SIMD

- víc ALU ☺ levné → vekt. instrukce (už první v ~1974)
  + vekt. registry (většinou 256 - 512 b)
- synchronizace ☹ drahá

- samotný výpočet ~ 10 % energie, ta režie ~ 90 %
  na CPU (~20 % na GPU)

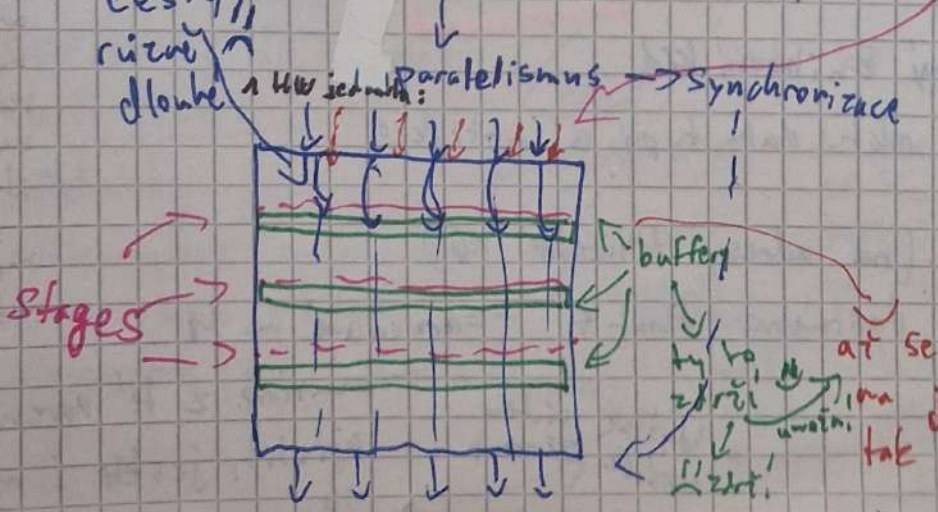
Vekt. registry

- neotypované - to, co tam je, se interpretuje podle té instrukce, která se na tom udělá
- lze použít i jako takovou další úroveň cache

Vekt. instrukce

- prostě normální, ale operandy jsou velké
  (vezmi 2 reg., něco s nima udělej, a ulož to do jiného reg.)
- N matematických op. najednou
  - L→ N lajn (lanes), (nezávislých (ale nemusí)) dokonce většina instrukcí z instr. sady
  - např. nějaké přesuny bitů, kryptograf. operace
- dnes podpora pro neuronové sítě (např. 16b floaty)

- hezké instrukce:
  - N HW jednotek ✗✗ 1 HW jednotka ✗✗ K HW jednotek → N/K batchů



1 instr. tam nacpe za sebou ty operandy → jednodušší manipulace s daty

nakonec to vyjde rychleji #batchů × hlubka

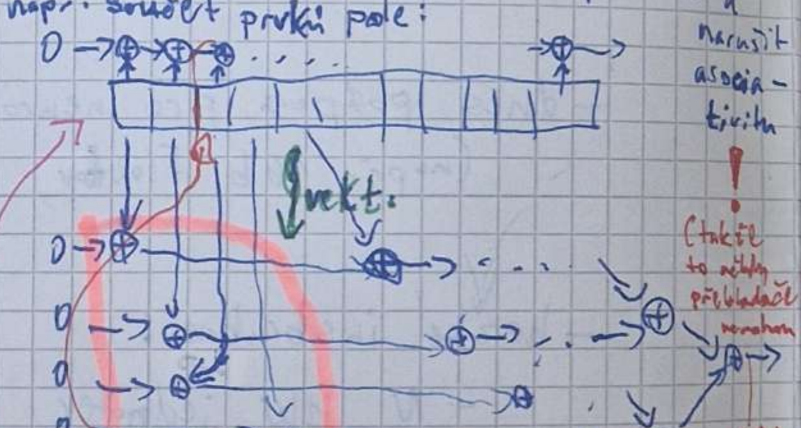
Paměťové přesuny

- kvůli vekt. instr. se musela zvětšit tloušťka těch kanálů (jediné ty ji dokážou celou využít → další důvod je používat)
  - ↓ L1 cache ⟷ výp. jádro procesoru
  - ↘ např. memcpy – ani ne pro ty vekt. aritmetiku, ale kvůli těm přesunům

SW support

- vektorizace překladači
  - cyklus ⇝ spojí se N po sobě jdoucích iterací do jedné
    - ↳ instr. se tím mohou přeházet a narušit asociativitu ! (takže to většinou překladače nemohou udělat, protože by to vedlo na jiný výsledek) ! např. u floatů

např. součet prvků pole:



  - polyhedrální kompilace – prohazování zanořenosti cyklů ...
- vektorizovaný knihovní kód
- explicitní vekt. dat. typy a instrukce
- zarovnání (na adresy dělitelné 16)
  - třeba dostaneme float* – zarovnané na 4 – musíme z té adresy zjistit, jestli i na 16 ⟶ a kdyžtak si ten začátek posunout

SIMD na Intel/AMD CPU

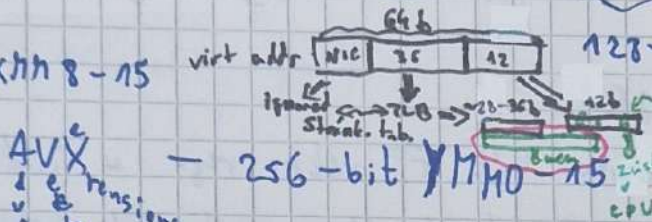
1997 – Intel využil existující fp (80-bit) registry mm0–mm7 na intové operace (8, 16, 32-bit)
– AMD i 32-bit floaty

1999 – SSE – 128-bit registry xmm0-7

2001–2007 – SSE2–SSE4 – 64-bit floaty a 8/16/32/64-bit inty

2003 – (AMD64) x64 – xmm 8–15

2011 – Intel + AMD – AVX (extensions) – 256-bit ymm0–15



128-bit. vektory

fp operace

2013 – AVX2 – + intová aritmetika
– gather/maskstore instrukce
↓ načítání i nesousedních úseků z paměti
↓ zapisuje se do souvislého úseku, ale jen nevymaskované části

2012 – (Intel) IMCI – 32 (512-bit) registrů zmm0–31, 60 jader
– používáno jako coprocesor
– ale FUJ

2016 – (Intel) AVX512
– 32 registrů, 512-bit
– jako hlavní procesory

Vekt. instr. → #include <immintrin.h>

- přístupy do paměti
- arit. instr.
- podmínky
- inter-lane aritmetika
- inter-lane míchání
- konverze (typy, skalár ↔ vektor)

sub r1, r2

latence = doba, která se musí počkat na tu instrukci

(inverzní) propustnost = kolik toků je na odstartování instrukce (např. 0.5 → 2 instr. za takt)

r1 -= r2
→ ale překladač umí dělat iluzi tohohle: r3 = r1 - r2
(a stejný princip je i u vektorových instr.)

(umí se odstartovat 1/latence instr. za takt)

Přístup do paměti

souvislé úseky

- load/store — adr. nedělitelné 16 B
- loadu/storeu (unaligned) – i nezarovnané operandy
- stream_load – obchází cache (většinou platí jen pro L1) – a někdy to je stejné jako load, když to celé asi

(spíš nechodí na přednášky)

dnes to umí všechny instr. kromě loadu, ale když to není zarovnané, tak je to pomalejší

\+ jiné vekt. instr. mohou mít jeden operand z paměti

Saturované operace = když by to něco přetekl, tak to vezme toho zůstane na min/max

Datové typy

– třeba mohou být FP vekt. reg. a jiné intové reg.

\+ i nějaké zkratky, že to jde třeba rovnou do ALU bez zastávky v registrech

≥ 3 typy loadů:
- intové
- single prec. floaty
- double prec. floaty

nesouvislé úseky: → pomalejší než souvislé!
- gather
- scatter (jen AVX512)

ale když to jinak nejde, tak je to rychlejší, než to dělat nějak skalárně

– dalším vstupním vekt. reg. jsou určeny adresy

gather: for(i in 0,...N-1): $v[i] = a[c * x[i]]$ (1/2/4/8 podle velikosti toho elementu)

scatter: for(i in 0...N-1): if($m[i]$) $a[c * x[i]] = v[i]$

Podmínky

```
for (...)
  if (...)
    x[i] = f(...)
  else
    x[i] = g(...)
```

musí se udělat obojí

a ze závor se udělá podmíněné přiřazení

- v SSE a AVX:

e[i] = a[i] == b[i] ? c[i] : d[i]

cond = cmpeq(a,b) → true: samé 1 ~> -1
→ false: samé 0 ~> 0

left = and(cond, c) ⇒ cond: 1 → c, 0 → 0 ; left = cond & c

right = andnot(cond, d) ⇒ cond: 1 → 0, 0 → d ; right = ~cond & d

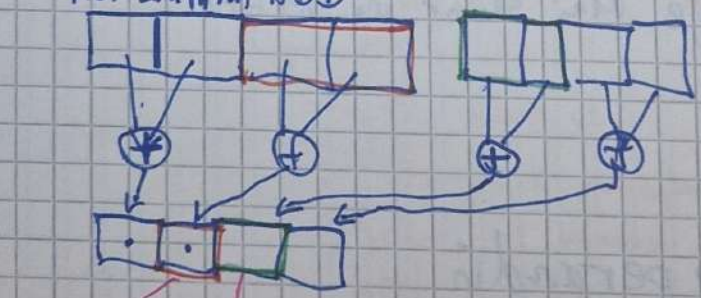
(pozor, vlastně je to notand)

e = or(left, right)

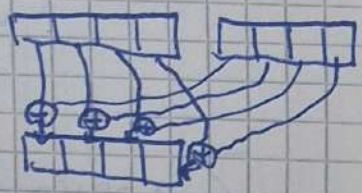
- v AVX512:
  - maskovací registry
  - porovnávací op. do nich dávají ty výsledky (1 bit/lajnu)
    - pak se v další instr. použije ta výsledná maska (+ dá se jí alt. hodnota, která se tam má dát, když ta maska je 0) (nebo je maska, která tam dá 0)

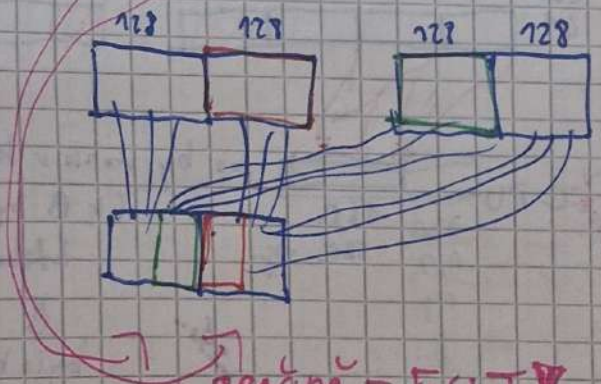
Inter-lane aritmetika

Horizontální ADD:



vs normální:





opačně - FUJ! POZOR!

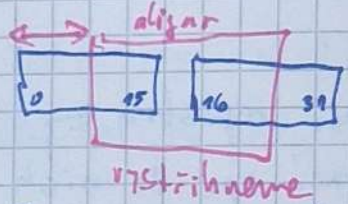
Inter-lane shuffles

unární: $r[i] = a[p[i]]$ (jiný reg.; konst. v instr.)

binární: $r[i] = (p[i] \& TOP\_BIT) ?\ b[p[i] \& LOW\_BITS] : a[p[i] \& LOW\_BITS]$

- permute, shuffle, alignr

alignr ↳ bit. posun směrem k nižším bitům → hodí se třeba pro čtení nezarovnaných hodnot, že je načteme zarovnaně a posuneme si je (vytrhneme si z nich tu užitečnou část)



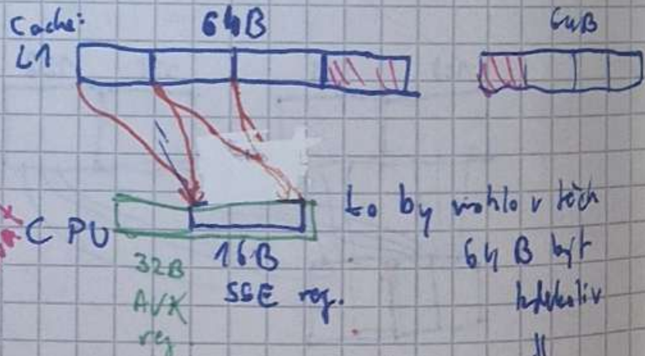
← právo je nalevo (je to LE)

- extract – vekt. reg. → skalár / menší vekt. reg.
- insert – opačný extract
- broadcast – rozkopíruje daný skalár do vektoru
  - varianty: • set1 • setzero • cast

Zarovnání paměť. operandů

SSE: nezarovnáno na 16B → segfault

AVX: nezarovnáno na 16B → pomalejší



to by mohlo v těch 64B být kdekoliv ⇓ bylo by na to potřeba hodně dalšího

tady to taky stačí na 16B ← ale když je to na konci tak je to víc loadů

↑ překladač udělá když použijeme speciální typy — __m256...

jsou tam jenom 4 a když je to nezarovnané tak se to ještě tak dá ALU

céčkový malloc zarovnání na 8 a do C++17 neumělo new předat informaci o tom typu té alokační mašinérii, takže tam to taky bylo na 8B

když reg, tak můžeme použít alignas(16)

a ještě když to tak není zarovnané tak se to v ALU musí doposunovat

alignas

- struct alignas(16) aligned { int32_t a[4]; }; ← vlastnost typu
- alignas(16) int32_t v1[8]; ← vlastnost proměnné

nejde to dát třeba do vectoru jako std::vector<aligned<int32_t>> !!

↳ tam se musí používat ty speciální typy __m256..

⇒ když pak chceme používat skalární, tak se musí použít reinterpret_cast (v.data() - to nám dá __m256i*

→ to dáme do reinterpret_cast<int32_t*>(...)

- čuňárna - na data nějakého typu se začneme dívat jako na data jiného typu
  - např. pro zapsání do binárního souboru - castneme na char
- může se tím porušit záruka toho, že když se přistupuje k různým typům, tak nemohou ležet na stejném místě:

```
double *p = ..
int    *q = ..
:
*p = ..
*q = ..
```

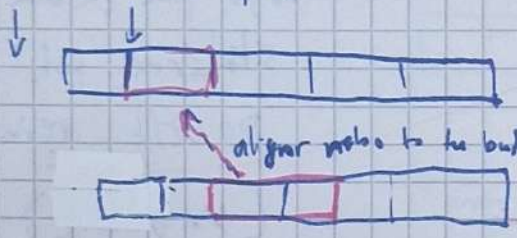
překladač může prohodit → ale reinterpret_castem to rozbijeme - může nám takhle prohazovat naše skalární a vect. instr.

- v C++23 to řeší std::launder - chová se jako fence - nedá se přes něj prohazovat !! naštěstí ale překladače to umí rozumí a funguje to

Zarovnání za runtimu

- pointer reinterpret castnout na `std::intptr_t` a vrátíme % 16

pro víc polí:



Vektorizace kompilátorem

- nejvnitřnější cykly s přehledným chováním

```
for (int i=0; i<N; ++i)
  a[i] = b[i] + c[i]
```

překladač dělá rovnou takle

loop unrolling - ekviv. úprava

```
for(i=0; i+3<N; i+=4){
  a[i]= b[i]+ c[i];
  a[i+1] = ...
  ⋮
  a[i+3] = ...
}
for(; i<N; ++i)
  a[i] = b[i] + c[i];
```

vektorizace

nemusela by být ekviv. úprava

```
① for(i=0; i+3<N; ++i){
  _mm_store_ps(a+i,
  _mm_add_ps(
    _mm_loadu_ps(b+i),
    _mm_loadu_ps(c+i)));
}
② for(; i<N; ++i)
  a[i] = b[i] + c[i];
```

kdyby se to a,b,c nějak překrývala ! ⇒ ale překladač musí zaručit ekvivalenci

- podmínky autom. vektorizace:
  - predikovatelná podmínka
    např. $i+3<N \Rightarrow i<N$, $i+M<N$ ...
    musí se vědět
  - počitatelné smyčky
    → # iterací lze za běhu určit podle té podmínky před vstupem do smyčky
    → např. tam vevnitř nesmí být nějaký if a ta řídicí proměnná se nesmí nijak měnit v tom těle
  - nesmí tam být závislosti mezi iteracemi:

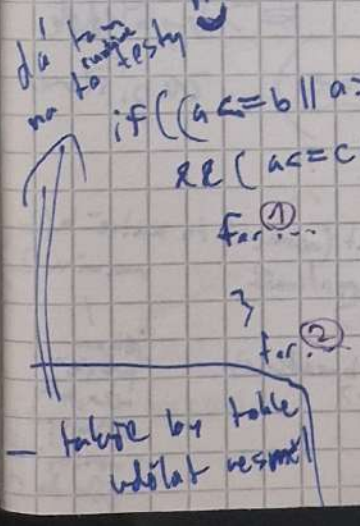
$s = s + a[i]$
- každá další iterace závisí na předchozím s

$a[i+1] = a[i] + 1$
- závislost přes elementy toho pole

$a = b + 1$
$a[i] = b[i] + c[i]$
- aliasing

- ale my chceme:
  - vyhledávat v poli a skončit, když najdeme
    → nepředvídatelná smyčka
  - akumulátor
    => závislosti mezi iteracemi smyčky
  - vygenerování výstupního pole
    → může tam být aliasing vstupního a výstupního pole

téměř všechny užitečné smyčky budou mít nějakou překážku



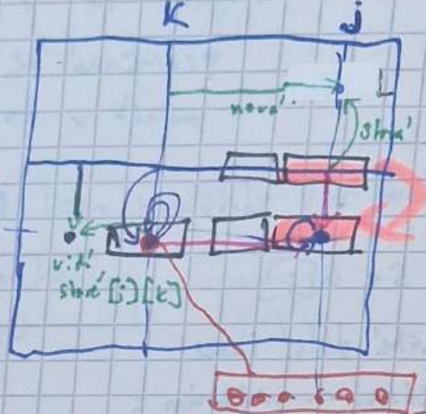
1.D.Ú: Floyd-Warshall

konstr.: alokace + zavolá clear()

velikosti matic budou dělitelné 64 ☺ - není potřeba řešit zarovnání

násobení matic: C[i][j] = add (C[i][j], a[i][k] * b[k][j]);
F-W: min, +c, +c

→ aliasing



lok. proměnná

i == k ⇒ C[k][j] = std::min (C[k][j], C[k][k] + C[k][j])
min !! ☺ 0, 0x7FFF ⇓ nepřičte

před vstupem do j-čkového cyklu se dá instr. broadcast na rozmnožení C[i][k]

→ to přičteme

⇒ minimum s tím C[k][j]

⇒ výsledek zapíšeme

1) zákl. řešení:
vektorizujeme jenom ten nejvnitřnější cyklus - ten j-čkový
⇒ i · j · (add + min + 2 · load + store)
zdržuje ☹

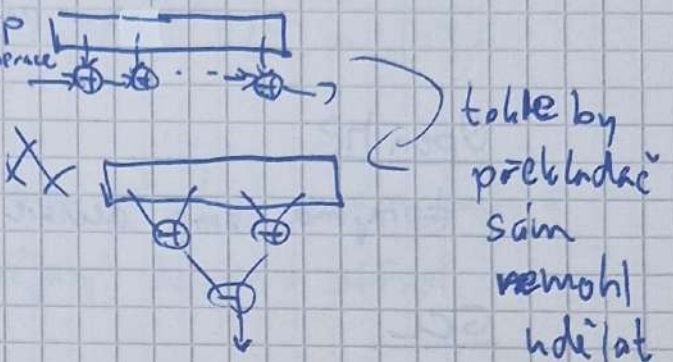
2) budeme najednou dělat několik řádek (4):
4|16|32 ? ⇒ experimentovat (normálně to může být maximum 2)
C[i][k] → broadcast
C[i+1][k] → broadcast
⋮
⇒ $\frac{1}{4}$ · j · (4 · add + 4 · min + 5 load + 4 · store)

jenom prostě těch dat se to vyplatí s mocninou dvojky ve skutečnosti do regs.

32 elementů
3^2 na AVX 512

C++ rozšíření pro optimalizace

- ekviv. úpravy, které by ale překladač zhodnotil jako nevýhodné
- neekviv. úpravy, které by překladač nesmí dle pravidel jazyka udělat
  - my tu správnost nenarušíme
  - narušíme



Inlining

⊕ ⇒ odstranění overheadu instrukcí call + ret
⇒ větší hřiště pro překladač
- ví, co se tam děje
- meziproceduální optimalizace

⊖ ⇒ delší překlad
⇒ větší velikost kódu

Intel

#pragma inline/forceinline/noinline ⇐ k volání

GCC

void f() __attribute__((always_inline/flatten/noinline)) ⇐ k funkci

always_inline: všechna volání se musí inlinovat (a když tak kompil. chyba)

flatten: všechna volání z této funkce chceme inlinovat (pokud to jde)

32 elementů
3 × 2 na AVX512

Vynucení vektorizace

Intel

```
#pragma vector always   <- před smyčkou
#pragma simd
```

↑ zvektorizuje jen, pokud je to bezpečné

Open MP

```
#pragma omp declare simd
```

GCC

```
void f() __attribute__((simd))

for(...){
a[i]=f(b[i]);
}
```

nešlo by normálně zvektorizovat

→ 1) inline 2) zvektorizovat NEBO se vyrobí další verze té funkce, která bude mít vektorové arg. a výst. hodn.

```
for(... i+=4){
  a[i...i+3]=f_4(b[i...i+3])
}
```

Ignorování potenciálních závislostí

Intel

```
#pragma ivdep
```

→ ignore vector dependencies

GCC

```
#pragma GCC ivdep
```

ale FUJ

překladače jsou různě líné

říkáme, že má ignorovat ty závislosti, o kterých ví, že jsou jistě

```
void f(int *a, int k, int c, int m){
#pragma ivdep
for(int i=0; i<m; i++)
  a[i] = a[i+k] * c;
}
```

ale tímhle garantujeme, že je to v pohodě

když k bylo záporné, tak to vadí

SolarisCC (Oracle)

`#pragma noalias (pointer, pointer [, pointer]...)`

`#pragma may_not_point_to (pointer, variable [, variable]...)`

restrict - C99

nebo někde `__restrict__`

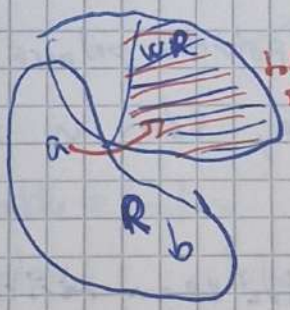
```
void copy(int * restrict a, const int *b, int n){
    for(int i=0; i<n; i++) a[i] = b[i];
}
```

všechny paměťové přístupy se dají vystopovat k přístupu na nějaký pointer + offset

když se pomtije restrict pointer na a, tak se zavazujeme, že ke všemu, na co přistupujeme přes ten pointer, se pak bude přistupovat už jenom přes ten pointer a

pro proměnné jsou šuplíky:

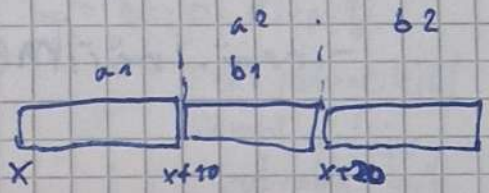
- v restrict proměnná má vlastní
- ostatní mají jeden společnou



přístupy jdou prohazovat, pokud jsou dotčené proměnné v různých šuplíkách

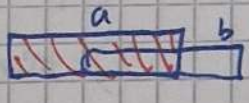
```
int m[100];
int * restrict x = m;
```

- copy (x, x+10, 10);
- copy (x+10, x+20, 10);



ALE:

- copy (x, x+10, 20); NE - oblasti se překrývají

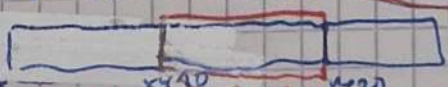


FUJ

```
{ int * restrict y = x;
  copy(y, x+10, 10);
  copy(y+10, x+20, 10);
```

povolili jsme interleave těch volání copy!

překladače



y copy zapisuje na různá místa v paměti => může přepsat cokoliv přes x a zapisujeme přes y

Ladění rychlosti programu

Intel VTune Profiler

Problematický kousek (hotspot) – místo, kde má cenu optimalizovat

Problémy:
- je těžké odhadnout to úsilí na tu optimalizaci
- je těžké odhadnout efekt té optimalizace

tedy: hotspot je místo, kde celkový čas vykonávání dělený velikostí toho kódu je největší

chceme zahrnout i ten čas procedur, které jsou odsud volány ?

Instrumentace (=> není potřeba podpora HW ani OS :-))

→ modifikace programu, aby sám u sebe něco změřil

↳ ta ale ten chod ovlivní => nemíra cenu měřit čas, ale počty průchodů nějakými důležitými místy

– např. měříme, kolikrát a odkud byla volána každá procedura

a z toho se pak odvodí i další místa

– optimalizace řízené profilem:
1) pustí se instrumentovaný program (na reprezentativních datech) — tohle je to těžký
2) ten výsledek profilování se předhodí dalšímu překladu a podle toho se optimalizuje

např. na těch hotspotech se pustí plná řešička nějakého NP

(a jinde jenom částečná)

úplného problému

Sampling

⇒ zvenku se koukáme na normální běh toho programu

↳ podle toho, jak někdo prásknu

čím:
- podle IP
- podle call stacku
- v náhodných okamžicích ⇒ nezáv. na běhu programu
- podle nějakých událostí (např. každých 1000 zápisů do paměti)
  - je potřeba HW podpora, která počítá ty události

- softwarové samplování
  - zmanipuluje se časovač, je potřeba podpora OS
    (OS používá jeden z hodin, které tikají jednou za 10 ms)
    - ! pozdější přerušení
    - řekneme, že to bude častěji
    - použijeme jiné hodiny
  - pozor, takhle se budou samplovat i ostatní procesy a kernel ⇒ bezp. díra!
    - pro pokročilejší samplování je potřeba být root

- hardwarové samplování
  - CPU udělá po daném počtu daných událostí přerušení
    - tikání hodin - takhle procesory :)
    - # instrukcí (ale pozor - CPU spouští spoustu instr., ale jen některé doběhnou do konce - počítat všechny, nebo jen dokončené? ↳ Intel)
    - přístupy do paměti
    - špatná predikce skoků

z toho jich jde trackovat jenom pár najednou

budeme si to nastavit prakticky

(a jinde jenom částečně)

úplného problému

Analýza závislostí

→ zjištění, že nějaká instrukce musí být puštěna až po jiné

(+ latence – minimální doba, která mezi nimi musí být)

→ část. usp. na instr.

– HW nějaké záv. může ignorovat: – např. když jsou nejisté

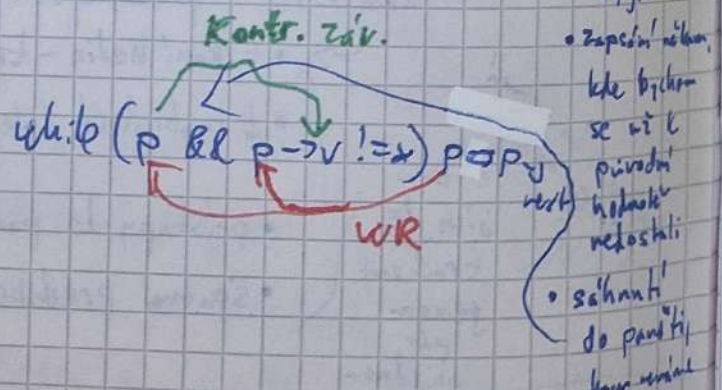
(→ a pak třeba zjistí, že nevěl → pustí znovu)

– závislosti:

- datové – Write-Read
  - typicky latence – read musí počkat, až to ta první spočítá a zapíše
- anti dependence
  - když bychom to prohodili, tak dostaneme nějaký jiný stav
  - Write-Read
  - Write-Write

Samotným instrukcím to nevadí, ale nám ano

- kontrolní
  - je tam instr., která může způsobit nějaký vážný problém
    - např. vyvolat výjimku
    - zapsání něčeho, kde bychom se už k původní hodnotě nedostali
    - sáhnutí do paměti, kam nemáme



2. D.Ú.

Bsearch

bsearch_inner – Read only dat. struktura ⟶ pole, ve kterém vyhledáváme, data už v konstr. dostaneme seřazená, my si to třeba musíme doplnit na velikost mocniny dvojky

bsearch_outer – Hvězdno vlastní, rev

konstr: osize = kolik těch hodnot budeme dělit zároveň vyhledávat (N)

↓

naalokujeme si buffery,

↓

mainicializujeme dat. struktr.

↓

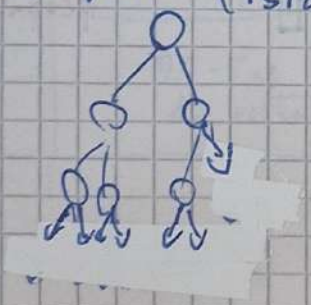
bucketize se pak bude několikrát volat s daty vel. osize

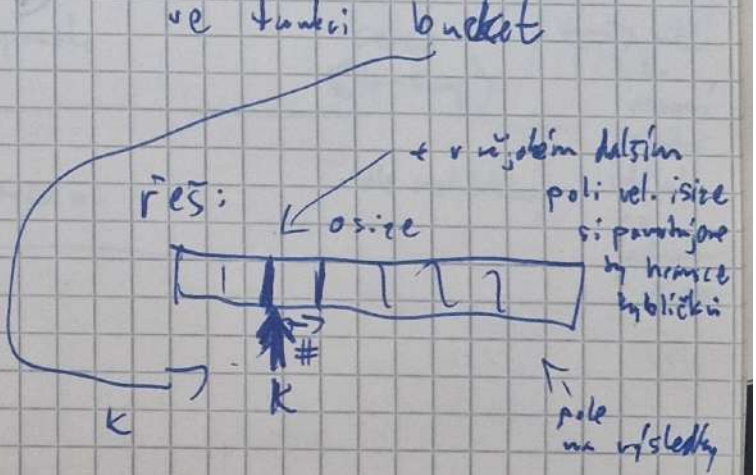
↓

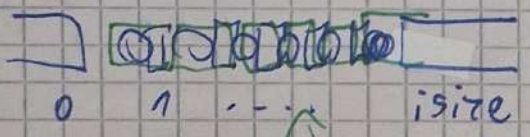
výsledek si schováme a vracíme ve funkci bucket

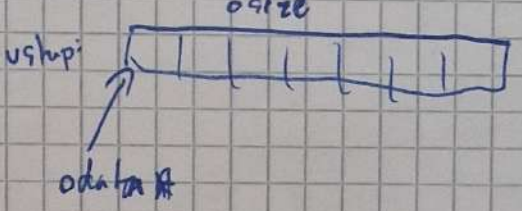
není měřeno

vst. data: (isize = 6)









chceme: čas osize * $\log_2$(isize)

trošičku vektorizovat

Optimalizace vnořených smyček

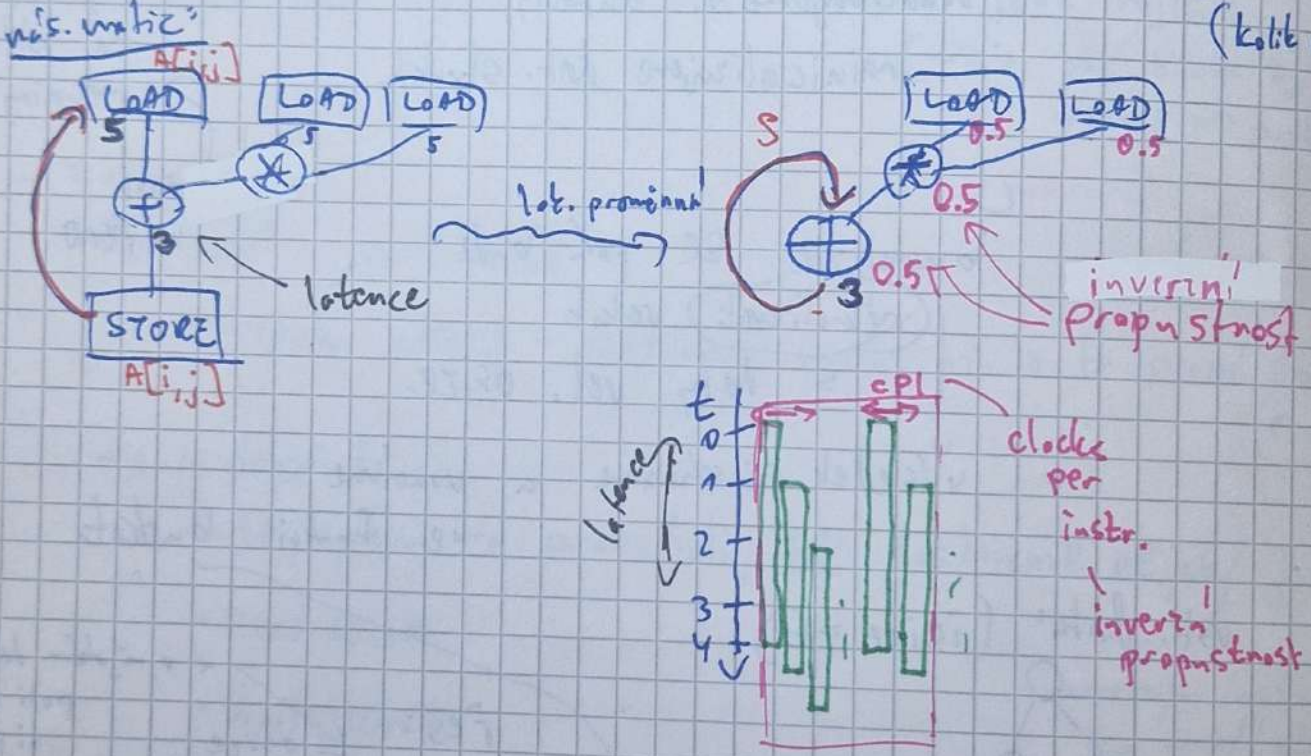
→ změna závislosti

→ využití algebry (např. že XOR je kom. a asoc.)

– polyhedrální kompilace – řeší se vztahy mezi datovým a iteračním prostorem

– brzdí nás: • latence operací

• # operací, které ten procesor může najednou dělat (kolik má výp. jednotek)



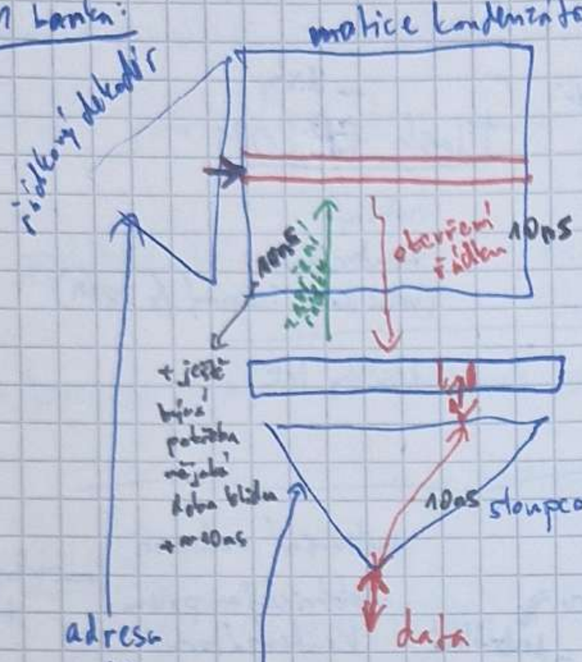
Paměti

	SRAM (– cache)	DRAM (– hlavní op. paměť)	Flash EPROM (– disky)
princip záznamu	el. obvod se 2 stab. stavy	nabitý kondenzátor	nabitý kondenzátor * (velmi dobře izolovaný X DRAM)
výdrž záznamu	po dobu napájení	desítky ms	desítky let
princip čtení	měření napětí na výstupu obvodu	vybití kondenzátoru a změření proudu ⇒ destruktivní	ovlivnění vodivosti elektrickým polem kondenzátoru → bezztrátové!!
princip zápisu	změna napětí vstupu obvodu	nabití kondenzátoru	nabití/vybití kondenzátoru (např. tunelováním) – takle je na tom to technicky náročné
tranzistorů na bit	6	1	1/3
výrobci	USA	Japonsko	Korea (Samsung) → nikdo to neumí pořádně udělat na jednom čipu
latence	0.5-5ns dle velikosti	10-20 ns	dle architektury

* MOS
metal
oxide
semiconductor

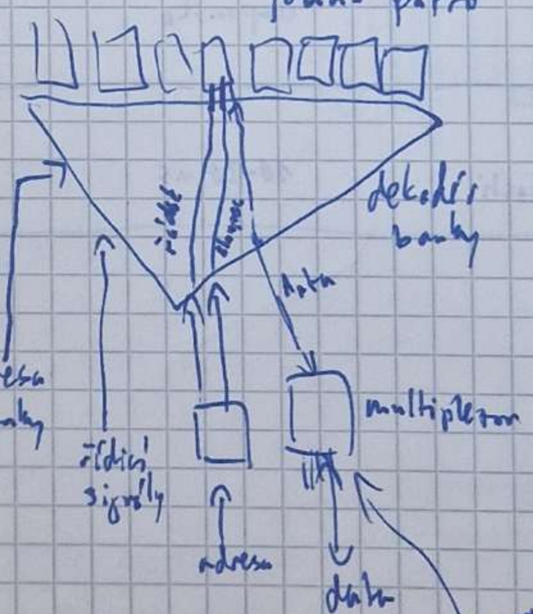
DRAM

1 banka:



signály:
RAS – read addr select
CAS – column addr select
} multiplexováno (CAS je potřeba až po RAS)

tohle celé ×~8
a nad tím je ještě jedno patro



rank ← a tohle s tím patrem nad tím

modul → 2 ranky

v DDR3 (8GB) jsou 2 moduly

kanály (nezávislé ⇒ větší propustnost)

v DDR5:
- modul přes dva kanály (= subkanály)
⇒ větší paralelismus

DDR4 – 2400
⇒ hodiny = 1,2 GHz

(např. 16) víc čipů se společnými řídícími signály
×8 → 64b dat

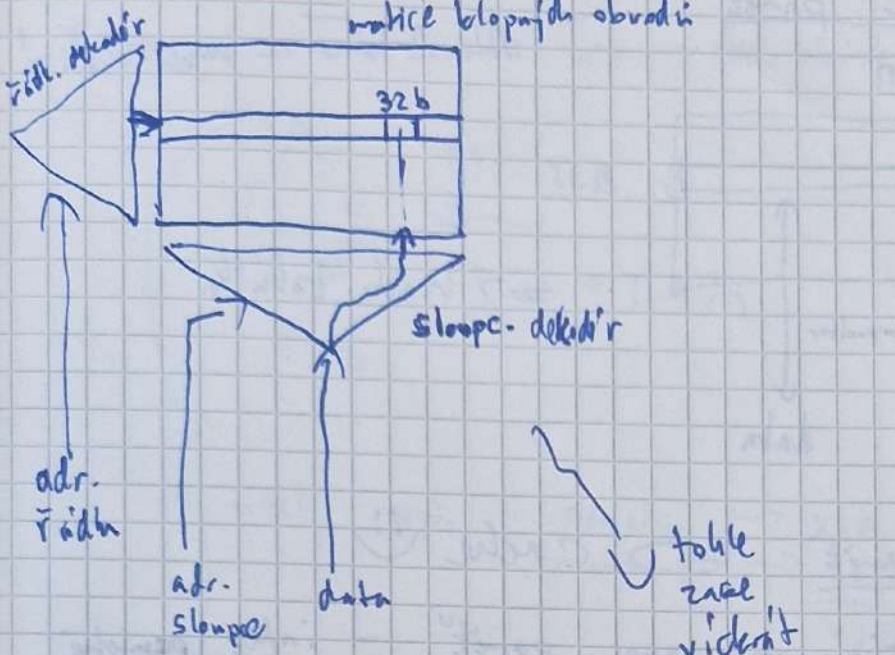
⇒ SDRAM
synchronní! ⇒ řízeno hodinami

např. dodáme adr. řádku
⇓
po 15 taktech musíme dodat adr. sloupce
⇓
po dalších 15 taktech dostaneme ta data

mohli se toho dělat víc najednou (1 banka zaměstnaná, tak můžeme říct jiné, ať dělá něco dalšího...)

např. 2/4/8
ale není nutné (skrýt se)
→ v několika cyklech v balících po 4/8 (v DDR5 po 16)
a lze to líp paralelizovat

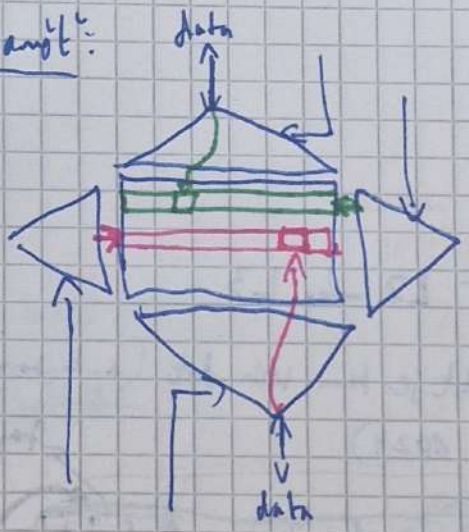
SRAM



Dvoubránová paměť:

(v TLB)

(a jiných malých pamětích)



=> paralelně dva řádky najednou

(jinýma než v CPU)

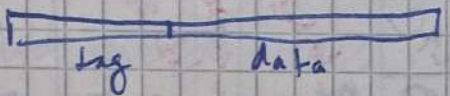
Plně

Asociativní paměť (např. TLB1)

nebo prediktory skoků

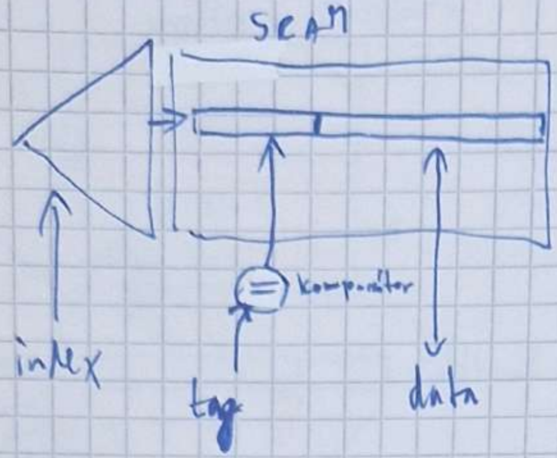
– nejvýše desítky řádků

= SRAM rozdělená tak, že nějaká pam jednotka (~ stavby sítě) bude doplněná komparátorem



→ zrovna zadáme tag, ty komparátory to paralelně porovnají, a ten, kde to pasuje, nám dá data

Přímo mapovaná asoc. paměť



— Ale FUJ — moc kolizí

=> hash. tabulka

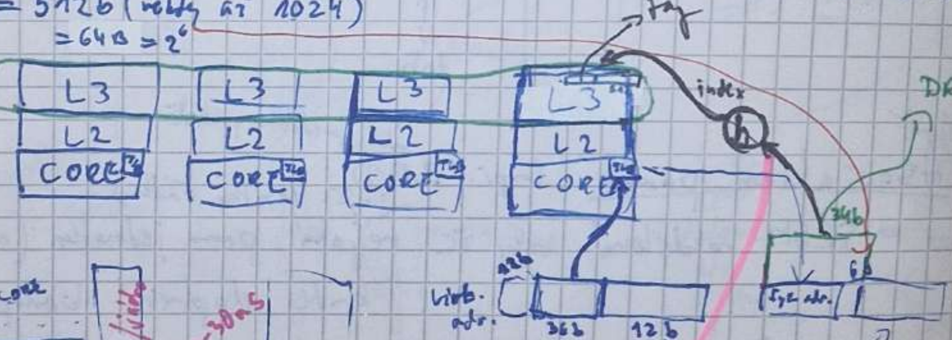
N-cestná asoc. paměť => cache ☺

=> n-krát přímo map. asoc. paměť — index paralelně do všech a ta, u které sedí tag, se použije

(často index je nějakým hashem tagu)

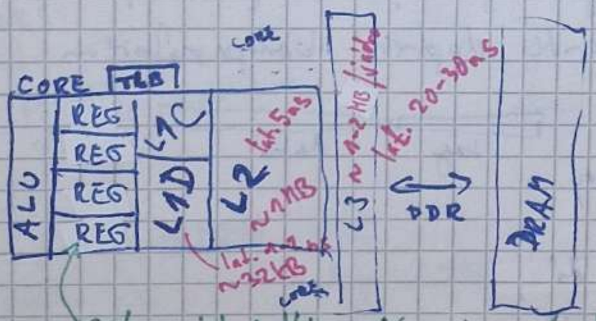
LLC (last level cache) — L2 nebo L3

- 1–2 MB na jádro
- cache line = 512b (někdy až 1024) → kolik je tam těch dat (a k tomu je ještě ta adresa)
  = 64B = 2^6



ale dělá se to takhle:

— jenom se z toho kousek ukousne => index (ten vybere nějaký řádek)
=> porovná se tag
=> tím máme ten řádek ☺

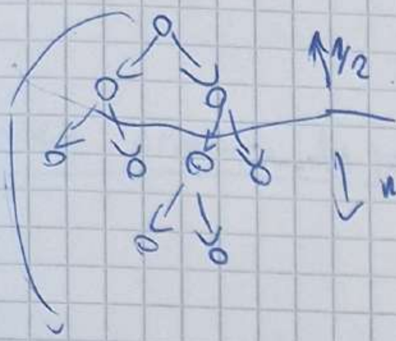


sady registrů ∀ hyperthreadingové vlákno

- inkluzivita = když je to v jedné úrovni cache, tak je to i v těch nižších
  — dnes L1 inkl. s L2, ale L2 ne inkl. s L3

Stromy v cachích

• TLB má ~1000 záznamů => ~1000 stránek efektivně



1 uzel např. std::string -> 24B

2x unique_ptr -> 2×8B

40B < 64B řádka

data toho stringu – zase někde jinde

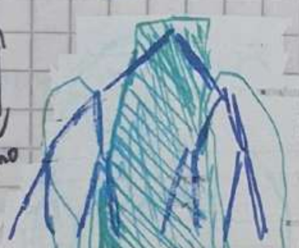
+ to nemusí být zarovnané na řádky => takže to možná může být i přes dva řádky => 128 čtení

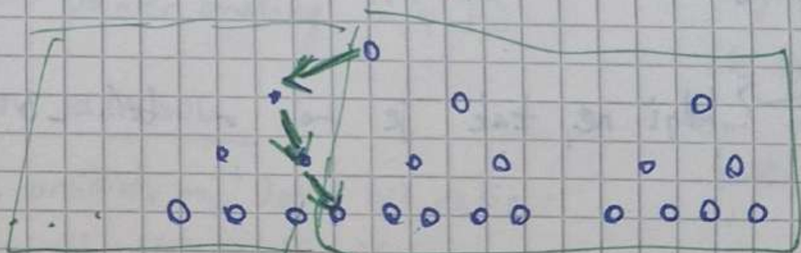
64kB L1 = 1024 řádek

32-bit čísla (BVS) v poli: 64B = 16·4B, B = velikost bloku

každý uzel zabírá 2-4 řádky !! :(

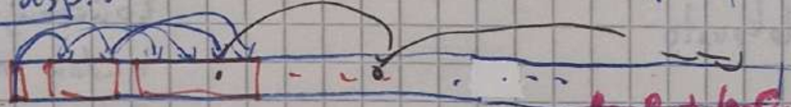
#řádků cache C, $\log_2 C$ trvale nacachováno



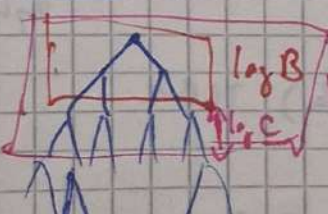


$\log_2 B$ => Cache nám pomáhají až takhle v těch hlubokých patrech dole

jiné usp.:



log B + log C



log(B*C) trvale v cachi.

=> výhodnější pro paralelní procházení

(…řádku)

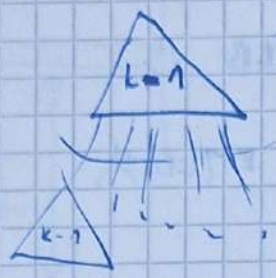
…ten řádek :)

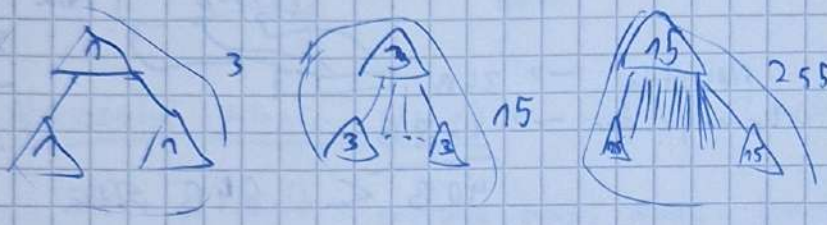
van Emde Boasovy stromy

k přepážek v prostoru klíčů

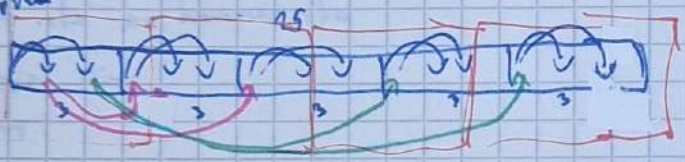
strom o n uzlech = (k+1) stromů po (k-1) uzlech

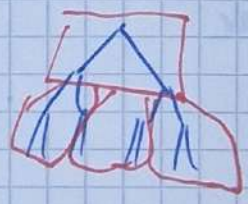
$(k^2 - 1 = n)$
$\Rightarrow k = \sqrt{n+1}$





pro 15 prvků:





blízko ve stromě
⇒ blízko v paměti
:)

pokr. cachí

když jsou přistupují na disj. adresy, tak vše, co je v L2, je i v L3 (a co není v L2 není ani v L3)

↳ když ne, tak je to v tý L3 jen jednou :)
⇒ zbyde tam místo na další data, která se do L2 nevejdou
(ale zase zápisy na data, co jsou ve více L2, způsobují, že si je budou hodně navzájem vyhazovat :))

! ty přístupy do cachí jsou pipelinované – ty latence nezmenšují, že by se to po tu dobu zablokovalo

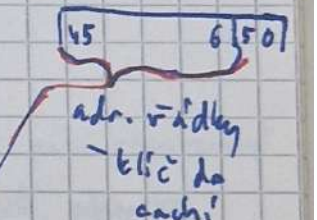
TLB

- 2 úrovně: – ~64 záznamů (pro 4 kB stránky) TLB1
  ⇒ 256 kB adr. prost.
  – ~1024 záznamů TLB2, lat. 2–3 ns
  – společné pro hyperthread. vlákna

není-li záznam v TLB:
- page walk (2-4 přístupy do paměti) ↑ přes cache!
  } procházení stránk. tabulek
- page Fault → OS

proces přístupu do paměti: `mov zmm1, [rbx+8*rax+1000]`

1) spočte se ten výraz (64 bit)
↓
virt. adresa (48 bit)
↓ 36 bitů (horních) ... zbylých bitů
2) TLB1 — 36 bitů → fyz. adr. (46 bitů)
↓
TLB2
↓ page walk



adr. řádku – tlíč do cache

z toho se vše pentíje jako tag a dolních 6 bitů jako index

ale ten ani nebyl zvenčím tím překladem, takže se to udělalo už předtím

pentíje se to jako index do L1D ⇒ z toho je tag a ten se porovná s tímhle tagem

paralelně s tím překladem

Algoritmy a cache

přístupy do paměti:
- s předvídatelnou adresou
  - lin. průchody polem – procesor pozná a dělá „prefetching" – také někdy prefetching (+překladač pomáhá)
  - lin. průchody s větším skokem (matice) –
- s náhodnou adresou
  - hash. tabulky ⇒ velká latence
    - dá se pomoct paralelismem
  - bucket sort
  - bin. vyhl.
  - spojové struktury ⇒ pointer chasing !! FUJ

Přístupy s předvídatelnou adresou

- cache line ⇒ husté lin. průchody mají dobré hit ratio
- write buffers ⇒ zápisy obvykle nezdržují
- HW a SW prefetching
- latence se skryje paralelním vykonáváním jiné užitečné činnosti

Cache - oblivious algoritmy

cache - aware = alg. vyladěn pro konkr. parametry cache

cache-oblivious = víme, že cache ∃, ale neznáme její parametry

↓

nepočítáme aritm. op., ale přesuny mezi hl. pamětí a cachí (# cache missů)

→ n ... |vstup|, C ... |cache| ⇒ $O(f(n,C))$

cache missů se dá vyjádřit jako násobek aritm. slož. alg.:

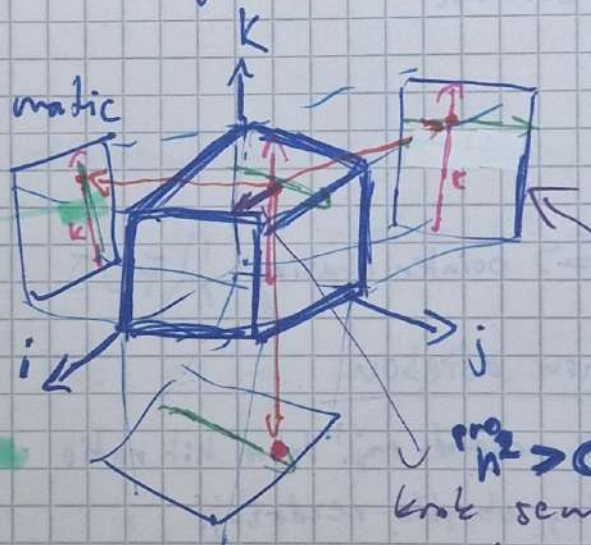
$$O(f(n,C)) = O(t(n) * g(n,C))$$

↓ jak často dochází ke cache missům

navíc pro velká n často $O(g(n,C)) = O(g(C))$

u dobrých alg. s rost. C klesá $g(C)$ k nule

Př.: nás. matic



pro $n^2 > C$

krok sem znamená celé nové tohle

→ vyhazujeme si data, která budeme potřebovat

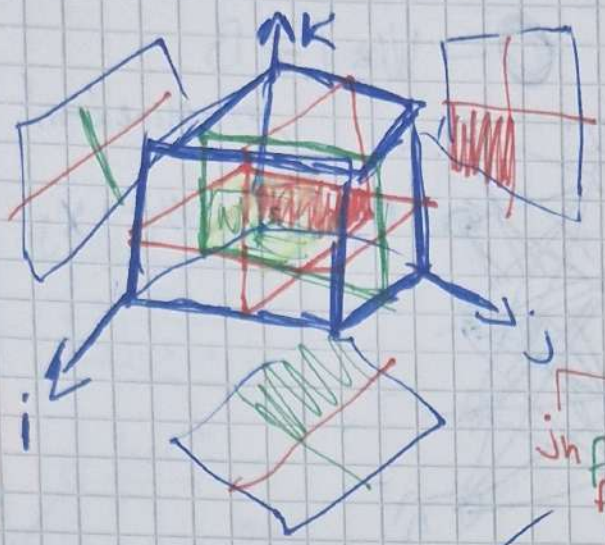
⇒ cache vůbec nepomáhá !!

data vel. n^2, n-krát je tam sčítáme ⇒ n^3

celou matici budem z hl. paměti stěhovat n krát

$t = O(n^3) \Rightarrow g(C) = 1$!!

Lépe: matice zvenšíme



$j = j_h \cdot \frac{n}{2} + j_l$

```
for i_H in (0, m)        → 1 . i . (n/m)²
  for j_H in (0, m)
    for k_H in (0, m)
      for i_L in (0, n/m)
        for j_L in (0, n/m)
          for k_L in (0, n/m)
```

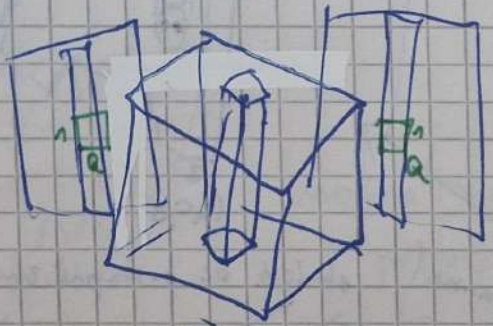
typicky se dělí $m = 2$ (a pak rekurzivně tak, aby se to tam vešlo)

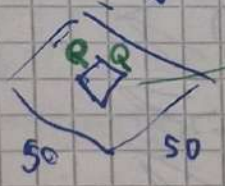
$O(C^{-1/2} * n^3) \rightarrow g(C) = C^{-1/2}$

když $n' < C \cdot \frac{1}{3}$ (L1 cache)

vyjde to asi: $n' \approx 50$

na nejnižší úrovni:





$\rightarrow Q^2 + 2Q$ celkem do registrů

(např. AVX512: 32.16)

$\rightarrow Q \approx 20$

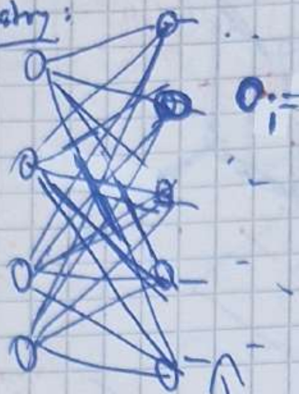
$\frac{L1 >> R}{ALU} = \frac{2}{Q} \rightarrow \frac{1}{10}$ → 1 přesun na 10 operací

DCNN

$$O = W * I + B$$

ReLU

Fully connected vrstvy:

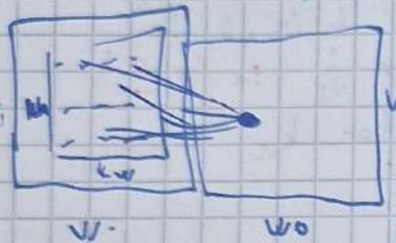


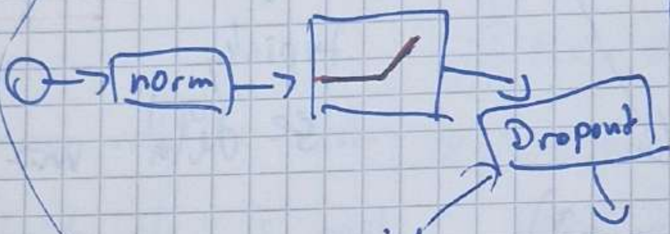
→ nelineární

$$o_i = f\left(\sum w_{ij} \cdot x_j + b_i\right)$$

tohle se učí!

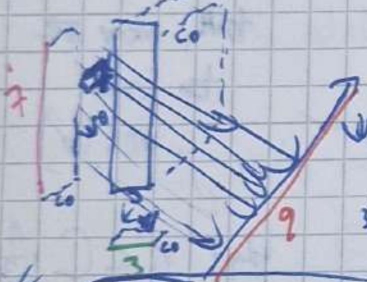
ještě normalizace, aby ty hodnoty byly kolem 0 (stř. hodn. 0, rozptyl 1)





jednou za čas se náhodně data vynuluj! (v době učení)

$w_i = w_o + k_w$

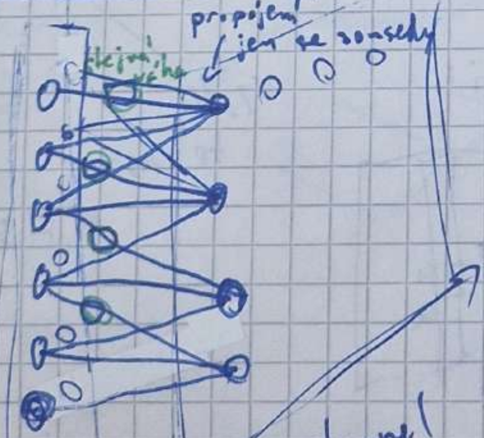


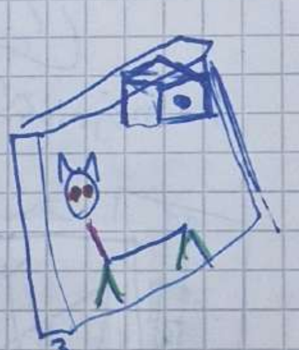
$3 + 7 + 9 = 19$ paměťových přístupů ($10N + 9$)

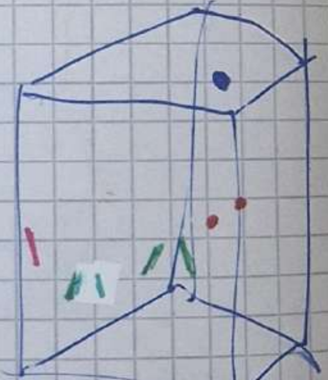
$3 \cdot 7 \cdot N = 21N$ aritm. op. !!

Konvoluční vrstvy:

jedno z těch rozměrů







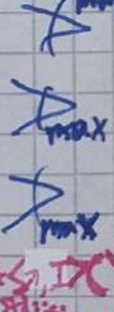
channel

channel

obrázek se postupně zmenšuje:

1)

2)




```
static_for(make_index_sequence<N>{}, [](std::size_t<I>) (auto I) { ... });

template<std::size_t ...I, typename F>
void static_for(std::index_sequence<I...>, F&& f) {
    (f(I), ...);
}
```

lépe: `(f(std::integral_constant<std::size_t, I>()), ...);`

void forward – 1x pro celý ~~balík~~ (16) obrázků ∈ [0, 15]

=> o 1 rozměr matic (vnější for cyklus)

volá 6 dalších

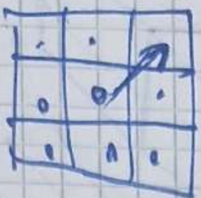
↑

1. je to nejsložitější → optimalizace

Zbytek jednodušší

↳ zkombinovat do jednoho (a zvektorizovat)

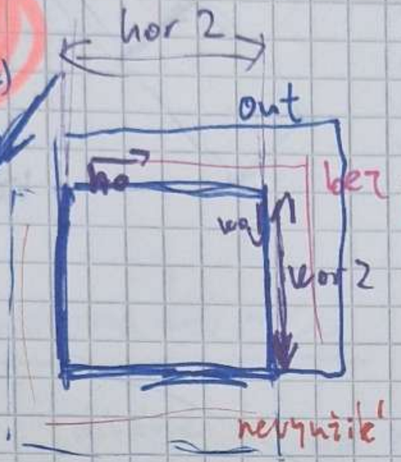
↳ tohle je další optimalizace



(3 x 3) kernel

↑

khr, kwr ∈ [0, 2]



hor 2, wor 2 – výstupní range

inv, wtr, outr – tenzory (váhy)

↓

4-rozměrný

⇓

outb = outr[b]

↳ 3 rozměry

(šikovné rozdělování na bloky) => r

JAK

Změna zanořenosti cyklů!

jsou tam pomocné funkce (viz dokumentace)

až to bude mít 1 rozměr, tak na tom jde udělat .data() => vekt. instr. !!

(možná upravit weights v permutation-policy

h w c ~ c)

+

vektorizace

tagged :: co : na typech x -> x′, x′ -> x

↑

+ operator ~ na hodnotách

možná nejvnitřnější cyklus přepsat na hezký bez těch typovaných indexů, aby to překladač autovektorizoval (nebo vektorizovat ručně – možná bude potřeba parciální specializace)

konv. vrstva:

↓

vekt. instr.

hor_batch = 7

khr_size = 3

↑

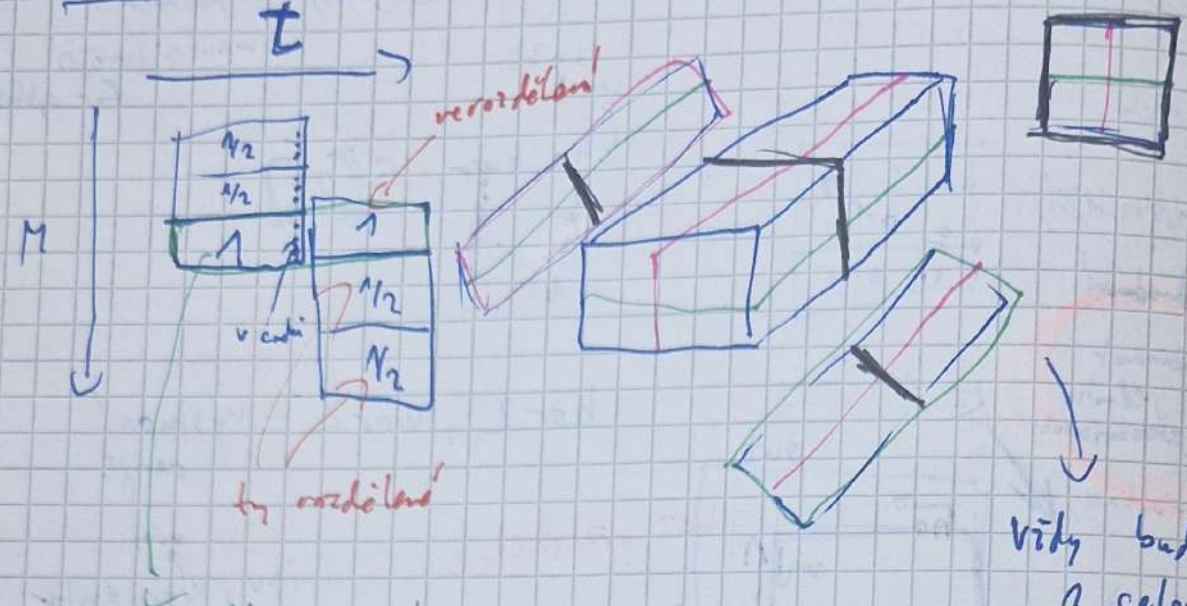
static_loop

(v: v_m ← vektor)

perfectly_strided<LANES>(car) > pomocí na održování

car / LANES

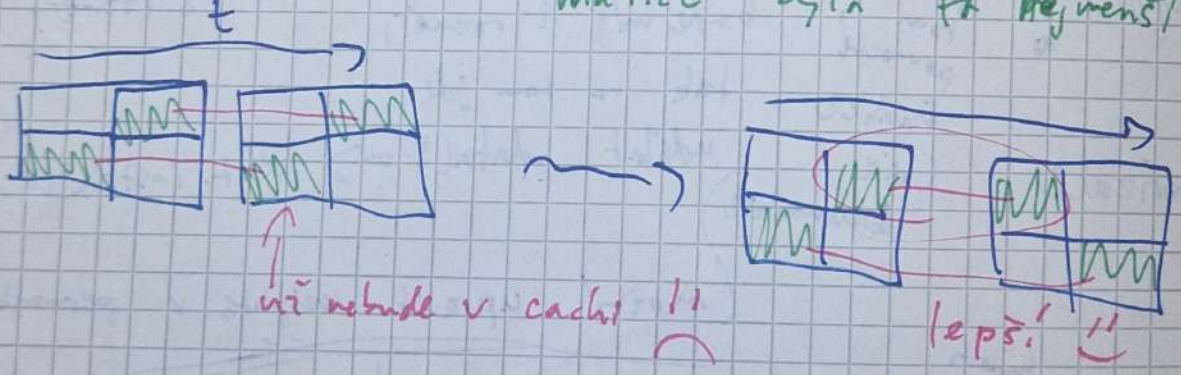
Ideální alg. – násobení matic



tenhle průnik chceme minimalizovat

Vždy budeme mít 1 celou matici a zbylé dvě rozdělené na poloviny

takže dělit tak, aby ta nedělená matice byla ta nejmenší



↳ hledáme vhodnou křivku vyplňující iterační prostor

Z-křivka
- z pořadového čísla skládáním bitů (ale blbě se to programuje)

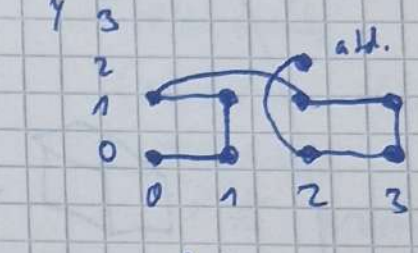
Hilbertova křivka
- výhy na to ani nepotřebujeme
- sousední pole – my chceme naopak, aby sousedé se lišili v co nejvíce bitech
- ale komplikované je generování !!

Zjednodušení Hilbertovy k.
- Gray kód
- v jedné dimenzi $z = i \oplus (i \gg 1)$

y	x
0	0 0
0	0 1
0	1 0
0	1 1
1	0 0
1	0 1
1	1 0
1	1 1

← z-křivka

y	x	
0	0 0	Gray
0	0 1	←
0	1 1	
0	1 0	
1	1 0	
1	1 1	
1	0 1	
1	0 0	



skoky jen v jedné dimenzi (1)

Cache matematicky

<u>předp.</u> 1 vlákno, díváme se na nějaký podproblém, ~~cache dělá~~ perfektní LRU

zajímá nás # cache missů,

T := doba běhu toho podproblému

C := velikost cache

A := množina všech dotčených adres, adres na které jsme tento podproblém přistoupili v čas. intervalu (t_1, t_2)

$m(t_1, t_2)$:= # různých adres, ↑ splňuje Δ-nerovnost

$m(w)$:= průměr $m(t, t+w)$ přes $\forall t \in [0, T)$

když vybereme t_1, t_2 a $m(t_1, t_2) \geq C$

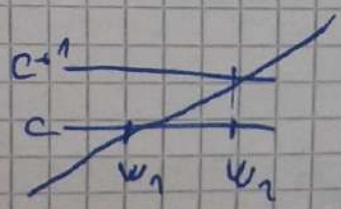
$\Rightarrow$ cache miss

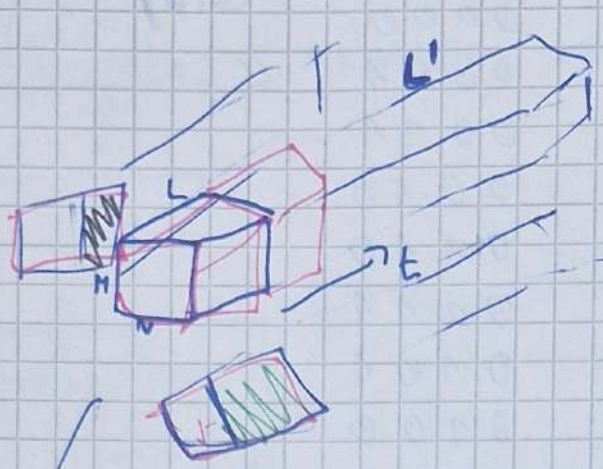
~~$= 8 \cdot 8 \cdot 8 = 512$~~

~~$|A| = 3 \cdot 8 \cdot 8 = 192$~~

$$\rightarrow \chi(C) = \frac{\partial m(w)}{\partial w}\left(m^{-1}(C)\right)$$

↑ frekvence cache missů





Pořád stejná
⇒ chceme, aby zůstala
v cachi

Zařídíme, aby se nám
vešly 2 tyč sousední
krychličky

$= M \cdot N \cdot L$

$+ = M \cdot L + N \cdot L' (- M \cdot N)$

Frekvence cache misses $= \frac{1}{M} + \frac{1}{N} + \frac{1}{L'}$

co nejmenší - ideálně 1

$2 \cdot M \cdot L + 2 \cdot N \cdot L + M \cdot N \leq C$

minimalizujeme

podobně

$M = N, \quad M \approx \sqrt{C}$

velikost L_0 - registrů

a podobně pak
pro vyšší úrovně
cache - aby se nám vešly 2 tyhle celé krychly

$M \cdot L' + 2 \cdot N \cdot L' + 2 \cdot M \cdot N \leq C_1$

velikost L_1

CPU

Tranzistory

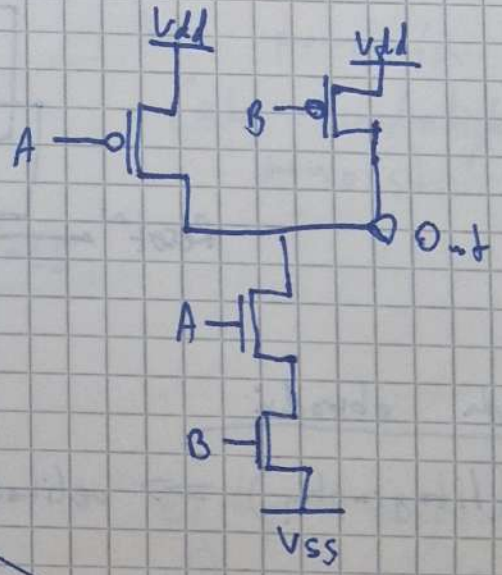
- CMOS ⇒ polem řízený tranzistor
  - Complementary Metal-Oxide-Semiconductor
    - jsou tam obě polarity (n-typ + p-typ)
  - chová se jako elektronky (napětí na řídicí elektrodě řídí průchodnost toho tranzistoru)

vrstvy:

- Paměti používají odlišné technologie

Hradla

NAND – ¬(A&&B)

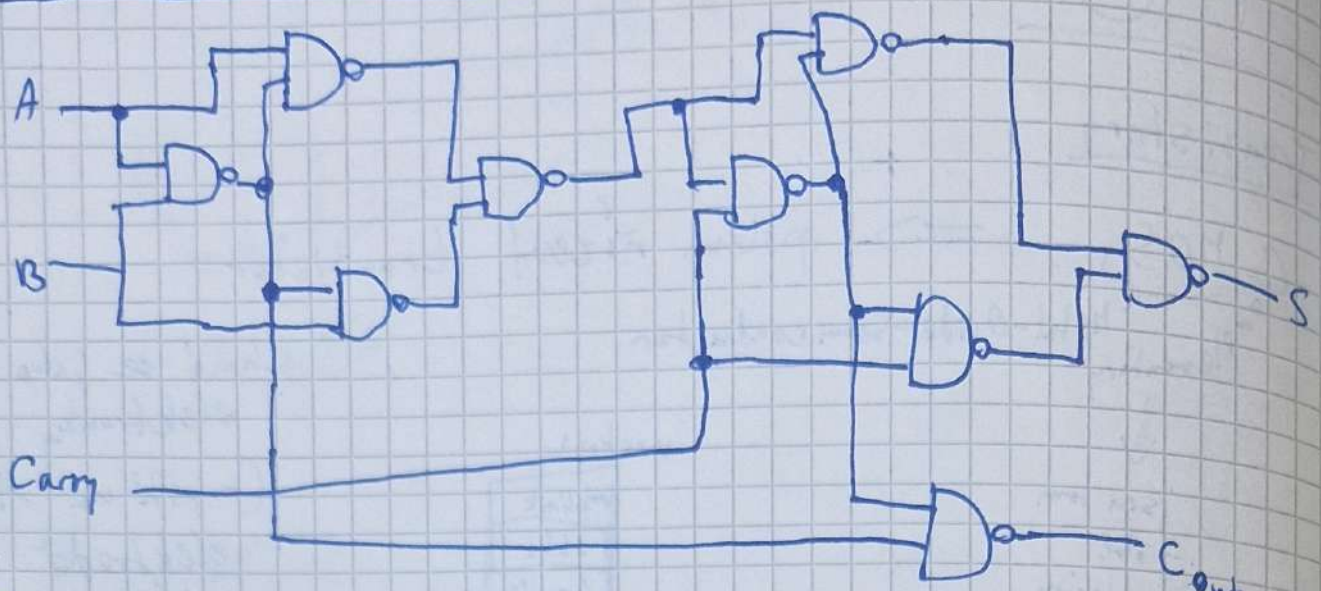


Spotřeba energie:

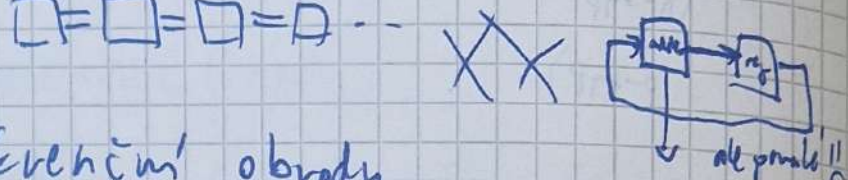
- ty dráty se chovají jako malinké kondenzátory
  - ⇒ chvilku trvá, než se vybijí
    - ⇒ spotřebování zbytečné energie (nJ)
  - platí Demardovo škálování – čím menší, tím menší kap. kondenzátory ⇒ klesá spotřebovávaná energie
- když se zavřou jedny ty tranzistory, tak chvilka trvá, než se zavřou i ty další ⇒ mini zkrat
- ty tranzistory nejsou nikdy zavřené úplně → něco tam poteče trochu teče ⇒ spotřeba energie

tohle je asi největší problém

1-bitová úplná sčítačka

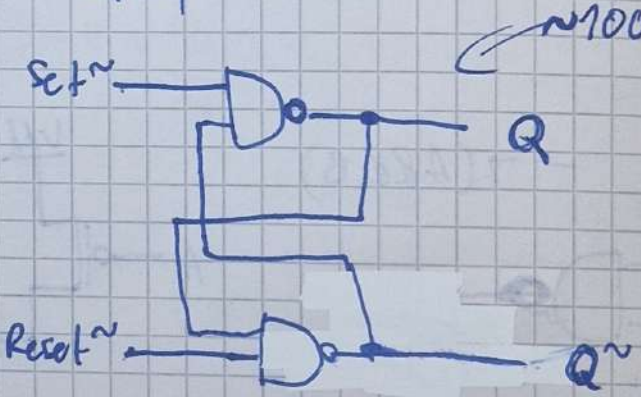


N-bitová sčítačka



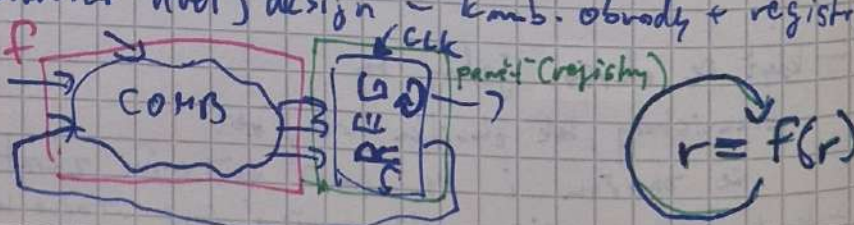
Kombinační ~~x~~ sekvenční obvody

- acyklický graf
- cykly



Design těch obvodů

1) process (litography) => velikosti součástek, množství proudu, ... => frekvence, ...
2) physical design => masky
3) transistor level => sítě tranzistorů
4) logic gate level => sítě hradel
5a) RTL (register transfer level) design - komb. obvody + registry



5b) System-level design – SystemC, Verilog, VHDL

→ vlastně takové prog. jazyky – programy takové, že umožňují sekvenční vykonání ⇒ to se převede na RTL

ty nástroje vyvíjí třeba firma Synopsis

Násobička

- školní alg.

1-bit. násobička – jenom AND hradlo ☺

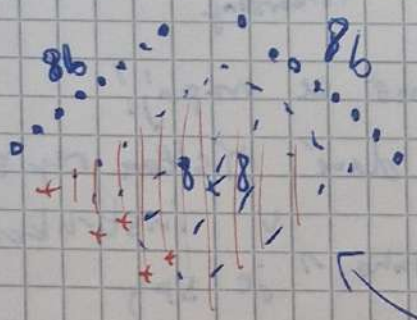
⇒ těžké je to sčítání výsledků

$O(N^2)$ zpoždění ☹

s pomocnými registry

⇒ $O(N)$ když to uděláme sekvenčně bez pomocných registrů !! (ale kvadraticky množství sčítaček)

Wallacova násobička – dnes



– 64 * AND

schovat ty věci pod obon posčítání

triky s uspořádáním

⇒ postupně se redukuje #bitů (ze 3 bitů se udělají 2)

pipelining ☺

$O(\log_{3/2} N \cdot N)$ vrstev

(napříč pro 64b je to 21 vrstev)

Obecně

- sekvenční impl. => min. tranzistorů, velká latence (mnoho děl.)
- čistě kombinační impl. => hodně tranzistorů, malá latence
- kombinační impl. rozdělení na fáze (stage) => pipelining => větší propustnost

paralelizace:
- pipeliny
- SIMD
- multithreading

} nezlepšují latenci, ale zvyšují propustnost

Out-of-order execution

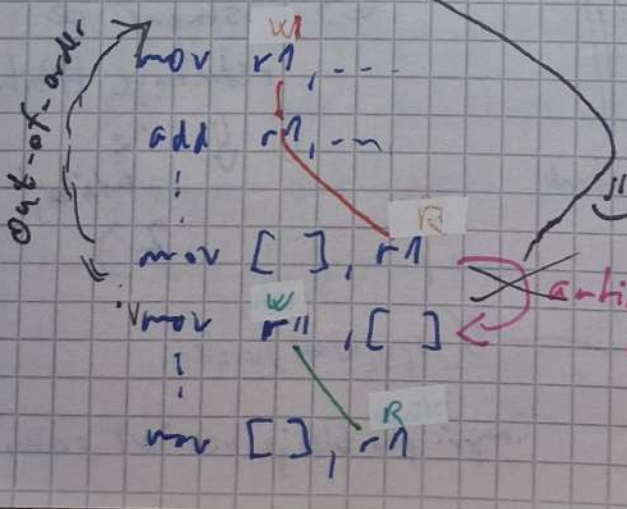
instr. => mikrooperace ~> out-of-order fronty ~> výk. jednotky

- zápisy do paměti in-order -> retirement jednotka
- loady se dějí i z fáze retirement = store forwarding

Renamer

fyzických registrů je ~160, na ně se mapují architekturální registry (ty dané číslem registru v instrukci)

~> 16 (z toho 1 je SP) celatis.



-> může se vykonávat out-of-order

Hyperthreading

- výk. jednotky obsluhují více vlákem (=> při branch miss prediction se nevyhazuje celá pipeline, ale jenom ty instr. daného vlákna :))

Branch prediction

- na podmíněné skoky nutné, ale používá se i na nepodmíněné

- ret ve front-endu se emuluje HW zásobník na návratové adresy z returnů :)