1 ms = 10^{-3} s (millisecond)

1 us = 10^{-6} s (microsecond)

* 1 ns = 10^{-9} s (nanosecond)

Hz = cycle / s 1 kHz = 10^3 Hz

1 mHz = 10^6 Hz * 1 GHz = 10^9 Hz

in Bytes:

char (1) short (2) int (4)

float (4) double (8) long: 4 - 32b, 8 - 64b

pointer: 4 - 32b, 8 - 64b

2's complement: n-bit ⇒ Range: $[-2^{(n-1)}, 2^{(n-1)}-1]$

① Sign bit: 0 (+), 1 (-)

② exponent: 8 b (floating point)

Biased base -127 encoding

Add 127 to the value of the actual exponent to encode: Eg. -127 → 0.---0 (0), 0: 0(.---) (127), 1: 10.---0 (128), 127: 1.---1 (255)

③ significand (mantissa): 23 b

most significant bits after decimal

Eg. $(10.625)_{10} \rightarrow (1010.101)_2 \rightarrow (1.0101 \times 2^3)_2 \Rightarrow$ 0 | 10000010 | 0101010----0

130 = 127 + 3

	LC2K	ARM-LEG
	8 Reg, 1 word	32 Reg, 2 words
	4B Reg	8B Reg
	2^{16} memory	2^{64} memory
	8 insts	40 insts
	word-addressable	byte-addressable

little endian: 0x0A0B0C0D

0D	0
0C	1
0B	2
0A	3

big endian

0A	0
0B	1
0C	2
0D	3

2 KB is 2^{11} B *

$2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$

$2^5 = 32$, $2^6 = 64$, $2^7 = 128$, $2^8 = 256$

$2^9 = 512$, $2^{10} = 1024$

1 word = 4B, 1 Hex Digit ⇒ 4b

1 B = 8b, 1 KB = 2^{10} B ≈ 1K B,

1 MB = 2^{20} B ≈ 1M B, 1 GB = 2^{30} ≈ 1B B

RISC (Eg. LC2K, RISC-V, ARM)	CISC
1. fewer, simpler insts * small CPI (usually one)	1. more, complex insts * higher CPI
2. encoding of insts are usually the same size (more popular)	2. encoding of insts are usually in different sizes
3. simpler hardware	3. complex hardware
4. Program is larger, since every inst is simple, require more insts overall	4. Short, expressive programs. easier to write by hand, each (complex) inst is doing more things

write w/ cache

store w/ no allocate	write-back	write-through
hit	write cache	write cache + mem
miss	write to memory	write to mem
replace block	if dirty, write to memory	Do nothing

store with alloc	write-back	write-through
hit	write cache	write to cache + memory
miss	read from mem to cache, allocate to LRU block, write cache	read from mem to cache, allocate to LRU block, write to cache + memory
replace block	if dirty, write to mem	Do nothing

* 1. reduce compulsory miss = increase block size
2. reduce capacity miss = increase block size or increase cache size
3. reduce conflict miss = increase # blocks (ways) per set - associativity

* overhead bits = valid bits + tag bits + dirty bits + LRU bits

Block O.S. is not stored in the cache line

* Defined as part of the ISA:
- number of registers
- virtual address size

* Not defined as part of the ISA: - size of the cache - physical address size - whether a register is caller or callee saved - whether the PTs are single vs. multi

Detect and stall/avoidance:
2 noops for immediate data hazard
3 noops for control hazard

Detect and forward:
1 noop for immediate instruction dependent on "lw"

Speculate and Squash: 3 noops for mispredictions

AMAT = Total access time / # accesses

two possibilities for a physically-addressed: TLB access time + cache time + memory time

① TLB hit → cache
② TLB miss → go memory (PT) hit → cache hit
miss → go memory

"page fault"

Context switch: clear TLB

AMAT = TLB latency + TLB miss rate · (PT latency + Page fault rate · disk latency) + (TLB hit rate + TLB miss rate · PT hit rate) · (cache latency + cache miss rate · Mem latency)

* Problems with Physical Memory
① Physical Memory is limited
② Physical Memory isn't secure ⇒ Processes should not access each other's data *
③ Physical memory isn't consistent

Physically-addressed cache
- more common
- after VM-PM translation
- slower

VA → TLB lookup (hit → cache access; miss → PT walk (hit → update TLB; miss → disk access → update TLB and memory and PT → Data to processor and update cache))

Virtually-addressed cache
- less common
- faster access
- more complex

VA → Cache access (hit → data to cpu; miss → TLB lookup (hit → memory access → update cache; miss → PT walk (hit → update TLB; miss → disk access → update TLB, PT and memory)))

Context switch = clear TLB + cache

AMAT = cache latency + cache miss rate · (TLB latency + TLB hit rate · Mem latency + TLB miss rate · (PT/memory latency + page fault rate · disk latency + page hit rate · Mem latency))

* Page table entry size not mistakened as page size

(1) Single-level Page Table

entries = # Mappings = # virtual pages

size of page table entry = size of control bits + PPN bits

(2) Multi-level Page Table

total # entries in leaf level = # Mappings = # Virtual Pages

Size of leaf level Page Table entry = Size of control bits + PPN bits

Total # entries in Intermediate level = # PTs in Next level

Size of Intermediate level Page Table Entry = size of control bits + PPN bits / Physical Address size

*	FA	set-associative	directly-mapped
# sets	1	2^n	# blocks (#lines)
# set bits	0	n	$\log_2$ (# blocks)
# ways	n	k	1

$\log_2$ (# ways)

A	B	C	D	E	F
10	11	12	13	14	15
1010	1011	1100	1101	1110	1111

Detect and stall:

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
add 1 2 3	IF	ID	EX	ME	WB								
nor 3 4 5		IF	ID*	ID*	ID	EX	ME	WB					
add 6 3 7			IF	IF	IF	ID	EX	ME	WB				
lw 3 6 10						IF	ID	EX	ME	WB			
sw 6 2 12							IF	ID*	ID*	ID	EX	ME	WB

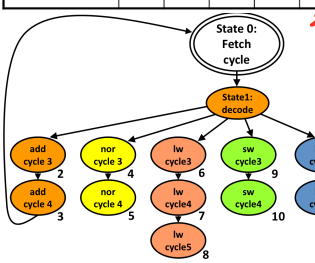
we paid the penalty once, we don't need to pay it again

Detect and forward:

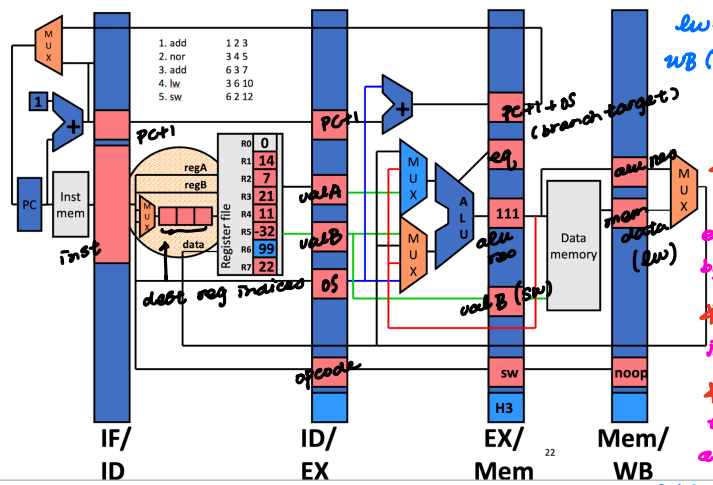
Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
add 1 2 3	IF	ID	EX	ME	WB								
nor 3 4 5		IF	ID	EX	ME	WB							
add 6 3 7			IF	ID	EX	ME	WB						
lw 3 6 10				IF	ID	EX	ME	WB					
sw 6 2 12					IF	ID*	ID	EX	ME	WB			

usually the longest

* D latch
1. When the Gate (clock) is high, Q (output) = D (input) meaning the latch is transparent
2. When the Gate (clock) is low, Q latches to the value of D at that instant and remembers, even if D changes later
* D flip-flop
Only store the value the instant the clock changes (edge)
* optimal achievable serial runtime:
add/nor: read memory (fetch) + read register file + ALU + write register file
lw: read memory (fetch) + read register file + ALU + read memory (loading) + write register
sw: read memory (fetch) + read register file + ALU + write memory
beq: read memory (fetch) + read register file + ALU (comparison)
noop / halt: read memory (fetch)
* direction prediction for jalr is not worse than beq because jalr is always taken



* single-cycle: longest inst
multi-cycle: longest component (write memory)
* Detect and forward: add/nor: forward reg from MEM (ExMEM) to the start of Ex and forward from WB (MEMWB) to the Ex
lw: forward reg from WB (MEMWB) to the start of Ex (1 stall required for immediate dependency)
* when counting # insts executed, we don't consider nops! (software)
* When an integer is loaded from memory into a reg, endianness defines the order bytes are placed in the reg
* Function parameters passed in memory are always stored on stack
* It is possible for a write-through cache to write less total bytes of memory than a write-back cache when running the same program
Write-through and write-back becomes the same if we only accessed a particular block one time before we evict. Write-back works the best if we access a particular block multiple times before evicting



* SRAM:
· volatile: need constant power to keep data
· Fast: 1/10 ns read time
DRAM: · volatile · Slower 40 ns
Disk: · non-volatile = holds information even when no power is supplied
* Reducing a processor's frequency decreases main memory latency measured in terms of processor's cycles.
frequency ↓ ⇒ clock period ↑ ⇒ time = clock period · # clocks ↓

* Block offset bits = $\log_2$(block size)
Set index bits = $\log_2$(# sets)
Tag bits = address size − set index bits − block offset bits
Address size = $\log_2$(size of memory)
cache size = # cache blocks · block size

* For a given program, memory accesses to its page table tend to exhibit lower spatial locality than memory accesses to its data

* PTs use VPNs to index into the PT to find PPN
* 1. When data is in the data cache: # cycles to access cache
2. When data must be fetched from memory:
* Assuming we have 2-levels of PTs
① hit in TLB: cycles for cache + cycles for memory
② miss in TLB and hit in two PTs: cycles for cache + cycles for 2 PTs + cycles for memory
3. When data must be fetched from disk
① miss in 1st level PT = cycle for cache + cycle for 1st level PT + cycle for disk + disk
② miss in 2nd level PT = cycle for cache + cycle for 2 PTs