LLISTA DE PROBLEMES 1

# 1.1

Penseu en la següent instrucció: and rd, rs1, rs2 Interpretació: Reg [rd] = Reg [rs1] AND Reg [rs2]

1. Quins són els valors dels senyals de control generats pel bloc “control” de la figura 1 per a aquesta instrucció? Determineu el guany en velocitat d’un cas comparada amb l’altre. Doneu el resultat en %.

Senyal de dades => porten dades per processar.

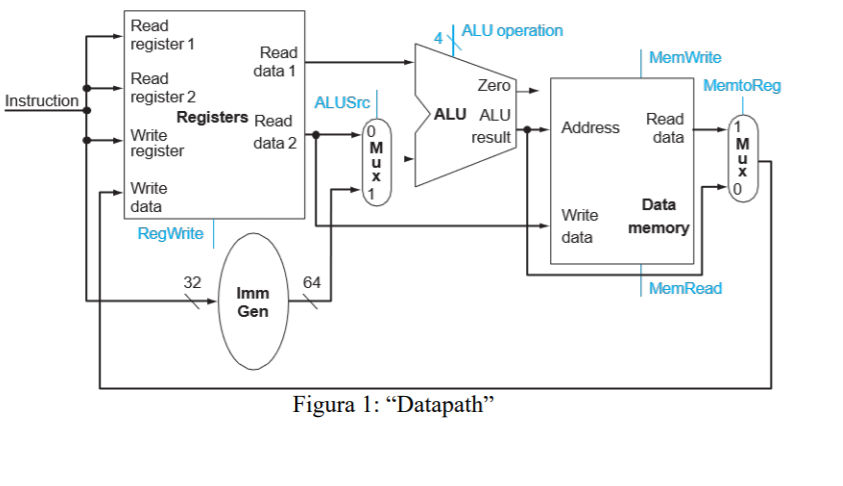
Senyal de control => ens permeten en el mateix HW fer instruccions diferents

| Reg Write | ALUscr | ALUop | Mem Write | MemRead | MemToReg |
| --- | --- | --- | --- | --- | --- |
| 1  (escriu al banc) | 0  (fa servir R i no Imm) | and | 0  (no escriu) | 0  (no llegeix) | 0  (ve de l’ALU) |

2. Quins recursos (blocs) fan una funció útil per a aquesta instrucció?

Tots menys els de escriure o llegir la mèmoria ja que al ser una instrucció de tipus R (resgistre) no els utilitzarem.

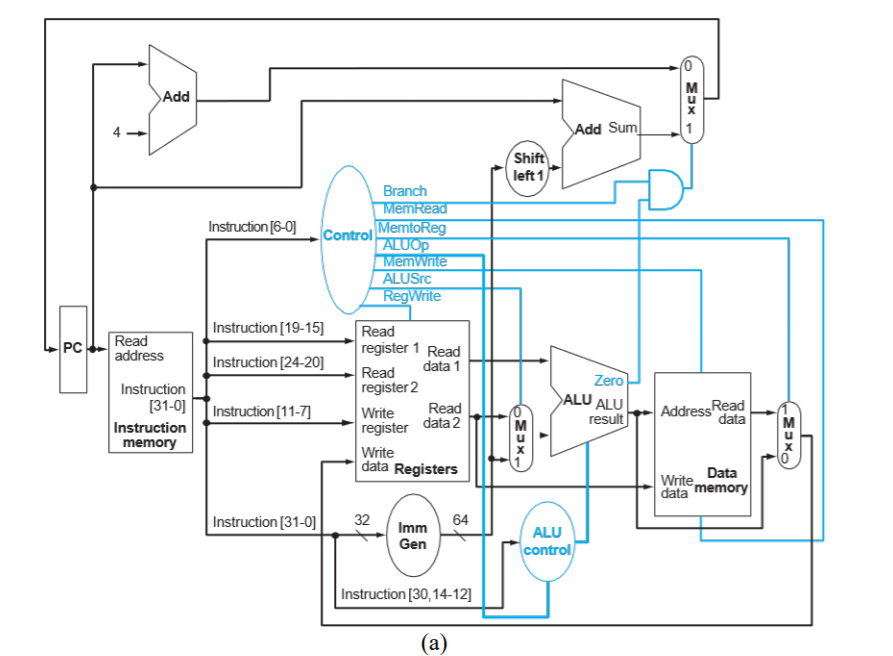
3. Quins recursos (blocs) no produeixen cap sortida per a aquesta instrucció? Quins recursos produeixen una producció que no s'utilitza?

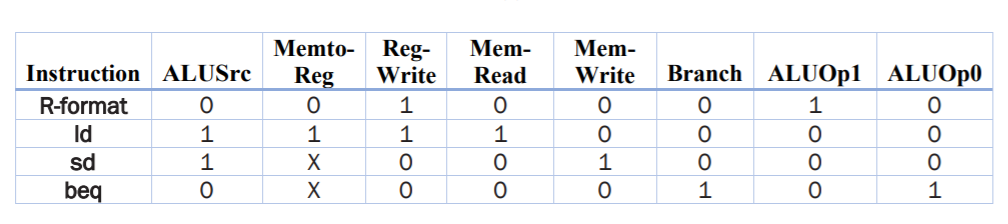


Està tot explicat a la taula de l’apartat 1.

# 1.2

Expliqueu cadascun dels “no m’importa” (“X”) de la figura 2b.



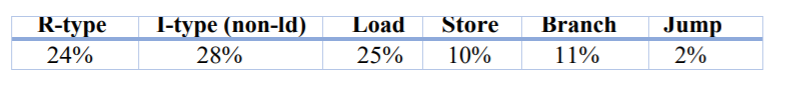


MemToReg => mira el resultat de l’ALU o de la memòria i el porta al registre per escriure.

En els salts condicionals (beq) no importa ja que és un comparador, no volem escriure a cap registre i el sd (store) escriu el valor d’un registre a la memòria. Per això no ens importa com estigui la senyal de control MemToReg i li posem una ‘X’.

# 1.3

Penseu en la combinació d'instruccions següent:



1. Quina fracció de totes les instruccions utilitza la memòria de dades?

Només la necessitaran els Load i els Store: 25% + 10% = 35%

2. Quina fracció de totes les instruccions utilitza la memòria d’instruccions?

La faran servir totes perquè és on hi ha les instruccions: 100%

3. Quina fracció de totes les instruccions utilitzen l’extensió de signe?

Tots menys els R-type: 28% + 25% +10% + 11% + 2% = 76%

4. Què fa l’extensió de signe durant cicles en què no és necessària la seva sortida?

Tot genera una sortida, sempre es farà una extensió de signe però si no es necessita s’ignorarà en el MUX.

# 1.4

Quan es fabriquen xips de silici, els defectes de materials (per exemple, el silici) i els errors de fabricació poden provocar circuits defectuosos. Un defecte molt comú és que la línia d’una senyal es "trenqui" i registri sempre un 0 lògic, sovint s'anomena un error "stuck at 0". Ajudeu-vos de la figura 1.

1. Quines instruccions no funcionen correctament si la senyal MemToReg està “stuck at 0”?

No es podran realitzar ni Store ni Load.

2. Quines instruccions no funcionen correctament si la línia ALUSrc està “stuck at 0”?

No agafarà l’inmediat (I-type, Load,Store)

# 1.5

En aquest exercici, examinarem detalladament com s’executa una instrucció en un “datapath” d’un cicle únic (single cycle). Els problemes d’aquest exercici fan referència al cicle de rellotge en què el processador obté la següent instrucció: 0x00c6ba23 (ISA RV64I).

1. Quins són els valors de les entrades de la unitat de control ALU per a aquesta instrucció?

0x00c6ba23 = 0000 0000 1100 0110 1011 1010 0010 0011 ( no pot ser R)

| 0000000 | 01100 | 01101 | 011 | 10100 | 0100011 |
| --- | --- | --- | --- | --- | --- |

SD x12, 20(x13)

La ALU op = 00

ALU control lines = 0010

2. Quina és la nova adreça del PC després d'executar aquesta instrucció? Ressalteu el camí a través del qual es determina aquest valor (figura 2).

La nova adreça serà PC + 4 ja que és adreçable byte a byte.

3. Per a cada mux, mostreu els valors de les seves entrades i sortides durant l'execució d'aquesta instrucció. Enumereu els valors que són sortides de registre com Reg [xn] (on n és el nombre de reg).

ALUsrc : Inputs: Reg[x12] i 0x0000000000000014 ; Output: 0x0000000000000014 MemToReg : Inputs: Reg[x13] + 0x14 i ; output:

Branch: Inputs: PC+4 i PC + 0 x0000000000000028

4. Quins són els valors d'entrada per a l'ALU i les dues unitats addicionals?

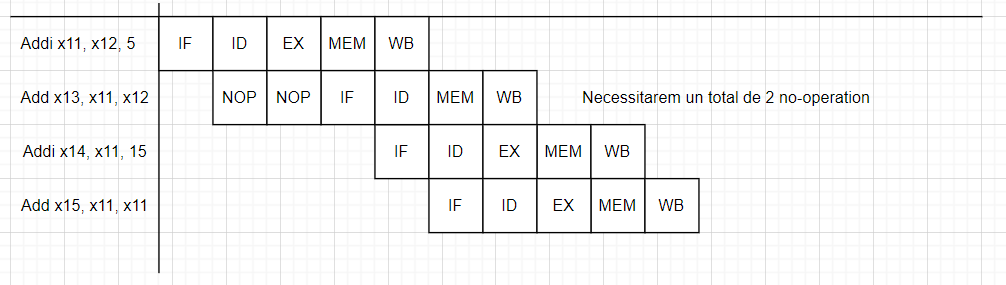
ALU inputs: Reg[x13] i 0x0000000000000014

PC + 4 adder inputs: PC i 4

Branch adder inputs: PC i 0x0000000000000028

# 1.6

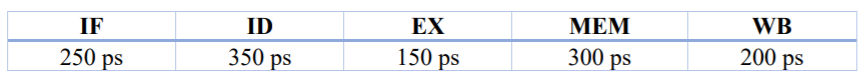
Afegiu tantes instruccions “NOOP” com creieu al codi següent per tal que pugui funcionar en un pipeline de 5 etapes del processador RISC-V sense “Forward Unit”. Recordeu que tenim 32 registres que van des de l’x0 fins a l’x31.



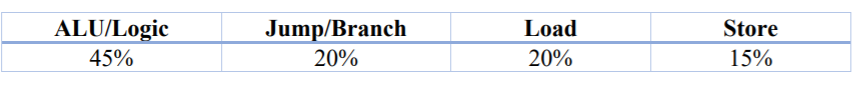
falta EX a la segona instrucció !!!

# 1.7

En un processador RISC-V de 5 etapes examinem com afecta el pipeline al temps de cicle del rellotge del processador. Les qüestions d’aquest exercici suposen que cada etapa triga un temps diferent en executar-se, en particular els temps d’execució de cada etapa són els següents:



Suposeu també que les instruccions executades pel processador es desglossen de la manera següent:



1. Quin és el temps de cicle de rellotge mínim (en ps) en un processador amb pipeline i en un sense pipeline ?

Sense pipleline cada execució s’executa de manera independent ocupant tot el hardward. La seva durada mínima és la suma de les seves durades = 1250 ps.

En el cas de pipeline el cicle ha de ser tan gran com l’etapa que ocupi més ja que es poden executar més d’una etapa en un cicle. Això vol dir que cada cicle serà com a mínim de 350 ps.

2. Quant temps triga en executar-se una instrucció de tipus Load en un processador amb pipeline i en un sense pipeline ?

Sense pipeline trigarà 1250 ps. Amb pipeline tindrem 350 ps x 5 = 1750 ps

3. Si podem dividir una etapa del pipeline en dues noves etapes, cadascuna trigarà la meitat de temps en executar-se que l’etapa original, quina etapa dividiríeu i quin és el nou temps de cicle del rellotge del processador amb pipeline?

Dividiriem la ID ja que és la que triga més (cada una trigaria 175 ps) ara el nou temps de cicle seria 300 ps (la de MEM) que és la de l’etapa que dura més.

4. Suposant que no hi ha “stalls” ni “hazards”, quina és percentatge d’utilització de la memòria de dades per a les instruccions executades pel processador amb pipeline?

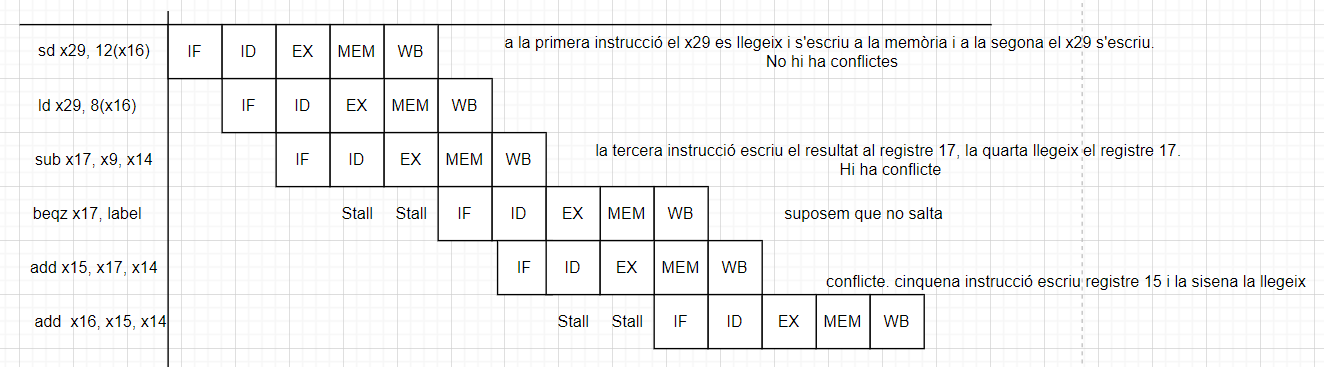
Memòria de dades s’utilitza per Load i Store. Això vol dir que el percentatge serà del 35% (20% Load, 15% Store).

5. Suposant que no hi ha “stalls” ni “hazards”, quina és percentatge d’utilització del port d’escriptura dels Registres per a les instruccions executades pel processador amb pipeline?

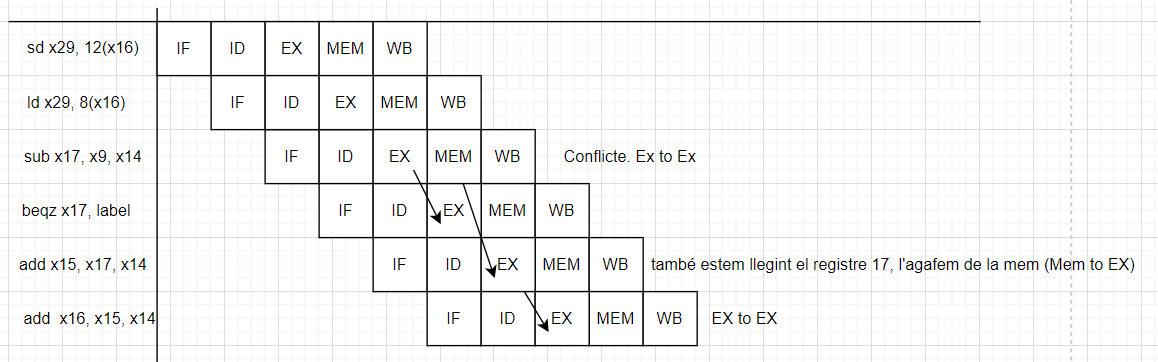
ALU (45%) i les Load (20%) Necessitarem el 65% del temps.

# 1.8

1.Calculeu el temps de computació si en el processador NO tenim Unitat de Forwarding Paths. Ompliu la taula emprant “stalls” per solucionar les possibles dependències de dades. (aquest codi pot requerir més o menys cicles dels tabulats)



2. Calculeu el temps de computació si en el processador SÍ tenim Unitat de Forwarding Paths. Ompliu la taula emprant “stalls” per solucionar les possibles dependències de dades. (aquest codi pot requerir més o menys cicles dels tabulats)



# 1.9

Assumim que el 20% de les instruccions executades per un programa són de tipus “branch”. No hi ha “stalls” deguts a dependència de dades. Tenim predicció estàtica de salt del tipus “no es compleix”. Els temps “perdut” (penalty) és d’1 cicle de rellotge:

1. Determineu el temps d’execució dels dos casos següents: en el 30% dels “branch” la condició es compleix. En el 70 % dels “branch” la condició es compleix.

Número total instruccions no el sabem = Ins.

20% d’instrccions són branch => 80% no ho són.

primer cas (funciona 30%)

noBranch = Ins x 0.8 Branch = Ins x 0.2

branchCompleix = Branch x 0.3 branchNoCompleix = Branch x 0.7

t = Ins x(0.8 x 1cyc + 0.2 x 0.3 x 2cyc +0.2 x 0.7 x 1cyc) = ins x 1.06

segon cas (funciona el 70%)

noBranch = Ins x 0.8 Branch = Ins x 0.2

branchCompleix = Branch x 0.7 branchNoCompleix = Branch x 0.3

t = Ins x(0.8 x 1cyc + 0.2 x 0.7 x 2cyc +0.2 x 0.3 x 1cyc) = ins x 1.14

2. Determineu el guany en velocitat d’un cas comparada amb l’altre. Doneu el resultat en %.

t1 = Ins x 1.06

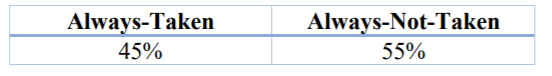
t2 = Ins x 1.14

Guany = (t2 / t1) - 1 = 0.075= 7.5%

# 1.10

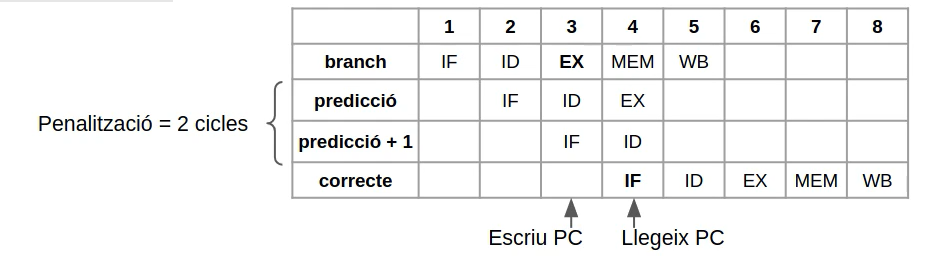
La importància de tenir un bon predictor de “braches” depèn de la freqüència amb què s’executen els “branches” condicionals. Si també considerem la precisió del pronòstic dels “branches”, tots dos conceptes determinaran la quantitat de temps que el processador es passa aturat a causa de “branches” imprevistos. En aquest exercici, suposem que el desglossament d’instruccions en diverses categories d’instruccions és el següent:

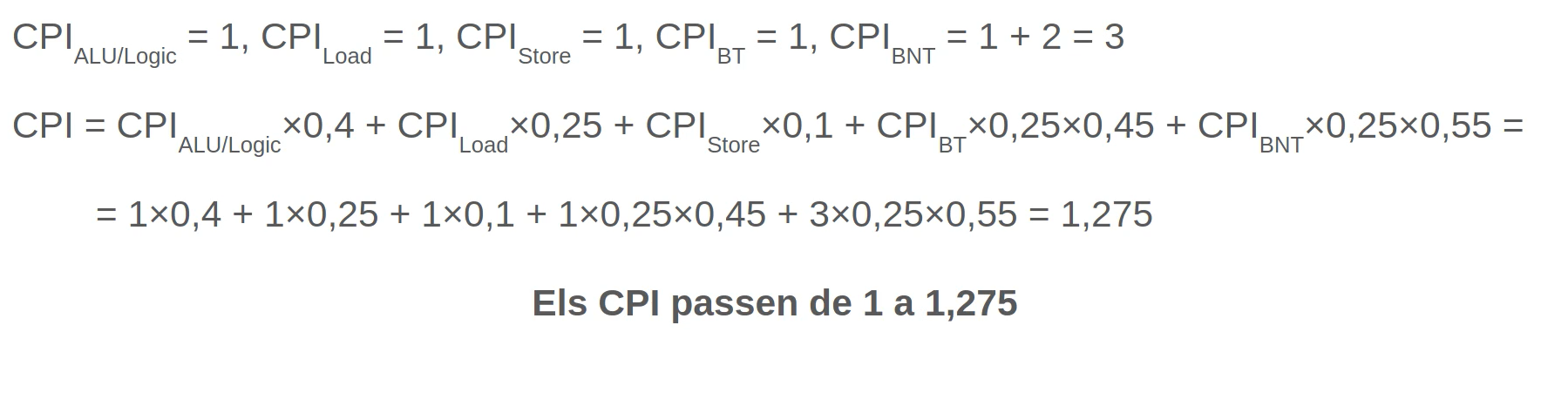




Suposeu que fem servir un processador RISC-V de 5 etapes, on els “branches” es resolen al final de l’etapa d’execució. Suposeu, també, que el programa és infinit i partim d’un CPI ideal (no tenim hazards deguts a dades).

1. Quin és l’increment en el CPI degut a la mala predicció del predictor “Always-Taken”?





Dos cicles de penalització més el cicle que triguem en executar la instrucció (primera predicció escriu i a la segona llegeix). CPI d’haver-nos equivocat és de 3.

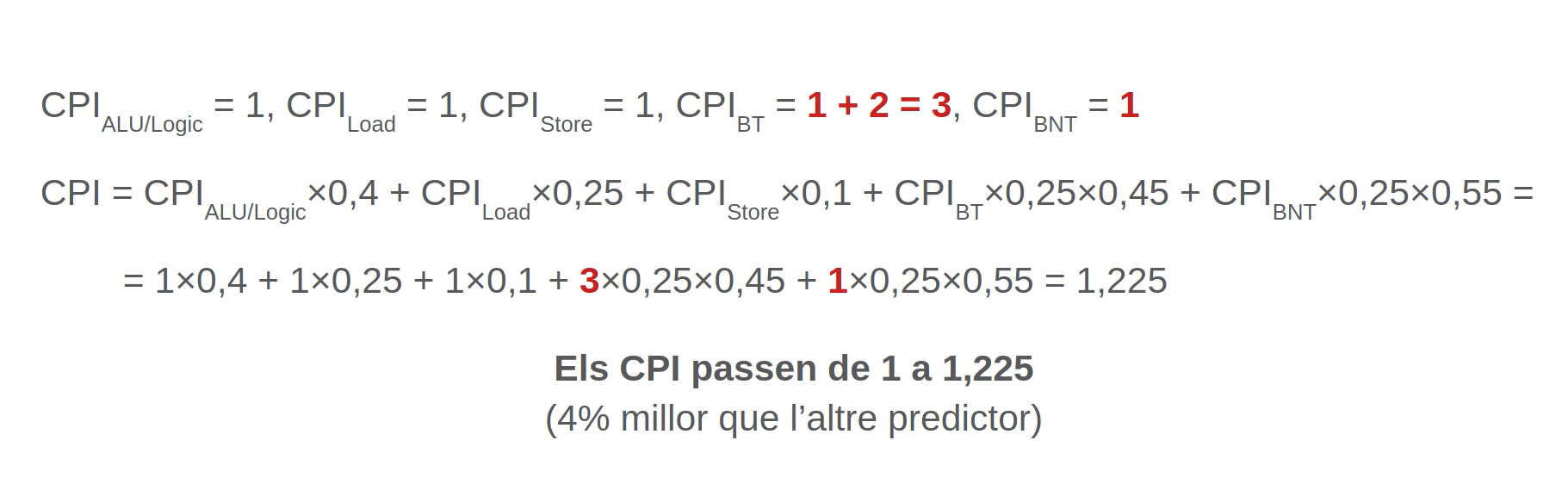
BT = branch taken

BTN = branch not taken (CPI de 3)

la resta d’instruccions tindran un CPI de 1.

S’han d’aplicar els percentatges que ens donen a l’enunciat.

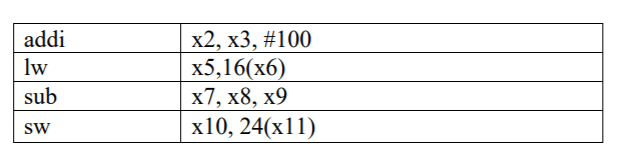
2. Quin és l’increment en el CPI degut a la mala predicció del predictor “Always-Not-Taken”?



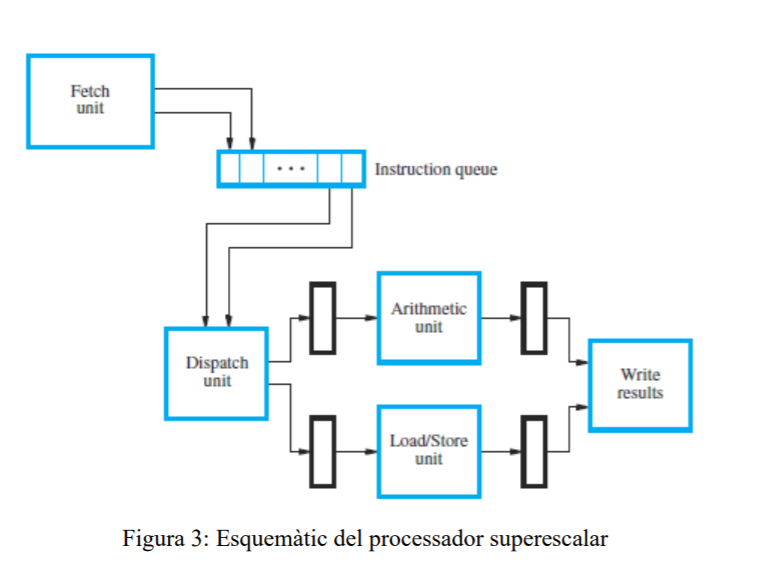
3. L’ús d’una solució dinàmica, com una taula de d’història de salts, milloraria les prestacions ? Compara-ho amb els dos predictors estàtics emprats en aquesta qüestió.

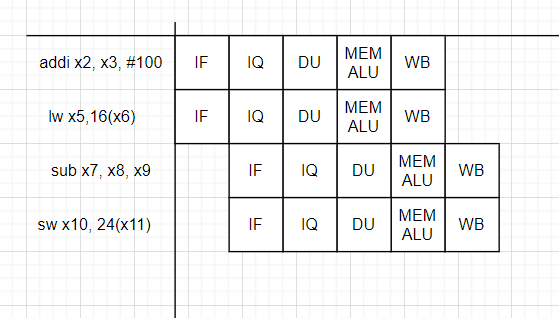
# 1.11

Un processador superescalar com el de la Figura 3 ha d’executar les següents instruccions:



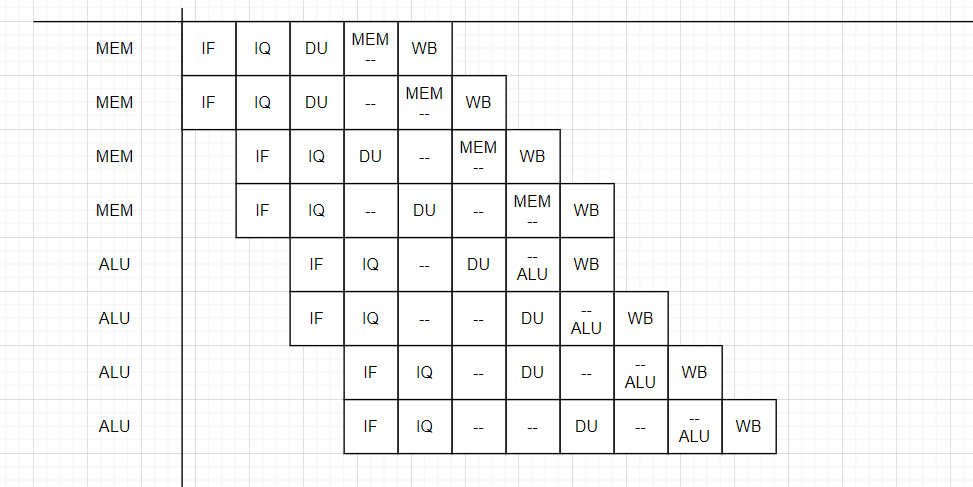
Suposant que tots els registren ja contenen els valors, dibuixeu un diagrama mostrant el flux d’instruccions dins del pipeline.

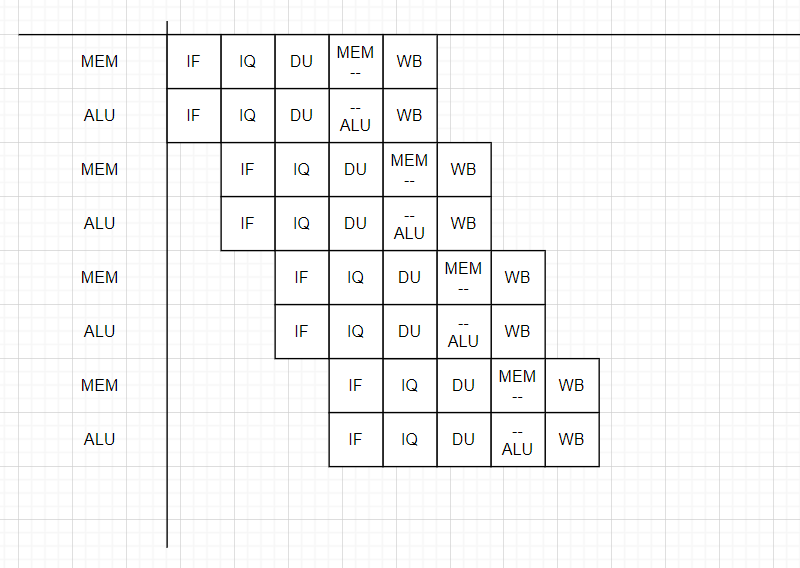




# 1.12

Considereu un programa que consisteix en 4 accessos a memòria i 4 instruccions aritmètiques. Assumiu que no hi ha cap risc de dependència de dades entre les instruccions. Dues versions d’aquest programa s’executaran en un processador superescalar com el de la Figura 3. La primera versió del programa executa els 4 accessos a memòria consecutivament i després realitza les operacions aritmètiques. La segona versió del programa té les instruccions de memòria intercalades amb les aritmètiques. Realitza dos diagrames, multicicle per a comparar les dues versions del programa.





# 1.13

Assumiu que un programa no conté instruccions tipus “branch”. El programa s’executa en un processador superescalar com el de la Figura 3. Calculeu quin és el temps d’execució si tenim una mescla d’instruccions que consisteix en un 75% instruccions aritmètiques i un 25% instruccions d’accés a memòria ? Quin és el guany en velocitat si comparem aquest processador amb un com el de la Figura 1 (ambdós utilitzen el mateix rellotge)?

IPC quan omplim les dues unitats funcionals = 2 i IPC = 1 quan només n’omplim una. Assumim que estan intercalades : IPCmem x 0.25 + IPCalu x 0.25 = IPCmem+alu = 0.5 la resta d’instruccions són el 0.5.

IPCmem = 1, IPcmem+alu = 2

CPImem = 0.5 CPImem+alu = 1

CPIsuperescalar = CPImem x0.5 + CPImem+alu x 0.5 = 0.5x0.5 + 0.5x 1 = 0.75

GUANY = 33%

# 1.14

LLISTA PROBLEMES 2

# 2.1

Paraules de 8 bits.

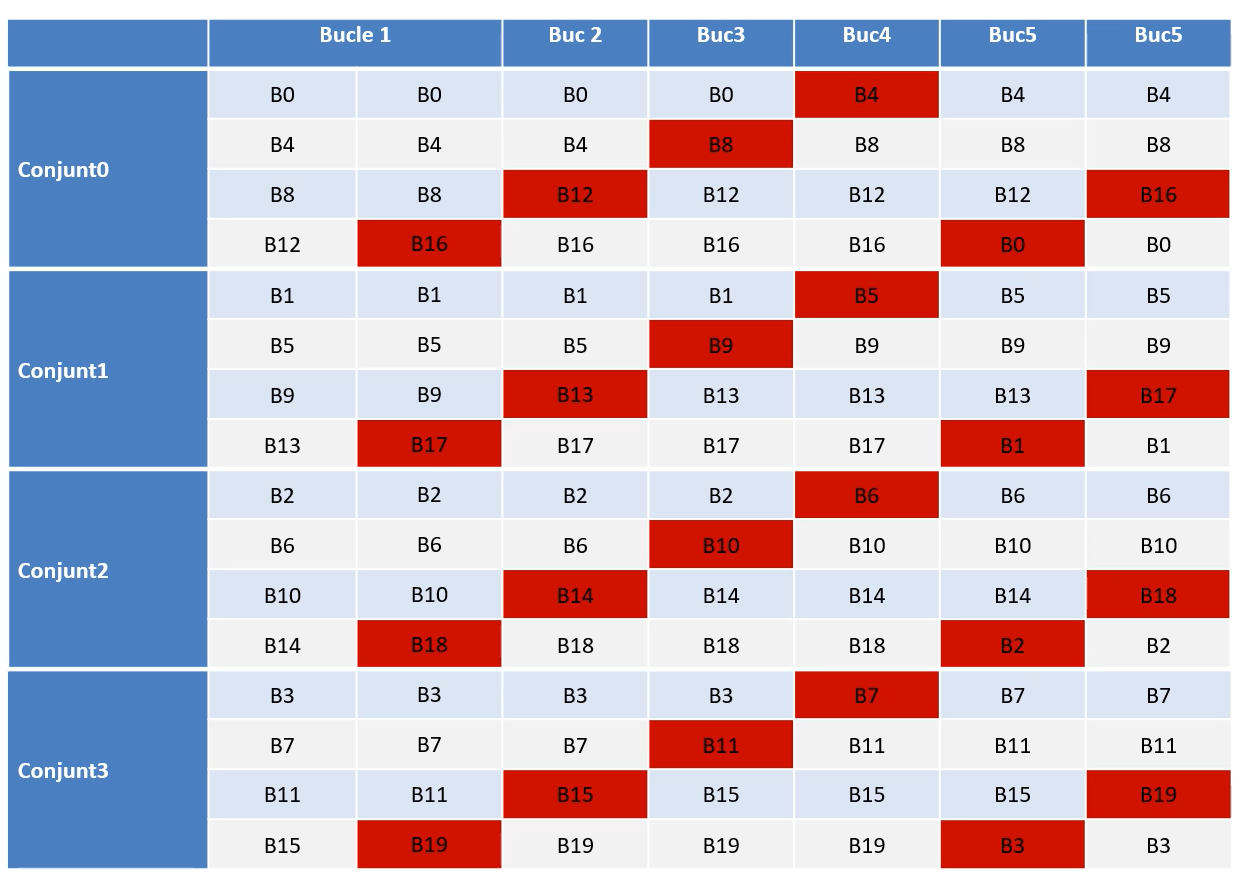
• Memòria Principal = 1MiByte.

• Caché: 1kibiparaules,

Associativa de 4 camins. 64 paraules per bloc. Un programa executa un bucle en el qual s’agafen les primeres 1280 paraules de memòria (de 0 a 1279) 5 vegades. El temps d’accés a Memòria Principal (tmp) és 17 cops més gran que el d’accés a caché (tc). tmp=17·tc Calculeu:

1. El número de bits en cada un dels camps: TAG, CONJUNT (SET), PARAULA (WORD) i BYTE (B).

2. El benefici que obtenim amb la caché, suposant que utilitzem l’algoritme MRU de reemplaçament de blocs. La caché està inicialment buida.



1280 / 64 = 20 blocs

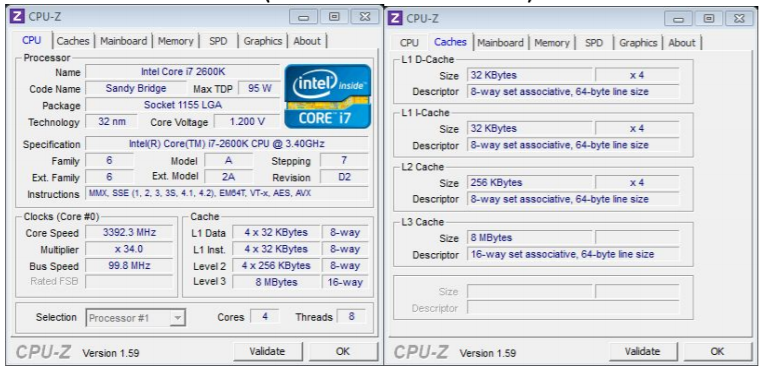
3. Creieu que algun altre algoritme seria més efectiu? Quin?

LRU en aquest tipus d’exercicis és ineficient així que no .

Presenteu els resultats a la següent taula (els càlculs a baix):

| Nombre de Línies (blocs) de la Caché (C) | C = K\*S, C =1KiB/64 = 16 linees |
| --- | --- |
| Nombre de Camins de la Caché (K) | 4 (4 camins) |
| Nombre de Conjunts de la Caché (S) | 4 (16 = 4\*S) => i = 2 |
| Bits necessaris pel TAG | 20 -2-6 = 12 |
| Bits necessaris per definir el Conjunt (S) | i = 2 |
| Bits necessaris per definir la Paraula (W) | 64 paraules x bloc (1byte)  W = log(2) 64 = 6 |
| Bits necessaris per definir el Byte (B) | 8 bits = 1byte B = log(2)1 = 0 |
| Bits totals de l’adreça | 1MiByte = 2^20 bits. Com a mínim el nostre bus te 20 bits. |
| Temps sense Caché | Tmp = 1280 paraules \* 5 cops \*17Tc = 108800Tc |
| Temps amb Caché | omplir cache = 1280 \*1cop\*17Tc  dades MP(2n bucle9: 20blocs\*64paraules/bloc = 1280 paraules  dades cache 2n bucle: 60blocs(hit)\*64 par.  = 3840 paraules(hit)  Tc = 1280\*1 +1280\*17Tc + 3840\*Tc = 47360Tc |
| Benefici | Tmp/Tc = 2.297 |

# 2.2



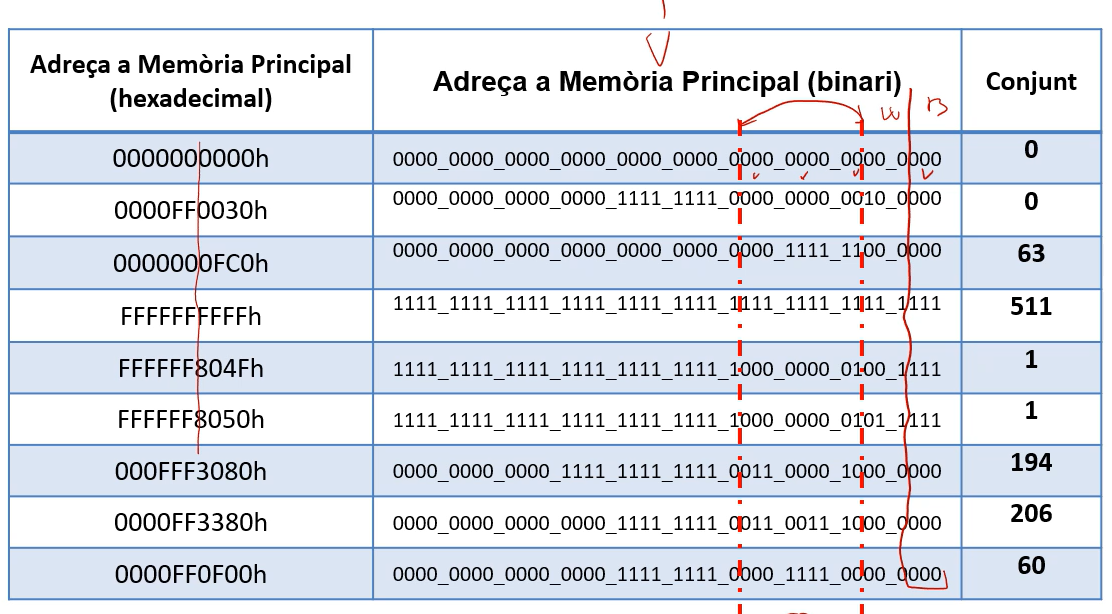
Respecte al sistema de Caché, ompliu la següent taula:

| Longitud Paraula del Processador (bits/Bytes) | 64 bits |
| --- | --- |
| Bits totals de l’adreça | 40 bits |
| CARACTERISTICA CACHE | L1 DADES |
| Nombre de Línies (blocs) C | C = 32Kib /64 bitslinea = 512 |
| Nombre de Camins K | 8 |
| Nombre de Conjunts S | S = 512 / 8 = 64 |
| Bits necessaris pel TAG | 40 -3-3-6 = 28 |
| Bits necessaris per definir el Conjunt (i) | i = log(2)64 = 6 |
| Bits necessaris per definir la Paraula (W) | 64/8 = 8 W = log(2)8 = 3 |
| Bits necessaris per definir el Byte (B) | 64 = 8bits B = log(2)8 = 3 |
| CARACTERISTICA CACHE | L2 DADES |
| Nombre de Línies (blocs) C | C = 256Kib /64 bitslinea = 4Kilinies |
| Nombre de Camins K | 8 |
| Nombre de Conjunts S | 512 conjunts |
| Bits necessaris pel TAG | 40 -9-3-3 = 25 |
| Bits necessaris per definir el Conjunt (i) | i = 9 |
| Bits necessaris per definir la Paraula (W) | 64/8 = 8 W = log(2)8 = 3 |
| Bits necessaris per definir el Byte (B) | 64 = 8bits B = log(2)8 = 3 |
| CARACTERISTICA CACHE | L3 DADES |
| Nombre de Línies (blocs) C | 8MiB /64 = 128KiB |
| Nombre de Camins K | 16 |
| Nombre de Conjunts S | 8k |
| Bits necessaris pel TAG | 40 -13-3-3 = 21 |
| Bits necessaris per definir el Conjunt (i) | 13 |
| Bits necessaris per definir la Paraula (W) | 64/8 = 8 W = log(2)8 = 3 |
| Bits necessaris per definir el Byte (B) | 64 = 8bits B = log(2)8 = 3 |

A la caché de nivell 2 del problema anterior, indica a quin conjunt pertany cada una de les dades amb les següents adreces de memòria principal:

| Adreça a Memòria Principal (hexadecimal) | Conjunt |
| --- | --- |
| 0000000000h | 0 |
| 0000FF0030h | 0 |
| 0000000FC0h | 63 |
| FFFFFFFFFFh | 511 |
| FFFFFF804Fh | 1 |
| FFFFFF8050h | 1 |
| 000FFF3080h | 194 |
| 0000FF3380h | 206 |
| 0000FF0F00h | 60 |

passar a binari i mirar els del conjunt (i)



# 2.3

El nostre sistema (processador de 32 bits+ memòria cau + memòria principal) utilitza adreces de memòria de 32 bits i té una memòria principal de mida: 1 GiB. Té una memòria cau de 4 KiB organitzada en la manera associativa a k camins, amb 4 blocs per conjunt i 64 bytes per bloc.

A. Calcula el nombre de bits en cada un dels camps d'etiqueta, “set” (conjunt) i paraula de l'adreça de la memòria.

longitud paraula = 4bytes B = 2

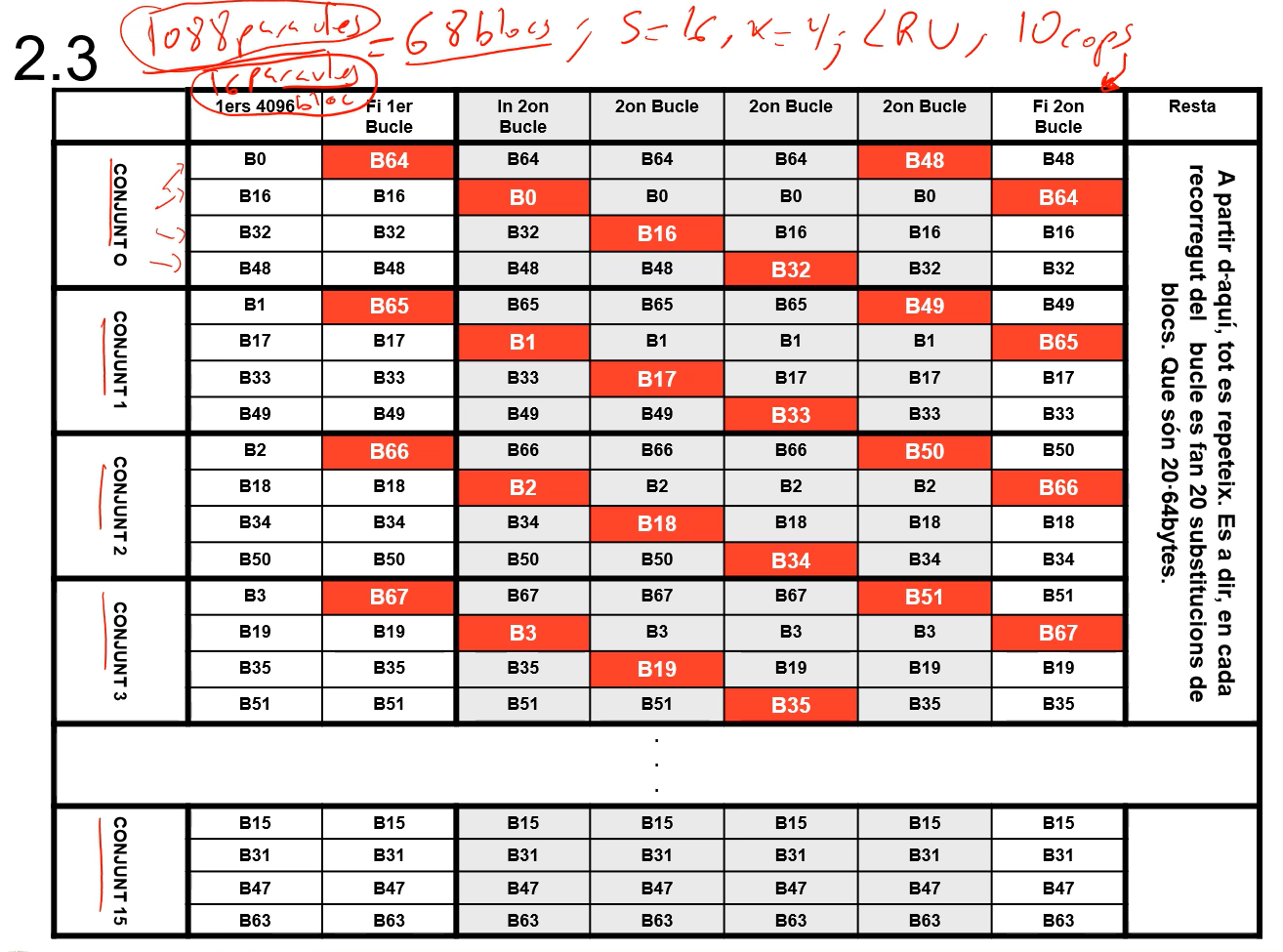
W = 64 bytesbloc/4bytesparaula = 16 paraules/bloc W = log(2)16 = 4

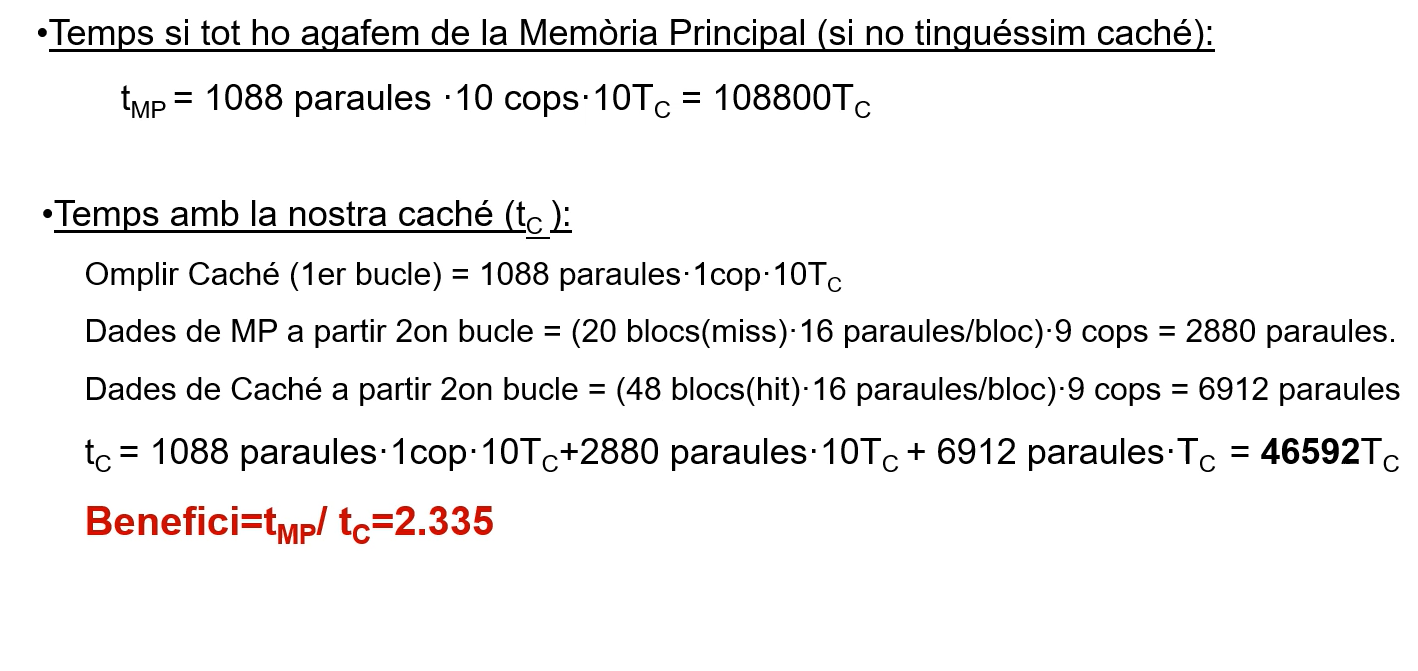
C = 4Kib/64bbloc = 64 blocs

S = 64/4 = 16 i = log(s) S = 4

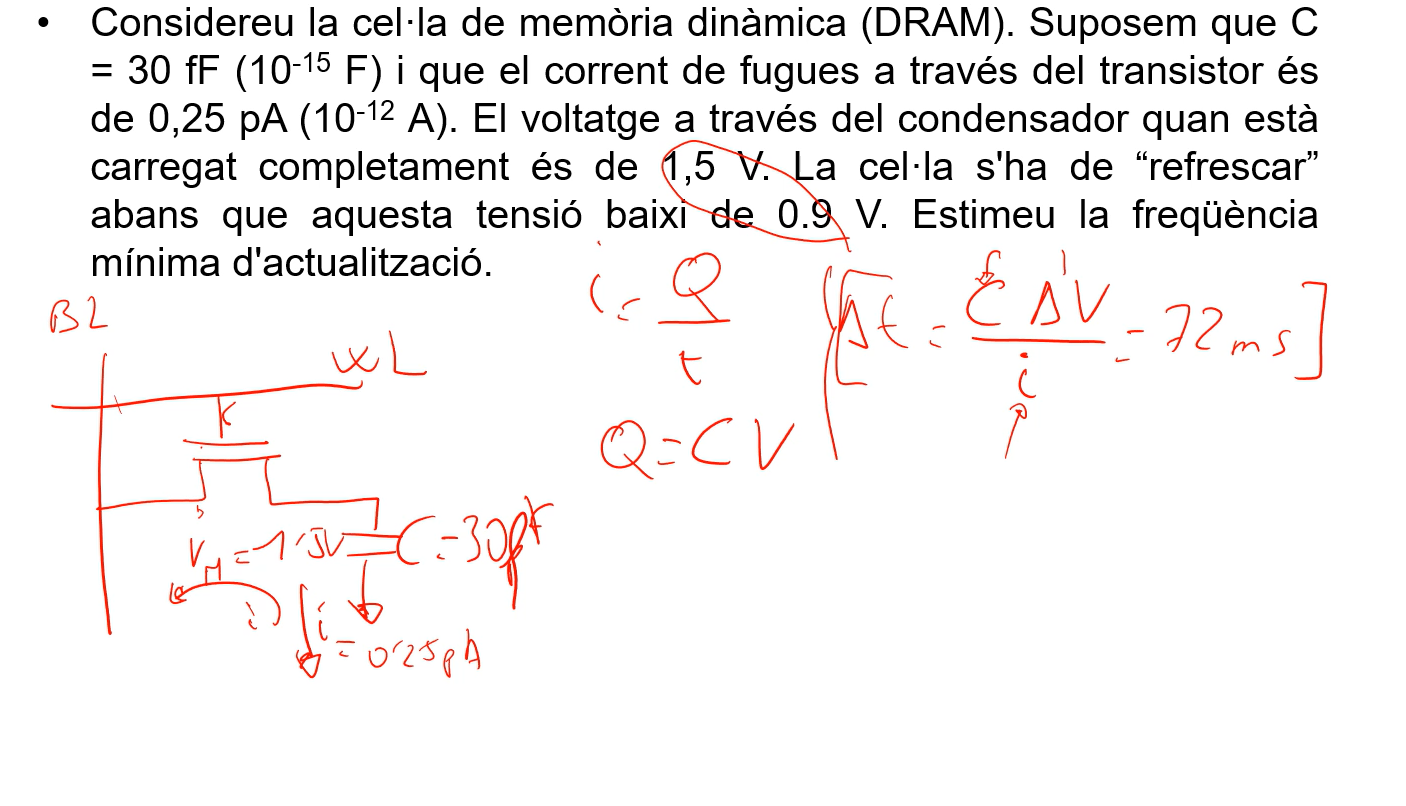
TAG = 32 -4-4-2 = 22 bits

B. Suposeu que la memòria cau està inicialment buida. Suposem que el processador obté 1088 paraules cadascuna de les ubicacions de la paraula successiva a partir de la ubicació 0. Després repeteix aquesta seqüència nou vegades més. Si la memòria cau és 10 vegades més ràpida que la memòria principal , calculeu el factor de millora derivat de l'ús de la memòria cau. Assumeixi que l'algoritme LRU s'utilitza per a la substitució de blocs





# 2.8



EXAMEN PARCIAL 1

# 1.

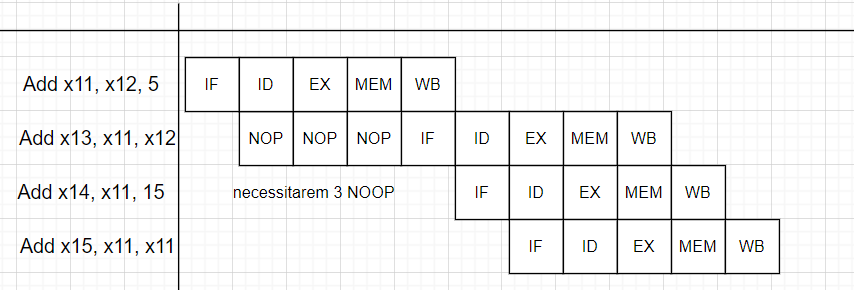
Afegiu tantes instruccions “NOOP” com creieu al codi següent per tal que pugui funcionar en un pipeline de 5 etapes com l’emprat al processador de problemes. En aquest cas tenim 32 registres que van des de l’x0 fins a l’x31. (0.5 punts)

Add x11, x12, 5

Add x13, x11, x12

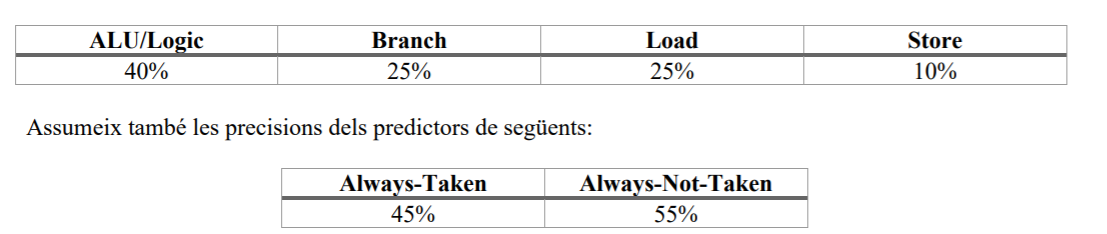
Add x14, x11, 15

Add x15, x11, x11



# 2.

La importància de tenir un bon predictor de “braches” depèn de la freqüència amb què s’executen els “branches” condicionals. Si també considerem la precisió del pronòstic dels “branches”, tots dos conceptes determinaran la quantitat de temps que el processador es passa aturat a causa de “branches” imprevistos. En aquest exercici, suposem que el desglossament d’instruccions en diverses categories d’instruccions és el següent: (1.5 punts)



A)Quin és l’increment en el CPI degut a la mala predicció del predictor “Always-Taken”?

CPI quan encerta = 1, CPI falla = 2

CPiincrement = 0.25\*(1-0.45)\*2 = 0.275

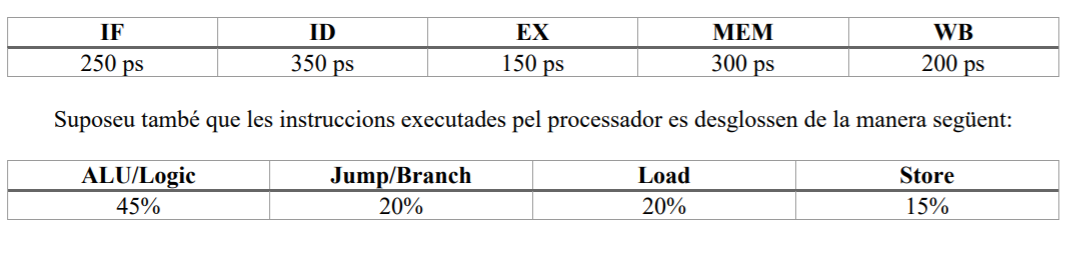
CPIpenalities = 1 + 0.275 = 1.275

B) Quin és l’increment en el CPI degut a la mala predicció del predictor “Always-Not-Taken”?

CPIincrement = 1 +( 0,25 \*(1-0.55)\*2 = 1.225

# 4.

En un processador amb un pipeline de 5 etapes, com el de problemes, examinem com afecta el pipeline al temps de cicle del rellotge del processador. Les qüestions d’aquest exercici suposen que cada etapa triga un temps diferent en executar-se, en particular els temps d’execució de cada etapa són els següents: (2.5 punts)



A)Quin és el temps de cicle de rellotge mínim (en ps) en un processador amb pipeline i en un sense pipeline ?

sense pipeline 1250 ns, amb pipeline 350ps

B)Quant temps triga en executar-se una instrucció de tipus Load en un processador amb pipeline i en un sense pipeline ?

amb pipeline triga 350\*5 = 1750 ps

sense pipelie triga 1250ps

C)Si podem dividir una etapa del pipeline en dues noves etapes, cadascuna trigarà la meitat de temps en executar-se que l’etapa original, quina etapa dividiríeu i quin és el nou temps de cicle del rellotge del processador?

la Id ara trigaria 175ps i el nou temps de cicle seria 300 ps

D) Suposant que no hi ha “stalls” ni “hazards”, quina és percentatge d’utilització de la memòria de dades per a les instruccions executades pel processador?

load i stores =>35 %

E)Suposant que no hi ha “stalls” ni “hazards”, quina és percentatge d’utilització del port d’escriptura dels Registres per a les instruccions executades pel processador?

ALU i load => 65%

# 5.

