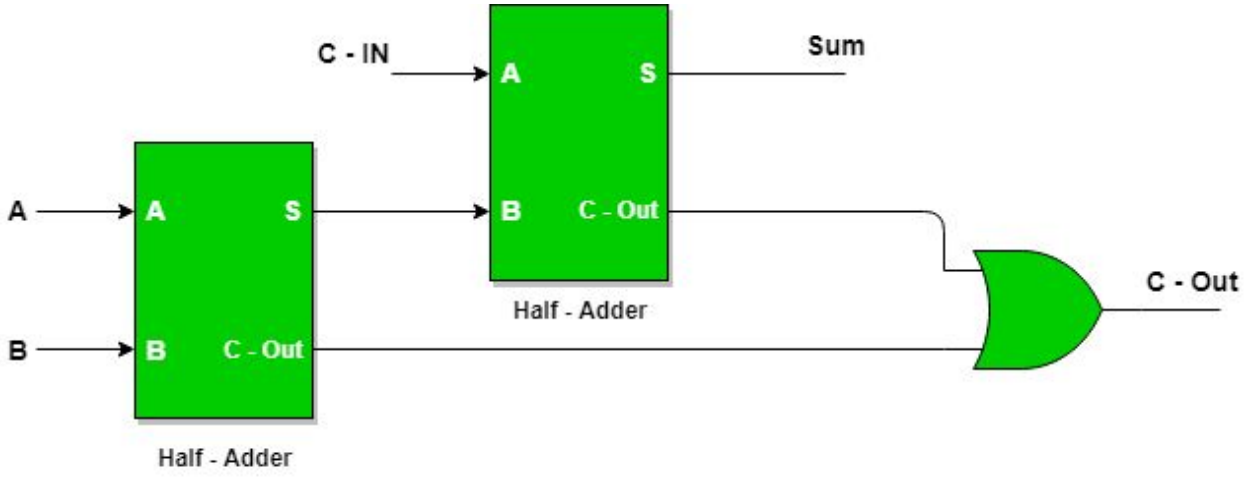
Máme řadič jednocestné datové cesty složený z kombinačních obvodů a chce popsat, jak z toho vyrobit řadič pomocí ROM. V jaké situaci by bylo vhodnější mít řadič pomocí ROM než pomocí komb. obvodů? - 1 bod

- Kombinační obvod je možné reprezentovat booleovskou funkcí
- Booleovská funkce jde reprezentovat tabulkou
- Řádky tabulky (hodnoty kontrolních signálů) odpovídající nějaké instrukci uložíme do ROM a budeme je adresovat op codem té instrukce
- Jednoduché
- Problém: “Jak vyrobit ROM, která je rychlejší než datová cesta?”

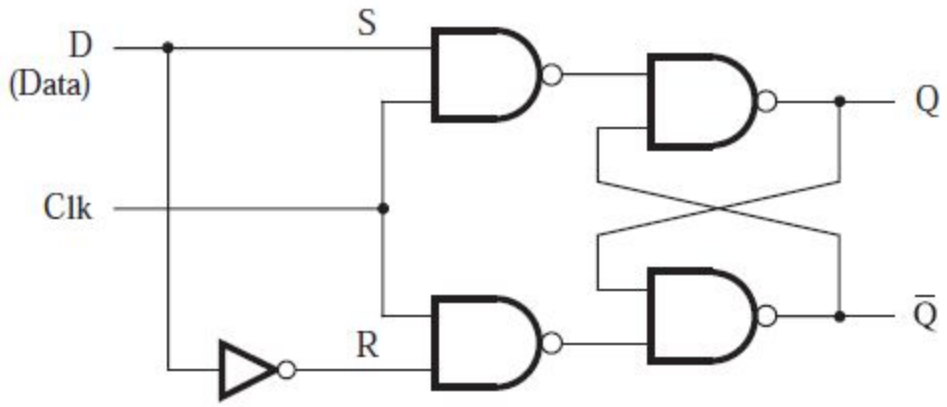
Sčítačka

- Kombinační logika
    - Neobsahuje paměť → nemají vnitřní stav
    - Hodnota výstupu závisí pouze na hodnotě vstupu
- Sekvenční logika
    - Obsahují paměť → vnitřní stav
    - Výstup závisí na vstupu a vnitřním stavu
- Z hradel jsme schopni vyrobit jednak kombinační obvody, ale i sekvenční obvody
- Př. kombinačního obvodu je **sčítačka**



Inputs			Outputs	
A	B	C – IN	Sum	C - Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

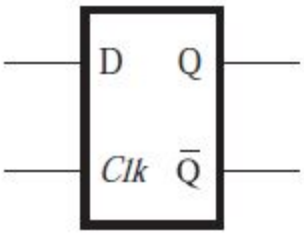
- Př. sekvenčního obvodu je **D latch**



(a) Circuit

Clk	D	Q(t + 1)
0	x	Q(t)
1	0	0
1	1	1

(b) Characteristic table



(c) Graphical symbol

Proč jsou cache lines mocniny dvojky? - 1 bod

- Obvykle kopírujeme vždy celou řádku cache

- Řádky cache chceme indexovat
- Pomocí k bitů dovedu indexovat 2^k řádku cache
- Proč tedy mít méně řádku než dovedu indexovat?

Multilayer cache (L1 a L2)

- Dneska všechny počítače používají cache, protože procesory jsou neuvěřitelně rychlé a DRAM paměti pomalé
- Dokonce se využívá více úrovní cachí (dneska spíše 3 úrovně, budeme ale uvažovat 2 úrovně)
- Pokud L1 cache nemá data a L2 cache má, tak *miss penalty* bude pouze access time do L2 cache což bude mnohonásobně rychlejší než access time do DRAM
- Pokud nemá L1 ani L2 cache data, tak se musí sáhnout do DRAM, bude to trvat nepatrně déle, musíme přičíst miss penalty L1 a L2 cache
- L1 cache je obvykle **menší** (s menšími řádky) než v jedno cachových systémech
- L2 cache je naopak **větší** (s většími řádky) než v jedno cachových systémech
    - Větší asociativita
- L1 — redukuje **hit time**
- L2 — redukuje **miss rate**
    - Aby se nemuselo šahat do DRAM

Co jsou lokální proměnné a jak je zařízeno, že volané metody vidí své lokální proměnné a při návratu (return) opět vidíme své původní proměnné.

- Lokální proměnné jsou proměnné, jejichž životnost je uvnitř metody
- Každé metodě odpovídá na zásobníku *stack frame*
- Ten obsahuje argumenty, návratovou adresu a lokální proměnné metody
- Před návratem (záleží na konvenci) metoda uklidí svůj stack frame, až na návratovou adresu, kterou uklidí volající
- V případě rekurze se prostě takhle zaplňuje a vyprazdňuje zásobník

Jaké problémy nastávají u *superskalárních dynamic multiple issue* procesorů při zpracování výjimek / přerušení a jak se tyto problémy řeší.

- Často se superskalární dynamic multiple issue pipelina pojí s spekulativním prováděním instrukcí, což nám často umožní (pokud je spekulace správná) zlepšit instruction level parallelism
- HW spekulace

    - spekulativní výsledky se bufferují do té doby, než se zjistí, že buďto spekulativní nejsou, nebo se zjistí, že spekulace byla špatná a udělá se rollback
    - Problémem jsou výjimky — provedeme špatnou spekulaci (např. špatně odhadneme adresu) a nastane výjimka
    - Všechny tyto věci (výjimky, zápisy do paměti) drží v bufferech a provedou se (dostanou na povrch), až potom co se zjistí, že tyto výsledky nejsou spekulativní
    - Jinač se to discardne
- SW spekulace
    - Dělá kompilátor
    - Vygeneruje kód, který zkontroluje jestli spekulace byla správná
    - Rollback se obvykle dělá tak, že se zavolá nějaká fix-up routine, kterou kompilátor vygeneroval pro případ selhání spekulace
    - Výjimky se opět ignorují do doby, než je jasné, že by opravdu měli nastat (spekulace byla správná)

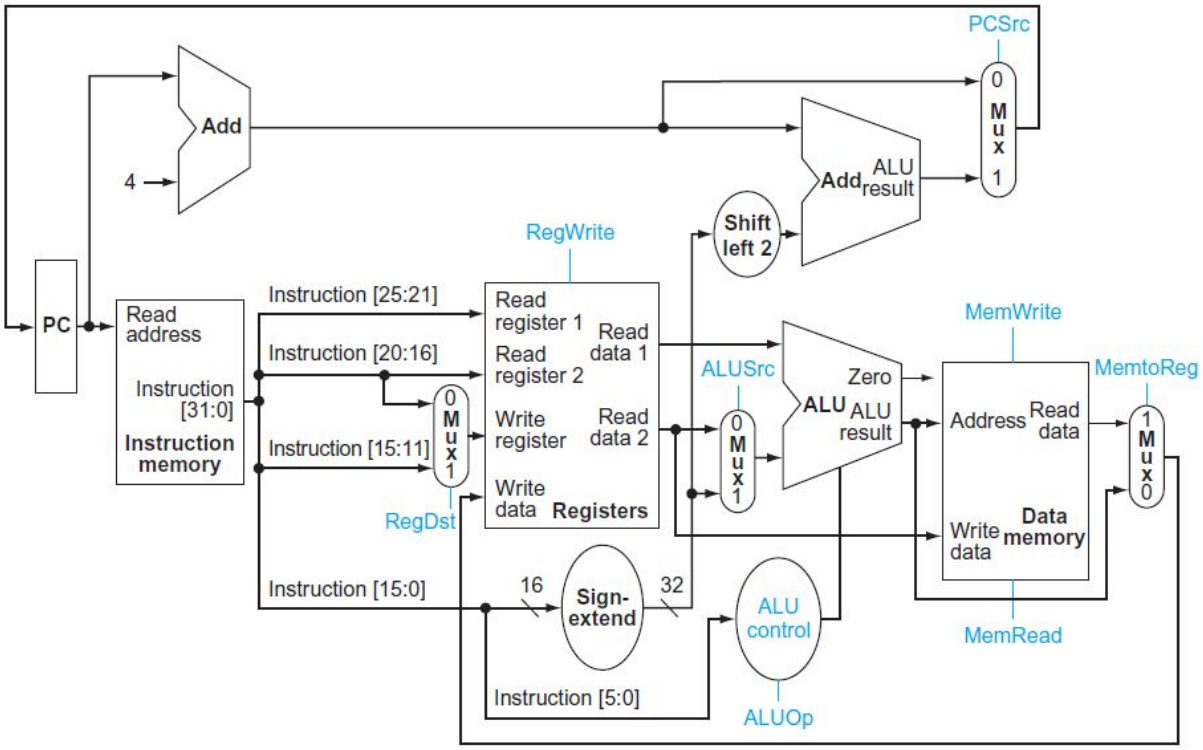
V čem je SRAM výhodnější pro cache v procesoru než DRAM?

- SRAM
    - Rychlost
    - 6 tranzistorů na bit
    - Pro velké kapacity nevhodná kvůli obrovskému množství součástek
    - **Obsah není nutné periodicky obnovovat**
- DRAM
    - Kapacita (hustota)
    - 1 tranzistor a 1 kondenzátor na 1 bit
    - Pomalé
    - **Nutný refreshing** (jinak se zapomenou data)

Co je to *write buffer*, proč se používá a k čemu jeho použití u cache vede.

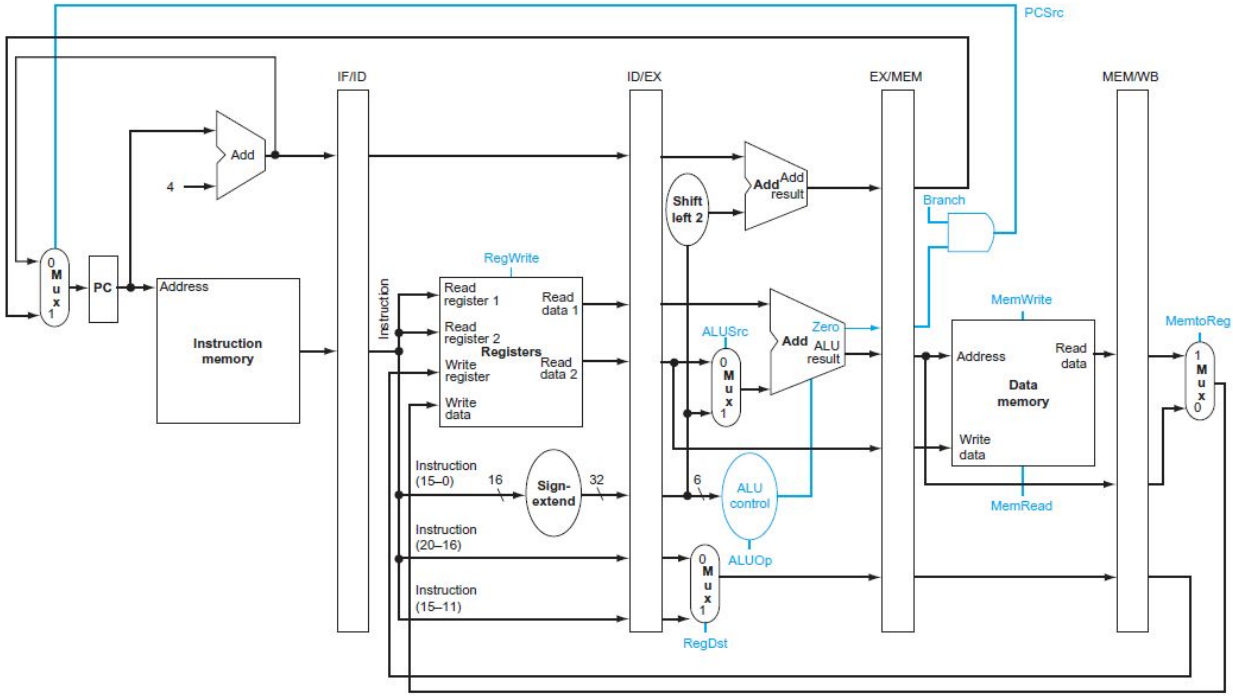
- Write buffer je fronta, která obsahuje data, která čekají na zápis do paměti
- Místo zápisu do cache a paměti a více než 100 taktovém čekání se data zapíší do cache a write buffer a pokračuje execution dále
- Položka se z write bufferu odstraní, až tehdy, když je zapsána do paměti
- Výhodné v případě krátkodobých burst zápisu a pak dlouho nic
- Samozřejmě pokud kontinuálně máme víc požadavků na zápis než stíháme zapsat do paměti, tak musí dojít ke stallu
- Pokud je write buffer plný také musí dojít ke stallu

Popište jak funguje a jaké komponenty obsahuje jednocyklová datová cesta.



- **PC**
  - Adresa do Instruction memory
- **Instruction memory**
  - Paměť obsahující instrukce
- **Register file**
  - Čtení registrů
  - Zápis do registrů
  - Control řídí write enable
- **ALU**
  - Provádí aritmetické a logické operace
  - Její funkci nastavuje control
- **Data memory**
  - Paměť obsahující data
  - Control řídí signál povolující zápis

Vícecyklová datová cesta



- Datovou cestu rozdělíme na 5 částí, které trvají přibližně stejně dlouho
- **IF** (Instruction fetch)
    - Přečteme z Instruction memory instrukci, kterou máme vykonávat
    - Sčítačka, která natvrdo přičítá 4 ke IP (4 byty má instrukce)
- **ID** (Instruction decode / register file read)
    - “Rozebereme” instrukci
    - Přečteme registry z register filu
- **EX** (Execution / address calculation)
    - ALU spočítá výsledek
- **MEM** (Memory access)
    - Zapisujeme nebo čteme z data memory
- **WB** (Write back)
    - Zapisujeme do register filu
- Každá fáze pipeliny má asociovaný registr, ze kterého se přečte jaká instrukci právě vykonává
- Tyto registry obsahují i řídící signály, které se nastaví při fázi ID a pošlou do pipeline registru fáze EX

3) odvodit vzorec pro vztah zrychlení celého programu při zrychlení části programu (1 bod)

- "Zvýšení výkonnosti dosažitelné nějakým zlepšením je omezené mírou používání tohoto zlepšení"
- $S(x) = \frac{1}{p + \frac{1-p}{x}} \qquad 0 \leq p \leq 1,\ x \in \mathbb{N}$
- $\lim_{x \to \infty} S(x) = \frac{1}{p} \qquad p \neq 0$

6) na dané architektuře procesoru s daným programem je 10% instrukcí skoku, používá se statický always-not-taken prediktor s úspěšností predikce 60%, pipeline má 5 stupňů (IF,ID,EX,MA,WB), skok se vyhodnotí v EX. Jak přispějí neúspěšné predikce skoku k CPI v tomto programu, za předpokladu, že nenastávají datové hazardy, ani jiná zpoždění? (2 body)

- Pozorování
  - Pokud se predikce u instrukce skoku nepovede **ztratíme 2 takty**
- Nechť **B** označuje náhodný jev "instrukce skoku"
- Nechť **C** označuje náhodný jev "predikce uspěla"
- $P(B) = 0.1$
- $P(C) = 0.6$
- Dále předpokládejme, že **B** a **C** jsou nezávislé jevy
- $P(B \cap C^c) = 0.1 \cdot 0.4 = 0.04$
  - Pravděpodobnost, že náhodná instrukce je instrukcí skoku a predikce na ní neuspěje
- Předpokládejme, že $\#instructions$ značí počet provedených instrukcí programu
- Předpokládejme, že $\#clock\ cycles$ značí počet provedených taktů procesoru **bez jakýchkoliv stalls** (predikce je jakoby 100%)
- Víme tedy, že počet instrukcí skoku, které nebudou dobře predikovány je:
  - $\#instructions \cdot 0.04$
- Pro každou takovou instrukci ztratíme přesně 2 takty (fáze pipeliny trvá 1 takt), tedy celkově ztratíme:
  - $\#instructions \cdot 0.04 \cdot 2$

- Z definice: $CPI\ staré = \frac{\#clock\ cycles}{\#instructions}$
- A tedy
  $$CPI\ nové = \frac{\#clock\ cycles + 0.08 \cdot \#instructions}{\#instructions} = \frac{\#clock\ cycles}{\#instructions} + 0.08 = CPI\ staré + 0.08$$

7) jak funguje statická predikce skoku a proč je nevhodná pro vnořené cykly (2 body)

- Predikce, která není jinak závislá na historii
- Na tvrdo predikuji vždy skočím, nebo nikdy neskočím
- Jednoduché heuristiky (vzdálenost skoku)
- Heuristiky na základě nějakého bitu v instrukci

- **Vnořené cykly???**
  - Ve slidech a H&P knize pouze zmíňka o problému u dynamických 1-bitových prediktorů
  - Problém je v tom, že když se pořád skáče a pak se jednou neskočí, tak nám to obrátí tento primitivní prediktor
  - 2x se prediktor netrefí
  - U vnořených cyklů násobně více

14) v nějakém vlastním assembleru implementujte program, který bude od proměnné uložené v paměti odčítat jedničku, dokud nebude nula, popište jaké typy instrukcí jsou k tomu potřeba (2 body)

- Nechť se proměnná nacházející se v registru t0
- li $t1, 0 #t1 = 0
- li $t2, 0 #t2 = 0
- li $t3, 1
- beq $t0, $t1, KONEC #if(t0 == 0B) { goto KONEC }
- slt $t2, $t0, $t1 #if(t0 < t1) { t2 = 1 }
- beq $t2, $t3, KONEC #if(t0 < 0) { goto KONEC }
- ZACATEK:
- sub $t0, $t0, 1 #t0 = t0 - 1
- bne $t0, $t1, ZACATEK #if(t0 != 0) { goto ZACATEK }
- KONEC:
- nop
- ...

3) Jak ovlivňuje zvolený algoritmus rychlost programu (nebo tak nějak)?

- Cache-friendliness
  - Je nutné brát paměťovou hierarchii v potaz
  - Algoritmus s časovou složitostí $O(n)$ může dopadnout mnohem hůře než algoritmus s časovou složitostí $O(n \log n)$, který si bude dobře hospodařit s cachí
- Ovlivnění počtu vykonaných instrukcí (záleží na kouzlech kompilátorů)

1) jakým způsobem ovlivňuje překladač, programovací jazyk a architektura procesoru rychlost vykonávání programu (1 bod)

- Překladač
  - Efektivita překladu určitých částí kódu (využití efektivních instrukcí)
  - Optimalizace
    - Jednak zlepšení kvality kódu, odstranění zbytečných částí, náhrada méně efektivních za více efektivní
    - Hledání závislostí mezi instrukcemi a jejich přeskládání značně pomůže dynamic multiple issue procesorům
    - Static multiple issue jsou prakticky závislý v mnoha ohledech **pouze** na kompilátoru
  - **Kolik a jakých instrukcí bude program mít?**
    - (jak moc tento kód bude procesoru chutnat)
    - Souvisí s programovacím jazykem
- Programovací jazyk
  - Povolené konstrukty, abstrakce, zjednodušení (souvisí s kompilátorem, jak se s tím vypořádá)
  - Implementace standardní knihovny (pokud budeme používat)
  - Obrovská závislost na kompilátorech z hlediska efektivity
- Architektura procesoru
  - Na co HW má přímo instrukce
    - Některé konstrukty z programovacích jazyků mohou být obtížně proveditelné pro určitou architekturu, kde chybí nějaká instrukce a může být obšírné a náročné tuto instrukci nahradit za nějakou kombinaci ostatních
  - Jak moc je dobrý kompilátor pro danou architekturu
    - Umí použít vhodné instrukce na dané úkony?
    - Umí použít specifické instrukce dané architektury?

Hardware or software component	Affects what?	How?
Algorithm	Instruction count, possibly CPI	The algorithm determines the number of source program instructions executed and hence the number of processor instructions executed. The algorithm may also affect the CPI, by favoring slower or faster instructions. For example, if the algorithm uses more divides, it will tend to have a higher CPI.
Programming language	Instruction count, CPI	The programming language certainly affects the instruction count, since statements in the language are translated to processor instructions, which determine instruction count. The language may also affect the CPI because of its features; for example, a language with heavy support for data abstraction (e.g., Java) will require indirect calls, which will use higher CPI instructions.
Compiler	Instruction count, CPI	The efficiency of the compiler affects both the instruction count and average cycles per instruction, since the compiler determines the translation of the source language instructions into computer instructions. The compiler's role can be very complex and affect the CPI in complex ways.
Instruction set architecture	Instruction count, clock rate, CPI	The instruction set architecture affects all three aspects of CPU performance, since it affects the instructions needed for a function, the cost in cycles of each instruction, and the overall clock rate of the processor.

5) Intel vzal číslo (0x1E1010) z registru, uložil ho do paměti. Tam si ho přečet MIPS, zvětšil o jedna a uložil zpátky. Pak ho Intel přečet zpátky, ale číslo se zkazilo, tj. nebylo o jedna větší (0x1F1010). Proč se tak stalo?

- Všechny procesory dneska (Intel) jsou little-endian
  - Lowest address → least significant byte
- Zatímco MIPS byl (dle zadání) big-endian
  - Lowest address → most significant byte

9) Jak ovlivní zvětšení cache line hity / missy a přístupové doby.

- Přístupová doba
  - Nezávisí (skoro)
  - Maximálně když hledám nějaké slovo, tak záleží na míře asociativity cache
  - Ale to jsou detaily
- Miss penalty bude menší, protože bude nutné překopírovat méně dat
- Hit time obdobně jako miss penalty

- Ohledně miss rate záleží:
    - Pokud máme mnoho malých bloků a ubereme tím, že bloky zvětšíme, pak se naše miss rate může zmenšit
    - Pokud již ale máme málo bloků a ubereme tím, že bloky zvětšíme, pak se naše miss rate zvětší

10) Máte program, který vezme i-tý prvek pole A, přičte k němu jedna a uloží ho do i+1 prvku pole A. Napište takový program ve vlastním assembleru a řekněte, jaké instrukce na to potřebujete (popište je).

- Předpokládejme že adresa pole je 0x42
- i-tý prvek se nachází na offsetu 0x10
- Velikost jednoho prvku jsou 4 byty → i-tý prvek je prvek s indexem 4
- li $t1, 0x42
- lw $t0, 0x10($t1)
- addi $t0, $t0, 1
- sw $t0, 0x14($t1)

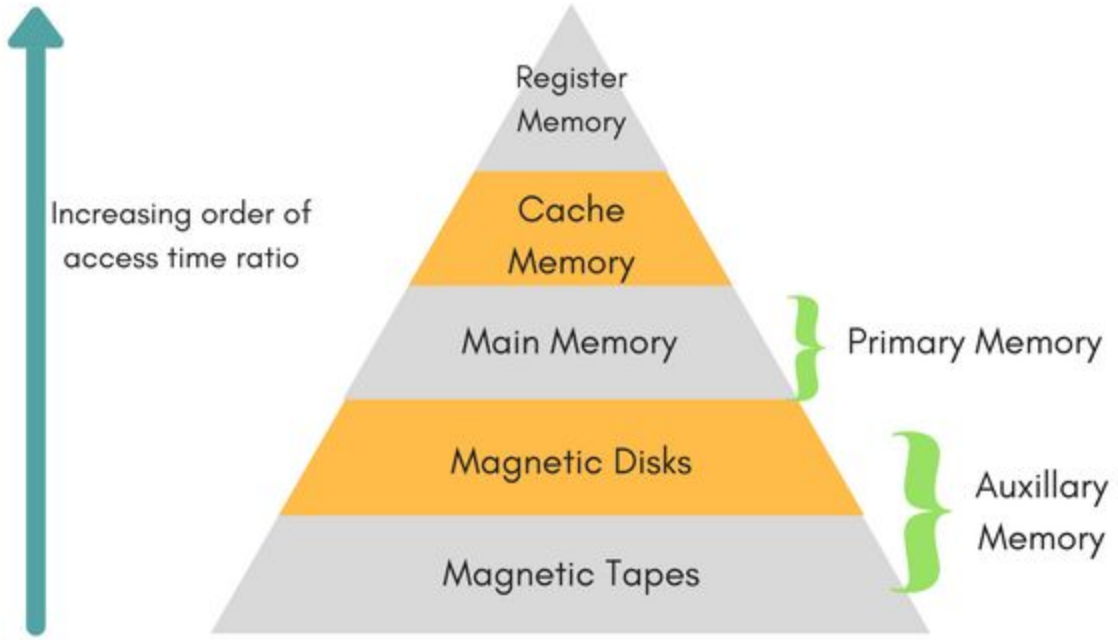
Vysvětlete pořádně princip fungování {direct-mapped, partial associative, full associative} cache

- Direct mapped
    - **(mem. block address) mod (number of cache blocks)**
    - Adresu paměťového bloku (který chceme vložit do cache) rozdělíme na 2 části
        - Index (k bitů, abychom uadresovali 2^k bloků cache)
        - Tag (k ověření jestli v cache je opravdu daný blok)
    - Jeden paměťový blok … jeden blok v cachi
    - Může se stát, že se nám 2 bloky, se kterými chceme pracovat budou vzájemně vyhazovat z cache
- Partial associative
    - k-way associative znamená, že každá množina má velikost k
    - **(mem. block address) mod (number of sets in cache)**
    - Adresa paměťového bloku je opět rozdělena na 2 části
        - Index (k bitů, abychom uadresovali 2^k množin bloků cache)
        - Tag (k ověření jestli v cache je opravdu daný blok)
    - Jeden paměťový blok se může nacházet **v libovolném prvku** množiny bloků asociované k danému indexu
        - Hledá se paralelně za pomocí komparátorů
- Full associative
    - Index má délku 0

    - Tag je vlastně celá adresa paměťového bloku
    - Paměťový blok může být úplně kdekoliv
    - Je nutné projít opravdu celou cache
    - Aby to bylo použitelně rychlé hledá se paralelně za použití obrovského množství komparátorů, které značně zvyšují cenu takovéto cache
- Nejčastěji se dnes využívají právě částečně asociativní cache (4-way associative)
- S větší velikosti cachí ztrácí plná asociativita význam
- Vyšší stupně asociativity (např. z 8-way na 16-way) často přinášejí jenom velmi malý užitek
- Největší užitek přináší skok z 1-way associative (direct mapped) na 2-way associative

1) Jak ovlivňuje rychlost procesor a paměťový subsystém? (1)

- Výkonnost počítače jako celku dneska zejména vázne na rychlosti paměťového subsystému
- Přístup do paměti RAM trvá ~ 200 taktů **(hodně vágní)**
- V jednom taktu dovedu na dnešních procesorech sečíst 50 čísel (paralelně)
- O ostatních paměťových médiích jako HDD nemá smysl uvažovat, protože jsou ještě mnohem pomalejší (8ms přístupová doba) a nejsou přímo adresovatelné (přistupujeme k nim přes OS)
- Aby nám paměťový subsystém držel krok s rychlostí procesoru snažíme se o:



- Nejrychlejší, nejdražší a nejmenší paměť je nahoře
- Nejpomalejší a obrovská paměť je dole

- Pokud budeme mít správná data ve správný čas v malé paměti nahoře, vytvoříme iluzi, že “Máme k dispozici paměť o velikosti HDD s rychlostí cache”

2) Jak se zjistí kam se má program vrátit po ukončení funkce (při příkazu return) (1)

- Některé procesory **nemají** nativní podporu zásobníku, ale mají k dispozici registry přímo určené pro argumenty a návratovou adresu
- Toto řešení není dobré, nemáme rekurzi
- Obvyklé řešení je, že máme k dispozici zásobník, kam se nejdříve návratová adresa a za ni případné argumenty
- Při návratu po sobě procedůra uklidí až na návratovou adresu, na kterou skočí, ale obvykle ji ze zásobníku odstraní volající
- Na Intelech je instrukce **CALL** (která uloží návratovou adresu na zásobník a skočí na adresu procedůrky)
- Ale v praxi toto celé může být jinak:
    - Např. v C++ se na MSVC předává this pointer vždycky v ECX na Intelech
    - Proč by nemohla být návratová hodnota taky předána nějakým jiným způsobem?
    - Záleží na kompilátoru a standardech

4) Popište přístup do plně asociativní cache. (2)

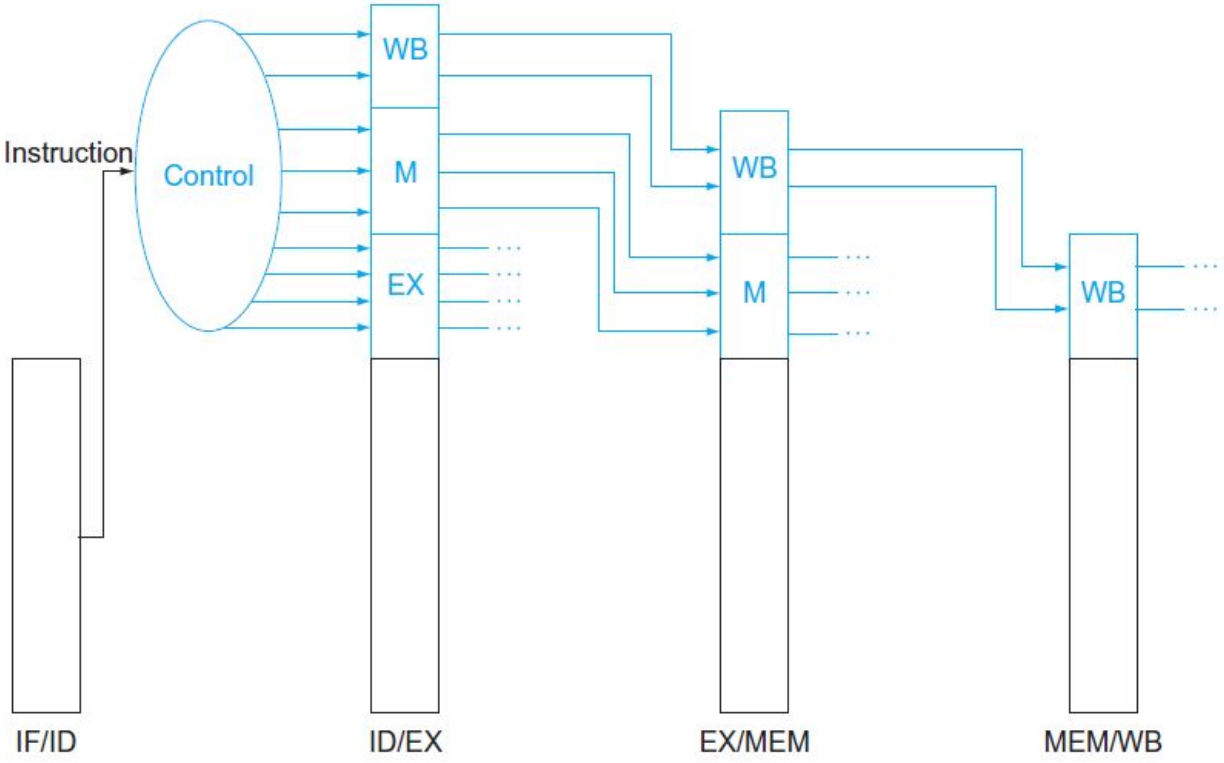
- Zápis
    - Zapíšu do volného bloku
    - Pokud jsou všechny plné, tak kandidáti na vyhození jsou všichni
        - LRU (většinou aproximace)
        - Pseudorandom (většinou není o tolik horší, nekdy i lepší pro velké cache)
- Čtení
    - Blok paměti, který hledám může být v libovolné cache line
    - Je třeba prohledat celou cache
    - Aby to bylo prakticky rychlé dělá se to paralelně pomocí obrovského množství přítomných komparátorů, kvůli kterým jsou takové cache velmi drahé

5) Co dělá řadič pipeline-y? (2)

- Logický obvod generující řídící signály
- Hodnoty signálů závisí na instrukci
- **Co dělá control obecně**
- Přepíná řídící signály pro multiplexery
    - Řídící signál vybírá který vstup bude na výstupu

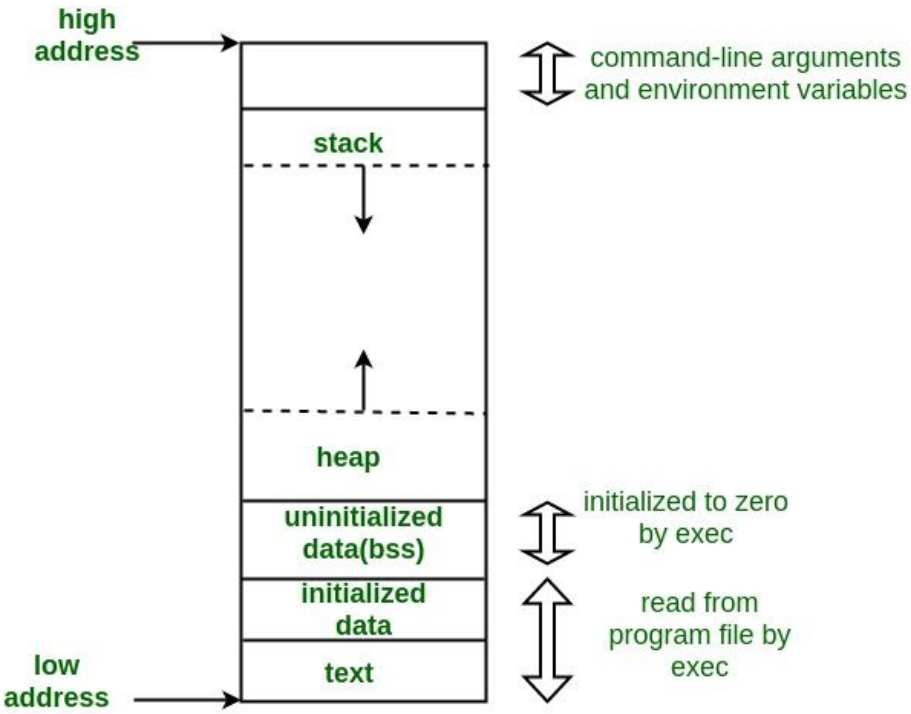
    - Je věc, která se má zapsat do registru přes register file z paměti nebo ALU?
- Řídí ALU
    - Podle OP codu instrukce přepne ALU na sčítačku, odčítačku, děličku,...
- Ovládá write enable signály do paměti a do registr file

- Čím se liší pipeline control od obyčejného řízení



- Hned v první fází control uloží do registru 2. fáze všechny signály, které budou potřeba
- Instrukce potom prochází pipelinou spolu s řídícími signály
- Řídící signály se pak vytáhnou z registru asociovaného s danou fází pipeliny, stejně jako instrukce

7) Co ovlivňuje počet vnořených funkcí? (1)



- Takhle nějak vypadá memory layout programu v C++
- Vidíme, že stack roste proti heapu
- Jedna funkce na zásobníku = stack frame
- Když zásobník doroste k heapu dojde ke stackoverflow
- Počet vnořených funkcí ovlivňuje:
    - Omezená paměť zásobníku (máme třeba 1MB)
    - Plná paměť (zásobník dorostl heap)

8) Velmi podobná šestce. Kdy v rámci zpracování instrukce se procesor dozví, že došlo k přerušení či výjimce a co s tou instrukcí udělá. (1)

- Záleží o jakou výjimku se jedná
- Předpokládejme, že jde o přetečení (např. při sčítání)
- V tomto případě se to dozví řízení procesoru z **overflow** bitu, který bude jeden z výstupních z ALU
- Adresa instrukce sčítání, která způsobila výjimku se uloží do EPC
    - Exception program counter

9) Něco k forwardingu/bypassingu (1)

- Uvažme dvě instrukce
    - add **$s0**, $t0, $t1
    - sub $t2, **$s0**, $t3
- Tyto instrukce budou v pipelině za sebou
- Problém je v té zvýrazněné závislosti
- Instrukce sub chce ve 3 fázi předchozí výsledek, v té době bude instrukce add ve 4. Fázi, výsledek však zapíše do paměti až v 5. fázi
- A máme 1 takt stall

- Avšak výsledek té sčítací operace již je k dispozici již ve 4. fázi, jenom není vepsán do paměti
- Tento problém řeší **forwarding/bypassing**, závislost prostě předáme tam, kde je třeba a vyhneme se tak čekání na zápis do paměti a následné čtení z paměti
- Často je možné se vyhnout kolizím pouze šikovným přerovnáním instrukcí (dělá kompilátor)
- Existují i situace, kdy se sejdou dvě nevhodné instrukce a musí se holt počkat třeba takt

10) Zadané tři procesory s frekvencí, počtem instrukcí a celkovým časem. Jak se musí změnit frekvence jednoho procesoru, aby běžel tak dlouho jako ten druhý? (1)

- Následují dvě klasické CPU performance rovnice
- $CPU\ time = Instruction\ Count \times CPI \times Clock\ cycle\ time$
- $CPU\ time = \frac{Instruction\ Count \times CPI}{Clock\ rate}$

11) Výhody Von Neumannovy architektury. (2)

- Oproti Harvardské architektuře máme jednu paměť, ve které jsou jednak data i kód
    - (Aby nebylo možné jen tak z některých dat udělat kód, tak jsou obvykle klasifikované stránky operačním systémem podle toho, jestli je z nich možné spouštět kód)
- To sice na první pohled nemusí vypadat jako výhoda, ale je to výhoda
- Můžu ovlivnit, změnit kód, který bude procesor bude vykonávat
    - Bez toho aniž bych musel odletovávat ROMku s firmwarem
- V Harvardské architektuře mám obvykle ROM paměť s kódem, který procesor vykonává, tomuto kódu se říká **firmware**, pokud bychom ho někdy chtěl změnit, tak jedině externě
- Nejde aby si sám počítač změnil firmware v tomto případě

- Z toho vyplývá další věc, kterou Harvardská architektura nemá, a to je **flexibilní velikost kódu a dat**
- Poslední mě napadá **jednoduchost**

13) Jak procesor čte instrukce? Jak pozná co je operand v instrukci a co je instrukce (nebo něco takového) (1)

- Uvažujme, že se bavíme o **MIPS**u
- Aktuální adresa instrukce jejíž vykonávání začíná je uložena v registru **PC**
- Máme oddělenou paměť pro instrukce a data
- Tedy instrukci si přečte z **Instruction memory**

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

- Na MIPSu máme pouze 3 formáty instrukcí
- Podle prvních 6 bitů, tj. **op** se pozná o přesně jakou instrukci se jedná → tedy se ví i do které kategorie spadá

14) Napište ve vlastním asssembleru jak byste od 5bitu do 10 bitu nějakého Intu zapsali 5 bitové číslo. (2)

- MIPS assembler
- Add $t0, $t1, $t2 # t0 = t1 + t2

Jméno	Číslo	Použití	Jméno	Číslo	Použití
$zero	0	Konstantní hodnota 0.	**$t8 – $t9**	24 – 25	Další mezivýsledky.
$at	1	Rezervováno pro assembler.	**$k0 – $k1**	26 – 27	Rezervováno pro jádro OS.
$v0 – $v1	2 – 3	Hodnoty výsledků v výrazů.	**$gp**	28	Global pointer.
$a0 – $a3	4 – 7	Argumenty funkcí.	**$sp**	29	Stack pointer.
$t0 – $t7	8 – 15	Registry pro mezivýsledky.	**$fp / $s8**	30	Frame pointer.
$s0 – $s7	16 – 23	Uschovávané registry.	**$ra**	31	Návratová adresa.

- Pár MIPS instrukcí, které se mohou hodit:
- lw $t0, 32($t1) #načti do t0 obsah paměti s adresou t1+32
- sw $t0, 32($t1) #ulož do paměti na adresu t1+32 hodnotu t0

- addi $t0, $t1, 10 #t0 = t1 +10

- li $t0, 10 #t0 = 10

- beq $s0, $s1, 0x48 #if (s0 == s1) { PC = 0x48 }
- bne $s0, $s1, 0x48 #if (s0 != s1) { PC = 0x48 }

- j 0x48 #PC = 0x48

- slt $s1, $s2, $s3 #if(s2 < s3) { s1 = 1 }

- Pokud je úkolem vyříznout 5. až 10. bit nějakého 32 bitového integeru a uložit ho do 32 bitového integeru
    - andi $t0, $t0, 0x7E0 #0x7E0 jsou jedničky na bitech 5-10
    - srl $t0, $t0, 5 #shift doprava o 5 pozic

2.) Kdy dochází k nutným/studeným výpadkům (compulsory/cold miss), jak se liší od jiných výpadků cache.

- 4C model (někdy se uvádí 3C a coherency misses se vynechají)
    - Compulsory / cold-start misses
        - K těmto výpadkům cache dochází, pokud přistupujeme k datům, která ještě předtím nebyla v cachi
        - Částečně se dá řešit např. technikou zvanou **prefetch**, načtu do cache data, která vím/tuším, že se mi budou hodit
    - Capacity misses
        - Výpadky způsobené plnou cachí
        - Tj. něco jsme vyhodili z nedostatku místa, ale potom jsme to někdy potřebovali zase načíst do cache
    - Conflict / collision misses
        - Ve full-associativity cachích k těmto výpadkům nedochází
        - Čím větší stupeň stupeň asociativity, tím menší šance, že tento výpadek nastane
        - V direct-mapping cachích obrovský problém — musíme vyhodit něco z cache (co se nám může hodit), protože se to namapuje na stejné místo, ačkoliv cache je třeba poloprázdná
        - (Největší skok je mezi direct-mapping a two-way associativity, pak už to není tak markantní)
    - Coherency misses
        - Data z cache mi invalidoval jiný procesor (pomocí nějakého cache coherency protocolu)

4.) V tabulce byly časy jednotlivých stupňů pipeline (IF, ID, EX, MEM, WB). Zadání bylo spočítat periodu taktu jednocyklového a pipelined procesoru.

- Takt procesoru musí být dostatečně dlouhý, aby se v něm stihla provést každá fáze pipeliny
- Tedy by takt měl být max($\mathbf{T_1}$, $\mathbf{T_2}$, $\mathbf{T_3}$, $\mathbf{T_4}$, $\mathbf{T_5}$), kde T_i je čas i-té fáze pipeliny

5.) Na obrázku byl stavový automat dynamického prediktoru. Zadání bylo spočítat jeho přesnost při vykonávané posloupnosti vyhodnocených skoků.

- Dynamické prediktory
    - **Důvod**: statické prediktory, které vždy predikují skok nebo obráceně jsou nevyhovující pro delší a multiple-issue pipeliny
    - **Myšlenka**
        - Prediktor bude mít vlastní buffer indexovaný koncovou částí adresy (stejně jako direct mapped cache)
        - Do vlastního bufferu si bude zaznamenávat, jestli se skok naposledy provedl nebo ne
        - Podle toho se nějak “magically” rozhodne
        - Vzhledem k tomu, že prediction buffer bude indexovaný pouze koncovou částí adresy může docházet k “false matchům”
    - Nejtriviálnější **1-bitové** prediktory jsou však **obvykle také nevyhovující**

    - **Příklad**
    - Consider a loop branch that branches nine times in a row, then is not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?
    - **Řešení**
    - The steady-state prediction behavior will mispredict on the first and last loop iterations. Mispredicting the last iteration is inevitable since the prediction bit will indicate taken, as the branch has been taken nine times in a row at that point. The misprediction on the first iteration happens because the bit is fllipped on prior execution of the last iteration of the loop, since the branch was not taken on that exiting iteration. Thus, the prediction accuracy for this branch that is taken 90% of the time is only 80% (two incorrect predictions and eight correct ones).

    - Ideálně bychom si představovali, že přesnost prediktoru se bude blížit procentuálně hodnotě s jakou se skok provede (zejména pokud se skok provede velmi pravidelně s 90% pravděpodobností)
    - V tomto případě existuje jednoduché řešení: **Nahradíme 1**-bitový prediktor za **2**-bitový prediktor, pro změnu predikce se musí 2x za sebou prediktor splést
- Poznámky:
    - V MIPSu není žádný dynamický prediktor, vzhledem ke krátké pipelině a kvůli tomu, že doba potřebná na určení, jestli se skok provede je pouze jeden takt
    - Kvůli tomu se rozhodli použít tzv. delayed branch scheduling
    - Kompilátor naplánuje nějakou vhodnou nezávislou instrukci k provedení místo taktu pauzy
    - U složitějších procesorů s dlouhými pipelinamy a delší dobou nutnou k zjištění povahy skoku je toto nepraktické

7.) Otázka na strukturální hazard.

- Hazard
    - Situace, kdy není možné vykonat následující instrukci v následujícím taktu (tak, jak by se očekávalo)
    - **Structural hazard**
        - Hardware neumožňuje aby taková to kombinace instrukcí byla provedena
        - U MIPSu by se nám to mohlo stát, kdybychom měli pouze jednu paměť, chtěli bychom číst z paměti instrukci a operand v jednom taktu
        - Jinak MIPS byl navržen tak, aby byl pipelinenován, tedy ke strukturálním hazardům dojít nemůže
    - Data hazard
        - Data, která bychom potřebovali, nejsou k zatím k dispozici
    - Control hazard = branch hazard
        - Instrukce, která měla proběhnout v nějakém taktu neproběhla, přečetli jsme a vykonávali místo ní jinou

8.) Co se stane při výjimce Co se musí zajistit pro její obsloužení a co její obsluha obnáší

- Obecný průběh výjimky
    - Procesoru musí uložit adresu instrukce, která způsobila výjimku do **EPC** (exception program counter)
    - Předat řízení operačnímu systému na nějaké specifikované adrese

    - Operační systém si udělá svoje… Poté program ukončí, nebo spustí program tam, kde skončil (ví kde — EPC)
- Problém: Aby OS vyřešil co s výjimkou, musí znát důvod
    - MIPS pro tenhle účel má tzv. *Cause Register*
    - Druhá častejší metoda je využití **vectored interrupts**
        - Adresa, na který OS dostane předané řízení určuje důvod/povahu výjimky
- **Před obsluhou výjimky je nutné vyprázdnit pipeline**
- **Důraz na zachování konzistentního stavu procesoru**

9.) Jak výměna procesoru za rychlejší model ovlivní response time a throughput.

- Response time = execution time
    - Jaký procesorový čas trvalo počítači dokončit daný úkon?
- Throughput = bandwidth
    - Kolik práce počítač/počítače udělali za daný čas?
- V dnešní době je už **velmi obtížné** neustále zlepšovat procesory tak, aby se zvyšoval jejich **response time**, tj. vlastně aby zadarmo všechno fungovalo výměnou procesoru rychleji
- Zatímco výkon dneska zvyšujeme spíše zvyšování bandwidthu
- Př. nový CPU má o 0.5Ghz větší takt a o 4 jádra více
    - Zvýšení taktu sníží response time všech jader procesoru
    - Paralelní využití všech jader **znatelně** zvýší throughput nového procesoru

10.) write back vs. write through, write allocate vs. write no allocate

- Zapisujeme data, pokud je zapíšeme jenom do cache, tak cache bude nekonzistentní oproti paměti
- Pokud všechno budeme rvát hned do paměti, tak to bude trvat
- **Write-through**
    - Nejjednodušší mechanismus
    - Při zápisu propíšeme data přes cache až do paměti
    - Pokud nastane **write miss** (v cachi jsou na kolizní adrese jiná data) a uvažujeme **write allocate**
        - Nejdříve natáhneme data, která chceme přepsat do cache
        - Potom přepíšeme cache a propíšeme nová data až do paměti
    - Bez **write allocate** tj. **write no allocate**
        - Přímo zapíšeme data pouze do paměti, bez natahování dat do cache

        - Občas se to hodí, třeba když OS nuluje stránky, nechce si vymlátit celou cache
    - Tento způsob nedosahuje příliš vysokého výkonu
    - Při burst zápisech pomůže **write buffer**, kde se bufferují data, která se postupně propisují
- **Write-back**
    - Bloky (řádky) cache budou obsahovat tzv. **dirty bit**, který bude označovat, jestli jsou data konzistentní
    - Data zapíšeme **pouze** do odpovídajícího bloku cache
    - Modifikovaný blok (dirty_bit = 1) je propsán níže hiearchií pouze v případě, že by měl být vyměněn
    - Mnohem složitější na implementaci než write-throught

11.) Co je superskalární procesor s dynamickým plánováním instrukcí. Proč je výkonnější než skalární.

- Skalární procesory
    - Procesory, které v jednom taktu vykonávají **právě jednu** instrukci
- Superskalární procesory
    - Procesory, které v jednom taktu vykonávají **více než jednu** instrukci
    - Multiple issue procesory
    - Static multiple issue
        - Balíčkování instrukcí, **VLIW**
        - Závislost kompilátoru na typu CPU, nutnost rekompilace
    - Dynamic multiple issue
        - V nejjednodušším provedení se instrukce provádějí v pořadí a procesor je rozhoduje, jestli v jednom taktu je možné provést 0,1 nebo více instrukcí
        - I zde je nápomocný kompilátor — přeskládává a plánuje instrukce dle závislostí
        - Avšak závislost na kompilátoru není taková jako u VLIW, máme garanci, že program nemusíme při výměně procesoru za jiný model rekompilovat
        - Dynamické plánování umožňuje hw komponenta procesoru, procesor si “vybere”, které instrukce v daném taktu provede, aby co nejvíce zabránil hazardům a nucením zastavením pipeliny
- Výhoda superskalárního procesoru, oproti skalárnímu by měla být zřejmá — jakýkoliv superskalární procesor se alespoň snaží o nějaký instruction level parallelism, zatímco skalární nikoliv

12.) wall clock time vs CPU time

- Wall clock time = response time = elapsed time
    - Celkový čas nutný k dokončení požadavaného úkonu
    - Součástí je čas procesoru, přístupy do paměti, IO operace, práce procesoru na jiných úkonech
- CPU time = CPU execution time
    - Vyloženě pouze čas, který stráví procesoru prací na našem úkonu
    - Všechny ostatní věci se sem nepočítají — jako např. čekání na IO
- Pozor na to, že čas, který **zaznamená** uživatel používající náš program bude však **Wall clock time**