1 Cache

Memory technology	Access time	cost/GB
Static RAM (SRAM)	0.5-2.5ns	2K-5K$
Dynamic RAM (DRAM)	50-70ns	20-75$
Magnetic disk	5-20ms	0.2-2$

	SRAM	DRAM
Used for	Cache	Main memory
Density	Low (6[1])	High (1)
Power	Higher	Lower
Content	Static (last forever)	Dynamic (refreshed regularly)

SRAM vs. DRAM. [1]Number of transistor cells.

1.1 The memory bottleneck

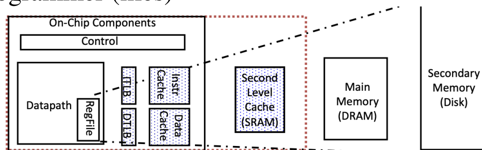
- Typical CPU clock rate is 2GHz (=0.5ns cycle time).
- Typical DRAM access time is 30ns ≈ 60 cycles.
- Typical main memory access is 100ns (200 cycles): DRAM (60), precharge (20), chip crossings (60, overhead (60).
- Average instruction references are 1 instruction word, 0.3 data word.

Memory delay is mostly communication time. Read/write a bit is fast. It takes time to select right bit and route the data to/from bit. This problem gets worse (processor-memory performance gap). CPUs get faster, memories get bigger.

1.2 Memory hierarchy

Large memories are slow and fast memories are small. 💡 Take advantage of the **principle of locality** to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology.

- Registers ↔ memory: by compiler
- Cache ↔ main memory: by cache controller hardware
- Main memory ↔ disks: by OS (VM), V-to-P mapping (TLB), and programmer (files)



1.2.1 Locality

- **Temporal**: tend to be referenced again soon → keep most recently accessed data items closer to the processor
- **Spatial**: nearby addresses tend to be referenced soon → move **contiguous words** closer to the processor
- Locality depends on type of program behavior.

1.2.2 Levels

Block (or line) is unit of copying, maybe multiple words.

- **Hit**: accessed data is present in upper level
  - Hit ratio: hits/accesses
- **Miss**: if absent, block copied from lower level
  - Miss ratio: misses/accesses = 1 – hit ratio
  - Miss penalty: time to replace a block in that level
- Hit/miss ratio is per level.

✔ Average Memory Access Time (AMAT) = t_{Hit} + miss rate × t_{Miss} (t is time i.e. latency)

💡 **No × (1-miss rate) to hit!** Miss cost = hit cost + penalty

1.3 Direct mapped cache

Each memory block has designated block in cache.

- Address mapping: (block address) modulo (no. blocks in cache)
- Tag at each cache block that contains the upper portion of the address to identify the block

1.3.1 Example

One word block (4B), Cache size of 1K=1024 words (or 4KB). Bits for tag? 💡 32 - 2 (byte offset) - 10 (index) = 20 bits.

- Read index [11-2] → ([31-12] = tag) & valid → Hit, get data
- Exploits temporal locality

✔ Address bits = Byte offset (∵ word size) + Cache index + Tag

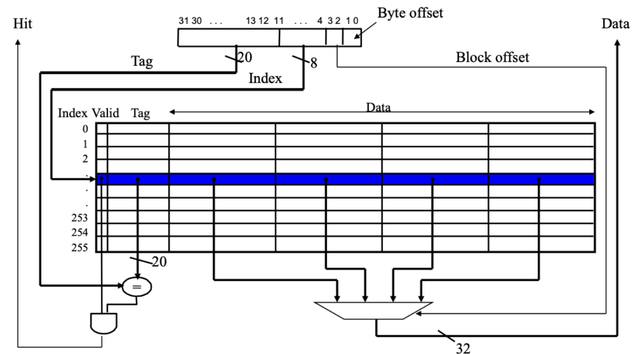
1.3.2 Multiword: Adding spatial locality

✔ Address bits = Byte offset + Cache index + Tag + Block offset

✔ Cache size = $2^{\text{Cache index}} \times$ (Data blocks size + Tag + Valid field)

? Larger blocks...
  - should reduce miss rate ∵ spatial locality. But in fixed-sized cache, it means less index → competition ↑ → miss rate ↑
  - lead to pollution, i.e. cache data not used
  - miss penalty ↑ → may override benefit of reduced miss rate.

➔ Sweet spot in block size (if cache size fixed)



1.4 Handling cache hit/misses

1.4.1 Load

Cache hit: CPU proceeds normally

Cache miss: Stall the pipeline, fetch block from next level of hierarchy, and resume pipeline: (i) I$: restart instruction fetch, (ii) D$: complete data access.

1.4.2 Store

Cache hit: keep consistent with main memory

- **Write-through**: always update memory → Slow, CPU is stalled
- **Write-back**: remember that block is modified (dirty bit), update memory when dirty block is replaced.
  - 💡 Consider for loop. Only write memory once.
  - Sometimes need to flush cache: I/O, DMA, multiprocessing
- In both cases, may use **write buffer** that write to memory on background. CPU stall only when buffer is full.

Cache miss: naive way is to stall pipeline, fetch block, install in cache, update cache, and resume pipeline. But why fetch if we're updating it?

- **Write-allocate**: Update cache, memory update depends on write-through/back. Anticipate further use of block (temporal locality)
- No-write allocate: Update memory, cache unmodified

1.4.3 Multiword block considerations

Load miss: miss penalty grows as block size grows. Two solutions:

- Early restart: processor resumes execution as soon as the requested word is returned
- Requested word first: complicate hardware (overhead!)

\+ Nonblocking cache: allow processor to continue to access cache while cache is handling an earlier miss → need dependency check, much complicated hardware

Write miss: If write-allocate, must first fetch the block from main memory and then write the word. Or could end up with **garbled** block in cache (rest of the block is old data).

1.4.4 Sources of cache misses

1. **Compulsory**: first reference to block
   → Increase block size (spatial locality)
2. **Capacity**: cache cannot contain all blocks accessed by program
   → Increase cache size
3. **Conflict** (collision): multiple memory locations mapped to the same cache location
   → Increase cache size / Increase associativity

1.5 Measuring cache performance

Assuming cache hit costs are included as part of the normal CPU execution cycle, then

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{CC}$$
$$= \text{IC} \times \underbrace{(\text{CPI}_{\text{ideal}} + \text{Memory-stall cycles})}_{\text{CPI}_{\text{stall}}} \times \text{CC}$$

Memory-stall cycles come from cache misses. Miss penalty is measured in processor clock cycles needed to handle a miss.

- ✔ Read-stall cycles = read/program × read miss rate × read miss penalty
- Same for write; can reduce write-stall cycles with write buffer
- For write-through caches, we can simplify this to:
- ✔ accesses/program × miss rate × miss penalty

Relative cache penalty increases as processor performance improves (faster clock rate and/or lower CPI).

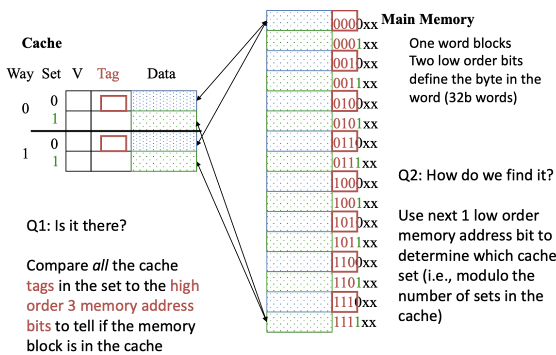
1.6 Reducing cache miss rate

1.6.1 Set-associative cache: Allow flexible block placement

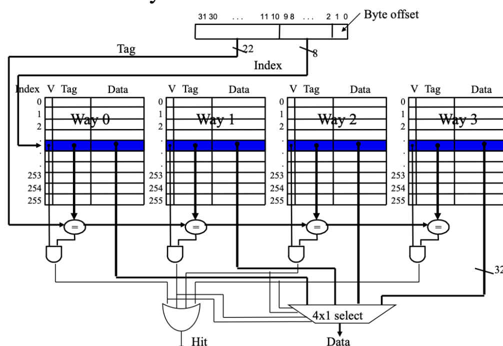
Direct mapped cache is subject to **conflict** by only allowing a memory block to map to exactly one cache block. ⟷ **Fully associative** cache allow a memory block to be mapped to *any* cache block

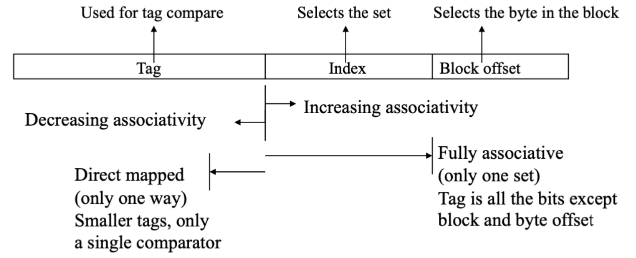
⚠ No index anymore → Cost of tag comparison
∴ Only used for very small cache (8, 16 entries)

A compromise is a ***n*-way set associative cache** where a memory block maps to a unique set of size *n*.



- A cache with 8 entries can be one-way i.e. direct mapped, 2-way i.e. 4 sets, 4-way i.e. 2 sets, or 8-way i.e. fully set associative. All same size; except slight difference ∵ metadata.
- Increased associativity reduces miss rate but with diminishing gains.





Range of set associative caches

- ✔ Cache entries = Number of sets × Set size (index)
- So if fixed-size cache, 2× associativity - index bit and ++tag bit.

We need **replacement policy** now. Least Recently Used (**LRU**) replaces the block that has been unused for the longest time. For 2-way, need one bit per set, and set the bit when a block is referenced. But for *n*-way, we'll need more bits. **Random** replacement policy gives ≈ the same performance as LRU for high associativity.

N-way cache costs:

- *N* comparators (delay and area)
- Data available after set select & hit decision: MUX delay, parallel ⊘
- ↔ direct mapped can just assume a hit and recover later if miss

1.6.2 Multi-level caches

With advancing tech, enough room on the die (chip) for bigger L1 caches or for a second level of caches–normally a unified L2 cache (i.e. holds both instructions and data) or even a unified L3 cache.

$$AMAT = hit_latency_{L1} + miss_rate_{L1} \times (hit_latency_{L2} + miss_rate_{L2} \times miss_latency_{L2})$$

Two-level cache AMAT

Design considerations for L1 and L2 caches are very different

- Primary cache should focus on **minimizing hit time** to support shorter CC → smaller with smaller block sizes
- Secondary cache should focus on **reducing miss rate** to reduce penalty of main memory access → larger with larger blocks size, higher associativity

1.7 Further ideas

- Victim cache: small buffer holding most recently discarded blocks
- Check write buffer and/or victim cache on read miss–may get lucky!

2 Virtual memory

Use main memory as a cache for secondary memory.

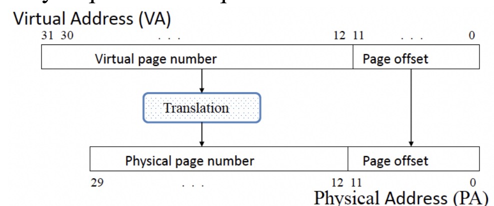
- Efficient and safe sharing of memory among multiple programs
- Easily run programs larger than the size of physical memory
- 💡 Again the principle of locality!

Each program is compiled into its own *virtual* address space.

- Address space is divided to pages (fixed size) or segments (variable)
- The starting location of each page (either main or secondary memory) is contained in the program's **page table** (at DRAM!)
- **Page fault**: VM miss i.e. page is not in physical memory

2.1 Address translation

Each memory request first requires an address translation.



Page tables can get awfully large. For 32-bit address space with 4KB pages, which means 12 bits for offset, we will need 2^{20} entries. For 4 bytes PTE, it is 4MB per process. → Multi-level page table!

Disk access takes millions of cycles, so minimize page fault rate: fully associative placement, smart replacement, etc.

Page table stores placement information.

- If page is in memory: PTE stores PPN and status bits (valid, dirty)
- If not i.e. valid=0: PTE can refer to location in swap space on disk

Page fault is handled by OS code.

1. Use faulting VA to find PTE
2. Locate page on disk
3. Choose page to replace. If dirty (💡 write-through is impractical), write to disk first.
4. Read page into memory and update page table
5. Restart process from faulting instruction

Approximate LRU based on heuristic that PTEs are accessed frequently.

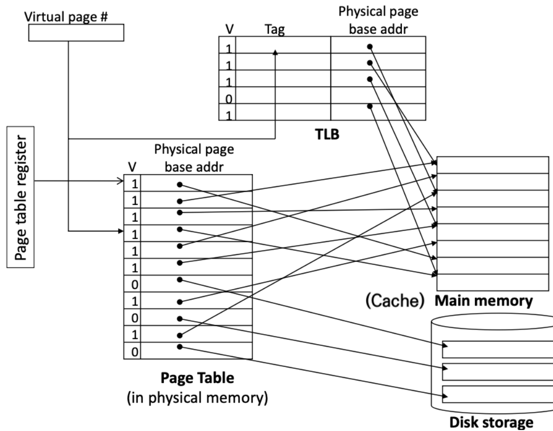
1. Reference bit (or use bit) in PTE set to 1 on access to page
2. Periodically clear use bit to 0 by OS
3. Evict a page with reference bit = 0

2.2 Translation lookaside buffer

Translation of VA to PA takes an extra memory access → expensive! Use a TLB–a small cache on-chip that keeps track of recently used address mappings.

- Usually fully associative
- 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss
- 0.01%–1% miss rate ∵ access to page tables has good locality
- TLB misses have 2 cases: (i) miss could be handled by HW or SW, (ii) true page fault (infrequent) handled by OS

Must recognize TLB miss before destination register overwritten.



It is beneficial to have separate TLB for instructions and data ∵ different locality. TLB can be multiple levels.

2.2.1 Can overlap the cache access with the TLB access

$$\begin{aligned} \text{PA} &= \text{PPN} + \text{Page offset} \\ &= \text{PA tag} + \text{Cache index} + \text{Block offset} + \text{Byte offset} \end{aligned}$$

If bits for page offset ≥ bits for cache index + block offset + byte offset, we have all information we need to access cache before TLB. ∴ can **parallelize** TLB access with cache access, and then compare PA tag and tag of cache entries. **Then number of sets limited by page size**. To increase cache size, must increase associativity → can make cache slower.

2.3 Hardware support

- Detection of page fault
- Dirty and reference bits update

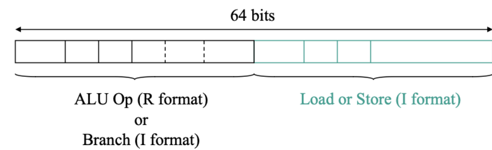
3 Instruction-level parallelism

- **Super-pipelining**: more stages!
  - 👍 Shorter critical path, reduce CC
  - 👎 More hazard, more forwarding/hazard hardware
    - Higher flush cycles for branch misprediction
    - Higher L1 load-to-use latency (in cycle) → bubbles ↑ → CPI ↑
- **Multiple-issue**: more than one instructions at every stage
  - Now CPI < 1, so use IPC (instructions per clock cycle) metric
  - Speculation: guess what to do with an instruction so we start operation ASAP.
  - 👎 Data/control dependencies are more likely / Resource conflict

3.1 Static multiple-issue aka very long *in* word (VLIW)

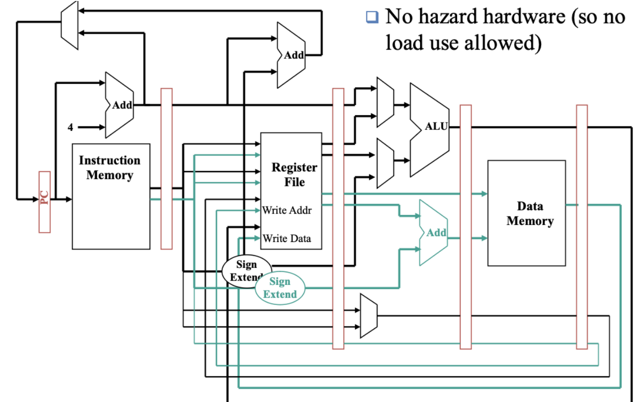
- **Compiler-driven**: decide *ins* to execute simultaneously at compile time (i.e. static)
- Issue packet: set of *ins* that bundled together. Not an arbitrary mix; Determined by resources required and order (dependency) → pad with noop if necessary.
- Multiple functional units, multi-ported RFs, wide program bus

3.1.1 Dual-issue VLIW MIPS



Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	

Need 4 read ports and 2 write ports and a separate memory address adder (for load/store).



Harzards are more severe. EX forwarding will not work for two *ins* in same packet. It is same for load-use hazard. → Need more **aggressive code scheduling**.

1. Eliminate if-else branch structures to **predicated instructions**. (similar to ternary operator). This converts control hazard to data hazard. Useful for VLIW.

2. **Loop unrolling** makes multiple copies of the loop body and *ins* from different iterations are scheduled together → more ILP

```
lp:  lw    $t0,0($s1)    # $t0=array element
     addu  $t0,$t0,$s2   # add scalar in $s2
     sw    $t0,0($s1)    # store result
     addi  $s1,$s1,-4    # decrement pointer
     bne   $s1,$0,lp     # branch if $s1 != 0
```

	ALU or branch	Data transfer	CC
lp:		lw $t0,0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$0,lp	sw $t0,4($s1)	4

Scheduled code <u>without</u> loop unrolling. `addi` precedes `addu` and `sw`. 4 CCs to execute 5 *ins* → IPC=1.25 (↔ best case 2.0)

```
lw    $t0,0($s1)     # $t0=array element
lw    $t1,-4($s1)    # $t1=array element
lw    $t2,-8($s1)    # $t2=array element
lw    $t3,-12($s1)   # $t3=array element
addu  $t0,$t0,$s2    # add scalar in $s2
addu  $t1,$t1,$s2    # add scalar in $s2
addu  $t2,$t2,$s2    # add scalar in $s2
addu  $t3,$t3,$s2    # add scalar in $s2
sw    $t0,0($s1)     # store result
sw    $t1,-4($s1)    # store result
sw    $t2,-8($s1)    # store result
sw    $t3,-12($s1)   # store result
addi  $s1,$s1,-16    # decrement pointer
bne   $s1,$0,lp      # branch if $s1 != 0
```

(a) Unrolled code

ALU or branch	Data transfer	CC
addi $s1,$s1,-16	lw $t0,0($s1)	1
	lw $t1,12($s1)	2
addu $t0,$t0,$s2	lw $t2,8($s1)	3
addu $t1,$t1,$s2	lw $t3,4($s1)	4
addu $t2,$t2,$s2	sw $t0,16($s1)	5
addu $t3,$t3,$s2	sw $t1,12($s1)	6
	sw $t2,8($s1)	7
bne $s1,$0,lp	sw $t3,4($s1)	8

(b) **with unrolling**. 8 CCs for 14 *ins*. IPC=1.8

The compiler uses **register renaming** to solve name dependencies and ensures no load use hazards occur. VLIW primarily depend on the compiler for branch prediction.

3.1.2 Pros and cons

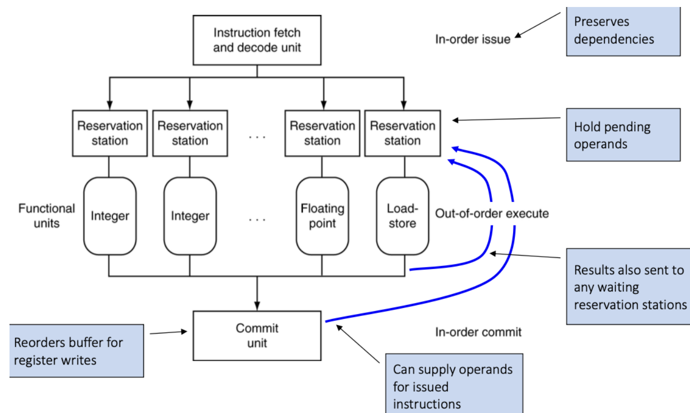
- 👍 Simpler hardware, potentially more scalable
- 👎 Programmer/compiler complexity and longer compilation
- 👎 Object(binary) code incompatibility
- 👎 Code bloat: noop/loop unrolling waste program memory space
- → Big failure; Compiler cannot know much about program

3.2 Dynamic multiple-issue aka *superscalar*

- **HW-driven**. Allow the CPU to execute instructions **out of order** to avoid stalls, but commit result to registers in order.
- Avoids need for compiler scheduling–though it may still help.

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

Example. Can start `sub` while `addu` is waiting for `lw`.



1. **Instruction-fetch and issue** (IFD unit): fetch instructions, decode them, and issue them to a FU to await execution
   - Instruction lookahead: fetch/decode/issue *ins* beyond current *in*
2. **Instruction-execution**: as soon as the **source operands and the FU** are ready, the result can be calculated
   - Processor lookahead: complete exec of issued *ins* beyond curr *in*
   - Machine parallelism: multiple FUs
3. **Instruction-commit**: when it is safe to, write back results to the RFs or D$ (i.e. change the machine state)

3.2.1 Why dynamic?

- Not all stalls are predictable, e.g. cache misses.
- Branch outcome is dynamically determined

3.2.2 Output dependence: write after write

↔ true dependence: read after write

```
lw    $t0,0($s1)
addu  $t0,$t1,$s2
. . .
sub   $t2, $t0, $s2
```

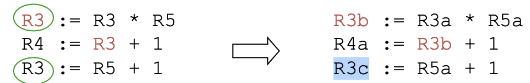
Example. If `addu` occurs first, `sub` gets incorrect value

3.2.3 Antidependence: write after read

When a later instruction (that executes earlier) produces a data value that destroys a data value used as a source in an earlier instruction.

3.2.4 Storage conflicts and register renaming

Anti- and output dependencies arise because the limited number of registers. → Register renaming.



- **Processor**: renames the original register identifier to new one (not in visible register set). Now register names are unique. On instruction issue to reservation station (RS),
  - If operand is available in RF, reorder buffer, or FU output: copy to RS (i.e. proceed)
  - If not, provide again to RS (i.e. stuck)
- **HW**: assigns register from a pool of free registers

3.2.5 Speculation

- Predict branch and continue issuing
  - ⚠ Don't commit until branch outcome determined
- Load speculation:
  - Predict the effective address or loaded value
  - Load from ongoing store (e.g. in write buffer)
  - Bypass (forward) stored value to load unit
  - → Avoid load and cache miss delay
  - ⚠ Don't commit load until speculation cleared

3.2.6 Multi-issue doesn't work as much as we'd like

- Some dependencies are hard to eliminate e.g. pointer aliasing (two pointers pointing same value)
- Some parallelism is hard to notice ∵ limited window size
- Memory delay and limited bandwidth → hard to keep pipeline full

4 Shared memory multi-processing

- Job-level (process-level) parallelism
- Parallel processing program (thread-level): single program run on multiple processors (cores)

Chip multi-Processors (CMPs) contain multiple cores in single integrated circuit (IC). **Key questions**:

Q1 How do they share data?
Q2 How do they coordinate?
Q3 How scalable is *arch*? How many processors can be supported?

Flynn's taxonomy by data and control (instruction) streams.

- Single-instruction single-data (SISD): Uniprocessor
- SIMD: GPU, MISD: limited use-case, MIMD: most common

4.1 Communication in multiprocessors

- **Message passing**: explicit messaging by programmers
  - Interface: MPI (message passing interface)
  - Code is running in both sender and receiver
- **Shared memory**: dominant for small- and medium-sized MPs
  - Single OS for all nodes
  - Implicit communication by loads and stores
  - Caches can hold copies of same *addr* → Cache coherence problem
  - Interface: `pthread`, OpenMP library

4.2 Shared memory multiprocessor (SMP)

Q1 Single address space shared by all *procs*
Q2 *Procs* communicate through shared *vars* in memory via load/store
- Use of shared data must be coordinated via synchronization primitives (locks) that allow access to only one *proc* at a time
- Uniform memory access(UMA) or nonuniform (NUMA)
  - NUMA: each *proc* have local memory of faster access time → harder programming but scale to larger sizes

4.3 Coherence problem: Propagating writes

4.3.1 Update-based protocols

- All updates must be sent to other caches and main memory
- Huge write traffics through the networks to other caches
- Useful for producer-consumer communication

4.3.2 Invalidation-based protocols

- Send invalidations (efficient!) to other caches to update
- Access to invalidated → cache miss, writer (or memory) provide data

Unit of invalidation is cache blocks. Even if only a part is updated, need to invalidate the entire blocks. What if two *proc*s read/write different part to same cache block? (**false sharing**) → Programmers can align data structure to cache block size

4.3.3 Two classes of invalidation-based protocols

- Cache controller determines cache state. Need to know **sharing state** i.e. which caches have copy for given address?
- **Snoop**-based: each *proc* has cache controller
- **Directory**-based: has centralized repo (directory) of sharing states

4.3.4 Snoop-based protocol

1. Any cache miss request must be put on the bus
2. All caches and memory observe bus requests (snoop tag lookup)
3. All caches check their cache tags and put responses:
   - Just sharing state (I have a copy)
   - Data transfer (I have a modified copy, sending it to you!)

Two sources of cache state transition: CPU (load/store) or snoop (request from other processors). **Architecture** for snoopy protocols:

- Extended cache (coherence) states in tags
- A set of wires connect all nodes and memory e.g. bus
- Serialization by bus: only one *proc* is allowed to send invalidation

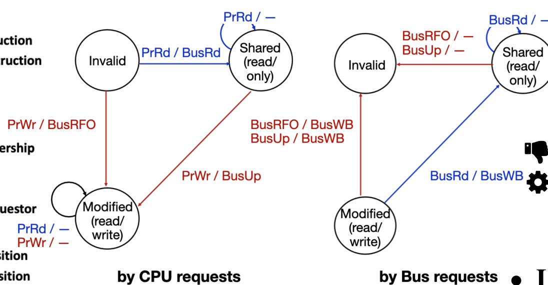
4.3.5 MSI protocol → 2 bits required

- **M** (modified): valid and dirty
  - Only one M copy can exist in the entire system
  - Can update without invalidating other caches
  - Must be written back to memory when evicted
- **S** (shared): produced by read, valid and clean / **I** (invalid)

CPU requests
- Processor Read (PrRd): load instruction
- Processor Write (PrWr): store instruction
- Generate bus requests

Bus requests (snoop)
- Bus Read (BusRd)
- Bus RFO (BusRFO): Read For Ownership
- Bus Upgrade (BusUp)
- Bus Writeback (BusWB)
- May need to send data to the requestor

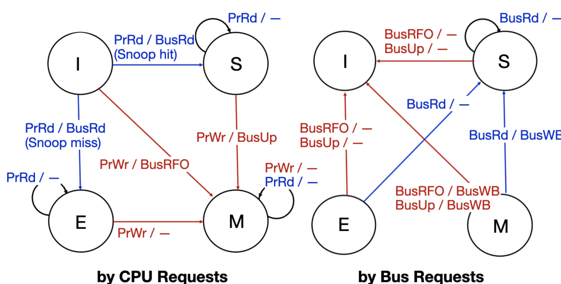
Notation: A / B
- A : event which causes state transition
- B : action generated by state transition



4.3.6 MESI protocol

Load → store sequence is common. And high chance that on other caches have a copy. Then why send `BusUp` when S → M?

```
Load   R1, 0 (R10)    → bring in read only copy
Add    R1, R1, R2
Store R1, 0 (R10)      → need to upgrade for modification
```



→ Add **E** (Exclusive) state
- Valid, clean, and no other caches have a copy of the block.
- E→ M transition is free (no bus transaction).

4.3.7 Multi-level caches

If without inclusion property, must snoop both caches. But commonly, with complete inclusion property:

- Snoop only L2 caches first
- If snoop hits L2, forward snoop request to L1
- If L1 have dirty copy, write-back to L2 (and memory in some designs)

4.4 Synchronization

- Should tell when it's safe for different processors to use shared data
  ⚠ This is not a coherence problem. This is a semantic information that should be provided by programmer.
- **Critical section**: code segment where processes/threads access shared resource
- **Synchronization** prevents parallel access to critical section.
- Barrier: all threads should reach barrier to pass it
- **Locks**: low-level primitive to regulate access to shared data
  - Critical section between `acquire` and `release`

4.4.1 Spin locks

Processor continuously tries to acquire, spinning around loop

```
li      R2,#1
lw      R3,0(R1)     ;load var
bnez    R3,lockit    ;≠ 0 ⇒ not free ⇒ spin
sw      R2,0(R1)
```

This `acquire` doesn't work. Multiple threads can pass `bnez`. ⚠ *Mem* value shouldn't change between load&store. → Need **atomic load& store**

- **Test and set**: tests a value and sets it if the value passes the test
- **Atomic exchange**: interchange a value in a *reg* for a value in *mem*
  - Synchronization variable in memory: 0 if free, 1 if locked
  - `acquire` = set register to 1 and swap
  - New value in register determines: 0 if success, else 1

```
         li      R2,#1
lockit:  exch    R2,0(R1)      ;atomic exchange
         bnez    R2,lockit     ;already locked?
```

👎 Includes a write which invalidates all other copies → bus traffic
→ Start by repeatedly reading the variable. Try exchange when it changes. ("**test and test&set**")

```
try:     li      R2,#1
lockit:  lw      R3,0(R1)      ;load var
         bnez    R3,lockit     ;≠ 0 ⇒ not free ⇒ spin
         exch    R2,0(R1)      ;atomic exchange
         bnez    R2,try        ;already locked?
```

👎 Read and write in 1 *in* makes critical path too long → use 2 *ins*!
⚙ **Impl**: only separated by HW; SW sees as one *in*. Load invalidate other caches. Until store is completed, any invalidation from other cache is held.
- **Load-linked** (or locked) (LL) and **store-conditional** (SC)
  - LL returns the initial value, SC returns 1 if succeeds i.e. no other store to same *mem* location since preceding load else 0
  ⚙ **Impl**: remember last load-locked address (lla). Invalidation to lla from other *proc*s set lla (iteself!) to 0. SC fail if lla is 0.

Example doing atomic swap

```
try:  mov    R3,R4       ; mov exchange value
      ll     R2,0(R1)    ; load linked
      sc     R3,0(R1)    ; store conditional
      beqz   R3,try      ; branch if store fails (R3 = 0)
      mov    R4,R2       ; put load value in R4
```

Example doing fetch & increment

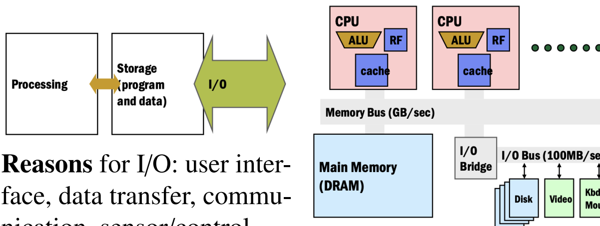
```
try:  ll     R2,0(R1)    ; load linked
      addi   R2,R2,#1    ; increment (OK if reg–reg)
      sc     R2,0(R1)    ; store conditional
      beqz   R2,try      ; branch store fails (R2 = 0)
```

Spin lock implementation

```
lockit:   ll      R2,0(R1)     ;load var
          bnez    R2,lockit    ;≠ 0 ⇒ not free ⇒ spin
          addui   R2,R2,#1
          sc      R2,0(R1)
          beqz    R2,lockit
```

Coarse-grained lock	Fine-grained lock
One for large DS	Many for different parts of DS
Simple code, bad performance	Difficult code

5 I/O and bus

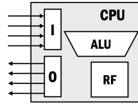


Reasons for I/O: user interface, data transfer, communication, sensor/control

Overview. How do CPUs talk to I/O?

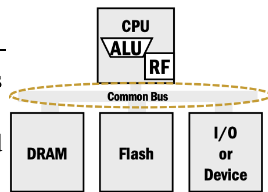
5.1 I/O mechanisms

5.1.1 I/O Port Registers (special purpose)



- Dedicated register for I/O
- 👍 Easy to use, low latency
- 👎 Not generally applicable

5.1.2 Memory-Mapped I/O: a general approach



- Abstract all I/O devices as memory.
- Map a subset of unused memory addresses to registers of external devices
- Load/store instructions perform I/O
- Memory and devices on bus respond only to their own address ranges

⚠ **Idempotency**: Memory load/store semantics are idempotent. However, reading/writing a device register can imply other state changes.
⚠ **Cache** can go very wrong. Usually disallow on mmap addresses.
⚠ **Slow** and consumes CPU cycles. The unit of `lw` is too small!

5.1.3 Direct memory access (DMA)

💡 Allow I/O device read/write large blocks from/to memory directly! CPU only issue commands to DMA engine.

- **Use PA**: No translation necessary, but pages can be interleaved. → DMA command unfriendly.
- **Use VA**: DMA command friendly. Can copy any size. But requires translation.
- 💡 OS can allocate contiguous pages for DMA
- 💡 Smart DMA engines: OS creates in memory a linked-list of commands for moving contiguous blocks

5.1.4 Which I/O mechanism to use?

Performance considerations

I/O Bandwidth = transfer size / transfer time

Transfer time = { overhead } + {transfer size / raw_bandwidth}

	Raw bandwidth	Setup overhead	Suitable when transfer size
DMA	High	Large	Large
mmap	Low	No	Small

CPU considerations: Fraction of I/O, How long can I/O wait?

5.2 When to serve I/O?

- **Polling I/O**: CPU keeps checking.
  - Consider a keyboard device with 2 mmap registers:
    - `READY`: A read returns true if a new character is available
    - `DATA`: A read returns the new character typed and resets `READY` if no more characters are available
  - Most of the time nothing happens. Inefficient for infrequent, but latency-sensitive I/O events
- **Interrupt-Driven I/O**: doesn't bother CPU, but costs ↑ cycles
  - Interrupt vector holds the reason of interrupt
  - I/O interrupt → CPU switch to interrupt handler
  - → Suitable for very infrequent (any human input interface), very long-latency operations (e.g. end of DMA transfer)

5.3 Bus

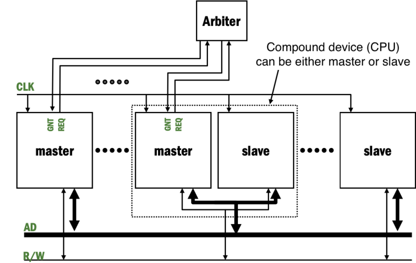
- A common datapath connecting multiple devices → Reducing interconnection cost
- Single driver, multiple receivers at a time
- Time-multiplexed by transactions → Bandwidth is shared
- **Protocol**: a set of handshake signals
- **Devices**: **master** initiate transactions, **slaves** respond, **arbiter** manages shared bus usage (special)

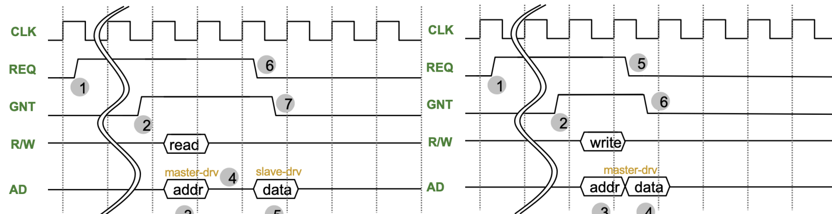
5.3.1 Bus transactions

1. Master requests ownership for the bus by asking arbiter
2. Arbiter grants ownership to master
3. Master drives address for all to see
4. Slave claims transaction (i.e. "that address belongs to me")
5. Master/slave drives data (depending on read/write) for all to see
6. Master terminates the transaction and bus ownership

5.3.2 Basic bus signals

- `CLK`: all devices synchronized by a common clock
- **Private** signals to/from arbiter per master:
  - `REQ` (output): assert to request ownership; de-assert to signal the end of transaction
  - `GNT` (input): ownershipis granted
- **Broadcast** signals shared by all devices
  - `R/W` (bi-directional): bus commands e.g. read, write
  - `AD` (address/data bus, bi-directional): ma drives address during address phase, ma/sl drives data during data phase





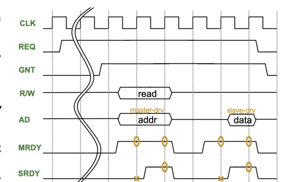
1. **Master requests bus**
2. **Arbiter grants bus**
3. **Master drives address/command, to be sampled on clock-edge**
4. Bus-turnaround cycle
5. Slave drives data
6. Master signals final cycle
7. Arbiter acknowledges

1. **Master requests bus**
2. **Arbiter grants bus**
3. **Master drives address/command, to be sampled on clock-edge**
4. Master drives data
5. Master signals final cycle
6. Arbiter acknowledges

5.3.3 Asynchronous bus protocols

We assumed slave can decode address in 1 cycle and respond in 1 cycle. Realized only with fixed-latency protocols → Async handshaking!

- Driver only asserts `MRDY` when `AD` value is valid. Receiver only asserts `SRDY` when ready to accept `AD` value
- A bus cycle is valid if `MRDY` and `SRDY`
- Receiver only pays attention if the driver is ready. Driver repeats value until the receiver is ready



Async read transaction

5.3.4 Performance

- **Latency**: Request/grant latency is *fn* of bus contention and arbitration strategy. Transaction latency is *fn* of slave reaction time.
- **Throughput/bandwidth**: N-byte bus at freq f has peak bandwidth Nf. Each transaction has overhead cycles; can be amortized.

5.3.5 Error (bit-flip) detection

The parity (number of 1's *mod* 2) bit for receiver to check. Guaranteed to detect odd flips but not even flips or identify which bits flipped.

A Problems

A.1 Slide 10

A.1.1 p.20-23

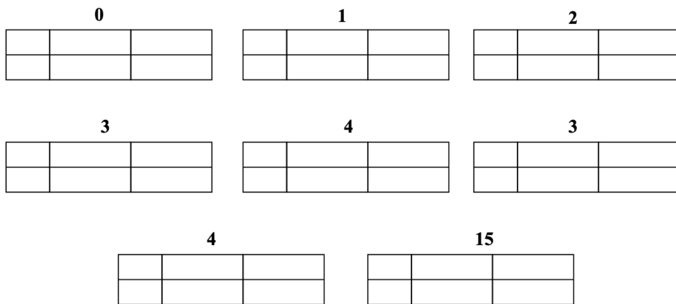
8-blocks, 1 word/block, direct mapped. For all access, determine hit/miss and cache block. Fill in the two tables. For data, the format is `Mem[10110]`.

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Word addr	Binary addr	Hit?	Cache block
22	10 110	Miss	110
26			
16			
3			
16			
18			

A.1.2 p.27

2-words direct mapped cache of size 2. Tag is 2 bits. Text means word address. Address is 4 bits. **Why**? Fill in the blank, indicate hit/miss and cache eviction.



A.1.3 p.31

How many total bits are required for a direct mapped cache with 16KB of data and 4-word blocks assuming a 32-bit address?

A.2 Slide 11

A.2.1 p.9

A processor with a CPI_{ideal} of 2, a 100 cycle miss penalty, 36% load/store instr's, and 2% I$ and 4% D$ miss rates. What is CPI_{stall}?

- What if the CPI_{ideal} is reduced to 1? 0.5? 0.25?
- What if the D$ miss rate went up 1%? 2%?
- What if the processor clock rate is doubled (doubling the miss penalty)?

A.2.2 p.23

A processor with a CPI_{ideal} of 2, a 100 cycle miss penalty (to main memory) and a 25 cycle miss penalty (to L2$), 36% load/stores, a 2% (4%) L1 I$ (D$) miss rate, add a 0.5% L2$ miss rate. What is CPI_{stall}?

A.2.3 p.15-16

Compare 4-block caches: Direct-mapped, 2-way set associative, and fully associative. Block access sequence is 0, 8, 0, 6, 8 (address). No byte offset. For cache content the format is `Mem[6]`.

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss				
8	0	miss				
0	0	miss				
6	2	miss				
8	0	miss				

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss				
8	0	miss				
0	0	hit				
6	0	miss				
8	0	miss				

Block address		Hit/miss	Cache content after access			
0		miss				
8		miss				
0		hit				
6		miss				
8		hit				

A.2.4 p.26-27

- CPU base CPI = 1, clock rate = 4GHz
- L1 miss rate/instruction = 2%
- Main memory access time = 100ns
- L2 cache access time = 5ns
- Global miss rate to main memory = 0.5%

What is miss penalty (as in cycles) and effective CPI with

1. Just primary cache?
2. L2 cache?

A.2.5 Bonus: LRU for n-way cache

? How many bits need to implement true LRU in n-way cache?
💡 Assume a 4-way cache. We want to remember a whole order of accession. Then there are 4! = 24 permutations. So we need 5 bits. So we need $\lceil \log n! \rceil$ bits. **?** Now think about the **access update and decoding logic** for this.

A.3 Slide 11

A.3.1 p.20

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	Hit	Hit	Yes – what we want!
Hit	Hit	Miss	Yes – although the page table is not checked if the TLB hits
Miss	Hit	Hit	Yes – TLB miss, PA in page table
Miss	Hit	Miss	Yes – TLB miss, PA in page table, but data not in cache
Miss	Miss	Miss	Yes – page fault
Hit	Miss	Miss/ Hit	Impossible – TLB translation not possible if page is not present in memory
Miss	Miss	Hit	Impossible – data not allowed in cache if page is not in memory

A.3.2 p.23

? Cache tag uses physical address, so need to translate before cache lookup. Why not a virtually addressed cache, which would only require address translation on cache misses?

💡 Two programs which are sharing data will have two different virtual addresses for the same physical address–aliasing–so have two copies of the shared data in the cache and two entries in the TLB which would lead to coherence issues.

A.4 Slide 13

A.4.1 p.4

A 6 GHz, four-way multiple-issue processor can execute at a peak rate of 24 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4.

A.5 Slide 14

A.5.1 p.6

$$T_{\text{new}} = T_{\text{parallelizable}}/100 + T_{\text{sequential}}$$

$$\text{Speedup} = \frac{1}{(1 - F_{\text{parallelizable}}) + F_{\text{parallelizable}}/100} = 90$$

100 processors, the F (fraction) of sequential part for 90× speedup?

A.5.2 p.7-8

Workload is sum of 10 scalars (sequential) and 10×10 matrix sum (parallelizable). Speedup if 10 processors? if 100 processors? What if 100×100 matrix?

A.5.3 p.32

Step	P1		P2		P3		Bus		Mem
	State	Value	State	Value	State	Value	Action	Proc	Value
	I		I		I				10
P1 read A	S	10	I		I		BusRd	P1	10
P2 read A	S	10	S	10	I		BusRd	P2	10
P2 write A (20)	I		M	20	I		BusUp	P2	10
P3 read A	I		S	20	S	20	BusRd	P3	20
P1 write A (30)	M	30	I		I		BusRFO	P1	20

A.5.4 p.37

Step	P1		P2		P3		Bus		Mem
	State	Value	State	Value	State	Value	Action	Proc	Value
	I		I		I				10
P1 read A	E	10	I		I		BusRd	P1	10
P1 write A (15)	M	15	I		I		None		10
P2 read A	S	15	S	15	I		BusRd	P2	15
P2 write A (20)	I		M	20	I		BusUp	P2	15
P3 read A	I		S	20	S	20	BusRd	P3	20
P1 write A (30)	M	30	I		I		BusRFO	P1	

B T/F

- Larger cache size always reduce the miss rate. (T)
- Larger cache size increases the access time. (T)
- Instruction cache has write port. (F)
- The memory speed is unlikely to improve as fast as processor cycle time. (T)
- As you increase level of memory hierarchy, worst case penalty increase. (T) But CPI decreases / is beneficial because it is very unlikely to reach that worst level (∵ miss rate is multiplied in AMAT). (T)
- The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller (i.e., faster) but have a higher miss rate. (T)
- The L2$ hit time determines L1$'s miss penalty. (T)
- L2$ local miss rate >> global miss rate. (T)
- In inclusive cache, lower level memory always contain data of higher level memory. (T)
- Page fault handler is in hardware. (F)
- TLB misses are much more frequent than true page faults. (T)
- TLB miss handled by OS is slower but more flexible. (T)
- Processes cannot access physical memory directly, but only through translation by the page table. (T)
- Page fault is detected by hardware (and then handled by OS). (T)
- In VLIW, we need n PCs to handle n issues. (F)
- Noops don't count towards performance when calculating IPC of multi-issue processors. (T)
- Loop unrolling reduces the number of conditional branches. (T)
- In superscalar, if exceptions occur the only registers updated will be those written by instructions before the one causing the exception. (T)
- Antidependence is like true dependence but reversed.
- A single device could have both master and slave functions. (T)
- A memory is always slaves. (T)
- A CPU is always slaves. (F)
- `REQ/GNT` is an example of asynchronous handshake. (T)