Instruction Set Architecture

CISC: good code density, assembly programming common, typically microcoded. Microcode complicates exception handling and is an unnecessary overhead. RISC: exposes pipeline to the compiler, makes the common case fast and assumes the use of HLL.

Specifies the functional behaviour of an instruction set processor. Register type: accumulator, stack, general purpose. Instruction types: three address, two address, one address, zero address (stack based). Operands: memory-memory (may have cache registers), register-memory or register-register (load/store). Calling conventions for arguments, stack usage etc. Byte ordering. Alignment. Condition registers or flags (may impact code scheduling). Encoding (may impact code density, consume memory and power). Static/dynamic interface (try and shift functionality into the myopic compiler).

Pipelining

$speedup = \frac{time\,with\,enhancement}{time\,without\,enhancement} = \frac{1}{(1-Fraction_{enhanced})+\frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$

Make the common case fast: all enhancements consume design and implementation resources and may slow the common case.

$\frac{Time}{Program} = \frac{Insructions}{Program}\frac{Cycles}{Instruction}\frac{Time}{Cycle}$

Temporal and spatial locality are applied to the instruction reference and data reference streams.

$Power_{dynamic} = \frac{1}{2}Capacitative\,Load \times Voltage^2 \times Frequency\,Switched$

$Power_{static} = Current_{static} \times Voltage$

Can use functional (ISA level), performance (micro-architectural level) and RTL (implementation specific details) simulation. Can be trace or execution driven. Need to have benchmarks that prevent benchmark engineering. Can summarize performance results with arithmetic mean, weighted arithmetic mean, geometric mean ($(\prod_{i=1}^{n} a_i)^{\frac{1}{n}}$) or harmonic mean ($\frac{n}{\sum_{i=1}^{n}\frac{1}{a_i}}$).

Structural hazards arise from resource conflicts, data hazards from inter-instruction dependencies and control hazards from instructions that change the program counter.

Data dependencies are either anti-dependencies (write after read), output dependence (write after write) or true (read after write).

We can prevent data hazards by adding hardware to avoid hazards (e.g. interlock or forwarding) or by scheduling code to prevent data dependence becoming a data hazard (e.g. branch delay slots).

We can reduce the penalty of control hazards by computing simple branch results earlier in the pipeline (e.g. in the register read stage) but ultimately will have to add branch prediction or add a branch delay slot (single slot filled 60-75% of the time).

Exceptions require us to be able to restart execution. If we can guarantee that instructions prior to the faulting one have executed and later ones have not begun to execute we support precise exceptions. This can be done by tagging the faulting instruction and handling it at the end of the pipeline (write-back stage). To resume we may need to save multiple PCs if we use delayed branching.

We could introduce multiple intermediate execution pipelines for each functional unit, but this introduces further pipeline hazards (may need to interlock on write-back and must ensure the multiple pipelines respect data dependencies) and complicates exception handling.

Pipelines are limited by the overheads of introducing additional registers, clock distribution cost, difficulty of logic balancing and increasing stalls due to pipeline hazards.

Branch Prediction

Static branch prediction: annulling branches allow the compiler to make predictions: if it is wrong then the delay slot instruction is nullified. When coupled with feedback this achieves a 90% prediction rate.

Dynamic branch prediction:

- Bimodal (1 bit) counter: predict last taken.
- 2 bit counter scheme less sensitive to single misprediction, 3 bits limit of usefulness: read prediction from MSB.
- Can have branch history table indexed into by m bits of the branch address, each entry being a counter: don't worry about conflict aliasing due to approximation.
- Local history predictors contain a per-address shift register storing the last outcomes at that address: this can index into an address-local or shared pattern history table which holds counters.
- Global correlating predictors choose a counter to access based on the behaviour of recent branches and m bits of the address.
- Tournament predictors incorporate a local history and global correlating predictor and choose between them on the basis of a per-address 2 bit counter.

Can have negative (interfering) and positive (reinforcing) aliasing in prediction tables.

Typically need a branch target buffer (CAM) to store target addresses by branch address because we need the target address by the end of the IF stage. To improve accuracy you can use a small hardware stack for return addresses to predict the targets of RET instructions, which will vary dynamically.

Super-scalar

Super-pipelined processors have M sub-stages per pipeline stage. Super-scalar processors have multiple simultaneous pipelines out to the super-scalar degree P.

Typical structure:

1. Fetch and decode in program order as usual.
2. Rename registers and deposit renamed instructions in the issue window.
3. When operands are available, instructions in the issue window are dispatched to functional units (wakeup select upon operands broadcast).
4. Executed instructions are retired in program order to allow precise exception handling.

To do highly concurrent instruction fetch, need to be able to do unaligned (wrt. cache line width) reads and potentially non-sequential addresses.

Trace caches cache dynamic instruction sequences (in partially decode form) along with branch results. Trace hits occur when fetch addresses match the tag and branch predictions match the branch flags.

Register renaming implemented via a list of free registers. A register map table records the current architectural (ISA) to physical register mapping. Multiple mappings may be stored to allow recovery upon exception. Renaming exposes more ILP by removing “false” dependencies.

Can do in-order issue with instructions issued from quartet, decoder only advancing once that quartet had been issued. Out of order execution more typical since costs essentially no cycle time.

Memory *aliasing*: two memory references involving the same location. Memory *disambiguation*: determining if two references will alias.

Memory operations can be executed out of order but stores must not be speculative and earlier instructions must have their exceptions handled. Memory carried dependencies must also be respected. The adopted consistency model may also limit the extent to which we can retire instructions out of order.

Loads and stores are put in separate queues in program order. Can do store-to-load forwarding if have sufficient aliasing information. Can do speculative loads where some store addresses earlier in program order are not known, when the address arrives the speculation is checked and an exception raised if needed. Load speculation can be prevented with a dynamic load wait table saying which loads have previously caused exceptions upon speculation. May need to scan the queue for load-load ordering violations depending on consistency model.

Two main techniques to allow speculative execution and exceptions: reorder buffers and unified register files. In reorder buffer results are generated out of order and are only committed to the register file when all earlier instructions have completed. If a problem is detected the buffer is cleared and instruction fetch restarts. However, we now need to forward information from the reorder buffer by using a scan upon read or register mapping table which can mention a buffer entry. With a unified register file, we store all registers in a single large file. A map is maintained recording the future mapping of architectural registers to physical ones (used by renaming), and a second in-order map is maintained as instructions commit that maintains a mapping corresponding to current the architectural register file. Upon an exception we copy the in order mapping to the future mapping: this still needs a reorder buffer but it is simpler. This can be extended to allow recovery from branch misprediction by storing the register mapping table every time we encounter a branch.

Super-scalar techniques are limited by complexity, poor scaling characteristics of the hardware structures and diminishing returns on ILP.

Software ILP

VLIW packs multiple independent operations into a single “very long” instruction. Execution is statically scheduled by the compiler.

Unfortunately, ILP within a basic block is limited. To mitigate this, you can use loop unrolling, which additionally eliminates some branch instructions and a number of loop count increment/decrements. Code size and register pressure may increase however. Software pipelining schedules instructions from different iterations of the loop to create a new loop body consisting of independent instructions that may execute in parallel. It symbolically unrolls the loop and prevents spurious pipeline fills/drains. However, this may lead to complex prologue and epilogue code being generated.

Can also do global scheduling to move instructions between basic blocks. Trace scheduling finds the most likely path through the basic blocks using static branch prediction and optimizes the instruction schedule within the trace. The compiler inserts bookkeeping code at the the trace boundaries (entries and exits) to compensate for code motion. Superblocks are traces without branches back in, and can be created by copying the original basic block: this removes the need for compensation code on entries but not exits.

Predicated instructions can eliminate some branches, and can be emitted via *if conversion*. However they complicate data forwarding and probably get annulled late in the pipeline.

Hardware can support memory reference speculation: do a speculative load early and check it is valid at the original load location, if not execute fixup code. Implemented by hardware table of speculated load addresses checked by all later stores.

Rotating register files aid software pipelining by reducing the need for unrolling. The rotating register base is incremented on every iteration of the loop.

Parallelism

Coarse grained multi-threading has only one threads instruction in the pipeline at once:

- Thread level parallelism: used for multiprogramming, within applications.
- Switch-on-event multi-threading loads in a new thread when the original blocks on e.g. cache miss, timeout. Ensure a small switch penalty by using a short pipeline and caching a few instructions from the next thread. Could provide pipeline registers per-thread to allow stalls without the need to flush.

Fine grained multi-threading has a new thread selected on every clock cycle:

- May remove the need to detect and resolve inter-instruction dependencies (e.g. data forwarding) and can hide latency (removing the need for data cache?)
- Can provide strong performance guarantees if scheduling policy is predictable (e.g. round robin)

Simultaneous multi-threading can utilise the functional units left idle by the execution of a single thread to execute those instructions from a second one. However, this increases cache pressure and adds protection issues (e.g. need to add a thread ID to TLB and cache).

Cache Memories

Unified caches allow storage of both instructions and data, but it is common for a pipelined processor to want to access instruction and data memory in the same cycle. Separate caches allow this and also tuning for the two different workloads.

Large cache blocks better exploit spatial locality but decrease the number of blocks we can store. Larger blocks will also increase our miss penalty (time taken to load the block).

Direct mapped cache: use part of an address to index directly into a table, compare entry against tag bits. Has a low access time but suffers from repeated evictions even when there are many free entries.

Set associative caches use part of an address to index a fully associative set that can hold a number of entries.

Fully associative caches make use of parallel CAM to locate addresses in any location. By making them highly associative instead power requirements can be reduced (only power on one bank at a time). However they have high area requirements.

Block replacement policies are needed for set and fully associative caches: typically an approximation to LRU such as FIFO or NLU, or even random.

Write policies: choice of write allocate or not (if data isn't in cache on write) and write through (allows cache coherence in SMP) or write back (allows write buffering).

Cache misses are compulsory (block is brought into the cache for the first time), capacity misses (working set exceeds cache size) or conflict misses (due to many banks mapping to the same set).

Caches are typically virtually indexed and physically tagged so you can do a TLB lookup in parallel with cache access and validate the tag is correct. This works as long as the page size is $>$ block size so the low bits that are used to index into the index are invariant. Pure virtually addressed caches must be flushed on context switch and may suffer from aliasing of physical addresses.

Write buffers hide the lower throughput of the underlying memory system, allow write coalescing/bursting and allow store-to-load forwarding. However it requires that read after write hazards are detected and serviced from the buffer or the buffer is flushed.

Multilevel inclusion has L1 data always being present in the L2 cache: this ensures that consistency between caches and I/O or other SMP caches can be determined by checking the L2 cache only. However, it uses valuable space and may be hard when block sizes differ. Multilevel exclusion is better for instruction caches because coherency is rarely a problem.

Can reduce miss penalty via reading critical word first or reading only part of a block, especially when blocks large (but this requires more tag bits).

Victim caches attempt to reduce conflict misses by storing a fully associative record of the last lines displaced from L1.

Non-blocking caches useful in super-scalar processors since they allow servicing hits or even misses during a miss.

Can design algorithms so that they work with caches:

- Ensure working set fits into the cache.
- Improve spatial locality by favouring sequential accesses to non-unit strides.
- Fuse loops that access the same data to improve temporal locality.
- Fiss loops that have large bodies to get better spatial locality within the loop.
- Merge or pad arrays to avoid references to different arrays mapping to the same cache block (easy with a direct mapped cache and three contiguous arrays of size which is a multiple of the block size).

- Cache blocking: organize computation on e.g. matrix so that you operate on just the subset of the data structure that will fit into the cache at any given time.

- Prefetch upcoming data in hardware or software (not too early or you will pollute the cache!): can be done by having cache misses fetch the next sequential block as well as the requested one, or detect other strided accesses to blocks and get the next one in the sequence.

Main Memory

SRAM: 6 transistors, retains data in standby mode with minimal power. Very low latency compared to DRAM. Standard process means it can share a die with processor.

DRAM: single transistor and capacitor, must be constantly refreshed. Read is destructive. Optimised for capacity and cost. Organized as a bank of pages ≈ 1024 bits wide. Large page size provides data burst capability: may match cache line size to page size. Banks are organized to hold different address ranges so that requests to different memory addresses may be serviced in parallel in some cases. Careful organization of memory can ensure that addresses do not map to the same bank of DRAM. Memory controllers will perform scheduling of memory access so that we don't have to perform all DRAM operations sequentially.

Have to deal with transient and persistent errors in small memory cells. This is done with redundant memory components and ECC/parity schemes.

Through-die vias enable die stacking: DRAM/SRAM layered on top of the actual CPU. Non-volatile memory has limited erase/write cycles but can be useful as a cache for slower persistent storage (i.e. disks).

Vector Processors

Provide ISAs that work on vectors. Vector registers hold fixed number of elements and are massively multi-ported to allow lots of parallelism.

Allow explicit exploitation of data-level parallelism while reducing complexity and energy per operation (fewer dynamic instructions, less switches, regular pattern of accesses).

Vectors are partitioned into multiple lanes, each containing their own functional units. Elements are interleaved across the lanes in order.

Initiation rate is how many elemental operations are completed per cycle for each vector operation. *Start up time* is the latency until the first result word from memory reaches the vector register: to maintain an initiation rate of 1 the memory system must be capable of providing/accepting data at this rate.

Complication of non-unit strides (e.g. in matrix multiplication) solved with load vector with stride instruction. Variable vector lengths dealt with VL register that controls length of vector operation, strip mined loops to cope with fixed size chunks of larger vectors. Use vector chaining/tailgating to begin reading/writing over results of previous vector instruction due to in-order element production and consumption. Can introduce mask vector to control where operations are applied. Sparse vectors typically supported via gather/scatter instructions.

SIMD supports data parallel operations on wide registers in commodity processors.

Stream processors rely on your computation being organized into streams and kernels. Allow exploitation of ILP, data and task parallelism simultaneously for applicable applications.

Chip Multiprocessors

Shared address space/message passing possibilities for communication in distributed memory multiprocessors.

Symmetric shared memory architectures suffer from cache coherency problem. Solved via local bus snooping for writes and updating/invalidating affected cache entries. Write-through caches allow this to be implemented most efficiently, but cause lots of bus traffic. To use with write-back caches processors must be prepared to service read requests they see on the bus from their write buffers.

Sequential consistency: the result of any execution is the same as if the operations of all the processors were executed in some sequential order and the operations of each processor obey the local program order. This is violated by write buffers, overlapped writes, caches etc: relaxed memory consistency models are used to permit this.

Niagara: single issue pipeline using fine grained parallelism and directory-based cache coherency (maintained by the L2 caches and lists sharers for a given cache block: writers notify sharers to invalidate their cache).

Special Purpose Architectures

GPUs are increasingly programmable and are switching to a unified architecture to cope with vertex and pixel heavy workloads.

Parallel supercomputers may come with FPGA as programmable coprocessor.

There is a trend to replace hardware with software to decrease time to market and increase flexibility. To do this within hard real time requirements, multiple cores with resource reservations can be used. Typically these are arranged in a tile configuration with statically scheduled network communications.

Place and route tools ensure communication locality. This configuration is particularly suited to stream processing applications.

Configurable processors allow designers to optimize CPUs for specific applications. Might introduce special instructions for operations that are hard or inefficient to express in software.