What was it like in 1960?

- No cache
- Just started to pipeline
- Punch cards
- Core memory
- Fairly weak

OOO – Why bother?

- Runtime (dynamic-only) behavior
  - Cache misses
- “Architectural” independence
- 1960 weak compiler

Conditions Mem -> Issue

- Wait for
  - Room in the fetch buffer
- Do
  - Put something there

Conditions Issue -> Read

- Wait for
  - Functional unit available for execution of the instruction in the front of the fetch buffer and
  - All active instructions do not have the same destination register.
- Do
  - Reserve the functional unit
  - Reserve the destination register
  - Reserve the source registers (if available)
    - Else register interest in their producer functional unit

Read -> Execute

- Wait for
  - The operands to be valid in the register file
- Do
  - Compute
  - Unregister interest in your input operands

Execute -> Complete

- Wait for
  - After latency of the functional unit
- Do

Complete

- Wait for
  - Previous active instructions dependent on the destination have read it
- Do
  - Notify dependent scoreboard entries of this completion
  - Release the functional unit
  - Write to, and release destination register

When does this work?

- Functions of different latencies
  - Loads, stores, mult, div, rem, add, sub
- Instructions are interleaved “well”
- Works for read after write (RAW)
- Or no dependencies

When does this not improve performance?

- Waist of resources (maybe slower)
  - ADD R1, r0, r3
  - ADD R2, r4, r1
  - ADD R10, R2, R12
- Worse:
  - ADD R1, R2, R3
  - ADD R1, R4, R5
- Structural limitations
  - MUL
  - MUL
  - ADD

Make this faster

```
MUL     R1, R2, R3
SUB     R10, R1, R13
ADD     R1, R5, R6
```

Make this faster

- MUL R1, R2, R3
  ADD R4, R5, R1

Make this faster

- MUL R1, R2, R3
  MUL R15, R16, R17
  SUB R7, R8, R1
  ADD R1, R5, R6