Multicycle Pipeline Operations

Material Covered

Section 3.7.

Long-Latency Operations (Topics)

Typical instructions: floating point

Pipelined v. non-pipelined execution units

Initiation interval and latency

Placement in Chapter-3 DLX pipeline

Timing diagrams

Problems Introduced by Long-Latency Operations

Functional Unit Structural Hazards

Occurs when initiation interval ¿ 1

E.g., in DLX second of two consecutive FP divides must stall because of hazard.

Register Write (MEM Stage) Structural Hazards

Occurs when ¿ 1 operations could simultaneously enter MEM.

(MEM can normally accommodate only one instruction.)

RAW Hazards

Due to longer-than-one-cycle computation time.

WAW hazards

Occurs when operations complete out of order.

Precise Exceptions

A headache because an instruction can be ready to write . . .
. . . long before a preceding instruction raises an exception.

Handling Functional Unit Structural Hazards

Hazard can be eliminated . . .
. . . by duplicating or fully pipelining functional units.

Otherwise, instructions must be stalled.

Hazard easily detected:

Units provide a *ready-next-cycle* signal to ID stage.

Instruction stalled if ready-next-cycle for needed unit is 0.

Handling Register Write (MEM Stage) Structural Hazards

Method 1: Delay instruction in ID.

Include a shift register called a *reservation register.*

Each cycle the reservation register is shifted.

A 1 indicates a “reservation” to enter MEM.

Bit position indicates time . . .
. . . with the LSB indicating two cycles later . . .
. . . the next bit indicating three cycles later . . .
. . . and so on.

The ID stage controller, based on the opcode of the instr., . . .
. . . knows the number of cycles before MEM will be entered.

It checks the corresponding reservation register bit . . .
. . . if it’s 1 then IF and ID are stalled . . .
. . . if it’s 0 then the bit is set to 1 and the instruction proceeds.

If such a stall occurs . . .
. . . the reservation register is still shifted . . .
. . . and so a 0 will eventually move into the bit position.

Method 2: Delay instructions ready to enter MEM

Each functional unit provides a signal . . .
. . . indicating when it has an instruction ready to enter MEM.

One of those signals is chosen (using some method) . . .
. . . the corresponding instruction moves to MEM . . .
. . . while the others are stalled.

Comparison of Method 1 and 2

Method 1 is easier to implement . . .
. . . since logic remains in one stage.

In contrast, logic for method 2 would span several stages . . .
. . . since stages back to IF might need to be stalled . . .
. . . and so critical paths would be long.

Method 2 is more flexible . . .
. . . since priority could be given to longer-latency instructions.

Handling RAW Hazards

The interlock mechanism for RAW hazards . . .
. . . must keep track of registers with pending writes . . .
. . . and use this information to stall instructions.

Consider, `add f1, f2, f3`.

Check if any uncompleted preceding instructions write `f2` or `f3`.

If so, stall until register(s) written or can be bypassed to adder.

Possible RAW Interlock Implementations.

Brute Force: Check all following stages

As done for integer operations, check following stages …
… for pending write to register.

Each stage of every pipelined unit must be checked.

Too expensive.

Register file includes *ready bit* for each register.

Ready bit normally 1, indicating no pending writes (so value valid).

When instruction issued, bit set to 0 …
… when instruction completes and result written, set back to 1.

Instruction stalls if either operand's ready bit is 0 …
… *and* cannot be bypassed.

Handling WAW Hazards

The interlock mechanism for RAW hazards . . .
. . . will also handle those WAW hazards . . .
. . . in which there is an intervening write.

If there is no intervening write . . .
. . . the earlier instruction is nulled.

Precise Exceptions

Problem is registers written out of order . . .
. . . so some registers must be *unwritten* . . .
. . . so that when handler starts . . .
. . . it must *seem* as though . . .
. . . all instructions before faulting instructions executed . . .
. . . while no instructions after faulting instruction execute.

To do this either . . .
. . . add lots of stalls so instructions do finish in order . . .
. . . or need to *unexecute* instructions.

The first option is fine for debugging, too slow otherwise.

The second option requires lots of hardware.

Unexecuting Instructions

An instruction is unexecuted . . .
. . . by restoring the previous contents of any register it wrote.

Method 1: *History File*

History file holds replaced values.

These are used to undo writes.

Method 2: Writes to register file are buffered.

Register writes (register number and new value) ...
... are first placed in a buffer ...
... possibly out of program order.

Writes from buffer to register file performed in order ...
... waiting for long-latency operations to complete.

Register reads check the buffer first, then the register file.

When an exception occurs ...
... only writes preceding the faulting instruction ...
... are made from the buffer to the register file.

Disadvantage: Checking both buffer and register file is time-consuming.

Method 3: *Future File*

Two register files maintained, *main* and *future*.

Future file written as instruction complete . . .
. . . main file written in program order.

Future file is used for reading registers.

At an exception, . . .
. . . main file updated up to faulting instruction . . .
. . . future file is effectively erased . . .
. . . its contents replaced by main register file before handler starts.

Other Precise Exception Methods

Above methods expensive.

There are other ways of providing precise exceptions.

Method 1: Non-excepting versions of floating-point instructions.

Only use slow, excepting, versions where needed.

Method 2: Stall just long enough to ensure precise exceptions.

Call an instruction that will definitely finish *committed*.

Functional units can be designed so that …
… if there will be an exception …
… it will happen early. (E.g., check for zero divisor.)

With such functional units …
… it can be determined that instructions are committed …
… soon after they enter the functional unit.

In this method, stall instructions …
… until all preceding instructions are committed.