**PRACTICAL Week 2: The CPU-OS SIMULATOR, part 1**

**Objectives**

This lab introduces the CPU-OS simulator and some simple assembly language programs. These programs are used to show the Fetch-Decode-Execute cycle and the use of RAM to store variables. It uses the program: “*loopcount.sas*”

**Introduction**

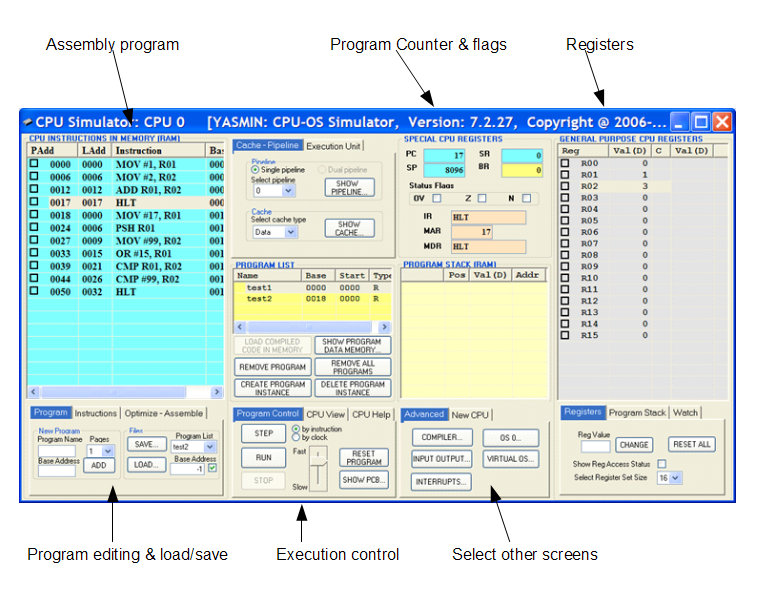
The CPU-OS simulator is an integrated simulator for a processor (Central Processing Unit: CPU) and operating system (OS) that allows you to experiment with a wide range of hardware & software features. The processor is not based on any specific processor. It has some features that are RISC like (multiple identical registers) and some features that are CISC like (such as variable length instructions). We will have a separate lecture on RISC vs. CISC systems later in the module. The simulator comes complete with a high level language (HLL) compiler, so it is possible to write either high level or assembly level programs. The compiler converts programs to the assembly language level so that you can see how the compiler works. As with the processor, the HLL is not a particular language: it’s a fairly generic C-like language. It has standard loops, subroutines and so on, as well as some support for OS type features.

The simulator was developed at Edge Hill University with the help of funding from the HEA. It is freely available under a creative commons licence. <http://www.teach-sim.com/>.

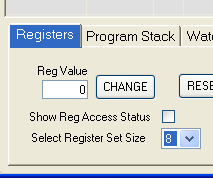
On Moodle, under Practical2, download, install and then run the CPU-OS on your computer. You also need to download “loopcount.sas” program located inside CPU-OS Instructions folder.

**The simulator screen**

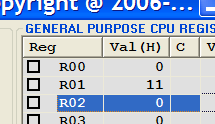
The main screen has a number of sections. The left hand side shows the current program in assembly language. The general purpose registers are shown on the right. Special purpose registers, such as the Program Counter (PC), Stack Pointer (SP) and Status Flags (Overflow, Zero & Negative) are shown in between. Most features are explained by hovering the cursor over them.



An interesting feature of the simulator is that you can select the number of general purpose registers there are: the 'Register Set Size' (from 8 to 64). I usually keep it at 8 or 16 to keep the screen from becoming too cluttered.

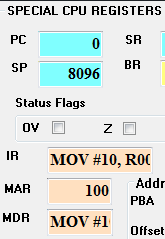


The registers hold 32 bits each. You can select whether the contents are displayed in binary (B), decimal (D) or hex (H) by clicking on the top of the Val column.



**Special Registers**

As with most processor, there are some special purpose registers. **PC**, the Program Counter, tracks where the processor has got to in a program. The Flags indicate status: has there been an overflow (OV)? Is the result zero (Z)? **IR** is the Instruction Register that holds instructions while they are being decoded. **MAR** holds addresses being sent out from the processor, **MDR** holds data coming into the processor. **SP** is the Stack Pointer: a register used to locate the stack, a temporary storage location in RAM.

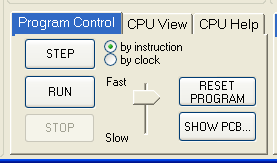


**Entering Programs**

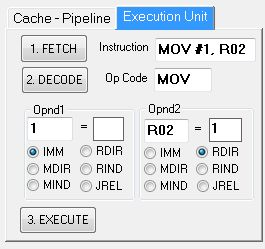
Assembly language and high level programs are entered in different ways. Assembly level programs are entered in a slightly idiosyncratic way: you select them from a menu one at a time. This is a bit tedious, but you will only ever be doing this for short programs. You can save programs and load them again. The high level language system is a bit more intuitive in that you type in programs with an editor which allows you to cut and paste from other documents. If you have done any programming in the past, this should seem fairly natural.

**Running programs**

The simulator can run in a variety of modes. The most obvious way to run it is to use the RUN or STEP buttons to execute the code normally one instruction at a time. If you RUN the code, then you can select the speed it runs at using the slider. This can be useful if you are visualising some aspects (such as the contents of registers) as the program executes. The line that is about to be executed is highlighted in grey.

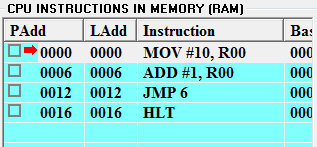


If you want to see the detailed sequence of operations, then you can also select the execution to go by clock rather than by instruction. This means that the instructions are broken down into the different stages (fetching the code, decoding and execute the code). This can be useful if you are looking at details of how instructions move through the instruction pipeline for instance. If you do that, the fetch.. decode.. execute sequence can be controlled directly by using specific buttons for these actions. These buttons are on the ‘Execution unit’ tab which is not shown by default. This bit of the simulator is a bit buggy: you don’t always get the correct part of the sequence displayed.



**A simple program**

Click the LOAD button. The button is located near the bottom left. Load the program “loopcount.sas” that you downloaded from Moodle. You should see this 4 line program. The Physical Address (PAdd) where each instruction is stored is shown on the left (LAdd is the Logical address and used by the operating system.) You can see what each line of the program does by clicking on the STEP button. If you try STEPping through the program you will see a number of things. The main section of the program is a simple loop that repeats for ever. The program starts with the number 10 being put into the general purpose register R0, then each time round the loop, one is added to the register. You can use the RESET PROGRAM button to go back to the beginning at any stage.



Check you understand what is going on by answering the following questions

1. What does this simple program do?

Your answer:

**It adds 10 to R00 and then keeps adding one ad infinitum, could be used as a counter for how long it takes to execute the loop???**

1. The next instruction to be executed is highlighted in grey. What register holds the address of this instruction?

Your answer:

**the IR**

What length is each instruction, i.e. what is the difference in the address of each instruction?

Your answer:

**6,6,4**

1. There is a ‘Halt’ instruction at the end of the program: HLT. Is it ever executed?

Your answer:

**No, since the loop never terminates**

1. Given the answer to the previous question, why is it that the HLT instruction is ever highlighted?

Your answer:

**It’s being decoded**

1. Normally we will be able to see what is going on more clearly by using the STEP button, but you should also try using the RUN button to let the program execute normally. Experiment with the Fast…Slow slider to see the range of speeds that are available. If you leave the program running, where do you see the loop count incrementing?

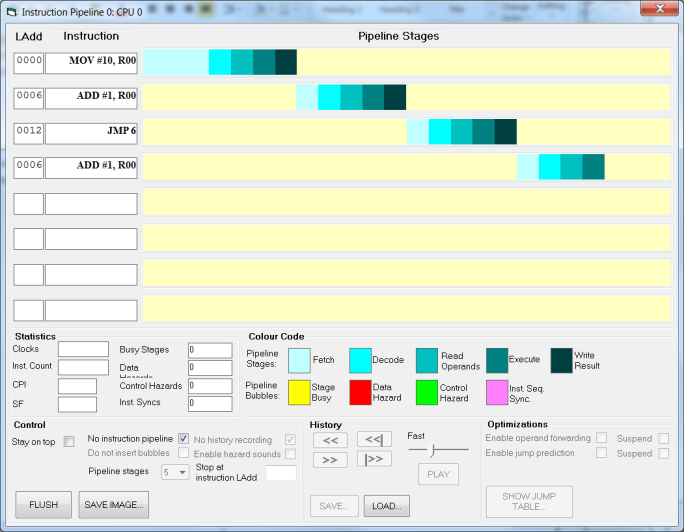
Your answer:

**In the value of R00**

**THE INSTRUCTION PIPELINE: FETCH, DECODE, EXECUTE**

Normally when you are running through a program, the instructions are executed with the **by instruction** box selected (next to the STEP button). In order to be able to see the workings of an instruction pipeline, you need to select the **by single tick** instead. Now each press on the STEP button is equivalent to a single clock, rather than a whole instruction. If you try it out, you will see that a typical instruction on this processor takes about 4 or 5 clocks to execute.

A clock cycle, or simply a "cycle," is a single electronic pulse of a CPU. During each cycle, a CPU can perform a basic operation such as fetching an instruction, accessing memory, or writing data. The frequency of a processor is measured in clock cycles per second



If you now select the **SHOW PIPELINE** button near the top middle you will get another window showing the instructions as they run.

Try running through a few instructions: you will see how they are colour coded to show the stages as they run: 1 Fetch, 2 Decode, 3 Read operands, 4 Execute, 5 Write result. If you tick the **Stay on top** box on the bottom left of the pipeline window, you will be able to see the pipeline while you are clicking at the main window to step. The default view, as seen here, is for there to be no instruction pipeline: one instruction is not even fetched until the preceding one is finished. Later on we will look at how adding an instruction pipeline affects things.

On the main simulator screen, press the **RESET PROGRAM** button to start again. Now we want to look at some of the special registers. Making sure that execution **by single tick** is still selected, step through the program by pressing **STEP**. You will see the contents of MAR and MDR change in the main window.

1. What is the significance of the number that appears in MAR?

Your answer:

**It’s the address of the data being sent out from the processor**

1. What is the significance of the contents of MDR?

Your answer:

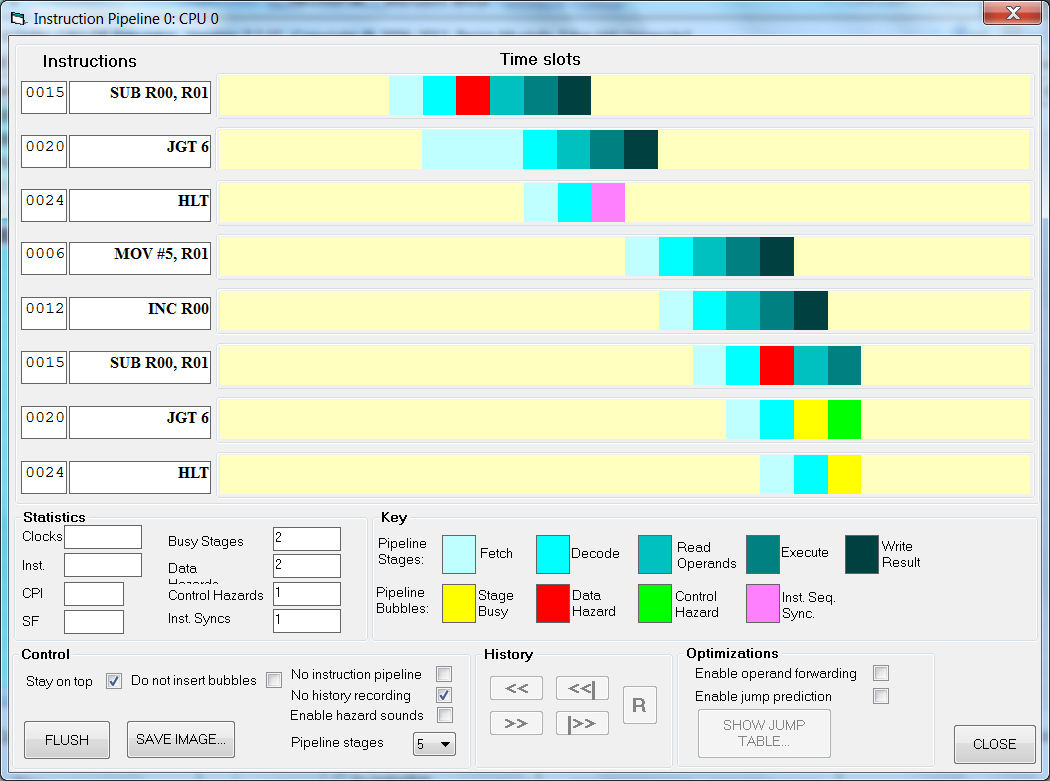
Holds the data (in this case the instructions) being sent out

1. On a real processor, the contents of MDR would not be letters. What would it be?

Your answer:

binary? I wanted to say blocks but uh

In current processors, it is common to add hardware so that several stages of an instruction can be worked on at the same time. So, the second instruction could be fetched at the same time as the first is being decoded and so on. The simulator can show you this. Press the **RESET PROGRAM** button to start again. Clear the tick from the **No instruction pipeline** box in the instruction pipeline window. Now as you step through the program you will see how the processor speeds up the program by working on multiple instructions at any one time.



If you compare this picture with the earlier one, you will see that instructions are being finished more quickly. However, things don’t always go smoothly. There are times when the processor has to pause. Perhaps the data isn’t ready (the previous instruction may not have stored the result of its calculation). Or perhaps the processor can’t tell which instruction to fetch next because it needs to calculate that first (e.g. a conditional jump). These are examples of ‘hazards’. In this case a data and a control hazard respectively.

Hazards are only delays; they don’t cause errors or data loss. As they do slow things down a bit, sophisticated processors use techniques to reduce the delays. Perhaps you won’t to fetch an instruction but there is a conditional jump and you won’t know until the calculation is done which way to go. A Pentium will fetch instructions from both possible branches, and then only use the one that the calculation ends up going to.