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: 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.

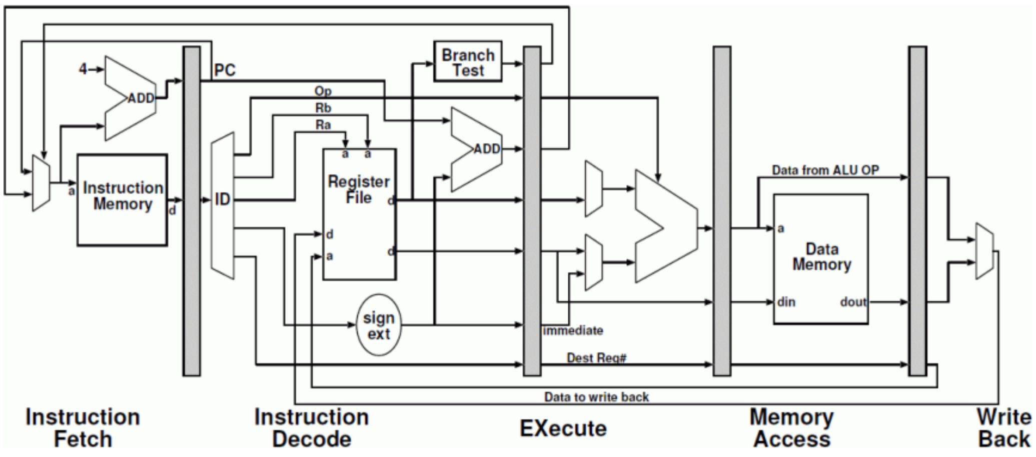
ISAs either break binary compatibility or carry architectural baggage. Embedded ISAs are more flexible as binary compatibility is less of an issue (mostly purpose-made binaries for product lifetime).

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. Can embrace the branch delay slot and force compilers to place an instruction there.

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

- Condition codes: branch instructions have attached flags for the ALU (test against 0, allow overflow, ...)
- Condition registers: comparison operations store results in a given register, branches use those.
- Compare-and-branch: comparison and branch within a single instruction (Java bytecode).

Static Predictor

Always guess 'branch hit' or 'branch miss'. Can improve results by allowing branch instructions to be tagged with a bias bit, that changes which way the processor guesses. Bit can be set by compiler, maybe using a profiling run to get an approximate distribution.

One-level Predictor

Use a table of registers, indexed by a portion of the *address of the branch*. 1 bit entries store taken/not taken. 2-bit entries can use a counter, to add a bit more consistency. Any more than 2 bits aren't much more effective.

Don't care about collisions in the table as prediction is inaccurate anyway.

Two-level Predictor

Local-history two-level predictors use a table of shift registers (history registers, like 01101) storing the branch's latest taken/not-takens. Also keep a table **per branch**, indexed by that branch's history register with entries being 2-bit counters like above.

First level is the shift register holding a branch's history. Second level is using that history register to look up the prediction.

Global history predictor uses a single shift register to hold the global history of branches, rather than a specific hash of them. Second level is the same, just the first level is less interesting.

The use of shift registers with the counters aims to **capture patterns**.

Check in supo notes that Daniel agreed with the global predictor definition.

Tournament Predictor

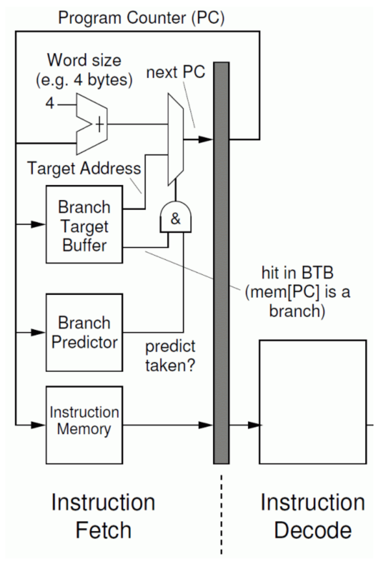
Local and Global predictors are effective in different cases. Use both, and pick whichever has been most accurate for a specific branch in the past.

Limits

- Need a training period of low accuracy before predictors get up to speed.
- Will always be malicious inputs.
- Can be very expensive to implement (two big tables for each branch...)
- Using hashing on the branch PC, so can have negative/positive/neutral aliasing of branches.

Branch Target Buffer/Cache

As well as predicting whether a branch will be taken, can also predict where it'll go: if we predict it being taken, save time working out the destination.
Keep a table indexed by the branch address with entries being cached target addresses.
All this happens during instruction fetch, so if we guess correctly on branch prediction and the cached value is correct, we don't incur **any** branch penalty.



Return Address Predictors

Functions can be called from all over the place so predictions are normally inaccurate.

Just store a stack of PCs from before branches.

Avoiding Branches

ARM conditional instructions allow transforming control dependencies (branches) into data dependencies (normal instructions with conditional flags). End up with nullified pipeline instructions but will never have to flush the pipeline. Only works for simple branches where it's worth inlining. In the end, branch prediction is just accurate enough that this is unnecessary.

Superscalar Processors

Exploit instruction-level parallelism (ILP) to replace stalls with instructions.

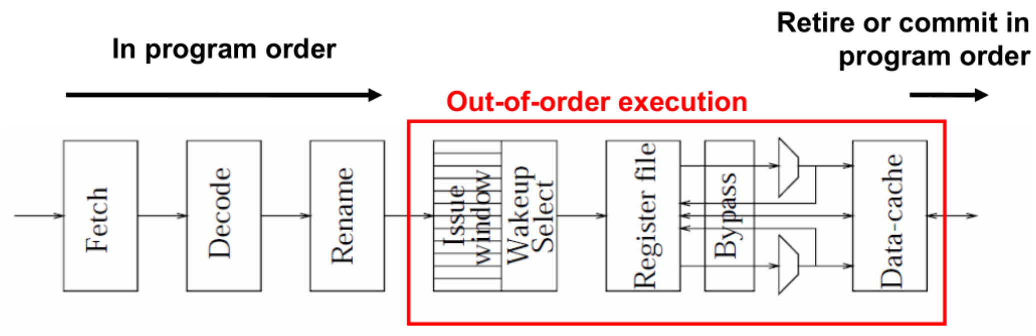
Superpipelined

Further subdivide pipeline macro-stages into M micro-stages that **all do the same thing**: each clock allows for M instructions to complete.

Check supo notes for why superparallel isn't used.

Reminder that pipeline stages can be divided as far as we want: the 5-stage pipeline is human-friendly but each stage can be divided into subtasks arbitrarily.

Superscalar



Run P pipelines in parallel. IPC $\leq P$, clock period stays the same as in a pipelined processor.

Upper limit on ILP is due to true data dependencies. Can exceed even this by using special multi-input or extra-fast functional units to run multiple cheap instructions within a cycle (eg. `ADD`).

Processor can't sustain an execution rate faster than the fetch rate.

Fetch Techniques

Need multi-ported instruction caches for fetching up to P instructions per cycle. Can be complex if the instructions don't **align** with the cache lines (eg. we fetch 4 instructions starting from 2 when the cache lines start from 0, 4, ...).

Register Renaming

Can reduce the effects of name dependencies (using the same register) by having a large set of **physical registers** which are mapped onto by the **architectural registers** which are exposed to the program.

The rename stage keeps a list of free registers and the current mapping from architectural to physical registers and rewrites instructions just before they enter the out-of-order portion of the pipeline.

Output registers are **always renamed**, as otherwise we're not actually helping hide name dependencies. Essentially rewrites into SSA form.

Out-of-order execution

Compiler is often unable to schedule instructions optimally as it can't disambiguate memory addresses, can't predict branches, works with a limited number of registers.

Maintain a buffer of instructions waiting to execute. Issue instructions from the buffer in any order once they're ready to execute (all operands are available and a functional unit is free).

When an instruction is scheduled, its destination register is marked as ready for during the next cycle, so subsequent instructions can get scheduled.

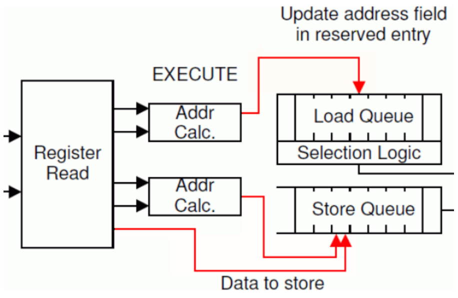
Usually want to schedule loads as early as possible, as they take a long time and often free up lots of other instructions.

- Load Bypassing: execute loads before stores **if they access different memory locations**.
- Store-to-Load Forwarding: if a load reads from the same location as a store, just forward the stored value to the load instruction – completely skip memory read.
- Load-to-Load Forwarding: forward data from an earlier read to a later one.

Load/Store Queues

Inbetween execute and memory access, keep two queues for load/store instructions. Each queue is in program order, indexed by the instruction address and containing the instruction. Operands might not have been calculated (queue could contain `4: STORE 3 0x...` and `5: STORE 4 ?`, where `?` will get updated when the out-of-order address calculation happens).

Stores can't be undone, so need to ensure that exceptions from earlier instructions are handled before performing a store, as well as that any speculation (of data or branch) has been confirmed.



- Always execute store instructions in program order: never reorder. Ensures that exceptions and mispredicted branches are handled.
- Before performing a load (moving it out of the load queue), search the store queue for a store to the same memory location. If there's a matching (youngest) store, skip the memory read and just use the value that's going to be stored (store-to-load forwarding). Stores might not have addresses calculated, so we need to treat loads as speculative (might get rolled back later).
- When a store address is calculated, update the instruction in the store queue and search the load queue for speculative loads to the same address – need to roll them back.

Load/Store Queues are expensive as they compare wide memory addresses and need to be content-addressable, and usually multi-ported.

Avoiding Load/Store Queues

Alternate approach: disallow store-to-load forwarding, and just flush the pipeline when out-of-order store/load instructions are detected to have been performed out of order.

Use a hash table of counters indexed by accessed addresses to detect problems: increment the counter when a load is issued and decrement it on commit. If a store commits and the counter is non-zero, flush and restart the pipeline.

Precise Exceptions and Rollbacks

Exceptions are detected when processing one instruction in the instruction window, some of which have been performed and some not. Should only allow commits if no potential for an exception.

The Reorder Buffer

Keep an array of **register results** (not instructions) at the end of the pipeline in program order, the relevant slot being updated when an instruction executes. When all earlier instructions have completed, commit the results to the architectural register file.

When committing, if we detect a mispredicted branch or exception we just clear the reorder buffer and restart the pipeline.

Now have **two locations** for register contents: when an instruction uses operands, need to look in the reorder buffer for the youngest change to the register, and if it's not present, fall back to the real architectural register file.

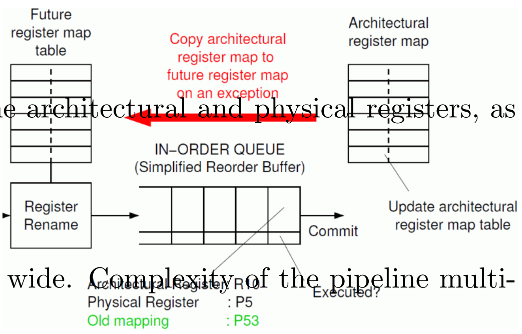
Unified Register File

Use a single large register files instead of an auxiliary buffer, and use two mapping tables: each table is the state of a register-renaming.

- Front-end table is used by all running instructions and provides the latest speculative register values.

- Back-end table is the correct, committed mapping.

When there's an exception or misprediction, can just directly copy the back-end table over the front-end table to replace the speculative registers.



Future register map table
Copy architectural register map to future register map on an exception
Architectural register map
IN–ORDER QUEUE (Simplified Reorder Buffer)
Register Rename
Commit
Update architectural register map table
Architectural Register R10
Physical Register : P5
Old mapping : P53
Executed?

Need an in-order queue representing instructions, holding the architectural and physical registers, as well as the old physical register for freeing up.

Limits

Diminishing returns from exploiting ILP when pipelines get wide. Complexity of the pipeline multiplied, so harder to optimise and verify.

Requires high instruction fetch rate, imposes knock-on performance requirements on other components. Still centralised, so doesn't scale well.

VLIW

Very Large Instruction Width processors move the scheduling complexity out of the processor and into the compiler.

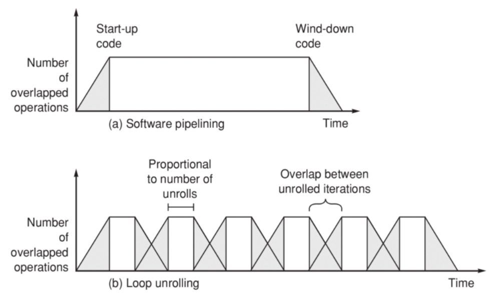
Important idea is that all performance techniques used on VLIW have to be implemented in the commpiler – the below techniques are **compiler techniques**, not hardware ones.

Idea is to use an ISA which supports packing a number of independent instructions into a single long instruction. Each slot in the instruction has a fixed function (integer op, load/store, floating-point, ...).

Latency subtleties come into play: operations have different durations, so the compiler is forced to assume worst-case delay for each (assume eg. 3 instruction delay before memory ops are completed). If an operation takes longer (cache miss), the pipeline is just stalled.

Local Scheduling

Scheduling within a basic block has limited ILP (less than 2 normally). Basic blocks are typically 5 instructions long (branch every 5 instructions).



Number of overlapped operations
Start-up code
Wind-down code
(a) Software pipelining
Time
Proportional to number of unrolls
Overlap between unrolled iterations
Number of overlapped operations
(b) Loop unrolling
Time

Loop Unrolling

Replicate the loop body multiple times and change the loop condition to match. Adjust memory access offsets to account for being in a different block. Increases the size of the basic block in the loop body, so higher ILP inside the body.

Requires some setup/teardown where there's very limited ILP (eg. need to load/store values before starting/finishing and there's no non-memory operations we can do in the meantime).

Software Pipelining

Remember this is specifically a **compiler** technique for **VLIW**.

Overlap different iterations of the loop so that we avoid the setup/teardown associated with loop unrolling. Each VLIW instruction has a slot filled by each iteration.

Register renaming can get difficult if we need a result computed by an old iteration, as the register will get overwritten.

- Can add explicit register moves to store the results we need.

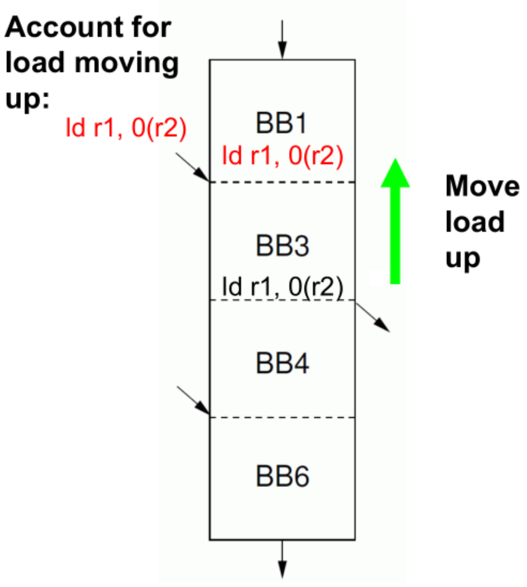
- Can do unrolling first, to make the kernel larger and bring the result into scope.
- Use a **Rotating Register File**: each iteration of the loop gets a fresh set of registers.

Global Scheduling

Can move instructions between basic blocks to increase ILP, but it's complex: the state space is huge and finding an approximately optimal solution is expensive.

Trace Scheduling

Focus on the common case: find the most likely **path** through the basic block graph using eg. annotated code/profiling, and name it the **trace**.
The trace has side entrances and exits, caused by branches of other paths leading into the most probable one or branches moving us off the trace.
Can optimise instruction layout within the trace rather than globally, then insert book-keeping instructions to compensate for the code motion when we move into/out of the trace. Moving in and out is expensive, but following the trace directly has a very high ILP.



Conditional Instructions

Conditional instructions are effective for VLIW as they can fill slots in instructions that would otherwise be blank, even if they don't end up being executed.

Memory Reference Speculation

Notable because it's the compiler speculating about **runtime behaviour**. Requires hardware support.

Can move instructions up the basic block graph, with the assumption that later blocks will be run: then use a special instruction to indicate that if the blocks aren't run, the processor should restart to preserve semantics and exception behaviour.

Egẇe can move loads before stores if the compiler guesses they won't use the same address, but then the compiler inserts an instruction later that tells the processor to check the addresses were different, and if not, restart.

Multithreaded Processors: Concurrency

Not parallelism, just running multiple threads on the same core. Often care about throughput over performance of a single thread, ie. in server workloads.

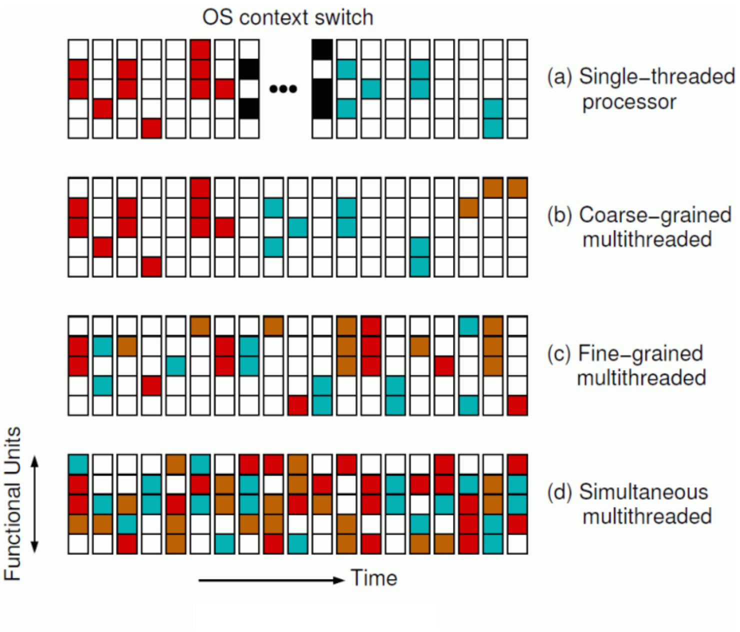
ILP is relatively fine-grained parallelism: ensuring we get maximal throughput of instructions. TLP is coarser, based on independent workloads.

Coarse-grained Multithreading

Switch to an alternative thread when the active thread stalls (cache miss, IO, ...). Aim is to hide cache miss penalties and maintain higher utilisation.

On a switch, we flush the pipeline, store any state, reload the new state, and start fetching new instructions. Need to ensure side-effects of currently executing instructions are either saved or rolled back.

- Shorter pipelines reduce thread switch penalty, as we don't need to refill the pipeline as much.
- Can use a **thread-switch buffer** to store a few instructions from each thread to avoid instruction cache misses.
- **Pipeline registers** on each pipeline stage can hold in-execution state for a few threads at a time, then swap them in and out of the main pipeline quickly. Prevents the need to refill the pipeline.


OS context switch
(a) Single–threaded processor
(b) Coarse–grained multithreaded
(c) Fine–grained multithreaded
(d) Simultaneous multithreaded
Functional Units
Time

Fine-grained Multithreading

A new thread is selected on each clock cycle: different instructions in the pipeline belong to different threads. Use eg. round robin to select threads.

Static thread schedules can be wasteful if there aren't many threads, but can simplify dependency-handling logic: if we know the spacing between each thread, we can remove eg. data-forwarding.

Fine-grained multiprocessors without caches can provide strict and predictable performance guarantees, compared to a pipeline processor with caches and coarse thread-switching.

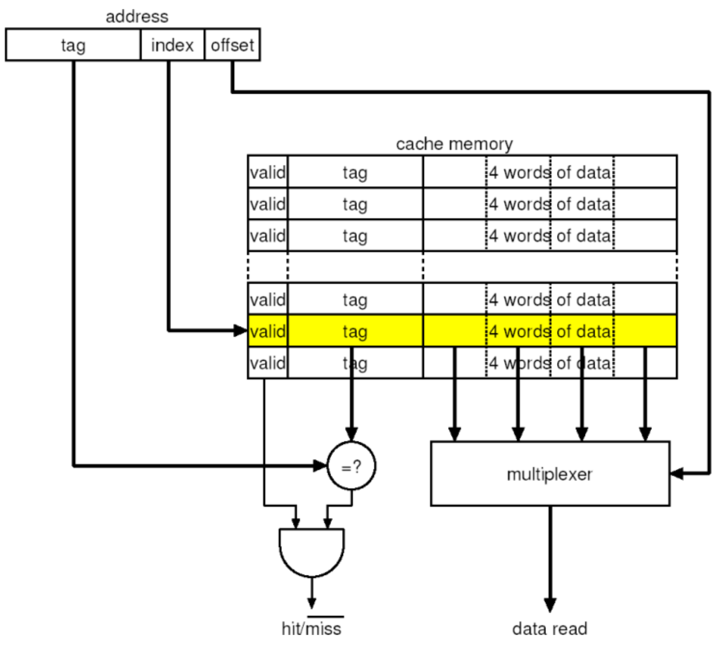
Can use dynamic thread-selection policies, or avoid switching to stalled threads, etc.

Simultaneous Multithreading

Superscalar processors with multithreading and out-of-order execution can allow for instructions from different threads to be in-flight simultaneously.

Caches

Can use software-controlled caches/scratchpads for realtime systems, or hardware managed caches. Smaller is faster.
Unified cache is data+instructions combined (level 2+), level 1 is separated caches. Separated isn't just faster, also means **can access instructions and data at the same time**.
Block = Cache Line
Low associativity is fast but bad because there's only one place for each block, so evictions are frequent. High associativity is good because we search caches in parallel which is fast and allows multiple blocks (expensive hardware). Fully associative is normally bad.



Direct Mapped Cache

Each block can only live in one place in the cache: the block address 'mod' the number of blocks in the cache. Fast, but suffers from collisions: cache lines are evicted even when there are free entries. Middle bits (index) are used as index into the cache. Low bits (offset) are used as cache line index. High bits (tag) are used to check for validity.

Fully Associative Cache

Data can be stored in any location, not just a single one: have to check tag against all entries in parallel, so expensive to implement (but very fast).

Set Associative

Combines direct and associative caching: directly identify fully-associative sets of blocks. Can be the fastest type of cache.

For an n-way set associative cache, there are multiple sets of n cache lines. Each block is direct-mapped to a set, but can be held in any location within a set (fully-associative, parallel lookup).

A block's **tag** maps to a set. Any **way** in the set can contain the block.

Ways can be held in different caches, maybe with different latencies: ideally want the most frequently accessed cache line to be in the lowest-latency way.

Block Replacement

Always replace blocks in a direct-mapped cache, but set-associative and fully associative need policies.

Least Recently Used takes advantage of temporal locality, but can be expensive to implement. Random performs well on average, approximate LRU schemes exist.

Write Policies

- Write Through: write to cache and higher-level memory.
- Write Back: only write modified cache blocks when they're evicted.

If data isn't in the cache when we write to a memory address:

- Allocate Write: read the data into cache, then overwrite it.
- No Allocate Write: don't load lines to the store (lines are only loaded on a read miss).

Usually write-through + allocate-write and write-back + no-allocate-write.

Inclusive caches ensure that addresses in higher-level memory are a superset of address in lower-level memory. Exclusive caches don't enforce this. Inclusion is good if the higher cache is much bigger, otherwise exclusion is more efficient. Instructions are usually exclusive.

If an inclusive higher cache **shadows** the tags of a lower cache, it means that it keeps a directory of the tags contained in the lower cache (L2 cache shadowing the L1 cache tags means L2 knows what addresses are in L1, but not the values).

Cache Misses

The three reasons for cache misses (the three C's):

- **Compulsory**: A compulsory miss happens when a block is brought into the cache for the first time.
- **Capacity**: A capacity miss occurs when the number of blocks exceeds the capacity of the cache.
- **Conflict**: Direct mapped and set-associative caches suffer conflict misses on collisions.

Optimisations

Write Buffers sit before a cache/main memory and buffer writes to it: don't write to the slow memory in case the value gets overwritten soon, or we write to an adjacent memory location and can merge writes. Need to look in the buffer when reading in case there's a buffered value.

L1 caches are normally small with low associativity. L2+ caches are higher associativity.

Victim Caches are small, fully-associative caches for storing lines evicted from L1. Reduces **conflict misses**.

Non-blocking Caches work with out-of-order execution to continue serving data even while servicing a miss for a different instruction. **'Hit under miss'** allows servicing hits while handling a miss, **'Miss under miss'** allows handling multiple misses at the same time. Can reduce impact of a cache miss if there is enough ILP.

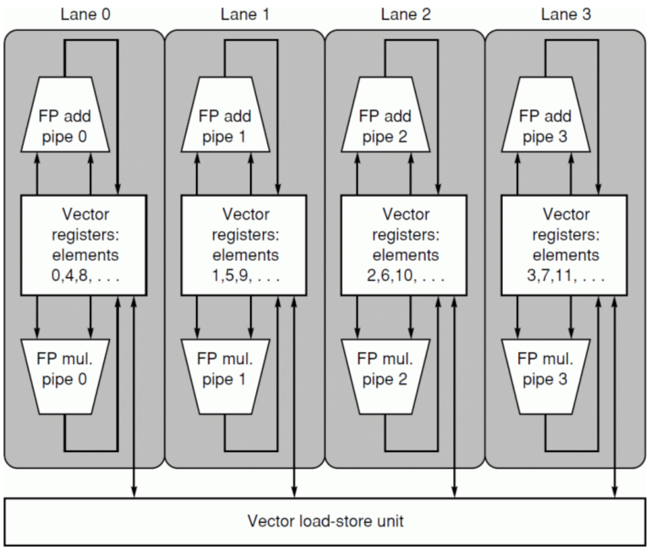
Loop fusion, combine two loops on the same array into one for higher temporal locality. **Loop fission**, split loops on different arrays to save cache space. n-dimensional loops should use indices in an order that leads to sequential memory accesses.

May need to insert padding between large arrays in case the memory addresses of the arrays map to the same locations in cache (eg. contiguous arrays of length 2KB will all map to the same index in a direct-mapped cache).

Memory prefetching can be done in parallel by the processor if hinted. Processor can do automatic prefetching by fetching sequential blocks after a miss. More complicated schemes.

Vector Processors

Vector processors provide operations acting on vectors, rather than individual registers/memory locations. Not like VLIW, as we only run one instruction at a time: the operands are just larger.
Use vector registers (fixed size), multiple entries with one name (lots of ports on register file). Also have scalar registers, for providing eg. operand values. Functional units specifically designed to operate on such registers.
Vector operations are dense and predictable, so are easy to optimise. Massive reduction in number of instructions fetched/decoded/executed.


Lane 0
Lane 1
Lane 2
Lane 3
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0,4,8, . . .
Vector registers: elements 1,5,9, . . .
Vector registers: elements 2,6,10, . . .
Vector registers: elements 3,7,11, . . .
FP mul. pipe 0
FP mul. pipe 1
FP mul. pipe 2
FP mul. pipe 3
Vector load-store unit

Functional unit are **pipelined** and split across **lanes**: elements of vectors are interleaved across lanes. After the pipelines start producing, the results emerge from each lane in order, so they can be easily written to the destination register.

Complexities

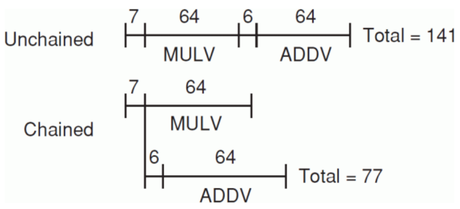
Requires extremely high memory bandwidth.

Not all vectors are full of data: require an attached scalar register containing the vector length.

Need to load data from memory separated by a fixed stride (eg. 2D arrays). Memory units support reading/writing with a scalar stride per vector element.

Optimisations

Chaining: Reduce the cost of **read-after-write** dependencies between operations by starting the second instruction when the first operation has finished with the first element.
Tailgating: Chaining but applied to **write-after-read** hazards: allow overwriting the elements of one vector as soon as they've been read.
Vector conditional execution: want to handle per-element conditionals. Use a vector mask (vector containing booleans) to control execution of an operation.
Pointers: support gather/scatter instructions which are load/store but operate on each element of a vector.



```
MULV v3, v1, v1
// Chain v3 (RaW)
ADDV v4, v3, v1
// Tailgate v3 (WaR)
MULV v3, v2, v2
```

SIMD ISA Extensions

Often perform vectorisable computation on general purpose processors (multimedia, crypto, DSP, ...). Add SIMD instructions to general-purpose ISAs.

Use wide registers (eg. 128 bit scalar) as a short vector (8 16-bit values). Special FUs for processing them.

Chip Multiprocessors

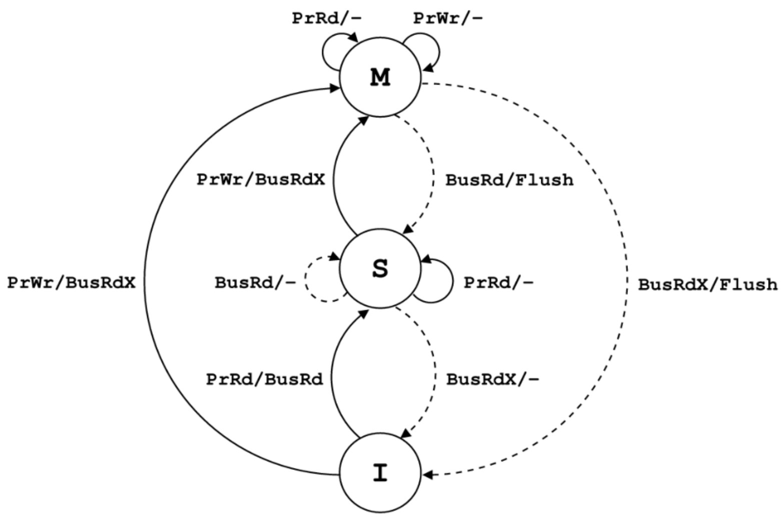
Cache Coherence

Ensuring multiple cores can't see different values at a memory location at the same time (if one core writes and it gets stuck in cache and not written back to memory).

Write-through caches are easy, as we can always get the latest value from memory. However, as each write generates bus traffic, they scale badly. Write-back caches are more commonly used on multiprocessors.

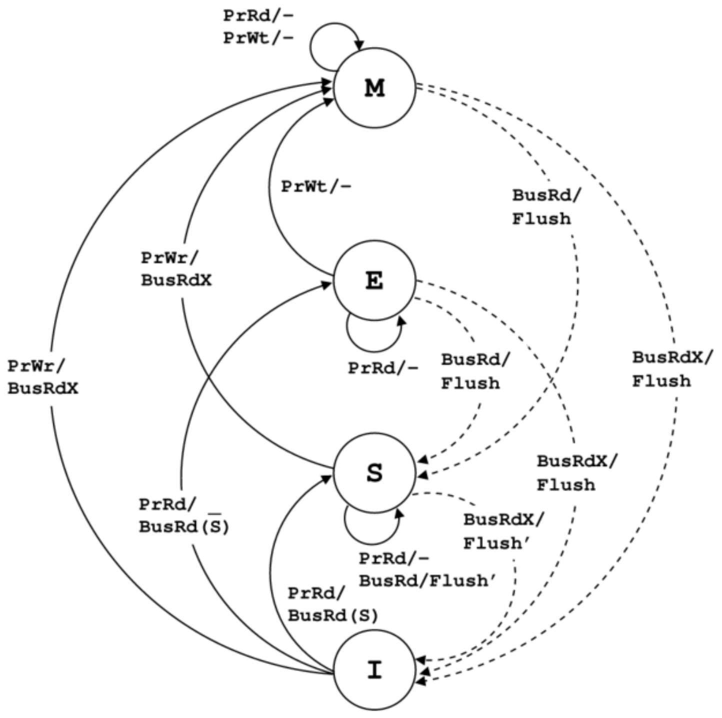
MSI

Write-back invalidation protocol.



MESI

Add an Exclusive state, where we're the only cache containing the block **and we've not modified it**. As soon as it's modified, we switch into the modified state, but we can read in the Exclusive state without any transitions.
More effective than MSI in multi-processor environments, as we can modify blocks owned exclusively by us without touching the bus.



Directory-based Cache Coherence

Snoopy buses don't scale well, as transactions are broadcast to all caches.

Directory-based protocols use a 'directory' responsible for arbitrating access to each block of the cache. Processors send requests for access to the directory. The directory keeps track of the owner and sharers.

Example (PPQ): L2 directory shadows write-through L1 tags (contains copies of each stored tag). Additionally store with each tag which caches have a copy. When a processor writes to L1 it writes through to L2 and tells all sharing caches to invalidate.

Cache Consistency

Consistency restricts the order in which memory operations from one core are seen by others: the order in which loads/stores become visible.

Sequential Consistency

Each core must preserve the program order of loads/stores.

Implemented speculatively, as full preservation is inefficient. Memory operations to different locations make no semantic difference to sequential consistency, so can be reordered.

Very easy to break: write buffers allow loads to overtake stores, and non-blocking read operations from out-of-order or speculative execution.

Total Store Order

Allow **loads to overtake stores**: doesn't make much semantic difference for most programs but allows for write buffers, so faster than sequential consistency.

Programmers can use memory fence instructions if they need to enforce loads before stores.

Interconnects

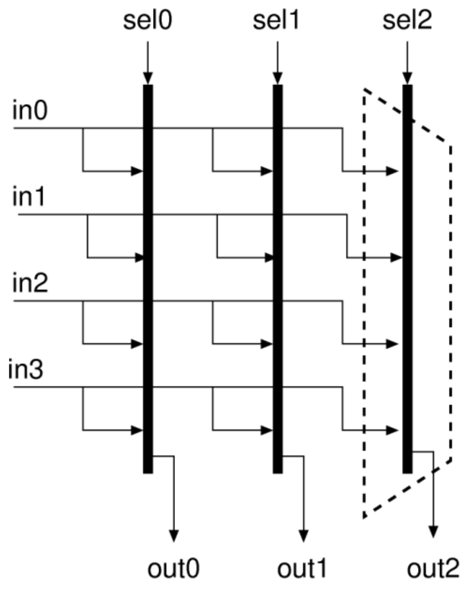
Some interconnects are bidirectional (bus, ring, by virtue of being broadcast). Others are unidirectional, so bidirectional communication requires twice the wiring (crossbar).

Bus Network

Broadcast, all nodes can read. Centralised arbiter provides access to bus.

Ring Network

Point-to-point interconnects to form a ring between nodes.
Supports concurrent data transfer, can still broadcast but nodes need to help forward.
Centralised or distributed control, but may need deadlock prevention with distributed.
Routers at each node are trivial.



Crossbar Network

Can directly connect any input to any output, without intermediate stages and without rearranging existing connections.

Very expensive, as it's quadratic cost with number of inputs/outputs.

General

Modules: processor cores, caches, etc. Connect to the network at terminals.

Nodes: Switches/routers, **not necessarily cores/caches**. Linked by **channels**.

Direct Networks: Each node is also a terminal (all caches etc. are routers as well).

Indirect Networks: Nodes are either terminals or switches (not both).

Diameter: The greatest minimal hop-count between all pairs of **terminals** in the network.

Packet: Higher-level message between network clients. Comprised of **flits** (flow-control digits), the smallest unit of information recognised by the flow-control method.

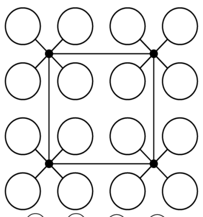
Bisection Width: Minimum number of wires that must be cut to divide the network into two equal-sized sets of nodes. The bandwidth over the bisection is the **bisection bandwidth**.

Mesh Topology

Each node has a channel to the nearest neighbour. Easy to map to the physical die.

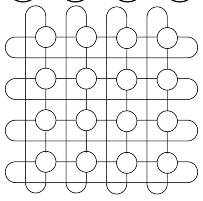
Potentially high hop count (diameter is $2(n-1)$).

Concentrated Mesh Topology



Increase mesh scalability by clustering terminals around each node.
Average hop count decreases, so latency drops. Lower bisection bandwith.

Torus



Doubles the bisection bandwidth of a mesh, and reduces the load of the central channels.

Hierarchical Networks

Eg. use low-latency crossbar for local communication between a processor and its private caches where scalability is fixed, then a global mesh network for connected processors.

Flow Control

Circuit Switching

Reservation of a circuit, without overhead on each message. Channels often shared by multiple circuits using TDMA or similar.

If lots of short packets need to be sent to different destinations, the setup/teardown overhead is significant.

Packet Switching

Different approaches to packet buffering on the on-chip switches and routers. Remember that the data being sent is flits, partial packets.

- **Store-and-Forward**: Buffer the entire packet then retransmit.
- **Cut-Through**: Wait for there to be enough space for the full packet in the destination buffer, then transmit what we get as soon as we have it (don't buffer it all before sending on).
- **Wormhole**: Send flits as soon as they're received. Like Cut-Through except we only need a single flit of space rather than a whole packet.

Routing

Usually simple deterministic routing protocols: channels don't go down, can be pre-programmed (although it then fails to deal with high load).

- **XY Routing**: Forward along then down in a mesh. Can avoid deadlock by restricting turn directions.
- **Dynamic Routing**: Gather network information and change routes.