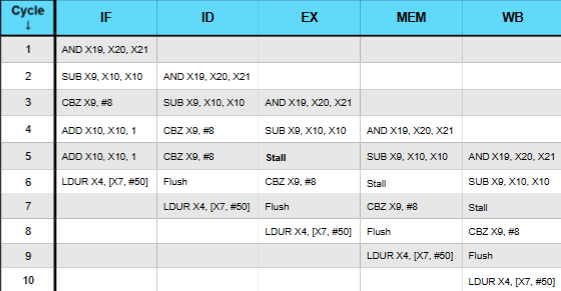
|  |  |
| --- | --- |
| Cache hit | Cache miss |
| When data is found at a given memory level | When data is not found |
| Hit rate: percent of references found at a given memory level | Miss rate: 100 – Hit rate |
| Hit ratio: number of hits / total number of references | Miss ratio: 1 – Hit ratio |
| Hit time: time needed to access data at a given memory level | Miss penalty: time required to process a miss |



- **fastest to slowest**: registers < L1 cache < L2 cache < main memory < disk < archival

- the idea is to transfer data between the memory with faster access times to reduce overall time.

- **spatial locality**: instructions tend to be accessed sequentially (they are near each other in memory); arrays are together in memory

- **temporal locality**: reuse of data or instructions previously accessed

- **L1** = primary cache and on same chip as CPU, **L2** = secondary cache can be on the same chip or separate; L2 is larger than L1 and has a longer access time

- **split cache system**: instruction (I-cache) and data (D-cache) are separate

- the separation of data from instructions provides better locality, at the cost of greater complexity

- 1) Blocks containing a desired item is loaded into cache; 2) blocks are copied from main memory into L2 and L1 cache; 3) L1 contains a subset of information in L2; 4) cache line holds a block read from main memory

- blocks are in main memory, lines in cache, blocks and lines are the same size

- **write through**: memory is updated in parallel with every write; somewhat defeats the purpose of having a cache

- **write back**: updates only the cache copy of the item; main memory is updated when the item is removed from the cache; must record whether cache line was written (dirty); may cause issues with concurrent access (e.g. shared memory multiprocessors)

- **lookaside cache**: accesses to cache and to main memory occur in parallel; main memory access is cancelled if hit in cache occurs; tends to lower average memory access time; increases CPU to memory traffic

- **look through cache**: first level cache is checked first; next level is only checked if miss occurs; avoids unneeded CPU-to-memory traffic; tends to increase average memory access time

Instead the system is now **a direct mapped** cache of size 28 bytes, with nothing else changed. How many bits are in the tag, line, and offset field? If the address 0xABC is requested, what are the values of those fields?

Number of lines = cache size / line size 28 bytes/cache / (4 bytes/line) = 26 lines / cache log(26) = 6, 6 bits in set field

Offset = 2 bits, line = 6 bits, tag = 12 – 6 – 2 = 4 bits

Ex.) Consider a system with a main memory access time of 200 ns supported by a L1 cache having a 10 ns access time and a hit rate of 90%, and an L2 cache having a 20 ns access time and a hit rate of 95%. **What is the AMAT if for a look aside cache**?

0.9 \* (10 ns) + 0.1 \* (0.95 \* 20 ns + 0.05 \* 200 ns) = 9 + 0.1 \* (19 + 10) = 9 + 1.9 + 1 = 11.9 ns

Consider a system with a main memory access time of 200 ns supported by a L1 cache having a 10 ns access time and a hit rate of 90%, and an L2 cache having a 20 ns access time and a hit rate of 95%. **What is the AMAT if for a look through cache**?

10 ns + 0.1 \* (20 ns + 0.05 \* 200 ns) = 10 + 0.1 \* (20 + 10) = 10 + 3 = 13 ns

A processor has a combined data and instruction cache with a hit rate of 0.99. If it is a cache miss, there is a penalty of 100 cycles. Ideal CPI, with no cache miss, is 4. 30% of instructions are load/store. What is the average CPI?

For instructions in cache, CPI is:

0.7 \* 4 + 0.3 \* (0.99 \* 4 + 0.01 \* 104) = 2.8 + 0.3 \* (3.96 + 1.04) = 2.8 + 0.3(5) = 4.3

Now add the effect of instruction cache misses

0.99(4.3) + 0.01(104.3) = 4.257 + 1.043 = 5.3

- **compulsory misses**: unavoidable due to initially empty cache; **capacity**: due to limited size of cache; **conflict**: due to different blocks mapping to the same cache line

- main memory is split into blocks & blocks will be loaded into cache; bits in memory address = log2 (main memory bytes); block number = address / block size

- addresses of items map to blocks; **number of bits in offset**: log2 (Block size)

- **number of blocks**: main memory size / block size; **number of lines** = cache size / block size

- n-way associative means there are N blocks per set

- **fully associative:** any memory block can go into any empty cache line; all lines are checked to detect cache hits or misses; requires hardware to compare all lines in parallel; a valid bit for each line is set if the line is occupied; block is loaded into vacant cache line if miss occurs; if no line is vacant, a replacement must occur. Has a tag field and offset field. Offset field = log2 (Line size). If a stored tag matches the tag field in the address and the line is valid, there is a hit. Needs a separate comparator for every cache line.

- **direct mapped:** only one comparator is needed for the entire line. A direct mapped cache with N lines: Block B of memory maps to line L = B mod N; B is the block number, not the address. Split into 3 fields: offset field indicates location within line, line field selects a unique line of cache, tag field is whatever is left over. **Offset width** (OW) = log2 (Line size). **Line # Width (**LW) = log2 (Number of lines). **Tag width** = bits in address – LW – OW. Multiple addresses with same line number causes conflicts. Cache line can hold only one candidate block. Other vacant lines may be unused. Causes increased miss ratio. Can hurt performance. 0 hit ratio if alternating between conflicting addresses.

- **set associative:** blocks are put into sets; a number of blocks can be mapped to and put in the same set; reduces conflicts; addresses are divided into tag, set, and offset; set field determines set (group of lines) to which the memory block maps; tag field identifies which line within the set contains the block (if it is there); offset field identifies the location within the line . **Offset width** = log2 (Block size). **Set Width** = log2 (Number of sets). 2-way set associative cache (2 lines per set). Block can be loaded in any vacant line in set. Replacement is required if set is full. Most common choice for replacement scheme is LRU (least recently used) or first in, first out.

Ex.) A system has a main memory size of 212 bytes. It has a block size of 4 bytes. How many bytes wide is the (A) address and the (B) offset field

A) log(212) = 12 bits B) log(4) = 2 bits

Ex.) The system described above has a cache of size 28 bytes. How many bytes are in the tag and offset field if this cache is **fully associative**? If the address 0xABC is requested, what is the tag and what is the offset?

Offset = 2 bits (from first example), tag = 12 bit address – 2 bit offset = 10 bits

0xABC = 1010 1011 1100 tag = 1010101111 offset = 00 (order: tag - offset)

Ex.) Instead the system is now a **2-way associative cache** of size 28 bytes, with nothing else changed. How many bits are in the tag, set, and offset field? If the address 0xABC is requested, what are the values of those fields?

How many sets? 2-way associative means each memory block is mapped to 2 lines (a set) in the cache

Number of sets = cache size / size of set Size of set = (line size \* number of lines in set)

28 bytes / cache / (4 bytes/line \* 2 lines/set) = 25 sets / cache

Log(25) = 5, so 5 bits in the set field offset = 2 bits, set = 5 bits, tag = 12 - 5 - 2 = 5 bits

A screenshot of a computer

Description automatically generated

A white board with red writing

Description automatically generated

- datapath: reads an instruction, executes instruction, writes registers

- combinational elements: gates, combination of gates (ex.) ALU); outputs depend only on inputs

- state elements: memory/stage; data read from register is state value at previous clock cycle; data written to register will be the new state at the next clock cycle

A diagram of data processing

Description automatically generated

Pipelining ex.) A single cycle processor with a 1000 ps clock cycle can be split into 5 stages with latencies (duration) of 200 ps, 300 ps, 150 ps, 100 ps, 250 ps

New cycle time: 300 ps

How long does it take an instruction to get through the pipeline: 5 x 300 ps = 1500 ps

What is the speedup of processing 5 instructions: 5 (stages) + (5 – 1 instructions) = 9, pipelining cycle time = 9 x 300 ps = 2700 ps,

single processor cycle time = 5 x 1000 ps = 5000 ps, speedup = 5000/2700 = 1.85

- hazard: cause the instruction to not be able to execute

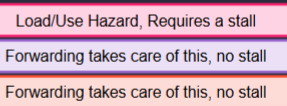
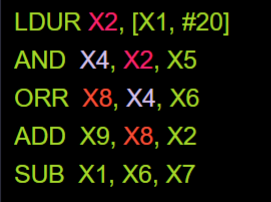
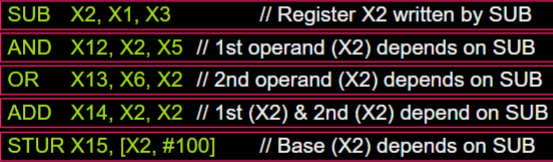
- structural hazard: two instructions need the same hardware in the same cycle

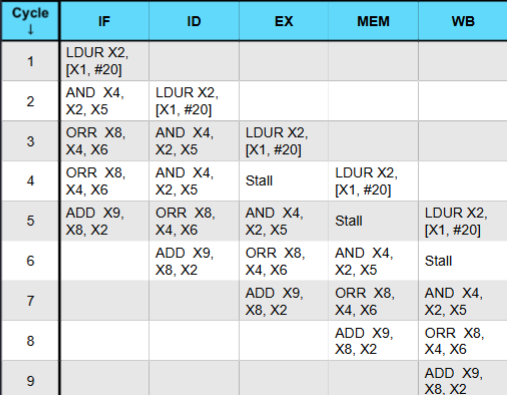
- data hazard: an instruction needs data from a register before it has been calculated

- control hazard: which instruction to fetch is unknown because the result of a branch decision is not known yet

- structural hazard solutions: **Harvard Architecture** separates data and instruction memory; **Split-Cycle I/O** where write occurs on the first half of the cycle and read occurs in the second half of the cycle

**Data Hazard Below:**





-Digital Logic: 1 = Asserted/true; 0 = Deasserted/false

Local branch history: what happened at this particular address in the past? Global Branch History: What has happened at all branches? Tournament Predictor: Use more than one predictor and choose the one that has been more accurate

- datapath: based on register transfers required to execute instructions

- control causes the correct actions to happen at the correct time

- Von Neumann Architecture: CPU is made up of datapath and control unit, then there is also a memory unit, input unit, and output unit

- control bus sends signals that request or notify other units of actions

**Pipelining**

- key point: separate resource for each stage

- speedup = old time / new time

- for a pipeline, each stage is a cycle

- the cycle time is the time of the longest stage

- each instruction takes five cycles to execute

A screenshot of a table

Description automatically generated

8 bits = 1 byte

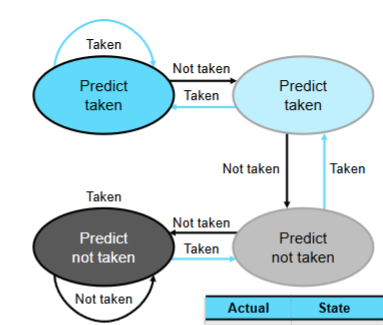
**static branch prediction:**

**-** backward conditional branches always taken (when the branch statement is at the end of the loop).

- forward conditional branches always not taken (when branch statement is at the front of the loop).

A screenshot of a table

Description automatically generated



Is the instruction LDR? Is the register being written by LDUR needed by the instruction in the register read stage? If yes, send message to stage 2 to send control signals that are all zeroes to stage 3. Send message to stage 1 to send same control signals to stage 2.

A diagram of a machine

Description automatically generated

- Forwarding is used when a value needed has been calculated, but has not been written to the register yet (the value hasn’t made it to the writeback stage)

- If forwarding won’t work, i.e. the necessary value has not been calculated, a stall is needed

- the X2 needed for the AND command is not available until after cycle 5 for the LDUR command. So the forwarding can only occur after cycle 5, so AND and ORR would need to stall for a cycle.