Lecture 5: Out of order, Dynamic versus Static scheduling

Hossam A. H. Fahmy

Cairo University, Faculty of Engineering

Overview

1. Going out of order
   - Dynamic scheduling
   - Register renaming
   - Advanced techniques
2. VLIW
   - Can compilers help?
3. What did we learn?

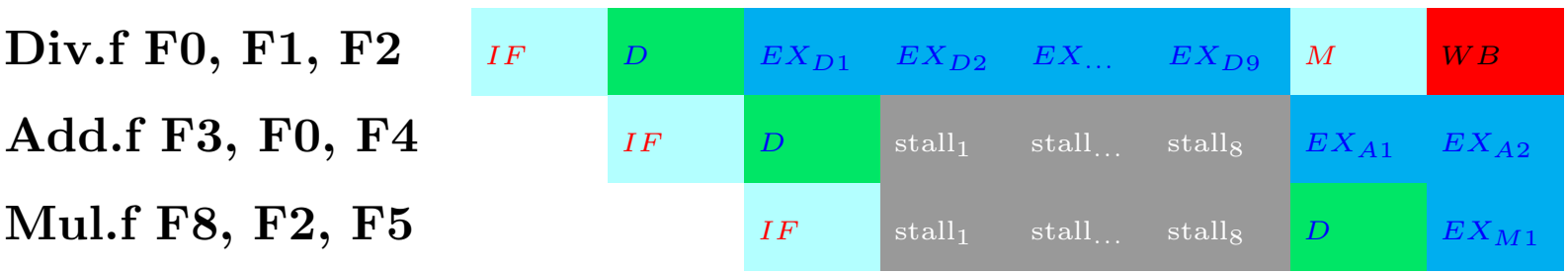
Where are we?

- We pipeline the instructions to enhance the throughput.
- We included long instructions.
- We handled exceptions.
- We managed in order issue to multiple pipelines in parallel.

Now we are ready to have out of order execution.

- Dynamic scheduling in the hardware.
- Static scheduling in the software.

Why do we go out of order?



- The **Add.f** must wait because of the RAW hazard.
- The **Mul.f** does not need to wait. It can start *before* the **Add.f** instruction.

Dynamic scheduling

Execution: is non-sequential (not the original order)

- This reduces stalls,
- improves the utilization of the functional units, and
- enables parallel execution.

Exceptions: must be precise. We must maintain the *appearance* of sequential execution. This is important but hard.

Instruction buffer

- Instructions are brought from memory and *dispatched* to the decoder in order.
- The decoder must be able to look at multiple instructions to re-order.
- Some instructions will be *issued* (started) while others are waiting.

The instructions reside in the *instruction buffer* while the decoder checks them. (Different names are used by various people.)

Dispatch and Issue

Dispatch: is the first part of decoding.

- The new instructions get a location *in order* in the instruction buffer.
- If the buffer is full, the dispatching unit *stalls all the following instructions*.

Issue: is the second part of decoding

- Start the execution, i.e. send instructions from instruction buffer to execution units *out of order*.
- An instruction that has to *wait does not delay the following instructions*.

Register renaming

Div.f	**F0, F1 ,F2**		**Div.f**	**L0, L1 ,L2**
Add.f	**F3, F0, F4**	$\Rightarrow$	**Add.f**	**L3, L0, L4**
Mul.f	**F0, F1, F2**		**Mul.f**	**L5, L1, L2**
Add.f	**F4, F0, F2**		**Add.f**	**L6, L5, L2**

Think of the registers as *names* not specific locations.

On a write, allocate a new location and record it in a map table.

On a read, find the location in the table of the most recent write.

De-allocation occurs when the dependent RAW hazards are cleared.

This is a neat idea that might be implemented in either the hardware or the software. It eliminates WAW and WAR hazards.

Other techniques: Scoreboards and Tomasulo's algorithm

- The scoreboard technique uses a centralized approach to check and resolve the dependencies. It was first implemented in CDC6600 in 1964.
- The Tomasulo data flow technique uses a distributed approach where the reservation station may also contain the *value* of the register not just a tag. This amounts to a register renaming scheme. It was first implemented in IBM 360/91 in 1967.

Those who are interested can read more about these techniques in the "Quantitative Approach".

Very Long Instruction Word

We have seen some problems with multiple issue superscalars:

- N^2 dependence checks (large stall and bypass logic),
- N^2 bypass buses (partially fixed with clustering), as well as
- wider fetch and problems with branch prediction.

In VLIW,

- a single issue pipeline that has N parallel units is used,
- the compiler only puts independent "instructions" in the same group,
- VLIW travels down the pipeline as one unit, and
- in *pure* VLIW machines the processor does not need to do any dependence checks.

VLIW purity

In a pure (classical/ideal) VLIW design the compiler schedules the pipeline including the stall cycles.

$\Rightarrow$ The compiler must know the exact latencies and organization of the pipeline.

Problem 1: These details vary in different implementations. We must recompile the code. (TransMeta recompiles on the fly.)

Problem 2: Even for a specific implementation, the latencies are not fixed. What shall the hardware do for a cache miss?

Real implementations are not ideal.

Scheduling and issuing

Schedule: Decide the *order* of the instructions.

- Put independent instructions between the slow operations and the instructions that need their results.

Issue: Decide the *time* a specific instruction starts.

- Once all the dependencies are clear we can start.

	Schedule	Issue
Pure VLIW	SW	SW
In-order superscalar	SW	HW
Out-of-order (dynamic)	HW	HW

Scheduling: Compiler or HW

Compiler:

+ Large scope (may be the whole program).
+ Leads to a simpler hardware.
– Low branch prediction accuracy.
– No information about memory delays (cache misses).
– Difficult to speculate and recover.

Hardware:

+ Better branch prediction accuracy.
+ Dynamic information about memory delays.
+ Easier to speculate and recover.
– Finite resources to buffer instructions.
– Complicated hardware (harder to verify, may lead to slower clock).

Compiler techniques

We want to increase the number of independent instructions.

Loop unrolling: Put more than one iteration in sequence in a wider loop.

Software pipelining: Similar to what happens in hardware, a part of the first iteration is done with a part of the second iteration.

Trace scheduling: Programs include other things beyond loops.

Those who are interested can read more about these techniques in the "Quantitative Approach".

Where are we now?

- Pipelines.
- Exceptions.
- Muliple issue.
- Dynamic scheduling.
- Static scheduling.

Next we go to the memory system.