Floating point numbers:

Single precision: 32 bits [1 sign][8 exponent][23 Fractions]
Half precision: 16 bits [1 sign][5 exponent][10 Fractions]
value = (−1)^s×(1. Fraction)×2^E
decode encode excess (-/+)= 127
EXAMPLE:
1 10000011 001 0100 0000 0000 0000 0000
1 = Negative
10000011 = 131 - 127 = 4
1 + .00101 = 1.15625
-1 x 1.15625 x 2^4 = -18.5

-18.5
- = 1
18.5 = 10010.1
10010.1 = 1.00101 x 2^4
4 + 127 = 131 = 10000011
1 10000011 00101

RISCV float Example:

```
addi t0, zero, 0 # t0 = 0
fcvt.s.w fa0, t0 # sum = 0.0
slli t1, t0, 2 # t1 = i * 4
add t2, a0, t1 # t2 = x + i
add t3, a1, t1 # t3 = y + i
flw ft0, 0(t2) # ft0 = x[i]
flw ft1, 0(t3) # ft1 = y[i]
fmul.s ft2, ft0, ft1 # ft2 = x[i] * y[i]
fadd.s fa0, fa0, ft2 # sum += x[i] * y[i]
```

Single-cycle implementations

Processor Signals:
OPCODE to Control:
Inst AluSrc MemtoReg RegWrite MemRead MemWrite Branch AluOP
R-Type 0 0 1 0 0 0 10
Lw 1 1 1 1 0 0 00
sw 1 X 0 0 1 0 00
beq 0 X 0 0 0 1 01
JAL X 0 1 0 0 0 1 0
JALR X 0 1 0 0 0 1 1

Instruction opcode ALUOp Operation funct7 funct3 ALU function ALU operation
lw 00 load word xxx xxxx xxx add 0010
sw 00 store word xxx xxxx xxx add 0010
beq 01 branch if equal xxx xxxx xxx subtract 0110
R-type 10
add 000 0000 000 add 0010
subtract 010 0000 000 subtract 0110
and 000 0000 111 and 0000
or 000 0000 110 or 0001

Performance:
Performance Equations:
CPU Time = Clock Cycles X Clock Cycle Time
CPU Time = Clock Cycles/Clock Rate

Clock Cycles = Instruction Count x CPI

Performance = 1/CPU Time

CPI = Clock Cycles/Instruction Count

N = Execution timex/Execution timey

CPI = Instruction Counti / Instruction Count

Average CPI = CPI * percentage …

Speedup = CPU Time before / CPU Time After

The best speedup you can achieve by optimizing the 40% code is 1/.6 = 1.6

AMAT = Hit time down + Miss rate down × Miss penalty down
Parameters we can tune:
Cache size up
Block size up

Pipeline:
Forwarding:
ADD/SUB: EX/MEM -> EX
AND: MEM/WB -> EX
LW/SW: MEM/WB to MEM
LW->AND: MEM-> EX

Cache:
SRAM for speed, DRAM for storage

Temporal Locality: Keep most recently accessed data items closer to the processor
Spatial Locality: Move blocks consisting of contiguous words closer to the processor
Block size is always a power of 2

How to find block offset?
log_2(block size)
EXAMPLE:

How many bits are in block offset for block size of 64 bytes?
Answer: log_2(64) = 6

Used to differentiate info in block

How to get bits in block addresses?
Memsize - offset = block address

How to find cache index?
log_2(# of cache blocks)
EXAMPLE:
# of blocks = 1024
log_2(1024) = 10

Used to find block

Finding the actual index:
Use the index provided or block address mod # of blocks

Comes right after the offset in block address

When CPU needs to load byte, the address is sent to cache and returns hit or miss if tag bit and index is the same

Valid bit:
At the beginning, all cache blocks are invalid and data needs to be loaded
Checks to see if the data is valid or not
Block address is made up of tag and index

Copies higher block address to cache after not finding it and “evicts” old one so it will be miss if trying to find it

OVERVIEW:

Direct-Mapped Cache:
Block address is mapped to exactly one location in the cache
Many blocks are mapped to the same cache block
A tag is associated with each cache block for identifying a block
The access is a hit if and only if the cache block is valid and the tags match
Otherwise, it is a miss
On a miss, a new block is loaded into cache, replacing any old block

Offset:
Identify a byte/word in a block
Number of bits in the offset is determined by the block size

Cache index:
Locate a block in cache
Number of bits in index is determined by the number of blocks in cache

Tag:
Make sure the block found in cache is the correct block
Bits in an address excluding cache index and offset bits

Cache Access Summary

Memory address is divided into 3 fields. All bits are used!

1. Use cache index to locate a block in cache
2. Check if it is the block to be accessed.

If the cache block is valid AND the tag in cache matches the tag from the address

The access is a hit

Use the offset to select the byte/word

Else

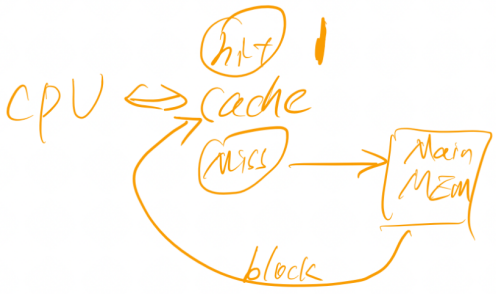
It is a miss

Load a block from memory into cache

Update tag

Set the valid bit

Access cache again



Structure and Access:

Cache size = # of blocks * block size
Amount of data that can be stored in cache

One word = 4 bytes
Finding how many blocks:
# of blocks = Cache size / block size

Total number of bits = number of blocks * (number of bits per block + tag size + validation bit)
EXAMPLE:
Cache Size = 4 Kib
Block Size = One Word = 32 bits
4096/4 = 1024 = # of blocks
32 - log_2(4 bytes) - log_2(1024) = 20 tag size
1024 * (32 + 20 + 1) = 54272

Spatial vs temporal = x vs y

Cache performance

Memory stall cycles per instruction examples:
Suppose 20% of the executed instructions in a program are load/store. Instructions miss rates of the instruction and data caches are 5% and 20%, respectively. The miss penalty is 50 cycles for both caches.

Misses per instruction instruction cache: 50 * .2 * .05 = .5
Misses per instruction data cache: .2 * .2 = .04

100 cycles miss penalty, 36% load/store
instructions, and 2% I cache and 4% D cache miss rates.
I Cache Misses/instruction = 1 × 0.02 = 0.02
D Cache Misses/instruction = 0.36 × 0.04 = 0.0144
Misses/instruction = 0.02 + 0.36 × 0.04 = 0.0344
MemoryStallCycles/Instruction = 0.0344 × 100 = 3.44
If CPIperfect is 2, what is the speedup of having a perfect memory?
CPIstalls = 2 + 3.44 = 5.44
The speedup of a perfect memory: 5.44 / 2 = 2.72
Percentage of time on memory stalls: 3.44 / 5.44 = 63%

AMAT = Hit time + Miss rate × Miss penalty
What is the AMAT for a processor with a 1 ns clock cycle time, a
miss penalty of 60 clock cycles, a miss rate of 3% misses per

instruction and a cache access time of 1 clock cycle?
AMAT = 1 + 0.03 × 60 = 2.8 cycles
Compulsory (cold start or process migration, first reference):
First access to a block, “cold” fact of life, not a whole lot you can do about it. If you are going to run “millions” of instruction, compulsory misses are insignificant
Capacity:
Cache cannot contain all blocks accessed by the program
Conflict (collision):
Multiple memory locations mapped to the same cache location
Larger cache → More data in cache → Lower miss rates
Larger block → More data are loaded in cache on a miss

Read hit/miss what we usually have
Write hit - through or back /miss - allocate or no allocate
Through memory or evict the cache
Load in cache then memory or only write to memory