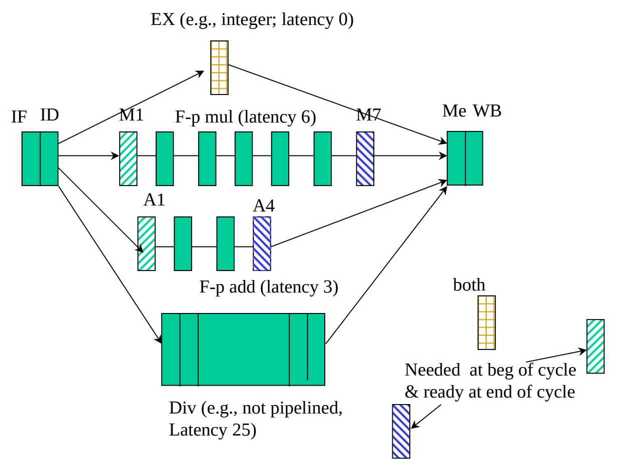
How to improve (decrease) CPI

- Recall: CPI = Ideal CPI + CPI contributed by stalls
- Ideal CPI =1 for single issue machine even with multiple execution units
- Ideal CPI will be less than 1 if we have several execution units and we can issue (and “commit”) multiple instructions in the same cycle, i.e., we take advantage of Instruction Level Parallelism (ILP)

Extending simple pipeline to multiple pipes

- Single issue: in ID stage direct to one of several EX stages
- Common WB stage
- EX of various pipelines might take more than one cycle
- Latency of an EX unit = Number of cycles before its result can be forwarded = Number of stages –1
- Not all EX need be pipelined
- IF EX is pipelined
  - A new instruction can be assigned to it every cycle (if no data dependency) or, maybe only after *x* cycles, with *x* depending on the function to be performed



Hazards in example multiple cycle pipeline

- Structural: Yes
  - Divide unit is not pipelined. In the example processor two Divides separated by less than 25 cycles will stall the pipe
  - Several writes might be “ready” at the same time and want to use WB stage at the same time (not possible if single write port)
- RAW: Yes
  - Essentially handled as in integer pipe (the dependent instruction is stalled at the beginning of its EX stage) but with higher frequency of stalls. Also more forwarding paths are needed.
- WAW : Yes (see later)
  - WAR no since read is in the ID stage
- Out of order completion : Yes (see later)

RAW:Example from the book (pg A-51)

F4 <- M	IF	ID	EX	**Me**	WB													
F0 <- F4 * F6		IF	ID	st	**M1**	M2	M3	M4	M5	M6	**M7**	Me	WB					
F2 <- F0 + F8			IF	st	ID	st	st	st	st	st	st	**A1**	A2	A3	**A4**	**Me**	WB	
M <- F2					IF	st	st	st	st	st	st	ID	EX	st	st	st	**Me**	WB

In blue data dependencies hazard

In red structural hazard

In green stall cycles

Note both the data dependency and structural hazard for the 4th instruction

Conflict in using the WB stage

- Several instructions might want to use the WB stage at the same time
  - E.g.,A Multd issued at time *t* and an addd issued at time *t + 3*
- Solution 1: reserve the WB stage at ID stage (scheme already used in CRAY-1 built in 1976)
  - Keep track of WB stage usage in shift register
  - Reserve the right slot. If busy, stall for a cycle and repeat
  - Shift every clock cycle
- Solution 2: Stall before entering either Me or WB
  - Pro: easier detection than solution 1
  - Con: need to be able to trickle the stalls “backwards”.

Example on how to reserve the WB stage
(Solution 1 in previous slide)

Time in ID stage	Operation	Shift register
t	multd	000 000 001
t +1	int	001 000 010
t + 2	int	011 000 100
t + 3	addd	110 00X 000

Note: multd and addd want WB at time t + 9. addd will be asked to stall one cycle

Instructions complete out of order (e.g., the two int terminate before the multd)

WAW Hazards

- Instruction *i* writes f-p register Fx at time *t*
  Instruction *i* + *k* writes f-p register Fx at time *t* - *m*
- But no instruction *i* + *1*, *i* +*2*, *i*+*k* uses (reads) Fx (otherwise there would be a stall)
- Only requirement is that *i* + *k* ´*s* result be stored
  - Note: this situation should be rare (useless instruction *i*)
- Solutions:
  - Squash *i* : difficult to know where it is in the pipe
  - At ID stage check that result register is not a result register in all subsequent stages of other units. If it is, stall appropriate number of cycles.

Out-of-order completion

- Instruction *i* finishes at time *t*
  Instruction *i* + *k* finishes at time *t* - *m*
  - No hazard etc. (see previous example on integer completing before multd)
- What happens if instruction *i* causes an exception at a time in [*t*-*m*,*t*] and instruction *i* + *k* writes in one of its own source operands (i.e., is not restartable)?

Exception handling

- Solutions (cf. book pp A-54 – A-56 for more details)
  - Do nothing (imprecise exceptions; bad with virtual memory)
  - Have a precise (by use of testing instructions) and an imprecise mode; effectively restricts concurrency of f-p operations
  - Buffer results in a "history file" (or a "future file") until previous (in order) instructions have completed; can be costly when there are large differences in latencies but a similar technique is used for OOO execution .
  - Restrict concurrency of f-p operations and on an exception "simulate in software" the instructions in between the faulting and the finished one.
  - Flag early those operations that might result in an exception and stall accordingly

Resources for Exploiting ILP (ct'd)

- IF and ID: Allow several instructions to be fetched, decoded, and issued (sent to the execution units) in the same cycle.
- Superscalar machines are those that have multiple instruction issues. We will distinguish later on between those that require instructions to be issued in program order and those that allow out-of-order issues.
- Note that these extensions might result in out-of-order completion of instructions. Mechanisms will be introduced in the WB stage to enforce in-order completion (commit).

Exploitation of Instruction Level Parallelism (ILP)

- Will increase throughput and decrease CPU execution time
- Will increase structural hazards
  - Cannot issue simultaneously 2 instructions to the same pipe
- Makes reduction in other stalls even more important
  - A stall costs more than the loss of a single instruction issue
- Will make the design more complex
  - WAW and WAR hazards can occur
  - Out-of-order completion can occur
  - Precise exception handling is more difficult

Where can we optimize? (control)

- CPI contributed by control stalls can be decreased statically (compiler) or dynamically (hardware)
- Speculative execution
  - Branch prediction (we have seen that already)
  - Bypassing Loads (memory reference speculation)
  - Predication

Where can we optimize? (data dependencies)

- Hardware (run-time) techniques
  - Forwarding (RAW; we have seen that)
  - Register renaming (WAW, WAR)

Data dependencies (RAW)

- Instruction (statement) S*j* dependent on S*i* if

  $O_i \cap I_j \neq \varnothing$
  - Transitivity: Instruction *j* dependent on *k* and *k* dependent on *i*
- Dependence is a program property
- Hazards (RAW in this case) and their (partial) removals are a pipeline organization property
- Code scheduling goal
  - Maintain dependence and avoid hazard (pipeline is *exposed to the compiler)*

Name dependence

- Anti dependence $O_j \cap I_i \neq \varnothing$
  - Si: …<- R1+ R2; ….; Sj: R1 <- …
  - At the instruction level, this is WAR hazard if instruction *j* finishes first
- Output dependence $O_i \cap O_j \neq \varnothing$
  - Si: R1 <- …; ….; Sj: R1 <- …
  - At the instruction level, this is a WAW hazard if instruction *j* finishes first
- In both cases, not really a dependence but a “naming” problem
  - Register renaming (compiler by register allocation, in hardware see later)