1 Úvod

Poznámka
Probírali jsme C (typy, literály, operátory, konverze,)

Pozor
Proměnné mohou být uložené s paddingem (když vyžadují zarovnání (`_Alignof(typ)`), např. ve struktuře s char a int se mezi char a int vloží 3 byty). Také nejsou všechny kombinace funkční (např. u float), tedy není dobré jen tak přistupovat na náhodná místa.

Padding není v poli! A na začátku (a pouze tam) struktur.

Navíc ve struktuře je zaručeno pořadí, takže char int char zabere 12 bytů (na konec se přidává padding tak, aby zarovnání fungovalo i v poli takových struktur).

Pozor
„0číslo“ je číslo v oktanové (osmičkové) soustavě.

V 16 soustavě se používá pro exponent znak `p`, ne `e`.

Dva řetězce s mezerou uprostřed se spojí.

Pozor
Na floatech rovná se moc nefunguje. Např. NAN (not a number) není roven ničemu, ani sám sobě.

Pozor
Menší typy než int se při výpočtu a předávání do funkcí bez definovaných parametrů převádějí na int.

Pozor
Za návěštím musí být příkaz (klidně prázdný), nemůže tam být deklarace ani konec bloku.

Pozor
Switch funguje interně jako goto, tedy tam musíme použít break. (Pokud chceme propadnout dále, píše se `\\fall through`).

Poznámka
Switch může skákat i do bloků (tj. např. můžeme zahájit první iteraci cyklu podle proměnné někde uprostřed).

Definice 1.1

Existuje `_Static_assert(podmínka, zpráva)`, které umí ověřit něco při kompilaci (a v případě neúspěchu vrátit zprávu), např. `sizeof(int)==4`.

Poznámka
Existují s proměnným počtem příkazů, na to se hodí `<stdarg.h>` a `va_arg`. Ale nepředává žádná metadata! (Na druhou stranu, zde se předávají pouze věci větší než / stejně velké jako int.)

Definice 1.2 (Fáze překladu)

1. Konverze do kompilační znakové sady a nahrazení trigrafů (náhrad za chybějící znaky, tvaru `??něco`).
2. Slepení řádků končících na \.
3. Rozdělení na preprocesorové tokeny (aby pp mohl ignorovat sémantiku C) a mezery, komentáře -> mezery.
4. Preprocesor (direktivy, makra, includy).
5. Konverze do cílové znakové sady (kdybychom kompilovali na jiný počítač).
6. Slepení navazujících stringových literálů.
7. Konverze pp-tokenů na tokeny, zapomenutí mezer.
8. Syntaktická a sémantická analýza.
9. Linker.

Definice 1.3 (Preprocesorové direktivy)

`#` musí být na začátku řádku, za ním mezery a pak direktiva až do konce řádku.

Umí `#if, #else, #endif, #elif`, ale nepřetypovává. Naopak umí ještě `defined(ident), defined` speciálně `#efdef ident, #ifndef ident`. Všude jinde se nahrazují makra, nedefinované identifikátory (i klíčová slova) jsou 0.

Dále umí `define ident[(arg)] expanze` a `undef`. (Pozor na mezery před (arg)).

K tomu ještě umí `#include <jméno>` nebo `#include "jméno"` makra. (Systémové nebo naše knihovny.) Uvnitř `#include` se expandují makra (ale neslučují se stringy). Proti vícenásobnému inkludování se lze ochránit tak, že soubor uzavřeme do `#ifndef` s nějakým unikátním makrem, které si nadefinujeme.

`#warning, #error` dělá obvyklé věci, `#pragma, _Pragma` si naopak volí sám kompilátor, co má dělat (`_Pragma` nemusí být na začátku řádku).

Definice 1.4 (Nahrazování maker)

Zjistíme, zda se máme vůbec nahrazovat, posbíráme argumenty (a rekurzivně expandujeme ty), volání makra nahradíme jeho definicí s argumenty (speciálně `#cosi` vrací `"cosi"`, `##` slepuje makra), ve výsledné posloupnosti hledáme další makra (případně blokujeme rekurzi).

Pozor
Důležité je závorkovat.

2 Procesory

2.1 i386 = x86

Definice 2.1 (Datové typy)

Integer 8/16/32 a někdy 64 bitů. Float 32 (16+8 float)/64 (53+11 double)/80 (16+64 long double) a někdy 16 (11+5, pouze ve vektorech). Vektory 128 bitů, musí být homogenní a obsahuje libovolný předchozí typ (krom long double).

Definice 2.2 (Paměť (z pohledu uživatele))

Pole bytů. Integery se ukládají little-endian, float mají také nějaké uspořádání, vektory jsou v běžném pořadí.

Definice 2.3 (Registry)

EAX (32b, obsahuje navíc AX = 16bitů a ten obsahuje AL a AH = 1byte), EBX, ECX, EDX.

ESI a EDI (source a destination index, dnes už umí skoro všechno, avšak neexistuje SIL apod.).

ESP (stack pointer, stack roste směrem dolů, tedy stack pointer ukazuje na nejnižší adresu, uložení na zásobník je: snížení ESP a uložení dat na ESP), EBP (base pointer, hraje (± zaniklá konvence) roli při předávání argumentů, dnes již také běžný registr).

EIP (instruction pointer, ukazuje na další instrukci programu). EFLAGS (stavový registr, jednotlivé bity mají vlastní význam: nula, záporný, přetečení, směr čtení z paměti).

ST0-ST7 (teoreticky) (floatový stack), FPUSR (floating point unit status registr, stavy jako v EFLAGS), FPUCR (FPU control registr, řídící bity).

XMM0-7 (128bitové vektorové registry), dnes již často YMM0-7 (256b), MXCSR (control a status registr).

Definice 2.4 (Asembler)

Existují 2 různé syntaxe (pozor, mají opačně operandy (tj. kam se ukládá výsledek)) Intel (spíš Windows, `ADD EBX,1`) a AT&T (UNIX, `addl $1,%ebx`, kde `$` značí literál a `\%` registr, `l` (32) obdobně jako q (64b), w (16b), b (8b) značí, s čím pracuje instrukce). Budeme používat AT&T syntaxi.

Operandy `$číslo` = literál, `%registr`, `číslo` adresa v paměti, `(%registr)` adresace registrem, `číslo(%registr)` = adresa je registr + číslo (registr nemění), `číslo(%r1, %r2, mul)` = adresa je r1 + (r2 krát mul) + číslo (např pro pole, mul může být 1, 2, 4, 8), implicitní operant (nepíše se, instrukce má automatický operand), `.Návěští`.

Definice 2.5 (Konvence předávání argumentů funkce a výsledků)

Tzv. ABI (Application Binary Interface) (závislé na architektuře, OS, programovacím jazyku). Existuje procesorová funkce CALL x, přidá na zásobník aktuální adresu a přeskočí na danou adresu. RET naopak ze zásobníku vytáhne IP. Argumenty funkce se ukládají na zásobník před CALL, po RET je musí volající odstranit. (1. argument je nejblíže SP, poslední nejdále).

Části zásobníku patřící k 1 volání se říká stack frame. EBP ukazuje do na původní hodnotu EBP, která je přesně pod návratovou adresou, tj. návratová adresa je `EBP + 4`. Pod EBP je prostor na lokální proměnné a pod nimi pracovní prostor, který končí ESP. Výsledek funkce se vrací v EAX, pokud je $< 32b$ a v EDX horní polovina, EAX dolní, pokud je $< 64b$ a $> 32b$. V ST0, pokud je to float, jinde jinak.

Volaný může libovolně používat EAX, ECX, EDX, ST0-7 a část EFLAGS (tzv. Scratch registry), zbytek musí uvést do původního stavu.

2.2 AMD64

Definice 2.6 (Datové typy)

Integer je i 64 výjimečně 128. Jinak je vše při starém.

Definice 2.7 (Registry)

Nově máme 64b registry RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, které jsou univerzální, tj. se jim říká R0-7. (Z nich se stále dají vykousnout menší registry, ale 16b jsou pomalé), navíc přibyli R8-15. EIP a EFLAGS se změnily na RIP a RFLAGS. Floatová jednotka zůstala (ale většinou se používá jen na long double, zbytek umí vector).

Vectory přibyli XMM8-15 a YMMO8-15.

Definice 2.8 (ABI)

Už se téměř vždy předávají argumenty v registrech. A floatové argumenty / výsledky přešly do XMM0-7 (80b stále v ST0).

Navíc se smí používat tzv. red zone, tj. 128 pod SP. (Jsou kvůli tomu různé hacky, např. v Ctrl-z).

V GCC je implicitně nastaven přepínač -fomit-frame-pointer tj. nepoužívá se EBP, pokud není potřeba.

Poznámka
Dříve byl pouze proces připojený sběrnicí k paměti a periferii. Postupně se ale vyvinuly 2 paměti: statická (klopné obvody, spolehlivá, ale má malou hustotu dat) a dynamická (kondenzátor, destruktivní čtení, zapomíná – nutnost občerstvování)). Také se začalo narážet na rychlost světla. Tedy se přidali cache.

Definice 2.9 (Cache)

Řádky jsou nejčastěji 64 bytové.

Nejdříve můžeme mít plně asociativní cache. Např. 32 bitová adresa je rozdělená na 26b čísla řádku a 6 bitů pozice v řadku. Cache je pak prostě pole položek (dvojic tag (= číslo řádku = horních 26b adresy) a obsah řádku). To je však pomalé vůči adresování.

Takže máme přímo mapované cache. Ta adresu rozděluje na tag, index a pozici v řádku, tag se používá pro kontrolu, zda je to opravdu daný řádek, a indexem (nebo jinak hashovanou adresou) se indexuje cache. Ta ale aliasuje (adresy lišící se o 64 bitů se navzájem vyhazují z cache).

Tedy se používá množinově asociativní cache, která má pro každý index množinu míst (předchozí si pro každý index pamatovala nejvýše 1 záznam).

Pro vyhazování z množin se používá LRU = least recently used.

Pro zrychlení se navíc staví hierarchie cachí. Přibližné parametry jsou: L1 cache jsou dvě, instrukční a datová, obě 32kB, instrukční 4-cestná, datová 8-cestná. Druhá (a třetí) cache jsou dohromady, v řádech MB a také 8-cestné.

Definice 2.10 (Zápis)

Můžeme buď zapisovat write-through (= rovnou zapíšeme do paměti) nebo write-back (= máme dirty bit u každého záznamu v cachi, pokud chceme zapsat něco, co není v cachi, musí se to přečíst).

Definice 2.11 (Memory Management Unit)

Už jen kvůli oddělení procesů je dobré zavést překlad mezi adresami programu (virtuální adresy) a fyzickými adresami.

Paměť se rozdělila na 4kB stránky, tedy 12 bitů adresy se nepřekládá (offset), zbytek je číslo stránky. Paměť je ale docela malá, tedy se místo seznamu stránek používá strom. Např. může být adresa (64b) rozdělena na 16b rezervovaných (sign. extension. záporné položky jsou systému, kladné user módu), 9 + 9 + 9 + 9 jsou indexy v dané úrovni stromu, 12b je offset (v položce v MMU jsou to flagy: present, writeable, user/system, accessed, dirty, global, kešovatelnost (např. síťové věci nejsou)).

Překlad může skončit dřív nejenom neúspěchem, ale i tak, že místo další úrovně je tam rovnou větší offset, tzv. huge page.

Takhle by to bylo ale velmi pomalé, tedy máme TLB (translation look-aside buffer) = cache na překlady. To má zase hierarchie: ITLB1 (asi 128 položek), DTLB1 (pouze čtení zpaměti, 16 malých + 16 velkých, 4-cestná), TLB2 (256 malých + 32 velkých, 4-cestná).

Poznámka (Interakce cache – MMU)
Cache adresovaná virtuálně – nefunguje, protože fyzická adresa může mít více virtuálních a zápis by tedy neproběhl pro celý svět. Ale je rychlejší.

Cache adresovaná fyzický – funguje, ale je pomalá.

Kompromis – indexovaná virtuálně (6b indexu + 6b offsetu se nezmění překladem, tj. vlastně i fyzicky indexovaná) a tagovaná fyzicky (překlad proběhne zároveň s dotazem na cache, jelikož index máme hned, tedy můžeme začít i s dotazem).

Definice 2.12 (Přepnutí kontextu)

Každý proces má typicky vlastní překlady, tedy při každé změně procesu se přepíše CR3, tj. vyhodí se vše z TLB. Ale i jiné změny (např. alokace) vyvolají změnu TLB (což se alespoň z části řeší explicitní invalidací položky). Dále se to řeší global bitem a PSID.

Definice 2.13 (Bit global, PSID)

Nastavený bit global na 1 zaručuje, že daná stránkovací knihovna nebude vyhozena při přepnutí kontextu.

Zároveň lze udělat i explicitní invalidaci stránky. Dále existuje PSID (process context ID), které se přiřadí stránkám (do dolních volných 12 bitů). Potom při změně kontextu řekneme, zda používáme nové/recyklované PSID a podle toho ne-/flushnem položky s tímto PSID (respektive ještě existuje i explicitní invalidace).

Definice 2.14 (Další přiblížení paměti)

SDRAM (synchronous dynamic) je matice (± čtvercová, 64MB má např. 8kB řádky) DRAM buněk (dynamické = čtení je ničí) + SRAM registr (1 řádek, statický = lze z něho číst dvakrát). Pracuje se tak, že se tzv. Activate řádek, který daný řádek přesune na SRAM registr. Po provedení čtení a úprav se provede tzv. PRECHARGE, čímž se data vrátí zpět.

V realitě je více registrů (např. 4).

2.3 Výpočet

Poznámka (Historicky)
Fetch, decode, read input, alu, write output, PC++, (opakovat).

Definice 2.15 (Pipelining (neboli dnes))

Jednotlivé části vykonávají činnosti paralelně, tedy se vykonává více instrukcí zároveň. Parametry: latence (jak dlouho trvá cesta celou pipelinou) a propustnost (kolik instrukcí za takt můžeme do pipeliny poslat). Můžou se však vytvářet bubliny (některé instrukce jsou na více taktů, nebo závislé, proto se ALU např. rozděluje na 2 takty, aby pipeline byla plynulejší). Větší problém jsou však skoky (proto se predikuje, kam se skočí).

Protipól skalárního procesoru (popsaného zde) jsou vektorové procesory, které mají mnoho ALU apod.

Definice 2.16 (Superskalární procesor)

Mezi skalárními a vektorovými procesory jsou ještě superskalární procesory, které: fetch, decode, read in. (vlastně také ve scheluderu), scheluder, ALU1 + ALU2 + …, write out.

Musí se však udržovat zdání lineárního průběhu (hlavně závislosti), např. se scheluder dívá na vstupy z a výstupy do paměti. Další problém jsou výjimky (např. dělení nulou). Celkově se to řeší tak, že dekodér rozebere operaci na μ-operace. Naopak μ-kro operace lze i slévat (např. podmínka + podmíněný skok).

Také probíhá register renaming, tedy se udržuje zobrazení mezi architekturovými registry (ty, co jsou v programu) a interními registry (fyzickými). Např. pokud 2 instrukce používají shodný pracovní registr, tak se přejmenuje, aby mohly běžet zároveň.

Nakonec probíhá tzv. spekulativní vyhodnocování (do registru se zapisují pouze ověřené instrukce, ale vyhodnocují se i vyspekulované (např. po skoku, po možnosti výjimky, …) s označením, že ještě nejsou ověřené).

Definice 2.17 (Predikce (skoky))

1) statická predikce (bez kontextu): např. skoky dopředu většinou neskočí (u podmínek nevíme, u výskoků z cyklů je to většinou – neskočí), skoky dozadu skočí (protože cykly).

2) dynamická predikce (Branch Target Buffer): pamatuje si u každého skoku, jak dopadl minule a pak předvídá. Starší položky se vyhazují. Funguje např. tak, že se odčítá 1, když neskočil, přičítá 1, když skočil, čím větší, tím pravděpodobnější stav (tzv. saturační čítač, který když by měl přetéct na libovolnou stranu, tak zůstává) (takhle se to dělalo v prvním takovém procesoru: pentiu, ale byla tam chyba).

Nebo v 2) můžeme mít pole saturujících čítačů indexované historií (všemi možnými kombinacemi n-tic skočil/neskočil, tj. 2^n různých historií).

Definice 2.18 (Globální historie skoků)

Pamatuje si historii p posledních (libovolných) podmíněných skoků (občas indexuje i adresou, kam se má skočit). Většinou ve formě hešovací tabulky (potřebuje indexovat dlouhou historii) a máme tzv. agree predictor, který predikuje, které predikce v hešovací tabulce potřebujeme.

Definice 2.19 (Smyčkový prediktor, nepřímé skoky, call/ret)

Smyčkový prediktor si umí zapamatovat, jak dlouhé smyčky jsou.

Existuje i prediktor na výslednou adresu nepřímých skoků.

Stack prediktor má stack adres callů, takže umí zjistit, kam skočí ret.

Hybridní predikce pak předpovídá, který prediktor se má použít.

Definice 2.20 (AMD prediktor)

AMD (Zároveň) používá perceptron pro lokální predikci a TAGE (tag geometric) = prediktory s historiemi délek 2^i, kde každý složitější opravuje (trvá mu to déle, takže se mezitím už stihne něco spočítat a musí se to zahodit, ale alespoň se nezahazuje vše) ten menší…

3 Více procesorů / jader

Definice 3.1 (Symetric Multi-Processing)

Na 1 sběrnici je připojeno více CPU. Symetrii nám ale kazí fyzika (o tom příště) a cache. Proto existuje MESI protokol (na synchronizaci cachí), účastní se ho cache. Stav řádku v cachi: Modified (data mám jen já a jsou novější), Exclusive (moje cache je jediná, která má tento řádek, a data jsou aktuální), Shared (data mám já a možná ještě někdo a jsou aktuální), Invalid (data nemám).

Na začátku Invalid. Pak se vyšle požadavek na čtení, a když do toho nikdo nevstoupí (read bez intervence) přejdeme Exclusive. Při intervenci přejdu do Shared. Exclusive -> (write) Modified -> (write back) -> Exclusive. Z Exclusive mohu zapomenout, tedy -> (flush) Invalid. Pokud chci zapisovat do Shared, pošle se request for ownership -> Exclusive.

Navíc existuje rozšíření MOESI, která rozšiřuje Shared o Owned, kde vlastník Owned se zavazuje data zapsat.

Definice 3.2 (cc/NUMA)

Cache-Cogerent Non-Uniform Memory Architecture. Procesory jsou připojeny ke stejné sběrnici, ale každý má vlastní paměť, do které mohou ale přistupovat i jiné procesory.

Definice 3.3 (Programování na SMP)

1. oddělené procesy (programy) – propojení rourami / sockety apod. (mohou být i na jiných počítačích).

2. vlákna (threads) (na Unixu příkaz fork nebo clone). Tradiční rozhraní je definované standardem POSIX (pthreads). Ještě existuje (od C11) součást standardní knihovny C – C11 threads.

3. explicitní sdílení (např. blok sdílené paměti). Zde jsou rozhraní SysV IPC a POSIX IPC (pletou se, protože se v nich funkce podobně jmenují, ale SysV IPC nechceme používat!).

Pozor (Proglémy se sdílením)
1. důvod, proč řešit synchronizaci – race conditions (časové hazardy) v programu (úseky závisející na pořadí příkazů, např. inc. a read variable). 2. důvod – relativní viditelnost změn mezi CPU. 3. důvod – procesory přehazují pořadí instrukcí.

Efektivita – viz MOESI, minimalizovat sdílená R/W data. (Pozor na falešné sdílení, tj. 2 různá data v 1 cache řádku. Pomůže posix_memalign() nebo je zabalit do struktury, která se bude zarovnávat na řádky cache).

Bezpečnost syscallů a knihovních funkcí – nebezpečné, thread-safe, signal-safe (lze volat v programu a zároveň probíhajícím přerušení, např. převod do času v č. zóně není signal-safe a zavolání této funkce v signal-handleru může vyvolat deadlock (pokud byla v tu chvíli využívána)), specifické záruky (např. zápis na konec souboru zaručuje, že se soubor bude tvářit ve finální velikosti, takže 2 zápisy na konec se nepromíchají).

Definice 3.4 (Thread-local variables)

pthread_getspecific (otravné a pomalé), tedy GCC zavádí ___thread a C!! _Thread_local.

Každé vlákno má vlastní instanci proměnné. V implementaci je použit registr GS z tzv. segmentových registrů, které ukazují na nějakou „základní“ adresu (adresu, ze které se pak počítají ostatní adresy proměnných).

Definice 3.5 (Synchronizační primitiva)

Zámek (mutex = mutual exclusion) – stav = odemčeno / zamčeno, operace = lock (nebo acquire) / unlock (nebo release), při lock se buď zamkne (pokud je odemčený), jinak se čeká.

Semafor – stav = $x \geq 0$, operace = down (nebo p) / up (nebo v)[a], při down se buď sníží ($x > 0$), nebo čeká.

Condition variable – stav = mutex + fronta procesů, operace = wait / signal, wait čeká na změnu stavu, tedy signal. Mutex je tam, aby se dala udělat kontrola stavu a wait atomicky (= dohromady).

Synchronizační primitiva jsou dnes již rychlá (hlavně v linuxu, např. mutex je implementován jako tzv. futex (typicky user-space, výjimečně kernel)), ale zámky jsou sdílená R/W data, takže budou stejně docela pomalá. Další problém je, kolik jich mít (každý prvek seznamu vs. celý seznam), řeší se hešovacími zámky.

[a] `https://en.wikipedia.org/wiki/Semaphore_%28programming%29#Operation_names`

Definice 3.6 (Paměť sdílená mezi procesy)

Jedna možnost je vytvořit sdílený soubor, na linuxu pomocí nmap. Tím dostaneme nějakou stránku (stránky), u které linux garantuje, že bude pro stejné procesy stejné. Navíc má linux „složku“ tmpfs, kde jsou soubory uložené pouze v RAMce, tedy je to i dost rychlé (jinak se propisují zápisy na disk). Jinde není nmap, ale msync.

Jinak existuje Sys IPC, ale o tom se bavit nebudeme, a POSIX IPC, na linuxu nmap v /dev/shm/.

Interakce s NUMOu je tak, že se dá zapnout striped allocate, takže se data rozdělí rovnoměrně mezi nody. Pokud má každý proces data téměř výhradně pro sebe, tak pokud jsou nody stejně vyvážené (tj. nepřestěhuje ho to při jeho běhu), tak se zase dají alokovat přímo na tomto nodu.

Definice 3.7 (Atomické operace)

Většinou: read/write, inc/dec, exchange, compare & exchange (atomicky přepíše s nějakou podmínkou, je potřeba na implementaci zámků) nebo test & set.

Definice 3.8 (Polopropustné bariéry)

mutex.lock je bariéra typu acquire, tedy nic se nesmí přesunout před ní. mutex.unlock je bariéra typu release, tedy nic se nesmí přesunout za ni. (Opačně se ale operace přesouvat mohou.)

Store by mělo být release a load acquire.

Definice 3.9 (Futex)

Vázaný na fyzickou adresu v paměti, má hodnotu 32b intu, má uvnitř jádra frontu čekajících procesů a 2 operace futex_wait (s očekávanou hodnotou) a futex_wake (s počtem procesů).

Definice 3.10 (Transakční paměť)

Struktura je START … (čtení a zápisy) … COMMIT. V kódu není při čtení vidět žádná změna po START a všechny zápisy před COMMIT nejsou nikde jinde vidět. Při COMMIT se zkontroluje, zda se data se kterými se pracovalo nezměnili, případně se začne znovu (tzv. rollback, vrátí se vše lokálně změněné do původního stavu).

Softwarově implementováno pomocí žurnálu a commit locku. Ale je to pomalé a existují hardwarové implementace (IBM má, Intel zatím nemá 100% funkční), využívá MESI a hierarchii cachí.

Poznámka (Co potkáme dnes)
Více CPU na 1 čipu – více jader (samostatná CPU + sdílené I/O a část keší) nebo SMT (symmetric multi-threading (Intel to nazývá hyper-threading), sdílené výpočetní jednotky mezi procesy).

AMD Buldozer zase uvedl model, kdy jádro má vlastní scheduler, int. jednotky, registry a L1D. Naopak fetch + decode, LI1, floatové jednotky a L2 mají společné 2 jádra.

Definice 3.11 (Sběrnice)

Rychlost se udává v šířce sběrnice a počtu cyklů za sekundu, tedy kolikrát se přenese šířka za sekundu.

FSB = 64 bitů, 10^3GT/s (gigatransferů za sekundu); RAM DDR2-1066 = 1.066 GT/s, 64b; dnes DDR4 = 2.1 GT/s, 64b; PCI = 33 MT/S, 32b; PCI Express (PCIe) = 1-32 linek, 2.0 = 500MB/S per lane, 3.0 = 985 MB/s per lane;

HyperTransport: 8/16/32 bitů na linku, 2 módy: I/O mód (strom, komunikace procesor — periferie) a koherentní (obecný graf, komunikace procesory — procesory). Z toho vznikl HTX connector a z toho NUMA connect, které umí propojit spoustu HT propojených topologií. Tam už nefunguje MOESI, takže se používá directory-based koherence.

Definice 3.12 (HT-Assist)

Directory based coherence protocol. Každá buňka RAM má svůj home node, tam se řeší synchronizace. Home-node posílá dotaz všem a zároveň čte z RAMky.

Definice 3.13 (Vektorové operace)

Tzv. SIMD (Single Instruction Multiple Data) paralelismus. Aplikace: lineární algebra (zpracování obrazu a zvuků, machine learning), blokové zpracování dat (zda se v tomto velkém souboru nachází 0), tentýž algoritmus pro různé vstupy.

1996 Intel MMX (multimedia extensions) (archaická věc, ale stále se odráží v dnešním SIMD) – 8 64-bitových vektorových registrů (MM0-MM7) sdílených s FPU (muselo se však přepínat a to bylo ukrutně pomalé), integerová (i saturační) aritmetika (add, sub, mul). 1998 3DNow rozšířilo MMX o floatové vektory (2011 zrušeno).

Intel SSE (Streaming SIMD Extensions) – registry XMM0-XMM7 (v 64-bit módu až XMM15), které jsou 128-bitové ($16 \times 8b, 8 \times 16b, 4 \times 32b, 2 \times 64b$, poslední dva i floatové operace). Dále vznikly další verze…

ADDPD (= add + packed + double (=64b)), ADDSUBPS (= add + sub (na střídačku, vhodné např pro furierku) + packed + single (=32b)), CMPEQPD (= CMP (compare) + EQ (rovnost) + packed + double) vyrobí bit masku, MAXPD (maximum), HADDPS (= horizontal + add + p + s) sčítá abcd+efgh = (e+f, g+h, a+b, c+d).

PADDW (=packed + add + word), PADDSW (saturační verze), PAND, PAVGB (průměr), PMINB, PMAXB, PCMPEQB.

Pak ještě přesuny: MOVAPD (= move + aligned + packed + double), MOVDQA (= move + double quad + aligned), MOVUPD (=…unaligned…), MOVDQU. A permutace: PSHUFD (řízeno 8bit tabulkou u instrukce, kde určujeme, co se má vzít, kde).

Délky jsou $S = 32$, $D = 64$ floaty, $B = 8$, $W = 16$, $D = 32$, $Q = 64$, $DQ = 128$.

AVX (Advanced Vector eXtensions) – 256-bit registry YMM0-15 (rozšíření XMMn), původně jen floatové (AVX2 má navíc integery, které také nevyžaduje alignment a umí gather (vektorové indexování)).

Definice 3.14 (Programování vektorových instrukcí (GCC))

Autovektorizace (GCC umí rozbalovat smyčky na vektorové operace). Nebo přímo GCC vektory (`typedef int __attribute__((vector_size(16))) vec;`, kde 16 (libovolná velikost, GCC si to srovná) je v bytech a int určuje velikost 1 prvku). Pak $+-=$ funguje po složkách, můžeme indexovat (`x[i]`), násobit po složkách ($*$), nebo přímo instrukce: `__builtin_ia32_paddb(x, y)`. Nakonec existuje i Intelovské C.

Definice 3.15 (/proc/cpuinfo)

Obsahuje informace o všech procesorech (virtuálních), včetně třeba flags, kde jsou vyjmenované featury, co procesor umí.

Dále se probírali nové featury Céčka (hlavně chary). Následně se probíral projekt Larrabee (Intel), speciálně procesor Knights Landing.i Nakonec AMD Zen a Zeppelin.

Poznámka (Bezpečnostní chyby)
Sideefekty na cachích prosakují informace o šifrovacích klíčích (AES i RSA). Prostě jednoduše zaplníme cache, když běžíme na druhém jádře na stejném procesoru, a díváme se, které adresy byly vyhozeny.

Ingredience jsou spekulace a side-efekty (na cachi, při HT vytížení jednotek).

Meltdown: Obcházení přístupových práv ke stránkám: práva se ověřují spekulativně. Výjimka by nám to ale mohla zašumět, tedy se to musí provést buď v transakci, nebo za mispredikovaným skokem. Výhradně Intel.

Spectre: (v rámci jednoho adresního prostoru) obejití kontrol tak, že se mi věci přečtou jen spekulativně a já pak vyhodím výjimku, čímž se věci nezahodí. Velký problém JavaScriptu, navíc na všech procesorech s predikcí skoků.

```
int f(uint i) {
    if (i < n) return x[i] else error();
}

y[64*f(i)];
```

Spectre pokračování – zneužití druhé části predikce skoků: prediktor, kam se skočí (naučím na svém programu, kde nějaký skok skáče na pořád stejnou adresu, potom skok v oběti skočí spekulativně tamtéž (v rámci kódu oběti)). Obrana je retpoline: místo skoku se zavolá funkce, která si přenastaví RSP, aby se vrátila tam, kam jsme chtěli skočit. (Tj. zakáže konkrétní predikci skoku a ne všechny v rámci celého procesoru.)

L1TF (level 1 terminal fault) (foreshadow): pokud překlad stránky selhal (zneplatněnou položkou), stejně (Intel neumí zahodit instrukci jinde než na konci) instrukce pokračuje s adresou danou číslem neplatné stránky, který použije. Normálně je malé, ale přes virtualizaci se dá zvýšit na adresy, které už obsahují data. Funguje však jen na datech v L1 cache. Lze se bránit flushem L1 cache, ale zkazí nám to HT.

MDS (Fallout, RIDL, Zombieload) (2019): útoky na buffery (hlavně na store buffer, kde se hromadí zápisy (jak predikované, tak aktuální, aby se zprůměroval zápis), ale i load buffery a fillbuffery), kde se porovnává pouze část adresy: 1. naplním store buffer, 2. zavolám kernel, který zapíše někam, kam nemůžu, 3. čtu z adresy, kam mám přístup a která se shoduje v dolních 12 bitech. Řeší se bariérami kolem syscallů a přerušení a uspání / probuzení vlákna.

Na informacích o procesoru si můžeme přečíst řádek bugs…

Poznámka
Nakonec bylo ukázáno, jak se optimalizoval sort pro LibUCW a vnější třídění (= třídění souborů).