SArchitektura - Řešené zkoušky

stačí stručně, DONE znamená vyřešenou otázku, pls je request o to, aby otázku někdo vyřešil

00 - 2017 06 14

Otázka 1

Otázka 2

Otázka 3

Otázka 4

Otázka 5

Otázka 6

Otázka 7

Otázka 8

Otázka 9

Otázka 10

Otázka 11

Otázka 12

Otázka 13

00 - 2018 05 31

Otázka 1

Otázka 2

Otázka 3

Otázka 4

Otázka 5

Otázka 6

Otázka 7

Otázka 8

Otázka 9

Otázka 10

Otázka 11

Otázka 12

Otázka 13

24 - 2020 06 05

Otázka 1

Otázka 2

Otázka 3

Otázka 4

Otázka 5

Otázka 6

Otázka 7

Otázka 8

Otázka 9

Otázka 10

Otázka 11

Otázka 12

27 - 2020 06 18

Otázka 1

Otázka 2

Otázka 3

Otázka 4

Otázka 5

Otázka 6

Otázka 7

Otázka 8

Otázka 9

Otázka 10

Otázka 11

28 - 2020 06 24

Otázka 1 [1] - DONE

100 miliard instrukcí

Chceme aby processor P2 bezel 9 s misto 10 s a proto musime zmensit pocet instrukce na 10%, a proto to bude 27*10**9

Otázka 2 [1] - DONE*

Můžeme si udělat pravdivostní tabulku pro logickou funkci řadiče (v tomto případě je to kombinační obvod). Tedy známe kontrolní slova. Ty pak můžeme zapsat do ROMky adresované instrukcemi.
ROMka se hodí, protože to může být třeba EPROM a my můžeme řadič později updatovat.
* - mám pocit, že pro to je ještě nějaký lepší důvod

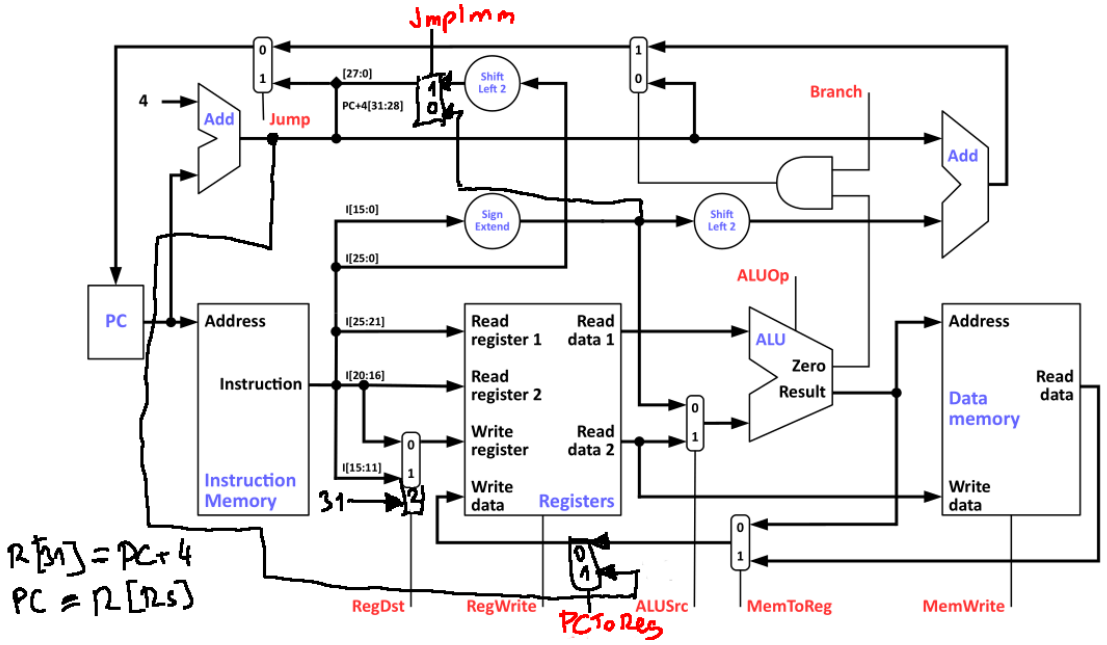
Otázka 3 [2] - DONE

Uplna scitacka ma vstupy a1, b1 a ci a vystupy s1 a ci+1

S = A AND !B AND ! Ci + B AND !A AND !Ci + Ci AND !A AND !B + Ci AND A AND B
S = A XOR B XOR Ci
Ci+1 = A AND B AND !Ci + A AND B AND Ci + Ci AND B AND !A + Ci AND A ABD !B + Ci AND A AND Bo
Ci+1 = A AND B + Ci (A XOR B)

A1	B1	CI	XOR	Ci+1	Si
0	0	0	0	0	0
0	1	0	1	0	1
0	1	1	0	0	0
1	1	1	0	0	1
1	0	0	0	0	1
1	1	0	1	0	1
1	0	1	1	1	1
0	0	1	1	1	0

Otázka 4 [2] - DONE



Jump = 1
RegDst = 2
RegWrite = 1
PCToReg = 1
AluSrc = to je jedno
ALUOp = to je jedno
MemToReg = to je jedno
Branch = to je jedno
MemWrite = 0

Otázka 5 [1] - Done

Pokud jednocyklove - musi to proste byt soucet vsech faze, coz je 200+150+120+190+140 = 800 ps, pokud chceme udelat z toho pipeline, tak musi kazda faze mit stejnou delku, coz je maximalni delka ze vsech stupnu 200*5 = 1000 ps.

Otázka 6 [2] - DONE

Mějme třeba 1bitový dynamický prediktor skoků. Má dva stavy: T a NT. Potom máme-li cyklus, který se zopakuje 5x a prediktor je na začátku ve stavu NT, vypadá sekvence predikcí nějak takhle:
1 prediktor si myslí NT, ale mýlí se
2 prediktor si správně myslí T
3 prediktor si správně myslí T
4 prediktor si správně myslí T

5 prediktor si myslí T, ale mýlí se
Prediktor se splete jednou až dvakrát.

Máme-li vnořený cyklus, kde vnitřní se provádí 4x a vnější 3x. Prediktor je zase na začátku NT:

vnější	vnitřní	predikce	ve skutečnostik
1	1	NT	T
1	2	T	T
1	3	T	T
1	4	T	NT
1		NT	T
2	1	T	T
2	2	T	T
2	3	T	T
2	4	T	NT
2		NT	T
3	1	T	T
3	2	T	T
3	3	T	T
3	4	T	NT
3		NT	NT

Prediktor se splete poka ždé, když vylezeme z vnitřního cylku a chceme skočit na začátek vnějšího.

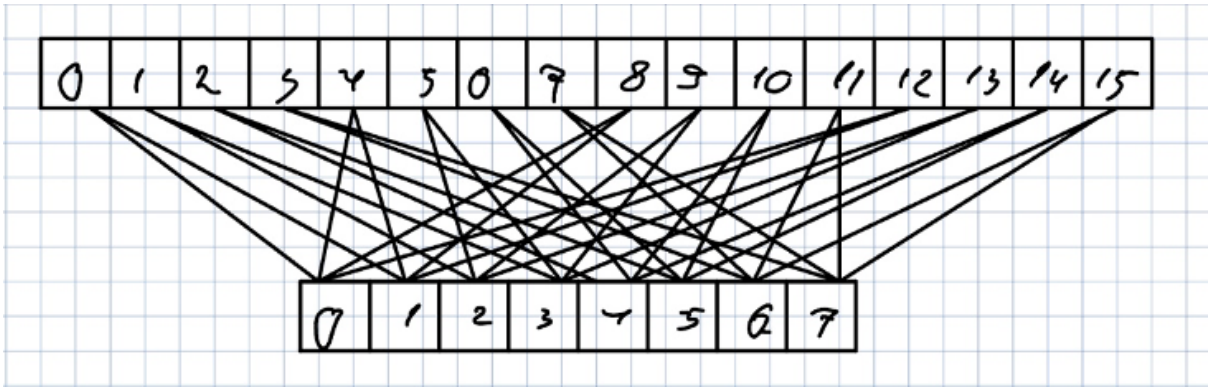
Otázka 7 [1] - DONE*

DRAM. Oproti SRAM je pomalejší, ale levnější (na jednom kondenzátoru lze uložit více bitů najednou, ale to jsme si neříkali). Od DRAM se vyžaduje co nejmenší cena na bit. Od SRAM požadujeme rychlost.
* - Není nějaký lepší důvod pro to, proč je DRAM levná?

Otázka 8 [2] - DONE

Konfliktní miss způsobuje malá asociativita cache. Program chce pracovat s až moc cache linami, které mají v cachi stejný index a tedy je často nutné některou z nich z cache vyhodit, což když se k ní procesor zase snaží přistupovat způsobí miss. Conflict missy budou nastávat, pokud máme malou asociativitu cache (a taky pokud máme malou kapacitu, protože to znamená, že hodně cache line bude sdílet stejný index).

Otázka 9 [2] - DONE



Adresu rozdelime na tri casti: tag + index + offset. Index se resi kam do cache chceme dat, offset je v ramci bloku pameti, tag je zbytek adresy, pouzivame ho abychom zarucili je to spravny blok
Pokud velikost cache line je 4 bajty, tak na offset potrebujeme 2 bity, mame 4 setu v cachu a proto potrbebujeme 2 bity pro index, zbytek 12 bitu je tag. Musime ulkadat tag pro kazdou polozku a proto mame (11+1)*8=96 bitu overhead.

Otázka 10 [2] - DONE

Write through – zapis probiha inhed do pameti I do cache(pokud je dany dany blok pritomen).
Write-back – zapis provadi pouze do cache, pamet se aktualizuje az pri vytesneni daneho bloku
Write-allocate znamena, ze pri write miss, chybejici blok ze nacte do cache a pak zapise.
Write-no-allocate – data se zapisi primo do pameti.
Pro write through ma vetsi vyhodu write no allocate, protoze zapis jde rovnou do pameti, neni potreba plnit cache blok, ktery mozna nebude dale pouzit
Pro write-no-allocate ma smysl pouze write-allocate, write-no-allocate nedava smysl, data nejsou v cache, nemame kam zapisovat

Otázka 11 [2] - DONE

False sharing je situace kdyz vice vlaken pristupuji ke svym vlastnim promennym, ale teto promenny fizucky nacharzi ve stejne cache line a kvuli tomu mame neustale synchornizace mezi processory

```
int[1000] counters

void worker_counter(int worker_id){
        while(true){
            counters[worker_id]++;
        }
}
```

Je mozne vyresit pridanim padding ke kazde promenne, a musi mit velikost stejnou jako cache line

Otázka 12 [2] - DONE

viz v testu 30

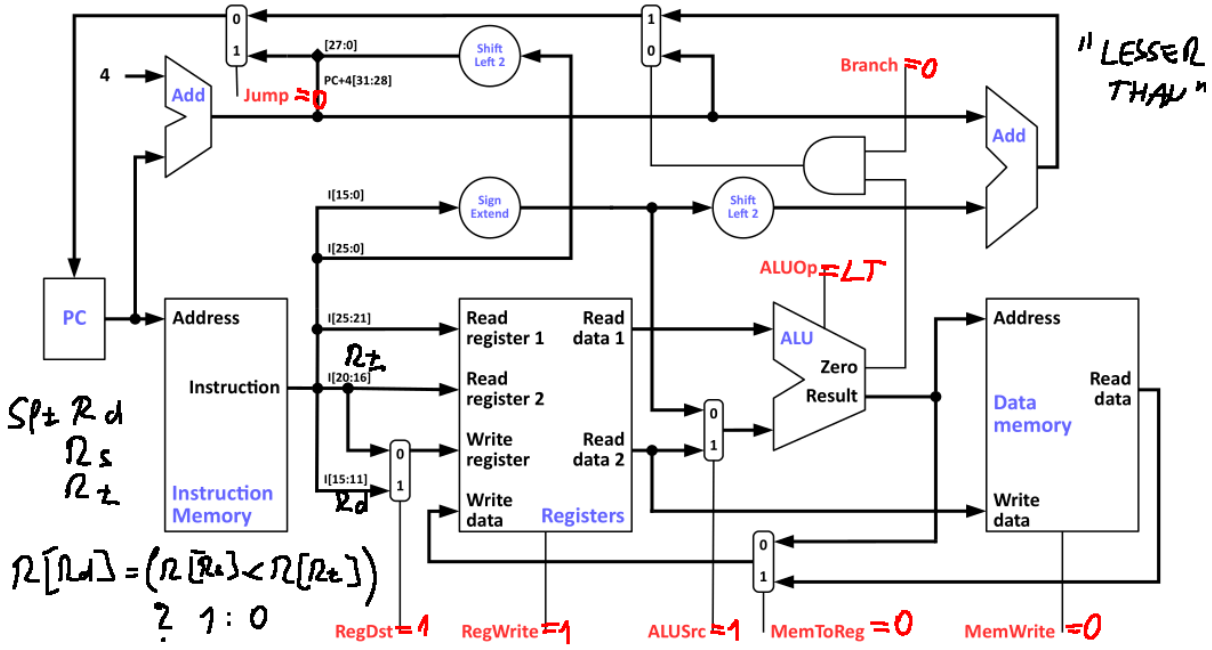
29 - 2020 07 01

Forum post: [Odkaz na forum]

Otázka 1 [1] - DONE

Funkce f(op code) -> control word, kterou implementuje ROM je vlastně jen logická funkce a tu určitě můžeme implementovat i pomocí kombinačního obvodu.
Použít kombinační obvod se hodí, když chceme ušetřit peníze. Můžeme totiž použít jen to nejmenší možné množství transistorů potřebné na implementaci funkce. Zároveň řešení pomocí kombinačního obvodu je, myslím, rychlejší. Na druhou stranu, nebudeme mít možnost řadič později updatovat, jako bychom měli, kdybychom použili EPROM.

Otázka 2 [2] - DONE



V levé části obrázku dělá datová cesta dvě věci. Potencionálně může poslat immediate hodnotu do PC, ale to nechceme a proto je Jump=0. Zároveň zde vybírá v registrovém poli, do kterého registru se bude zapisovat. RegDst=1, jelikož chceme, aby tenhle registr vybírala 15:11 část instrukce, tedy Rd část. Zároveň RegWrite=1, jelikož opravdu chceme zapisovat do registru.

U ALU si vybíráme, jestli jí jako druhý vstup půjde hodnota druhého read registru nebo immediate hodnota. My porovnáváme dva registry a proto ALUSrc=1. Chceme od ALU operaci, která vydá jedničku, pokud první read registr obsahuje data menší než druhý read registr. Tuhle operaci jsem nazval LT. Musel jsem si jí vymyslet, protože bez úprav v datapath bychom podle mě jinak Slt instrukci vykonat nemohli.

Branch=0, protože nechceme branchovat. MemToReg=0 a MemWrite=0, jelikož z pamětí nezacházíme.

Otázka 3 [2]

Otázka 4 [2] - DONE

lw $1, 0 ($2) ; load
add $1, $2, $3 ; use
- způsobuje data hazard
- nelze se mu vyhnout forwardingem, jelikož ve chvíli, kdy je add v EX a potřebuje výsledky , je lw teprve v MA a svoje výsledky tudíž nemá
- no a tím pádem musíme pipelinu stallnout jedním nopem

Otázka 5 [1] - DONE

V 5% pripadu beq predtim ztracime dve instrukce(IF+ID stupni) a proto CPI byl 0.9 + 0.05 + 0.05 * 7/5 = 1.02 CPI

Po pouziti preditktru a nahrazenim pouze 0.05 * (1-0.95) = 1/400 instrukce muzime resit pomoci stallu, coz je blizko k CPI = 1.

A proto zrychleni bude blizko 2%

Otázka 6 [1] - DONE

- 80% instrukcí bylo jasných
- 95% instrukcí bylo predikováno správně
- potom 15% bylo netriviálních a správně predikovaných
- 20% bylo celkem netriviálních
- no tak potom přesnost prediktoru je 3/4

Otázka 7 [1] - DONE

Odpověď 1:
???
Veľkosť bunky: 4B
Identifikovaná zero-based indexom.

Odpověď 2:
- velikost buňky operační paměti je slovo a to je dnes většinou jeden byte
- buňka paměti je identifikována paměťovou adresou
*-nebo mám na velikost buňky odpovědět něco jiného?

Velikost jedné buňky operační paměti z pohledu architektury počítače je 1 bajt (8 bitů) a je jednoznačně identifikována pomocí své adresy. V moderních počítačích se běžně používají 64bitové adresy, dříve 32bitové. Na fyzické úrovni (například v DRAM) je každý bit uložen v samostatné paměťové buňce složené z kondenzátoru a tranzistoru.

Otázka 8 [1] - DONE

- protože DRAM je levnější a my chceme u hlavní operační paměti maximalizovat hustotu cena/bit
SRAM ma vyhodou to, ze je rychlejsi a proto pouzivame pro cache

Otázka 9 [1] - DONE

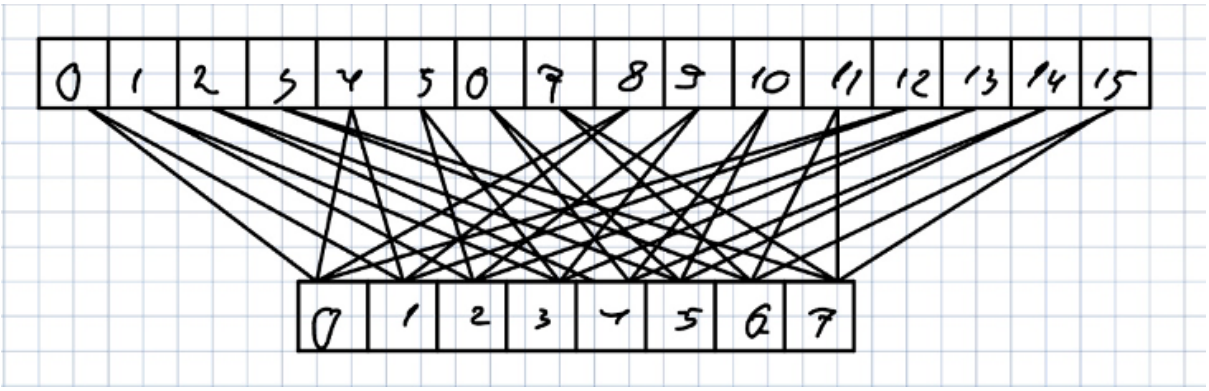
- Princip, že programy mají tendenci přistupovat k datům, která jsou v paměti blízko dat, ke kterým program už přistupoval. Tohohle využíváme tak, že při přenosu z nižší vrstvy hierarchie do vyšší přeneseme i okolí slova, které bylo požadováno.
- Potom požadavků o data z nižší vrstvy je méně.

Temporal lokalika je duvod proc se dava smysl pouzivat hierarchickou pamet umožňuje držet často používaná data v rychlé vrstvě paměti a tím zrychlit běh programů.

Otázka 10 [2] - DONE

- Natáhli jsme si z nižší vrstvy data A. Později jsme natahovali data B, ale protože tato vrstva (tato cache) byla moc malá, museli jsme obětovat data A. Poté jsme chtěli pracovat s daty A, ale ty už jsme neměli, takže došlo k missu. A takový miss (způsobený nedostatkem kapacity cache) je capacity miss.
- Narozdíl od conflict missu, tomuto bychom se nevyhli, ani kdybychom měli sebevíc asociativní cache.
- Abychom tyto missy omezili, musíme zvýšit kapacitu cache.
- Čekal bych, že tyto missy budou nastávat hodně u L1 cache, protože ta často plně asociativní, ale malá.
- U pamětí si můžeme vybrat jenom 2 za 3: Kapacita, rozumná cena a rychlost. No a protože drahé procesory nechceme a u L1 prioritizujeme rychlost, kapacita holt bude menší.

Otázka 11 [2] - DONE



Adresu rozdelime na tri casti: tag + index + offset. Index se resi kam do cache chceme dat, offset je v ramci bloku pameti, tag je zbytek adresy, pouzivame ho abychom zarucili je to spravny blok
Pokud velikost cache line je 4 bajty, tak na offset potrebujeme 2 bity, mame 4 setu v cachu a proto potrbebujeme 2 bity pro index, zbytek 12 bitu je tag. Musime ulkadat tag pro kazdou polozku a proto mame (11+1)*8=96 bitu overhead.

Otázka 12 [2] - DONE

False sharing je situace kdyz vice vlaken pristupuji ke svym vlastnim promennym, ale teto promenny fizucky nacharzi ve stejne cache line a kvuli tomu mame neustale synchornizace mezi processory

```
int[1000] counters

void worker_counter(int worker_id){
        while(true){
            counters[worker_id]++;
        }
}
```

Mame 1000 vlaken na ruznych processor a kvuli tomu, ze countery jsou fyzicky na stejne cache line, tak processor musi synchronizovat obsah mezi sebou

Otázka 13 [2] - DONE

1. P2 write, P2 goes to Modified state and issues BusRdX, so P1 and P0 goes to Invalied state
2. P0 read, P0 goes to shared state and issues BusRd, so P2 goes from Modified to Shared, P1 stays.in Invalid
3. P1 read, P1 goes from invalid to Shared and issues BusRd, both P2 and P0 remain in Shared state
4. P0 write, P0 goes from Shared to Modified and issues BusRdX, both P1 and P2 go from Shared to Invalid state
5. P2 read, P2 goes from Invalid to Shared state and issues BusRd, P0 goes from Modified to Shared, P1 remains in invalid state

30 - 2020 07 09

Řešení nekoho z Fóra : Odkaz na reseni (mirror)
Hodnoceni toho reseni: Odkaz na Hodnocení (mirror)
A zde je nezávisle vypracované další řešení:

Otázka 1 [1] - DONE

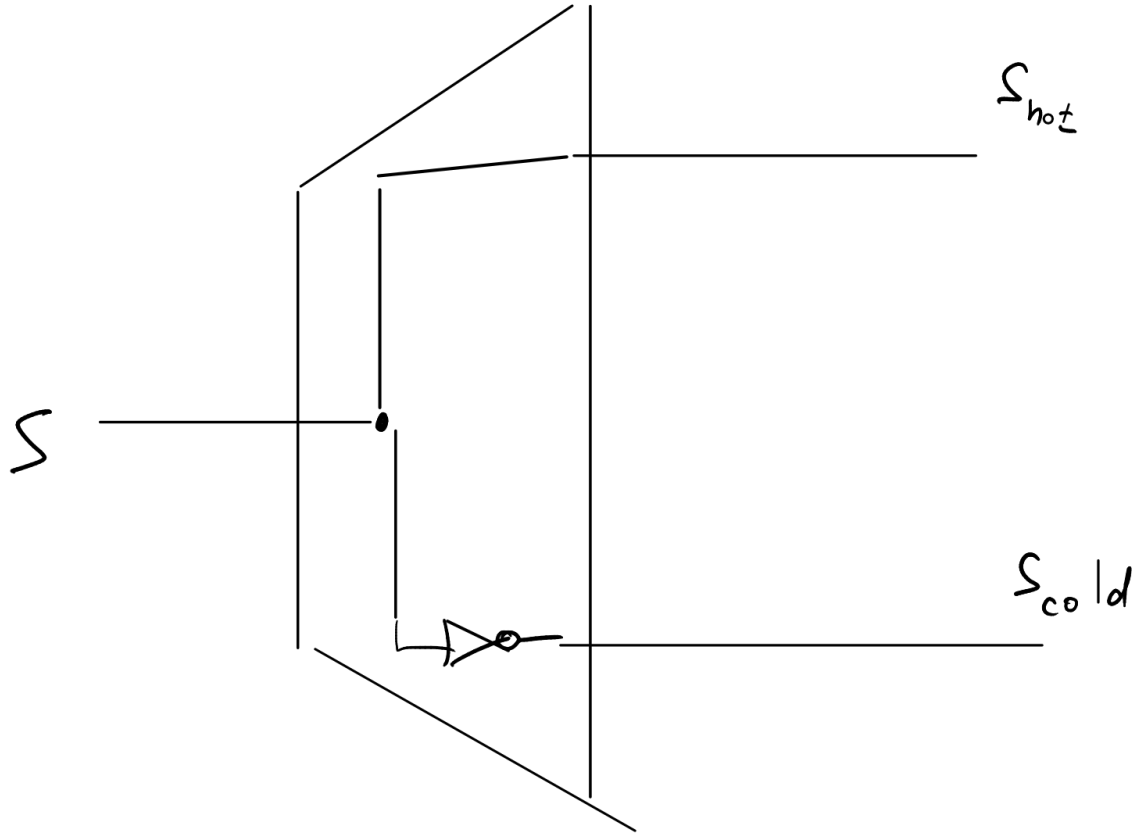
Response time ovlivněn nebude. Sekvenční task každého uživatele bude trvat stejně dlouho nezávisle na počtu jader. Throughput se ale zvýší, jelikož server bude schopný zpracovávat více tasků najednou.

Otázka 2 [1] - DONE

60% kroku simulace zrychlíme 5x, tedy celkový čas se zrychlením bude 40% + 12% = 52% toho původního. (0,52s)

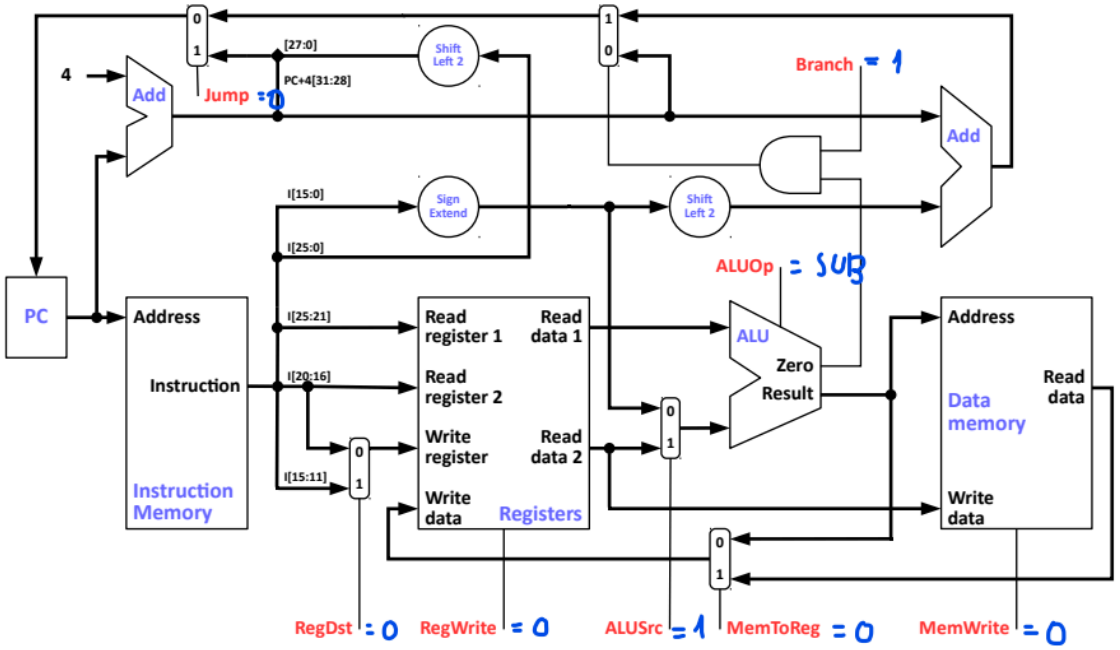
Otázka 3 [2] - DONE

s	x0	x1	xs
0	1	1	1
0	1	0	1
0	0	1	0
0	0	0	0
1	1	1	1
1	1	0	0
1	0	1	1
1	0	0	0



Myslím, že se tomu říká selektor

Otázka 4 [2] - DONE (T)



Precte data z registru - porovna je - pokud nula branchne (moc nevim co k tomu vice rict :D)

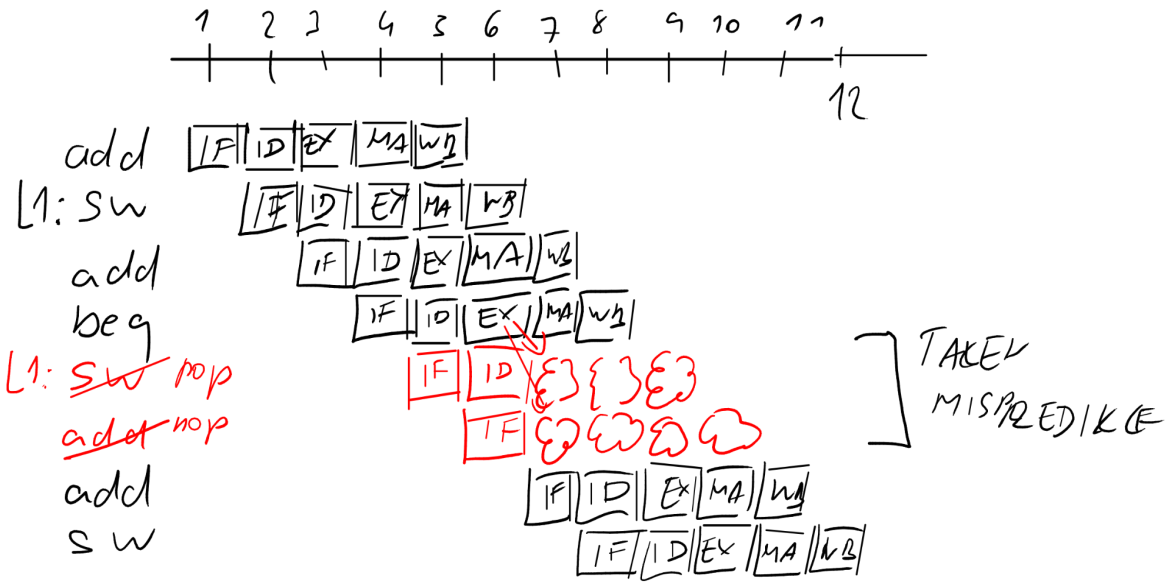
Otázka 5 [1] - DONE

Jednocyklový procesor bude muset v jednom cyklu stihnout všechny fáze datové cesty. Tedy sečteme jejich latence a máme 200 + 150 + 120 + 190 + 140 = 800ps délku cyklu. Pipelinovaný procesor v jednom cyklu musí stihnout taky všechny stage pipeliny, ale může to udělat najednou místo po sobě. Tedy vybereme největší z latencí a máme 200ps délku cyklu.

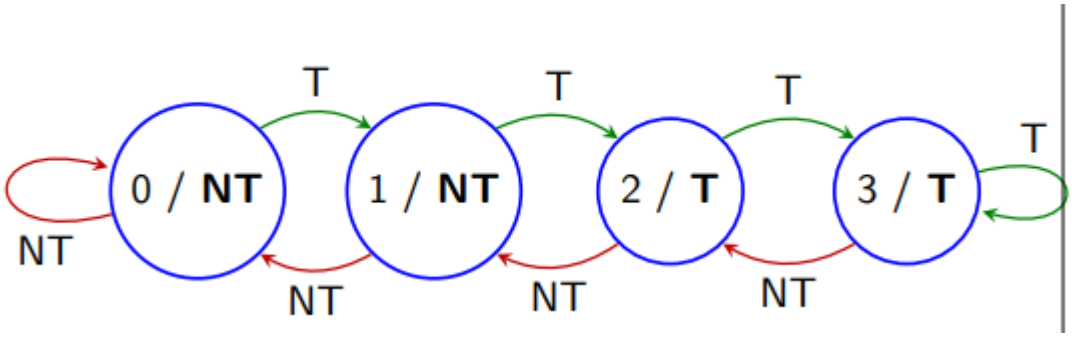
Otázka 6 [1] - DONE

ALU instrukce zapisují do registru
lw instrukce zapisují do registru
snad nemusíme počítat rozjezd a zastavování pipeline
potom ALU a lw tvoří 60% instrukcí, takže vytížení zápisového portu registrového pole bude 60%

Otázka 7 [2] - DONE



Otázka 8 [2] - DONE



Na konci této sekvence bude vždy prediktor ve stavu 1. Potom průběh jedné sekvence vypadá takto:

stav	prediktor	doopravdy
1	NT	T
2	T	T
3	T	T
3	T	NT
2	T	NT

Tedy při neustálém opakování této sekvence bude mít prediktor 40% úspěšnost.

Otázka 9 [1] - DONE

Protože náboj v kondenzátoru v DRAM má tendenci vyrovnávat se s nábojem v okolních kondenzátorech. Tedy kdybychom paměť neobnovovali, náboje by se zcela vyrovnaly, čímž by informace byla ztracena (dostali bychom vlastně (ono prakticky to asi bude složitější) paměť plnou jedniček nebo nul).

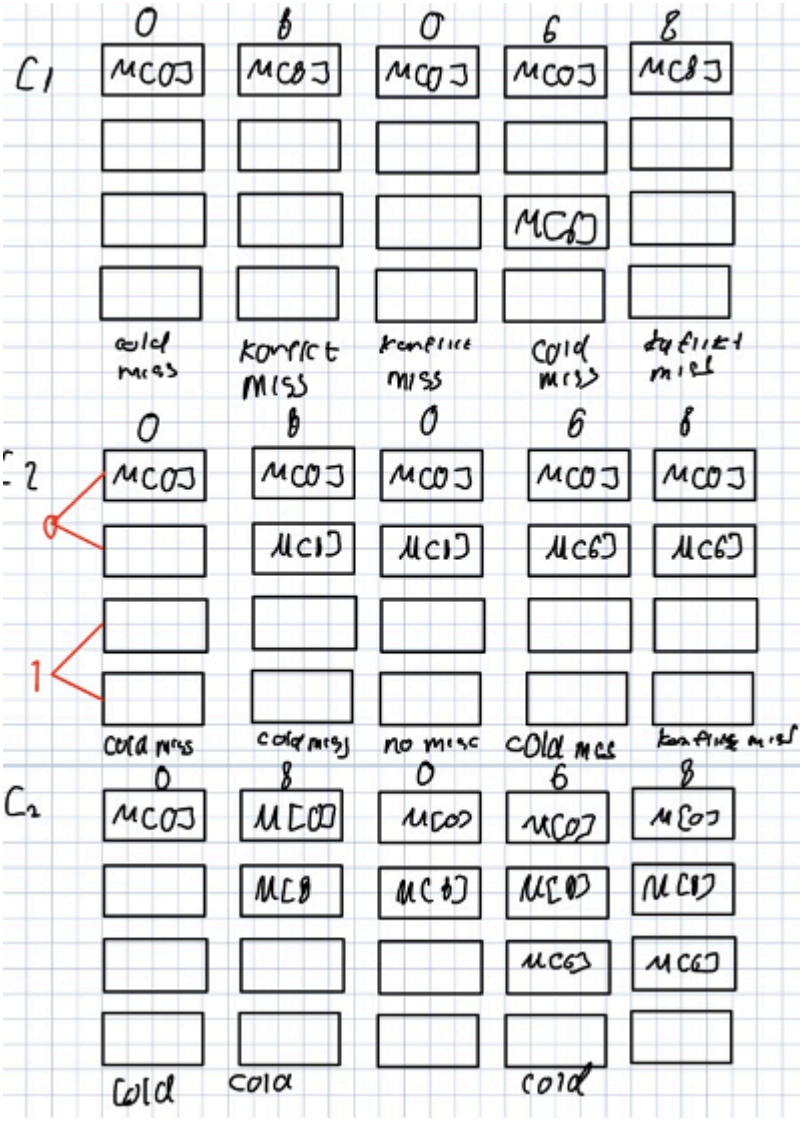
Otázka 10 [1] - DONE

Hlavně kvůli časové lokalitě cachujeme. Programy mají tendenci používat stejná místa v paměti opakovaně. Tedy my toho můžeme využít a místa, ke kterým program přistupuje často držet v rychlejší vrstvě paměťové hierarchie než data, ke kterým moc/vůbec nepřistupuje.

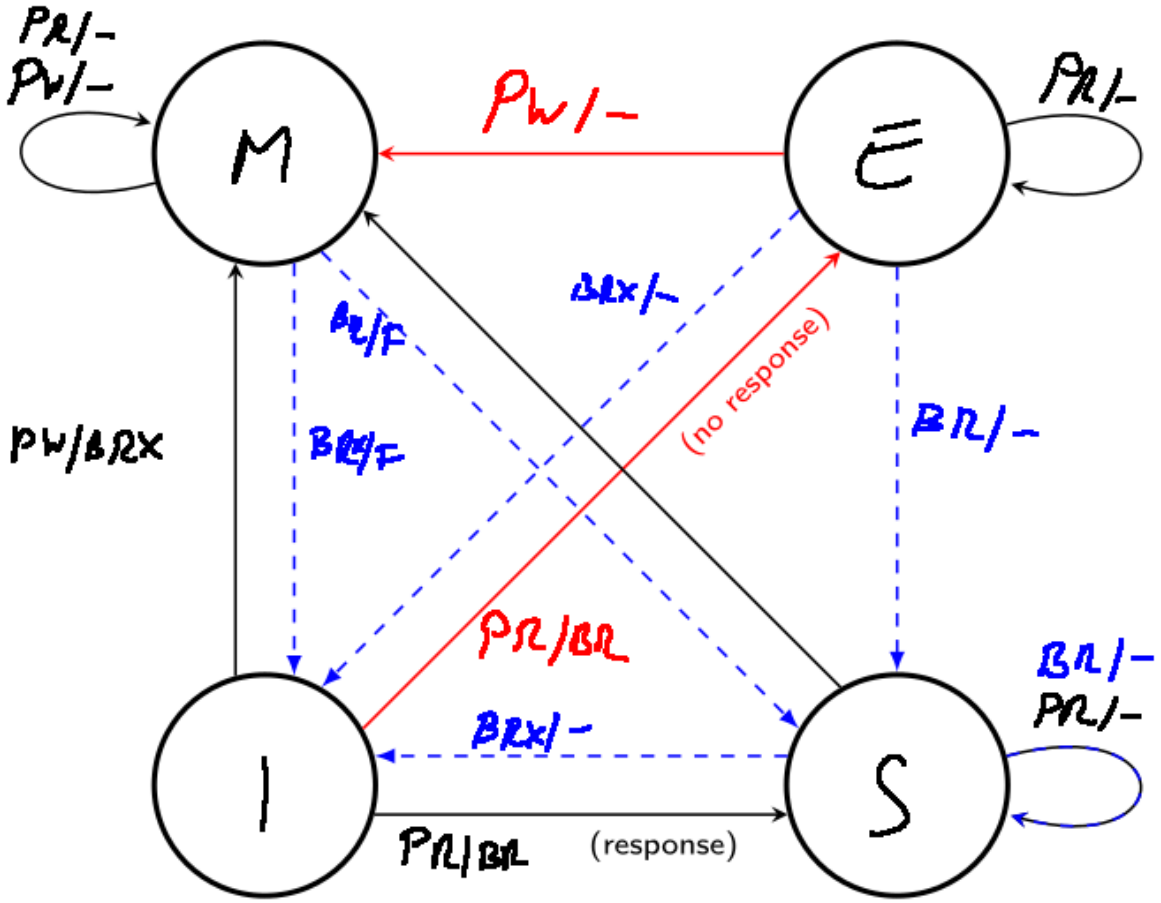
Otázka 11 [1] - DONE

Čím vyšší asociativita paměti, tím menší je šance, že nastane collision miss. Ten se děje tehdy, když bylo pro nějaký cache line index málo ways a tak jsme museli nějakou cache line vyhodit. Pokud bude více ways, toto se nebude stávat tak často.

Otázka 12 [3] - DONE



Otázka 13 [2] - DONE



100 - 2021 06 17

Stejný jako 27