Sekvenční a implicitně paralelní systémy

Úvod do sekvečních systémů

Urychlení výpočtů na úrovni hardware

Urychlení výpočtů na úrovni software

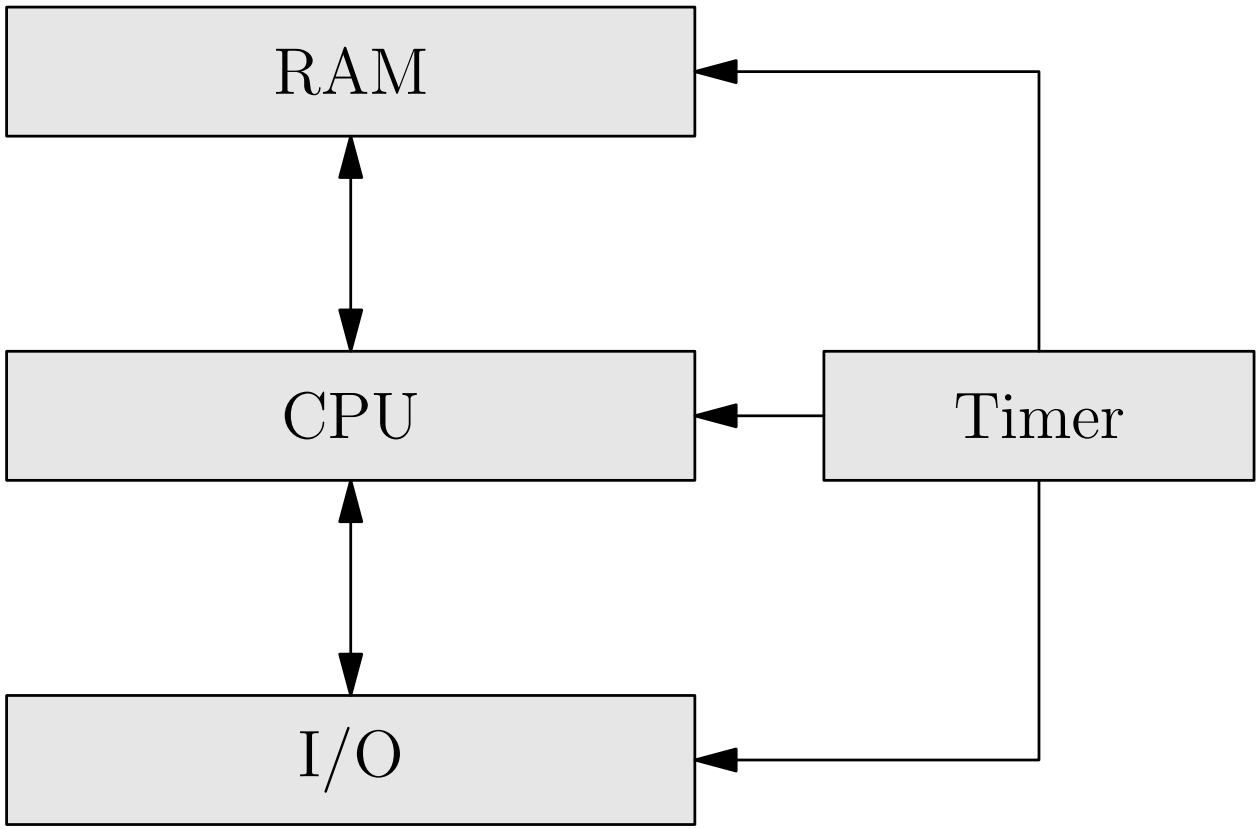
Sekvenční systém

Definition

Za **sekvenční systém** budeme považovat systém založený na von Neumannově architektuře s **jednou** centrální jednotkou - CPU.

Sekvenční systém může být doplněn o **aritmeticko logickou jednotku** - ALU.

Von Neumannova architektura - připomenutí



Jak urychlit výpočet na sekvečním systému?

- na straně hardware

Jak urychlit výpočet na sekvečním systému?

- na straně hardware
  - zvýšit výkon CPU

Jak urychlit výpočet na sekvečním systému?

- na straně hardware
  - zvýšit výkon CPU
  - zrychlit spojení mezi CPU a RAM

Jak urychlit výpočet na sekvečním systému?

- na straně hardware
  - zvýšit výkon CPU
  - zrychlit spojení mezi CPU a RAM
- na straně software

Jak urychlit výpočet na sekvečním systému?

- na straně hardware
  - zvýšit výkon CPU
  - zrychlit spojení mezi CPU a RAM
- na straně software
  - eliminovat zbytečné instrukce programu nebo instrukce náročné na provedení

Jak urychlit výpočet na sekvečním systému?

- na straně hardware
  - zvýšit výkon CPU
  - zrychlit spojení mezi CPU a RAM
- na straně software
  - eliminovat zbytečné instrukce programu nebo instrukce náročné na provedení
  - optimalizovat přístup do paměti

Jak zvýšit výkon CPU

- zkrátit čas nutný ke zpracování 1 instrukce
  - urychlit časovač (Timer) = zvýšení taktu
  - použít tzv. *pipelining* = jistá forma paralelizace
- provádět více operací najednou
  - tím již dostáváme paralelní systém
  - ale tzv. *superskalární zpracování* patří mezi implicitně paralelní systémy

Jak zvýšit výkon CPU

- zkrátit čas nutný ke zpracování 1 instrukce
  - urychlit časovač (Timer) = zvýšení taktu
  - použít tzv. *pipelining* = jistá forma paralelizace
- provádět více operací najednou
  - tím již dostáváme paralelní systém
  - ale tzv. *superskalární zpracování* patří mezi implicitně paralelní systémy

Definition

Je-li některý systém schopný paralelně zpracovávat kód psaný pro sekvenční systém, mluvíme o **implicitně paralelním systému**.

Zvýšení taktu časovače

- to je technicky velmi náročné, poslední dobou se frekvence CPU nemění $\approx 3GHz$
- zvyšování frekvence vede k velkému nárůstu produkovaného tepla
- i samotné zvyšování frekvence vyžaduje zásah do architektury

Pipelining I.

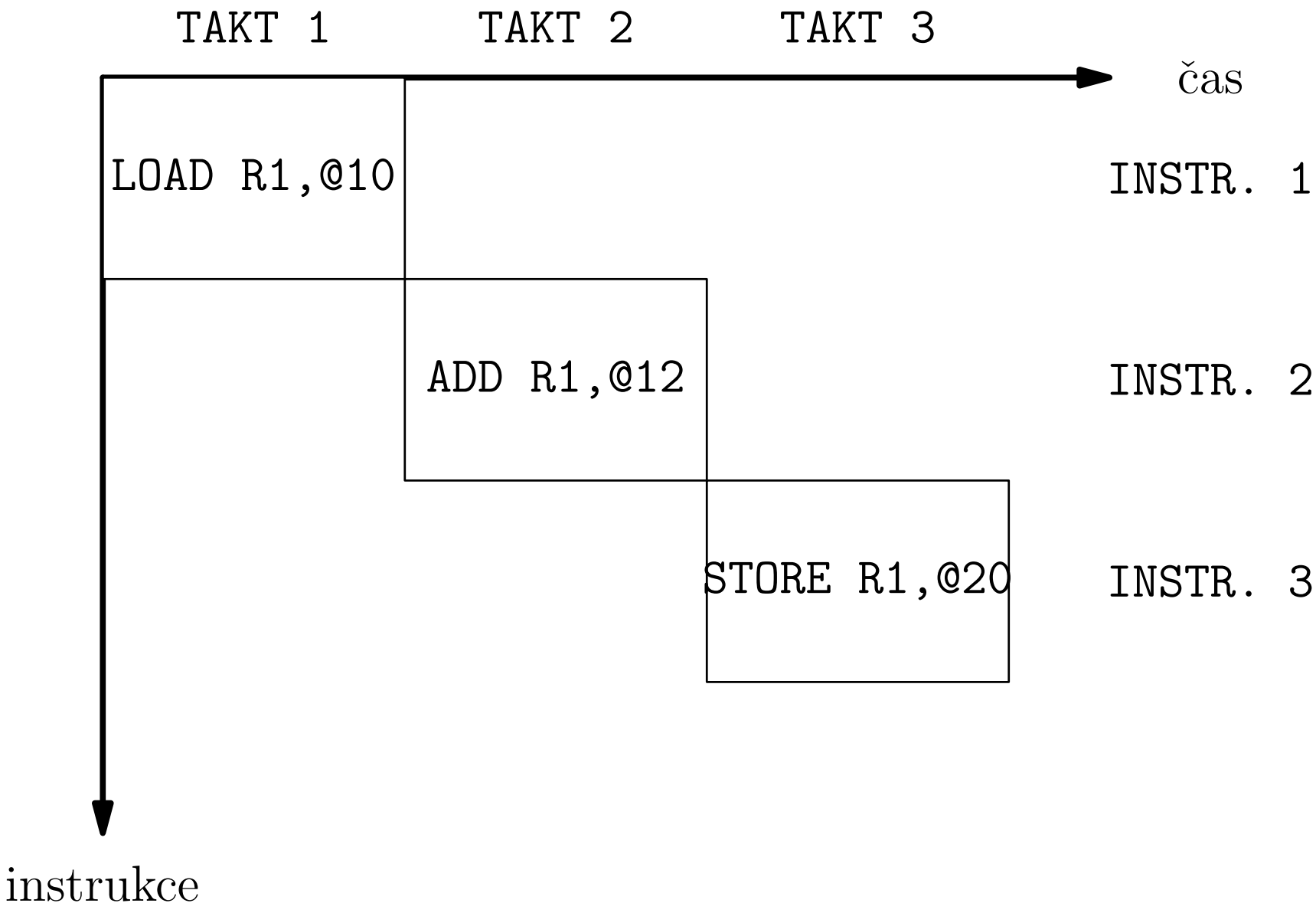
- umožňuje urychlit zpracování instrukcí
- dovolí taktování na vyšší frekvence

Pipelining II.

Vezměme si následující kód:

```
LOAD  R1, @10 # nacti do reg. 1 hodnotu z adresy 10
ADD   R1, @12 # pricti do reg. 1 hodnotu a adresy 12
STORE R1, @20 # zapis reg. 1 na adresu 20
```

Pipelining III.



Pipelining IV.

PIPELINING se podobá výrobní lince

- zpracování instrukce se rozdělí na více kroků, např.

Pipelining IV.

PIPELINING se podobá výrobní lince

- zpracování instrukce se rozdělí na více kroků, např.
  - načtení instrukce (instruction fetch) - IF

Pipelining IV.

PIPELINING se podobá výrobní lince

- zpracování instrukce se rozdělí na více kroků, např.
  - načtení instrukce (instruction fetch) - IF
  - dekódování instrukce (instruction decode) - ID

Pipelining IV.

PIPELINING se podobá výrobní lince

- zpracování instrukce se rozdělí na více kroků, např.
  - načtení instrukce (instruction fetch) - IF
  - dekódování instrukce (instruction decode) - ID
  - provedení instrukce (instruction execution) - IE

Pipelining IV.

PIPELINING se podobá výrobní lince

- zpracování instrukce se rozdělí na více kroků, např.
  - načtení instrukce (instruction fetch) - IF
  - dekódování instrukce (instruction decode) - ID
  - provedení instrukce (instruction execution) - IE
  - načtení operandu (operand fetch) - OE

Pipelining IV.

PIPELINING se podobá výrobní lince

- zpracování instrukce se rozdělí na více kroků, např.
  - načtení instrukce (instruction fetch) - IF
  - dekódování instrukce (instruction decode) - ID
  - provedení instrukce (instruction execution) - IE
  - načtení operandu (operand fetch) - OE
  - zápis operandu (operand write) - OW

Pipelining IV.

PIPELINING se podobá výrobní lince

- zpracování instrukce se rozdělí na více kroků, např.
  - načtení instrukce (instruction fetch) - IF
  - dekódování instrukce (instruction decode) - ID
  - provedení instrukce (instruction execution) - IE
  - načtení operandu (operand fetch) - OE
  - zápis operandu (operand write) - OW
- tyto kroky jsou jednodušší na provedení

Pipelining IV.

PIPELINING se podobá výrobní lince

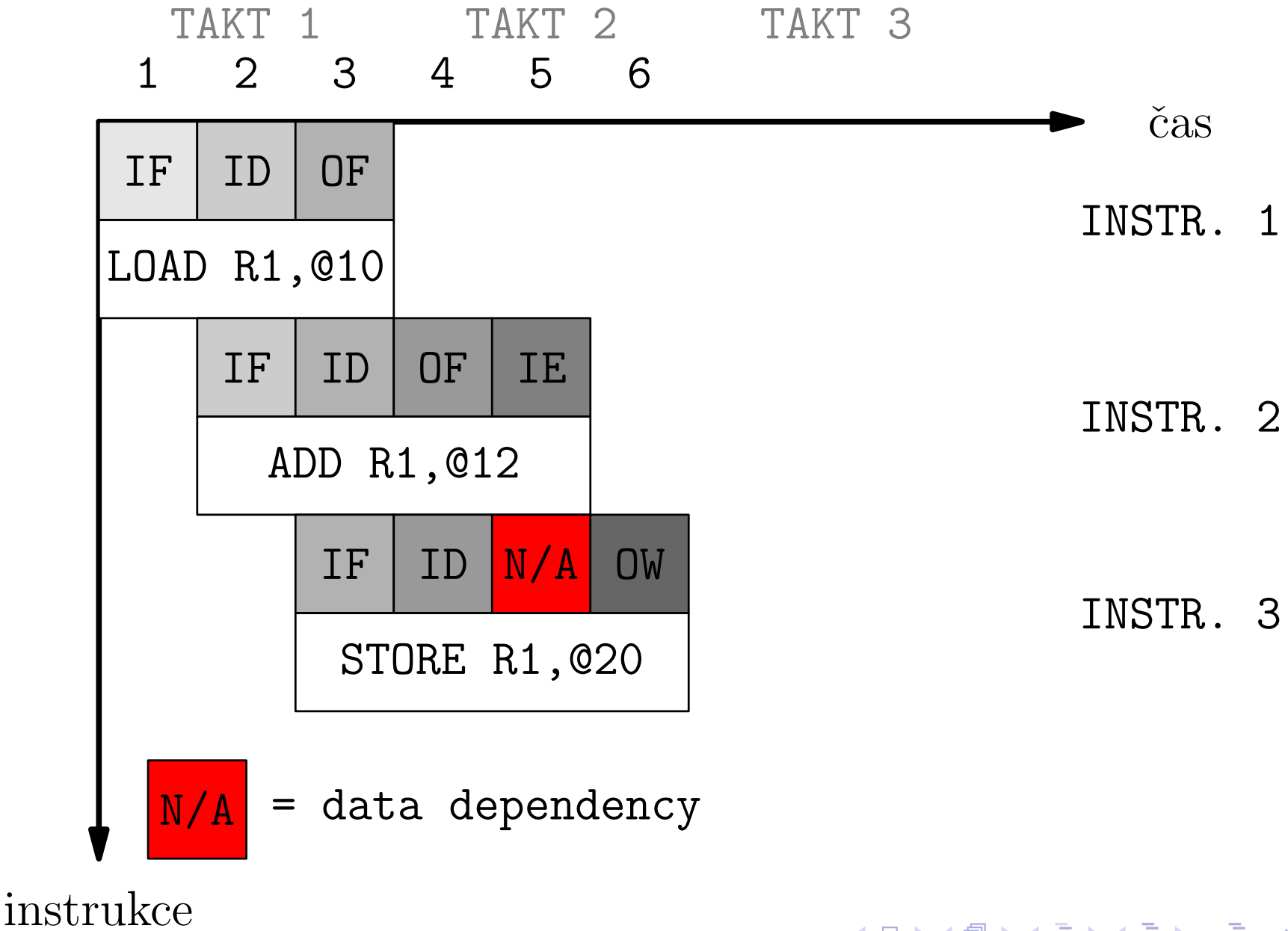
- zpracování instrukce se rozdělí na více kroků, např.
  - načtení instrukce (instruction fetch) - IF
  - dekódování instrukce (instruction decode) - ID
  - provedení instrukce (instruction execution) - IE
  - načtení operandu (operand fetch) - OE
  - zápis operandu (operand write) - OW
- tyto kroky jsou jednodušší na provedení
- díky tomu lze zkrátit délku jednoho taktu - ten nyní odpovídá jednomu kroku zpracování

Pipelining IV.

PIPELINING se podobá výrobní lince

- zpracování instrukce se rozdělí na více kroků, např.
  - načtení instrukce (instruction fetch) - IF
  - dekódování instrukce (instruction decode) - ID
  - provedení instrukce (instruction execution) - IE
  - načtení operandu (operand fetch) - OE
  - zápis operandu (operand write) - OW
- tyto kroky jsou jednodušší na provedení
- díky tomu lze zkrátit délku jednoho taktu - ten nyní odpovídá jednomu kroku zpracování
- provádění lze zřetězit

Pipelining V.



Pipelining VI.

Pipelining se potýká se třemi typy závilostí, které snižují jeho efektivitu:

- **datová závislost** - data dependency

Pipelining VI.

Pipelining se potýká se třemi typy závilostí, které snižují jeho efektivitu:

- **datová závislost** - data dependency
  - nastává ve chvíli, kdy jedna instrukce potřebuje data, která ještě nejsou spočtena

Pipelining VI.

Pipelining se potýká se třemi typy závilostí, které snižují jeho efektivitu:

- **datová závislost** - data dependency
  - nastává ve chvíli, kdy jedna instrukce potřebuje data, která ještě nejsou spočtena
- **závislost na zdrojích** - resource dependency

Pipelining VI.

Pipelining se potýká se třemi typy závilostí, které snižují jeho efektivitu:

- **datová závislost** - data dependency
  - nastává ve chvíli, kdy jedna instrukce potřebuje data, která ještě nejsou spočtena
- **závislost na zdrojích** - resource dependency
  - v situaci, kdy dvě instrukce chtějí přistupovat např. na stejné místo v paměti nebo do stejného registru

Pipelining VI.

Pipelining se potýká se třemi typy závilostí, které snižují jeho efektivitu:

- **datová závislost** - data dependency
  - nastává ve chvíli, kdy jedna instrukce potřebuje data, která ještě nejsou spočtena
- **závislost na zdrojích** - resource dependency
  - v situaci, kdy dvě instrukce chtějí přistupovat např. na stejné místo v paměti nebo do stejného registru
- **závislost na podmínce** - branch dependency

Pipelining VI.

Pipelining se potýká se třemi typy závilostí, které snižují jeho efektivitu:

- **datová závislost** - data dependency
  - nastává ve chvíli, kdy jedna instrukce potřebuje data, která ještě nejsou spočtena
- **závislost na zdrojích** - resource dependency
  - v situaci, kdy dvě instrukce chtějí přistupovat např. na stejné místo v paměti nebo do stejného registru
- **závislost na podmínce** - branch dependency
  - v situaci, kdy není znám výsledek výpočtu, jenž může ovlivnit, jakým způsobem bude kód dále prováděn (např. nejsou data k vyhodnocení podmíněného skoku)

Pipelining VI.

Pipelining se potýká se třemi typy závilostí, které snižují jeho efektivitu:

- **datová závislost** - data dependency
  - nastává ve chvíli, kdy jedna instrukce potřebuje data, která ještě nejsou spočtena
- **závislost na zdrojích** - resource dependency
  - v situaci, kdy dvě instrukce chtějí přistupovat např. na stejné místo v paměti nebo do stejného registru
- **závislost na podmínce** - branch dependency
  - v situaci, kdy není znám výsledek výpočtu, jenž může ovlivnit, jakým způsobem bude kód dále prováděn (např. nejsou data k vyhodnocení podmíněného skoku)
  - udává se, že v průměru se vyskytuje jedna podmínka na každých 5-6 instrukcí

Pipelining VII.

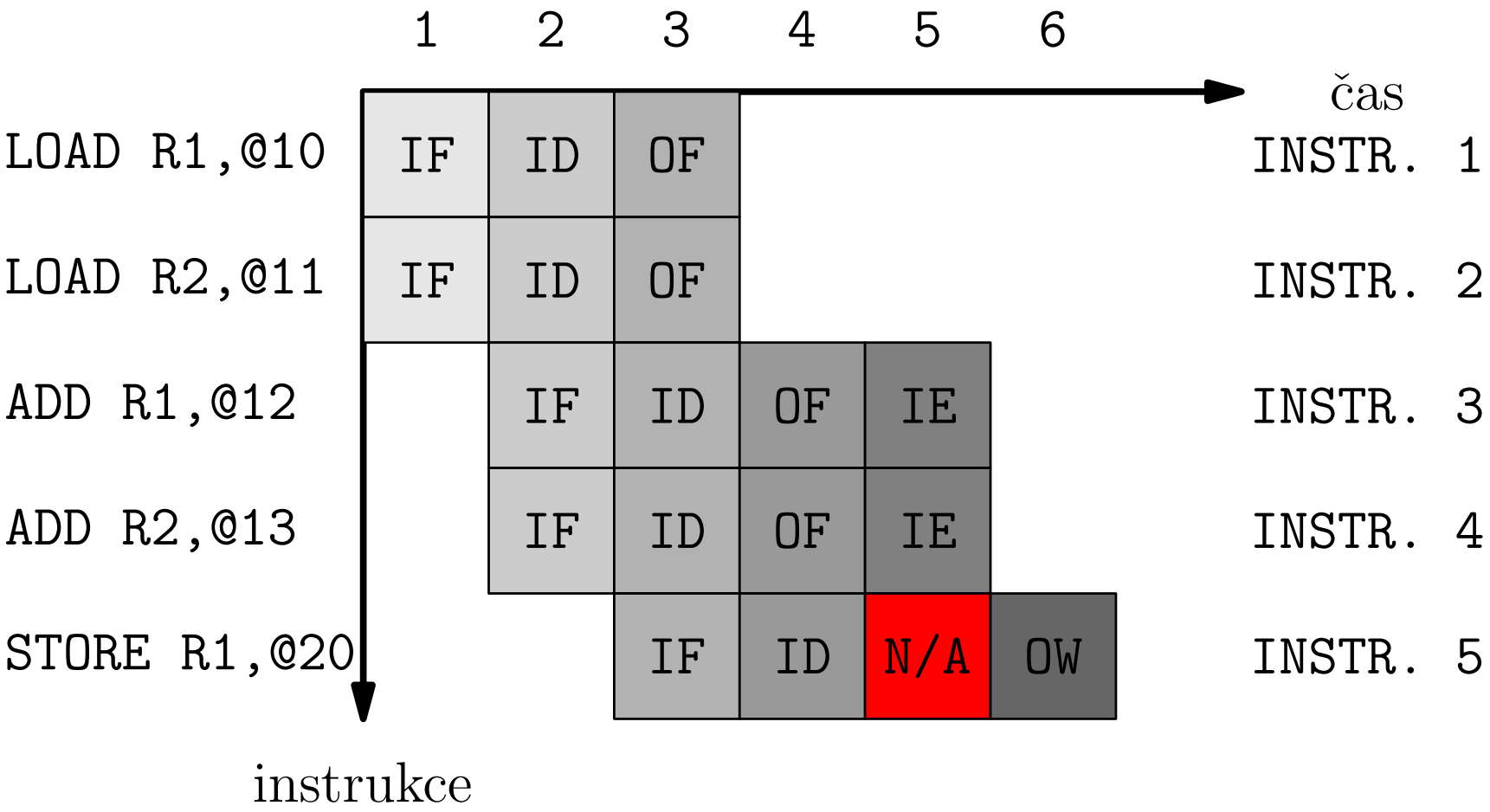
Problémy se závislostmi se řeší pomocí:

- **prováděním instrukcí mimo pořadí** - out of order
  - procesor přehazuje pořadí zpracování instrukcí tak, aby se vyhnul zmiňovaným závislostem
  - může to provádět i překladač při různých optimalizacích
  - většinou nepomáhá u závislostech na podmínce
- **předvídání výsledků operací** - speculative execution
  - procesor se pokusí uhodnout, která větev programu má větší šanci, že bude prováděna
  - tu pak začne pomocí pipeline zpracovávat ještě před vyřešením samotné podmínky
  - to má překvapivě velkou úspěšnost - 80-90%
  - pokud předvídání selže, je nutné začít zpracovávat druhou větev
  - to si vyžaduje vyprázdnění celé pipeline, což vede k velkému zpomalení
  - to je důvod, proč nelze dělat pipeliny příliš dlouhé
    - dělení na menší elementární kroky umožňuje rychlejší taktování
    - Pentium IV má údajně 20-ti stupňovou pipeline

Superskalární zpracování I.

- jsou-li dvě instrukce na sobě zcela nezávislé, je možné je zpracovat obě současně
- lze tak využít dvě nebo více pipeline
- k efektivnímu využití je potřeba mít instrukce dobře setříděné
  - to lze dělat během překladu - překladač
  - nebo za chodu programu - procesor
- Intel zavedl superskalární zpracování v procesorech Pentium I
- GPU - Graphics processing unit - mohou mít několik desítek pipeline

Superskalární zpracování II.



Jak urychlit komunikaci mezi procesorem a pamětí? I.

- výkon CPU roste mnohem rychleji než rychlost RAM
- proto má každý počítač často hned několik úrovní pamětí různé velikosti a rychlosti
  - L1, L2, L3 cache
  - RAM - operační paměť
  - HDD - pomocí swapování může rozšířit virtuální paměť

Jak urychlit komunikaci mezi procesorem a pamětí? II.

Rychlost komunikace mezi pamětí a procesorem je dána:

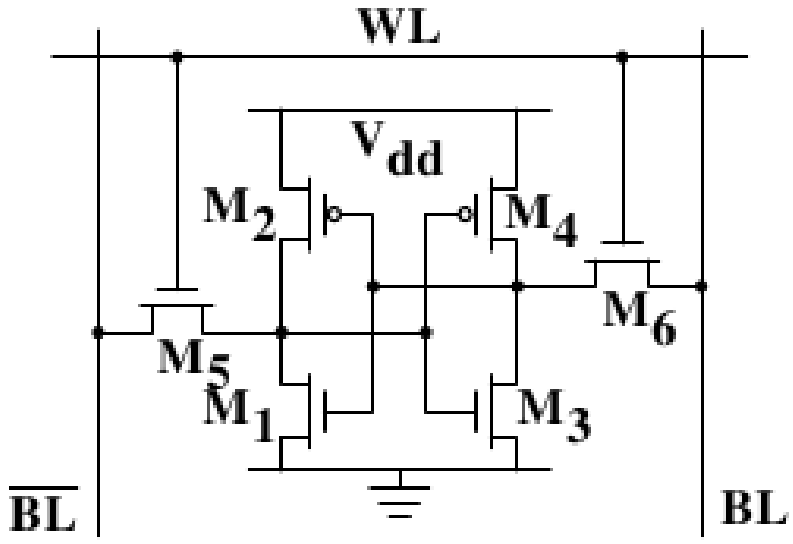
- **latencí** (latency)
  - udává, za jak dlouho je paměťový modul schopný vyhledat požadovaná data
  - je podstatná tehdy, pokud čteme malé bloky dat z různých míst v paměti
- **přenosovou rychlostí** (bandwidth)
  - udává, jak rychle lze přenášet bloky dat po jejich nalezení
  - uplatní se při práci s velkými souvislými bloky dat

Paměťový subsystém

Rozlišujeme dva základní typy pamětí:

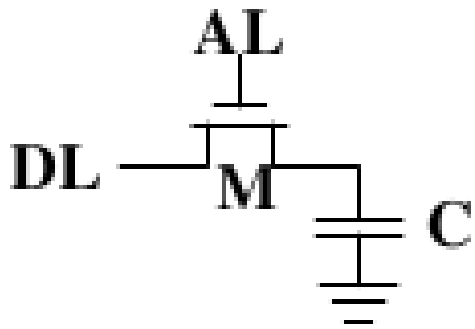
- statická paměť - *static RAM*
- dynamická paměť - *dynamic RAM*

Statická paměť



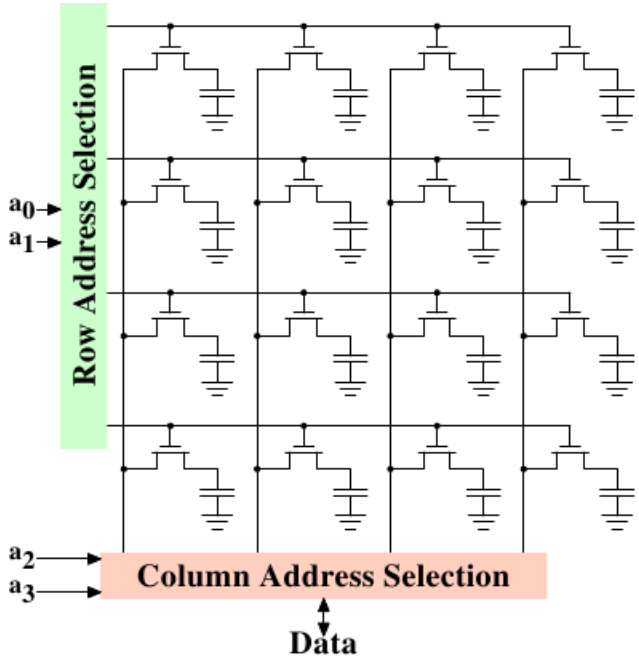
- jeden bit je reprezentován 6 tranzistory
- tato paměť je velmi rychlá a nevyžaduje žádné obnovovací cykly
- využívá se zejména pro cache procesorů
- je velmi drahá

Dynamická paměť



- jeden bit je reprezentován 1 tranzistorem a 1 kondenzátorem
- tato paměť je jednodušší, ale vyžaduje obnovovací cykly pro dobíjení kondenzátoru
- kondenzátory se vybíjejí při čtení, ale i samovolně
- během obnovovacího cyklu nelze s pamětí pracovat
- tento typ paměti je ale výrazně menší (a tedy levnější) než statická paměť a má menší spotřebu elektřiny

Organizace dynamické paměti



- DRAM nelze organizovat lineárně tj. připojit každou paměťovou buňku k CPU přímo
  - při 4 Gb RAM by to vyžadovalo 2^{32} spojů
- adresa buňky se proto zakóduje jako binární číslo a je pak zpracována demultiplexorem
- aby nemusel být demultiplexor příliš složitý, organizuje se DRAM do 2D pole

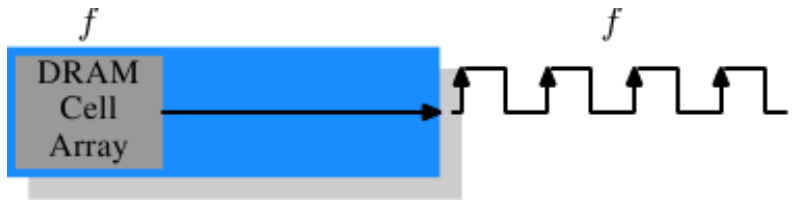
Přístup do DRAM

- celá adresa se rozdělí na index řádku (RAS) a index sloupce (CAS)
- paměťový modul nejprve přijme RAS a "připojí"požadovaný řádek
- následně přijme CAS a "připojí"požadovanou buňku
- pokud pracujeme s několika buňkama v jednom řádku, lze k nim přistupovat jen pomocí poslání jednoho RAS a několika CAS
- to může být výrazně rychlejší
- **je lepší zpracovávat data sekvenčně nebo v malých blocích, než náhodně přistupovat na různá místa v paměti**

Typy pamětí DRAM

Nejčastěji se používá synchroní DRAM tj. SDRAM.

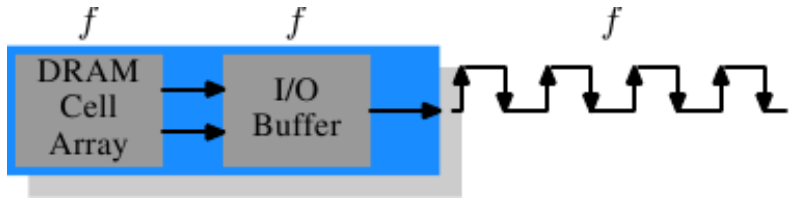
- nejjednodušší typ je SDR SDRAM (*Single Data Rate SDRAM*)



- paměťový modul a sběrnice jsou taktovány stejně tj. 100-266 MHz.
- při 100 MHz je datová propustnost $100 \times 64 Mb/s = 800 MB/s$.

Typy pamětí DRAM

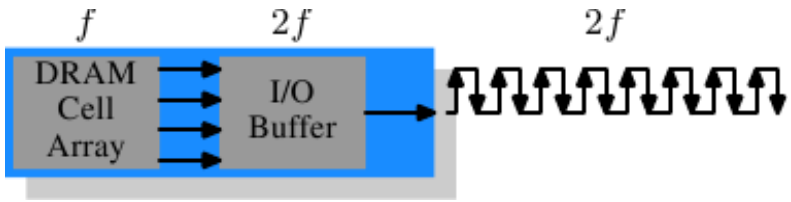
Nástupce pamětí SDR SDRAM je DDR SDRAM (*Double Data Rate SDRAM*)



- narozdíl od SDR SDRAM přenáší data i na sestupné hraně signálu
- aby bylo co přenášet, obsahuje paměťový modul vyrovnávací paměť, která je připojena ke dvěma paměťovým čipům
- při 100 MHz je datová propustnost $2 \times 100 \times 64 Mb/s = 1,600 MB/s$.
- zvýšila se tím datová propustnost, aniž by se navýšil takt paměťových modulů, což by bylo energeticky velmi náročné

Typy pamětí DRAM

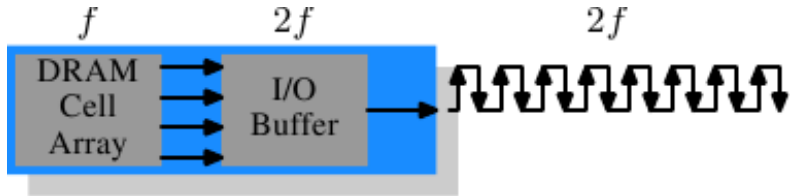
DDR2 SDRAM - zvyšuje frekvenci přenosu dat na dvojnásobek



- paměťové čipy jsou taktovány stejně, ale řadič pracuje rychleji
- proto je potřeba číst v jednom taktu z celkem 4 čipů
- při 100 MHz je datová propustnost $4 \times 100 \times 64 Mb/s = 3,200 MB/s$
- při 266 MHz je datová propustnost $4 \times 266 \times 64 Mb/s = 8,512 MB/s$

Typy pamětí DRAM

DDR3 SDRAM - zvyšuje frekvenci přenosu dat na čtyřnásobek



- při 100 MHz je datová propustnost
  $8 \times 100 \times 64 Mb/s = 6,400 MB/s$
- při 266 MHz je datová propustnost
  $8 \times 266 \times 64 Mb/s = 17,024 MB/s$

Typy pamětí DRAM

- existují už i paměti typu DDR4
- s tím, jak roste frekvence pro přenos dat, je problém připojit více paměťových modulů k jednomu CPU
- rozdíl ve fyzické vzdálenosti od CPU způsobuje problémy se synchroním časováním
- řešením jsou paměti FB-DRAM (Fully Buffered DRAM)
- využívají buffer pro řešení problému s nedokonalou synchronizací

Typy pamětí DRAM

- viděli jsme, jak dochází k urychlení přenosu dat mezi CPU a RAM
- bylo ho dosaženo jen díky paralelnímu přístupu do více paměťových modulů
- z modulu se čte blok 64 bitů = 8 bytů najednou
- pokud program ale využije jen jeden byte, není datová sběrnice využita plně efektivně
- podobný trik využívá i technologie Dual/Triple channel
  - data se ukládají do dvou až tří paměťových modulů napřeskáčku

Žádná z jmenovaných technologii zatím výrazně nesnížila latenci.

Jak urychlit spojení mezi procesorem a pamětí? III.

Za tím účelem se používají malé a rychlé vyrovnávací pamětí postavené na čipech SRAM - **cache**

- jde o poměrně malou ale velice rychlou paměť
  - velikost je řádově 1/1000 velikosti operační paměti
  - přístup do RAM je cca. 200 cyklů CPU
  - přístup do cahce je 3-15 cyklů CPU
- často je implementována přímo na stejném čipu jako procesor a běží na stejném taktu
- její správa je plně v režii CPU, programátor nebo OS mají jen malou možnost ovlivnit práci s cache
- využívá faktu, že u operační paměti je snažší zvýšit přenosovou rychlost, než zkrátit latenci
- do cache se načítají větší souvislé bloky dat, ke kterým je pak rychlejší přístup

Cache

- procesor předpokládá, že právě běžící program pracuje s menšími ucelenými bloky dat a nepřistupuje do operační paměti na zcela náhodné adresy
  - tomu se říká prostorová lakolita - *spatial locality*
- dále se předpokládá, že program nejprve zpracuje jeden blok dat, pak se posune k dalšímu bloku a k tomu původnímu se nevrací
  - to je časová lokalita - *temporal locality*

Cache

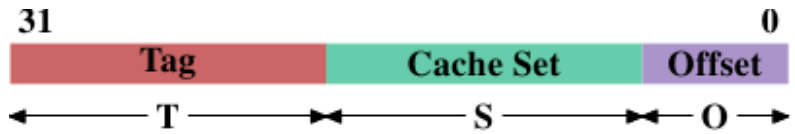
- procesor provádí tzv. *prefetching*, tj. načítání dat s předstihem
- pokouší se udhadnout, jaká data budou brzy potřeba a načíst si je do cache dřív, než si o to řekne program sám
- pak je schopen k nim poskytnout mnohem rychlejší přístup
- cache se dělí na
  - instrukční
  - datovou
- to je rozpor s von Neumannovou architekturou

Jak funguje cache?

- data se do cache přenáší ve větších blocích, nejčastěji 64 bytů, což se provede v celkem 8 přenosech
- SDRAM jsou pro tento přenos optimalizovány a ušetří se tím vysílaní RAS a CAS signálů
- bloku dat, který je přenesen do cache najednou se říká *cache line*
- každá cache line musí být označnea podle své adresy v operační paměti
- pokud chce program pracovat s určitou adresou, musí procesor nejprve zjistit, zda má tato data kešovaná nebo ne
- to musí být pochopitelně velmi rychlé

Plně asociativní cache

- nejjednodušší přístup k organizování cache je *fully associative cache*
- mějme pro jednoduchost systém s 32-bitovým adresováním
- celá adresa se rozdělí na tag a offset
- offset odpovídá adrese v rámci cache line, tj. má velikost $O = 6$ bitů
- zbývající část $T = 26$ bitů připadne tagu

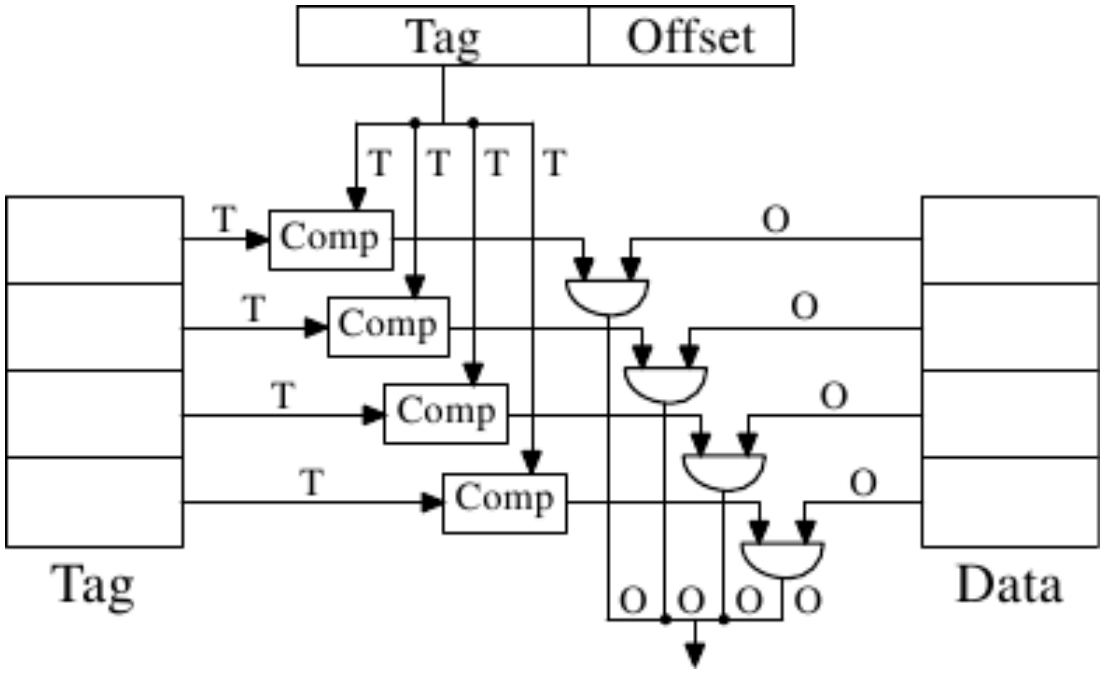


Obrázek: $T = 26$, $S = 0$ a $O = 6$

Plně asociativní cache

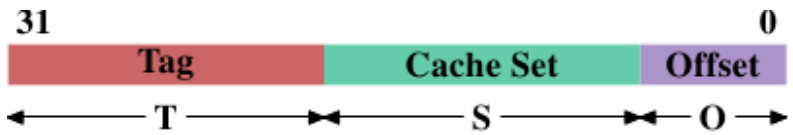
- u plně asociativní cache přísluší každé cache line jeden komparátor, který porovná tag uložené cache line s cache line, která je vyžadovaná od CPU
- to je velmi rychlé
- umožňuje to efektivně kešovat cache liny libovolně rozmístěné v paměti
- při velikosti cache 4 MB, máme celkem $4M/64 = 65,536$ komparátorů
- to by bylo velice náročné z hardwarovéhoi pohledu a drahé na výrobu

Plně asociativní cache



Cache s přímým mapováním

- mějme opět 4 MB cache tj. $65,536$ cache line
- celou operační paměť si můžeme rozdělit právě na $65,536$ segmentů, kterým se říká cache set
- každou cache set lze adresovat pomocí 16 bitů
- spolu s offsetem to dělá $16 + 6 = 22$ bitů
- zbylých 10 bitů zůstává pro tag

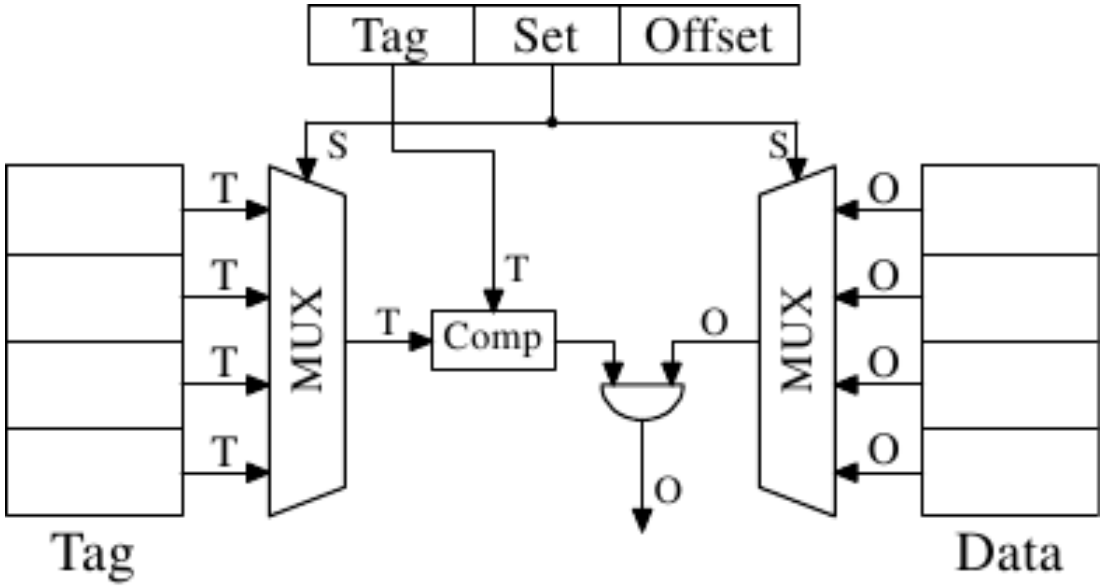


Obrázek: $T = 7$, $S = 16$ a $O = 9$

Cache s přímým mapováním

- pokud nyní CPU potřebuje vyhledat data v cache, podívá se na část adresy odpovídající cache set
- okamžitě ví, kam má do cache sáhnout (to provádí multiplexor)
- aby zjistil, zda je zde uložena požadovaná cache line, provede porovnání těch částí adresy, které odpovídají tagu
- na to stačí jeden komparátor

Cache s přímým mapováním



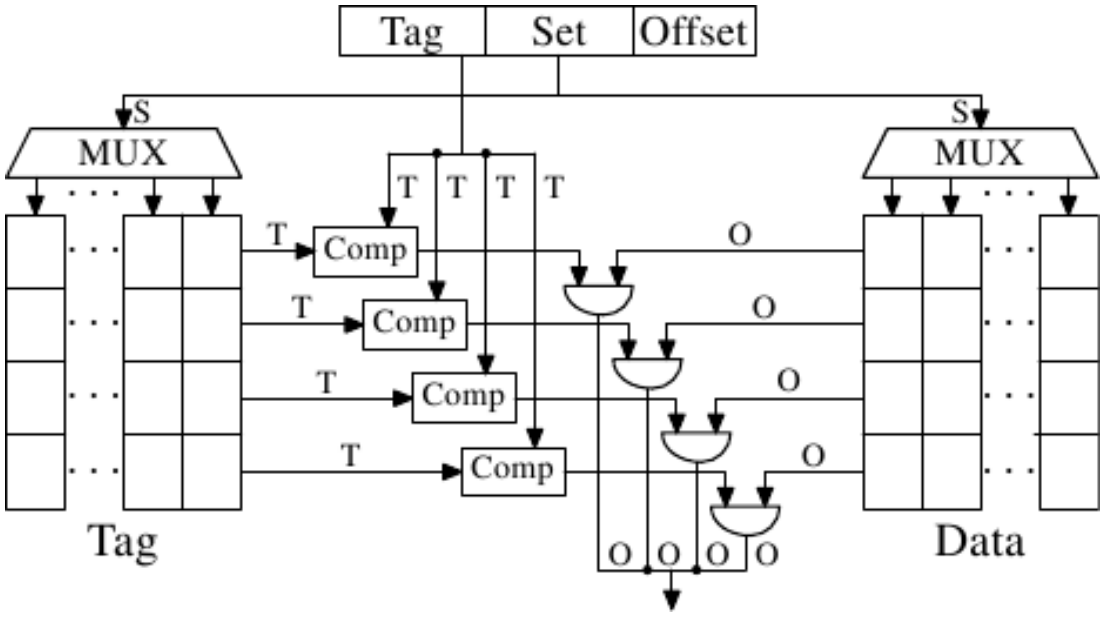
Cache s přímým mapováním

- tento typ keše je mnohem jenodušší na konstrukci
- problém je, že nelze mít v keši dvě cache line, které se liší jen tagem
- takové adresy se opakují po $2^{16+4} = 4M$
- to sice dobře souhlasí s podmínkou spatial locality, ale praxe ukazuje, že tento přístup není nejlepší
- v praxi se volí kombinace obou přístupů

Set associative cache

- celou keš rozdělíme na menší počet cache sets, ale do každé z nich umožníme uložit větší počet cache line – 2, 4 nebo 8
- velikost cache set adresy se zmenší o $1, 2$ nebo 3 bity a tagová část se příslušně zvětší
- multiplexor vybere příslušnou cache set
- v ní je nyní 2 až 8 cache lines
- pomocí 2 až 8 komparátorů se provede porovnání tagové adresy

Set associative cache



Set associative cache

L2 Cache Size	Associativity							
	Direct		2		4		8	
	CL=32	CL=64	CL=32	CL=64	CL=32	CL=64	CL=32	CL=64
512k	27,794,595	20,422,527	25,222,611	18,303,581	24,096,510	17,356,121	23,666,929	17,029,334
1M	19,007,315	13,903,854	16,566,738	12,127,174	15,537,500	11,436,705	15,162,895	11,233,896
2M	12,230,962	8,801,403	9,081,881	6,491,011	7,878,601	5,675,181	7,391,389	5,382,064
4M	7,749,986	5,427,836	4,736,187	3,159,507	3,788,122	2,418,898	3,430,713	2,125,103
8M	4,731,904	3,209,693	2,690,498	1,602,957	2,207,655	1,228,190	2,111,075	1,155,847
16M	2,620,587	1,528,592	1,958,293	1,089,580	1,704,878	883,530	1,671,541	862,324

Tabulka: Efekt velikosti cache, asociativty a cache line na počet "cache misses". Testováno na gcc.

Různé úrovně cache

- současné procesory obsahují několik úrovní keší
- označují se L1, L2 a L3
- L1 se dělí na datovou a instrukční
- čím menší číslo, tím měnší ale rychlejší keš

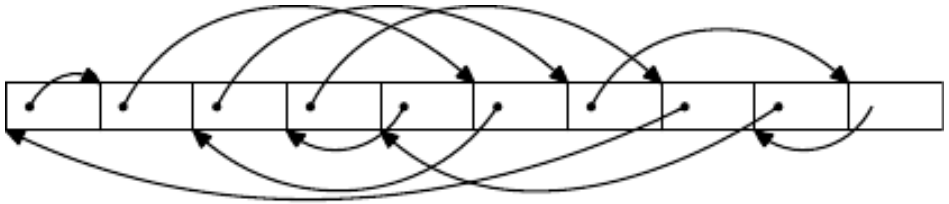
Efekt keše

Příklad:

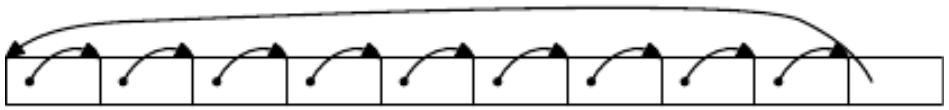
Budeme procházet následující spojový seznam:

```
1  struct element
2  {
3      struct element* next;
4      long int pad[ NPAD ];
5  }
```

- všechny prvky seznamu se alokují jako velké pole
- následně se propojí buď sekvenčně nebo náhodně



Random

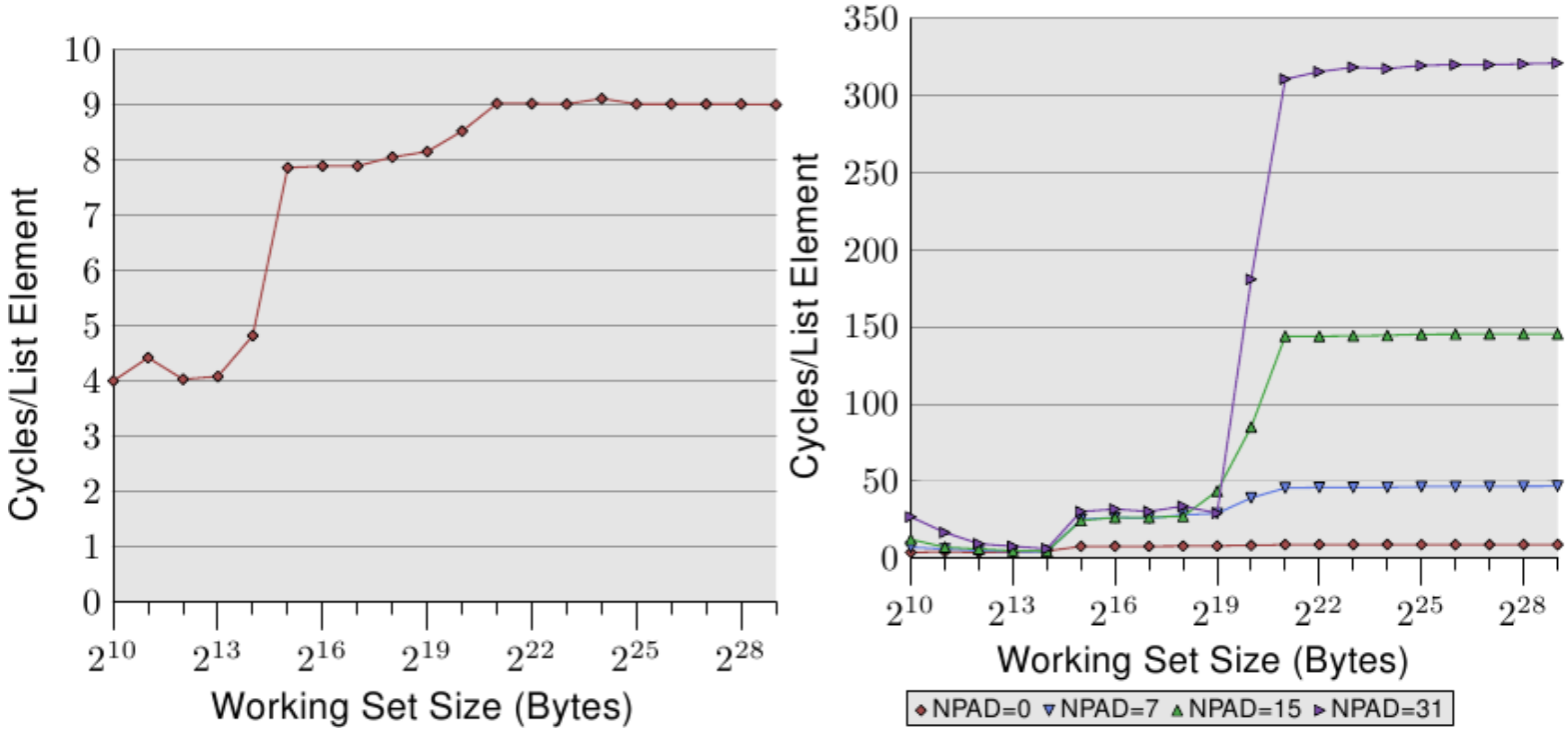


Sequential

Efekt keše

- seznam budeme pouze procházet, nebudeme číst ani měnit data v `pad`
- hodnota `NPAD` simuluje různou velikost jednoho prvku seznamu
- `long int` má stejnou velikost jako ukazatel, tj. 32 nebo 64 bitů podle toho, jestli máme 32 nebo 64 bitový systém
- hodnota `NPAD` způsobuje mezery mezi jednotlivými ukazately

Efekt keše - prefetching



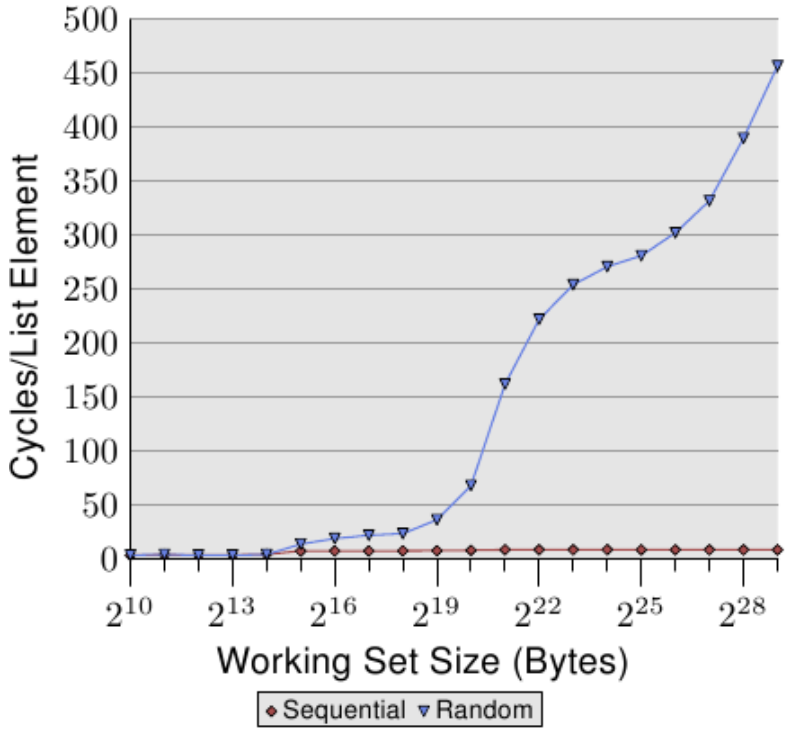
Obrázek: Výsledky pro sekvenční procházení seznamu na Intel Pentium 4 s 16Kb (2^{16}) L1 cache a 1MB (2^{20}) L2 cache.

- RAM $\approx$ 200 cyklů
- L2 cache $\approx$ 15 cyklů
- L1 cahce $\approx$ 4 cykly

Efekt keše - prefetching

- procesor provádí prefetching, tj. načítá několik cache line dopředu
- je-li `NPAD = 0`, do jedné cache line se vejde 8 až 16 prvků seznamu
- je-li `NPAD = 7` je to jen 1 nebo 2
- je-li `NPAD = 31` je jeden element větši než cache line

Efekt keše - sekvenční a náhodné čtení



Obrázek: Výsledky pro sekvenční a náhodné procházení seznamu na Intel Pentium 4 s 16Kb (2^{16}) L1 cache a 1MB (2^{20}) L2 cache.

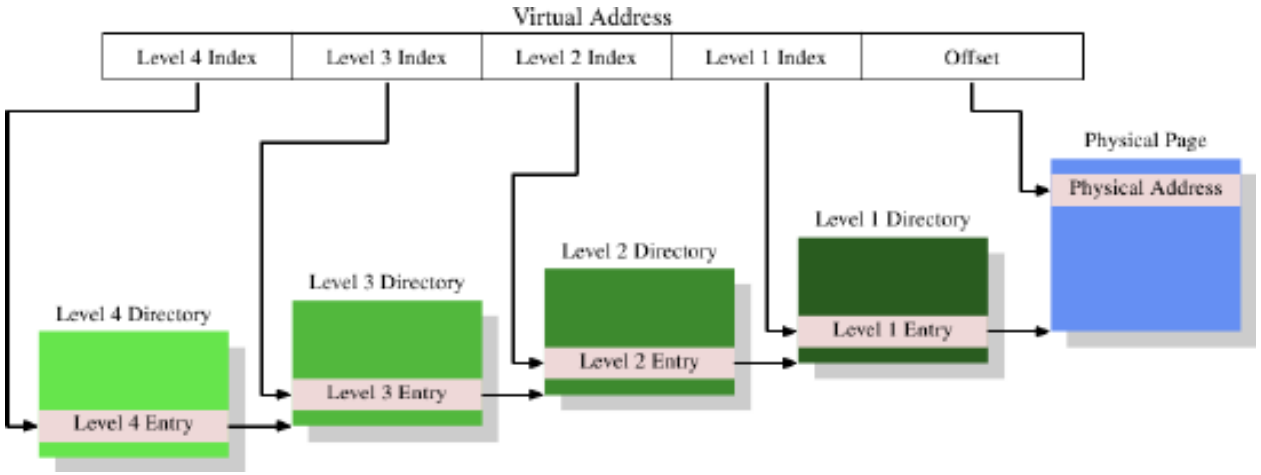
Zapisování dat

- pokud je nějaká cache line změněna, jsou dvě možnosti, co udělat
  - provést okamžitý zápis do RAM – *write-through cache*
    - je to jednoduchá strategie, ale na příliš účinná
    - pokud program opakovaně zapisuje do stejné proměnné, je většina zápisů do operační paměti zbytečná
  - provést zápis do RAM později – *write-back cache*
    - cache line se označí jako *dirty*
    - ve chvíli, kdy je potřeba uvolnit nějaká data z cache, kontroluje se zda cache line není takto označena a pokud ano, provede se zápis do RAM

Virtuální paměť a TLB

- aby mohl operační systém současně provozovat více aplikací/procesů, je nutné implementovat virtuální paměť
- každý proces pak vidí svůj vlastní adresový prostor a nemůže ovlivňovat ostatní procesy
- proces udává virtuální adresy ve svém a.p., které je pak nutné přložit na fyzické adresy
- paměť se procesům přiděluje po stránkách o velikosti 4kB
- operační systém spravuje systém tabulek, které adresují jednotlivé stránky

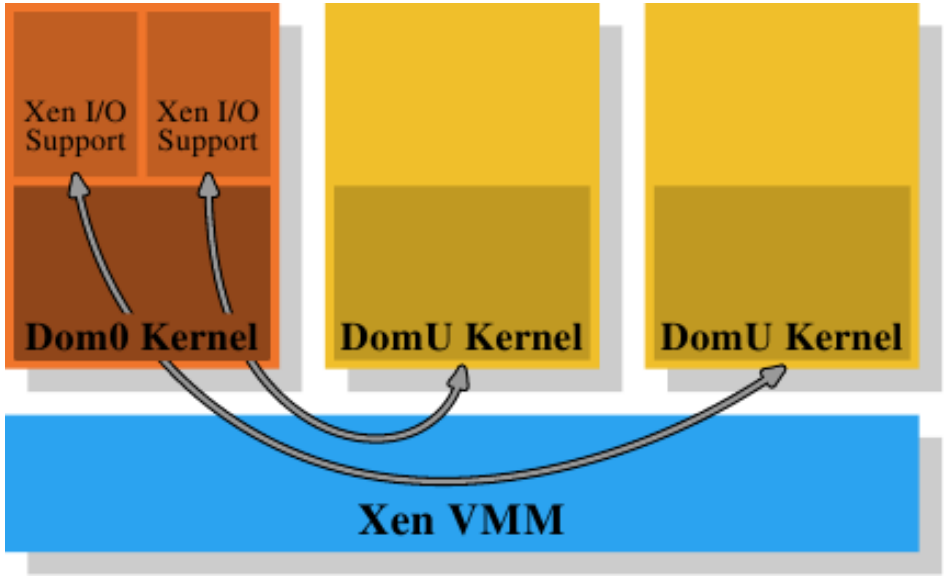
Virtuální paměť a TLB



Virtuální paměť a TLB

- než operační systém zjistí skutečnou fyzickou adresu, musí prohledat několik tabulek
- to je velmi pomalé
- pro urychlení provádí samotný překlad přímo procesor
- navíc se používá speciální cache pro již přeložené adresy
- té se říká TLB - *translation look-aside buffer*
- jdem o velmi malou cache o velikosti cca. 128 nebo 256 adress
- je velice rychlá, většinou fully associative
- dělí se na instrukční a datovou
- její obsah se zahazuje při každém přepnutí mezi různými procesy
- pokud chce jeden proces efektivně využít TLB, měl by pracovat s omezeným počtem paměťových stránek

Virtuální paměť a TLB



- virtualizace výrazně znesnadňuje překlad adres a využití TLB
- vzniká ještě jedna vrstva, takže to co původně byla fyzická adresa je nyní jen adresa adresového prostoru daného OS a musí být přeložena ještě jednou
- běžný procesor na to není vybaven
- moderní procesory mají rozšíření jako
  - Extended Page Tables – Intel
  - Nasted Page Tables – AMD

Jak psát efektivní kód pro sekvenční architektury?

- vždy vyvýjíme nejprve co nejjednodušší správně fungující kód
- je-li to nutné, provádíme optimalizaci
- předčasná optimalizace je cesta do pekel!!!
- neprve je dobré zvážit, zda jsme zvolili vhodný algoritmus
- potom optimalizujeme kód

Jak programovat pro efektivní využití CPU?

- pro optimální využití pipeline je dobré minimalizovat vyšetřování podmínek
  - podmínky uvnitř cyklů je dobré vytáhnout před cyklus
  - využít tzv. **rozbalování smyček** (loop unrolling)

Jak programovat pro efektivní využití cache?

Pro efektivní využití cache je nutná tzv. **temporal locality**.

- jakmile jsou určitá data jednou načtena do cache, je nutné toho co nejvíce využít
- snažíme se nejdříve zpracovat jeden blok dat, a pokud je to možné, už se k němu nevracet, protože později již může být odstraněn z cache z důvodu uvolnění místa pro další data

Jak programovat pro efektivní využití paměťové sběrnice?

Pro efektivní využití paměťové sběrnice je nutná tzv. **spatial locality**.

- šířka sběrnice udává minimální počet bitů, které jsou vždy přeneseny - tzv. **cache line**
  - bývá to 128 nebo 256 bitů
- načteme-li jeden `char` tj. 1 byte, je dobré, aby např. další `char`, který budeme potřebovat, následoval ihned za tím prvním
  - tím pádem je už v cache a ušetříme další přistup do paměti, který může trvat 100-500 cyklů CPU
- dynamicky alokované seznamy mohou mít velice špatnou space locality, je lepší je nahradit souvislými poli
  - to se hodně využívá např. při implementaci řídkých matic
  - uložení plné matice je pro účely násobení s vektorem vhodnější po řádcích, než po sloupcích

Jak programovat pro efektivní využití paměťové sběrnice?

Dále je možné využít:

- **prefetching** - načítání dat s předstihem
- **multithreading** - když jedno vláka čeká na data, je prováděno to druhé