Instruction scheduling

Comparing in-order and out-of-order processors

Introduction

In-order

Out-of-order

Conclusion

Static versus dynamic scheduling

Static scheduling (in-order)

- Instructions leave front-end in program order
- All data dependencies are resolved in the front-end

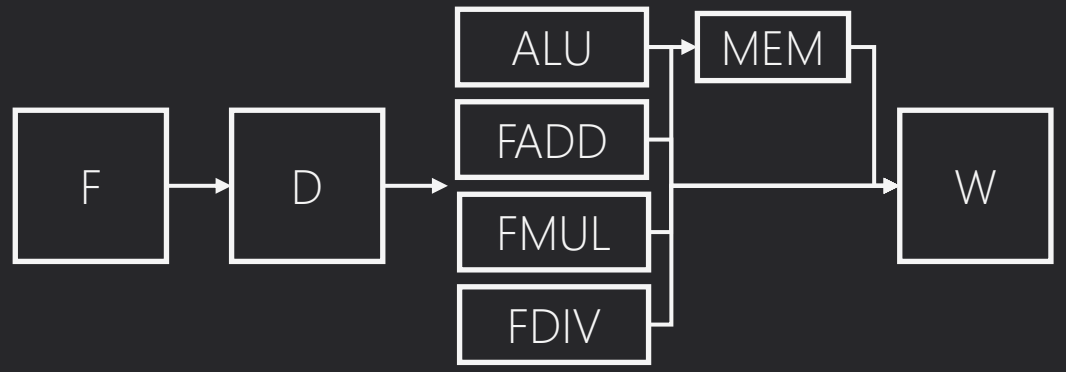
- e.g., our five-stage pipeline

Dynamic scheduling (out-of-order)

- Instructions can leave front-end out-of-order
- The *commit stage* (formerly write back) ensures program order is observed

Functional units (FUs)

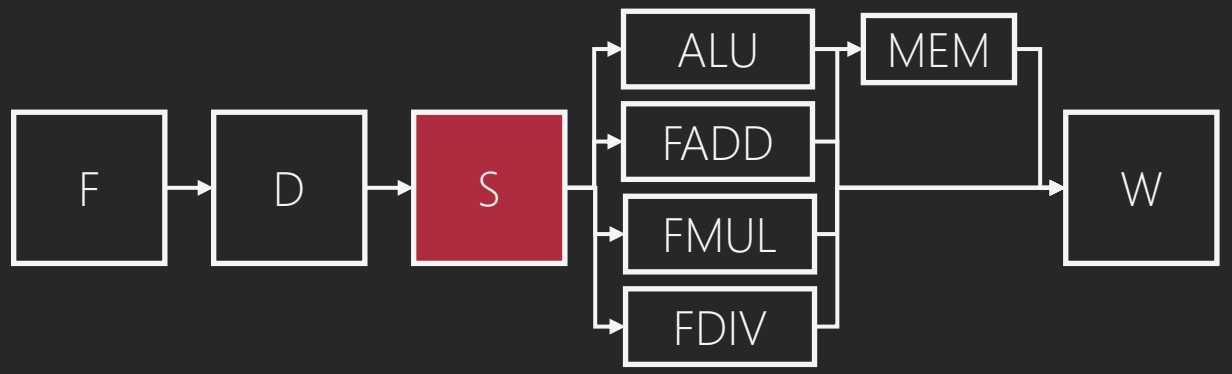
- Different units for different classes of operations
  - May or may not be pipelined
- Separate register files for integer and floating-point



- Functional units have different latencies
- Data hazards
  - Out-of-order write hazards
  - What is the cache miss latency?
- Structural hazards
  - Execute stage: Non-pipelined FUs
  - Write back stage

The issue stage

- Add an issue stage to the front-end of the pipeline
- The issue stage will stall until there are no dependence problems



When to issue instructions:

1. Is the FU busy?
2. Are the inputs ready? (RAW)
3. Is it correct to write back the result? (WAR, WAW)
4. Can we write back the result? (structural)

Tracking instructions in issue

Functional Unit	Busy?	F_i (dest)	F_j (src1)	F_k (src2)
ALU/MEM				
FADD				
FMUL				
FDIV				

A table (in hardware) tracks the status of FUs

- Add (precondition: no hazards)
- Remove (after write-back)

Can we issue instruction x?

1. `any(x.src1 == y.fi or x.src2 == y.fi for y in table)`
2. `any(x.dest == y.fj or x.dest == y.fk for y in table)`
3. `any(x.dest == y.fi for y in table)`

Issuing in-order

Assumptions

- Instructions are issued when:
  - There is no RAW hazard
  - The FU is available
  - (this is less strict than the earlier precondition)
- The operands are read by the issue stage and saved in the functional unit

Impact on hazards

- WAR hazards are eliminated
  - Table no longer needs source columns
- WAW hazards are still possible
  - FUs have different latencies

The limits of in-order scheduling

```
addi r6, r0, 100
loop:
    l.d f3, 0(r1)
    l.d f4, 0(r2)
    mul.d f5, f4, f3
    add.d f1, f3, f3
    mul.d f4, f3, f3
    ...
```

- Suppose
  - The first l.d hits in the cache
  - The second l.d misses in the cache
- There is a RAW dependence
  - mul.d and second l.d (f4)
- But there isn't one with subsequent instructions, like add.d
  - ILP is being left on the table

Issuing out-of-order

- Add a *buffer* to the issue stage
  - Contains instructions waiting to be issued
  - Superscalar processors fill up the buffer more quickly
- The decode stage adds to the buffer if,
  - The buffer is not full
  - The instruction does not cause a WAR hazard
  - The instruction does not cause a WAW hazard
- The issue stages issues from the buffer if there is no RAW hazard

The limits of out-of-order scheduling

```
1. div r1, r2, r3
2. add r4, r1, r5 # RAW (w/ 1)
3. add r5, r6, r7 # WAR (w/ 2)
4. add r1, r8, r9 # WAW (w/ 1)
```

- Instructions are only added to the buffer when there are no WAW or WAR hazards
  - But these are *name* dependencies

- The compiler is limited to the registers defined by the ISA
  - What if there are more?

Register renaming

```
1. div r1, r2, r3
2. add r4, r1, r5 # RAW (w/ 1)
3. add r5, r6, r7 # WAR (w/ 2)
4. add r1, r8, r9 # WAW (w/ 1)
```

After some register renaming,

```
1. div r32, r2, r3
2. add r33, r32, r5      # RAW (w/ 1)
3. add r34, r6, r7       # ~~WAR (w/ 2)~~
4. add r35, r8, r9       # ~~WAW (w/ 1)~~
```

- Suppose there are twice as many microarchitectural registers as architectural registers
  - ISA defined 32 registers: r0 to 31
  - Hidden from programmer: r32 to r63
- How would we support this?
  - More tracking in hardware
  - First done in IBM System/360 Model 91

Conclusion

- Recap of important points

A brief history

- m-way in-order processors dominated early
  - Compensated for lower IPC with much high frequency
  - e.g., ALPHA 21064, 21164, MIPS R5000

- m-way out-of-order processors dominated later
  - Moore's law allowed more resources to handle complex scheduling