1 Pipelining

We can **pipeline** each stage of instr execution, allowing multiple instrs to be **in-flight** *simultaneously*. This increases *throughput*, not *latency*. Speedup reduced by unbalanced stages, and **fill** / **drain** time. **Hazards** prevent exec in designated clock cycle:

- **Structural**: hardware can't support combination of instrs in simultaneous exec. (*e.g. FETCH / MEMORY overlap*). To fix introduce **pipeline bubble** (gap) or **instr buffer**.
- **Data**: One instr uses anothers output. Causes *stalls*. Fix with **forwarding** by getting data directly from ALU.
- **Control**: branch pred failure. Reduce stall by guessing earlier in decode stage.

2 Caches

A **direct mapped** 2^N byte cache, with block size 2^M has 2^{N-M} **cache lines**, with each mem addr split into 3 fields:

- Uppermost $32 - N$ bits: **cache tag**. *Which mem block is stored in cache line?*
- Lowest M bits: **byte select**. *Which byte in cache line is accessed?*
- Middle $N - M$ bits: **index**. *Which cache line to look in?*

For C bytes, B blocksize and A addr size:

- $\log B$ bits for byte select.
- $C \div B$ blocks stored.
- $I = \log(C \div B)$ bits for cache index.
- $A - I - \log B$ bits for cache tag.

To access memory within the cache:

1. Use **index** to select cache line.
2. Compare **tag** with the tag in cache line.
3. If match, **cache hit**. Use **byte select** to get the data.
4. If no match, **cache miss**. *Fetch* block from main mem, *update* cache line, use **byte select** to get the data.

Bad design, addrs spaced out by cache size will *overlap* - **associativity conflict**. An **N-way set associative cache** has N direct mapped caches (**ways**) in parallel. To access mem:

1. Use **index** to select 1 line per way.
2. Each **tag** is compared in parallel.
3. If a tag matches, **cache hit** ⋯
4. If no tag matches, **cache miss** ⋯

However, N comparators → complex hardware → slower access time. Extra MUX causes additional delay. *In DM cache, data comes before hit/miss signal, so we can assume hit and cancel later.* Issues:

- **Block Placement**: *where to place block?* affects perf as determines how data is accessed.
- **Block Identification**: As assoc increases, bits for tag decreases.
- **Block Replacement**: *cache replacement policy*.
- **Write Strategy**: can *write through* (always write to lower level mem, not solving cache coherence) or *write back* (only write to lower mem when replaced, requiring dirty bit, and absorbing repeated writes).

Turing Tax is the overhead for the *universality* of a computer.

3 Dynamic Scheduling (Tomasulo)

Instr Parallelism allows instrs behind a stall to proceed allowing *out of order* exec. Order constrained by *data*, *name* & *control* dependencies.

- **Read←Write (RAW)**: true dep (*data*).
- **Write←Read (WAR)**: anti dep (*name*).
- **Write←Write (WAW)**: output dep (*name*).

In **Tomasulo**, regs have a **tag** indicating which unit will produce its value. If tag empty, reg is up to date. Each unit has a **reservation station** which waits for operands to be available then executes. When an instr is issued:

1. **ISSUE**: Idle RS selected. If operands are **valid** (reg has no tag), set them. Set dest regs tag to RS ID. *Overwrite if necessary.*
2. **EXECUTE**: If operand becomes avail on **control bus** with matching tag, set it. When all avail, exec the instr.
3. **WRITE RESULT**: When result avail, broadcast on control bus with RS ID.

The **databus** delivers operands, tags & instructions at *issue*. The **control bus** delivers results & tags at *write result*.

Effective parallelisation, building dataflow dependency graph & register renaming. However, many associative stores & high wiring density of CDB.

3.1 Reorder Buffer

Tomasulo doesnt handle *exceptions* and *branch mispredictions*. Add **COMMIT** stage and maintain **reorder buffer**:

- On *issue*, **allocate** entry in **ROB** with instrs' *dst reg*, *value* or *tag*. ROB is a queue, maintaining program order.
- On *writeback*, **update ROB** entry.
- On *commit*, **process ROB** *in-order*. **Committed registers** (*duplicates*) are updated. On interrupt *flush ROB*, *reset issue-side regs* and *restart fetching* from correct addr.

Can only consider misprediction until instr reaches ROB head, and can only process 1 instr per cycle.

Register Update Unit (RUU) combines instr tracking & reg updates in one structure. RUUs integrate more functions, while ROBs simplify commit handling. In ROB, regs and ROB entries have a tag, so every reg, ROB entry & RS needs a comparator to monitor the DB. In RUU, the tags are the ROB entry numbers, so the ROB is indexed by the tag on the DB.

3.2 Speculative Memory

Store to Load Forwarding can forward uncommitted stores to load from the same addr. But by using *computed addrs*, we must speculate. On misprediction we flush & restart.

4 Branch Prediction

A **branch predictor** avoids *control hazards* without stalls. A **branch history table** uses *PC (lower bits)* to index table of n-bit vals whether the last branch was taken.

In a 1-bit pred, **aliasing** possible & 2 misses in a loop.

In a n-bit pred, not-taken decrs, taken incrs. $> 2^{n-1}$ predicts taken, and vice versa.

A **G-Selector** uses global branch history reg (BHR), an n-bit record of the last n branch outcomes, to index a BHT. These learn which global history patterns correlate with a branch being taken.

4.1 Target Prediction

A **branch target buffer** maps *PC*s to their predicted next *PC*, indexed by PC (lower bits), tagged with PC (higher bits), avoiding **aliasing**. Each entry has *branch PC tag*, *predicted PC* & *extra state bits*. Accessed in parallel with the instr cache in the *fetch* stage. Updated on branch **commit**. At decode, check if pred was correct; if not override PC for fetch stage and **squash** (disable) mem and writeback stages of mispredicted instr.

To store **return addrs**, we can either use a **stack** or **special reg**.

5 Cache & Memory

Average Memory Access Time (AMAT):

$$\text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

In *multilevel cache*, miss penalty is AMAT of the next level cache cache.

5.1 Miss Rate

Cache misses can be **compulsory** (*initial miss*), **capacity**, **conflict** & **coherence**. Increasing **block size** would decrease miss rate, but too large will have data load overhead.

In **victim caching**, has a small fully assoc cache between main cache and mem. This stores evictions from main cache, significantly reducing *conflict* misses.

In a **skewed assoc cache**, each way has a different hash function, reducing conflict misses, reducing assoc, more predictable perf, but overhead in implementing LRU.

A **stream buffer** can **prefetch** by fetching missed blocks along with several next blocks in mem into a small FIFO buf. SB is checked in parallel with cache. A **multi-way SB** has multiple SBs with different datastreams.

5.2 Miss Penalty

We can either:

- **Write Through**: all writes propagate down cache levels (**multilevel inclusion**). Each block has **valid bit**. Can always evict any block. Faster to evict, slower writes, incoherency.
- **Write Back**: writes to mem on evict. **Dirty bit** & valid bit used. Faster, lower bandwidth, better tolerance to slow memory, but more complex.

We can either:

- **Write Allocate**: on miss, load into cache, then write. Better for temporal locality.
- **Write Around**: on miss, write directly to mem. Better for poor temporal locality.

A **write buffer** between cache and mem can reduce miss penalty, at the cost of having to be read in parallel with the mem. Store 2 – 8 lines, and can **coalesce** writes to the same block.

In **early restart**, on miss, begin fetching missed block from mem. When the tag of the requested word is returned, resume execution. Remaining data is fetched simultaneously. This reduces miss penalty, not having to wait for the entire block to be fetched before resuming execution. In **critical word first**, fetch requested word first, then the rest of the block.

However, we can get a *race condition* if another load to the same block occurs before the rest of the block has been fetched. To fix this, divide cache line into sectors each with their own **valid bit**. We allocate in lines and deliver data in sectors.

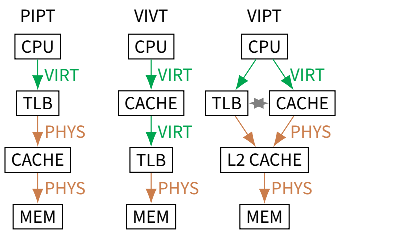
A **non-blocking** cache supplies data while a miss is serviced. *Hit under miss* lowers penalty by working during miss service. *Miss under miss* lowers penalty by allowing overlapping miss servicing. This significantly increases complexity and requires multiple mem banks.

5.3 Hit Time

Splitting pipeline stages involves cache accesses, allows more regular cache access, but with a higher latency. To support this:

- **Banking**: divide cache into independently accessible banks, mapped by lower order bits or hash funcs.
- **Duplicate** the cache.
- **Multiport RAM**: support simultaneous accesses, with two wordlines per row and two bitlines per column. Expensive and complex.

5.4 Address Translation



- **Phys Index, Phys Tag (PIPT)**: requires **TLB** lookup before cache access, adding latency.
- **Virt Index, Virt Tag (VIVT)** allows faster cache access, reducing hit time. Leads to *synonyms* & needs complex coherence handling.
- **Virt Index, Phys Tag (VIPT)**: data and tag accessed in parallel with **TLB** lookup, reducing hit time & avoiding synonyms. *Requires careful design to ensure cache size and page size compatible.*

Virt addr space divided into **pages** & mem divided into **page frames**, allowing non-contiguous malloc, reducing fragmentation & allowing easier swapping of pages.

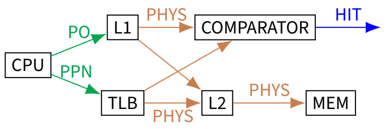
A virt addr is divided into a **virt page number (VPN)** (indexes **page table**, maps into *PPN*) & **page offset** (byte within page).

The **translation lookaside buffer** is a small cache of the PT, allowing for faster addr trans, reducing PT overhead. Often a highly associative cache.

Homonym: virt addr ↦ many phys addr in multiple procs. Virt indexed cache leads to many copies of the same data in cache, causing coherence issues. Avoid by flushing cache on context switches, or include PID in cache tag.

Synonym: many virt addr ↦ phys addr. In a virt indexed cache, virt addr is cached twice under diff phys addr, causing coherence issues. Avoid by forcing synonyms to have same index bits in the cache (**page colouring**: multiple free lists of phys pages).

In VIPT, if index has only phys addrs, we can access tag in parallel with trans. This limits cache size to page size × assoc:



5.5 DRAM

Stores data in square array of **cells**:

1. Row addr selects row to access via **wordline**.
2. Cells **discharge**.
3. Cell **state** latched by per-col **sense amps**.
4. Col addr selects data for output via **bitline**.
5. Data written back to selected row.

Since row is latched, subsequent accesses to the same row are faster (page mode). To access a diff row, first precharge the bitlines, then activate the new row. This means row access cycle time is larger than the row access time, as the data needs to be written back after it is read. After a while the charge leaks, so every cell must be refreshed every ≈ 64ms. This is managed by a microcontroller in the DRAM module. DRAM is unavailable during refresh, so we should aim to reduce it.

Error correction adds redundancy via **parity**/**ECC** bits, using **hamming codes**. **Memory upsets** can be caused by updating surrounding cells with **row hammering**, something fixed by *error correction* or *adaptive refreshing*.

We could achieve **mem parallelism** with parallel addressing & parallel data transfer by having many mem **banks**. Each bank can be accessed independently, allowing many operations to occur simultaneously, increasing bandwidth & reducing latency.

6 Side Channel Vulnerabilities

Side Channel Vulterabilities expose another thread's state/data by observing its effect on the system state. **Exfiltration**:

- **Prime & Probe**: attacker *primes* L2 cache by filling ≥ 1 sets with data. After victim executed, attacker probes the state of L2 by timing accesses, seeing which sets were evicted.
- **Evict & Time**: attacker runs victim, establishing base exec time. It then evicts a line of interest and reruns victim. Variations in exec time indicate which cache lines were accessed.
- **Flush & Reload**: relies on *shared virt mem*. Attacker flushes a line of interest. After victim execution, attacker reloads line by touching it, measuring time taken. A fast reload indicates victim touched this line, reloading it.

For exploitation, there must be a shared state affected by the execution of both attacker and victim:

- *Single core*: cache level, TLB, branch predictor, prefetchers, physical rename regs, dispatch ports.
- *Single NUMA domain*: mem controller.
- Cores may share caches, interconnect, etc.

To execute the victim code, the attacker must either *(1)* perform a **syscall**, *(2)* release a **lock**, *(3)* run simultaneously, *(4)* call as a **function** (testing language security).

Historically, to limit context switch cost, OS stored copies of its page addr trans alongside each proc, avoiding TLB flushes. However, this allowed **spectre** attacks to access kernel data.

Kernel Addr Space Layout Randomisation (KASLR) randomises placement of code and data, making spectre attacks guess where the data it wants is stored. This is not foolproof.

Kernel Addr Space Isolation (KASI) flushes TLB each time kernel is entered, mitigating spectre attacks despite perf impact. Overcome by **Spectre 2**.

7 Multithreading

- **Regular** issue has reduced utilisation due to stalls and dependencies.
- **Superscalar** issue: more perf but less util.
- **Chip MultiProcessing (CMP)** adds more cores, increasing util but not 1-thread perf.
- **Fine Grained Multithreading (FGMT)** rotates between threads every cycle to reduce stalls, but intra-thread dependencies still cause stalls.
- **Simultaneous MultiThreading (SMT)** allows instrs from many threads to be issued in the same cycle, increasing utilisation.

In SMT, n PCs and n reg decode maps:

- *Resource sharing*, individual threads slower.
- Thread contention (*cache/TLB thrashing*, ⋯).
- Need to manage *private* resources.
- SMT must *schedule* fairly.
- Side channel threat.

SMT threads exploit mem-sys parallelism by allowing many threads to issue mem ops simultaneously (**latency hiding**). Increasing SMT threads has overhead due to handling more regs (*reg renaming*), more complex scheduling, and increased contention.

8 Vectors & SIMD

Arithmetic Intensity is the ratio of FLOP to bytes of mem accessed. Sparse kernels have low AI. The *roofline model*:

- **Memory-Bound Region**: perf limited by mem bandwidth (*low arithmetic intensity*).
- **Compute-Bound Region**: perf limited by CPU's peak computational throughput.

Vec Instr Sets contain vec regs allowing exec in parallel **lanes**, and pred regs for masking ops. To ensure compiler vectorises: *(1)* make iterations a multiple of vecreg lengths, *(2)* ensure not overlapping pointers, *(3)* ensure no loop carried dependencies, *(4)* use SIMD intrinsics directly.

Single Instr Multiple Threads (SIMT) is a parallel arch allowing many threads to exec the same instr simultaneously on different data.

If loop body has **indirection** (*e.g.* `B[ind[i]]`), then we need a **vec gather** instr, less efficient than load as data may not be continguous. **Predication** & **masking** allow `if`s in the loop.

- **Vector Pipelining** execs vec instrs **serially** on a *static* pipeline. Words are forwarded to next functional unit (FU) as soon theyre ready, forming a pipelined chain.
- **UOB Decomposition** breaks down complex vec instrs into many micro-ops that can be execd in parallel across many exec units. Can be used when the provided vec length exceeds the hardware vec length. Each n-wide vec is split into $\frac{n}{m}$ m-wide vecs, where m is the hardware vec length. *They are committed together.*

9 GPUs

GPUs have *many cores*, with many *FU*s implementing **SIMD** model. *Less cache* per core, *fast context switches* and *no branch pred*. Contain many processor devices (**streaming multiprocessors, SM**), using FGMT to run many **warps** per SM.

- **GPU warp** = *CPU thread*
- **GPU thread** = *CPU lane*

Inside each *SM* there is a:

- **Multithread Issuer (MT)**: selects warp to issue in each cycle (*FGMT*).
- Explictly programmed **scratchpad mem**, warps on the same *SM* share this.
- **L1 Cache** with no coherency protocol.

Each chip has many DRAM channels, each of which has its own L2 Cache (*so no cache coherency protocol is needed between SMs*).

CUDA is a C ext for programming serial CPU code and parallel GPU **kernels**.

- Thread group = **thread block** (*1D, 2D, 3D*).
- Thread blocks = **thread grid** (*1D, 2D, 3D*).
- Threads in the same block share mem.

If threads in a warp **diverge** then each branch path is execd serially, disabling threads not on that path (*predication*). After this, reconverge.

9.1 SIMD vs SIMT

SIMD	**SIMT**
1 thread per lane	SIMD per thread
Adjacent threads access adjacent data for spatial locality.	Adjacent loop iters access adjacent data for spatial locality.
Load instr can result in a diff addr accessed by each lane.	SIMD vector load has access to adjacent locations.
Coalesced loads with adjacent accesses are very fast.	Gather instrs can fetch diff addrs per lane, but often serial.
Branch *coherence*	Branch *predictability*

10 Multicore & Cache Coherency

Power is the critical constraint of perf:

- **Dynamic Leakage** when signals change.
- **Static Leakage** when gates are powered on.

Dennard Scaling states as transistors get smaller, dynamic power gets smaller. Now, static leakage dominates power. *More efficient to have many parallel units at low clock and low voltage!*

To reduce power, *(1)* turn of units, FUs, or cores when unused, *(2)* dynamic voltage and clk regulation, *(3)* many cores lower clk, *(4)* turbo mode.

10.1 Programming Models

OpenMP: shared-mem model, with compiler pragmas specifying parallel regions: Use `#pragma omp parallel for`.

MPI: API for parallel programming using message passing. Has following funcs:

- `MPI_Init`: init MPI env.
- `MPI_Comm_size`: Get number of processes.
- `MPI_Comm_rank`: Get rank of current process.
- `MPI_Send`: Send message to another process.
- `MPI_Recv`: Recv message from another proc.
- `MPI_Finalize`: Terminate MPI environment.
- `MPI_Bcast`: Broadcast msg to all procs.
- `MPI_Reduce`: Reduce values from all procs.
- `MPI_AllReduce`: Reduce values and distribute result to all processes.

OpenMP is *easy* but leads to *unintended bugs*. MPI is *explicit*, but more *complex*.

10.2 Snooping Cache Coherency

Cache incoherency: multiple cores have local caches, one core updates a memloc, other cores have stale copies in their caches. *We need to know where to find the most recent data, and when data is stale.*

The goal is SC. Idea: invalidate other caches when a store occurs, forcing other cores to suffer a read miss and fetch the updated data from mem. Here, a snooping **cache controller** sits between a core's cache and the bus, monitoring all bus transactions and checking them against the tags of its cache.

10.3 Berkley Protocol

Each cache line can be *(1)* **invalid**, *(2)* **valid** - clean, potentially shared, unowned, *(3)* **shared dirty** - modified, shared, owned, *(4)* **dirty** modified, unshared, owned. On a **read miss**:

1. Broadcast request on the bus.
2. If another line is **Dirty**/**Shared Dirty**, supply data and sets its state to **Shared Dirty**, and ours to **Valid**.
3. Otherwise, fetch data from mem, set our cache line to **Valid**.

On a **write hit**, if **Valid**/**Shared Dirty**, set invalidation is sent & local state set to **Dirty**. On a **write miss**, *same as read miss* then all other caches **Invalidate** their copies, and our cache line becomes **Dirty**.

Since every bus transaction checks cache tags, there could be contention between bus and CPU accesses. To avoid this:

- **Duplicate** set of tags for *L1* to allow checks in parallel with CPU accesses.
- Use *L2* to **filter** invalidations. *Only works with multi-level inclusion*. Many systems force *cache inclusivity*, making this not an option.

10.4 Memory Models

For **atomics**, we split two mem access in one instructions into two:

- **Load Linked**: load value from mem addr.
- **Store Conditional**: store to mem only if no updates have occured since last LL.

With this we can build a lock:

```
try:
 LI    R2, #1     ; Load 1 into R2
lock:
 LW    R3, 0(R1) ; Load lock value
 BNEZ R3, lock   ; If lock != 0, repeat
 EXCH R2, 0(R1) ; Exch R2 with mem at R1
 BNEZ R2, try    ; If prev != 0, repeat
```

10.5 Interconnets

Snooping cache coherency protocols rely on a bus, a bottleneck for high core counts. To scale, we distribute DRAM around the system using **Non Uniform Memory Architecture (NUMA)**.

With an **interconnect network**, each node has its own mem. Each node has a **directory** that tracks the state of every block in every cache. The directory could track information:

- *per mem block*: simple but more traffic.
- *per cache block*: complex but less traffic.

Directory allows finding most recent copy of data (*by using a linked list structure*). This could be a major bottleneck.

- **ccNUMA**: each node has a fragment of DRAM, every phys addr has unique home node.
- **COMA**: each node has a fragment of DRAM, data can migrate between nodes adaptively.
- **NUCA**: cache is distributed, so access latency is non-uniform.