CompArch

Design Metrics

- Energy
- Power
- Performance
- Security
- Cost
- Power Efficiency
- Reliability

- Focus on common case: overall speed increases even if specific speed decreases.
- Amdahl's Law: $\text{speedup} = \frac{1}{\text{sequential} + \frac{1-\text{sequential}}{\text{speedup}_{\text{enhanced}}}}$
- Adding enhancements means lower transistor budget, more localised heat, slower clock freq, ... Might affect common case.
- $$\frac{1}{\text{performance}} = \frac{\text{time}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}}$$
  - Instruction count is affected by the ISA and compiler tech.
  - CPI is affected by micro-architecture and ISA.
  - Cycle time is affected by circuit design and micro-architecture.

ISAs

Each ISA is split into the System ISA and the User ISA. System ISA is privileged in some way.

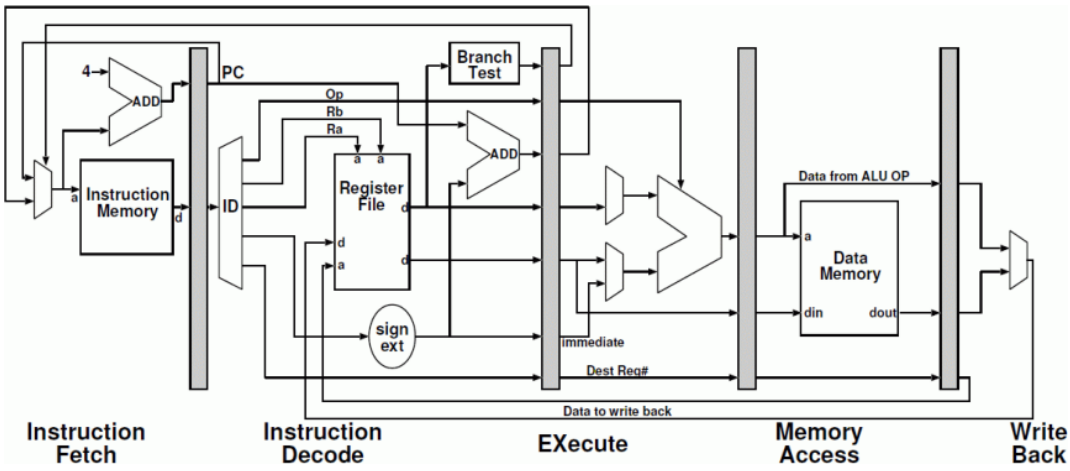
ISAs either break binary compatibility or carry architectural baggage. Embedded ISAs are more flexible as binary compatibility is less of an issue (mostly purpose-made binaries for product lifetime).

Can use eg. JVM to remove reliance on specific ISA, or allow the processor to dynamically translate (Intel CISC-to-RISC).

Microcode is code for running the ISA – miniature control computer. Useful for CISC but introduces performance and complexity overhead.

RISC ensures instructions are simple for faster fetching, easier pipelining. Relies on compiler to schedule and register-allocate.

Pipelining



Hazards

Hazards are phenomena that require stalling in order to preserve program semantics.

- Structural Hazard (CPU resource conflicts, like limited ALU ports)

- Data Hazard (inter-instruction dependencies)
- Control Hazard (instructions changing the PC like `jmp`)

Instruction dependencies exist between **any** ordered pairs of instructions, regardless of distance, and make **reordering** harder.

- True data dependence (result is truly required)
  - **RAW**: 1 and 2.
- Name dependence (same register used for multiple computations)
  - **WAW**: 1 and 4.
  - **WAR**: 2 and 3, 2 and 4.

```
1 ADD R1, R2, R3
2 SW  R1, 0(R4)
3 SUB R4, R3, R5
4 ADD R1, R2, R3
```

Structural hazards can be avoided entirely in hardware/ISA (eg. avoid structural by having worst-case number of on-chip resources), but it can slow the common case or simply be too expensive. More complex but faster to handle issues as they come.

Data hazards can be avoided by adding data-forward paths, or scheduling code to prevent data dependencies from becoming data hazards (**instruction scheduling** by compiler/hardware). Hazards from name dependencies can be solved by **register renaming** (compiler/hardware).

- Software interlock: compiler inserts instructions on instructions causing a hazard.
- Hardware interlock: pipeline stalls when hazards are detected.

Control Hazards can sometimes be avoided by branch prediction. In the simplest pipeline, just stall the fetches until the outcome of the branch is known. For simple tests ($r = 0$), could move test and target-address calculation into the decode stage. Requires dedicated hardware and logic for switching to, but reduces the branch delay slot by 1 cycle.

Exceptions

Page fault, illegal op-code, memory protection violation, arithmetic exception, I/O interrupt, ...

Often want to be able to restart execution after handling the exception: a pipeline supports **precise exceptions** if it guarantees that all instructions prior to the faulting instruction have been executed and all those after it have not begun execution.

Simple approach is to tag each instruction with its PC and a flag for whether it raised an exception. Execute stage sets the flag, and stages don't perform side-effects for instructions with the flag set. When the write-back stage sees a faulted instruction it flushes the pipeline.

Alternatively, hand over control to dedicated hardware when an exception occurs (eg. for TLB misses) without flushing the pipeline.

Multicycle operations

Not all instructions can/should complete in a single cycle (eg. floating point arithmetic, load/store operations).

Use multiple execution pipelines, with all the issues arising there: new hazards, harder exception handling, etc.

Limits

- Deeper pipelines have more expensive stalls/flushes.
- Cycle time determined by worst-case stage time.
- Hard to clock each stage at the same time.
- More stages increases complexity (forwarding paths, harder exceptions, ...)

- Pipelining registers introduce overheads.

Branch Prediction

138g