COMPUTER ARCHITECTURE LABORATORY,

STUDENT GUIDE

SESSION 4: Floating Point in Pipeline Processors

Goal: To analyze the impact that the operation with floating point data produces in a pipeline. Design simple algorithms with this type of data in DLX and optimize them.

READ THIS GUIDE CAREFULLY

Instruction Set Review

As can be found in the different reference documents of the DLX Instruction set (see Practice 1), this processor:

1. It has 32 Floating Point (FP) registers F0 to F31 of 32 bits that can be used as
   - individual Simple Precision registers (SP) F0, F1, F2, ..., F31
   - in pairs as Double Precision registers (64 bits, DP) in which case F0, F2, F4 ... F30 are use to access the pair. We will use DP

   The format for the FP representation is IEEE754.
2. To transfer data from a fixed point register or integer to FP: MOVFP2I, MOVI2FP ("move FP to I(nteger)") and to transfer data between FP records there are: MOVF, MOVD. (We won't use that)
3. To access FP data in memory there are the corresponding Load/Store versions: LF, LD, SF, SD. We will use SD and LD.
4. The FP status record is a special record that is used to evaluate jump conditions. Transfers from/to this record are made to / from GPRs. (We won't use that)

Find below a summary of FP operations in double precision (DP). In bold those we will use:

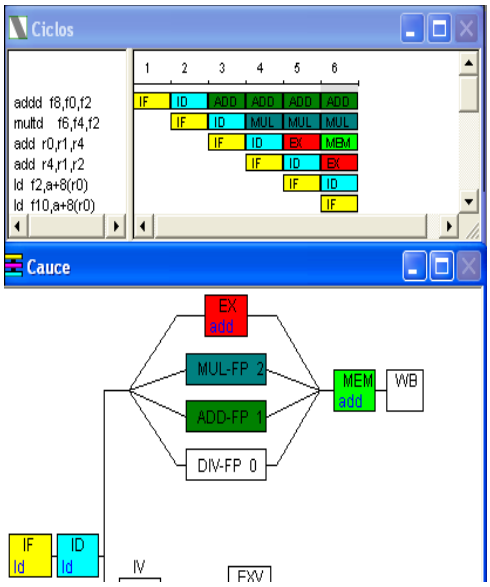
ADDD, SUBD	Add, Subtract in DP
MULTD, DIVD	Multiply, Divide in DP
CVTD2I, CVTI2D	Conversion: CVTx2y converts from type x to y. I=integer D=DP
ltD, gtD,leD.eqD.neD	Compare registers and put comparison bit in FP status register.
BFPT, BFPF	Test (True/False) Bit comparison in the FP and skip status register (16-bit offset from PC + 4)

NOTE: In the DLX simulator we watch the contents of FP registers and memory with floating-point data with the icons labeled "FP" and subscripts S or D (single/double precision).

As we will use double precision, **all data values are 8 bytes** in contrast to integers of 4 bytes. Pay attention to data section declarations for FP variables.

Introduction to Floating Point

Operations with data represented in floating point take longer than operations with integers, since several operations take place such as making exponents and mantissa additions, various adjustments such as alignment of mantissa, normalization, rounding, etc. Therefore, its passage through the EX execution stage takes more than one cycle. In other words, the LATENCY of a floating point operation may be greater than one cycle.



Although it seems obvious, it is important to note that the Functional Units for processing data in FP (addition, multiplication and division) are different from the "ALU" of integers, so it is possible that two or more instructions coincide in the EX phase, but each using different units.

For example, an integer addition and a FP addition can both be in the EX phase in the same cycle, since each one is performed in different functional units. Same thing happens with an integer addition and a multiplication in FP.

This fact can be clearly seen in WinDLXV where the pipeline is represented with this diversification in the data processing stage: EX for integers, E1, E1 .. or A1, A2 .. for FP. The figure shows a sequence in which three functional units are used simultaneously.

In the simulators we use, the latency is configurable by the user. For the following experiments we should set a latency of **4 cycles** (total duration of the data processing stage) for the addition unit (for addition and subtraction operations)

IMPORTANT:

- In DLXVSim, "Simulator" menu it is necessary to change the **configuration** by setting a latency of **4** (total cycles of the process stage). Then, you will have to **modify the hardware** (also in the Simulator menu); you will see the same message shown when the program starts. This saves the new configuration.
- In WinDLXV in **Configuration/Architecture** you have to put **3** latency cycles (here you put the cycles in excess compared to integers) and **DISABLE the segmentation** for the unit of addition-subtraction in floating point.
- In DLXVSim the LOAD-ALU stalls count as **scalar stalls** and the ALU-ALU and ALU-STORE as in **FP**
- In WinDLXV, RAW stalls are erroneously counted twice in some ocassions.

As some instructions use more cycles in EX than others, the pipeline looses its balance as FP processing instructions take longer than integer instructions. This sometimes results in **Structural Hazards**, when two instructions compete for the same resource, which can be the same functional unit of execution (EXn stages) or Memory (ME).

RAW data dependencies can also cause stalls of **more** than one cycle, even with the operand forwarding circuit, since FP operators in the EXn stage consume several cycles and not just one.

Another somewhat surprising effect is that the instructions are not completed in the same sequential order in which they entered the pipeline as some employ more cycles than others. ***All this is what you are going to check next.***

Structural Hazard

Let's look at the behaviour of the pipeline **in the absence of RAW hazard** using the instructions proposed below and always comparing the outcome from the **two** simulators.

Declare some values in the data area (a: .double 1,2,3 ...). Make sure operand forwarding is active. Hand make **a pipeline schema for each of the following cases with the help from both simulators and make sure you understand what happens.**

CASE A	Structural conflict by the adder in FP

`addd f4,f0,f2`
`addd f4,f0,f2`
`addd f6,f2,f0`	

CASE B	Structural conflict by ME

`addd f4,f0,f2`
`add r1,r2,r3`
`add r1,r2,r3`
`add r1,r2,r3`	

1. In case **A**, check that there are 3 **structural** stalls in each ALU instruction (except in the first one), since the floating point addition unit takes 4 cycles to compute a sum and the three instructions queue for its utilization, even if they do not have RAW data dependency.
2. In case **B**, a **structural** stall is also produced but, on this occasion, the conflicting resource is memory (stage ME). Explain why this situation is reached. You can substitute the entire additions for any instruction such as loads (*) or storage and you will still have the same structural stall in ME. This effect can only be seen in WinDLXV (DLXVSim does not consider this conflict of simultaneous passage through ME).

(*) If you use loads in FP make sure you use different FP registers in order to watch the experiment correctly in WindDLXV

Pipelining of Functional Units

Just as execution of instructions can be pipelined, a multi-cycle functional unit, such as the floating point adder, the multiplication and division units can also be segmented and perform in a pipeline.

This saves long stalls if there is a sequence of process instructions in FP that need to use the same functional unit. WinDLX allows pipeline operators by segmenting the units (configuration / architecture menu), but the other simulator, DLXVSim, does not.

Repeat CASE **A** of the previous tests in WinDLX but this time configure the adder with segmentation (Configuration / Architecture) and explain what happens.

Next, define a RAW dependency among the add instructions you have used and compare what happens now. Again, draw a diagram of the pipeline at the side and make sure you understand what happens:

WITHOUT Dependencies	Pipelined UF, absence of RAW

`addd f4,f0,f2`
`addd f4,f0,f2`
`addd f6,f2,f0`	

WITH RAW	RAW in pipelined UF

`addd f4,f0,f2`
`addd _____`
`addd _____`	

In the first case, structural stalls are avoided when using the same UF and performance is significantly improved.

To simplify the rest of the experiments, **we will not segment** functional units, so make sure you change it back to NOT-SEGMENTED in the menu.

RAW hazard analysis

Complete the following table (data vary according to the UFs latency):

Producer	***Consumer***	***Stalls cycles***	***Example***
FP ALU	FP ALU	3	addd **fx**,fa,fb / addd fa,fb,**fx**
FP ALU	Store		addd **fx**,fa,fb / Store **fx**
Load	FP ALU		ld **fx** / add fa,fb,**fx**
Load	Store		ld **fx** / sd **fx**

Remember that the sequence Load-Store in WinDLXV and DLXVSim for integers did not work in the same way?. Well, for the case of floating point data you will check it is consistent (which seems an inconsistence in WinDLXV).

WAW Hazard

A pipeline that runs unevenly can lead to situations like the ones we are going to study now through Cases 1 and 2 in the table below.

- In Case 1: you will verify that the second instruction **is completed before** the first instruction; Loads of memory data take the same number of cycles no matter whether data is an integer or a F number. By the way, do you think the same thing will happen if we replace the load instruction with an ALU instruction like add r1, r2, r3? … Of course! --Check it out.
- In Case 2 both instructions write into **f4**, which defines a **WAW (W**rite **A**fter **W**rite**) Data dependence** that would not be of any importance if the path in the pipeline were equally long for both instructions; but it's not like that. Then reflect and explain what happens. What would happen if this situation was not detected and solved by the processor?

CASE 1	**CASE 2**
`addd f4,f0,f2` `ld f6,a(r0)`	`addd f4,f0,f2` `ld f4,a(r0)`

Be sure to answer all questions and complete all diagrams and stalls: this is for sure going to be part of future evaluations.

Activities

For the proposed programs, the same must always be done:

a) First develop a non-optimized version (causing all possible stalls and wasting the delay slot with nop). Before running it, specify inside a comment all Hazards and stalls (0,1, 2 stalls, whatever). Compute expected cycles.

b) Next, run the program in both simulators and verify your predictions, executing loops step by step during at least one iteration. Make sure the programs works correctly and produces the expected output. Contrast the total planned cycles with outcome from simulators. Do any required adjustments.

c) In a copy of the program, optimize the program reordering instructions and filling DS with useful instructions so that there are no stalls left if possible. Recalculate cycles and obtain Gain or Acceleration.

d) As before, contrast the execution of the optimized program vs. outcomes from the simulators, making adjustments as necessary.

For the following activities, change the simulator settings so that they work with multiplication and addition floating point units that take **2 cycles in total** and are **not** segmented. Of course, use data forwarding and delayed branch options.

1. Develop a program that, given two vectors a and b of double precision FP numbers, calculate the following: C (i) = a(i)·cte + b(i). The cte is a real number in double precision. Reserve the necessary space for vector C. It is recommended to schematize the pipeline ONLY next to the instructions with hazards and/or stalls to understand what happens.
2. Transform the program from session 2 reverse.s (addition of two integer vectors a and b into a c vector in backward order) to use floating point data vectors.
3. Develop a program that computes the scalar product PE of two vectors of up to 20 FP numbers (PE = Σ a(i)·b(i)). Results should be stored in a memory location labelled PE. This code will be used in the following lab session, so, don't miss it and bring it with you.
4. Transform the **condsuma.s** program of session 3 for Control Hazard study so that the data vectors to be added are FP numbers. That is, vectors A, B, C and S must be double precision FP numbers.