Chapter 14
Microprocessor Design

14.1 A brief Introduction

You are probably reading this on some electronic device, and that will have some form of microprocessor feeding your screen and responding to your swipes or clicks. Microprocessors and their smaller cousins, microcontrollers (in household appliances, cars and spacecraft) are ubiquitous, so it is essential that we understand them.

This chapter does not simply discuss their structure and function, instead we aim much higher, to try to understand how they are *designed*. We shall present our design for a CPU which extends the "mu0" model, a conceptual model from Manchester University where we have added input-output. This design will be realized using VHDL.

Like buildings, CPUs are designed by architects, they have a particular architecture. Just as the architecture of buildings has changed over time, so has the architecture of CPUs. One significant factor in this change has been the cost of electronic components. You are well aware that computer programs are stored as instructions in memory. In the 1980s[1] memory was expensive, so CPUs were designed so that each instruction completed a complex processing operation. So a program occupied a small amount of memory. However such complex operations were often slow. This architecture used by Intel is called Complex Instruction set Architecture (CISC). When memory became cheap, the architecture moved towards simpler instructions which executed much more rapidly, but several simple instructions were required to complete each complex operation and this consumed more

[1] Intel made their first 4004 microprocessor in 1971. Their 8086 processor (origin of the Pentium series) was made in 1978 and the first Pentium was made in 1993.

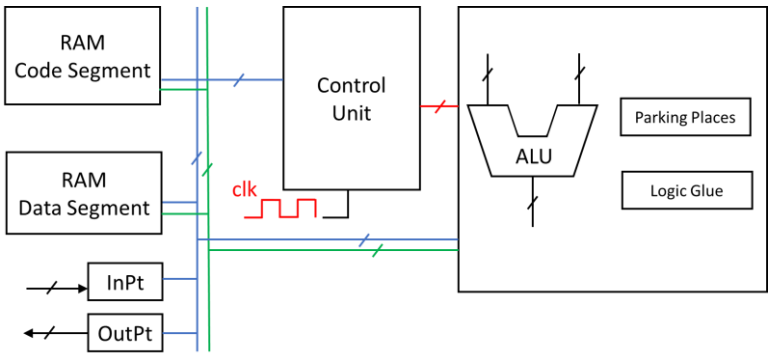
memory. Such architectures are known as Reduced Instruction Set architectures (RISC). There's an interesting back-story of why and how Intel migrated to this architecture for its Pentium processors, while retaining the illusion that the chips were in fact CISC, in order to main compatibility with applications that were coded for the original CISC chips.

14.2 Two Architectures Compared

So what are the mainstream CPU architectures we need to be aware of today? There are many, but let's have a look at two. The most important is the 'Single Instruction Single Data' (SISD) architecture. Think about this line of code,

```
count = count + 1
```

There is one item of data here (the variable count) and there is one instruction (add 1). I guess you will see that most of code you have written conform to this architecture. A hypothetical CPU design which supports this architecture is sketched below.



There are a number of components to this architecture. We have on the left:

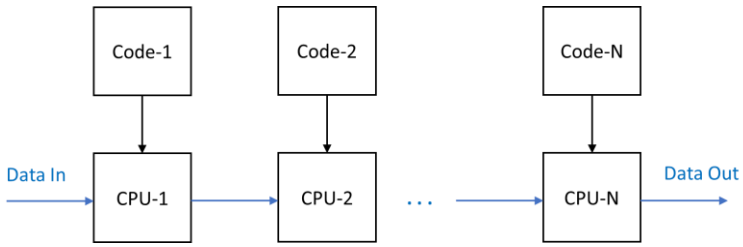
- A RAM Code Segment where our program instructions are stored.
- A RAM Data Segment where our variables are stored
- Input and Output Ports

On the right we have :

- The Arithmetic Logic Unit (ALU) which performs calculations such as add, subtract, etc.
- Some parking places for variables as they are shuffled around, and some logic to glue everything together.

In the centre we have the Control Unit which coordinates the entire CPU activity. The control unit is fed with an instruction, and it sends signals to the stuff on the right to make it the instruction actually happen (execute). This SISD architecture is the one we shall use.

Let's compare this with a different architecture known as a 'Systolic Array'. It's a bit like a production line where something (data) comes in at the left and passes through various stages of processing and exits at the right. Each stage has its own processing (code).

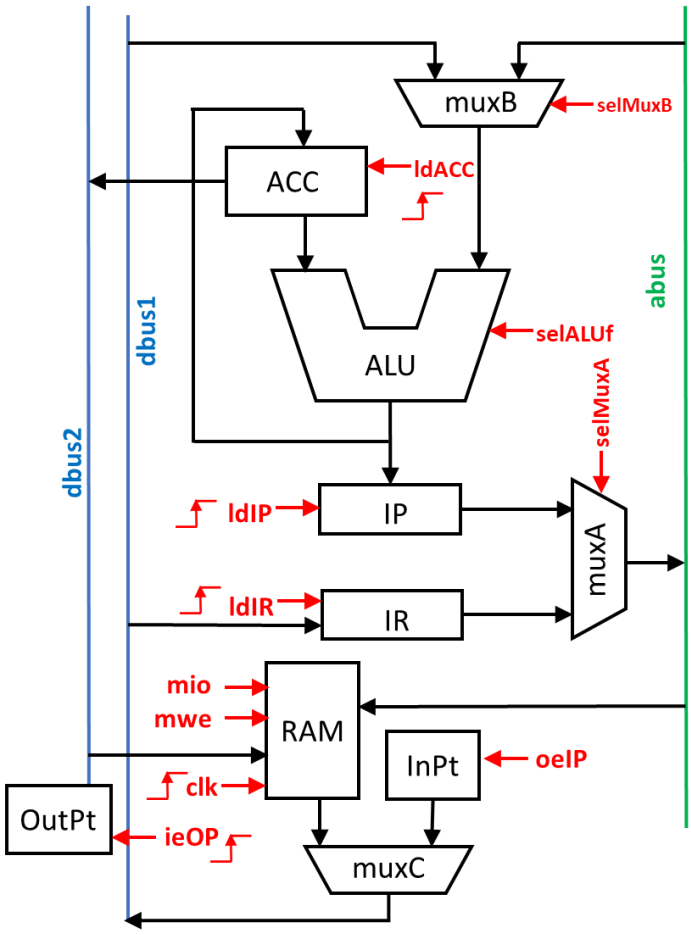


Each stage in this pipeline executes its own code, and the results are passed down to the next stage. A recent example we have been working on is to understand swimming fish. Their backbones are made of segments and each segment has a neural circuit which makes it bend, but also communicates to the downstream segment, which produces a travelling wave or swimming motion. In our model, the 'Head' segment (Code-1) codes an oscillator which provides a regular signal between +1 and -1. Then the following segments (Code-2,… Code-N) apply a phase delay in this signal, so each successive segment moves at a later time. This results in swimming. Our code has been implemented on chain ('systolic array') of Arduino Nanos and works well. But that's another chapter!

14.3 Introduction to CPU2S

This is the CPU we shall synthesize. There are two dimensions we must understand together, the CPU hardware and how it executes instructions. These instructions are not lines of code like C, but are assembler instructions, the lowest level of programming. Each instruction in the Instruction Set is converted directly to electronic signals.

First let's look at the hardware structure shown in the diagram below.



What do we recognize? Well, there is an arithmetic-logic unit (**ALU**) which does additions etc., and there is some **RAM**

memory. On real CPUs this is *cache* memory on the CPU die, not to be confused with RAM on the mother board. In our CPU2S design we can think of the RAM as partitioned into two *segments* the *code* segment (where the instructions are stored) and the *data* segment (where variables are stored). Programming a Pentium also uses code and data segments.

To understand the other components, let's start with the **ALU**, Fig.1. Its shape reminds us that there are two inputs at the top and one output at the bottom, So if we input 3 on the left and 5 on the right, the **ALU** could be instructed to add these and give us an output of 8. Now, one **ALU** input is connected to a *register*[2] **ACC**, which stands for 'accumulator'. The input to the accumulator is connected to the output of the **ALU**, so when the above addition is performed, the '8' would be written back into the accumulator **ACC**. If we did a second addition with 5 on the right, then we would have 8 + 5 = 13 and 13 would be written back into **ACC**. So you see where the name *accumulator* comes from, since it could accumulate the results of successive instructions.

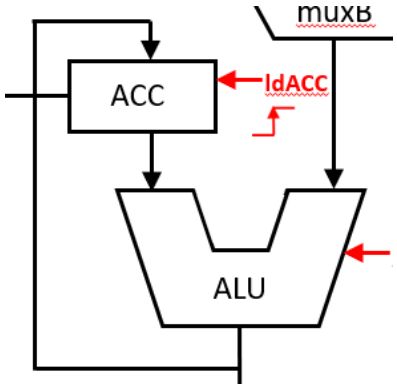


Figure 1 ALU and Accumulator inter-connecting signals.

Below the **ALU** you can see two other registers, the first is **IP** the *instruction pointer* which points to the address of the instruction in memory being executed. As a program runs, the **IP** will contain 0 for the first instruction, then 1 for the second, then 2, 3, 4, and so on. The other register **IR** is the *instruction register* which holds the actual instruction currently being executed. We'll come back to this in a moment, but let's focus on the **IP**. To get the instruction at the address held in the **IP** out of memory, the value of **IP** has to be propagated to memory as an address; it does this by travelling along the **abus** short for *address bus* which is shown in green on the right. To get to **RAM** it passes through a *switch* called **muxA**[3]. It's the job of the Control Unit to flip

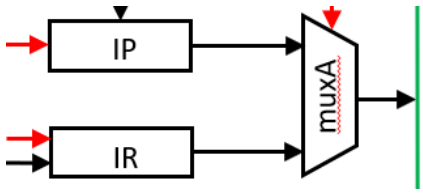


Figure 2 Instruction Pointer (IP) and Instruction Register (IR) switched onto the address bus.

[2] A register is high speed memory which holds one item of data.
[3] **mux** is short for 'multiplexer' which is the electronic engineering term for a switch.

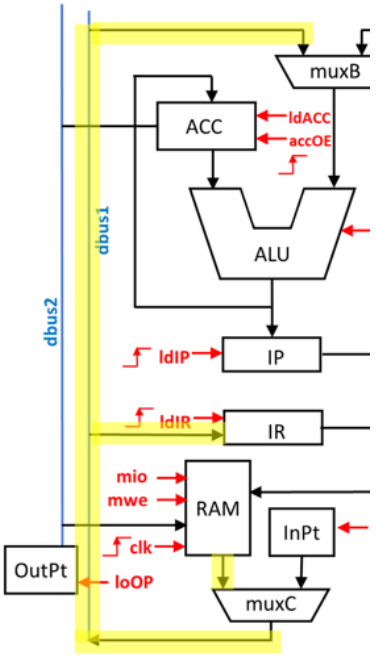


Figure 3 Data Path out of memory with two possible destinations.

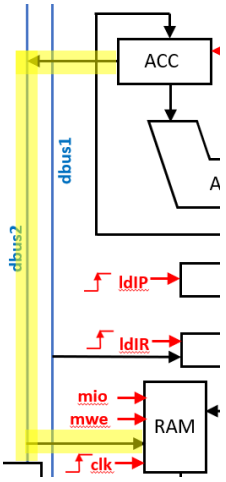


Figure 4 Data Path out of the accumulator into memory.

the switch so that **IP** is selected and not **IR**, we'll see the details of this later, see Fig.2.

Now let's look at what happens when we get some data out of **RAM**; it comes out at the bottom and passes through another switch **muxC**. Then it propagates onto one of the data buses **dbus1** and travels upwards. If you look carefully, you can see that the data can end up in one of two places. Either it could end up in the *instruction register* **IR**, which is what happens when we ask to get an *instruction* from the code segment in memory, or it could end up at the right input to the **ALU** (passing through another switch **muxB**), which is what happens when we ask to get an item of data from the data segment in memory, see Fig.3.

Now let's see what happens when we have done a computation and we wish to store the result in the data segment in memory. You now understand that the result of the computation is stored in the accumulator **ACC**. You can also see that there is a path downwards on **dbus2** from the **ACC** and into **RAM**. Of course, an address in the data segment needs to be supplied, and when we come to look at the details, we shall see how this comes down the address **abus**, see Fig.4.

There's two more hardware blocks that need introducing, which form the input-output system. The **InPt** is the input *port* which reads input switches, and the **OutPt** is the output port which is connected to LEDs. You can see that data comes down into the **OutPt** via **dbus2**, though you may correctly ask the question how do we know whether to send tata into **OutPt** or into **RAM**? Later! The input provided by **InPt** ends up on **dBus1** where it will probably end up in the **ALU** but **dbus1** can also be fed from **RAM**. So you can see the purpose of **muxC**, to select if we send **RAM** data *or* **InPt** data to **dbus1**. We can't do both at the same time, just like you can't eat a sandwich and take a drink at the same time, see Fig.5.

To bring this hardware section to an end, let us reflect on one important design decision. The diagram above has various components connected by lines (**abus, dbus1**, and other intermediate lines). These carry electronic signals and comprise several *bits*. Our CPU2S has been designed to be a '16-bit' device, which means that all of the lines, the registers, the ALU, the MUXes and in-out ports are 16-bits wide. The red lines (which we have not yet discussed) come from the Control Unit, and these are single-bit signals, or in the case of **selALUf** a four-bit signal). Later!

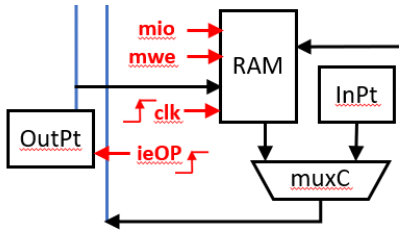


Figure 5 Input and Output Ports.

14.4 The Instruction Set

Now let's turn to the set of instructions we need to actually make the above hardware design really work. Remember I mentioned that hardware and the instruction set need to be designed together, that's why they are referred to as the *Instruction Set Architecture* (ISA). Also I said that every hardware component is 16-bits wide. This means that the *instructions* must be encoded with 16-bits,since they will end up in the **IR** which is 16-bits wide.

Let's take a simple computation, we wish to add two numbers in data segment memory, add them and write the result back into data segment memory. Assume that one number is at address *addr1*, the other is at *addr2* and the sum must be written at *addr3*. What instructions do we need? Well, we need to get these numbers out of memory. So we could start with this instruction

load accumulator from memory	**LDA *addr1***

which puts the data at *addr1* into the accumulator **ACC** ready to feed the **ALU**. We could follow this with the instruction

add with data from memory	**ADD *addr2***

which gets the data at *addr2* and loads it into the right input of the **ALU** and does the addition. As discussed above, the result would be sent to the accumulator **ACC**. So far, so good.

Now we need to store the value in the accumulator back into data segment memory at address *addr3*. We need an instruction like this,

store accumulator to memory	**STO *addr3***

Any other arithmetic operations such as **SUB *addr*** will work just like the **ADD *addr*.** Note that all of these instructions have two parts, the name or mnemonic (e.g., LDA) followed by the address in memory ***addr*.** Other instructions we shall need could be an input instruction,

input into accumulator	**IN**

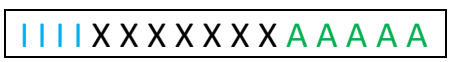
and an output instruction

output from accumulator	**OUT**

Other instructions to give us the capability to create loops and also if-then-else selections will be discussed later. For the moment, it is interesting to note the central place of the accumulator **ACC** in our design.

14.5 Encoding Instructions

In the above section, we have introduced some fundamental instructions, and we have given these 'human-readable' names such as **LDA *addr***. That's fine, we humans need that, but the CPU operates on signals and so we have to *encode* these human-readable forms into a 16-bit equivalent which can move, as signals, along the CPU's wires. Here's how we do the encoding of the instruction into 16 bits



The 4 bits on the left (blue) encode the instruction mnemonic, the 5 bits on the right encode the address, and the remainder

of the bits (Xs) are not used. Here X means ‘don’t care’. Let’s take a couple of examples.

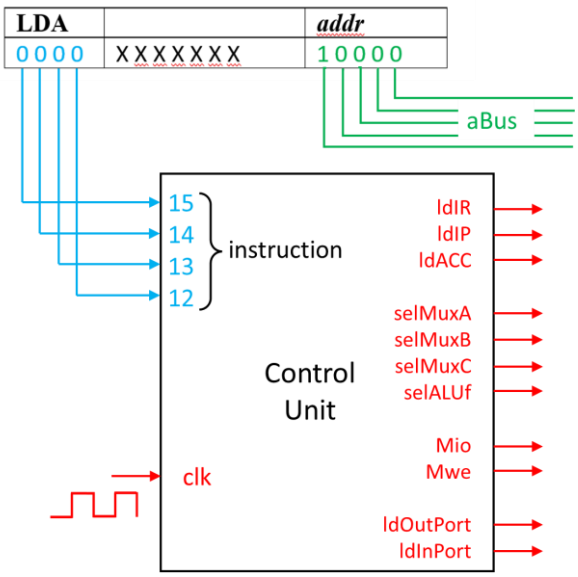
First the **LDA** instruction, and say we wish to load the accumulator with data in memory at address 16 (dec). Here’s the encoding for our CPU2S,

LDA		***addr***
0 0 0 0	X X X X X X X	1 0 0 0 0

and here’s the encoding for the instruction to store the accumulator into memory at address 17 (dec)

STO		***addr***
0 0 0 1	X X X X X X X	1 0 0 0 1

The above diagrams show what is inside the *Instruction Register* **IR**. These bits come out the register as signals on 16 wires. So what happens to them? Look at the diagram below.



The highest 4 bits which encode the instruction are passed into the Control Unit which outputs the control signals (red) which excite various components such as the muxes. The

lowest 5 bits are output to the address bus and make their way to memory as you understand.

14.6 Hardware Components

In this section we'll see how each of the hardware components that make up CPU2S function.

14.6.1 The Registers

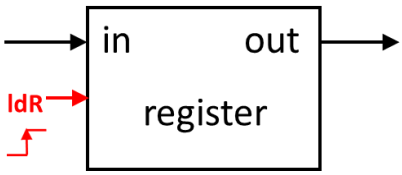
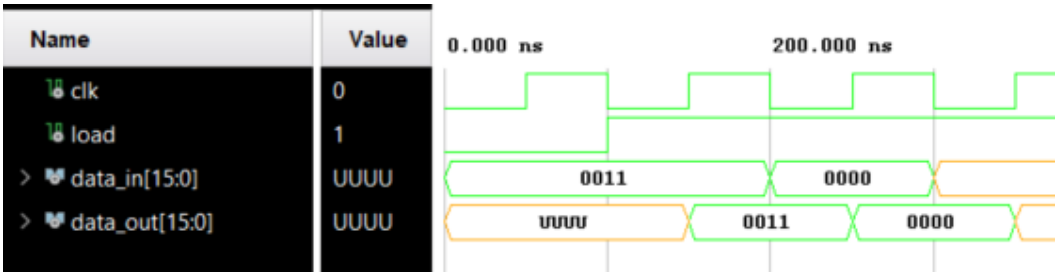


Figure 6 Register.

We have three registers in our design, the **IP, IR**, and **ACC**. They all have the same *behaviour* (in the VHDL sense too!). The structure is shown in Fig.6, you will recognize this as a D flip-flop (it's actually 16 flip-flops, one for each bit). There's an input and an output and a signal **ldR,** 'load register', when this is 1 the input is clocked into the register on the next **clk** rising edge. You can see this on the timing diagram below. Here's a Vivaldo testbench waveform.



Look at the rising **clk** edges. At the first the value of **load** is '0', so data_in is not passed through to data_out. On the second rising edge, **load** is '1' so the data_in '0011' appears at data_out. It stays there until data_in changes, **load** is '1' and there is a rising clk edge.

14.6.2 The ALU

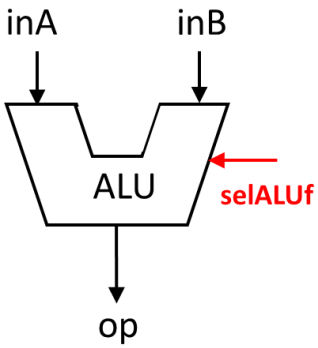


Figure 7 ALU showing inputs and output, and the function select signal (red)

Here we have two inputs **inA** and **inB** and a single output **op**. There is also a control input, shows as usual in red, which comes from the Control Unit. This is a 3-bit signal and the ALU performs different operations based on the states of these 3 bits, see Fig.7. Here are some examples of ALU operations which we summarize using VHDL notation.

sel_ALUf bit pattern	operation
"001"	op <= inB
"010"	op <= (inB + 1)
"011"	op <= (in A + inB)
"100"	op <= (inA – inB)

The third and fourth operations are the expected addition and subtraction, But what about the first two which look a little strange. The first one is a 'pass through' where the input passes through unchanged. This is used in using the ALU as part of the data path. The second instruction adds 1 to **inB**. This is used to increment the *instruction pointer* **IP** as the instructions are fetched in sequence. The Vivado testbench waveform shown below has been designed to test all four operations.

You can see from the **Value** column that **inA=0003** and **inB=0002**. The first input to **sel_ALUf** is **U** which means 'undefined'. The next is **1** and we can see that **inB** is passed through to the **op** which is correct; we are in 'pass through' mode. The next **sel_ALUf** input is **2** and the output is **inB+1** which, again is correct. Then comes **sel_ALUf = 3** the output is the addition of **inA** and **inB**. Finally, **sel_ALUf** is **4** which is **inB** subtracted from **inA**. So that's how the ALU works!

14.6.3 The Memory – RAM

Memory is composed of a lot of 16-bit wide cells where data is stored. To access any cell, you need its address, and of course there are two operations you can perform on memory, read and write. Fig. 8 shows the signals in and out of RAM.

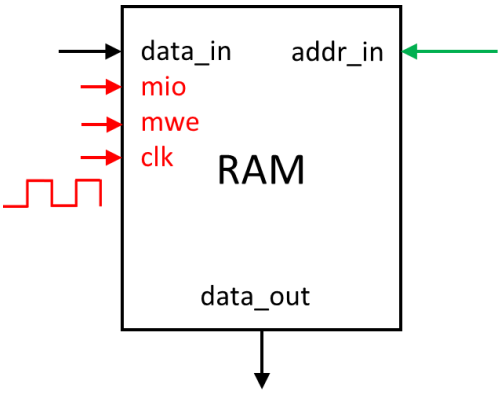


Figure 8 The structure of memory with address, data and control signals.

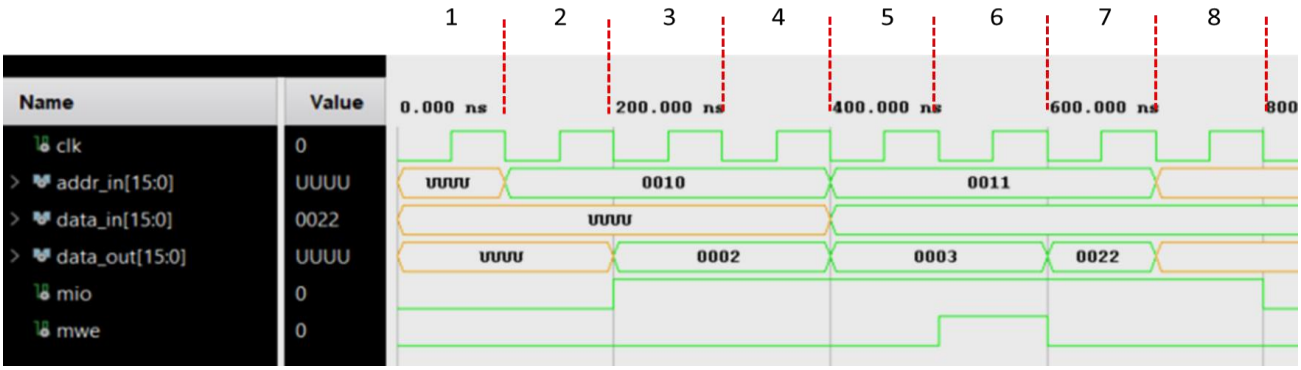
There is **data_in** and **data_out** and of course the address **addr_in.** Now to the control signals. When memory is being accessed, we must have **mio = '1'**. If it is **'0'** then the input-output ports are being used. The signal **mwe** stands for 'memory write enabe', so if **mwe = '0** then we are reading and if **mwe = '1'** then we are writing. Finally, we have **clk**. This is only used when we are writing to memory.

Memory is organized into two segment, the *code* segment where our instructions are stored and the *data* segment where data is being stored. The code segment starts at address 0, and the data segment starts at address 16) dec. Here's the data segment we shall use

```
44      -- data segment
45      X"0002", X"0003", X"0004", X"0005",
46      X"0001", X"0001", X"0002", X"0006",
47      X"0000", X"0000", X"0000", X"0000",
48      X"0000", X"0000", X"0000", X"0000"
49      );
```

These numbers are shown in VHDL hex format, the first row starting at address 16(dec) contains the numbers 2, 3, 4, 5.

A testbench waveform to show the memory at work is shown below where we have numbered the clock cycles to make things clear, see below.



So in clk cycle-1 and clk cycle-2 **mio** is low, so there is no memory activity requested, **data-out** is undefined, you can see the orange UUUU. Also, **data_in** is undefined since there is no incoming data. But at the start of clk cycle-2 the address 0010 (hex) 16(dec) appears on **addr**_in. At the start of clk cyle-3 **mio** is high, so memory is active and **mwe** is low, which indicates a *read from memory*. The **addr_in** bus signal is address 0010 (hex) = 16 (dec), so memory will be read from this address. The above *data segment* shows the data at this address is 2, so this appears on **data_out** as 0002 (hex) during clk cycles 3 and 4. Now look at clk cycle-6, here **mwe** is high which means data is being written into memory, and the address 0011(hex) 17(dec) is on the address bus, so this is where the data will be written. This data has value 0022(hex). Finally, at clk clycle 7 **mwe** drops low, so the memory is being read, and you can see that the data just written 0022(hex) appears on **data_out**. So this shows that the memory write instruction was correctly processed.

14.6.4 The MUX

A multiplexer is a switch, ours have two inputs and one output, see Fig.9. The input selected is defined by the MUX control signal **selMux** (which comes from the Control Unit). The operation is really straightforward, when **selMux** is '1' then **inA** is passed though to the output **op** and conversely when **selMux** is '0' then **inB** is passed through to **op**. You can see this happening in the testbench waveform below, where **inA = "0003"** and **inB = "0005"**. When **sel** is high the 3 gets through to the output and when **sel** is low the 5 gets through to the output.

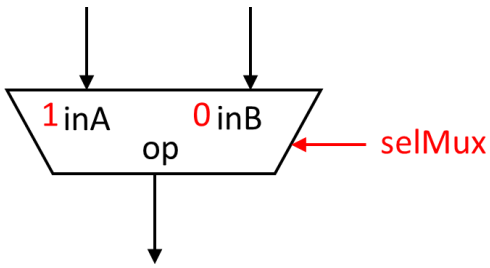
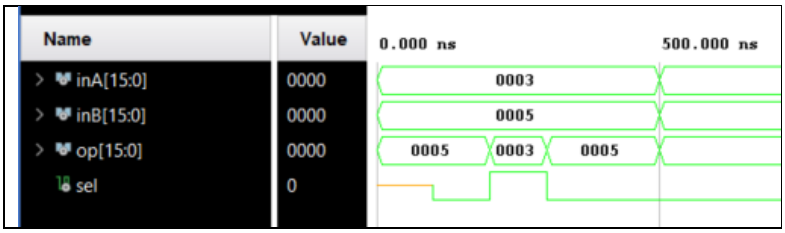


Figure 9 The MUX with two inputs, one output and the selection signal



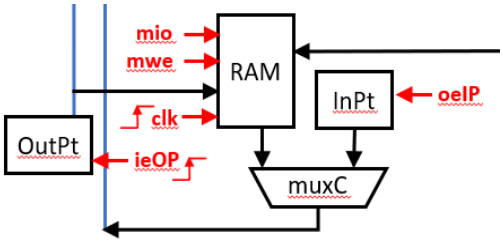


Figure 10 The Input and Output Ports.

14.6.5 The Input Port

This port (Fig.10) takes digital signals from the real world, e.g., from switches or buttons and passes them onto **dBus1** so the value can be loaded into the accumulator **ACC**. Of course, the RAM needs to communicate with **ACC** so there must be a selector between the input port and the RAM. This is the purpose of **muxC.** The behaviour of the Input Port is quite simple, the real world input signals are sent to the output of the Input Port when the signal **oeIP** (which means 'output enable of the Input Port") is '1' else nothing gets through.

14.6.6 The Output Port

This port is a simple register fed by **dBus2**. When its control signal **ieOP** is high on a rising clock edge, then the value of **dBus2** is passed through, and could, for example, illuminate some LEDs.

14.6.7 The Control Unit

This looks like a complex and hungry beast, perhaps the hardest part of the CPU to understand. Its job is to take the current instruction being executed, and to set all the control signals (the red arrows) in order to make the execution of the instruction to actually happen. So it's a logical input-output device which can be summarized as a truth table, therefore it's actually very simple to understand. Here's the truth table for our controller, we have only shown when outputs (red) are high, unless a low value is important.

opcode	Instr	mio	mwe	ldIR	ldIP	ldACC	selMA	selMB	seMC	selALU	ldACC	ldOut	oeIn
1000	IN	0	0			1	1	1	1	001			1
1001	OUT	0	0								1		
0000	LDA	1					1	1		001	1		
0001	STO	1	1				1						0
0010	ADD	1					1	1		011	1		
0011	SUB	1					1	1		100	1		
0100	JMP	1		1	1		1			010			

You can see that various outputs agree with the components we have presented so far, e,g, the IN and OUT instructions

set **mio** low since these are not concerned with memory. All other instructions are concerned with memory, so **mio** is high. You can see how the control signal sent to the ALU to select its function is correct, the values "011" and "100" agree with our discussion of the ALU. Other signals require a more in-depth discussion of CPU2S, which might appear in the future.

14.7 An Example Complete Program

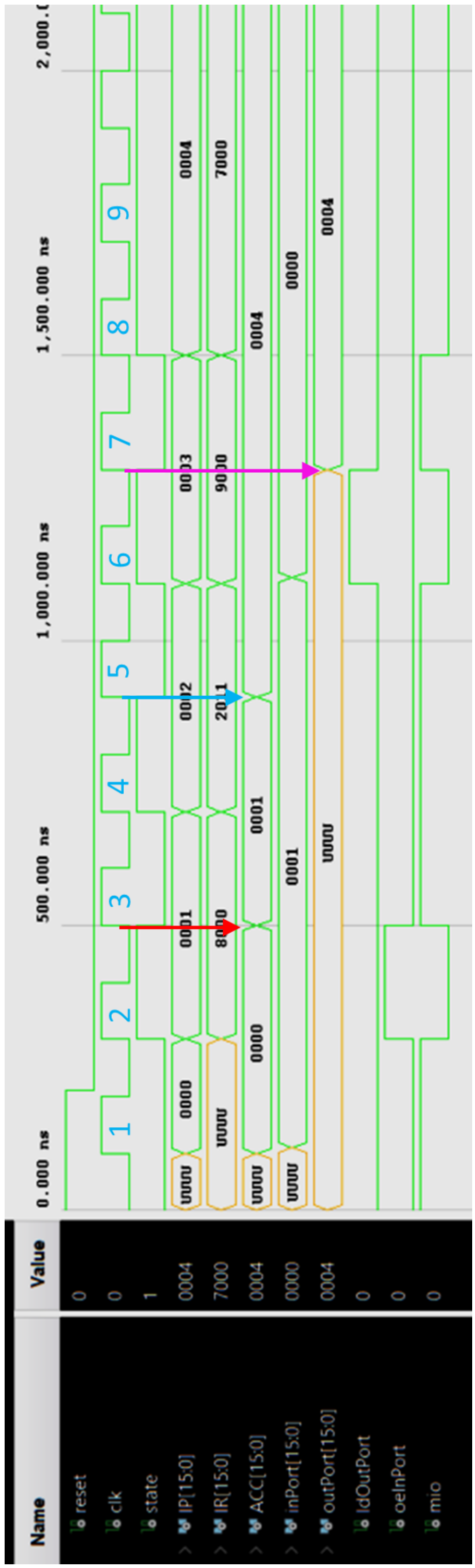
Let's have a look at a simple program that gets a number from the input port into the accumulator, then adds a number from memory to this, then outputs the result, in the accumulator, to the output port. Note the central position of the accumulator in all of this. Here's an annotated code snippet from the code segment, where the address 10 is of course hex which is 17 (dec).

```
27      -- code segment
28      X"8000",        IN
29      X"2011",        ADD 11
30      X"9000",        OUT
31      X"7000",        HLT
```

Also, here's the data segment where you can see the number to be added is 3.

```
44      -- data segment
45      X"0002", X"0003", X"0004", X"0005",
46      X"0001", X"0001", X"0002", X"0006",
47      X"0000", X"0000", X"0000", X"0000",
48      X"0000", X"0000", X"0000", X"0000"
```

The timing diagram resulting from execution of this program is shown below where some features have been highlighted, for example we have labelled the clock cycles.


Name
Value
reset
clk
state
IP[15:0]
IR[15:0]
ACC[15:0]
inPort[15:0]
outPort[15:0]
ldOutPort
oeInPort
mio
0.000 ns
500.000 ns
1,000.000 ns
1,500.000 ns
2,000.0
1
2
3
4
5
6
7
8
9
0004
7000
0000
0004
0000
0004

You can see the instruction pointer **IP** marching up 0, 1, 2, 3, 4 as our four instructions are executed in sequence. You can also see the instruction register **IR** contains our four instructions in turn: 8000 (IN), 2011 (ADD 10), 9000 (OUT) and 7000 (HLT).

Perhaps the most important point is to trace the data flow as the program executes, and we remember that the accumulator plays a pivotal role. Look at the **inPort** from clock cycle 1 to 6, it contains the input value 1. Now at the rising edge of clock cycle 3 this value is loaded into the accumulator **ACC** (red arrow), since the IN instruction (8000) is being executed at this time. Next by the rising edge of clock cycle 5 (blue arrow) the addition has been performed (the ADD 10 instruction is in the **IR** at this time) , and *at* this rising edge, the added number appears in **ACC**. Finally, during the output instruction (9000) the result in the accumulator is sent to the **outport** at the rising edge of clock cycle 7 (purple arrow). The next instruction is 7000 HLT, so we are all done.

14.8 Coda

I hope you have learned something from this chapter and the associated worksheet activities. I really hope this has left you with a feeling of awe for those electronic engineers who design our CPUs, that they synthesize electronic circuits where each component has its place and function. Of course they have a vision of the *whole* which is more than the *sum of its parts* (whatever that might mean), and their electronic circuits are designed with a particular Instruction Set in mind. To end where we started, there is a load of architecture going on here.