Static vs. dynamic scheduling

- Assumptions (for now):
  - 1 instruction issue / cycle
  - Several pipelines with a common IF and ID
    - Ideal CPI still 1, but real CPI won’t be 1 but will be closer to 1 than before
    - Same techniques will be used when we look at multiple issue
- Static scheduling (optimized by compiler)
  - When there is a stall (hazard) no further issue of instructions
  - Of course, the stall has to be enforced by the hardware
- Dynamic scheduling (enforced by hardware)
  - Instructions following the one that stalls can issue if they do not produce structural hazards or dependencies

Static vs. dynamic scheduling

- Assumptions (for now):
  - 1 instruction issue / cycle
  - Several pipelines with a common IF and ID
    - Ideal CPI still 1, but real CPI won’t be 1 but will be closer to 1 than before
    - Same techniques will be used when we look at multiple issue
- Static scheduling (optimized by compiler)
  - When there is a stall (hazard) no further issue of instructions
  - Of course, the stall has to be enforced by the hardware
- Dynamic scheduling (enforced by hardware)
  - Instructions following the one that stalls can issue if they do not produce structural hazards or dependencies

Dynamic scheduling

- Implies possibility of:
  - Out of order issue (we say that an instruction is issued once it has passed the ID stage) and hence out of order execution
  - Out of order completion (also possible in static scheduling but less frequent)
  - Imprecise exceptions (will take care of it later)
- Example (different pipes for add/sub and divide)
  - R1 = R2 / R3 (long latency)
  - R2 = R1 + R5 (stall, no issue, because of RAW on R1)
  - R6 = R7 - R8 (can be issued, executed and completed before the other 2)

Issue and Dispatch

- Split the ID stage into:
  - Issue: decode instructions; check for structural hazards and maybe more hazards such as WAW depending on implementations. Stall if there are any. Instructions pass in this stage in order
  - Dispatch: wait until no data hazards then read operands. At the next cycle a functional unit, i.e. EX of a pipe, can start executing
- Example revisited.
  - R1 = R2 / R3 (long latency; in execution)
  - R2 = R1 + R5 (issue but no dispatch because of RAW on R1)
  - R6 = R7 - R8 (can be issued, dispatched, executed and completed before the other 2)

Implementations of dynamic scheduling

- In order to compute correct results, need to keep track of:
  - execution unit (free or busy)
  - register usage for read and write
  - completion etc.
- Two major techniques
  - Scoreboard (invented by Seymour Cray for the CDC 6600 in 1964)
  - Tomasulo’s algorithm (used in the IBM 360/91 in 1967)

Scoreboarding -- The example machine

- The scoreboard keeps a record of all data dependencies
  - Keeps track of which registers are used as sources and destinations and which functional units use them
- The scoreboard keeps a record of all pipe occupancies
  - The original CDC 6600 was not pipelined but conceptually the scoreboard does not depend on pipelining
- The scoreboard decides if an instruction can be issued
  - Either the first time it sees it (no hazard) or, if not, at every cycle thereafter
- The scoreboard decides if an instruction can store its result
  - This is to prevent WAR hazards
An instruction goes through 5 steps

- We assume that the instruction has been successfully fetched (no I-cache miss)
- 1. Issue
  - The execution unit for that instruction type must be free (no structural hazard)
  - There should be no WAW hazard
  - If either of these conditions is false the instruction stalls. No further issue is allowed
    - There can be more fetches if there is an instruction fetch buffer (like there was in the CDC 6600)

Execution steps under scoreboard control

- 2. Dispatch (Read operands)
  - When the instruction is issued, the execution unit is reserved (becomes busy)
  - Operands are read in the execution unit when they are both ready (i.e., are not results of still executing instructions). This prevents RAW hazards (this conservative approach was taken because the CDC 6600 was not pipelined)
- 3. Execution
  - One or more cycles depending on functional unit latency
  - When execution completes, the unit notifies the scoreboard it’s ready to write the result

Execution steps under scoreboard control (c’ed)

- 4. Write result
  - Before writing, check for WAR hazards. If one exists, the unit is stalled until all WAR hazards are cleared (note that an instruction in progress, i.e., whose operands have been read, won’t cause a WAR)
- 5. Delay (you can forget about this one)
  - Because forwarding is not implemented, there should be one unit of delay between writing and reading the same register (this restriction seems artificial to me and is "historical").
  - Similarly, it takes one unit of time between the release of a unit and its possible next occupancy

Optimizations and Simplifications

- There are opportunities for optimization such as:
  - Forwarding
  - Buffering for one copy of source operands in execution units (this allows reading of operands one at a time and minimizing the WAR hazards)
- We have assumed that there could be concurrent updates to (different) registers.
  - Can be solved (dynamically) by grouping execution units together and preventing concurrent writes in the same group or by having multiple write ports in the register file (expensive but common nowadays)

What is needed in the scoreboard (slightly redundant info)

- Status of each functional unit
  - Free or busy
  - Operation to be performed
  - The names of the result $F_i$ and source $F_j, F_k$ registers
  - Flags $R_j, R_k$ indicating whether the source registers are ready
  - Names $Q_j, Q_k$ of the units (if any) producing values for $F_j, F_k$
- Status of result registers
  - For each $F_i$ the name of the unit (if any), say $P_i$ that will produce its contents (redundant but easy to check)
- The instruction status
  - Been issued, dispatched, in execution, ready to write, finished?

Condition checking and scoreboard setting

- Issue step
  - Unit free, say $U_a$ and no WAR
  - Record $Q_j, Q_k$ and $R_j, R_k$
  - Record $P_i = U_a$
- Dispatch (Read operand) step
  - $R_j$ and $R_k$ must be yes (results ready)
- Execution step
  - At end ask for writing permission (no WAR)
  - Check if $P_i$ is an $F_j, F_k(R_j, R_k= no)$ in preceding instrs. If yes stall.
- Write result
  - $U_a$ free and $P_i = 0$
Example

Load F6, 34(r2)
Load F2, 45(r3)
Mul F0, F2, F4
Sub F8, F6, F2
Div F10, F0, F6
Add F6, F8, F2

- - - - - - RAW

Assume that the 2 Loads have been issued, the first one completed, the second ready to write. The next 3 instructions have been issued (but not dispatched).

Instruction | Issue | Dispatch | Executed | Result written
--- | --- | --- | --- | ---
Load F6, 34(r2) | yes | yes | yes | yes
Load F2, 45(r3) | yes | yes | yes | yes
Mul F0, F2, F4 | yes | yes | yes | yes
Sub F8, F6, F2 | yes | yes | yes | yes
Div F10, F0, F6 | yes | yes | yes | yes
Add F6, F8, F2 | yes | yes | yes | yes

Functional Unit status
No | Name | Busy | Fi | Fj | Fk | Qj | Qk | Rj | Rk
--- | --- | --- | --- | --- | --- | --- | --- | --- | ---
1 | Int | no | | | | | | | |
2 | Mul | yes | F0 | F2 | F4 | | | | |
3 | Mul | no | | | | | | | |
4 | Add | yes | F8 | F6 | F2 | | | | |
5 | Div | yes | F10 | F0 | F6 | | | | |

Register result status
F0 (2) | F2 (1) | F4 () | F6 (4) | F8 (4) | F10 (5) | F12 ...

Dyn. Sched. CSE 471 Autumn '02 13