B35APO

ve schématech a memech
manuál pro podpivasniky, kteří se ulíli ze všech přednášek z apo

WARNINGY!
- **Běloruské a tatarské memy**
- **Žádné poskytnutí záruky**
- **Vulgární mluva**
- **Bez šotu tequilly se nevyznáš**

Autorkou tohoto humoristicke-vdělavacího pokladu je Hanna Hliavitskaya. Zpracovával Azat Mukhametshin.
Pomolete se za nedotčanost našich duš na FELu.

Úspěšné Hladové hry a kéž vás vždy provází štěstěna!

Předmluva

Manuál byl vytvařen pro mě za účelem opakování i pro vás, abyste měli alespoň malou část látky strukturovanou. Všechny zdroje budou přiloženy ke každé úloze ve tvaru odkazů na články nebo videí s timekódy. Úlohy jsou probrány především s praktického pohledu. Kdybyste očekavali podrobný ale stručný teoretický kurz APO, material se vám nehodí. Museli jste poslouchat Pišu :)
Velice se omlouvám za všechny chyby. Pokud jste našli nějakou, můžete mi slušně napsát a to opravím v dokumentu. Všechny úlohy jsou z „APO for dummies“ neboli ze zkoušek 2022 roku, takže nejsou tady žádná tajemství, prostě řešení pro hloupé, jako jsem já.

Bude to nudný, tíživý, nepochopitelný, ale já se pokusím vtipkovat vtipy a kreslit pro vás schématy, abyste nebyli smutní. Mockrát děkuji mého skvělého redaktora, který zlepšil text cenou svých nervů specialně pro vás. Hodně štěstí a trpělivosti. Pokud ale dokázali přečíst naše příjmení, APO nebude pro vás problémem =)

***manuál má** rus verze*

SUSUME!

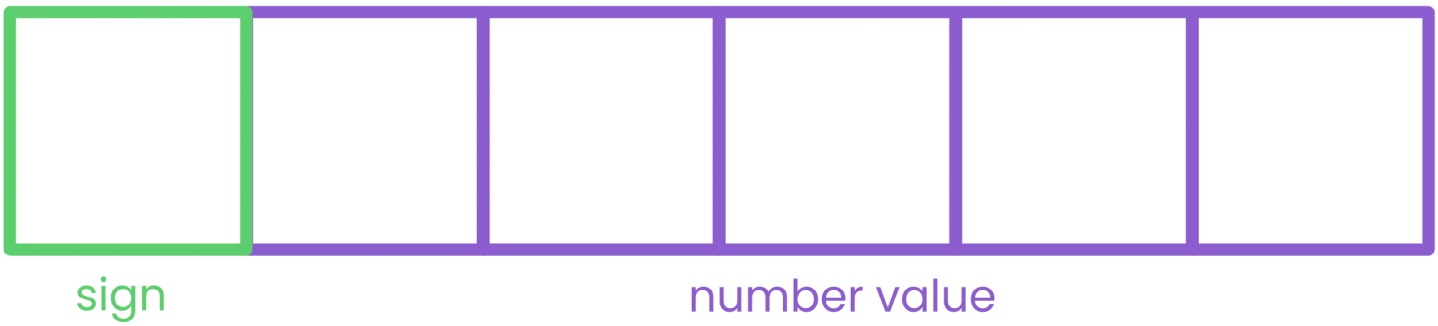
hAL(el)U(ja) kamarádi, vítáme binární aritmetiku!

Přiznávám se, zpočátku tohle téma tady nebylo. Měla jsem mylný pocit, že to všechno je zřejmé jako 2+2. Avšak po té, co jsem napsala zkoušku a místo 4 jsem dostala 70, uvědomila jsem si přitomnost problému. Jelikož je, budeme ho řešit.
Takže, všichni víme, co to jsou binární čísla a jak fungujou nejjednodušší bezznamenková čísla. Pokud si nepamatujete nic, Štěpan vám všechno vysvětlí. Pokud ale nejste totální nulou, tak takový příklad

```
  1←   1← 1← 1←
    1 0 1 1,1 0
+   1 0 1 0,1 1
-------------------
1 0 1 1 0,0 1
```

nezpůsobí vám žádný problém. Tak kde se problémy začínají? V záporných číslech a odčítání v principu.
Tedy jsme udělali (přesněji udělali to inženýři před mnoha lety) mini-kalkuláčku, která umí sčítat. A všechno je dobře, ale existujou i situace, ve kterých potřebujeme odečítat. Inženýr ale je člověkem, který úplně pochopil smysl slova lenost, takže se mu vůbec nechce pájet nový obvod pro jednou operaci. Proto se rohodlo řešit problém chytře: přetvořit sčítání v odčítání. A po tom i všechny operace. Zní blbě, ale jen zní. Ve skutečnosti to je velice perspektivný nápad, protože byl vynalezen *doplňkový kód*.

Převádět čísla do doplňkového kódu zní... vůbec nezní zřejmě. Ale pokusíme si to vyřidit. Všechná naše čísla leží v registrech. Registry se vypadají jako krabičky ze krabiček – trigerů, které mají pouze 2 stabilní stavy: 0 a 1. Registry můžou být různymi a obsahovat 8, 16, 32, 64… trigery. My zvolíme 6-bitový registr jako příklad.



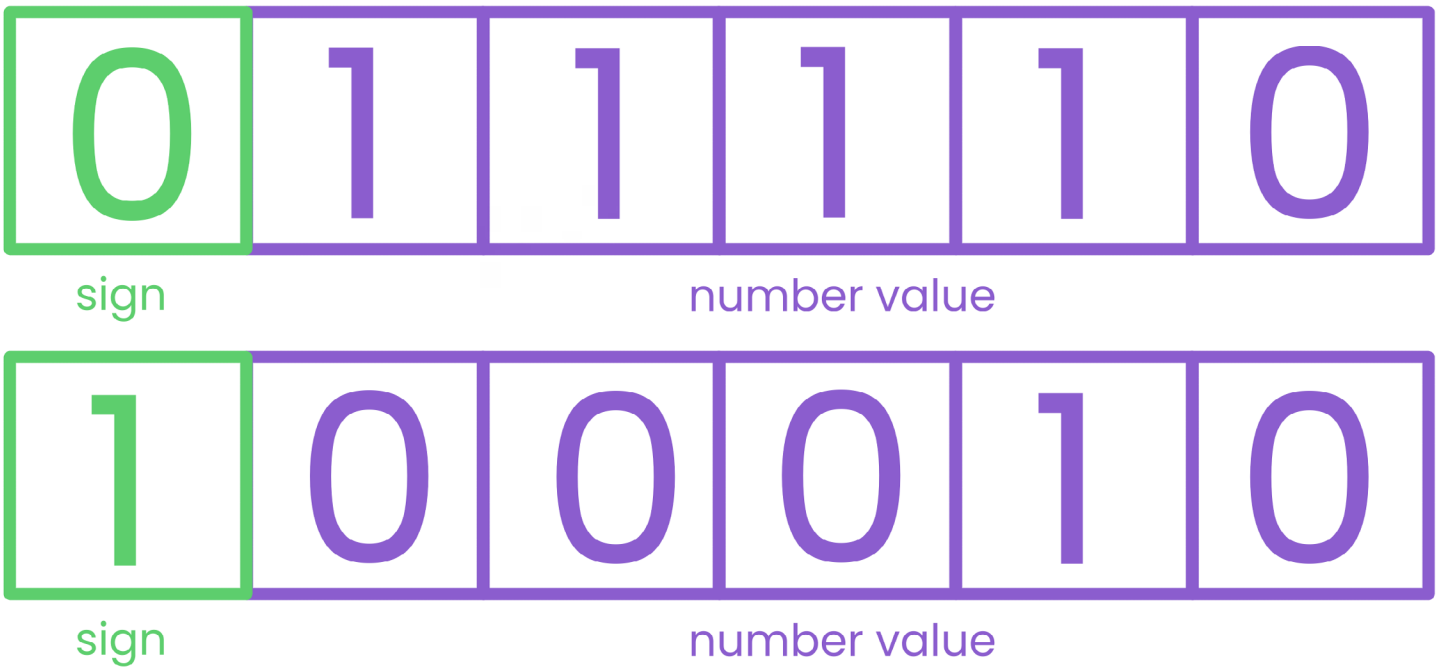
První bit je zelený ne nadarmo. Fakticky on je pomocný, odlišný od ostatních. Vše, co ukazuje, je znamémko čísla (0 je plus 1 znamená minus) Ostatní vyjadřují samo číslo.
Zkusme si tam zapsat 30, binarně 11110.

Protože číslo 30 je kladné, znaménkový bit je nulou. Kdybychom si vzali -30, obraz by vypadal takhle:

Ten celý div se jmenuje *přímý kód*, t.j. bez žádných fint. Tento kód je velmi jednoduchy na pochopení, ale težce se s nim počítá, proto se vydáváme na doplňkový kód.
Abychom se k němu dostali, potřebujeme ještě jeden mezikrok – *inverzní kód.*

Jedničkový doplněk (není doplňkový!) neboli inverzní kód se liší od přímého pouze v případě záporných čísel. Pro kladná je shodný s přímým. Jeho základ je velice prostý: všechny bity mimo znaménkového se mění z 0 na 1 a z 1 na 0 (tady a dále první schéma je číslem 30, druhá -30).

Inverzní kód nám skoro vyhovuje. Jediné, co zůstává, je přičítání jedničky (doslovně jedničky, v našem prípadě 000001) a dostaneme *doplňkový kód*, neboli *dvojkový doplněk*. Mimochodem, kladná čísla se zase shodují s přímým kódem.



Můžete klidně používat stejny algoritmus pro překlad z doplňkového do přímého kódu: nejprve provádíte inverzi, pak přičítáte jedničku.
Pozor ještě jednou: to všechno je potřeba pouze pro záporná čísla.
Podrobně to probrat můžete třeba tady nebo se podívat, jak taková čísla se spočítají, můžete taky tady ale s jiným time-kódem.

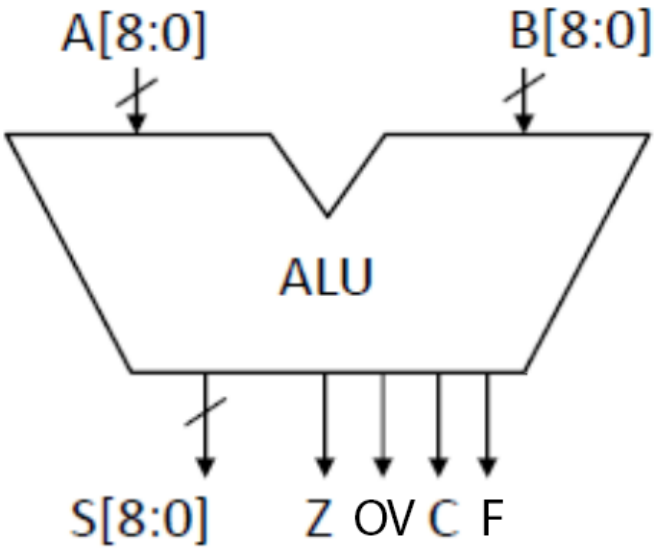
Gratuluji, nyní víte, co doplňkový kód je, ale existuje i jeho brácha – *kód s posunutou nulou*.
Skvěla přednáška o tomhle je zde.
Velice zjednodušenně a kratce: my posouváme interval od nuly do nějakého maximalního čísla (my máme $2^6 - 1 = 63$) o nějakou konstantu N. Táhle konstants se jmenuje *additivní konstanta* ($2^{(n-1)} - 1$ *kde n je počet bitů v registru*), a ještě se s ní sejdeme.
Jádro algoritmu je v tom, že k výchozímu číslu přičítáte N. A... to je vše. Pri dekódování odečítáte.

Ukažeme na příkladu -10 a konstantě 31. První krok: -10 + 31=21. Hotovo! Pro přehlednost přeložme to do binární soustavy: *010101*.
Mimochosem, přičítat lze jak binarně, tak i decimalně, nic na tom nezáleží.

Zvítězili jsme reprezentaci! Teď kdyby se vás úloha zeptala, tak poznáte čísla v libovolném tváru. Avšak kapitola je ob aritmetice, a proto se musíme porvat i s ní.
Vlastně zajímá nás ne aritmetické operace, ale kalkulačka, která je provádí.

Dámy a panové, vítejte ALUčko

Neříkejte mi, že nevíte, co to je. Přestaňte se klamat. Vsadím se o pivo, že jste všichni viděli tento obráz:



Možná tam bylo míň šípek, nebo byla jedna navíc od controlleru, která přeposílá instrukce, anyways. Poznáte naši kalkubľačku, která přijímá operandy A a B, provádí aritmetické kouzlo a vyplivuje výsledek S. Než se zeptáte:

a) Vlaječka Z se nastaví na jedničku pokud výsledek je nulou
b) C se nastaví na 1, pokud výsledek se neumístí do velikosti výchozích registrů (Overflow. Pozor, overflowem je C, zatímco OV je overflow flag a označuje něco jiného. Tady je odkaz na přesný popis té blbosti anglicky)
c) OV bude jedničkou, sečetli-li jsme dvě kladná čísla a dostali jedničku v znaménkovém bitu, nebo naopak: sečetli-li jsme dvě záporná čísla a dostali nulu v znaménkovém bitu. V ostatních případech bude nulou.
d) F bude, pokud Z, C a OV jsou nastavené na nulu

Spočítejme příklad a podíváme se, co se stáne. Předpokladejme, že naše input data jsou dvě binární čísla: *A = 1110 1101, B = 1001 0101*. Sčítáme a máme:

$$\begin{array}{r} 1110\ 1101 \\ +\ 1001\ 0101 \\ \hline 1\ 1000\ 0010 \end{array}$$

A zde to je jednoduchý: hned vidíme zbytečnou devátou číslici (ta co je červená). Tohle i je náš overflow, a proto nastavíme C na 1. Kdyby se výsledek umístil do osmi číslic, Céčko by bylo 0. Výsledek určítě není nula, proto i Z je nulou. Znaménkový bit se nezměnil svou hodnotu, proto OV necháme na pokoji.

Ano, to je natolik prostý. Ale čím jednodušší, tím je lepší pro nás.

IEEE 754 neboli svata čísla reálná

Jste rádi? Já jsem, neboť tohle je jediné téma, které si lze uvědomit, pochopit a požívat bez škody pro psychické zdraví.
Co je ieee-754? To je standard kódování reálných čísel ve vašich kompukterech. To jest to, jak vaše technika přežvykuje libovolná čísla s plovoucí desetinnou tečkou.

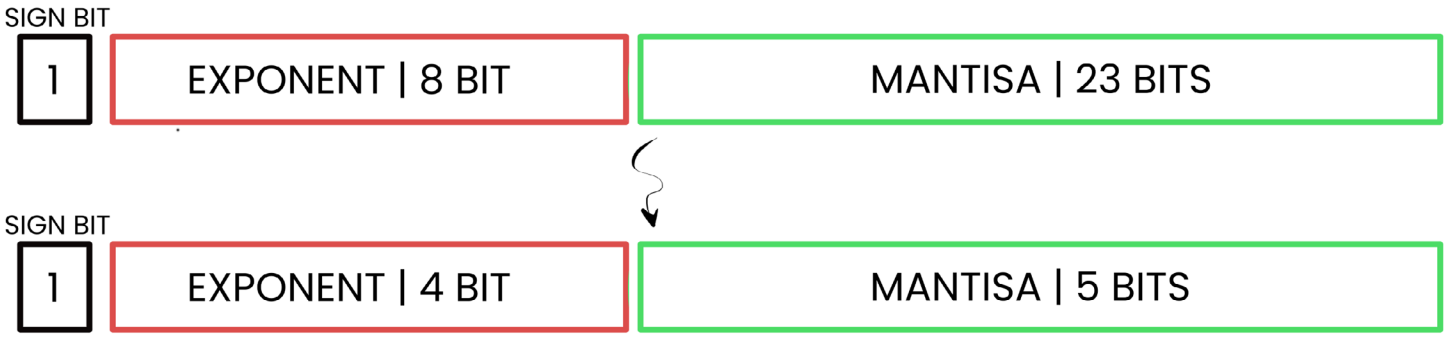
Ve standartu se číslo kóduje ve formátu 32 bitů. Podívat se na ieee zdravého člověka (pokud nerozumíte, co tady dělá zdravý člověk, zeptejte se svých kamarádů z Ruska, určitě vám všechno vysvětlí) můžete tu. Ale APO by nebylo sebou, kdyby nepřepsali všechno tak, jak se jim chce. Proto zkusme pochopit co odkud a kam.

<u>ÚLOHA</u>: máme číslo 5,71. Úkolem je přeložit ho do binárního tvaru pod podmínkou, že *exponent je 4 bity dlouhá a mantisa 5*.

„Wow wow, co je to za mantisa a exponent?“ možná se zeptáte vy, nedělali-li první domácí úkol. Ale prosím nepanikařte, za chvíli to probereme. Libovolné reálné číslo se skládá ze 3 částí:

a) Znaménkového bitu. On určuje, je-li číslo záporným (vyjadřuje se jedničkou) nebo myslí-li pozitivně (nula)
b) Exponentu. Tohle je ta samá příšerná věc s Éčkem a nějakými číslama. Prostě řečeno, to je posun desetinné čárky. Třeba 123,4 = 1,234e2 nebo 0,00078=7,8e-4 (všimli si, jak posun ovlivňuje znaménko exponentu? Skvělý! (Pokud ne, podivejte se pořádně))
c) Mantisy. Mantisa je všem, co je po čárce.

A to všechno schematicky vypadá takhle (nahoře je tvar 32-bitového čísla, dole je ten, do kterého nás poprosili převést):



Vdech-výdech, zatím to máme více méně dobře, že? Rozcvičte se prsty, tady se začíná binární aritmetika.

Zdálo by se, v čem je problém? Ber si číslice a strč je do odpovídajících krabiček. Ale není to až tak jednoduchý a existuje pár nuancí.
Za prvé: *normalizovaný tvar čísla*. Što heta za łajno? To je pravidlo toho, jak bys měl zapisovat čísla, aby vypadalo hezky a kompukter ho dokázal přečíst.
Normalizovaný tvar čísla s plovoucí čárkou je *jedna číslice (ne nula) před čárkou*.
Normalizovaný tvar <u>binárního</u> čísla je *jedna jednička před čárkou*.

Tahle jednička tam stoji (skoro, s výjimkou denormalizovaných čísel , která jsou určeny pro zápis velmi malých čísel) vždy. Jmenuje se skrytá, a má se na paměti. Proto naše mantisa ve skutečnosti je

vše binární po čárce, co vidíme + 1

Za druhé: *před spočtením exponentu, zvětšujeme ho o 127* (pro číslo o 32 bitech) Potřebujeme to pro to, abysme mohli reprezentovat čísla od -127 do 128. (Chceme zapisovat i velmi malá čísla, že jo?) Takže náš skutečný exponent je

binárně zapsaný exponent – 127

Táhle 127ička se vědecky jmenuje *aditivní konstanta*. Již jsme se s ní setkali, spočíta se takhle:

$$2^{(\text{počet bitů v exponentu}-1)} - 1$$

Je vidět, že v našem případě se ona bude rovnat $2^{(4-1)}-1 = 7$.
Na tom zatím ukončíme. Podívejme se na zadání

5,71. Kruté číslo, ale není bohužel binární. Pojďme ho přeložit, když nás poprosili.

Krok namba van: utrhujete celou část a činíme ji do binární podoby:

$$5_{10} = 101_2$$

Krok namba tu: berete si zůstatek po čárce a také překládáte. Připomínám, že za čárkou stupně dvojky jsou stále stupni dvojky, ale záporné. Nejjednodušší způsob převedení mantisy se vysvětluje v přednášce pána Štěpana. Budeme ho používat, ale zastavujeme se po pátém bitu! Máme je pouze pět:

1	0.71	0.5
0	0.2	0.25
1	0.2	0.125
1	0.075	0.0625
0	0.0125	0.03125

$$0.71_{10} = 10110_2$$

Krok namba sri: slepujeme celou a zlomkovou části, posouváme čárku tak, aby před ní stala jen jedna jednička (Dva poslední bity vynecháváme, protože se neumístí do našeho čísla):

1.0110110e2

Krok namba for: spočítáváme exponent, to jest 2+ naše konstanta, která se rovná 7 = 9 (1001_2).
Finále: sestavujeme naše číslo *0 1001 01101* ze znaménka, exponenty a mantisy
Skvělý, že jo? Přeložili jsme číslo, dokonce i správně. Není to zázrak? Tady si můžete zatleskat a vyrušit se na kousek teorie, který je potřeba pro další úkoly.
***Denormalizované číslo*, co to je? Za prvé, to jsou 5 speciálních čísel: *plus nula, minus nula*:**

0^+: 0 00000000 00000000000000000000000
0^-: 1 00000000 00000000000000000000000

***plus nekonečno, minus nekonečno*:**

inf^+: 0 11111111 00000000000000000000000
inf^-: 1 11111111 00000000000000000000000

a *NaN*, t.j. chyba, která se vyskytuje, když třeba dělíte nulu nulou. Například

***NaN*: 1 11111111 00000100000000000000000**

Vše, co má v exponentu jenom jedničky a má jedničky v mantise, je NaN'em.
Za druhé, velice malá čísla, která mají exponent menší než -127.

Fuj, na tom se teorie končí. Pojďme už dokončit ta čísla.

<u>ÚLOHA:</u> V našem systému, kde exponent je 4 bity a mantisa 5, najít kladné nejmenší a největší normalizované číslo v binárním a dekadickém tvarech.

Začněme z *nejmenšího*:
Binární podoba: 0 0001 00000
Dekadickou popíšu podrobněji.
Mantisu chápe i malé dítě, je nulou. Nezapomínejme ale *<u>přidat skrytou jedničku</u>*, máme 1.0
Exponent je 1, odčítáme aditivní konstantu, máme 1 - 7 = -6
Násobíme $1.0*2^{-6} = 0.015625$. Voila!!!

***Největší*:**
Binárně: 0 1110 11111
Dekadicky již bez podrobností: $1.96875 * 2^7 = 252$

Gratuluju! Teď umíte Překládat čísla s desetinnou čárkou!

chalera, navošta ja pračnułasia..

Cache neboli hlavní prdel ve vašem testu hned po assembleru

Co si říci? No, cache. Krutá, rychlá, ale velice malá paměť, která je přilepena k vašemu procesoru. Všechno ve prospěch výkonnosti a rychlosti běhu programu. Ale teorii si můžete přečíst i na Wikipedii nebo na začátku přednášky pána Štěpana (chráním vás, žádné odkazy nejsou s Píšou).
Co potřebujeme, abysme dokázali více méně úspěšně řešit úlohy v testu? Pojďme dle seznamu:

a) Co to je *slovo* (*paměťová jednotka*)? Slovem je jednotka informaci fixní délky, kterou váš počítač čte z paměti a zpracovává je.
b) Co to je *blok*? To je minimální část cache-paměti. Nejmenší možná délka bloku je délka slova, ale obvykle je rozměr slova * 2^n. Kousky právě té délky si bude brát cache. Cache prostě neumí číst jedno slovo (pokud blok není slovem), principiálně čte všechno blokama.
c) Co to je *set*? Sety jsou očíslované jednotky cache-paměti. Tady bude těžce, proto doporučuji kouknout na přednášku.
Samá o sobě se jednotka cachu skládá z tagu + bloku + bitu validity(toto byl by set pro úplně asociativní cache).

Všechno je hezky, ale kdybychom třídili slova pouze dle tagu komparátorů, které hledají match s buňkama, byli bysme v prdeli. Proto chytří lidé vymyslili tříděni dle setu (takový cache s sety se jmenuje Direct-Mapped Cache), což je skvěly, dobry, ale existuje obrovslké ALE
Co dělat, když dojde k kolizi setů na různých místech? My nemůžeme zapisovat různé tagy v ten samý set, protože se prostě ztratíme. Nedokážeme najít, kde leží kousek informace, který potřebujeme. Fakt máme vyhazovat data?

d) Obejdeme se bez vyhazování. (Хуй там!) Vytvoříme mnohocestnou cache. To, kolik máme cest, se jmenuje *stupeň asociativity*. Čím je to dobrý? No to znamená, že každý set dostává extra buňky. Proto kdybysme zaplnili první set-buňku 00 tagem 1001 a objevilo se další slovo pro set 00 ale s tagem 0010, nepotřebovali bysme nic vyhazovat. Prostě bysme šli do další cesty a přidali data do setu 00 tamhle.

No nevypadá až tak strašně, ale v abecedě jsme nedošli ani do ř. Pokud na této fázi ničemu nerozumíte, dejte si pivo a nestresujte. Během praxe se pochopí všechno. Rozcvičte se, usmějte se a pustíme se do praxe.

ÚLOHA: máme cache s rozměrem 2048 B (Pozor, velká B znamená BYTY!), stupeň asociativity je 1. Spolu s tím blok je 64 bytů, slovo je 32 BITŮ (ne bytů)
Dále zatím nebudeme ani číst, vypořádejme se s strukturou cachu. Je potřeba najít počet setů. Klidně zodpovíme, že je blbá otázka, a pišme následující:
2048 bytů cachu dělíme rozměrem bloka 64 bytů

2048 / 64 = 32 setů*

Dostáváme skoro počet setů. Proč skoro? Potřebujeme to vydělit stupněm asociativity, v našem případě jedničkou. Proč? Protože ten samý set v jiné cestě není nový set, je tím samým setem, takže pozorujeme, abysme nepočítali té samé věci několikrát. Celkem máme:

32 / 1 = 32 setů

Zatím se všechno vypadá dobře. Podívejme se, co máme udělat dál.
ÚLOHA POKRAČOVÁNÍ: Předpokládejme, že program začíná číst na adrese 0x33C a čte 692 slov 7 krát po řadě. Jaký set bude přečten z adresy 0x33C...
Stop, již slintáme a posloucháme hřmot talířů v hlavě. Co tady dělají adresy?
Tady se začíná vyprávění o tom, co to je adresa v počítači, které se pokračuje jen v příští kapitole. Zatím prostě porozumíme, jak cache vidí tuhle adresu. Máme nějakou 0x33C, a pokud jsme nezapomněli, jak se překládá do binárního tvaru, máme:

0011 0011 1100

Rozeberme naše Lego. Adresa musí obsahovat tag, číslo setu a chtělo by se, aby tam byl i offset, to jest pozice někde v bloku informace

sets num = 2^k, set = k bit 2^n = blok size -> offset = n bits

Rozměr bloku 64 -> 2^6 -> takže na definování offsetu potřebujeme ž bitů. Analogicky na set potřebujeme 32 ->2 -> 5 bitů. Celý zůstatek je tagem. Takže naši adresu můžeme vyložit jako

0011 0011 1100

Vypadá hezky, že? Set, do kterého budeme číst poprvé, je 01100, to jest 12 dekadicky. Současně jsme nalezli i offset 111100, což je 60. Pokud se náhodou pletete v adresách, tady je video s vysvětlením. Chtělo by se tady skončit, ale hit a miss se nevysvětlí.
Co to je hit? To je situace, při které se nám podařilo najít v cachi to, co jsme hledali. Miss znamená, že se nepodařilo. Protože jsme vynalezli cache pro rychlejší práci, chceme, aby se dařilo a nutné kousky informace se objevovali v cachi. Frekvence objevování se počíta docela snadně:

Hit rate = počet hitů / počet dotazů

To je ale kombinatorika pro suchary. Pojďme dopočítat naši úlohu.
ÚLOHA POKRAČOVÁNÍ ZASE: Spočítejte, k kolika hitů/missů dochází za jeden cyklus a za vše 7. Připomínám, že čteme 692 slov za cyklus.
Hrůza! Zase musíme přemýšlet. Začneme ale od začátku a od otázky o jednom cyklu. Co se vůbec děje, když vykonáváme první cyklus čtení?
Zaprvé, na začátku máme prázdný cache, což nám dává +1 miss. Slušně, ani jsme nezačali, ale již se pokazili. Ale načítáme první blok, co to vůbec znamená? Znamená, že jame přidali do cache procesoru n-ný počet slov, aby on dokázal je přečíst.
Kolik jsme přidali? Přesně tolik, kolik se umístí do bloku (připomínám: cachovi je to fuk, on bere informaci blokama). V našem případě máme blok 64 bytů, slovo 32 bitů, víme, že byte má 8 bitů, proto klidně dostáváme

(64*8) / 32 = 16 slov

To jest 16 slov procesor bere bez obtíží. Co ale bude, když půjde za 17?

Bude pich, neboli miss, a cache bude muset vzít další 16 slov. K takové situaci dojde přesně

692 / 16 = 43,25 to jest 44krát

+1 pich z prvního pokusu

celkem 45 *missů*, proto 692 – 45 = 647 *hitů*

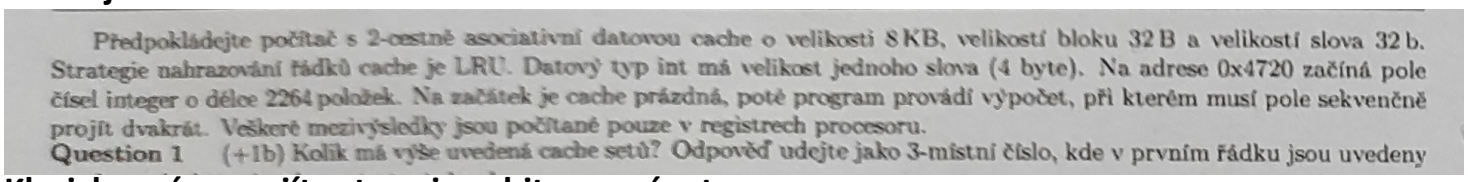
Jeden cyklus máme hotovo, ale co se bude dít dál? Bohužel, cache dokáže umístit jen (2048 * 8) = 512 slov, což je o 180 slov míň, než potřebujeme v jednom cyklu. To je smůla a znamená, že 180 slov, to jest 12 setů (180 / 16 = 11,25 a zaokrouhlujeme nahoru) budeme ztracovat dvakrát: jednou na začátku cyklu a jednou na konci. Tohle se bude dít v každé iteraci, mimo první. Celkem:

45 + (12 * 2 * 6) = 189 *missů* a (692 * 7) – 189 = 4655 *hitů*

Hrůza? Určitě. Ale gratuluji, zvládnuli jsme cache! Téma hierarchie zůstává na ramenech pána Štěpána. Věřím v něho.

Update úloh cachu: prdel, která pokazila výpočty, neboli mnohocestnost

Lidé zašli do zkoušky, a najednou jsme pochopili, že jsme debilové. Nebo to, že ti, kteří tvoří zkoušky a nevysvětlují nic tykajícího se praxe, jsou debilové. Každopádně se objevilo, že mnohocestnost strašně kazí výpočty a jsme nuceni počítat trošku jinači. Ale prosím, neplačte. Podařilo se nám najít snadné řešení.
Úloha je na obrázku:



Předpokládejte počítač s 2-cestně asociativní datovou cache o velikosti 8 KB, velikostí bloku 32 B a velikostí slova 32 b. Strategie nahrazování řádků cache je LRU. Datový typ int má velikost jednoho slova (4 byte). Na adrese 0x4720 začíná pole čísel integer o délce 2264 položek. Na začátek je cache prázdná, poté program provádí výpočet, při kterém musí pole sekvenčně projít dvakrát. Veškeré mezivýsledky jsou počítané pouze v registrech procesoru.
Question 1 (+1b) Kolik má výše uvedená cache setů? Odpověď udejte jako 3-místní číslo, kde v prvním řádku jsou uvedeny

Klasicky máme najít sety, missy, hity, první set.
Nalezení počtu setů a prvního setu se nemění. Pokud máte s tím problémy, podívejte se ještě jednou na předchozí kapitolu.
Pro následující výpočty potřebujeme jen počet setů:

(8 * 1024) / (32 * 2) = 128 setů

Problémy se začínají, když zkoušíme spočítat missy a hity. Pamatujeme si ze přednášek nebo mého mrzačeného vysvětlení, že mnohocestnost nám poskytuje možnost používat ten samý set několikrát, takže čtení nebude probíhat tak jednoduše, jak v jednocestné. Za účelem mého a vašeho psychologického zdraví, uvěřte na slovo vzorec pro missy

počet bloků, do kterých se umístí pole + rozdíl v blokech mezi cachi a polem
*** (stupeň asociativity + 1)**

Chápu, že zní nesrozumitelně. Spočítejme příklad:

Za prvé, potřebujeme počet bloků, do kterých se vleze naše pole celkem. Pro to potřebujeme počet slov v bloku. Jednoduché výpočty:

(32 * 8) / 32 = 8 slov v bloku
2264 / 8 = 283 bloků pro celé pole

Za druhé, potřebujeme rozdíl v blokech mezi cahci a polem:

(8 * 1024) / 32 = 256 bloků na celý cache

Za třetí, spočítáme rozdíl:

283 – 256 = 27 bloků

Finální výpočet:

283 + 27 * (2 + 1) = 364 missů

Hity počítáme bez problémů jako obvykle:

2264 * 2 – 364 = 4164 hitů.

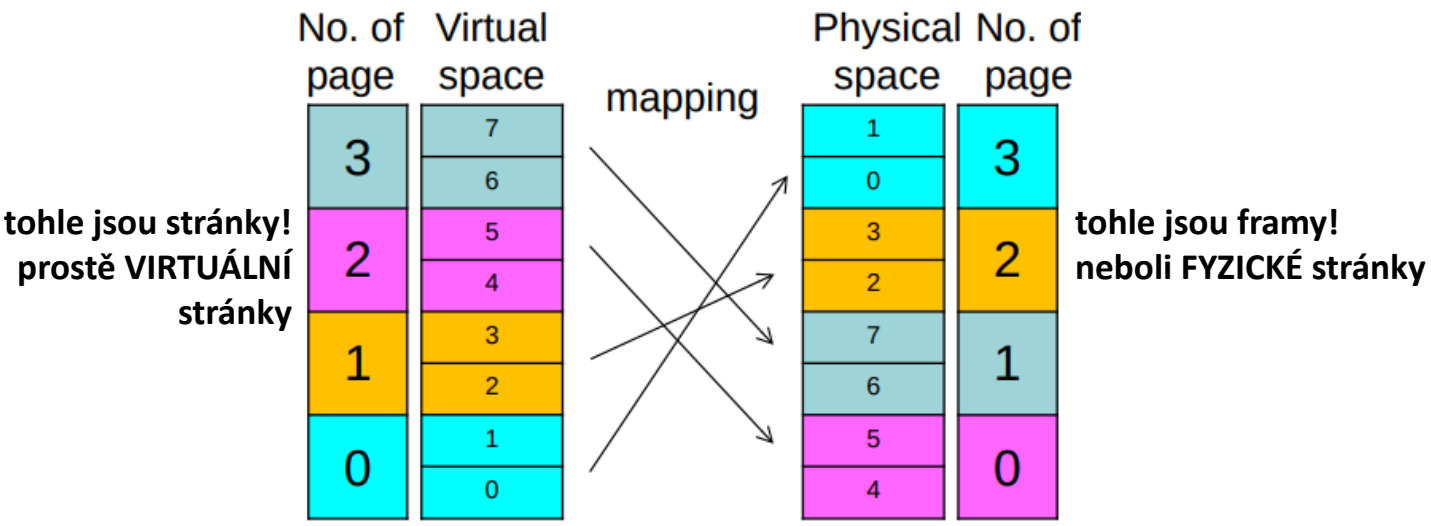
Vše!

Stránkování dokonce ani vtipů nebude

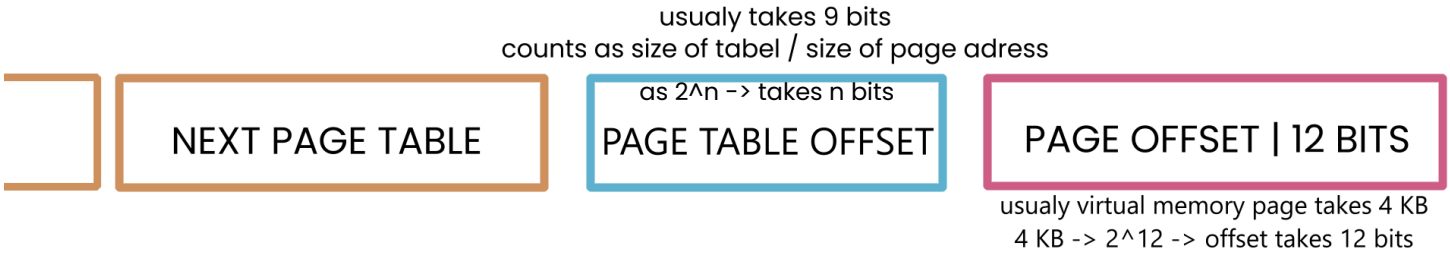
To, co bylo ještě na začátku, ale jsem si jistá, že 90% z vás neví, o co jde. Tedy co to jsou za stránky? Pokud přeskočíme teoretickou část o tom, jak je důležitý dělit paměť na stránky, a z těch stránek skládat další stránky-tabulky, to naše stránky potřebujeme k tomu, aby procesy se nahádali o buňky paměti. Tady se objevuje pojem *virtuální* a *fyzické* adresy. Úplné vysvětlení toho zázraku najdete v pdf od Píšy na stránce 54. Pro líní je tady moje vysvětlení. Operační paměť je omezený zdroj, ale potřebujeme je nějak dělit mezi procesy. Navíc občas je potřebujeme více, než máme

Řešením je *virtuální paměť*.
Virtuální paměť a virtuální adresy je omezeným kouskem paměti, který se poskytuje procesu operačním systémem. V mezích tohoto kousíčku se program vykonává. Virtuální paměť nám poskytuje možnost nestydatě napálit proces a podstrčit data ze sekundárních uložišť (třeba pevný disk) místo RAM. Proč můžeme? Program není vybíravý: řekl jsi „Máš 4 GB, pracuj“, tak program si myslí, že má 4GB nepřetržitě, s kterými on pracuje. I kašle na to, že jsi organizoval přístup sofistikovaným způsobem. Při tom proces nepotřebuje všechna data každou chvíli, proto větší část z nich může ležet ne v drahocenné RAM na perských kobercích, ale v garáži i vytahovat se jen v případě potřeby. Díky tomu se nemusíme furt starat o to, že se RAM skončí nebo že dva procesy náhodou přistupují ke stejné buňce paměti. Nedokážou přistoupit, protože pracuji *pouze se svými virtuálními adresami.*
Jak ale můžou pracovat takové virtuální adresy? Díky laskavému bloku ovládání paměti, který rozstrká virtuální adresy do adres fyzických a pomáhá načítat informaci z *fyzické adresy* pro náš procesor. Moment předávání paketu s informací ze sekundárních uložišť virtuálnímu prostoru se jmenuje *stránkování*

Navíc, zasunout libovolný kousek fyzické adresy do libovolné virtuální paměti je fantastika a utopie. My takhle nemůžeme. Ale stále potřebujeme dělat něco, proto používáme ~~berlu~~ technologii *mappingu*. Dělíme celou virtuální paměť na stejné kousíčky, které se jmenují *stránky*, a celou fyzickou taky dělíme na stejné kousky. Poslední kousky mají rozměr shodný se stránkami, ale jmenují se *framy* (od ang. frame). Mezi framama my i skáčeme. tohle jsou stránky!



Bohužel, to není vše. Stránek je hodně, musíme je nějak hledat. Za tímto účelem byly vytvořeny *Page Table*, uchovávající adresy stránek v paměti (Jo jo, téhle tabulky leží v RAM, stejně jako i stránky). Tabulky se můžou shromažďovat do dalších tabulek, a je to taková vtipná matrjoška.
Dost teorie, podívejme se, jak vypádá virtuální adresa a jak jemu rozumět.



Libovolná lineární (v našem případě virtuální, ale s rázným názvem) adresa se bude skládat z offsetu stránky (nižší bity) a offsetu tabulky. Výpočet je na schématu, ale praktický příklad spočítáme dle zadání.

<u>ÚLOHA</u>: 16-bitový umožňuje adresaci jednotlivých byte, organizace paměti je little-endian (připomínání: little-endian znamená, že nahoře jsou nižší byty. Big-endian – nahoře jsou vyšší byty. Detaily jsou u Štěpána), velikost stránky spravované jednotkou správy paměti je 16 byte a jedna položka 2-úrovňové tabulky pro překlad virtuální adresy zabírá právě 2 byte. Přitom předpokládejte, že jak velikost adresářové části tabulky, tak vlastní tabulky odkazů na stránky odpovídá právě jedné stránce. Tabulky se začínají číst na adrese 0x040. Procesor čte virtuální adresu 0x00DE

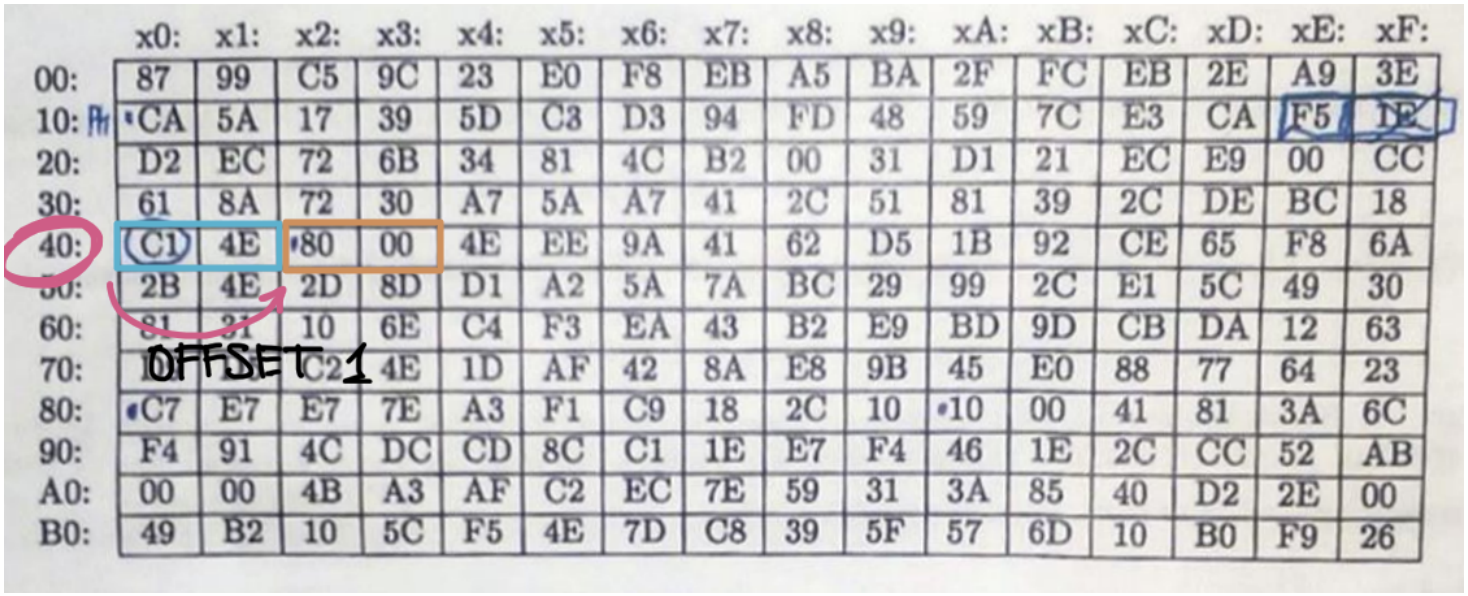
Okay, máme virtuální adresu 0x00DE. Zase si pripomínáme, že Lego je skvělé a začínáme ho rozebírat cihličky.

<u>Za prvé</u>: rozměr stránky je 16 bytů. Na stránce (Pozor, ne v tabulce, ale na stránce!) budeme počítat jednotlivé byty, protože procesor dovolil. Tedy náš offset se spočíta jako obvykle 16 -> 2^n -> 2^4, takže si offset bere 4 bity.
<u>Za druhé</u>: hledáme offset po tabulce. Rozměr je taky 16 bytů, ale přistupujeme k 2 bytům! Takže počet položek 16 / 2 = 8, což je dva na třetí, proto offset po tabulce zaplní 3 bity.
<u>Za třetí</u>: tabulky jsou 2-úrovňové. Tedy máme dvě tabulky (tabulka stránek a tabulka tabulek). Takže i v adrese budeme mít 2 takové položky.

Celkem naše virtuální adresa má 10 bit. Skvělý? Skvěly.
Vrátíme se k 0x00DE. Binárně *0000 0000 1101 1110* což je
0000 0000 1101 1110 z tabulky tabulek, tabulky stránek a stránky

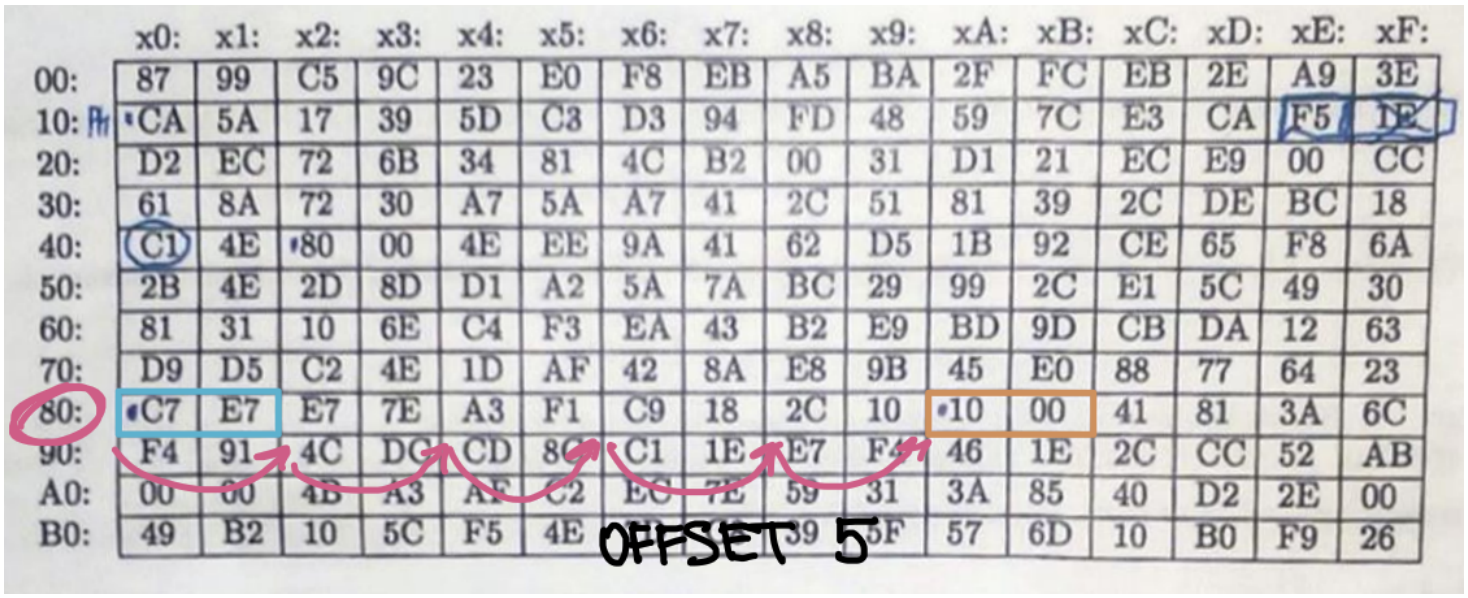
Doufám, že jste všechno pochopili z binárních čísel a je zřejmé, že máme 1, 5, a 14 jak offsety. Teď nejjednodušší část: přebíráme tabulku dle offsetů

Krok jedna: vstáváme na výchozí adresu (dle zadání to je 0x040) a posouváme se dle offsetu tabulky tabulek. Připomínám, že se posouváme po 2 byty!



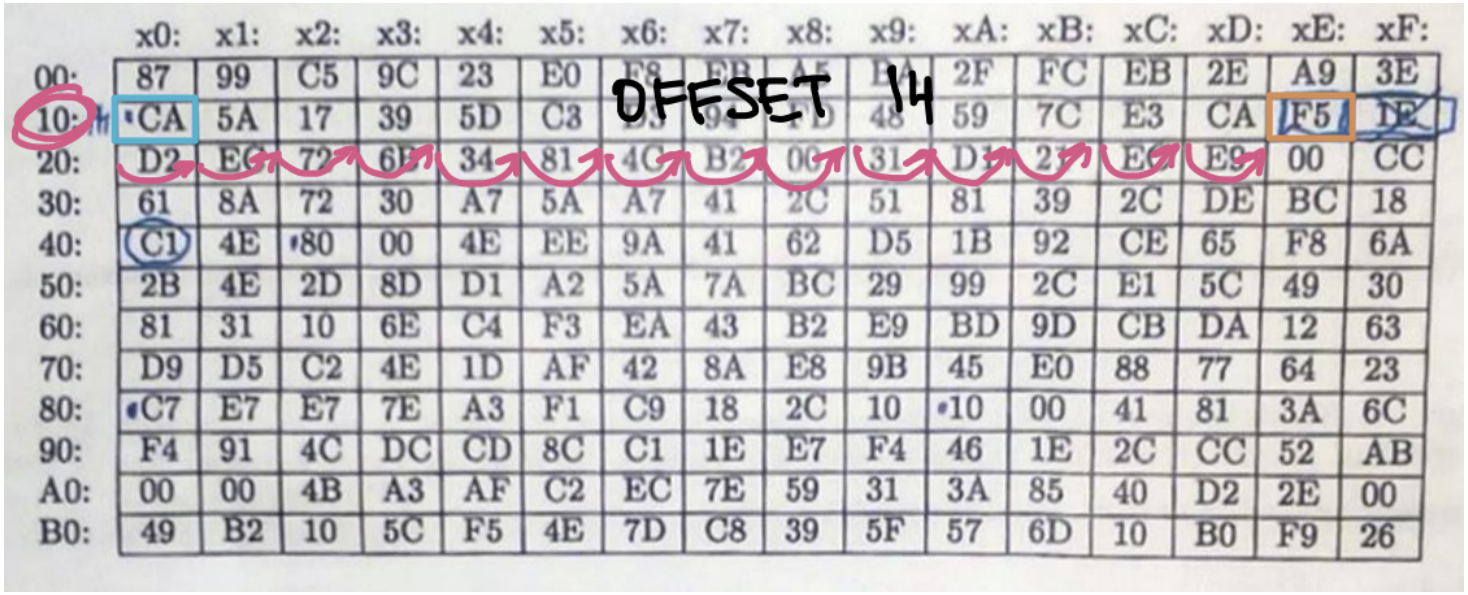
	x0:	x1:	x2:	x3:	x4:	x5:	x6:	x7:	x8:	x9:	xA:	xB:	xC:	xD:	xE:	xF:
00:	87	99	C5	9C	23	E0	F8	EB	A5	BA	2F	FC	EB	2E	A9	3E
10:	CA	5A	17	39	5D	C3	D3	94	FD	48	59	7C	E3	CA	F5	DE
20:	D2	EC	72	6B	34	81	4C	B2	00	31	D1	21	EC	E9	00	CC
30:	61	8A	72	30	A7	5A	A7	41	2C	51	81	39	2C	DE	BC	18
40:	C1	4E	80	00	4E	EE	9A	41	62	D5	1B	92	CE	65	F8	6A
50:	2B	4E	2D	8D	D1	A2	5A	7A	BC	29	99	2C	E1	5C	49	30
60:	[illegible]		10	6E	C4	F3	EA	43	B2	E9	BD	9D	CB	DA	12	63
70:	[illegible]		C2	4E	1D	AF	42	8A	E8	9B	45	E0	88	77	64	23
80:	C7	E7	E7	7E	A3	F1	C9	18	2C	10	10	00	41	81	3A	6C
90:	F4	91	4C	DC	CD	8C	C1	1E	E7	F4	46	1E	2C	CC	52	AB
A0:	00	00	4B	A3	AF	C2	EC	7E	59	31	3A	85	40	D2	2E	00
B0:	49	B2	10	5C	F5	4E	7D	C8	39	5F	57	6D	10	B0	F9	26

Vidíme [80][00], ale pamatujeme si, že jsme v systému little-endian a otáčíme byty. Dostáváme 0x0080, vstáváme na tuto adresu. Teď se posouváme po tabulce stránek s offsetem 5.



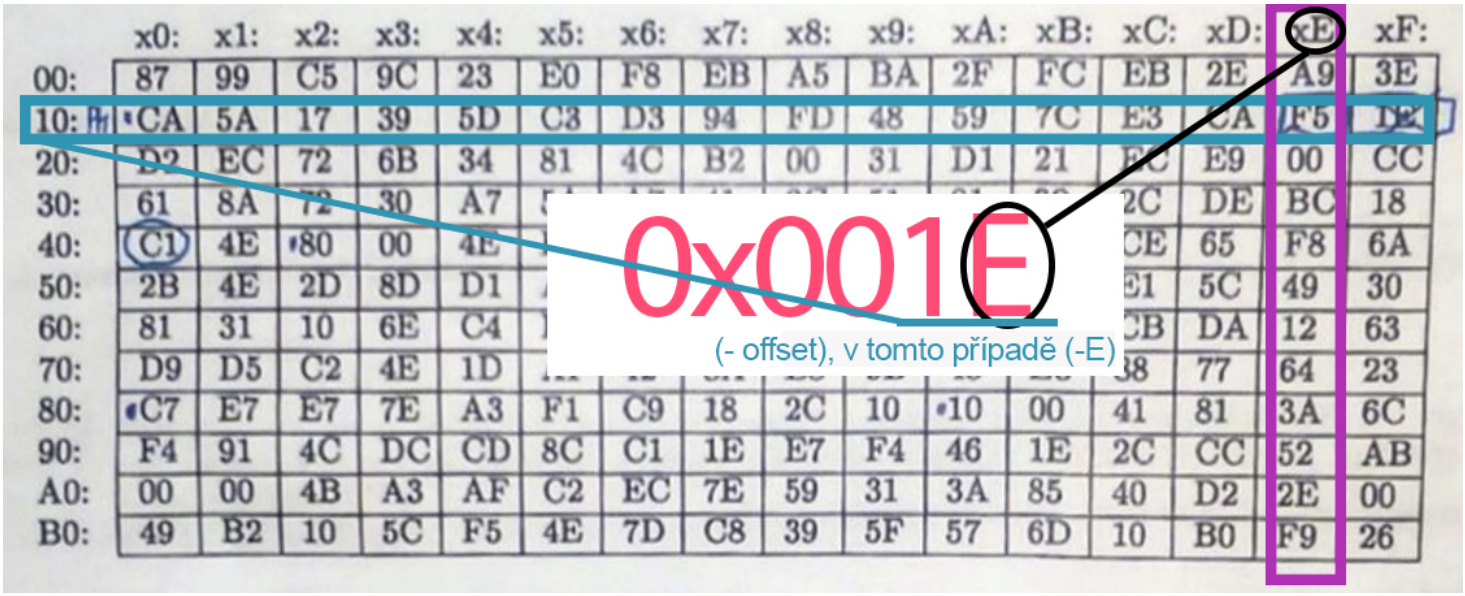
	x0:	x1:	x2:	x3:	x4:	x5:	x6:	x7:	x8:	x9:	xA:	xB:	xC:	xD:	xE:	xF:
00:	87	99	C5	9C	23	E0	F8	EB	A5	BA	2F	FC	EB	2E	A9	3E
10:	CA	5A	17	39	5D	C3	D3	94	FD	48	59	7C	E3	CA	F5	DE
20:	D2	EC	72	6B	34	81	4C	B2	00	31	D1	21	EC	E9	00	CC
30:	61	8A	72	30	A7	5A	A7	41	2C	51	81	39	2C	DE	BC	18
40:	C1	4E	80	00	4E	EE	9A	41	62	D5	1B	92	CE	65	F8	6A
50:	2B	4E	2D	8D	D1	A2	5A	7A	BC	29	99	2C	E1	5C	49	30
60:	81	31	10	6E	C4	F3	EA	43	B2	E9	BD	9D	CB	DA	12	63
70:	D9	D5	C2	4E	1D	AF	42	8A	E8	9B	45	E0	88	77	64	23
80:	C7	E7	E7	7E	A3	F1	C9	18	2C	10	10	00	41	81	3A	6C
90:	F4	91	4C	DC	CD	8C	C1	1E	E7	F4	46	1E	2C	CC	52	AB
A0:	00	00	4B	A3	AF	C2	EC	7E	59	31	3A	85	40	D2	2E	00
B0:	49	B2	10	5C	F5	4E	7D	C8	39	5F	57	6D	10	B0	F9	26

Dostáváme 0x0010 a jdeme na procházku po stránce. Pozor, tady přistupujeme k jednotlivým bytům!



	x0:	x1:	x2:	x3:	x4:	x5:	x6:	x7:	x8:	x9:	xA:	xB:	xC:	xD:	xE:	xF:
00:	87	99	C5	9C	23	E0	[illegible]				2F	FC	EB	2E	A9	3E
10:	CA	5A	17	39	5D	C3	[illegible]				59	7C	E3	CA	F5	DE
20:	D2	EC	72	6B	34	81	4C	B2	00	31	D1	21	EC	E9	00	CC
30:	61	8A	72	30	A7	5A	A7	41	2C	51	81	39	2C	DE	BC	18
40:	C1	4E	80	00	4E	EE	9A	41	62	D5	1B	92	CE	65	F8	6A
50:	2B	4E	2D	8D	D1	A2	5A	7A	BC	29	99	2C	E1	5C	49	30
60:	81	31	10	6E	C4	F3	EA	43	B2	E9	BD	9D	CB	DA	12	63
70:	D9	D5	C2	4E	1D	AF	42	8A	E8	9B	45	E0	88	77	64	23
80:	C7	E7	E7	7E	A3	F1	C9	18	2C	10	10	00	41	81	3A	6C
90:	F4	91	4C	DC	CD	8C	C1	1E	E7	F4	46	1E	2C	CC	52	AB
A0:	00	00	4B	A3	AF	C2	EC	7E	59	31	3A	85	40	D2	2E	00
B0:	49	B2	10	5C	F5	4E	7D	C8	39	5F	57	6D	10	B0	F9	26

Hurá, dostali jsme byte [F5]. Gratuluju! Dozvěděli jsme, co leží na této virtuální adrese. Mimochodem fyzická adresa je 0x001E (dle souřadnic obdélníku [F5]).
Korektura, která byla vnesena po kontrole textu na reálném člověku: pokud nerozumíte, jak se tvoří fyzická adresa, nakreslila jsem ještě jedné schéma pro vás :)



	x0:	x1:	x2:	x3:	x4:	x5:	x6:	x7:	x8:	x9:	xA:	xB:	xC:	xD:	xE:	xF:
00:	87	99	C5	9C	23	E0	F8	EB	A5	BA	2F	FC	EB	2E	A9	3E
10:	CA	5A	17	39	5D	C3	D3	94	FD	48	59	7C	E3	CA	F5	DE
20:	D2	EC	72	6B	34	81	4C	B2	00	31	D1	21	EC	E9	00	CC
30:	61	8A	72	30	A7								2C	DE	BC	18
40:	C1	4E	80	00	4E								CE	65	F8	6A
50:	2B	4E	2D	8D	D1								E1	5C	49	30
60:	81	31	10	6E	C4								CB	DA	12	63
70:	D9	D5	C2	4E	1D								38	77	64	23
80:	C7	E7	E7	7E	A3	F1	C9	18	2C	10	10	00	41	81	3A	6C
90:	F4	91	4C	DC	CD	8C	C1	1E	E7	F4	46	1E	2C	CC	52	AB
A0:	00	00	4B	A3	AF	C2	EC	7E	59	31	3A	85	40	D2	2E	00
B0:	49	B2	10	5C	F5	4E	7D	C8	39	5F	57	6D	10	B0	F9	26

Sloupečky. Jsou. Offsetem! Proboha, x na začátku není 0x na začátku čísla šestnáctkové soustavy. x znamená, že se číslo zleva mění dle řádku. Doufám, že teď to je jasný :3

Sláva bohu, to je konec. Pro finále jen doplním, že pro zvětšení nebo zmenšení velikosti zpracovávané virtuální adresy my přidáváme nebo odebíráme úrovni (tabulky tabulek). Třeba teď jsme měli zpracovávanou adresou 10 bitů, ale z nicnedělání jsme chtěli 16. Připomínáme si, že tabulkový offset potřebuje 3 bity. Rozumíme, že schází nám 2 tabulky do štěstí a 16 bitů. Proto i děláme tabulky 4-úrovňovymi.

Nyní to je určitě konec. Oddechněte si a nevzdávejte se. Dále máme minutku (v reálu minutek 10) čilu :)

Pipeline, hazardy a forvardy. To, co budete zbožňovat za lehké body

Takže, připomínejme si, čím se zabývá procesor. Minimálně čte instrukce a rozšiřovává, co je v ni psáno, a vykonává to. Ale jako vždy existuje problém: tyhle procesy se konají různou dobu, přičemž rozdíl je významný.
To je smůla, a zobecnění je protivníkem všeho, co se týče algoritmizace, takže jednotlivy takt dělíme minimálně na 5 fází:

1) Čtení instrukce z pamětí – *Fetch Instruction (IF)*
Na této etapě procesor tahá binární kód instrukce z cache paměti.

2) Dekódování instrukce – *Decode Instruction (ID)*
To je důležitý, pokud se vyrušili, vraťte pozornost zpět. První, co dělá váš procesor, je určení, co tento binární kód instrukce znamená. Jestli jste velice zvídaví a nedokážete žít bez rozumění všeho, můžete se zde seznámit, jak instrukce vypadá v binárním tvaru a jaký druhy jsou. Avšak to je zbytečný pro naše účely.
To, čemu musíte rozumět: na této etapě procesor bere všechny hodnoty z registrů/konstant a předává je do ALU. Proč tu informaci potřebujeme, dozvíte trošku později.

3) Vypočítání – *Execute Instruction (EX)*
Tady na scénu vychází ALU a vykonává své matematické-logické funkce. Není co říci navíc.

4) Přistup k paměti – *Memory Access (ME nebo MEM)*
Přistupujeme k paměti: píšeme do ní nebo čteme z ní slova

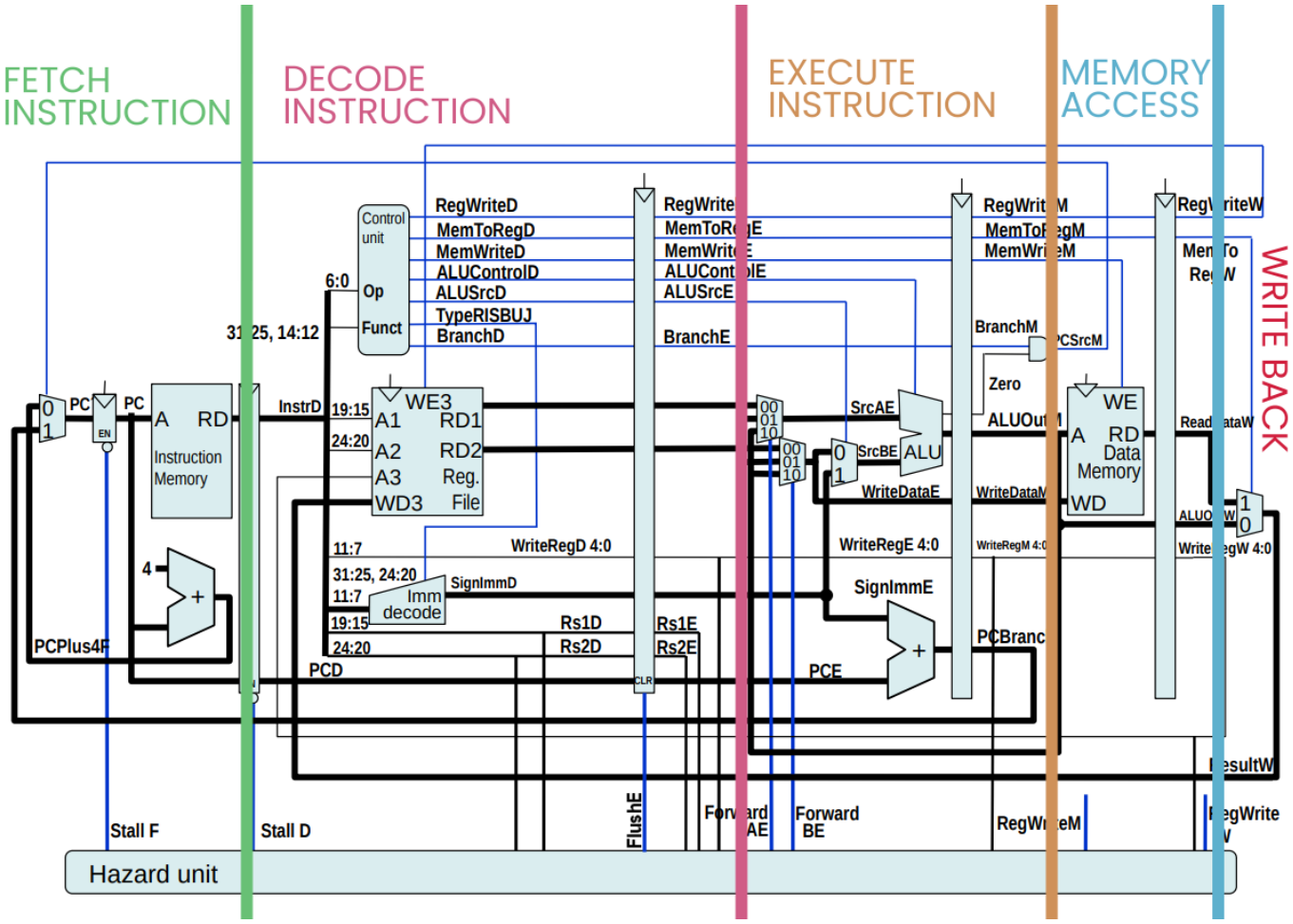
5) Zápis hodnot do registrů – *Write Back (WB)*
Pozor, není to zápis do paměti. To je zápis přesně do registrů! Proto to i je „back“, neboť vracíme hodnoty registrům.

Pokud váš mozek je stále s vámi, gratuluji, první úroveň obtížnosti jste zvládnuli. Stojí za zmínění, že v moderních procesorech vše procesy jsou rozděleny ještě víc. Ale my jsme podpivasnici, proto nám stačí i pět.

Té pět etap se vykonávají v každé instrukci. Každý takt procesoru se vykonává jedna etapa a vykonávají se po řadě. Takže na jednu instrukci potřebujeme 5 taktů, nezní to až tak strašně. Co ale když máme 125 instrukcí? Tady již je zřejmá neefektivita.
Dámy a pánové, vítejte pipeline!

Pipeline je náš bro, který nám poskytuje možnost šetřit si čas. Díky němu procesor může pracovat jako běžící pás. Jednoduše řešeno, části procesoru fungují nezávisle. Jedna část čte instrukci, druhá ji dekóduje, přeskočíme ještě dva kroky vysvětlováni, a poslední píše hodnoty do registrů.

Schématické rozdělení vypadá nějak takhle:



Ať jsme přečetli první instrukce, začali jsme dekódování, ale zóna fetch stojí bez práci. To je špatně a to je to, co řeší pipeline. Jakmile se zóna osvobodí (třeba pro fetch to znamená, že se úspěšně vykonává dekódování) ona začíná vykonávat následující činnost. Ostatní etapy se chovají analogicky. Tedy v romantické a utopické představě máme všechny instrukce umístěny tak, že nikdy nedochází k chybám (proč to není tak, dozvíme za chvíli), a pipeline funguje takhle:

cykly procesoru ->

	1	2	3	4	5	6	7	8	9
Instruct. 1	**IF**	**ID**	**EX**	**ME**	**WB**				
Instruct. 2		**IF**	**ID**	**EX**	**ME**	**WB**			
Instruct. 3			**IF**	**ID**	**EX**	**ME**	**WB**		
Instruct. 4				**IF**	**ID**	**EX**	**ME**	**WB**	
Instruct. 5					**IF**	**ID**	**EX**	**ME**	**WB**

Doufám, že toto schéma je pro vás srozumitelná (dělali jste třetí domácí úkol, že jo?). V případě, že stále máte otázky, můžete se vrátit k přednášce.
Problémy s pipelinem vznikají tam, kde zkoušeme načíst hodnoty registrů, které se ještě nestihli aktualizovat. Nejlépe se to vysvětluje tabulkama, než slovama.

cykly procesoru ->

	1	2	3	4	5	6	7	8	9
addi $9, $0, $11	**IF**	**ID**	**EX**	**ME**	**WB**				
and $10, $10, $9		**IF**							

Co se děje: jako první se klidně čte a vykonává instrukce add – prosté sčítání hodnot registru $11 a $0, výsledek se píše do *registru $9*. Dobře, to není těžký, ale co se děje dále? Dále se čte instrukce and, která potřebuje hodnotu *registru $9*… Ale my jsme ani nespočetli hodnotu registru, ani nejde o zápis (podívejte se do tabulky a na číslo taktu) a dekóder nemá co si vzít. Smůla, škoda.
V tabulce je vidět, že nutná hodnota se objeví v registru jen na 5. cyklu (žlutý čtverec). No, v daném primitivním pipelinu nemůžeme udělat nic, než čekat (čekání se označuje slovem *stall*)

cykly procesoru ->

	1	2	3	4	5	6	7	8	9
addi $9, $0, $11	IF	ID	EX	ME	WB				
and $10, $10, $9		IF	*stall*	*stall*	ID	EX	ME	WB	
ori $8, $0, 0xA7					IF	ID	EX	ME	WB

Jakmile se tam objeví hodnota (takt číslo 5) provedeme dekódování (zelený čtverec) a půjdeme dál. To taky dovolí procesoru načíst další instrukci.

Takovým způsobem i funguje primitivní pipeline, který nemá fičuru s názvem forwarding. Pokud dostanete takový, jste štístko. Ostatní jsou nuceni rozebrat, co to je za forwarding a co dělá skvělého.
Vraťme se k našim etapám. Pokud si vzpomenete, co přesně dělají, lze se dovtípit, že hodnoty ve skutečnosti jsou hotové již na konci EX. To jest na etapě ME a tím spíš i na WB bysme dokázali používat data, kdybysme dokázali číst hodnoty nejen z registrů. A to můžeme, pomocí *forwardingů*.

Táhle fičura nám dovolí zmenšovat počet stallů, a proto i zvýšit rychlost zpracovávání jednotlivých instrukcí. Více i nepotřebujeme =)
Výborné vysvětlení, jak to funguje se schématy a analýzou schématu procesoru můžete ohodnotit tady. My ale zkusíme přidat forwarding k předchozímu příkladu.

cykly procesoru ->

	1	2	3	4	5	6	7
addi $9, $0, $11	IF	ID	EX	ME	WB		
and $10, $10, $9		IF	*ID*	EX	ME	WB	
ori $8, $0, 0xA7			IF	ID	EX	ME	WB

Viděli jste, viděli? Prostě jsme přečetli hodnotu registru $9 z předchozího výpočtu a $10 jsme převzali z fáze dekódování. My jsme, přesněji procesor s pipelinem, pašáci.

Kde lze udělat takovou fintu nelze, přesněji lze ne bez problémů? Především tam, kde se nic nespočítá a ALU nic nedělá. Pokud jste se dívali na přednášku pána Štěpána, již víte, že se jedná o *sw* a *lw*.

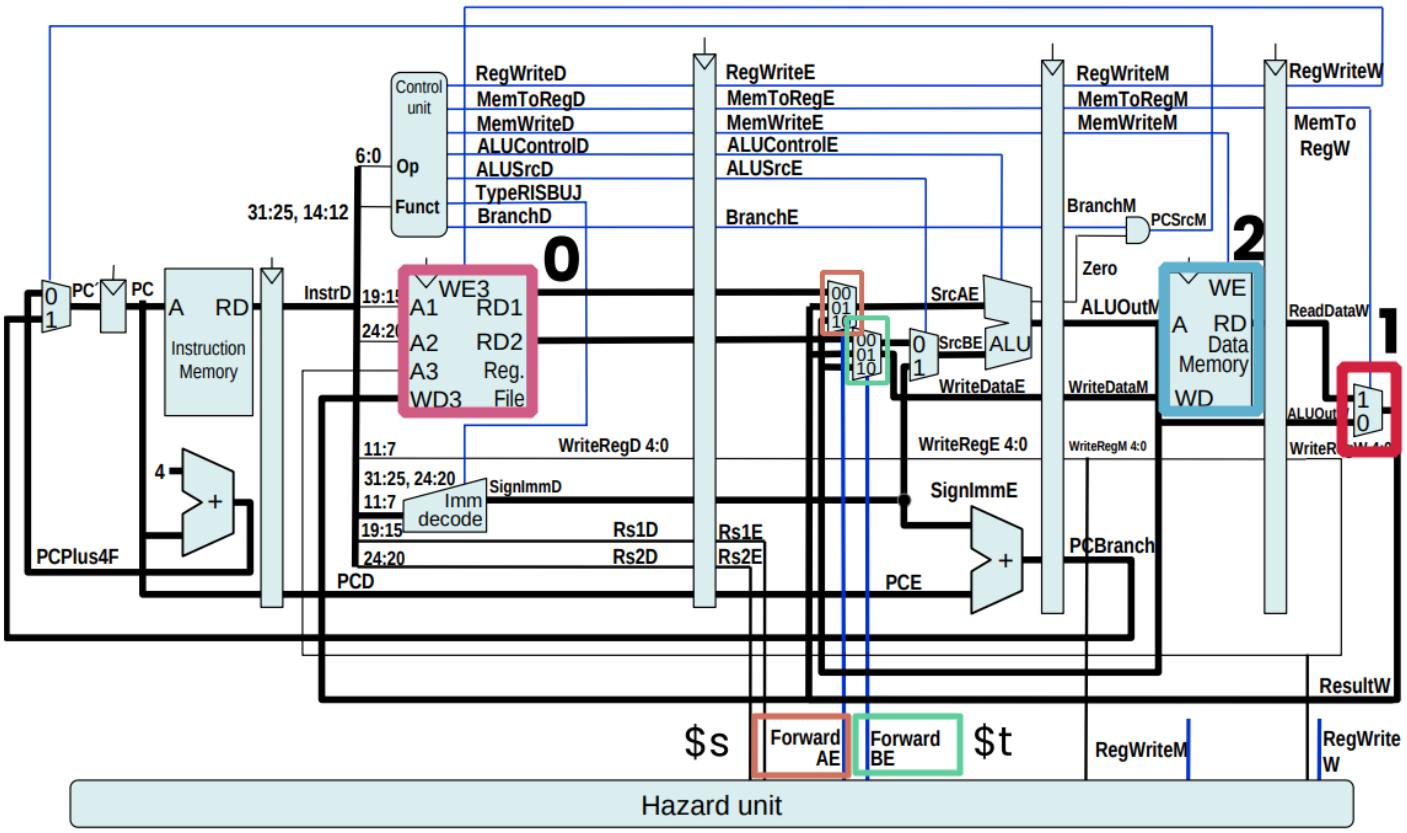
Prozkoumejme příklad s *lw*. Laskavě připomínám, že lw čte data z paměti do registru. To jest slovo bude nalezeno jen po etapě ME, a používat je smíme po WB

cykly procesoru ->

	1	2	3	4	5	6	7
lw $12, 0x400($0)	IF	ID	EX	ME	WB		
and $12, $12, $8		IF	*ID*	*stall*	EX	ME	WB

Ať tak, či onak jsme zkrátili jeden stall, ale potřebujeme čekat na slovo. Celkem, to je vše z teoretické časti. Nebylo až tak strašně, že? Říkala jsem, pipeline je váš bro. Milujte pipeline.

Vše, co zůstává, je řešení úlohy z testu, takže pojďme na to.
Máme úlohu a v ní obrázek:



Nápisy a barevné obdélníky na zkoušce nebudou, to jsou moje poznámky pro vás. O co se jedná? O Forwardingu. Kvůli tomu, že o forvardingu, musíme rozlišovat, odkud ten forwarding bereme (vlastně tohle i je úloha). Prostě řečeno, z jaké etapy dostáváme hodnoty.
Pamatujeme si:
a) *varianta 0*, znamená to, že jsme dobře zvládnuli i bez fint, prostě jsme použili z *ID*, nebo hodnoty registru nepoužíváme.
b) *varianta 1* znamená, že díru s hodnotami nám zavřel *WB*
c) *varianta 2*. Nepodařilo se vyřídit pomocí *ID*, ale *ME* nás zachránil

Dále, co znamená *Forward AE* a *Forward BE*? Upřímně řečeno, negarantuji správnost, ale sedí to s logikou: horní se píše vlevo od spodního v instrukci. Pokud registr-operand je jeden, je to spodní operand (SAKRA POZOR, poznámka na konci kapitoly). Abysme se pletli, co a kde se používá, díváme se do Pišove tabulky instrukcí. Připomínám: první registr v instrukci slouží pro výsledek! Není to operand!

ZADÁNÍ: pochopit, k jakým forwardingům dojde při vykonávání řady instrukcí v zeleném obdélníku.

Pozice	Kód	Instrukce
InstA	00854022	sub t0,a0,a1
InstB	00084fc3	sra t1,t0,31
InstC	20030000	addi v1,zero,0
InstD	01284824	and t1,t1,t0
InstE	00a91020	add v0,a1,t1
InstF	200a0000	addi t2,zero,0

pole	bity	instrukce	operace
opcode	26 ... 31	add $d,$s,$t	$d = $s + $t;
$s	21 ... 25	addi $t,$s,C	$t = $s + C;
$t	16 ... 20	sub $d,$s,$t	$d = $s - $t
$d	11 ... 15	slt $d,$s,$t	$d = ($s < $t) (set less than)
Csht	6 ... 10	sll $d,$t,Csht	$d = $t « Csht (shift left logical)
func	0 ... 5	srl $d,$t,Csht	$d = $t » Csht (shift right logical)
C	0 ... 15	sra $d,$t,Csht	$d = $t » Csht (shift right arithmetic)

Začínáme tím, že nakreslíme schéma pipelinu pro názornost, a zapíšeme dvojky pro ME a jedničky pro WB pro odpovídající operandy.

	1	2	3	4	5	6	7	8	9	10	AE	BE
Inst A	IF	ID	EX	ME	WB						0	0
Inst B		IF	ID	EX	ME	WB					0	2
Inst C			IF	ID	EX	ME	WB				0	0
Inst D				IF	ID	EX	ME	WB			1	0
Inst E					IF	ID	EX	ME	WB		0	2
Inst F						IF	ID	EX	ME	WB	0	0

Třeba v instrukci B potřebujeme operand $t0, který se používal před tím a dostaneme ho na etapě ME. V instrukci D potřebujeme $t1, který se naposledy objevoval v instrukci B a k momentu potřeby již byl na WB. S E situace je analogická s B na začátku: $t1 dostaneme z ME předchozí instrukce.
Gratuluji, vojáci! Přežili jste pipeline a já doufám bez velkých ztrát.
Pokračování slibuje nebejt tak dlouhým, takže kdybyste byli polekaní pětkou stránek pipelinu, uklidněte se =)

PROŠÍM POZOR!!!
Já NEJSEM Píša. Nemám ponětí, co měl v hlavě, když psal úlohy. V tomhle příkladu, konkretně V TOMHLE, jestli operand-registr je jeden, to je spodní. Prostě to takhle je. Přitom v mé variantě a variantách jiných lidí on byl HORNÍM.
Já vám důrazně doporučuji si vzít instrukci add $d $s $t a podle něj pochopit, co je AE a co je BE a opírat se na Píšův schéma. Vlevo (kde teď je $s) je AE, vpravo je BE.

Meh formuli, ale jenom trošičku. Rychlost vykonání programu

Pamatujete si, že jsme posuzovali cache a někde mimochodem připomínali že všechno je ke slávě rychlosti? Teď to náhledně spočítáme. Doporučuju během počtu se podivovat a vykřikovat „No to je něco!“, aby se alespoň nějak zdůvodnila nutnost této akce

<u>ÚLOHA</u>: Máme nějaký procesor s frekvenci 1GHz, který vyplní 1000 instrukci. Vykonaní instrukce trvá pravě jeden tik. Předpokládáme, že se jednou obrátit k paměti trvá 100 ns, a víme že v 25% případů musíme do paměti lézt. Kolik bude trvat vykonání programu?

Skvělá otázka, zkusíme to vyřešit.
Za prve, pro všechny, kdo též nepamatuje: 1 gigahertz označí, že 1 tik trvá právě jednu nanosekundu. Čílí vyrazí frekvence
Pak máme: 25% od 1000 instrukci budou trvat každá 100 nanosekund.

(1000 * 0.25) * 100 = 25000 nanosekund

Ostatní 75% zaberou 1 nanosekund pro sebe

(1000 * 0.75) * 1 = 750 nanosekund

<u>Celkem 25750 nanosekund</u>

Pokud není pochopitelné, jestli to mnoho nebo málo, zkusíme přidat k procesoru cache a podíváme se na změny

<u>ÚLOHA POKRAČOVÁNÍ</u>: Dodáme k procesoru cache s miss rate 50%. Obrácení na cache trvá 10 ns

V klidu, mám pro vás dobre noviny. Pro 75% instrukci se nic nemění, stějně 750 ns
Ale co s týmy 25%?
Za prve vše z 25% instrukci procesor bude hledat v cache

(1000 * 0.25) * 10 = 2500 nanosekund

Ale daří se nám pouze v polovině případů. To znamená, že pro 50% od 25% budeme muset navíc lézt v paměť

((1000 * 0.25) * 0.5) * 100 = 12500 nanosekund

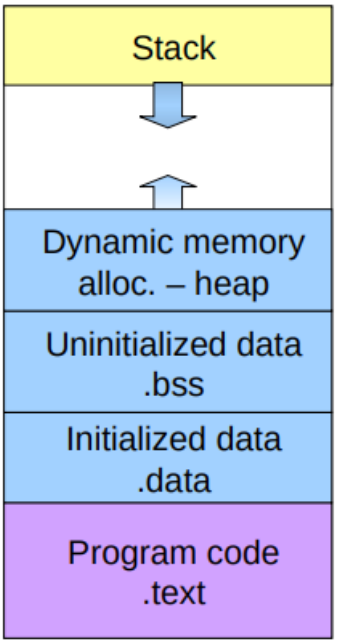
<u>Celkem 15750 nanosekund</u>

V srovnaní s výsledkem bez cache je jasně zrychlení, že?

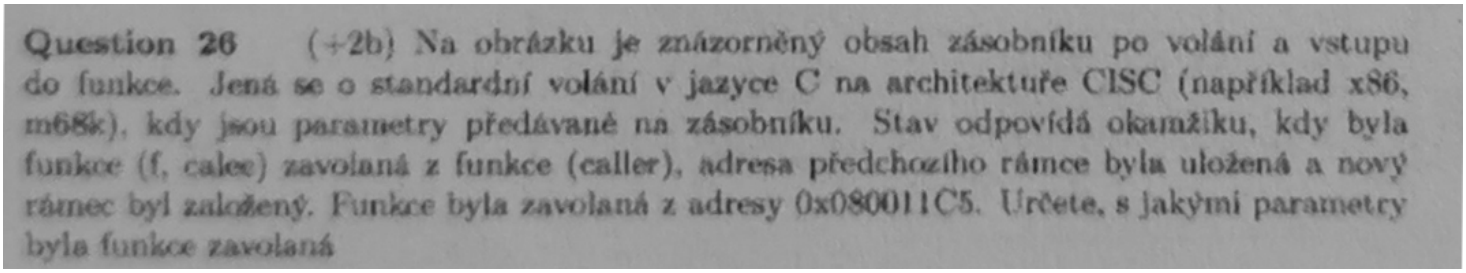
To, co využíváme ještě od prvního cviká z C, ale na co jsme kašli – argumenty funkci

Ano, chápu že vy též nerozumíte navošta nam heta. Ale toto je manuál pro podpivasniky, proto bude maximálně podpivasné vysvětlení. Anyway podrobnosti můžete si dozvědět tady. V pdf Píšy (stránka 4) jsem našla obrázek který zřejmě rozjasní všechno.

Pamatujete zásobník? Ano, přímo do něho se skládají argumenty funkce. Jak je vidět na obrázku, skládají se shora k dolu, ale tady je malá finta: zásobník visí hlavou dolu. Čili ho tail leží výš něž ho hlava. Tehdy první vložený element bude na samým konci zásobníku ale nahoře pamětí, poslední element naopak – dole v paměti ale ve hlavě zásobníku.
Funguje zásobník jak hromádka talířů – nelze prostě vzít cokoliv uprostřed, lze jenom vzít horní element (myslí si element ve hlavě zásobníku)
Takovou frontu nazýváme LIFO neboli Last In First Out.

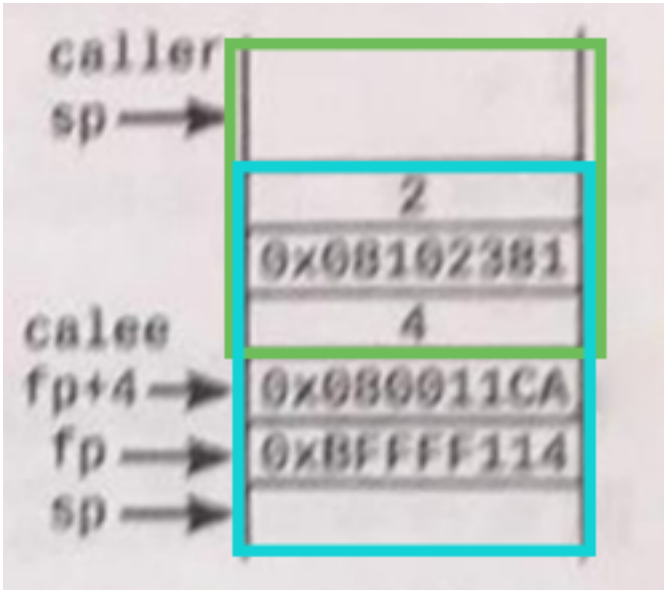


Ale v jakém pořadí se čtou argumenty v závorkách funkci?
Správná odpověď od konce


Question 26 (+2b) Na obrázku je znázorněný obsah zásobníku po volání a vstupu do funkce. Jená se o standardní volání v jazyce C na architektuře CISC (například x86, m68k), kdy jsou parametry předávané na zásobníku. Stav odpovídá okamžiku, kdy byla funkce (f, calee) zavolaná z funkce (caller), adresa předchozího rámce byla uložená a nový rámec byl založený. Funkce byla zavolaná z adresy 0x080011C5. Určete, s jakými parametry byla funkce zavolaná

Celý úkol je nahoře. Krátký smysl: uvnitř funkce caller se vyzví jiná funkce calee a musíme najít jí argumenty. Koukáme na obrázek. Zelený obdélník je framem, který se tyká caller, a světlo-modrý je pro calee.
Argumenty, s kterymi calee byla zavolaná leží ve frajmu nějvýš. Ale držím v hlavě, že hlava zásobníku je zdola a argumenty čteme od posledního ve závorkách

Pak funkce vypadá: *calee(4, něco, 2)*
Něco je pointerem na pole charů, což je string v C.



CPI + MIPS neboli několik vzorců do vašeho mozku

Co to je CPI? Cycles per instruction – střední počet cyklů zatracených pro jednu instrukci. Jak to spočítáme? Zkusíme na konkrétních hodnotách.

ÚLOHA: Máme procesor z minimálním CPI (jinak řečeno počet cyklu v ideálním případě, vůbec bez zpoždění) se rovná 4. Miss rate instrukční cache je 6%, a miss rate informační – 4%.
Když to vysvětlit v lidském jazyce: *procesor, když čte instrukce z cache, v 6% narazí do missu a když bude případně číst libovolně jiné data – miss se nastane v 4% případu*. Za miss v cache je penalizace 100 cyklů procesoru. A od celého počtu instrukcí 25% se zvrátí do paměti.

Momentálně nic není pochopitelně, jenom máme hromadu nějakých čísel. Ale už jsme přežili cache, nemusíme se bát.

Za prve na každou instrukci defaultně utrácíme 4 cykly.
Za druhé cely program čteme instrukce, což znamená že v 100% případů se obrátíme na instrukční cache. 6% od toho budou missy a za každý miss ztrácíme 100 cyklu navíc. Tehdy

(1 * 0.06) * 100 = 6 cyklu v průměru na instrukci

****tady 1 se označí 100%***

Do paměti jdeme pouze v 25% případů a v 4% od toho budou missy

(0.25 * 0.04) * 100 = 1 cyklus

Dohromady CPI = 11 cyklů

A momentálně tady vyřešíme otázku s MIPSem.
***MIPS* nebo *Million instructions per second* — je přibližná hodnota čisté výpočetní výkonosti počítačů. Jak spočítáme? Vyřídíme si to.**

ÚLOHA: Rychlost splnění fáze (latence) v procesoru je:
IF: 9 ns | ID: 6 ns | EX: 35 ns | ME: 6 ns | WB: 3 ns. Přičemž v průměru na 6 užitečných instrukce bude jedna bublinka (kvůli *stall* například). Jaký bude MIPS?

První potřebná hodnota – frekvence procesoru. Zjistíme tak: bereme nejdelší proces, v našem případě to je EX = 35 ns a počítáme, kolikrát vejde do jedné sekundy.

1 / 35 ns = 28,6 MHz(nano je 10^{-9} a mega je 10^{6})

Dal jde finta. Hledáme, kolik instrukcí zvládneme za jeden cyklus. V průměru jasně. Vyplníme podle podmínek 6 z 7 instrukci, jinak řečeno

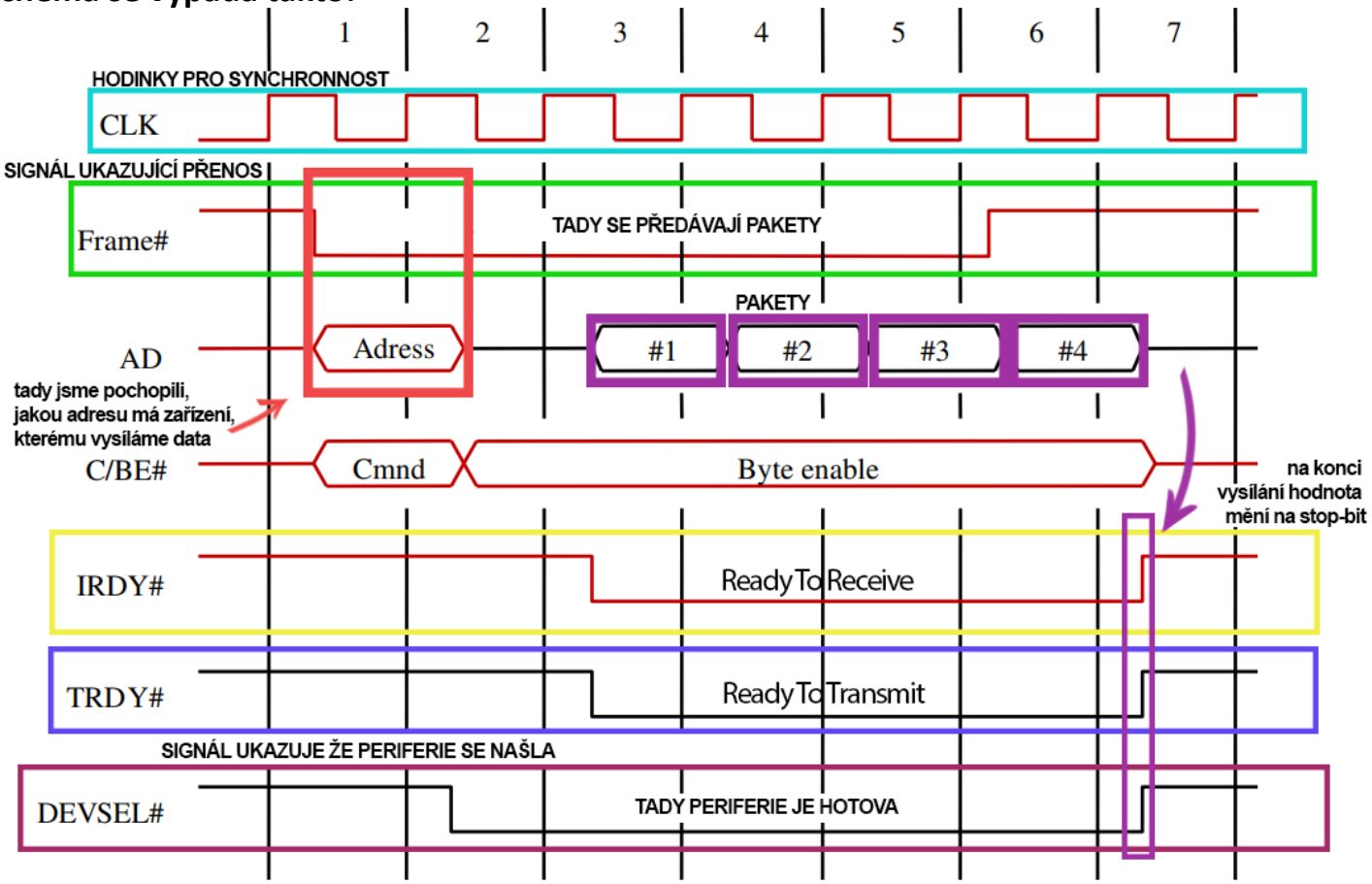
6/7 = 0.86 instrukcí

Pak děláme součin 0*.86 * 28.6 = 24.4 -> 24*. Táto hodnota i bude náš MIPS.

Ебаный свет этой периферии... Bolest, muky a utrpení*

nepřekládaný ruský vyraz znamenající zoufalství od nutností rozumět tématu, v tomto případě periferii.

Co je periferie? To jsou různé zařízení vstupu/výstupu přelepené k výpočetní krabičce vašeho počítače. Proč periferie nás zajímá? Protože komunikuje z výpočetní krabičkou a musejí se nějak domluvit mezí sebou. Tímto úkolem se zabývá PCI – Peripheral Component Interconnect. Taková hezká sběrnice uvnitř vašeho počítače.
Od začátku vám řeknu – tady už šotu tequilly nestačí, bude potřebá cela lahev. Ihned můžete běžet do přednášky Štepána, odkud minimálně musíte pochopit čím je PCI
Schéma se vypadá takto:

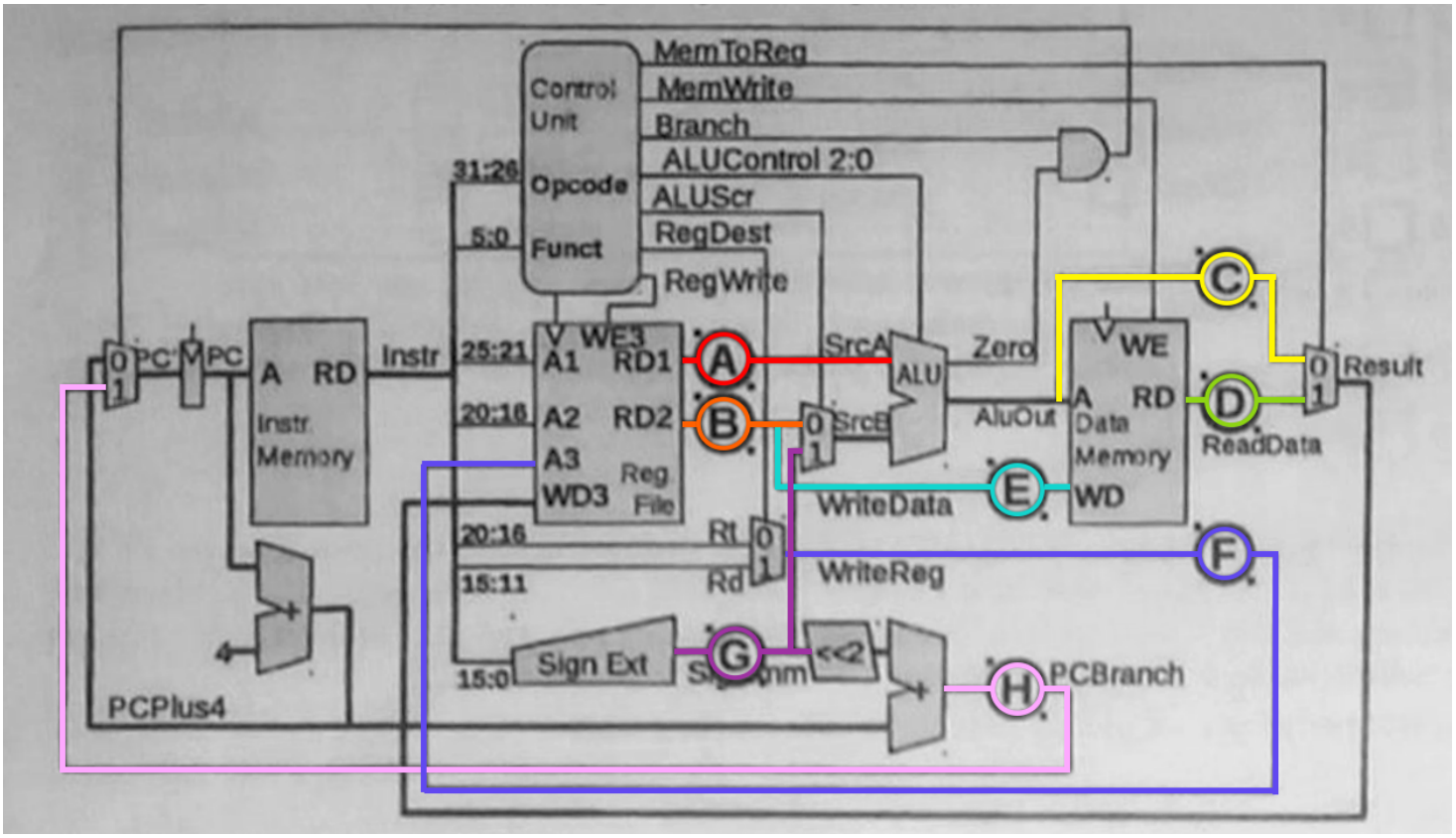


Je vám špatně? Rozumím vám, ale dobrá zpráva je v tom, že takový úkol naposled byl v roce 2019. Anyway, úkol je spočítat, kolik bajtů bylo předáno.
Spočítáme to pomoci počtu lilových paketu. Jejich velikost na schématu bude označen kolem signálu v tvaru (2^{n-1} ... 0). Toto je počet bitů v paketu. Rozdělíme 8kou a dostaneme počet bajt. Pakety budou odeslaný jenom když signály IRDY#, TRDY#, DEVSEL# spolu budou na 0, proto hledáme kolik paketu v takový interval vešlo a vynásobíme počet velikosti paketu. Výsledkem je kolik celkem bajtů bylo odesláno.
Bohužel fotka úkolu je rozmazaná a jediné, co pochopitelné, tak to, že otázky se tykají rolí každého signálu. Všechno jsem rozepsala na schématu.

A s periferii je to všechno. Doufám, že se nikdy znovu s ní nepotkáme.

Úkoly navíc, jaké nebyly probraný v první redakci: signály v procesoru a instrukce

Jak jsme už zjistili, instrukce pro procesor nejsou stejné. A ne každá instrukce bude využívat všechny částky procesoru najednou. Toto je docela logická jednoduchá a srozumitelná myšlenka: doufám, že asi nikdo nemyslí, že součet a čtení slova vyplní úplně stejným způsobem. Proto se od začátku pustíme do praxe.



Máme schéma z nějakými kružnicemi. Pro názornost jsem označila vše barvami. Pochopit odrazu, co se děje v této srande asi nelze, naštěstí každý proces je podepsaný vice méně srozumitelnými zkratkami. V abecedním pořadí:

A je prvním operandem
B vzniká když používáme tři registry (například v add $d, $s, $t kde je tři registry pro výsledek, první a *druhý* operand) neboli používáme dva registry čí hodnoty *se nemění* (instrukce sw)
C cesta po které můžeme obejít blok práce z paměti, kdy tu paměť potřebovat nebudeme a *měníme hodnoty v registrech.*
D cesta přes kterou čteme z paměti
E tady budeme naopak ukládat do paměti
F cesta pro writeback: zápis výsledku do registrů
G znakové rozšíření se objeví jakmile se objeví konstanta. Ahoj addi
H cesta pro vyplnění skoků mezí branci

Zkusíme podle tohoto schématu rozebrat několik instrukce a pochopit co jím bude potřeba.

1) lw $t, C($s)
Instrukce load word zjistí adresu a *čte* z ní slovo *do registru*. Přičemž druhý operand u nás konstanta (nějaká hodnota z $s spolu s offsetem C. Registr $t je pro ukládání čteného slova) což znamená že budeme potřebovat *znamínkové rozšíření*.
Celkem potřebujeme A pro první operand, G pro druhý (protože je konstantou), D pro čtení slova a F pro ukládání do registru.

2) add $d, $s, $t
Instrukce jednoduchého součtu. Máme operand A, operand B, registr pro výsledek, což je writebackem. Paměť nepotřebujeme.
Celkem A, B, C a F

3) sw $t, C($s)
Instrukce store word je trošku podobná lw: zjistí adresu ale *ukládá na ní*, nečte z ní, potřebné slovo. Dryhý operand je stejně konstantou, nezapomeňme na *znamínkové rozšíření*. Writeback se nebude provádět.
Existuje docela nezřejmá nuance – tady budeme potřebovat cestu B, protože máme dva registry ve kterých neměníme hodnoty (v tomto případě z $t bereme hodnotu a nezapisujeme v něho).
Proto potřebujeme A, B, G a E pro zápis slova do paměti

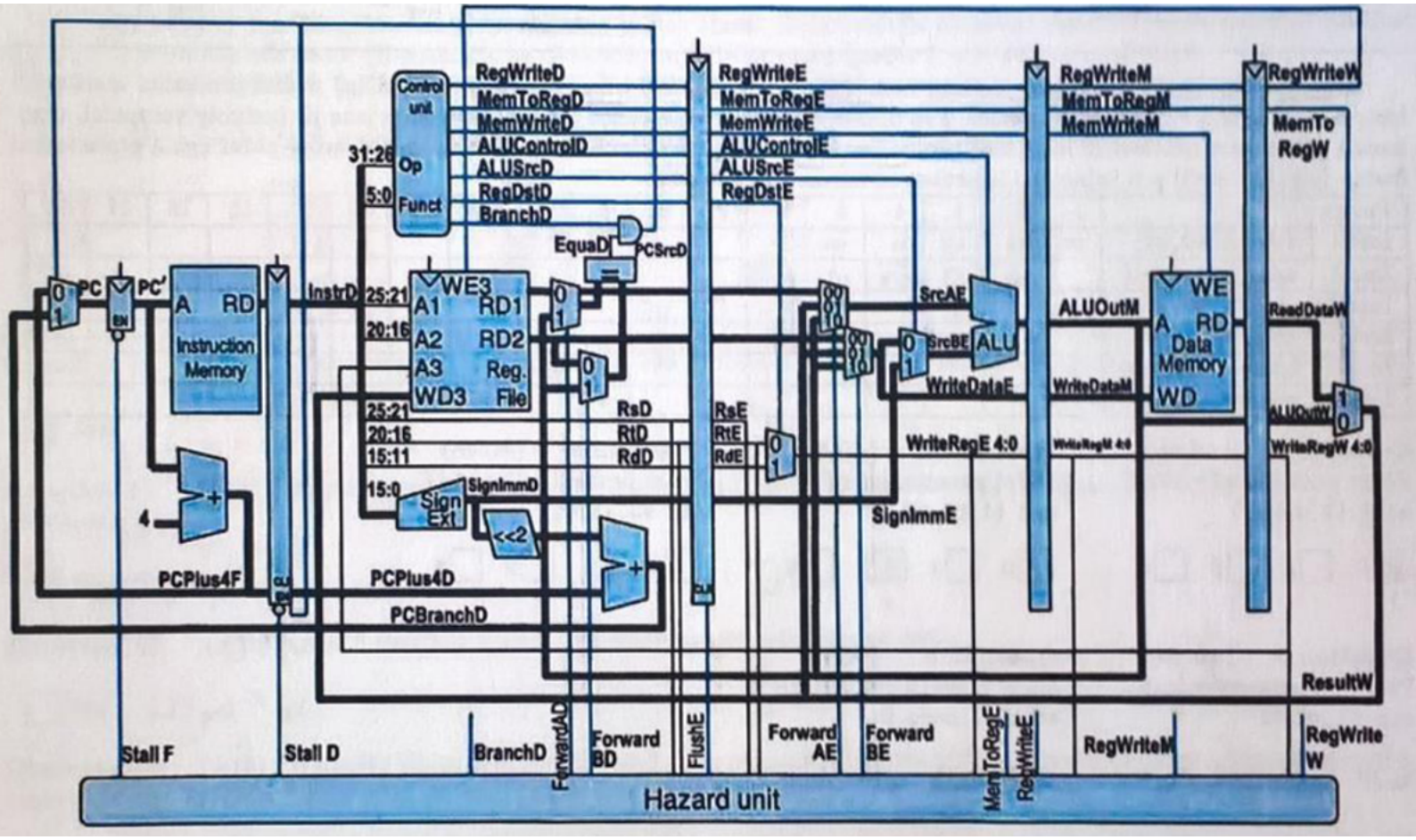
4) slt $d, $s, $t
Slt instrukce která *zapisuje v registr* $d jedničku pravé tehdy když hodnota registru $s je menší než hodnota registru $t, jinak bude zapásaná 0. Máte už dohady co budeme potřebovat?
A, B, C, F. Ano, stejně jak i v součtu.

Jestliže na této etapě jste pochopili, co se děje – gratuluju, můžete klidně s tímto pochopením jít dál. Jestli pro vás vše zní jako delirium tremens – upřímná soustrast, dál to bude mnohem hůř :))

Následující otázka: jaké instrukce procesor nebude schopen vykonávat při následujících podmínkách (podmínky budou uvedený pro každou otázku zvlášť).
Jako podmínky máme zablokované signály.

Pro zvládnutí úkolu zcela stačí pochopit, jaká funkce jaké signály využívá a používá-li jich vůbec. Ale když jste natolik zvídavý, jak je zvídavý můj redaktor, doporučuju alespoň prohlédnout přednášku o instrukcích, prolistat pdf Píší a kouknut na videa, kde se ukazují postupné splnění instrukce procesorem.

Dostanete schéma procesoru. Co s ním dělat? Dávat pozor na obrázek a chápat.



Question jedna: jaké instrukce se nezvládne vykonat procesor, když bude zablokovaný MemToRegW na hodnotě 0?
a) sub $d, $s, $t b) sw $t, C($s) c) add $d, $s, $t
d) lw $t, C($s) e) beq $s, $t, C f) addi $t, $s, C

Za prve přemýšlíme, podle potřeby vodíme prstem po spojením za účelem lepšího pochopení, což dělá MemToRegW? Podle názvu a podle schématu vidíme, že signál se používá pří *zápisu z paměti do registru*. Abysme dozvěděli, co nebude fungovat při zablokovaném signálu, je nutno zjistit jaké instrukce máme a co budou používat pří vykonání (ve všech otázkách jsou stejný instrukce proto popíšu jich jenom jednou a dál budu se vrátit k tomuto popisu):

***add $d, $s, $t* – součet hodnot z registru $s a $t, který je zapsan do registru $d**
***sub $d, $s, $t* – rozdíl hodnot z registru $s a $t zapsaný do registru $d**
***addi $t, $s, C* – součet hodnoty z registru $s a konstanty C zapsaný do registru $t. C bude znaménkově rozšířená.**
***sw $t, C($s)* – ukládání hodnoty (slova) z $t do adresy $s+C**
***lw $t, C($s)* – čtení hodnoty (slova) do $t z adresy $s+C**
***beq $s, $t, C* – vložení do PC (registr Program Counter který drží adresu momentálně konající instrukce) hodnoty C (bude znaménkově rozšířená) právě tehdy, když se hodnoty $s a $t rovnají.**

Podle popisu je zřejmě, kde píšeme něco *z paměti* do registru (pozor z pa-mě-ti, ne z ALUčka nebo jiných registru): pouze v *lw*. Pak naši odpověď je *d) lw $t, C($s)*

Question dva: Co bude při zablokovaném ALUOutW?
a) lw $t, C($s) b) sw $t, C($s) c) add $d, $s, $t
d) sub $d, $s, $t e) addi $t, $s, C f) beq $s, $t, C

První krok se zopakuje: přemysleme. ALUOutW -> výsledek *z ALU pro zápis*. Kde to potkáme Varianty *c) add $d, $s, $t, d) sub $d, $s, $t, e) addi $t, $s, C*.
Možná náhodně chcete ukázat i sw, ale prosím, nechte ho v klidu. sw *nepoužívá* pro ukládání žádné hodnoty z ALU.

Question tři: Co poškodí SignExt, když bude vždy nastavovat nejstarší bity na 0?
a) beq $s, $t, C b) sw $t, C($s) c) sub $d, $s, $t
d) lw $t, C($s) e) add $d, $s, $t f) addi $t, $s, C

Znamínkové rozšíření*. Kde ho máme? Tam, kde máme konstanty: *a) beq $s, $t, C, b) sw $t, C($s), d) lw $t, C($s), f) addi $t, $s, C.

Question čtyří: Co ovlivní zablokovaný na 0 WriteDataE?
a) addi $t, $s, C b) sub $d, $s, $t c) beq $s, $t, C
d) lw $t, C($s) e) sw $t, C($s) f) add $d, $s, $t

Ukládáme data do pamětí* pouze v jednom případě: *e) sw $t, C($s)

Question pět: ALUSrcE nefunguje, pak nefungujou..?
a) beq $s, $t, C b) addi $t, $s, C c) sw $t, C($s)
d) sub $d, $s, $t e) lw $t, C($s) f) add $d, $s, $t

ALUSrcE se nastaví na jedničku, když se v ALU vyplní součet s konstantou, ne s hodnotou jiného registru. Toto potřebují addi, sw a jeho kamarád lw, a chápu že taková sada může vehnat do slepé uličky. Můžete informace ověřit na stránkách 30-31 přednášky. Blokuje ALUSrcE na hodnotě 0 blokujeme instrukce *b) addi $t, $s, C, c) sw $t, C($s), e) lw $t, C($s)*.

Question 3+3(pozor ne 9): Co nejsme schopný vykonat při zablokovaným RegWriteE?
a) lw $t, C($s) b) beq $s, $t, C c) add $d, $s, $t
d) sub $d, $s, $t e) sw $t, C($s) f) addi $t, $s, C

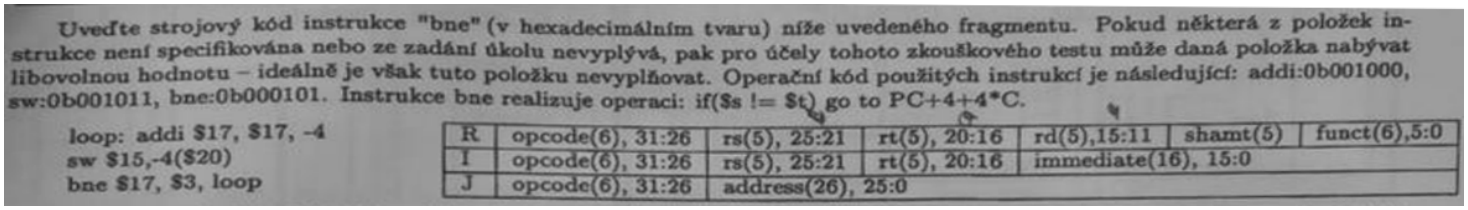
Kde něco píšeme do registru? Správné, ve variantech a) lw $t, C($s), c) add $d, $s, $t, d) sub $d, $s, $t, f) addi $t, $s, C. Proč ne v beq? Protože registr PC je zvláštní a kontroluje se zvlášť.

Dal budeme potřebovat ForwardBD. Zbystřete pozornost a zjistěte kam on vede (pozor nespleťte s ForwardBE! ForwardBD je víc nalevo).

Question sedm: Máme zablokovaný ForwardBD, pak nefungují..?
a) lw $t, C($s) b) addi $t, $s, C c) beq $s, $t, C
d) add $d, $s, $t e) sub $d, $s, $t f) sw $t, C($s)

Když správně našli ForwardBD, tak chápete, že vede nás k komparátoru, terý potřebujeme pouze v *c) beq $s, $t, C*.

Sláva bohu tento typ úkolu je za námi. Ale ještě chvilku vás zdržím na této kapitole a probereme ještě jeden úkol tykající se instrukce. S vaším dovolením začátek na screenshotu:



Uveďte strojový kód instrukce "bne" (v hexadecimálním tvaru) níže uvedeného fragmentu. Pokud některá z položek instrukce není specifikována nebo ze zadání úkolu nevyplývá, pak pro účely tohoto zkouškového testu může daná položka nabývat libovolnou hodnotu – ideálně je však tuto položku nevyplňovat. Operační kód použitých instrukcí je následující: addi:0b001000, sw:0b001011, bne:0b000101. Instrukce bne realizuje operaci: if($s != $t) go to PC+4+4*C.

```
loop: addi $17, $17, -4
sw $15,-4($20)
bne $17, $3, loop
```

R	opcode(6), 31:26	rs(5), 25:21	rt(5), 20:16	rd(5),15:11	shamt(5)	funct(6),5:0
I	opcode(6), 31:26	rs(5), 25:21	rt(5), 20:16	immediate(16), 15:0		
J	opcode(6), 31:26	address(26), 25:0				

A otázka: jak bude vypadat v hexadecimálním tvaru instrukce bne.
Za prve zjistíme že bne *je instrukce I-typu*. Všechny instrukce v MIPSu jsou velikosti 32 bit (4 bajty) ale mají různou strukturu. I-typ vypádá tak:

kde IMM konstanta. Kód instrukce bne je 5 *(000101)*. Registr $rs v našem případě je registr $17 *(10001)* a $rt je $3 *(00011)*
Celkem nejstarších 16 bitů budou vypadat:

000101 10001 00011

neboli v hexadecimálním tvaru 1623(každé 4 bity je jedním hexadecimálním číslem)

A co s konstantou? Tady potřebujeme využit vše své znalosti matematiky a pochopit, že hledaný posun je o dva řádky. Každá instrukce se ukládá v 4 bajty. Celkem potřebujeme *8 bajt nazpátky.* Na konající instrukce ukazuje PC. A z podmínek máme formuli

PC + 4 + (4 * C)

Víte co je C? Konstanta kterou hledáme. Vzpomínáme si že umíme sčítat a píšeme

PC - 8 = PC + 4 + (4 * C)

C = - 3

Na konci potřebujeme -3 nějak zapsat v binárním tvaru. Vrátíme se k první kapitole a podle ní převodíme

1 000 0000 0000 0011 přímý kód
1 111 1111 1111 1100 inverzní kód

1 111 1111 1111 1101 dvojkový doplněk kód
Neboli hexadecimálně FFFD

Celkem hexadecimálně 0x1623FFFD

Znovu MIPS, i znovu vypočítat instrukce..

No, jako i vy, mám toho po krk. Ale tato část bude aspoň rychlou.
<u>ÚLOHA</u>: Máme procesor, v kterém se všechny instrukce se na 4 typy: A, B, C, D a každý typ má svůj CPI

Частота CPU	CPI A-type	CPI B-type	CPI C-type	CPI D-type
1GHz	5	5	5	1

Pak víme že program se skládá z 20% instrukce typu A, 30% instrukce typu B,
10% typu C и 40% D. Co nakonec budeme mít v MIPSu (*Million instructions per second*)?

Tady se objeví na naší party druhá varianta spočtu MIPS: ne přes počet instrukce, ale přes CPI. V podstatě jednoduchá věc:

MIPS = Frekvence CPU / CPI programu,
A frekvenci potřebujeme v MHz!

Od začátku víme frekvenci 1 GHz -> 1000 MHz. A co to je CPI programu?

Máme 4 typy instrukce a CPI se u nich lisí. Když máme jejich procentovou relace, která chvála matematice se spolu dává 100%, můžeme spočítat

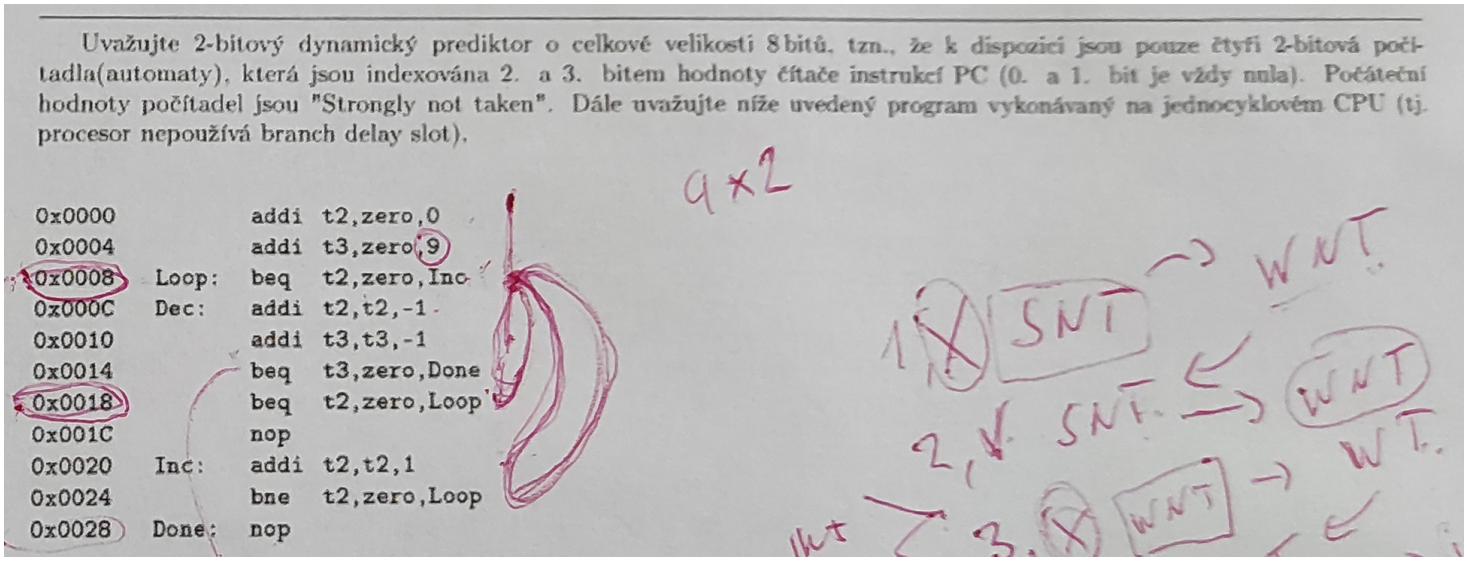
(0.2 * 5) + (0.3 * 5) + (0.1 * 5) + (0.4 * 1) = 3.4 cykly pro instrukci
Dosazujeme do vzorce na začátku:
MIPS = 1000 / 3.4 = 294

Skvělé! Teďka uděláme to obraceně: chceme najít, jak dlouho bude trvat program o 10^9 instrukce. Už víme CPI programu: 3.4. Frekvence 1 GHz a cyklus trvá pravé 1 ns. Sčítáme:

t = (CPI * čas vykonání cyklu)* počet instrukce = (3.4 * 1) * 10^9 = 3.4 * 10^9 ns
v sekundách -> 3.4 s

Blahopřeji k nejmarnější útratě vašeho času: predikce skoku

Jestli jste nevěděli, jak vypadá věštění z kávové sedliny, to je ono. Problematika úkolu tkví v tom, že procesor není senzibil, i když se dostává na branch, přesně neví, jestli bude dělat jump nebo prostě půjde dál po listu instrukci. Proto následující instrukce on preferuje načíst předem, aby zkrátil dobu vykonaní programu. I to jakoby pro nás je dobrou správou, dík mu za to, ale problémy se začínají od okamžiku, když budeme muset všechny skoky spočítat. Proč? Zeptejte se na to pana Píšu. Text úlohy:



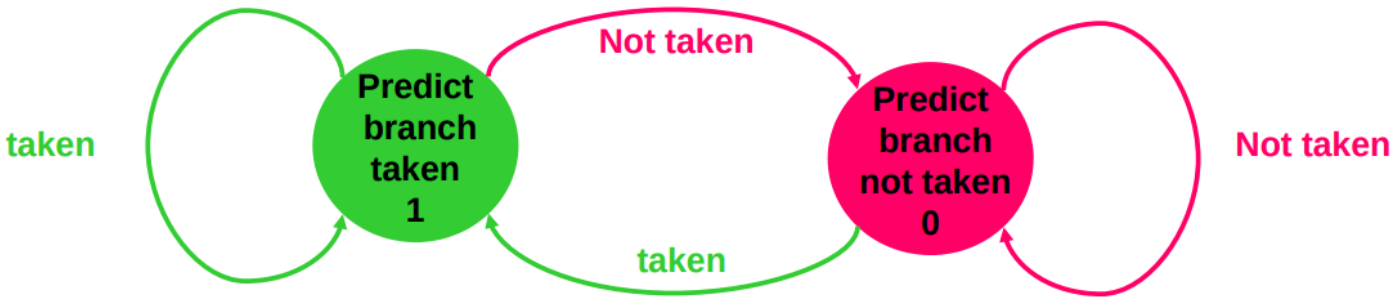
Uvažujte 2-bitový dynamický prediktor o celkové velikosti 8 bitů, tzn., že k dispozici jsou pouze čtyři 2-bitová počítadla(automaty), která jsou indexována 2. a 3. bitem hodnoty čítače instrukcí PC (0. a 1. bit je vždy nula). Počáteční hodnoty počítadel jsou "Strongly not taken". Dále uvažujte níže uvedený program vykonávaný na jednocyklovém CPU (tj. procesor nepoužívá branch delay slot).

```
0x0000          addi  t2,zero,0
0x0004          addi  t3,zero,9
0x0008  Loop:   beq   t2,zero,Inc
0x000C  Dec:    addi  t2,t2,-1
0x0010          addi  t3,t3,-1
0x0014          beq   t3,zero,Done
0x0018          beq   t2,zero,Loop
0x001C          nop
0x0020  Inc:    addi  t2,t2,1
0x0024          bne   t2,zero,Loop
0x0028  Done:   nop
```

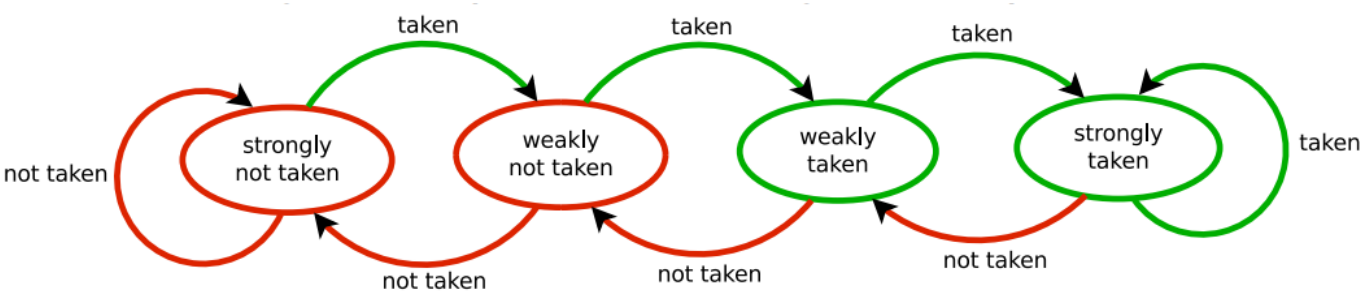

Máme dvoubitový dynamický prediktor o velikosti 8 bit. Jinak řečeno, máme 4 dvoubitových prediktorů. Co to vůbec je prediktor a proč z jednoho prediktoru o 8 bit přes slovo uděláme čtyři o 2? Dobrá otázka a podrobnou odpověď jako vždy můžete (asi) najit v přednášce. Ale začneme vyklad, náhle, od začátku.

Kdy my(procesor) narazíme na branch, máme jenom dvě varianty: pokračovat v kódu nebo skákat. Ale když se vrátíme k vyprávění o tom, že každý cyklus procesoru se vyplní pouze jedná fáze vykonání konkrétní instrukce, to zjistíme, že uvázli na mrtvém bodě: v prvním cyklu (IF) najdeme skok, avšak nebudeme vědět, jestli ho budeme vykonat. To je smutná věc, protože nejsme rádi vidět stall a čekat zbytečně cykly, minimálně nechceme čekat vždycky. A s tím nám pomůže prediktor.

Jednodušší to bude vysvětlit na přikladu jednobitového rediktoru, navíc nikdo negarantuje, že na zkoušce na něho nenarazíte.



V jednobitovém prediktoru život je opravdu jednoduchý: existuje stav *taken* (kdy procesor je si jistý, že skok bude) a stav *not taken* (obratná situace: myslí si, že skok nebude). Přepínaní mezi stavy zcela zřejmě zobrazený na obrázku na konci předchozí stránky.
Když se nachází ve stavu *taken* (1 v bitovém tvaru) procesor v následujícím cyklu na fáze IF přečte instrukci, na kterou ukazuje skok. Zřejmě: při *not taken*(0) na IF bude přečtena následující po branche instrukce.
Je docela možné, že teď tato sranda připadá moc abstraktně nebo vůbec neužitečně, ale pověřte – dokonce i primitivní prediktor má svou roli. Praktické, pro řešení tohoto úkolu potřebujeme dvoubitovou verze neboli *saturating counter.*



Logika se nemění, mění pouze počet stavu: zvětší až na čtyři, kde ve dvouch (*Strongly not taken и Weakly not taken*) budeme pokračovat v kódu a při ostatních (*Weakly taken и Strongly taken*) vezmeme instrukci, na kterou ukazuje skok.
Zcela krátká teoretická část se tady končí. Vrátíme se k tomu, že máme 4 dvoubitové prediktory a začínají od stavu Strongly not taken.
Jako předpoklad máme kód v assembleru, ve kterém jsou hrozné různých adres, ale nás se ptají jenom o 0x0008. Máme 4 prediktory, adres výrazně víc. Co děláme? Lepíme několik adres k jednomu prediktoru. Pozor, k prediktoru se lepí ne branch-řádky kódu, ale adresy! Když máte otázku „what the…?“ – odpověď bude na konci kapitoly.

Spolu z 0x0008 k danému prediktoru se přilepí navíc adresy 0x0018 a 0x0028. I když 0x0028 nedělá nám bordel (naštěstí na ní není žádný skok), tak 0x0018 svým beq ovlivní výpočty.

t2 = 0	0																		
t3 = 9	9																		
beq t2, 0, Inc	T		NT	T		NT	T		NT	T		NT	T		NT	T		NT	
t2--; t3--;			0 8			0 7			0 6			0 5			0 4			0 3	
beq t3, 0, Done			NT			NT			NT			NT			NT			NT	
beq t2, 0, Loop			T			T			T			T			T			T	
t2++		1			1			1			1			1			1		
bne t2, 0, Loop		T			T			T			T			T			T		

beq t2, 0, Inc	T		NT	T		NT	T		NT
t2--; t3--;			0 2			0 1			0 0
beq t3, 0, Done			NT			NT			T
beq t2, 0, Loop			T			T			
t2++		1			1			1	
bne t2, 0, Loop		T			T			T	

Koukáme do tabulky na předchozí stránce. Používaný registry t2 a t3, kolem kterých se motají skoky. Přičemž jsou 4 instrukce skoku a mají pouze dva stavy: skok se vyplní(v tabulce odznačeno jako T) nebo nevyplní se(NT). Čteme tabulku zleva napravo.
Jestli narazíme na T – přecházíme na následující sloupec.
Zkusíme rozebrat od začátku. Pří běhu programu t2 a t3 se přirazí k 0 a 9 patřičně. Pak dojdeme k beq na adrese 0x0008 kde se srovnají hodnota t2 s 0, i když se rovnají, vyplní se skok do neznáma k Inc (adresa 0x0020). Na Inc k hodnotě t2 se přečte 1 a pak vyplní se jiný skok na adrese 0x0024 podle instrukce bne. Tento skok nás vrátí na Loop (0x0008) kde už skok nebude vyplněn. Jdeme dal, zmenšíme t2 a t3 o jedničku, přeskočíme první beq, ale na následující (adresa 0x0018) se vyplní a skokem znovu vrátí na Loop.
A opakujeme tento cyklus 8-krát pokud t3 nebude se rovnat 0 a program se skonči. Teď kreslíme novou abrovskou (otázky ohledně pravopisu plz k redaktorovi) tabulku:

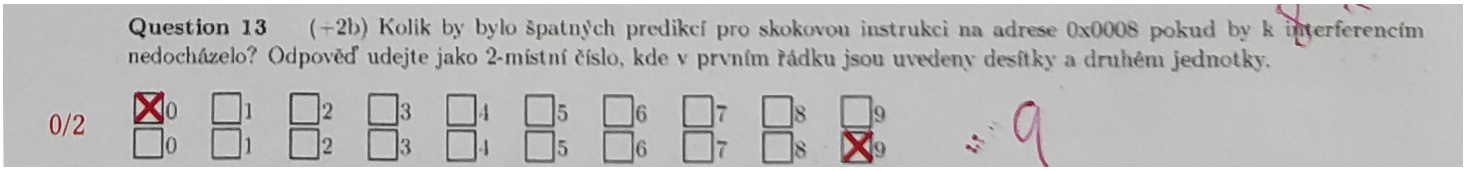
Taken/not taken	Stav prediktoru	Vztah predikce a prediktoru
T	SNT	X
N	WNT	O
T	SNT	X
T	WNT	X
N	WT	X
T	WNT	X
T	WT	O
N	ST	X
T	WT	O
T	ST	O
N	ST	X
T	WT	O
T	ST	O
N	ST	X
T	WT	O
T	ST	O
N	ST	X
T	WT	O
T	ST	O
N	ST	X
T	WT	O
T	ST	O
N	ST	X
T	WT	O
T	ST	O
N	ST	X

Do prvního sloupce zapisujeme, jestli se skok vyplní na adresách 0x0008(lilová barva) a 0x0018(žluta barva). Ve druhém sloupci zapisujeme stav prediktoru v okamžik, kdy jsme se dostali na tuto branch(na začátku bude Strongly not taken(SNT)). V třetím se nachází výsledek predikce kde X se označuje špatnou predikci, O správnou.

Stav prediktoru zaleží na předchozí řádkech! Jinak řečeno, k stavu předchozího řádku „přečítáme“, jestli skok byl vyplněn (také se koukáme na předchozí řádek). To je pochopitelně nakresleno na obrázku prediktoru.

T	+	SNT	=WNT
N		WNT	

Z této tabulky můžeme spočítat za prvé, kolikrát byla splněná instrukce z adresy 0x0008(spočítáme kolik lilových řádku, celkem 18), za druhé – kolik bylo špatných predikci (počítáme křížky (X) na lilových řádcích, takých bude 10).
Jestli dál přečíst úkol, najdeme tam až dvě otázky tykajících se špatných predikci. Kde druhá otázka:


Question 13 (+2b) Kolik by bylo špatných predikcí pro skokovou instrukci na adrese 0x0008 pokud by k interferencím nedocházelo? Odpověď udejte jako 2-místní číslo, kde v prvním řádku jsou uvedeny desítky a druhém jednotky.

0/2 ☒0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ☐8 ☐9
☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ☐8 ☒9

Tady, znovu, potřebujeme novou tabulku, ale pouze(!) pro adresu 0x0008

Taken/not taken	Stav prediktoru	Vztah predikce a prediktoru
T	SNT	X
N	WNT	O
T	SNT	X
N	WNT	O
T	SNT	X
N	WNT	O
T	SNT	X
N	WNT	O
T	SNT	X
N	WNT	O
T	SNT	X
N	WNT	O
T	SNT	X
N	WNT	O
T	SNT	X
N	WNT	O
T	SNT	X
N	WNT	O

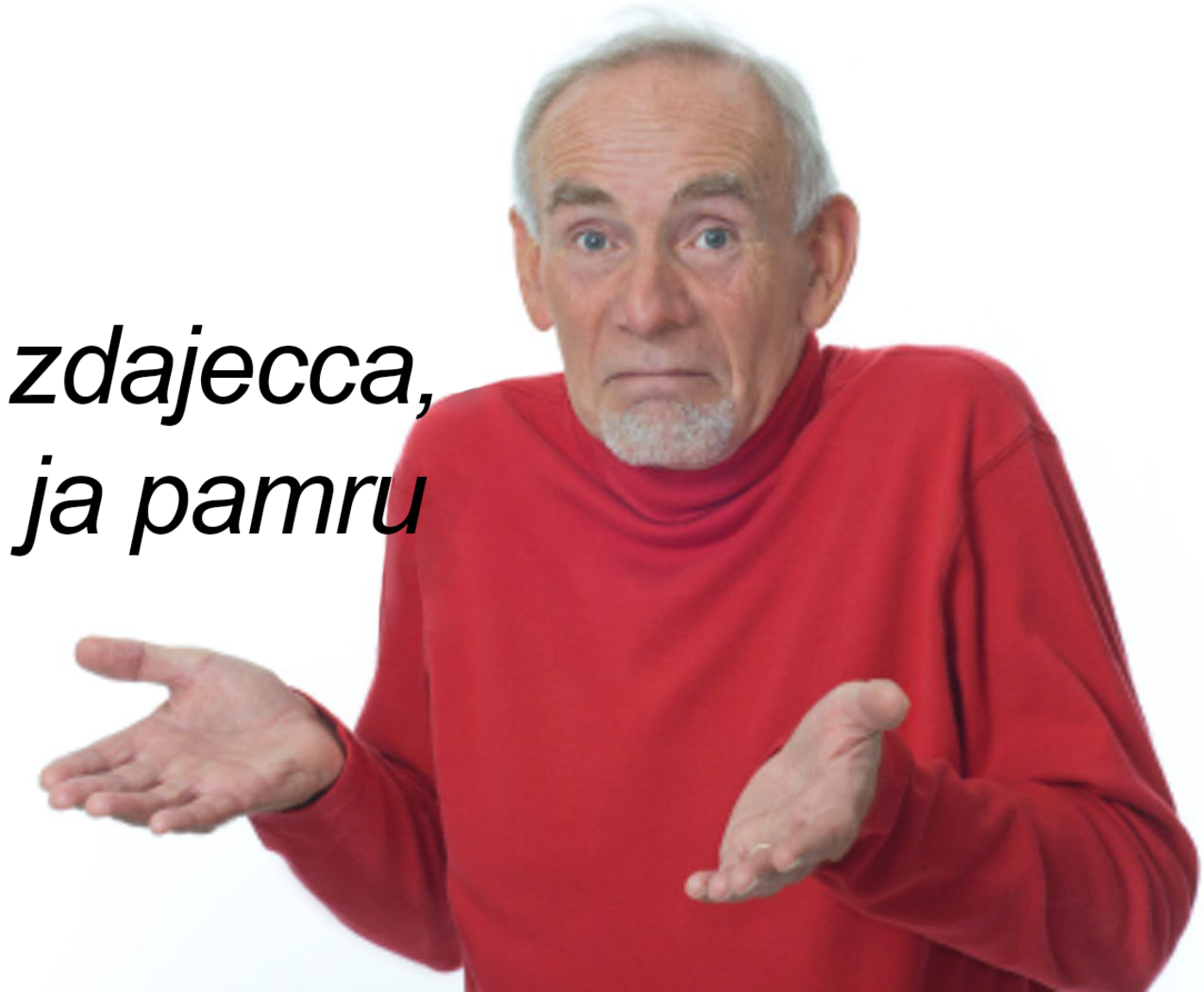
Co se stalo? Zachránili jsme se před kolizi odlepiv od prediktoru adresu 0x0008, která měnila stav. A spočítáme jenom pro 0x0008. Ptá se na špatné predikci – jich bude 9.

Fajn, gratuluji, zkušební úkol z predikce už je za vámi. Modlete se, aby ve vaší variantě cyklus neopakoval vice než 20krát, jinak takový úkol bude pro vás zábavou minimálně na půlhodiny.

V jednodušší variantě úkolu muže vzniknout jednobitový prediktor. Princip bude analogický, jediná změna – stav prediktoru bude skákat mezí T a NT.

K otázce, jak jsme našli, že náš prediktor odpovídá adresám 0x0008 a 0x0018. V úkolu je napsáno, že prediktory jsou indexovaný 2. a 3. bity adres. Numeraci začínáme od nejméně významného bitu(od konce) a od nuly (Píša nebyl by sebou kdyby začali jsme od jedničky). Adresa 0x0008 v bitech bude 0b0000 0000 0000 1000.

Teďka hledáme adresy se stejnými 2. a 3. bity, i když z přednášek jste vynesli alespoň něco, tak pro vás bude jasné, že hledání adresy jsou 0x0008, 0x0018, 0x0028. Chvála Alláhu na adrese 0x0028 není vůbec nic, takže se prediktor zabývá adresami 0x0008 a 0x0018



Congrats, dosáhli jste konce tohoto manuálu! Krátký kurz z APO ke zkoušce je za vámi a jste hotoví zvládnout předmět na C+. Doufám, že jsem zvládla zcela jasně a jednoduše vysvětlit, jak a s čím funguje procesor. Moc se omlouvám za možné chyby ve látce (případně i v češtině), materiály byly sebraný v co nejkratší čas, aby se víc lidí stihli připravit. Ještě jednou zopakuji svou vděčnost redaktorovi, díky jeho práce bylo opraveno opravdu hodně chyb.

Jestliže máte jakoukoliv otázku z témat manuálu můžete mně najit přes telegram @cherikora nebo v diskordu (#6596) a nebojte se se ptát.

Děkují vám za pozornost a přeju hodně štěstí!