In-order vs. Out-of-order Execution

In-order instruction execution

- instructions are fetched, executed & complete in compiler-generated order
- one stalls, they all stall
- instructions are **statically scheduled** by the hardware

Out-of-order instruction execution

- instructions are fetched in compiler-generated order
- instruction completion may be in-order (today) or out-of-order (older computers)
- in between they may be executed out of their compiler-generated order
- instructions behind a stalled instruction can pass it
- instructions are **dynamically scheduled** by the hardware

Dynamic Scheduling

Dynamically scheduled or out-or-order processors:

- after instruction decode
  - check for **structural hazards**
    instructions can be **issued** when a functional unit is available
  - check for **data hazards**
    instructions can be **dispatched** when their operands are have been calculated or loaded from memory
    (can now read registers & execute)

- ready instructions can execute before earlier instructions that are stalled, e.g., waiting for their operands to be computed
  - when go around a **load** instruction that is stalled for a cache miss:
    - use **lockup-free caches** that allow instruction issue to continue while a miss is being satisfied
    - the load use instruction still stalls
  - when go around a **branch** instruction:
    - the instructions that are issued from the predicted path are issued speculatively, called **speculative execution**
    - when the branch is resolved, if the prediction was wrong, **wrong path instructions** are flushed from the pipeline

Speculation

Instruction **speculation**: executing an instruction that might not be needed (just in case it is needed)

- must be safe (no additional exceptions)
- must generate the same results

Dynamic Scheduling

Instruction issue does **NOT** necessarily go in program order

- the hardware decides which instructions should issue next

program order (in-order processors, the fetch order)

```
lw  $3, 100($4)     in execution, cache miss
add $2, $3, $4      waits until the miss is satisfied
sub $5, $6, $7      waits for the add
```

execution order (out-of-order processors)

```
lw  $3, 100($4)     in execution, cache miss
sub $5, $6, $7      in execution during the cache miss
add $2, $3, $4      waits until the miss is satisfied
```

Data Dependences & RAW Hazards

Cause of the data dependence:

- the result produced by an instruction (the producer) is needed by a subsequent instruction (the consumer)
- example:

```
LD F0, 0(F2)        (potential RAW hazard)
SUBF _, F0, _
```

Cause of the RAW hazard:

- there is a >1 cycle delay between producing & using an operand e.g., a subsequent instruction uses a value loaded from memory

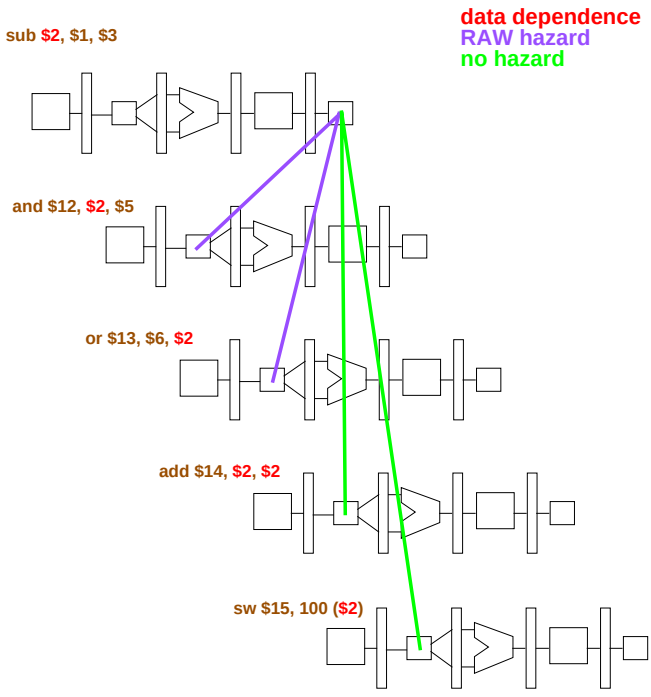
HW solutions

- forwarding hardware (eliminate the hazard)
- pipelined interlock (stall)

Compiler solutions

- code scheduling to place independent instructions between the producer & consumer

Dependences vs. Hazards



Name Dependences & WAR/WAW Hazards

Cause of the name dependence:

- the compiler runs out of registers, so has to reuse them
- example:
  - anti-dependence (potential WAR hazard)

```
DIVF _, F1, _
ADDF F1, _, _
```

  - output dependence (potential WAW hazard)

```
DIVF F0, F1, F2
ADDF F0, F2, F3
```

Cause of the WAR & WAW hazards:

- separate functional units (or execution pipelines) for different instructions that have different numbers of stages, i.e., instructions take different numbers of cycles
- out-of-order execution may allow the second instruction to complete before the first

HW solution

- register renaming (coming up soon)

Control Dependences & Hazards

Cause of the control dependence:

- which instructions are executed next depends on the value of a branch condition

Cause of the control hazard:

- branch condition (& the target) are not known before the next instruction is fetched

HW solutions

- dynamic branch prediction (2-bit, 2-level, combined, confidence predictors)
- BTB, return stack
- resume buffer

Compiler solutions

- schedule instructions between the branch and the paths
- static branch prediction