**Instruction Level Parallelism (ILP)**

Preserve the sequential semantics of the ISA but... try to execute as many instructions at once as we can.

- Review of dependences
- Renaming to eliminate false dependences
- Scoreboarding: hardware out-of-order execution
- Tomasulo's Algorithm: OOO execution = renaming

---

**Review: Dependences**

- Read-after-write (RAW)
  - ADD $S4, S5
  - ADD $T8, $T9
  - Also known as a "flow dependence"

- Write-after-read (WAR)
  - ADD $S4, S5
  - Also known as an "anti-dependence"

- Write-after-write (WAW)
  - ADD $S4, $T5
  - ADD $S5, $T5

WAR and WAW are "false dependences"
- The dependences have to do with names, not values
- They can be eliminated by "re-writing the code"

---

**Example Code**

```
LD F6, 34(R2)
LD F2, 45(R3)
MULTD F1, F2, F4
SUBD F1, F1, F2
DIVD F10, F2, F6
ADD F6, FR, F2
```

---

**Example Code with Renaming**

```
LD F6, 34(R2)
LD F2, 45(R3)
MULTD F1, F2, F4
SUBD F1, F1, F2
DIVD F10, F2, F6
ADD F12, FR, F2
```
**These Are Generally Applicable Ideas**

- String name = getName(id);
- String printStr = id + " ": + name;
- int
- String name = getName(id);
- String printStr = id + " ": + name;
- String name = getName(id);
- String printStr = id + " ": + name;
- printStr = id + " ": + name;

**Rewritten Code**

- String name = getName(id);
- String printStr = id + " ": + name;
- String name = getName(id);
- String printStr = id + " ": + name;
- printStr = id + " ": + name;
- Can't get rid of flow dependences by renaming
- Renaming costs memory
- Loops tend to produce false dependences
  - 'Loop unrolling' does what that sounds like
  - Unrolled loops can benefit from renaming

**Pipelining and Dependences**

- Structural hazards are structurally impossible
- WAR violation is structurally impossible
- RAW violation is structurally impossible
- RAW happens
- When "things go wrong" we stall
- Stalling thins away potential performance
- Why isn't there a hazard between MEM and IF?

**Instruction Level Parallelism (ILP): Pipelining**

- Pipelining is a form of ILP
  - More than one instruction in flight at a time
- Respecting sequential semantics
  - More to worry about dependences between instructions
- The structure of pipelines makes WAR and RAW easy
- Pipelines are:

  ![Pipelining Diagram]

  - in order issue
  - in order execute
  - in order commit
**Pipelines: Going Faster**

- How can we improve the performance of the 5-stage pipeline?
- We could try making the pipeline deeper – i.e., breaking individual stages up into multiple stages
  - Ideally, a 3-stage pipeline should support a cycle time about double that of a 5-stage, but...
  - More stages become more costly
  - Fluctuates (e.g., mispredicted branches) become more expensive
  - Diminishing returns...
- Additionally, some operations take a lot longer than others
  - For example:
    - Cache misses...
    - Floating point is slower than integer arithmetic
  - How should we deal with that?

**Floating Point and the Pipeline**

The MIPS pipeline with floating-point functional units.

**Pipelines: Going Wide**

- We have two basic choices:
  - Only one instruction may be in EX stage, no matter how long it takes it to get through there, or...
  - Let's cram instructions into EX as fast as we can
- Which should we do?
  - Reminder: We're trying to go fast...
- Putting multiple functional units in parallel is both a problem and an opportunity
- The Opportunity:
  - Hey, this is great! Why don’t I just stuff a bunch of ALUs, some memory interfaces, some that units, etc., in there?
  - More hardware = higher performance?
  - In fact, why don’t I issue more than one instruction per cycle/PP?
  - “Yield-based” = NOT part of today’s material, but not far from it
Going Wide: The Problems

- In order execution leads to under-utilization of hardware
- Parallel execution -> out of order execution / completion
  - Time per stage is not a constant
  - Structural hazards are possible
    - IF stage takes many cycles, and is nonpipelined
    - May need to refill more than one register in a cycle
  - Out of order execution
    - WAR/WAW dependencies may be longer
    - "Precise exceptions" are more difficult to implement
  - Out of order completion
    - WAW/WAR hazards are possible

Two Approaches Today

- Scoreboarding
  - In order issue
    - Out of order execution
    - Out of order completion
- Tommasulo’s Algorithm
  - In order issue
    - Out of order execution
    - Out of order completion
    - Register renaming to eliminate WAW and WAR dependencies
- Modern processors
  - Descendants of Tommasulo
    - Add a “re-order buffer” to achieve in-order completion
    - Eliminate precise exceptions

Scoreboarding

- Data path now has two largely decoupled pieces:
  - IF fetches an instruction each cycle
    - There is a small variable buffer of already fetched instructions
    - If issue stalls, the buffer fills
    - When instructions complete, they leave the buffer
  - Scoreboarding: 4-stage execution
    - Issue -> check structural/WAW hazards (stall and clear)
    - Resolve (ex) -> check WAR (and all speculative reads, read-miss)
    - Execute -> execute operation, Notify scoreboard when done
    - Write -> check for WAW (pull write and clear)
Dynamic MIPS Datapath

Scoreboard Example Cycle 62

Scoreboard Summary
- Speedup 1.7 from compiler; 2.5 by hand
- BUT slow memory (no cache)
- Limitations of 6600 scoreboard
  - No forwarding (First write register then read it)
  - Limited to instructions in basic block (small window)
  - Number of functional units (structural hazards)
  - Wait for WAR hazards
  - Prevent WAW hazards

Another Dynamic Algorithm: Tomasulo Algorithm
- For IBM 360/91 about 3 years after CDC 6600 (1966)
- Goal: High Performance without special compilers
- Differences between IBM 360 & CDC 6600 ISA
  - IBM has only 2 register specifiers/inst vs. 3 in CDC 6600
  - IBM has 4 FP registers vs. 8 in CDC 6600
  - (x86 has 4 general purpose integer registers...)
- Laid to Alpha 21264, HP 8000, MIPS 10000, Pentium II, PowerPC 654,
Installation of the IBM 360/91 in the Columbia Computer Center machine room in February or March 1960.

Tomasulo Organization

- Control & buffers distributed with Function Units (FU) vs. centralized in scoreboard.
- FU buffers called "reservation stations" have pending operands.
- Registers in instructions replaced by values or pointers to reservation stations (RS): called register renaming.
- avoids WAW, RAW hazards.
- More reservation stations than registers, so can do optimizations compilers can’t.
- Results to FU from RS, not through registers, over Common Data Bus that broadcasts results to all FUs.
- Load and Stores treated as FUs with RSs as well.
- Integer instructions can go past branches, allowing FP ops beyond basic block in FP queue.

Tomasulo Algorithm vs. Scoreboard
Three Stages of Tomasulo Algorithm

1. Issue—get Instruction from FP Op Queue
   - Reservation stations: no structural hazard
   - Credit issue: new & sends operands (rename registers)

2. Execution—operate on operands (EX)
   - When both operands ready, then execute
   - If not ready, watch Common Data Bus for result

3. Write result—Finish execution (WB)
   - Write to Common Data Bus to all waiting units;
   - mark reservation station available

   • Where’s the register renaming?
     - FU’s may wait to hear a result produced by a particular other FU — that’s a new name
     - Reservation stations copy the opened values — that’s their new names

Tomasulo v. Scoreboard

(IBM 360/91 v. CDC 6600)

- Pipelined Functional Units
  (8 issued, 3 store, 3 = 2 x 2)
  window size: ≤ 14 instructions
- No issue on structural hazard
- WAR: renaming avoids
- Broadcast results from FU
- Control: reservation stations

- Multiple Functional Units
  (1 load/store, 1 × 2 x 1 +)
  ≤ 5 instructions
- Same
- Staff completion
- Staff completion
- Write/read registers
- Central scoreboard

Tomasulo Example Cycle 57

- Again, in-order issue, out-of-order execution, completion