Lecture 9: More ILP

- Today: limits of ILP, case studies, boosting ILP (Sections 3.8-3.14)

ILP Limits

- The perfect processor:
  - Infinite registers (no WAW or WAR hazards)
  - Perfect branch direction and target prediction
  - Perfect memory disambiguation
  - Perfect instruction and data caches
  - Single-cycle latencies for all ALUs
  - Infinite ROB size (window of in-flight instructions)
  - No limit on number of instructions in each pipeline stage

- The last instruction may be scheduled in the first cycle

- What is the only constraint in this processor?

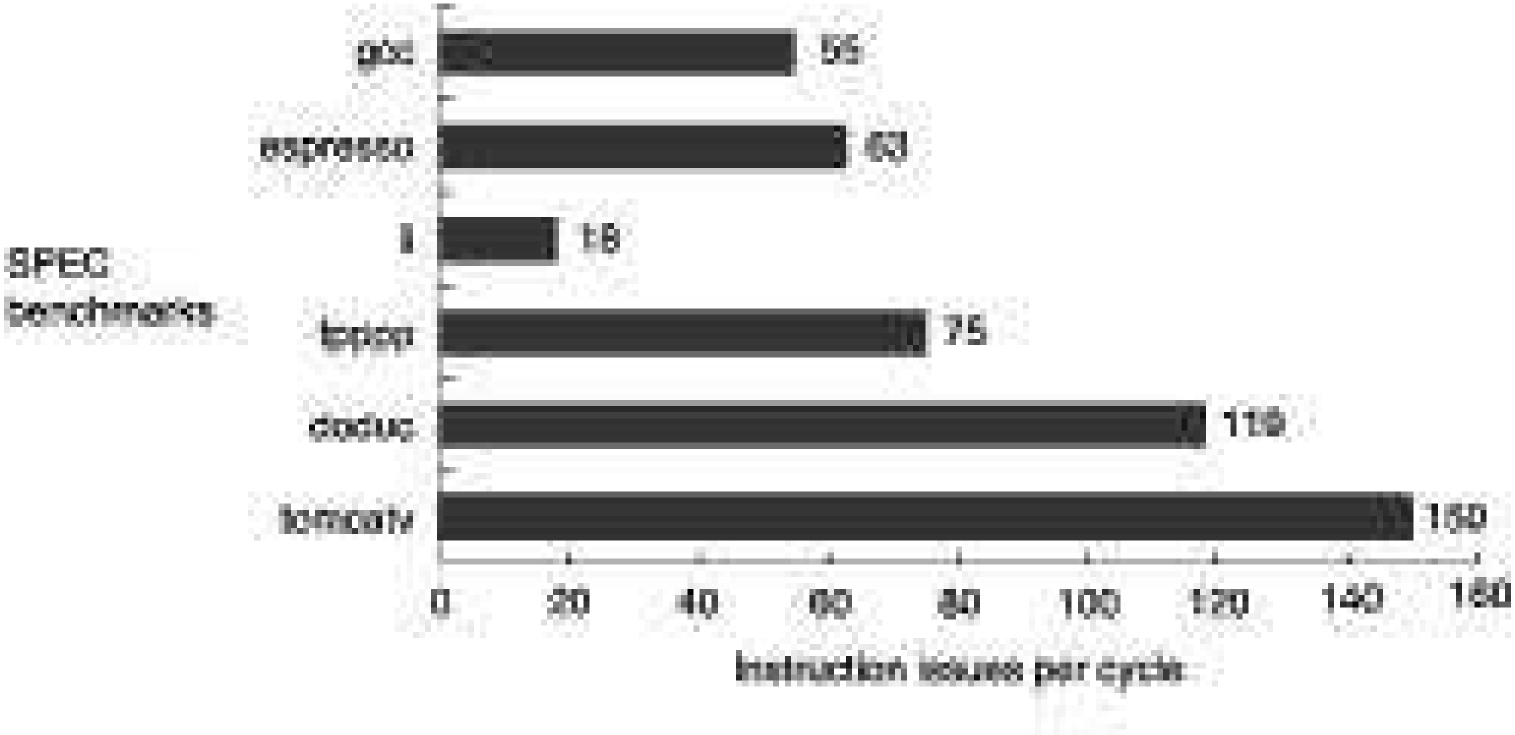
ILP Limits

- The perfect processor:
  - ➢ Infinite registers (no WAW or WAR hazards)
  - ➢ Perfect branch direction and target prediction
  - ➢ Perfect memory disambiguation
  - ➢ Perfect instruction and data caches
  - ➢ Single-cycle latencies for all ALUs
  - ➢ Infinite ROB size (window of in-flight instructions)
  - ➢ No limit on number of instructions in each pipeline stage

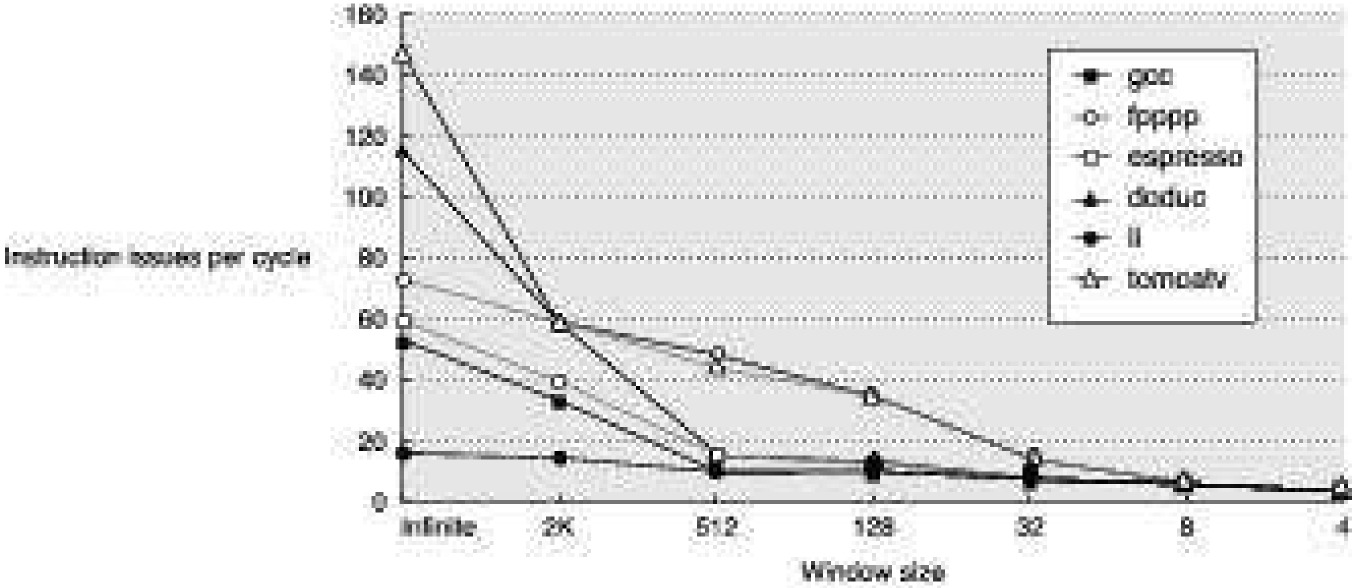
- The last instruction may be scheduled in the first cycle

- The only constraint is a true dependence (register or memory RAW hazards) (with value prediction, how would the perfect processor behave?)

Infinite Window Size and Issue Rate

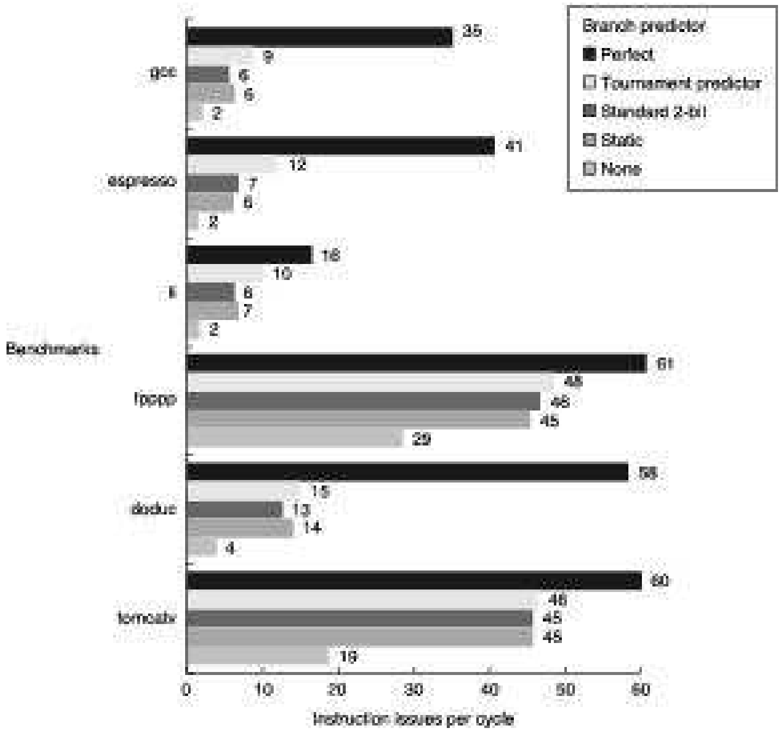


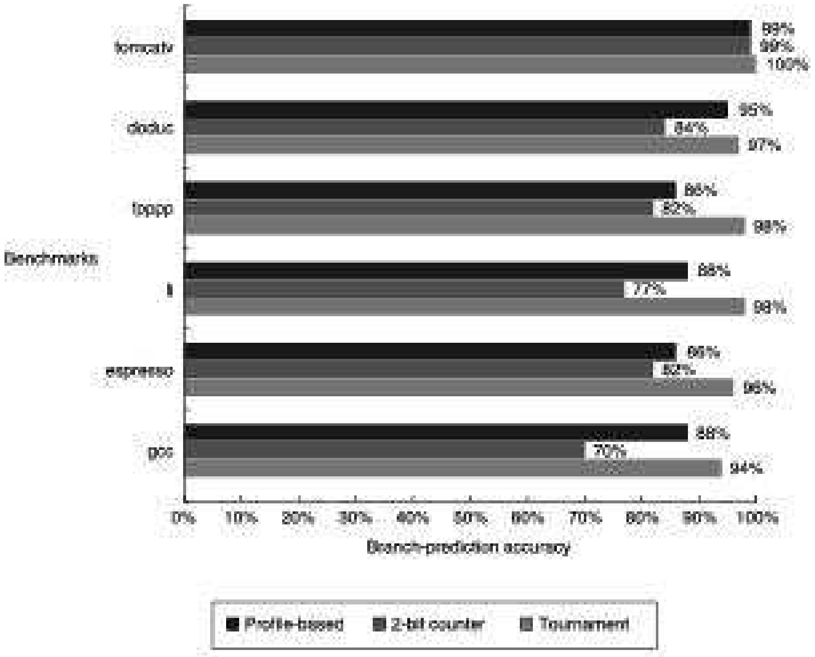
Effect of Window Size



- Window size is effected by register file/ROB size, branch mispredict rate, fetch bandwidth, etc.
- We will use a window size of 2K instrs and a max issue rate of 64 for subsequent experiments

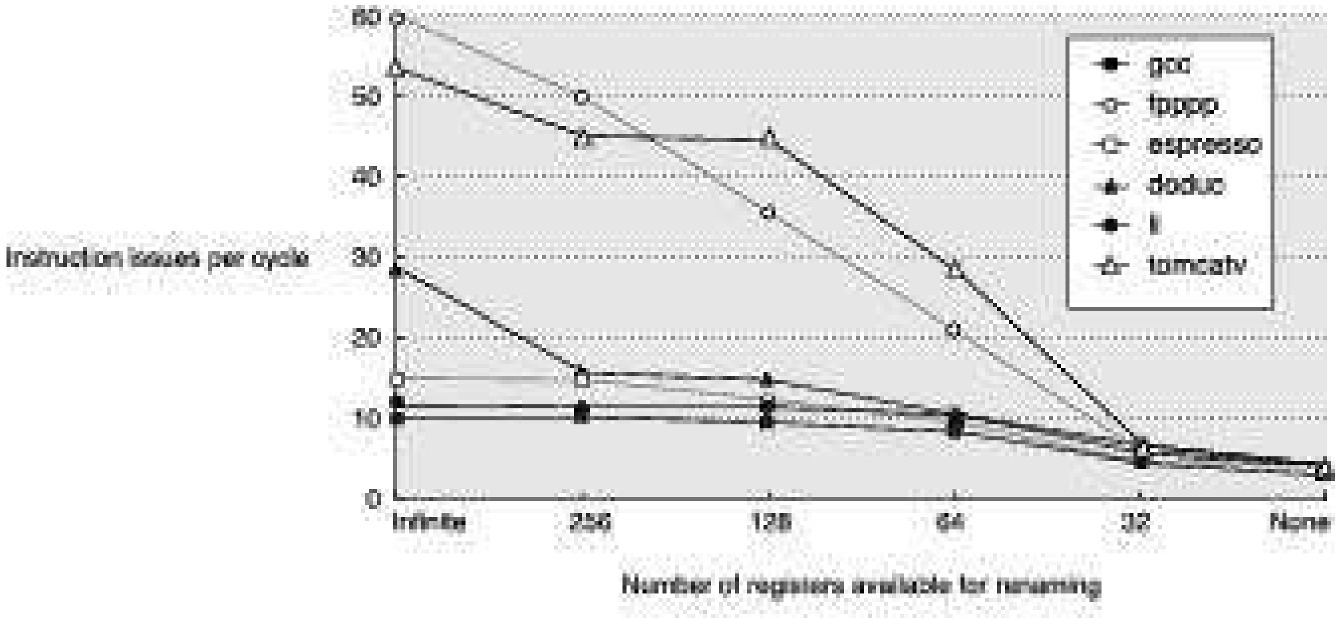
Imperfect Branch Prediction





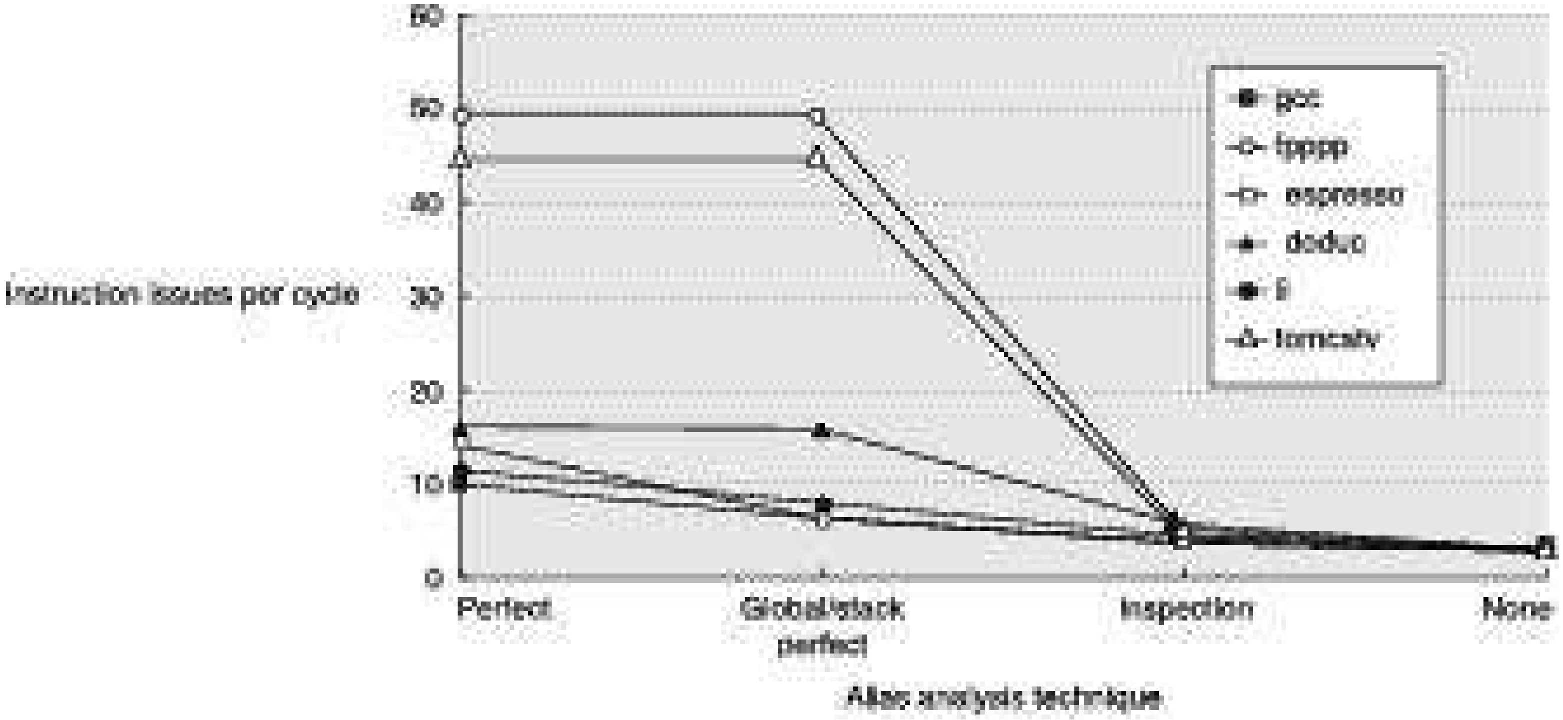
- Note: no branch mispredict penalty; branch mispredict restricts window size
- Assume a large tournament predictor for subsequent experiments

Effect of Name Dependences



- More registers → fewer WAR and WAW constraints (usually register file size goes hand in hand with in-flight window size)
- 256 int and fp registers for subsequent experiments

Memory Dependences



Limits of ILP – Summary

- Int programs are more limited by branches, memory disambiguation, etc., while FP programs are limited most by window size

- We have not yet examined the effect of branch mispredict penalty and imperfect caching

- All of the studied factors have relatively comparable influence on CPI: window/register size, branch prediction, memory disambiguation

- Can we do better? Yes: better compilers, value prediction, memory dependence prediction, multi-path execution

Pentium III (P6 Microarchitecture) Case Study

- 14-stage pipeline: 8 for fetch/decode/dispatch, 3+ for o-o-o, 3 for commit → branch mispredict penalty of 10-15 cycles

- Out-of-order execution with a 40-entry ROB (40 temporary or virtual registers) and 20 reservation stations

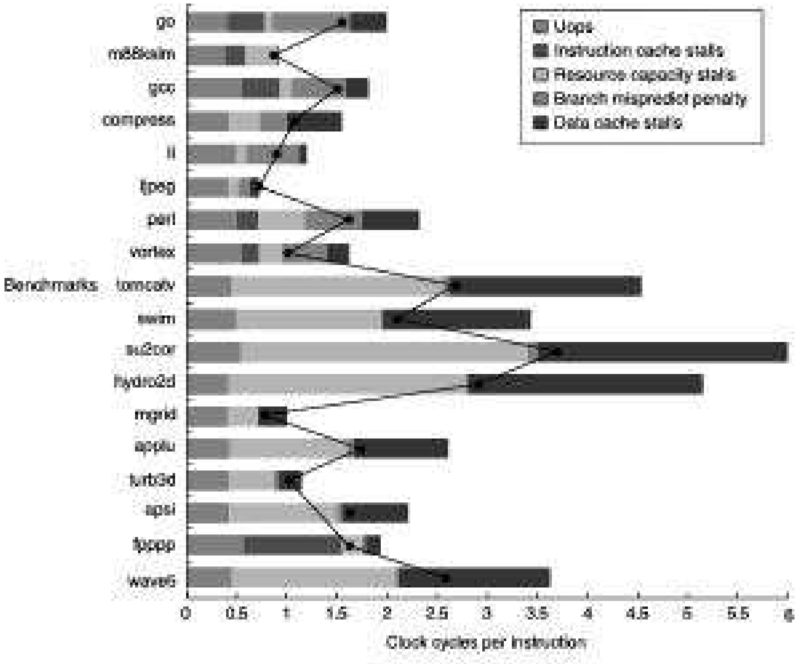
- Each x86 instruction gets converted into RISC-like micro-ops – on average, one CISC instr → 1.37 micro-ops

- Three instructions in each pipeline stage → 3 instructions can simultaneously leave the pipeline → ideal CPµI = 0.33 → ideal CPI = 0.45

Branch Prediction

- 512-entry global two-level branch predictor and 512-entry BTB → 20% combined mispredict rate
- For every instruction committed, 0.2 instructions on the mispredicted path are also executed (wasted power!)
- Mispredict penalty is 10-15 cycles

CPI Performance

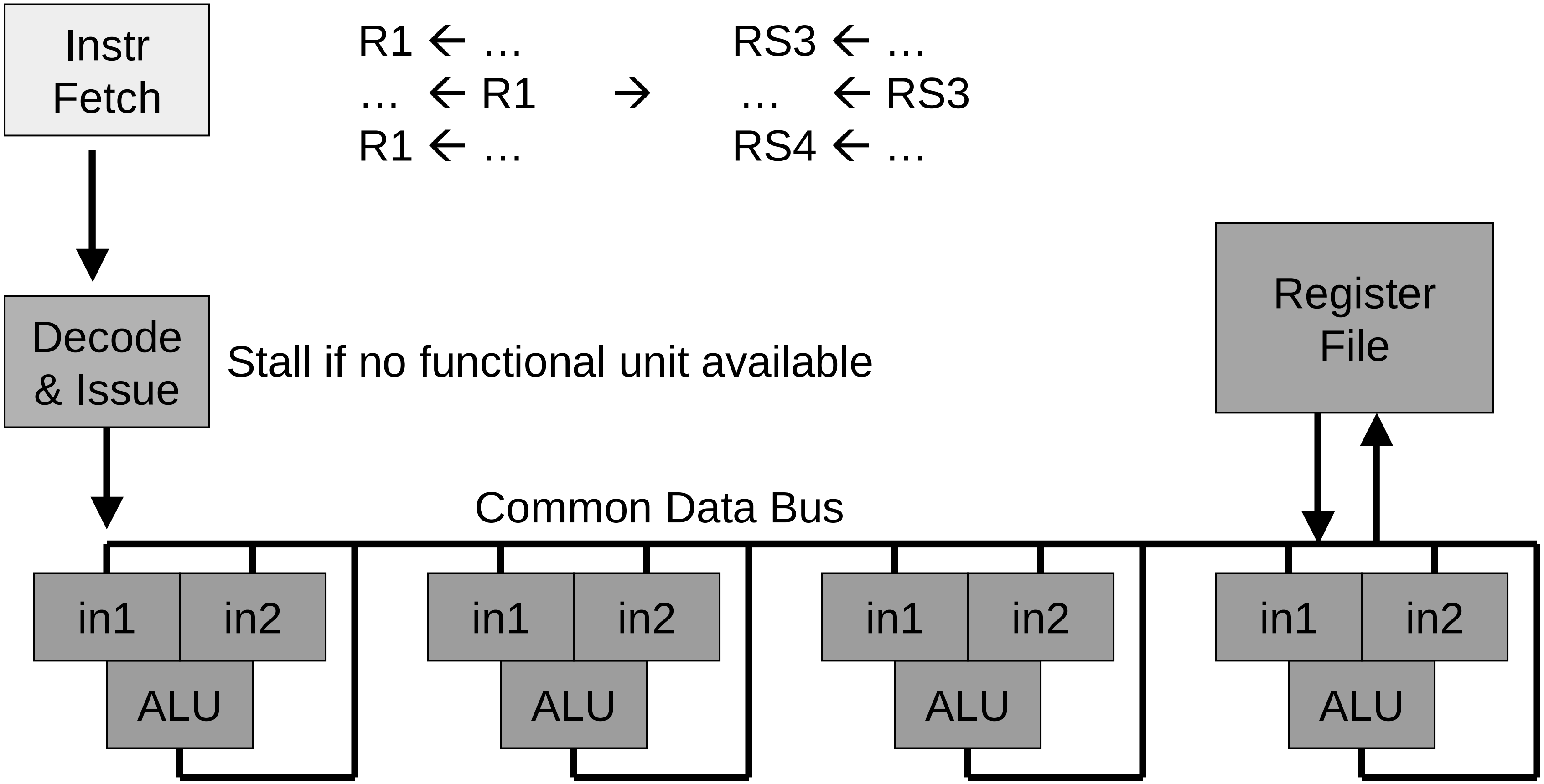


- Owing to stalls, the processor can fall behind (no instructions are committed for 55% of all cycles), but then recover with multi-instruction commits (31% of all cycles) → average CPI = 1.15 (Int) and 2.0 (FP)
- Overlap of different stalls → CPI is not the sum of individual stalls
- IPC is also an attractive metric

Alternative Designs

- Tomasulo’s algorithm
  - When an instruction is decoded and “dispatched”, it is assigned to a “reservation station”
  - The reservation station has an ALU and space for storing operands – ready operands are copied from the register file into the reservation station
  - If an operand is not ready, the reservation station keeps track of which reservation station will produce it – this is a form of register renaming
  - Instructions are “dispatched” in order (dispatch stalls if a reservation station is not available), instructions begin execution out-of-order

Tomasulo’s Algorithm



Instr Fetch

R1 ← …
… ← R1
R1 ← …

→

RS3 ← …
… ← RS3
RS4 ← …

Decode & Issue

Stall if no functional unit available

Register File

Common Data Bus

Instrs read register operands immediately (avoids WAR), wait for results produced by other ALUs (more names), and then execute, the earlier write to R1 never happens (avoids WAW)

Improving Performance

- What is the best strategy for improving performance?
  - ➢ Take a single design and keep pipelining
  - ➢ Increase capacity: use a larger branch predictor, larger cache, larger ROB/register file
  - ➢ Increase bandwidth: more instructions fetched/ decoded/issued/committed per cycle

Deep Pipelining

- If instructions are independent, deep pipelining allows an instruction to leave the pipeline sooner

- If instructions are dependent, the gap between them (in nanoseconds) widens – there is an optimal pipeline depth

- Some structures are hard to pipeline – register files

- Pipelining can increase bypassing complexity

Capacity/Bandwidth

- Even if we design a 6-wide processor, most units go under-utilized (average IPC of 2.0!) – hence, increased bandwidth is not going to buy much

- Higher capacity (being able to examine 500 speculative instructions) can increase the chances of finding work and boost IPC – what is the big bottleneck?

- Higher capacity (and higher bandwidth) increases the complexity of each structure and its access time – for example, access times: 32KB cache in 1 cycle, 128KB cache in 2 cycles

Future Microprocessors

- By increasing branch predictor, window size, number of ALUs, pipeline stages, IPC and clock speed can improve → however, this is a case of diminishing returns!

- For example, with a window size of 400 and with 10 ALUs, we are likely to find fewer than four instructions to issue every cycle → under-utilization, wasted work, low throughput per watt consumed

- Hence, a more cost-effective solution: build four simple processors in the same area – each processor executes a different thread → high thread throughput, but probably poorer single application performance

Thread-Level Parallelism

- Motivation:
  - ➢ a single thread leaves a processor under-utilized for most of the time
  - ➢ by doubling processor area, single thread performance barely improves

- Strategies for thread-level parallelism:
  - ➢ multiple threads share the same large processor → reduces under-utilization, efficient resource allocation Simultaneous Multi-Threading (SMT)
  - ➢ each thread executes on its own mini processor → simple design, low interference between threads Chip Multi-Processing (CMP)

Title

- Bullet