STUDENT GUIDE

SESSION 2: RAW hazards in DLX

DLX Pipeline and our tools

DLX pipeline consists of five stages

IF	***Instruction Fetch*** **and PC increment**
ID	***Instruction Decoding:*** **instruction decoding, register read, jump condition evaluation and PC load**
EX	***Execution*****: Perform Operation (ALU), effective addresses calculation(L/S)**
ME	***Memory*****: Memory access (Read or Write for L/S)**
WB	***Write Bank*****: Write results in register Bank**

The following table shows the operations performed at each stage of the pipeline for the different types of instructions.

Stage	**ALU** **Rd ← Reg1 op Reg2**	**Load Store** **Rd ← M(Reg1+Inm)** **M(Reg1+Inm) ← Reg2**	**Jump** **PC ← PC+ Inm**
IF	**RI ← Mem(PC) PC ← PC+4 Rbr ← PC+dato Inm**		
ID	**A ← Reg1** **B ← Reg2 / Dato Inm**	**A ← Reg1** **B ← Inm** **Rtmp ← Reg2**	**if(A.op.0)** **PC ← Rbr (Salto)**
EX	**C ← A op B**	**Rdir ← A + B** **SRdat ← Rtmp**	
ME	**D ← C**	**LRdat ← Mem[Rdir] (Load)** **Mem[Rdir]← SRdat (Store)**	
WB	**Rd ← D**	**Rd ← LRdat (Load)**	

TABLE of operations in the pipeline

Where:

- Inm is the immediate data and Reg1, Reg2 and Rd are the registers named in the instruction.
- Rest (in color) are segmentation and auxiliary registers.
- In the ID phase the contents of the indicated data are collected
- In addition, in each phase the information of the instruction being executed is preserved

This is described in a similar way in the Hennessy-Patterson chapters 5 and 6.

You must be familiar with Data Hazards and be able to identify dependencies of type Read After Write (RAW) and use a new simulator: WinDLXV, in addition to DLXVSim

Part I: Using WinDLXV and Drawing Pipeline Schemes

The teacher will introduce this simulator briefly.

1. Run program 7 (cycle counting) from Session 1. Compare resulting cycles with the numbers you obtained in DLXVSim. How many cycles does the execution of this code take **in each simulator**? Explain why.
2. Write a program that reads and adds 2 numbers and the reads a third number and accumulates the sum. Leave the result in a memory position called result. Declare data as needed. The idea is to maximize stalls in your code due to RAW dependencies.
   a) In WinDLXV select WITHOUT data forwarding in the "Configuracion" Menu. Observe and draw **a cycle-by-cycle pipeline chart for this case.**
   b) Without data forwarding, there is always a stall of 2 cycles for instructions with RAW dependencies. **True or false?**
3. Keep using no data forwarding. Run one of the programs you wrote in Session 1 (select from fibonacci generator, and rest of exercises) observing the pipeline, statistics, memory layout, etc. Explain total cycles given by the Simulator, that is, write the equation as in exercise 7 & 8 session1 Cycles = 3i + 1s + 10(7i+6s) +... where "i" stands for instruction and "s" stands for stall.

Part II: Data forwarding

The hardware technique called "data forwarding" allows mitigating the effect of RAW (Read After Write) dependencies. The EX and MEM stages can feed back the results generated towards the EX stage inputs. DLX always uses this technique in DLXVSim but in WinDLXV it can be activated or deactivated.

In a monocycle stage pipeline, the data forwarding technique allows RAW dependencies between **two consecutive ALU** instructions cause **no stalls**.

2. You should verify this and draw a pipeline chart.

If the RAW dependency is between a **load instruction and a subsequent ALU instruction**, data forwarding allows for a **single stop cycle** in the pipeline. Let's look again at the case of a RAW dependency between a load instruction and an ALU instruction:

lw **r1**, a	IF	ID	EX	***ME***	WB			
add r3,r2,**r1**		IF	ID	stall	***EX***	ME	WB	
instruction X			IF	stall	ID	EX	ME	WB

Despite the data forwarding technique there is still a stop cycle left. Sometimes this can be avoided by reordering the code with an instruction that **does not have dependencies.** For example, Instruction X below saves a cycle stall and the data can be forwarded through the bypass from ME back to EX on time:

lw **r1**, a	IF	ID	EX	***ME***	WB		
instruction X		IF	ID	EX	ME	WB	
add r3,r2,**r1**			IF	ID	***EX***	ME	WB

In addition to dependencies between LW-ALU and ALU-ALU instructions, RAW dependencies also happen among other types of instructions. Thus It is convenient to identify, for each type of instruction, in **which stage** of the pipeline **it produces (W)** data **and where it consumes**

(R) data. This will allow to determine if a producer-consumer dependency will cause a stall in the pipeline and the issue of instructions is stopped. Refer to the description of operations in the pipeline according to the type of instruction on the first page. Next activity below will help you study this point in detail.

For the rest of experiments you should **always select Data Forwarding** in the simulators. In DLXVSim, that is a default and cannot be set off, but in WinDLXV it must be activated (Menu "Configuracion").

3. Make a pipeline chart by hand for each of the following cases. Check the pipeline description in the front page for each instruction type involved. Identify the phases in which there is PRODUCTION and CONSUMPTION and whether there is data forwarding on time or not.

Once this is done, check your predictions with the two simulators.

sub R1,R2,R3 **add R4,R2,R1**
lw R1,n(r0) **add R2,R0,R1**
lw R1,n(r0) **sw n+4(r0),R1**
lw R1,n(r0) **sw n+4(R1),R7**
lw R1,n(r0) **bnez R1,fin**
sub R1,R2,R3 **bnez R1,fin**

Note: LW-SW instructions do not behave the same in way in the two simulators. One of them allows a result to be transferred from the ME output to its input by means of an data forwarding circuit, but this circuit has not really been described in the DLX documentation.

Remember: Always with work with data forwarding ON

4- In the following program, predict and then check how many stall cycles occur, where and why. Document it next to the instruction in which they occur within a comment.

How many cycles does the execution of this code take **in each simulator**? You should get a difference of **4 cycles -Remember why?**

```
   .data 0
a: .word  1,  2,  3,  4
b: .word 17, 18, 19, 20
c: .word 0, 0, 0, 0
   .text  100
ini:
   xor    r7, r7, r7
   addi   r4, r0, b    ; what value is there in r4?
   lw     r1,  4(r4)
   lw     r2,  b(r7)
   add  r3, r1, r2
   sw   0(r4),  r3
   addi r7, r7, #4
   add  r3, r1, r7
    sub     r4,r4,r3
   subi r2, r7, #4
    bnez    r2, ini
    nop
   trap #6
```

Instruction Reordering

Compilers can make changes in the order in which some instructions appear in programs to avoid stalls in the pipeline. For example between a load instruction and an ALU instruction having a RAW data dependency as shown before.

But when you reorder the code you must follow some rules: **you cannot alter the algorithm or the registers used**. Only some immediate data in the loading or storage instructions to adjust calculations of effective addresses can be changed whenever they are affected by the rearrangement.

5- Reorder the code of the previous program to avoid the identified stops. Verify that there are no remaining stalls.

6- Make a "**reverse.s**" program that adds two vectors a and b. The number of elements to add will be specified by a variable **n** and the result must appear in reverse order in vector c. For example, for the following statement of 6 elements:

```
n: .word 6
a: .word  1, 2, 3, 4, 5, 6
b: .word  11,12,13,14,15,16
c: .space 24
```

in c it should appear 22, 20, 18, 16, 14, 12. Remember that you have to put a NOP instruction after the jump (for now).

- You must use the **SLLI** instruction to calculate an index that is 4*n but in the WinDLXV simulator this instruction has problems (it does not move beyond the 5th bit). Then d**o not use WinDLXV for this exercise but only DLXVSim.**
- In the first instance do it **without optimizing**, that is, causing the **maximum** number of stops that may occur. **Analyze** the statistical results and the dependencies that produce stalls in the pipeline.
- Performs a **theoretical** calculation of the cycles that the program will consume, identifying the RAW hazards and number of stalls (0, 1 or 2) next to each instruction in which it occurs. (*) Write the total Cycles equation using "i" for instructions and "s" for

stalls. Then use the simulator to check if you are right. The calculations must match the statistics obtained in DLXVSim.

7- Optimize the previous program, reordering the instructions to MINIMIZE the cycles consumed by the execution (avoid all stalls). The calculations must match the statistics obtained in DLXVSim. Check results and compare with execution without reordering.

a) Obtain the Gain or Acceleration as the ratio between the cycles of either case for n = 10.

b) Recalculate for n = 200.

(*) Handmade pipeline diagram next to each instruction in which there is STALL by RAW dependence showing where it is produced.

- **ATTENTION**: the diagram should not be made for all instructions, **only** for the two involved in the RAW dependence