WinDLX Tutorial
A first example

Contents

Introduction

The DLX processor (pronounced "DeLuXe") is a pipelined processor used as an example in J. Hennessy's and D. Patterson's **Computer Architecture - A quantitative approach**. This tutorial describes a session using WinDLX, a Windows-based simulator, that shows how DLX's pipeline works.

The example used in this tutorial is very simple and is not meant to show all aspects of WinDLX. It should act only as a first introduction to the use of the application. When you have completed it, please refer to the help files; you can at every stage of a session get context-sensitive help by pressing F1. During this example, though, this will probably not be necessary.

Though every step of the example will be discussed in detail, basic knowledge in the use of Windows must be required. It must be assumed that you know how to start Windows, scroll using scrollbars, execute a double click or bring a window uppermost on the screen. The exact appearance of your screen cannot be foretold (e. g. Is a special icon covered by a window or not?), so you must be able to "tidy up" your screen without help.

You will need Windows 3.0 or higher for this simulation.

Installation

WinDLX consists of the files **windlx.exe** and **windlx.hlp**. Together with these you should have got some assembler code files with the extension **.s**. In this manual **fact.s** and **input.s** will be needed.

If you are familiar with the installation of Windows applications, you might as well skip now to the next chapter, A complete example, after making sure that **fact.s** and **input.s** are copied into the WinDLX directory.

To install WinDLX to Windows 3.1, please execute the following steps:

1. Create a directory for WinDLX, e. g. **C:\WINDLX.**
2. Copy all the WinDLX files you have got, at least **windlx.exe, windlx.hlp, fact.s** and **input.s** to the WinDLX directory.
3. If you have not already done this, enter Windows now.
4. Assuming that you use the German version of Windows, double click on ***Windows Setup*** in "Hauptgruppe".
5. Select ***Optionen*** and ***Anwendungsprogramme einrichten***.
6. Select ***Sie ein Anwendungsprogramm angeben lassen***, click ***OK*** and enter the WinDLX directory and the filename, e. g. **C:\WINDLX\WINDLX.EXE**.

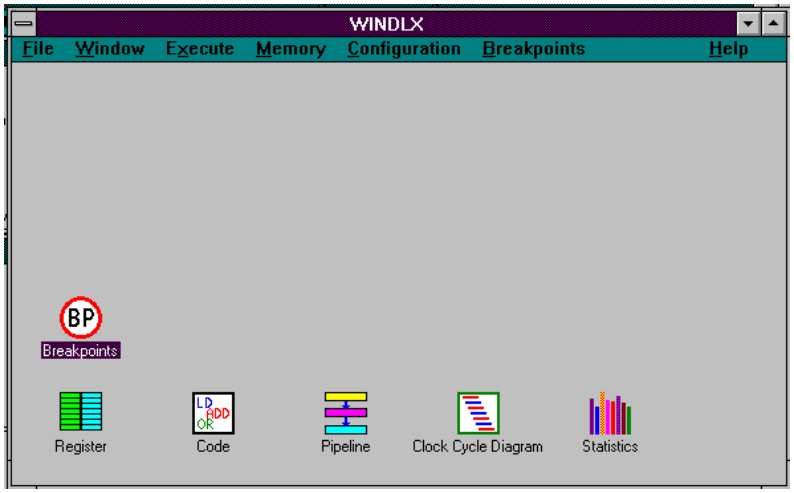
Windows will then automatically install WinDLX to the group "Anwendungen"; the icon looks like this:

A complete example

This chapter uses the assembler file **fact.s** in WinDLX assembler. The program calculates the factorial of a number you can enter on the keyboard. The file **input.s** will be required for this, too.

Starting and configuring WinDLX

WinDLX is started - like every Windows application - by double clicking on the WinDLX icon. A window (denoted main window in the future) with six icons appears. Double clicking on these icons will pop up child windows. Each of these windows will be explained and used later.



To make sure the simulation is reset, click on the ***File*** menu and click ***Reset all***. A window pops up and you will have to confirm your intention by clicking the ***OK*** button in the "Reset DLX" window.

WinDLX is capable of working with several configurations. You can change the structure and time requirements of the pipeline, the memory size and several parameters that control the simulation. Let us choose the standard settings; click ***Configuration*** / ***Floating Point Stages*** (read that as: click ***Configuration*** to open the menu, then click on ***Floating Point Stages***) and make sure that the following settings are given:

	Count	Delay
Addition Units:	1	2
Multiplication Units:	1	5
Division Units:	1	19

If necessary, change the settings by clicking in the appropriate field and editing the given numbers. When you are finished, click ***OK*** to return to the main window.

By clicking ***Configuration*** / ***Memory Size*** the size of the simulated processor's memory can be set. This should be 0x8000. Again, ***OK*** goes back to the main window.

Three more options in the ***Configuration*** menu can be chosen: **Symbolic addresses**, **Absolute Cycle Count** and **Enable Forwarding** should all be set, that is, a small hook should be shown beside it. If this is not the case, click on the option.

Loading testprograms

In order to be able to start the simulation, at least one program must be loaded into the main memory. To accomplish this, select ***File / Load Code or Data***. A list of assembler programs in the directory appears in a window.

As mentioned earlier, **fact.s** calculates the factorial of an integer number. **input.s** contains a subprogram which reads the standard input (the keyboard) and stores the integer in the general purpose register 1 of the DLX processor. To load these two files into the memory, do the following:

- click on **fact.s**
- click the ***select*** button
- click on **input.s**
- click the ***select*** button
- click the ***load*** button

The sequence of selection of the files is essential as it defines the order of appearance in the memory. Confirm the message **File(s) loaded successfully. Reset DLX?** by clicking ***OK***. The files are now loaded into the memory.

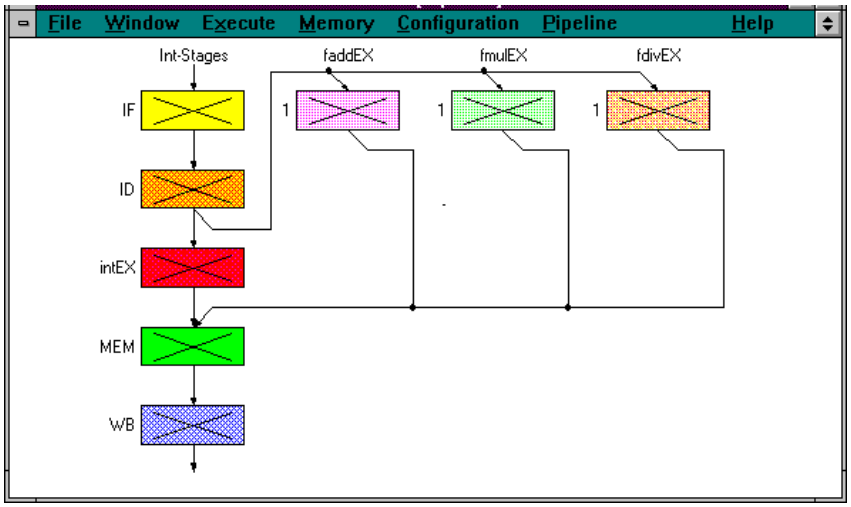
After these preparations the simulation is ready to begin.

Simulating

When looking now at the main window, you should see six icons, named (not necessarily in that order) "Register", "Code", "Pipeline", "Clock Cycle Diagram", "Statistics" and "Breakpoints". Clicking any of these icons will pop up a new window (a "child" window). The characteristics and the use of each of these windows will be introduced during the simulation.

Pipeline window

Let us first take a look at the inner structure of the DLX processor. To do this, double click on the icon ***Pipeline***. The appearing child window shows a schematic representation of DLX' five-stage pipeline. You should enlarge this window as much as possible, so that instructions held in the various pipe stages can be shown in the schematic.



The picture shows the five pipeline stages of the DLX processor and the units for floating point operations (addition / subtraction, multiplication and division).

Code window

The next window we will look at is the ***Code*** window. When double clicking the icon, you will see a three column representation of the memory, showing from the left to the right an address (symbolic or in numbers), a hex number giving the machine code representation of the command and the assembler command.

```
$TEXT          0x20011000          addi r1,r0,0x1000
main+0x4             0x0c00003c          jal InputUnsigned
```

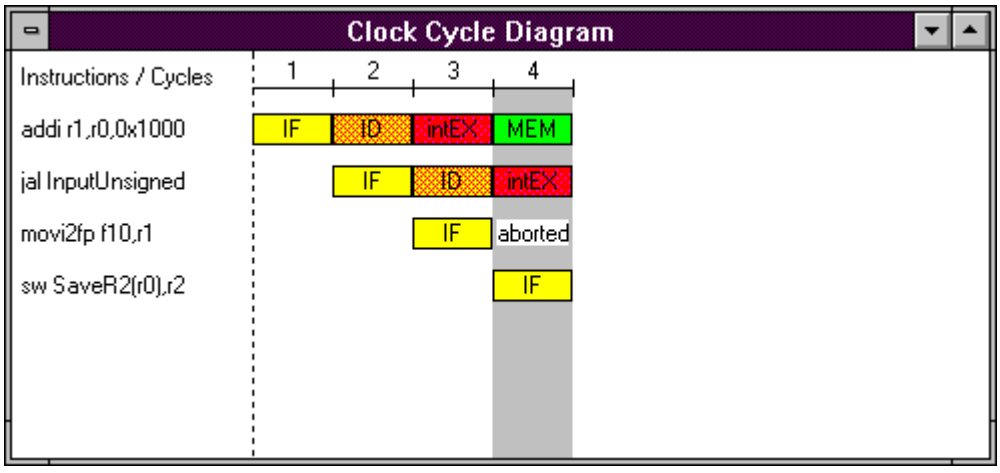
It is time to start the simulation now, so click ***Execution*** in the main window. In the appearing pull down menu, click ***Single Cycle***. Pressing ***F7*** has the same effect.

You will note that the first line in the window with the address $TEXT is now coloured yellow. Pressing ***F7*** advances the simulation for one time step; this changes the first line's colour to orange and the next line is coloured yellow. These colours show the pipeline stage the command is in. If you have closed the pipeline window, please re-open ***Pipeline*** again (double click on the icon). If the window is large enough, you can see that the command **jal InputUnsigned** is in the IF stage and the preceding command **addi r1, r0, 0x1000** is in the second stage, ID. The other blocks are marked with a cross, showing that no sensible information is processed in them.

Pressing ***F7*** again will re-arrange the colours in the code window, introducing red for the third pipeline stage intEX. The next ***F7***, however, will change the picture: the yellow line appears farther down and is probably now the only coloured line in the code window. Examining the pipeline window will show that IF, intEX and MEM are used but ID is not. Why?

Clock Cycle Diagram window

Another window will show further information. Iconize all child windows and open the ***Clock Cycle Diagram*** window. It contains a representation of the timing behaviour of the pipeline.



You can see that the simulation is now in the 4th cycle, the first command is in the MEM stage, the second in intEX and the fourth in IF. The third command, however, is denoted as "aborted". The reason for this: The second command, jal, is an unconditional branch. This fact is known only after the 3rd cycle, when jal has been decoded. During this cycle the command movi2fp (following after jal) has already been fetched, but the next executed command will be at another address. Therefore the execution of movi2fp must be aborted, leaving a "bubble" in the pipeline.

The branch address of jal is named "InputUnsigned". To find out the actual value of this symbolic address, click ***Memory*** in the main window and ***Symbols***. The appearing window shows the correspondence between the used symbols and the actual numbers. Select "name" in the "Sort:" area to have them sorted by name rather than by value. "G" after the value denotes a global, "L" a local symbol. "InputUnsigned" in the module "input" therefore is a global symbol standing for 0x144 and is used as an address. Please close the window now by clicking on the ***OK*** button.

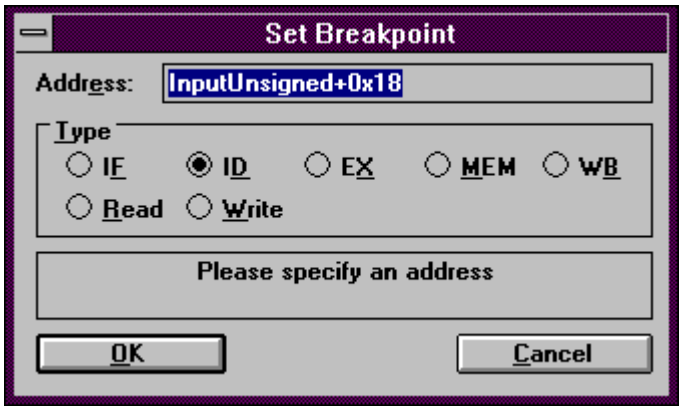
Pressing ***F7*** once more will bring the first command, addi, into the last pipeline stage. What has internally happened to execute this command can be examined by pointing to the line to be examined in the clock cycle diagram (the line containing the addi-command) and double clicking. A new window will pop up that contains a detailed description of the processor's internal actions for every pipeline stage. The window is denoted "Information about ..." referred to as the "information window" in the future. After having examined it, close the window by clicking the ***OK*** button. Double clicking on the third line, movi2fp, shows that only the first pipeline stage, IF, has been exectued and then the command was aborted due to a jump. Do not forget to click ***OK***.

(The information window can be brought up double clicking on a line in the code window or a stage in the pipeline window, too.)

Breakpoint window

When examining the code by opening the code window (double click on icon ***code*** if it is not already opened) you will notice that the next instructions are all nearly the same; they are **sw**-operations that store words from a register into the memory. Repeatedly pressing ***F7*** would be quite boring, so we will speed this up by using a breakpoint.

Please point now to the line 0x0000015c in the code window that contains the command **trap 0x5**. This is a system call to write to the screen. Click once (this will reverse the line) and click on ***Code*** in the main windows menu line (to do this, the code window must be uppermost on the screen). Select ***Set Breakpoint*** by clicking on it (make sure the line is still marked!). A new window "Set Breakpoint" pops up to let you decide what pipeline stage of the command shall be reached before execution of the program stops. This is ID by default. We will leave it at that; click ***OK*** to close the window.



Now in the **trap 0x5**-line in the code window, "BID" appears, showing that a break from program execution will occur when this command is in the decode phase.

To examine the defined breakpoints click on the icon ***Breakpoints***. A small window containing all breakpoints (only one so far) is shown. Re-Iconize the window again.

Now let the simulation run by clicking ***Execution*** / ***Run*** or simply ***F5***. A window will inform you that "ID-Stage: reached at Breakpoint #1"; it is closed by clicking ***OK***.

If you bring the clock cycle diagram window to the foreground by clicking on it, you will note something new: The simulation is now in cycle 14, but the line **trap 0x5** looks like

The reason for this is that the pipeline is cleared in DLX whenever a trap-instruction is found to avoid all possibility of problems. This is documented in the information window (double click on the trap-line to bring it up) with the note "3 stall(s) because of Trap-Pipeline-Clearing!" in the IF stage. (Do not forget to close the window again by clicking ***OK***.)

The instruction **trap 0x5** has already written to the screen. You can check this by clicking on ***Execute*** / ***Display DLX-I/O*** in the main window's menu line. The created window shows you the screen's appearance. As usual, ***OK*** will remove the window.

Register window

To go further in the simulation, click on the code window to bring it uppermost on the screen and scroll down (using the arrow keys or the mouse on the vertical scrollbar) to the line with the address 0x00000194, with the instruction **lw r2, SaveR2(r0)**. Set a breakpoint on this line (click on the line; press ***Ins*** as a shortcut or click on ***Code*** / ***Set Breakpoint*** / ***OK***). Use the same procedure to set a breakpoint on line 0x000001a4 **jar r31**. Pressing ***F5*** now to run the simulation further will bring a surprise: The DLX-Standard-I/O window pops up with the cursor blinking after "An integer value >1: ". Type in ***20*** and press ***Enter***; the simulation resumes and reaches breakpoint # 2 (***OK***!).

The picture in the clock cycle diagram window (bring it to the foreground by clicking on it) shows something new - red and green arrows between instructions (if you do not see them, scroll up the clock cycle diagram window using the scroll bar until you can examine simulation cycles 52, 53, 54, 55 and 56). Red arrows denote the necessity of a stall; the reason for this stall is explained in the line the arrow points to. In this case, we have R-Stalls, which means stalls due to RAW-hazards (an instruction needs the result of the previous instruction that is not yet known). Green arrows symbolize the use of forwarding, that is the use of a result before it is written back into the target register of the instruction.

Now it is time to examine the registers' contents. To do so, double click the ***Register*** icon in the main window. The register window shows you the values contained in the registers. Look especially at R1 to R5. Running the simulation to the next breakpoint (***F5***, ***OK***) will show that some values are altered. The **lw** instructions do just that: they load values from memory into registers.

If you want to advance the simulation without having to set a breakpoint, there is another possibility. Click on ***Execute*** / ***Multiple Cycles*** or simply press ***F8***. In the newly created window, type ***17*** and press ***Enter***. The simulation advances 17 clock cycles.

Scroll up the clock cycle diagram window until you see instruction cycles 72 to 78 at least. Two floating point operations (multd and subd - multiply/subtract double) each are executed on separate units during the EX stage, but they both need more than one cycle to terminate. Therefore the next instruction after these (j Fact.Loop) can be fetched, decoded and executed, but after that has to stall for one cycle to allow subd to finish its MEM phase.

Statistics window

Now we will examine the last remaining window, the statistics window.

Let the program finish its execution by pressing ***F5***. The message "Trap #0 occurred" (***OK***) shows that the last instruction, **trap 0** has been executed. Trap number 0 is not defined; this instruction is used as an end instruction to ensure termination of the program. Iconize all windows and double click the icon ***Statistics***.

This window provides information about general aspects (e. g. number of simulation cycles), the hardware configuration used in the simulation, stalls and their causes, conditional branches, Load-/Store-instructions, floating point stage instructions and traps. Usually, an absolute count of events and a percentage are given, e. g. "RAW stalls: 17(7.91 % of all Cycles)".

The statistics window is extremely useful to compare the effects of changes in the configuration. We will try this now:

Let us examine the effects of forwarding in the example. Until now, we have used this feature; what would the execution time have been without forwarding?

To accomplish that, note the total number of cycles (215) and stalls (17 RAW, 25 Control, 12 Trap; 54 Total) and close the statistics window; then click on ***Configuration***. To disable forwarding, click on ***Enable Forwarding*** (the hook must vanish). The following "WARNING: OK resets automatically the processor! Disable Forwarding?" should be answered with ***OK***. Remove all breakpoints by opening the breakpoints icon, clicking on the ***Breakpoints*** menu, clicking on ***Delete All*** and confirming by ***OK***. Then you can run the whole simulation at once with ***F5***, ***20 Enter*** and ***OK*** when trap 0 occurred. By re-examining the statistics window, you learn that the number of Control stalls and Trap stalls remained the same, but the number of RAW stalls was now 53 instead of 17, thus increasing the total number of simulation cycles to 236. With this information you can e. g. calculate the speedup gained by forwarding ($236 / 215 = 1.098 \Rightarrow DLX_{forwarded}$ is 9.8 % faster than $DLX_{not\ forwarded}$ with **fact.s**).

Further experiments

This tutorial somewhat hurried through the example out of the necessity to show all important features of WinDLX. The understanding of pipelining in general and the mode of operation of the DLX processor in particular, however, can only come to you if you work through this and other examples in greater detail and in a speed that suits you. You could especially change the configuration to see if an additional floating point adder is useful or if a faster division unit (less instruction cycles) justifies additional cost. Further you can simulate the effects of an optimizing compiler by rearranging lines in the source codes, thus avoiding RAW-stalls.

Refer intensively to ***Help***. You will find many details that could not be answered in this tutorial.

In general: "play" with WinDLX to get a "feeling" for the function of pipelining - WinDLX surely is a means to accomplish that.