Microcontroller System Design
Class Notes, Week 1.

Ahmet Onat Sept. 2022

1 Introduction

Almost all modern day products contain a microcontroller system to fulfill their job. In this course, we aim to introduce the fundamentals of designing microcomputer systems. The subject is quite broad and therefore we present the design of simple systems using the major building blocks of microcomputer peripherals. We also introduce some modern development tools. We believe that the students can then take over from there and will have the foundation to build on, for designing and implementing more complex systems.

We believe that the students come with a background in the design of electronic and logic circuits, and have taken a course in programming language; typially 'C'. This allows the course to concentrate directly on the subject matter. Since the subject is quite broad and requires the use of reference materials which change rapidly over the years (compared to more theoretical subjects which concentrate on a constant set of mathematical tools that are used repeatedly for design), we complement the lectures with laboratory work where the students learn by experiencing. Gaining fluency in using reference manuals of the hardware is also an important part of the course.

2 What is a computer system?

This question will undoubtedly appear absurd, as the computer is used constantly in daily life. However, we need to understand how the computer works and know its major components to be able to design computer systems from scratch. We need to know its capabilities and limitations. This section addresses that problem.

2.1 Cooking with a receipe

Let us imagine someone who does not really know how to cook. A receipe is needed. A receipe is actually a list of instructions with arguments. It may look like this:

- Take two eggs

- mix the eggs with 250g flour
- Add 200g sugar
- ...

The second component of cooking is to have a cook. The cook follows the instructions on the receipe and executes them. In our similie, the receipe is the **program memory** and the cook is the **central processing unit**, or **CPU** for short.

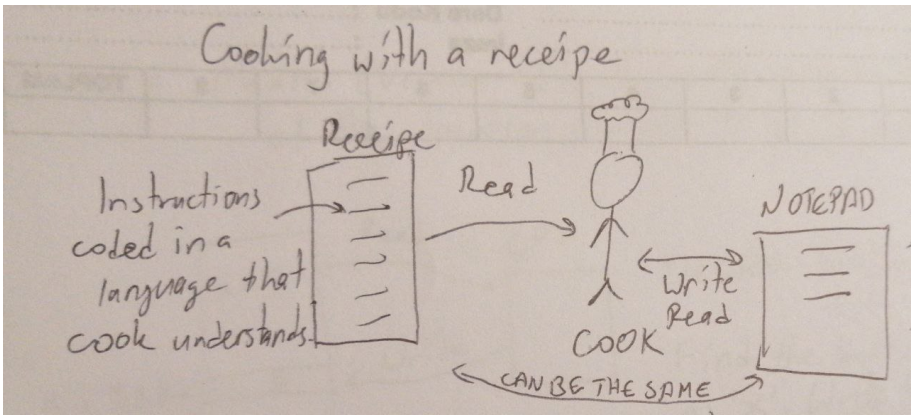

Figure 1: Cooking and computing are similar.

Lets take a more involved look. Take the third line of the receipe above:

- Add 200g sugar

This can be interpreted as an instruction "**Add**" and two arguments: "**200g**" and "**sugar**". We can interpret them in the following way:

- **Instructions**: If we think about it, there are only a finite (and small) number of instructions in cooking, perhaps 200 or so. We can say that the instructions on a receipe are selected from a menu of instructions.
- **Arguments**: For each instruction, there are some relevant arguments. For example you can say "Heat at 200^oC", but you cannot say "Heat at two eggs." Also, the number of arguments depend on the instruction. For example when you use the "mix" instruction, you must give it two arguments; mix what with what? The arguments of instructions in Step 2 and Step 3, for instance, must be different.

For each line of the receipe, the cook must do the following things:

- **Read** (fetch) the instruction
- **Decode** the instruction,
  - Understand what to do with it,
  - How many arguments must be read, if any
  - Read the arguments
- **Execute** the instruction with the arguments
- **Write** any result back.

The last item may need some explanation: If for example the receipe says "Cook for 20 minutes", the cook must record somewhere **when** it started the oven so that the end time can be decided.

We see that for cooking, there are two main components:

- Receipe
- Cook

and there are four steps in carrying out each line of the receipe:

- **Read** (fetch)
- **Decode**
- **Execute**
- **Write**

The computer model shown in Fig.2 looks quite similar to the above model.

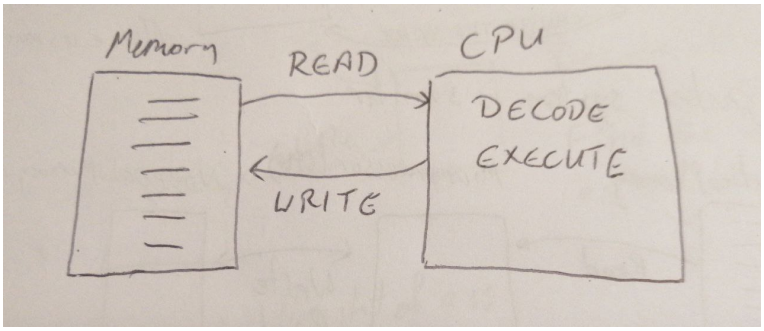

Figure 2: The minimal computer system.

3 One line of simple code

The computer is very similar to a cook. It reads **instructions** and **arguments** from a memory, and then **decodes** and **executes** them, **writing** the results back to the memory. No matter how big or small, this is roughly how computers work. Let us examine a simple line of code:

`x=5`

It can be broken down to:

1. 'x': Some special value that we must remember. (Think: if you want to remember something, you write it somewhere you know. Actually, the computer does not know 'x' at all. x is only the label of a location in the memory.) This is one of the arguments.
2. '=': Operator. **This is the instruction.**
3. '5': Another argument.

A few words about 'x': If you want to remember something, you write it somewhere you know. The computer writes it in a pre-determined location in the memory. Actually, the computer does not know 'x' at all. 'x' is only a label of a location in the memory; location 3826 in this example. See Fig. 3

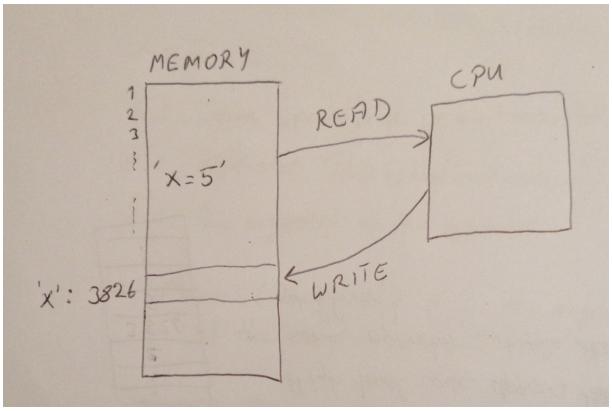

Figure 3: How a simple one line program is executed.

Here if we **decode** '=', we see that it means:

`Take the first argument and write its value at the location shown by the second argument: 'x'`

The whole expression x=5 can be thought of as: "`LOAD INTO MEMORY LOCATION (3826) THE VALUE '5'.`" where 3826 is the location in the memory with the label 'x'. However,

"LOAD INTO MEMORY LOCATION () THE VALUE ()." is awfully long for an instruction name. We can encode it with a number, say '1'. So whenever the CPU sees an instruction encoded as '1', it will decode it as the long description above. The memory content for the one liner code above will look like Fig.4. The computer sees the first item '1', and interprets it as the instruction and decodes it, then reads the arguments, and executes. Finally it writes the results back to the memory.

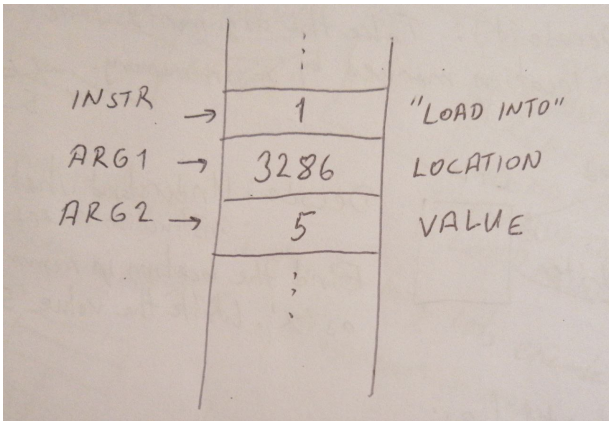

Figure 4: How the simple one line program is encoded in the memory.

This is the essence of the execution of all computer code.

3.1 Another example

Take the slightly more complicated code:

```
x=x+5
```

The mathematical operator is the same symbol, but now **it has a different meaning!** It is a different instruction compared to the previous one. It can be interpreted as:

```
"READ FROM MEMORY LOCATION (3826), ADD THE VALUE '5' TO IT,
  LOAD BACK THE RESULT INTO THE SAME MEMORY LOCATION."
```

This new instruction is also too long. Lets encode it as '2'.

The processor is doing the following when it executes this line of code:

- **Fetch**: Read the instruction.
- **Decode** the instruction.
- bf Read:

  - Read the memory location with label x (which is 3826 in this example)
  - Read the contents of memory location (3826) (it contains 3)
  - Read argument 2 (which is 5 in this example)
- **Execute**: Perform the addition.
- **Write**: Write the result back into memory location with the label 'x' (which, of course, is 3826 in this example.)

The execution of the above program `x=x+5` will look like Fig.5 and appear in the memory as sown in Fig.6.

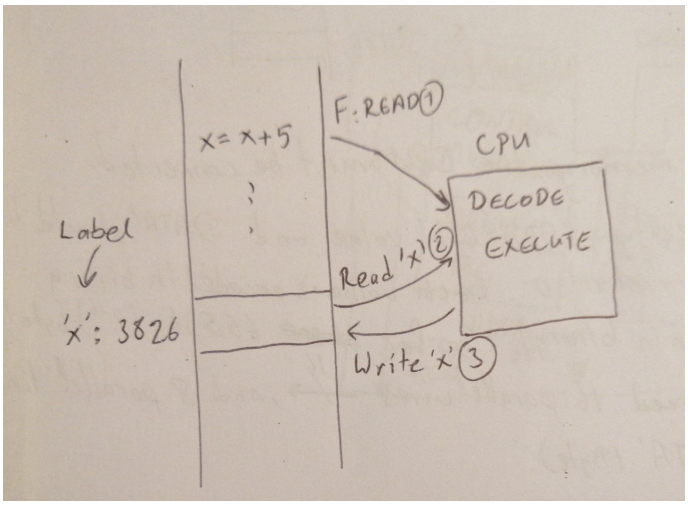

Figure 5: How the second example runs.

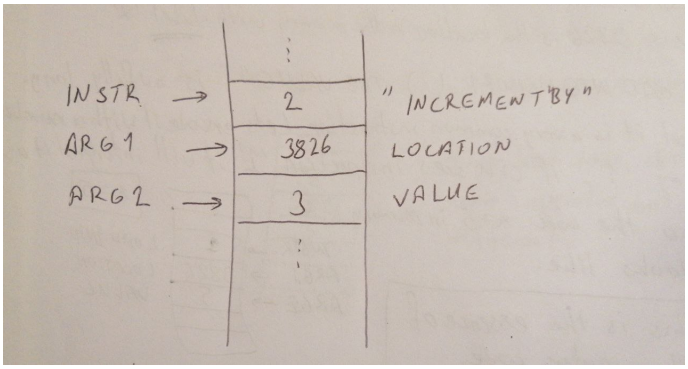

Figure 6: How the second example is encoded in the memory.

Note that in the high level program, the operators are the same, but they encode into completely different instructions.

4 Hardware basics

The processor fundamentals have been covered. We can now look into the hardware components of the computer in more detail. First some formal definitions are in order:

- **Address**: A specific location in memory. Typically we think of the memory as a sequence of locations to store one byte of data each. The first location is 'Address 0', then comes 'Address 1' and so on.
- **Data**: The value written in a given location in the computer system. If the value is in the memory, it is the value written in a given address location.
- **Bus**: A collection of parallel wire connections between components of the system with a common purpose. For example Address bus may consist of address line '0' (A0), to address line '15' (A15). So the Address bus consists of 16 wires consisting of address lines A0 to A15.
- **Register**: A memory location within the CPU for temporary storage of arguments or results.
- **Flag**: One specific bit within a processor that has a special meaning.

4.1 The connection with the memory

The CPU must be connected to the memory. It specifies the address value, and it specifies if this is a read operation, or a write operation. If it is a read operation, then the data will be transferred from the memory to the CPU and vice versa.

Each value is encoded in binary. Therefore for a memory of size 65536 bytes (2^{16}), there must be 16 parallel wires between the memory and the CPU to specify the address value. Similarly, there must be 8 parallel wires between them for the data value. Since it is tedious to draw so many parallel lines representing each wire, we draw them as a "**Bus**", simplifying the drawing. The number of wires in a bus is represented by a slanted cross line and a number above it specifying the number of connections. The system is depicted in Fig.7. It shows an address bus with 16 connections, a data bus with 8 connections and a single Read/Write connection. The meaning of 16 address connections is as follows: Since each of those connections can be a logic '1' or a logic '0', there are 2^{16} possible combinations, i.e. 2^{16} address locations. The address lines are typically named **A0, A1, ...**. The address locations in the memory are sequentially numbered, from 0 to (in this example) $2^{16} - 1$, which is 65535.

The CPU also manipulates control lines. For example in Fig.7, there is one wire driven by the CPU and it tells the memory whether this operation is a read or write operation. Since it must either be a read or a write, one single wire is sufficient. It is typically labeled as: `R/W`. This means if the line is a logic '1', the CPU specifies a read operation, and if the line is a logic '0', a write operation. The convention for such connections is that the letter at the numerator is associated with the logic '1' state and the one at the denominator, the '0' state.

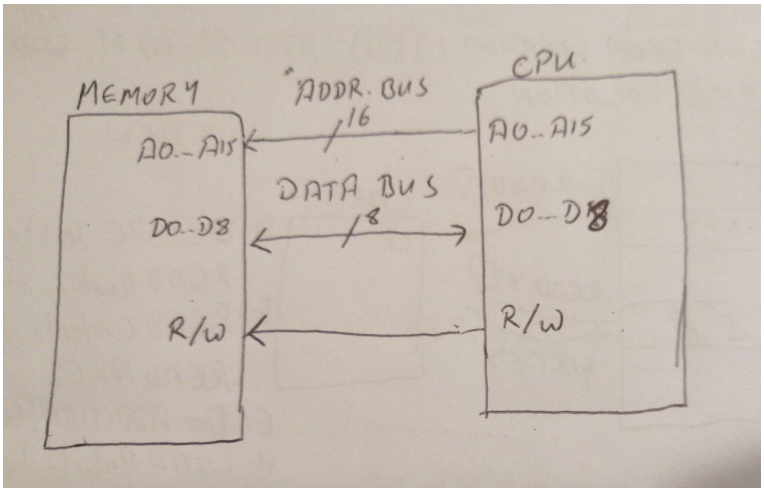

Figure 7: Minimal computer wiring diagram.

4.2 Connection of several devices to the CPU buses: The memory map

Until this point in the discussion, we have talked about only 1 device on the address bus. However, there can be several different devices. For example, most systems have one ROM (Read Only Memory) (the contents do not change between power cycles), one RAM (Random Access Memory) (the contents can be read and written), and several peripheral devices such as general purpose input and output (GPIO), analog to digital converters (ADC), timers (TIM) and so on. Whereas the memory devices occupy large amounts of addresses, peripheral devices typically occupy only a few. See Sec.5 for more details.

Each device is connected to the same address and data buses. Each respond to a different address, using "address decoders": They respond when the CPU accesses a specific memory range. Figure 8 shows the conceptual connection diagram. Think about this as the CPU "writing" postcards and the postman takes them to the correct address. The addresses of each device on the complete address space is called the **"Memory map"** of the computer system, and it is one of the most important components of the design. The programmer must

know where in the address space each device is located to be able to access them. For the microcontroller device, all of the peripherals and memory are on the same chip. Therefore, the memory map is already set by the manufacturer. See Section 2.2.1 "Memory mapping" and Fig. 2 of the User Manual of LPC824 for how the memory is arranged in this processor.

Remember that the LPC800 series devices are 32 bit processors. The address bus is 32 bits wide. It can therefore address $2^{32} = 4294967296$ address locations (4GB). However, for such a simple small processor, most of the address space is not occupied. (This means, if we write to the unoccupied locations, the data is lost -the postman cannot find anyone at that address- and if we read from those addresses, we will get random data.) The data bus is also 32 bits wide and data can be read off in 32 bit wide sections. It can also be read back in 16 or 8 bit wide sections.

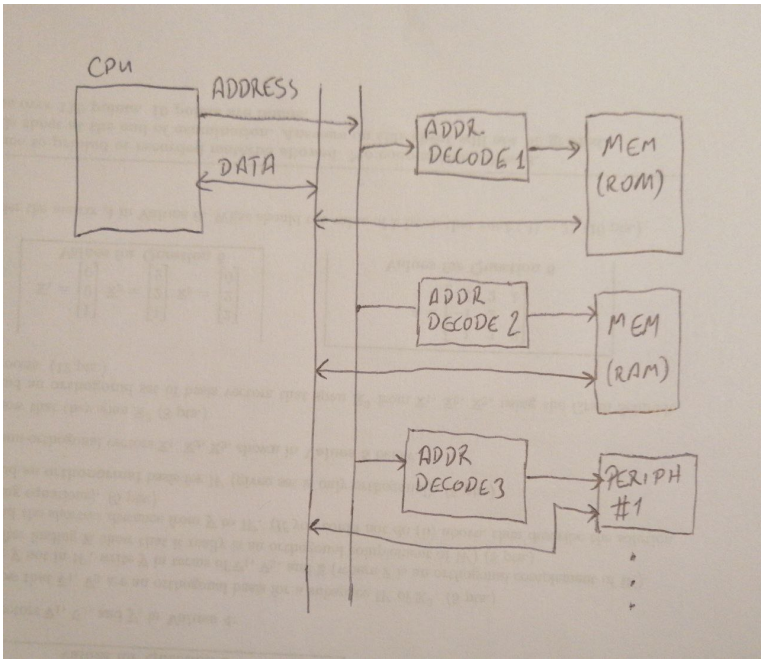

Figure 8: How several devices connect over common address and data buses.

In the LPC824 processor the main components of the memory map are as follows: Flash memory (ROM) is located starting from address 0x00000000 so this is where the address 0 of ROM is located. Every consecutive address is accessed by incrementing the address count by 1. It is 32kB long (0x8000), so the last ROM address location is 0x00007fff. The RAM is located from address 0x10000000, in a similar fashion. The RAM memory is 8kB long (0x2000 in hex), so the last RAM address is 0x10001fff. Peripherals are generally located from 0x40000000, and the base address of each peripheral device is shown in Sec. 2.2.1 of the Reference Manual, Fig.2. Unlike the RAM or ROM, each memory location of the peripheral

Device	Start	Length	End
ROM	0x00000000	0x8000	0x00007fff
RAM	0x10000000	0x2000	0x10001fff
Peripherals	0x40000000	-	-
GPIO	0xA0000000	-	-

Table 1: Memory map organization for LPC824. There are many peripheral devices, each with different numbers of registers, so the length and end address is not meaningful. The GPIO only has a separate start address than the other peripheral devices.

devices is called a register. Each bit of each register is physically connected to an electrical circuit, so that writing to or reading from them has a physical meaning. See Sec.5 for more details.

4.3 The program counter: PC

Since the cooking example implicitly we are making use of a line counter. Somewhere we keep the number of executed instructions in the receipe so that we can go to the next one. This is called the program counter (PC). It is a pointer that points to the next instruction in the memory and increments automatically as soon as it is executed. If we call a function (a code that resides in a different memory location), then the program counter must abruplty change. Such changes are managed by the CPU control logic.

4.4 Arithmetic logic unit: ALU

Almost all of the instructions of the CPU involve basic arithmetic or logic (AND, OR etc) operations. The arithmetic logic unit executes these instructions. It can only work with data that has already been read into the CPU; it cannot operate directly on data that resides in the memory. Similarly, its results will be temporarily stored within the CPU until they are written to the memory.

The number of bits that the ALU can operate on typically determines the data width of the processor. If it can only do 8 bit arithmetic, we call it a 8 bit CPU, if it can do 64 bit arithmetic, it is called a 64 bit CPU. Think of this as the number of digits on a handheld calculator. It can be a 8, 10, 12 digit calculator. We can do 12 digit arithmetic on a 10 digit calculator for example, but we need divide operations into 10 digit chunks. It will be slower and require temporary storage space. So if we can justify the cost, a large word width processor has more performance.

4.5 General purpose registers

As we have seen, data must be read inside the CPU and stored temporarily. There are a small number of registers that are tightly coupled with the ALU. There are generaly only 16 - 32 such registers. One of them can be selected to provide one argument of the arithmetic operation, and another the other argument. The result can be selected to be stored on yet another register.

Obviously the configuration of the connections between the registers and ALU has a great impact on the performance of the CPU, so this is an important part of the CPU design.

4.6 Status register

There are some important conditions in the processor that must be reported. The status register is used for this purpose. For example, if the result of the last operation turns out to be '0', this is important. It is reported by pulling a specific bit in the status register (called a "**Flag**") to logic '1', in this case the "**zero flag**". A negative result is also important and reported by another flag.. For example a comparison instruction such as

```
if (x==5)
```

is generally executed by calculating `x-5` and checking whether the result of the operation has set the zero flag or not.

All of the concepts above are represented in Fig. 9. Note where the Address and Data buses originate in the processor.

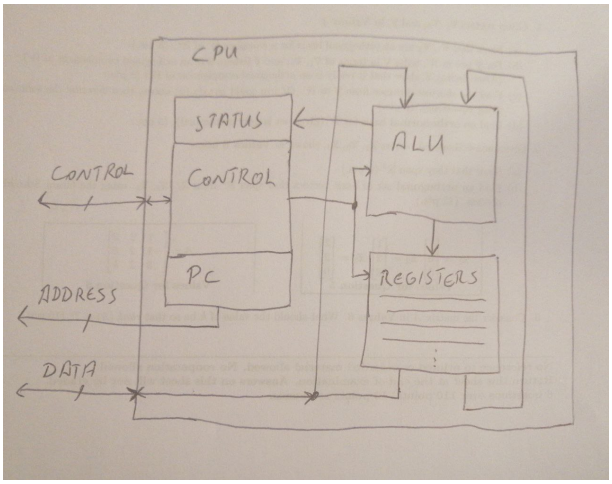

Figure 9: Basic structure of a CPU.

4.7 What happens after power-up

One remaining topic is, how does the processor start running a program after a power-on reset. The processor is hard-wired to start execution from a specific address. In the case of LPC824, this address is the first address location of the memory, 0x0000 0000 (space included to improve readability). The first value it fetches from 0x0000 0000 address is the location of the stack (i.e., the stack pointer). We will get back to stack later in the course. The second value which is 0x0000 0004 is the location of the code that the processor must execute. The processor is again hard-wired to load this value to the program counter and jump to the program that you have written. This special value is called the **"reset handler"**.

Therefore after power-up or after pressing the reset button, the processor goes to address 0x0000 0000 to read the value of stack pointer, and then 0x0000 0004 to read the location in memory of the main program, jumps to it, and starts execution.

One final note on this subject: Why does the processor read from 0x0000 0004 as the second value and not 0x0000 0001? This is because the memory is 8 bits wide but the processor is 32 bits. Therefore 4 consecutive address locations must be read to make up one 32 bit word and that makes the next address location up from 0x0000 0000 to be not 0x0000 0001 but 0x0000 0004.

5 Peripheral devices (input and output)

The computer must interact with the outside world. It does this through **"peripheral devices"**. Simply, these are special address locations which are internally connected to electrical circuits which have electrical connections to the package of the processor, and therefore can manipulate external signals, or can measure external signals. These special address locations are called **"registers"**.

5.1 Simple output peripheral

Let us consider a very simple peripheral device, the output register (Fig.10). This would be an electrical device where each bit of information written to the register (A.K.A. the memory address that the register resides in), appears at the pins of the microcontroller as voltages. A '1' written to any bit of the register appears as a voltage higher than V_{OH} in the **electrical specifications section of the datasheet** on the corresponding microcontroller pin, and a '0' written to any bit appears as a voltage lower than V_{OL}, again in the datasheet. (Typically we can consider logic '1' appearing as $3.3V$, and logic '0' as $0V$, if the pin is not electrically connected to a heavy external load. The current capacity of each pin is also given in the electrical specifications section.)

In Fig.10, the memory address 7856 contains the simple output register, and any data written to that register will appear as voltages at the output pins. Although Fig.10 shows 8 output pins for simplicity, the LPC824 processor has 28 input/output pins in each register. This number is limited by the number of available physical pins on the package, but could theoretically go up to 32 pins.

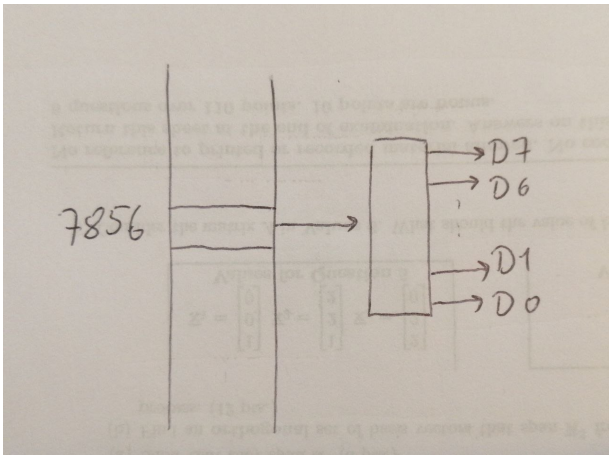

Figure 10: Simplest output device.

5.2 Simple input peripheral

Similarly, a simple input device can be conceived (Fig.11) where external voltages applied to the microcontroller system appear as bits at the register as logic values. Any voltage that exceeds V_{IH} written in the **electrical specifications section of the datasheet** will appear as a logic '1' and any voltage lower than V_{IL} will appear as a logic '0'. When the CPU reads from the register (A.K.A. the memory address that the register resides in), the value will reflect the input voltage levels. In Fig.11, the address appears as 7863.

Care must be taken **not** to apply higher voltages than the **"Absolute maximum ratings"** also specified in the **electrical specifications section of the datasheet** to any pin of the microcontroller. Each pin may tolerate a different voltage; for example most pins will not tolerate any voltage higher than V_{cc}, which is $3.3V$, but some may tolerate $5V$. Again the information is in the datasheet. If any higher voltage is applied, the microcontroller may become permanently damaged. Note that high voltage does not need to come from power supplies etc. For example, the outputs of USB to serial converters or sensors may be set to $5V$, and exceed maximum voltage specs. Again, in LPC824, there are 28 input pins rather than 8.

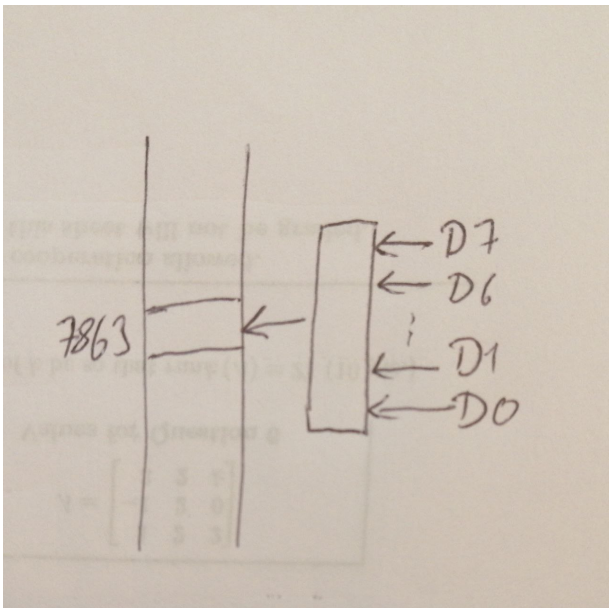

Figure 11: Simplest input device.

5.3 More complex input and output: GPIO

The peripheral devices in Sec.5.1 and 5.2 are too primitive and restrictive. A designer will typically not want all 32 outputs and 32 inputs in the design, but rather an arbitrary number of outputs and inputs, which may or may not add up to 32. Therefore actual devices in modern microcontrollers are more complex. Any of the pins can be configured as inputs or outputs, and some may be dedicated to completely different functions (See: Sec. 5.5) within the same register. This more general peripheral device is called a **"General Purpose Input Output (GPIO)"**.

To be able to do such configuration, the peripheral has some more companion registers. For example one companion register controls whether a particular pin is configured as an input or an output. Writing a '1' to a specific bit of this register may configure the pin as an input and '0' may configure it as an output. Similarly, another companion register controls whether a pin in the GPIO register is used as a GPIO, or as an alternate function. Again, writing a '1' may mean that the bit is connected to a physical GPIO pin, and '0' may mean that it is detached, and the pin is controlled by another function of the microcontroller such as a timer. We use "may" here, as it is even more complicated than this and the reader must consult the reference manual to learn about the exact configuration of the microcomputer at hand. There is no logic or theory to it; the designer at the semiconductor company has made the decision about how the design should be, and its description is in the Reference Manual.

5.4 Package

One important question is, "Which physical pin does the register contents appear at?". We are now at the boundary of logical descriptions and a physical device that we can buy over the counter. The **"pinout information"**, again can be found in the **"Pinouts and pin description"** section of the datasheet. However, the same microcontroller comes in packages with different pin counts. Care must be taken to ensure that the pins are identified using the correct package that we have. The `LPC824M201JHI33` microcontroller on the lab boards is in a "QFN33" package (Low profile Quad Flat No-lead package with 32+1 pins. Quad means that it has pins on all 4 sides. The"+1" refers to the ground pad underneath the package.) It has exposed pins as very fine coming out of the package. This is a very common package type.

Other types are QFN (Quad Flat No lead) where, the pins are copper slivers flush with the package outline, and BGA (Ball Grid Array) where pins are not visible, at the bottom of the package, arranged in a matrix. BGA allows the maximum of connection pins since it makes use of the whole package surface area rather than only the edges.

5.5 Alternate functions

The modern microcontrollers, even the smallest and cheapest ones are so complex and have so many integrated peripheral devices that if all of their functionality was brought out to a separate pin, we would need very large packages. For example, the LPC82x processor family, being a medium complexity device has a GPIO with (28 pins), 9 communication interfaces (4-5 pins each), 7 timers (4 or 10 pins each), 12 analog to digital converters, voltage supply, many clock and debugging pins besides others. Even this superficial description requires more than 200 pins. However, the LPC824M201JHI33 has only 32+1.

Since the semiconductor manufacturer cannot foresee or impose which combinations of peripheral devices that the end user will need, they tie the physical outputs of many peripheral to the same physical pin. Sometimes the same output is also tied to several different physical pins (counterintuitive?). The end user is then free to select which of the internal peripheral functions are brought out to the physical pin depending on their application.

At power-up, each pin has a default function (generally a GPIO input, so that the CPU does not apply unintended voltages to connected external devices). During initial configuration, the system is configured so that the desired functions are assigned to the desired phsical pins. The LPC824 has a very interesting feature that any peripheral device output can be connected to any physical pin (with some limitations). The user is completely free to configure the pinout of the processor as they wish.

Some microcontrollers have a dedicated programmable ROM for this and the configuration is written there semi-permanently so that at power up, the peripherals come up pre-configured.

For theLPC824, at power up, the pins come configured as GPIO inputs, and during the firsts steps of the programming, they are configured to the desired function.

6 Computer, microcomputer, microcontroller

This definition has deliberately been postponed until a basic understanding of the overall system is given. The computer is what is defined in Sec. 2, and the implementation as given in Sec. 4. A microcomputer is nothing but a computer, but in a smaller package in terms of physical size, power consumption, computational capability etc. In that sense, the distinction between a computer and microcomputer depends on the technology of the day, and changes in time. A computer of a few years back may be the same in terms of capability, but a computer of today may have equivalent power requirement to a microcomputer of yesteryear. A microcomputer may have the CPU and memory devices on the same chip or on different chips (typically on different chips).

A microcontroller is a CPU, RAM, ROM and several peripheral devices on the same chip. In that sense, we typically talk about **"microcontroller families"** where the same CPU is mated with a various selection of memory and peripheral devices, and the end designer selects one with the most suitable device that suits their requirement. The LPC824 is a microcontroller manufactured by NXP of the Netherlands. Formerly the processor division of Philips, the company was renamed to NXP in 2006. Philips is more widely known for its consumer electronics products such as TV sets and home appliances. However, since the 1950's it has also been one of the biggest semiconductor device makers, starting with transostors and logic gate ICs. It has grown to many different product areas from microprocessors to sensors and wireless devices.

6.1 The ARM processor

The NXP LPC8xx microcontroller family is based on the ARM 32 Cortex M0+ CPU. ARM processors started out as a British design, the "BBC Micro", standing for British Broadcastng Corporation Microcomputer System" designed in 1982 to make computers available to students and households (it rings a bell with Raspberry Pi, another successful British initiative with a similar goal). The design was by Acorn Computers. Struggling to find a microprocessor for their next designs, Acorn eventually designed their own, "Acorn RISC Machine", or "ARM" for short, in 1983. The design evolved over the years and with the help of other major computer manufacturers, became the "Advanced RISC Machine" (ARM again) in 1990. There are 3 lines of ARM processors: Cortex-A (Tablets, phones etc), Cortex-R (high performance real-time for infrastructure, communications, media players etc.), and Cortex-M (microcontroller applications).

Today, ARM dominates the embedded processor market. Howewer, as a business model, ARM itself does not produce any chips. Instead, it licences out the processor and peripheral design and specifications to other companies which design and build their own microcomputer chips and pay a loyalty to ARM for each chip they produce. Other major architectures are MIPS, PowerPC, SPARC-V, etc.

Some of the other more prominent microcontroller manufacturers are: ST Microelectronics (SGS Thompson), Infineon (Siemens), Renesas (Hitachi), Microchip, Atmel (now also owned by Microchip), Silicon Labs, Nordic Semiconductor, Texas Instruments. Although most of them are Western, many Far East manufacturers are emerging in the market with original designs, mainly ARM core, but also original designs. For example, the Tensilica processor family (starting with the ESP8266), Rockchip with devices in many areas etc. Far East products are also appearing in black market. A new contender in microcontrollers is SPARC-V which is also a 32 bit processor. It has a big advantage over ARM in its business model. ARM design is licensed (so a fee must be paid to ARM Holdings for each chip produced), but SPARC-V is an open source design, with no license fee obligations. As of 2023,the prices for SPARC-V processors can be as low as 10US cent. So every penny counts.

7 Where to go from here

This course is designed to familiarize the students to microcontroller systems. We will be designing and implementing devices starting from the simple to the more complex. We will be adding one more functionality at each step, and use all of the previous functionalities. The labs are a major part of the course, since without lab work which force the students to practice, the whole concept will remain as just a concept. It is the practice during the lab sessions which allow the students to learn.

To make the most out of the class, each student must try out the ideas as much as possible, make mistakes and learn from them.

It is very important that you **keep the processor manuals always at reach: The Reference Manual and the Datasheet**, and refer to them as much as possible. You will find a lot of material on the Internet, some of which are worked out examples of the lab work of this course, others are general information about the processor. We propose that when you are faced with a problem, first try to solve it on your own. If not possible, you can try out general examples that you find on the Internet, to guide you. But eventually, try to work out the labs on your own as much as possible, take it as a challenge. This is the way to digesting the material and adding it to your design arsenal.

Always remember: **"A good engineer requires Knowledge, Skill and Experience."** Knowledge has essentally become trivial, and is always at your fingertips with the search engine, or in textbooks. But you can only gain experience and skill through hard work trial and error. Knowledge will not let you stand out from the competition in your job or research career. Skill and Experience will...