 = 2^{n-1}$	$n-1$
32-bit word-addressable	2^n	$2^n/4 = 2^{n-2}$	$n-2$

Most architectures nowadays use byte-addressable memory. Let us take a look at how memory addresses map to the physical circuits that implement the memory.

RAM



A DRAM chip with 1 megabit capacity. Source: Wikipedia

RAM stands for *Random Access Memory*, which emphasises that the CPU can access any memory location in RAM (i.e., read from them or write into them) in basically the same amount of

time. This is different from, e.g., a hard disk, where it may be very fast to read the data that is currently under the read/write head, in a sequential fashion, but it can be very slow to read arbitrary pieces of data that are stored in different physical locations on the disk.

What we can see here immediately is that a RAM chip has a lot of *structure*. It seems to consist of a top and a bottom section, with each containing 16 square blocks in two rows, and each block in turn seems to have two parts.

This structure is no coincidence.

Memory Organisation

RAM is made up of *multiple* chips.

Each has a fixed size $L \times W$

- L: number of locations
- W: number of bits per location

E.g. $2K \times 8$ means 2×2^{10} locations of 8 bits each.

RAM chips are combined in rows and columns.

E.g. to build $32K \times 16$ memory out of $2K \times 8$ chips:

$2K \times 8$	$2K \times 8$
$2K \times 8$	$2K \times 8$
...	
$2K \times 8$	$2K \times 8$

Assume we want to use this RAM with a word-addressable architecture like MARIE.

How do we address individual locations?

This RAM has 16 rows of $2K \times 16$ each, which means it has $32K = 2^{15}$ words. In a word-addressable machine, we need a unique address for each word. Our addresses therefore must have 15 bits (with which we can express 2^{15} different numbers).

So how does the hardware know which RAM chip it should use when a given address is requested? We split up the addresses into two parts:

- Use highest 4 bits to select the *row*
- Use low 11 bits to select the word in the row

This is called *memory interleaving* (and in particular, high-order interleaving if the highest bits are used to select the row, and low-order interleaving if the lowest bits are used). In modern

architectures, interleaving can significantly improve memory performance, because it can allow the CPU to address several different memory chips at the same time (e.g. one for reading and another one for writing).

Indirect Addressing

Let us now go back to the software side, and take a look at how memory is accessed in MARIE instructions.

So far, all instructions we have seen are of the following form:

- *Store X*
- *Load X*
- *Add X*
- *Jump X*

All use the **value stored at address X**.

This is not very flexible!

We can only use fixed, precomputed addresses that are hard-coded into the instructions. But a typical coding pattern is to store data in an *array*, i.e., a sequence of consecutive locations in memory, often without a fixed length. With the fixed-address instructions above, there is no way to access e.g. the second element in the array, or loop through all elements to perform some operation (e.g. computing the average, or printing all of them).

We will now introduce a few new MARIE instructions that solve exactly this problem.

Indirect addressing

Instructions that use *indirect* addressing do not access the *value* at a location X, but they use X as a *pointer*: they interpret X as an address at which they can find the actual value for the operation.

Here are two instructions, both in the "normal" and the indirect addressing form.

***Add X*:** Load the *value* stored at X and add it to the AC register.

***AddI X*:** Use the *address* stored at X, load the value from that address, and add it to AC.

("add indirect")

***Jump X*:** Jump to address X.

***JumpI X*:** Use the *address* stored at X, and jump to that address.

("jump indirect")

In RTL, the difference becomes clear.

RTL for Add X:

```
MAR ← X
MBR ← M[MAR]
AC  ← AC + MBR
```

RTL for AddI X:

```
MAR ← X
MBR ← M[MAR]
MAR ← MBR
MBR ← M[MAR]
AC  ← AC + MBR
```

The indirect version loads the value from address X, then uses that value as an address (moving it from MBR into MAR), and loads the value stored at that address, before doing the addition.

To illustrate indirect addressing, let's assume that the memory contains the following values (written as hexadecimal numbers), starting from address A00:

Address	**Value**
A00	A03
A01	A04
A02	1FF
A03	203
A04	A23
A05	300
A06	000
A07	AAA

Let's assume that the program consists of the following sequence of instructions, using normal, direct addressing:

```
Load   A05
Add    A01
Output
```

This program would load the value 300 from memory address A05, then add the value A04 from address A01 to it, and output the result, which is D05. Now we'll modify the program to use an *indirect* addition instruction:

```
Load   A05
AddI   A01
Output
```

Again, the program loads the value 300 from address A05. But in the next step, it loads the value A04 from address A01, and then uses that value A04 as the actual address for the addition: it loads value A23 from address A04, and adds it to the 300, resulting in an output of D23.

Indirect addressing has a number of advantages:

- addresses don't need to be hardcoded
- we can *compute* the address!
- e.g. loop through a list of values

This example MARIE program loops through an array of numbers (starting at address 00F) and outputs each of them, until it finds a zero. Switch the Output mode in the MARIE simulator to ASCII to see the message! You can add arbitrarily many numbers to the array as long as you finalise the sequence with a zero. This is the usual representation of *strings* (i.e., sequences of characters) in many programming languages.

```
000  Loop,   LoadI Addr
001          SkipCond 800
002          Jump End
003          Add Sum
004          Store Sum
005          Load Addr
006          Add One
007          Store Addr
008          Jump Loop
009  End,    Load Sum
00A          Output
00B          Halt

00C  One,    DEC 1
00D  Sum,    DEC 0
00E  Addr,   HEX 00F
00F          DEC 70
010          DEC 73
011          DEC 84
012          DEC 0
```

Subroutines

A subroutine (also known as a procedure, function, or method, depending on the programming language), is a piece of code that

- has a well-defined function
- needs to be executed often
- we can **call**, passing **arguments** to it
- **returns** to where it was called from, possibly with a **return value**

Examples for common subroutines in high-level programming languages are System.out.println in Java, which takes a string as its argument and prints it to the console, or math.log in Python, which computes the logarithm of its argument and returns it.

Subroutines are probably the most important concept in programming: they allow us to *structure* a program, breaking it up into small parts. Of course, since all high-level languages are, in the end, executed by machine code instructions, we have to be able to write subroutines in machine code directly.

ISAs provide support for subroutines.

In MARIE,

***JnS X*:** Stores the PC into X, then jumps to X+1

X holds the **return address**

***JumpI X*:** Jump to address stored at X

(returns to the calling code)

Other Instruction Set Architectures found different solutions for this problem, e.g. storing the return address on a *stack*, or in a special *register*.

MARIE example subroutine:

```
            Load  FortyTwo
            Store Print_Arg
            JnS   Print
            Halt

FortyTwo,   DEC 42

            / Subroutine that prints one number
Print_Arg,  DEC 0               / put argument here
Print,      HEX 0               / return address
            Load Print_Arg
            Output
```

```
            JumpI Print         / return to caller
```

This code loads a number from memory and stores it in the location where the subroutine expects its argument. The JnS Print instruction then saves the current value of the PC into address Print (where we put a placeholder HEX 0), and jumps to address Print+1, where the actual subroutine starts.

The subroutine in this case just loads the argument from memory and outputs it. It then uses an indirect jump, JumpI Print, to jump back to the address pointed to by Print: remember that's where JnS saved the PC!

Run this code in MARIE to get a better intuition for how it works.

Wheeler Jumps

This is **additional** material that is **not going to be assessed**. It's just provided in case you want to learn more about an alternative implementation of subroutines.

Subroutines were invented very early in the history of computers. One of the people credited with inventing the concept is *David Wheeler*, who worked on the EDSAC machine at the University of Cambridge, UK, in the early 1950s.

But EDSAC didn't have instructions that use indirect addressing - so how was it possible to implement subroutines in EDSAC? The trick that Wheeler developed is to use *self-modifying code*, which means that the code overwrites instructions in memory to change its own behaviour.

The so-called *Wheeler Jump* consists of three steps:

1. The calling program loads the return address into the register (let's call it AC as in MARIE).
2. The subroutine overwrites its own final instruction with a jump instruction to the address stored in AC.
3. When the subroutine finishes, the new jump instruction continues execution where the calling program left off.

We can use the Wheeler Jump technique in MARIE to implement subroutines without JnS:

```
JumpFrom,   Load JumpFrom       / Load instruction "Load JumpFrom"
            Jump Sub            / Execute subroutine
            Halt

Sub,        Add Wheeler         / Add "magic number"
            Store SubReturn     / Overwrite final subroutine instruction
```

```
            Load FortyTwo       / Do something
            Output              / ... that's all
SubReturn,  HEX 0               / This is where the return jump goes

FortyTwo,   DEC 42              / Just some data
Wheeler,    HEX 8002            / Magic number that turns Load X
                                / into Jump X+2
```

So how does it work? The code uses two neat tricks. The first is in the first line: The instruction loads itself! So the value of the accumulator after executing the instruction is the bitpattern that represents "Load JumpFrom".

The second trick is that the subroutine adds a "magic" number to the accumulator when it starts. The magic number is hexadecimal 8002. A Load instruction is always of the for 1XXX, where 1 is the opcode for Load, and XXX is the address to load from. So adding 8002 to 1XXX results in 9YYY, where YYY is two greater than XXX. This encodes a Jump instruction (opcode 9) to the address two greater than the original Load JumpFrom - right behind the jump to the subroutine, and exactly where execution must continue upon return.

Try it out in the MARIE simulator!