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.

TODO!

Definice 2.9 (Bit global)

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).

Definice 2.10 (Další přiblížení paměti)

SDRAM (synchronous dynamic) je matice (± čtvercová) DRAM buněk + SRAM registr (1 řádek). 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.

2.3 Výpočet

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

Definice 2.11 (Pipelining (neboli dnes))

Jednotlivé části vykonávají činnosti paralelně, tedy se vykonává více instrukcí zároveň. Parametry: latence a propustnost. 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 2.12 (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 2.13 (Predikce (skoky))

1) statická predikce (bez kontextu): např. skoky dopředu většinou neskočí, skoky dozadu skočí (protože cykly).

2) dynamická predikce (BranchTargetBuffer): 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.

Definice 2.14 (Globální historie skoků)

Většinou ve formě hešovací tabulky a máme tzv. agree predictor, který predikuje, které predikce v hešovací tabulce potřebujeme.

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()).

Bezpečnost syscallů a knihovních funkcí – nebezpečné, thread-safe, signal-safe (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.

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, 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