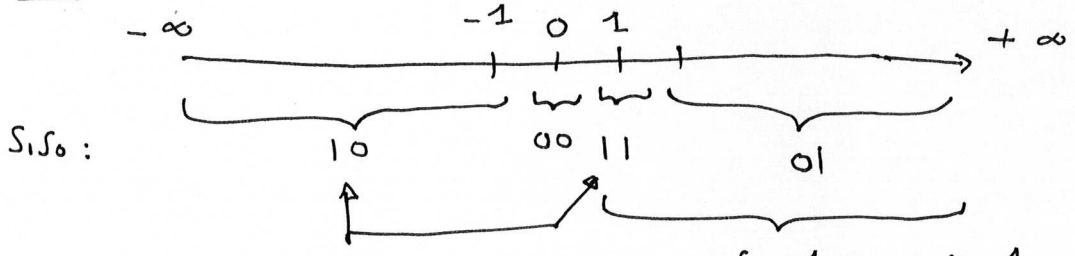
Q1

1.1) Manière de représenter les entiers relatifs sous forme binaire, de telle sorte que:
- on n'ait qu'un seul zéro
- on utilise le circuit d'addition classique pour les calculs

L'opposé d'un nombre est calculé en:
- complémentant tous les bits du nombre
- ajoutant 1 au nombre résultant

Le bit de poids fort indique le signe.

1.2)



$S_1 = 1$ si :
- $x_3 = 1$ (signe négatif)

ou
- $x = 1$

$S_0 = 1$ si $x \geq 1$, c'est-à-dire :
- $x_3 = 0$ (signe positif ou nul)

et
- $x \neq 0$ (non égal à zéro)
→ au moins un bit à 1

$$S_1 = x_3 + \overline{x_3}\,\overline{x_2}\,\overline{x_1}\,x_0$$
$$= x_3 + \overline{x_2}\,\overline{x_1}\,x_0$$
(par simplification)

$$S_0 = \overline{x_3}\,(x_2 + x_1 + x_0)$$

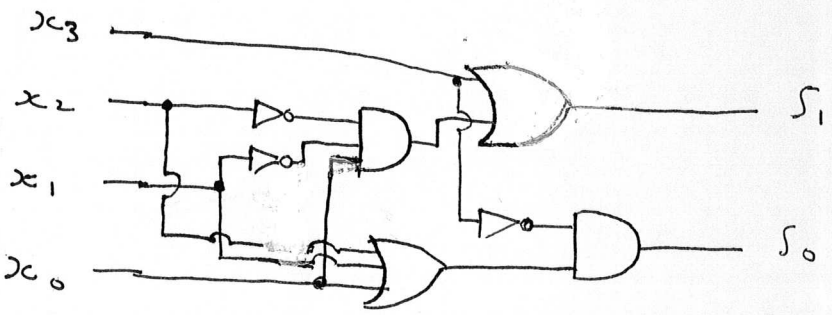
On pourrait aussi obtenir ces résultats avec des tables de Karnaugh :

S_1 :

x_3x_2 \ x_1x_0	00	01	11	10
00	0	1	0	0
01	0	0	0	0
11	1	1	1	1
10	1	1	1	1

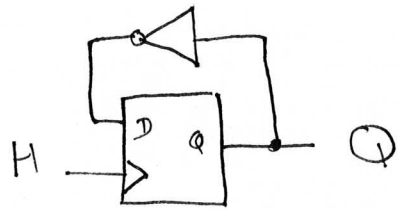
S_0 :

x_3x_2 \ x_1x_0	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	0	0	0	0
10	0	0	0	0

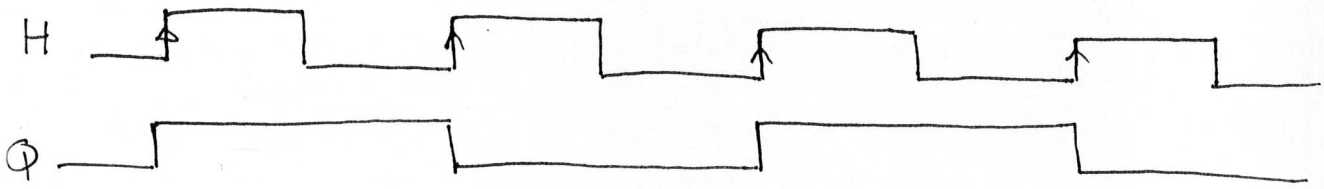


Q2

2.1)

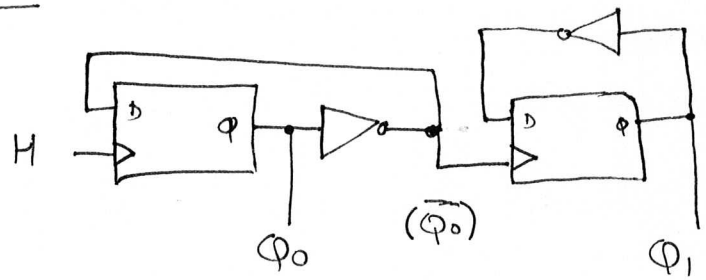


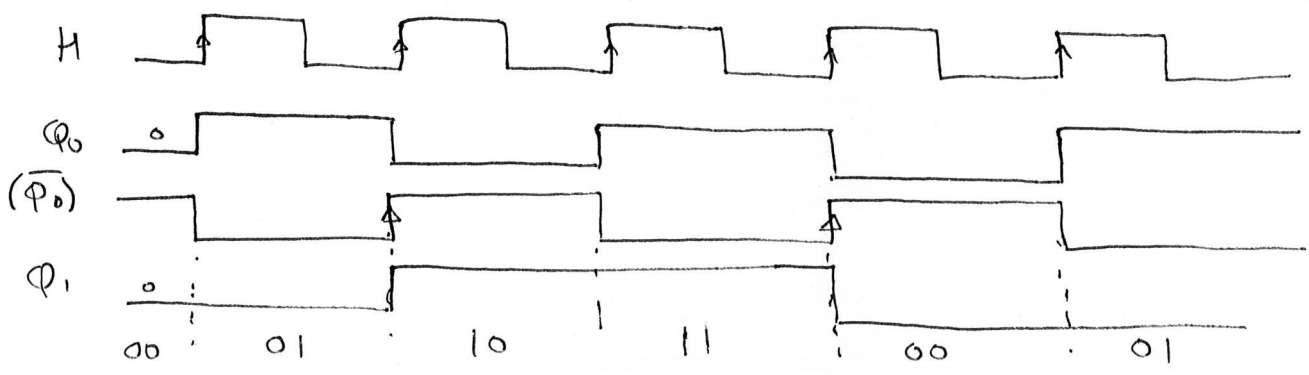
2.2)



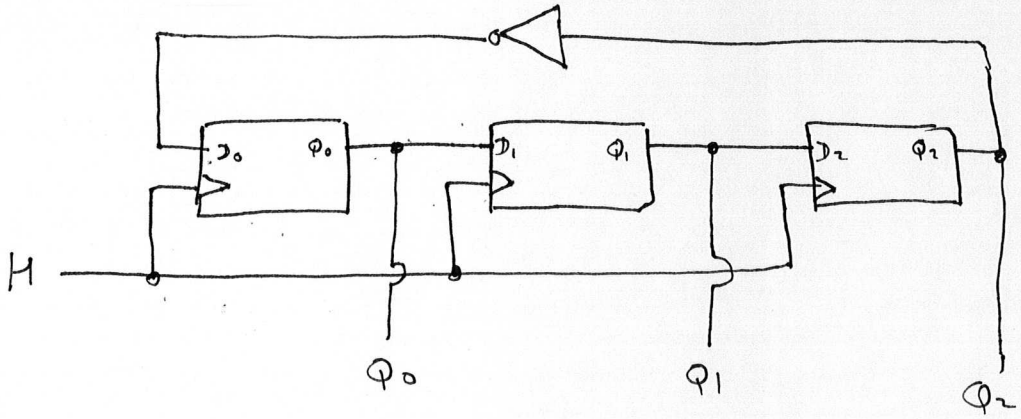
La fréquence du signal Φ est égale à la moitié de celle de H.

2.3)





2.4)



Q3

3.1) Les additions et multiplications sont entrelacées, donc on ne pourra pas faire grand' chose en termes d'optimisation malgré le pipe-line.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
f_1	F	D	$+_1$	$+_2$	W	→																				
f_2		F	D	D	D	D	$*_1$	$*_2$	$*_3$	$*_4$	W															Dépendance à R2
f_3			F	F	F	F	D	D	D	D	$+_1$	$+_2$	W	→												Attente libération pipe-line calcul
f_4							F	F	F	F	D	D	D	D	$*_1$	$*_2$	$*_3$	$*_4$	W							Dépendance à R2
f_5											F	F	F	F	D	D	D	D	$+_1$	$+_2$	W					Attente libération pipe-line calcul
f_6															F	F	F	F	D	D	$*_1$	$*_2$	$*_3$	$*_4$	W	//

- On est obligé d'attendre qu'une donnée calculée soit passée dans l'étage W pour pouvoir la lire à l'étage D, en l'absence de "data forwarding"
- Tant qu'on ne peut réaliser l'étape D (attente d'étage W ou de pipe-line), on "stalle" dans l'étage D : il reste occupé par l'instruction en cours de traitement, qui sera re-traitée au cycle suivant si cela est possible.

3.2)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	
f_1	F	D	$+_1$	$+_2$	W	→																			
f_2		F	D	D	D	D	$*_1$	$*_2$	$*_3$	$*_4$	W														Dépendance à R2
f_3			F	F	F	F	D	$+_1$	$+_2$	$+_3$	$+_4$	W	→												Stalling sur l'étage W (pas de doublage)
f_4							F	D	D	D	D	D	D	$*_1$	$*_2$	$*_3$	$*_4$	W							Dépendance à R2
f_5								F	F	F	F	F	F	D	$+_1$	$+_2$	$+_3$	$+_4$	W						Stalling sur l'étage W
f_6														F	D	D	D	D	D	$*_1$	$*_2$	$*_3$	$*_4$	W	Dépendance à R4

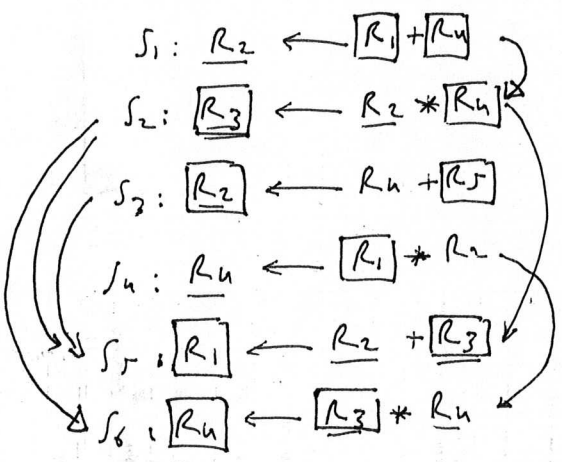
- On le voit, l'indépendance entre les pipe-line n'est pas ce qui permet de gagner beaucoup de cycles, avec le code tel qu'il est ordonné. Les dépendances et attentes entre données. Le fait que deux instructions ne puissent se "doubler" est aussi pénalisant.

3.3)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
S_1	F	D	$+_1$	$+_2$	W												
S_2		F	D	D	$*_1$	$*_2$	$*_3$	$*_4$	W								Dépendance à R2 mais data forwarding
S_3			F	F	D	$+_1$	$+_2$	$+_2$	$+_2$	W							Stalling sur l'étage W
S_4					F	D	D	D	$*_1$	$*_2$	$*_3$	$*_4$	W				Dépendance à R2 mais data forwarding
S_5						F	F	D	$+_1$	$+_2$	$+_2$	$+_2$	W				Stalling sur l'étage W
S_6								F	D	D	D	$*_1$	$*_2$	$*_3$	$*_4$	W	Dépendance à R4 mais data forwarding

On voit ici que, grâce au "data forwarding", la contrainte de non-"doublage" des instructions à l'étage W n'est en fait pas si pénalisante que cela.

3.4) Le code initial est le suivant :



- Les flèches représentent les dépendances "RAW" (registres soulignés)
- Les registres encadrés sont ceux dont le numéro ne peut être changé car on a besoin de conserver leur valeur à la fin de l'exécution du fragment, ou de la lire au début

On peut casser l'anti-dépendance entre S_1 et S_3 en renommant R_2 en R_6.
On peut réordonner pour insérer une instruction "utile" entre S_1 et S_2, pour ne pas attendre la fin du calcul dans le pipe-line de S_2.

S_1 : $R_6 \leftarrow R_1 + R_4$

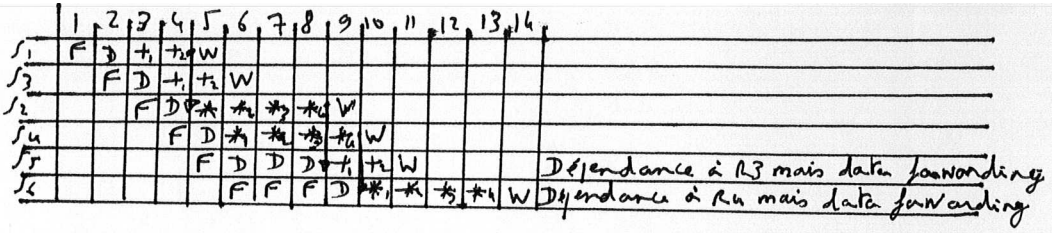
S_3 : $R_2 \leftarrow R_4 + R_5$

S_2 : $R_3 \leftarrow R_6 * R_4$

S_4 : $R_4 \leftarrow R_1 * R_6$

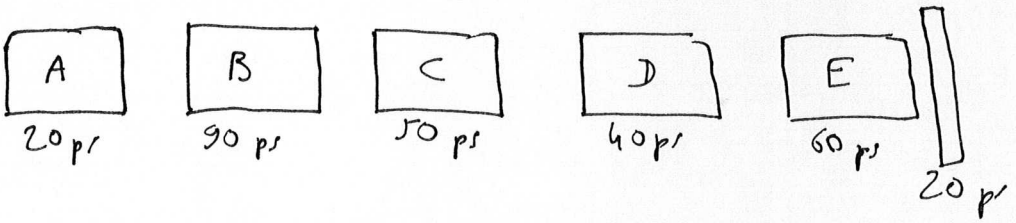
S_5 : $R_1 \leftarrow R_6 + R_3$

S_6 : $R_4 \leftarrow R_3 * R_4$

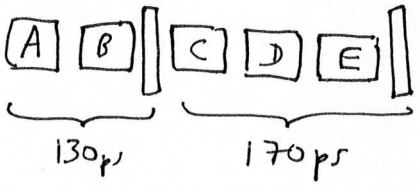


Q4

On a les blocs suivants (avec le registre final) :



4.1)

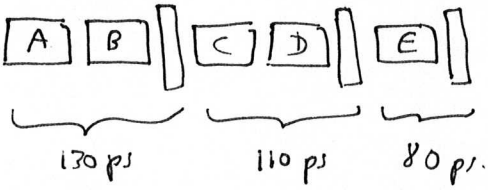


DM = 170 ps

Débit = $\frac{10^{12}}{170} \approx 5{,}88$ GHz

LM = 2 × DM = 340 ps

4.2)

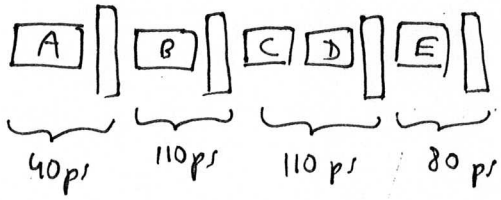


DM = 130 ps

Débit = $\frac{10^{12}}{130}$

LM = 3 × DM = 390 ps

4.3)



DM = 110 ps

Débit = $\frac{10^{12}}{110}$

LM = 4 × DM = 440 ps

On utilise 4 registres.

On ne peut faire mieux, car le bloc B donne la durée minimale et ne peut pas être coupé.

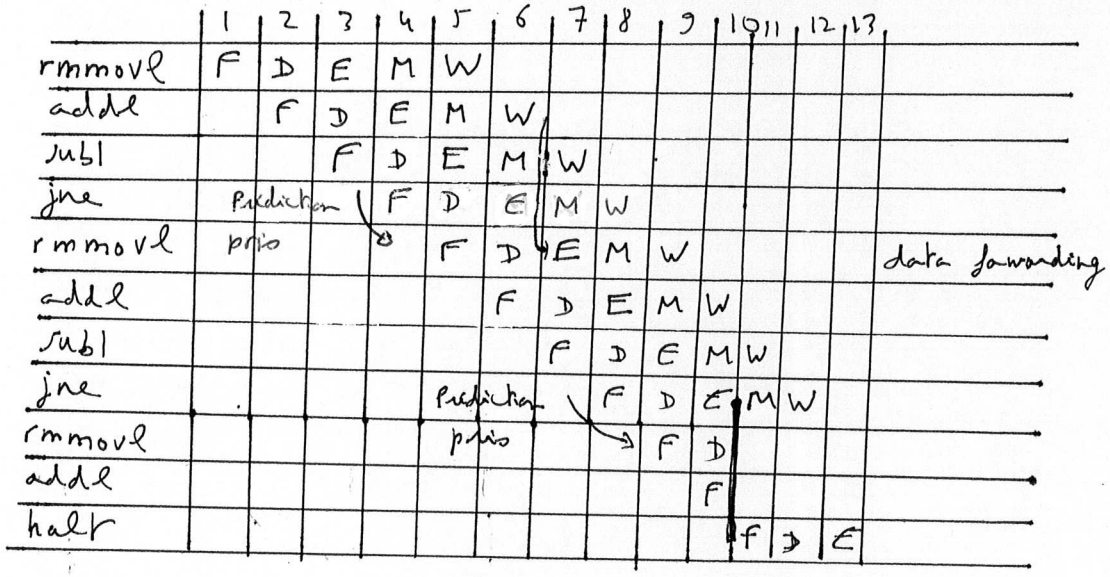
Avec un cinquième registre, on ne ferait qu'augmenter la latence sans améliorer le débit.

Q5

5.1) Ce programme initialise à zéro la zone mémoire (le tableau) commençant à l'adresse "t" et de taille "size" (ici, 2) entiers.

On suppose que les branchements conditionnels sont toujours pris, selon ce qu'indique le code HCL.

On a le déroulement suivant :



Le délai est donc de 10 cycles

5.2)

L'instruction RTOSL doit :

- récupérer la valeur de %eax dans valA, à travers le numéro rA
- " " " " %edi dans valB, " " " " rB
- incrémenter rB de 4 à travers l'ALU, ce qui donne valE
- utiliser cependant l'ancienne valeur de edi comme adresse de l'écriture mémoire. C'est donc valB qui doit être utilisé comme adresse, et non pas valE (ce n'est pas prévu dans le câblage traditionnel du processeur pipe-line, mais facile à réaliser)

Voir mentions feuille jointe.

5.3) Les intérêts d'utiliser une telle instruction sont :

- Gain d'un cycle (une instruction de moins à traiter)
- Économie de place mémoire dans le programme (une instruction de moins)
- Économie d'un registre à initialiser avec la valeur d'incrémentation (et donc aussi une instruction de moins à effectuer)

De par les arguments ci-dessus, si l'on utilise cette instruction dans le programme, on gagnera 3 cycles :

- 2 pour les 2 tours de boucle
- 1 pour l'instruction irmovl 4, %edx qui sera supprimée

5.4)

Ici, c'est %ecx que l'on chargera dans valB, pour lui ajouter la valeur -1

Voir mentions feuille jointe

5.5) Pour ne pas que l'instruction située en destination de l'instruction LOOP entre dans l'étage d'exécution au cycle suivant, il faut que le résultat de la décrémentation soit utilisable quasiment au moment où il est calculé, ce qui n'est pas faisable en pratique.
En revanche, on peut savoir <u>à l'avance</u> dans quel cas on aura fait une mauvaise prédiction : c'est quand, à la fin de l'étage de décodage, on se retrouve avec D_icode == LOOP et valB == 1, car on sait qu'après l'étage d'exécution on aura bien valE à 0.
On peut donc prendre ce cas particulier en charge dans le calcul du bit de mauvaise prédiction de branchement.

— Feuille à détacher et à rendre avec la copie —

Numéro d'anonymat :

```
################ Fetch Stage     ###################################

## What address should instruction be fetched at
int f_pc = [
# Mispredicted branch.  Fetch at incremented PC
M_icode == JXX && !M_Bch : M_valA;
# Completion of RET instruction.
W_icode == RET : W_valM;
# Default: Use predicted value of PC
1 : F_predPC;
];

# Predict next value of PC
int new_F_predPC = [
f_icode in { JXX, CALL } : f_valC;
1 : f_valP;
];

################ Decode Stage ######################################

## What register should be used as the A source?
int new_E_srcA = [
D_icode in { RRMOVL, RMMOVL, OPL, PUSHL  } : D_rA;
D_icode in { POPL, RET } : RESP;
1 : RNONE; # Don't need register
];

## What register should be used as the B source?
int new_E_srcB = [
D_icode in { OPL, RMMOVL, MRMOVL  } : D_rB;
D_icode in { PUSHL, POPL, CALL, RET } : RESP;
1 : RNONE;  # Don't need register
];

## What register should be used as the E destination?
int new_E_dstE = [
D_icode in { RRMOVL, IRMOVL, OPL} : D_rB;
D_icode in { PUSHL, POPL, CALL, RET } : RESP;
1 : RNONE;  # Don't need register
];

## What register should be used as the M destination?
int new_E_dstM = [
D_icode in { MRMOVL, POPL } : D_rA;
1 : RNONE;  # Don't need register
];

## What should be the A value?
## Forward into decode stage for valA
int new_E_valA = [
D_icode in { CALL, JXX } : D_valP; # Use incremented PC
d_srcA == E_dstE : e_valE;    # Forward valE from execute
d_srcA == M_dstM : m_valM;    # Forward valM from memory
d_srcA == M_dstE : M_valE;    # Forward valE from memory
d_srcA == W_dstM : W_valM;    # Forward valM from write back
```

, Loop

, Loop

D_icode in {STOSL} : REAX;

D_icode in {STOSL} : REDi;
D_icode in {Loop} : RECX;

```
d_srcA == W_dstE : W_valE;    # Forward valE from write back
1 : d_rvalA;  # Use value read from register file
];

int new_E_valB = [
d_srcB == E_dstE : e_valE;    # Forward valE from execute
d_srcB == M_dstM : m_valM;    # Forward valM from memory
d_srcB == M_dstE : M_valE;    # Forward valE from memory
d_srcB == W_dstM : W_valM;    # Forward valM from write back
d_srcB == W_dstE : W_valE;    # Forward valE from write back
1 : d_rvalB;  # Use value read from register file
];

################ Execute Stage ######################################

## Select input A to ALU
int aluA = [
E_icode in { RRMOVL, OPL } : E_valA;
E_icode in { IRMOVL, RMMOVL, MRMOVL } : E_valC;
E_icode in { CALL, PUSHL } : -4;
E_icode in { RET, POPL } : 4;
# Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
E_icode in { RMMOVL, MRMOVL, OPL, CALL,
     PUSHL, RET, POPL } : E_valB;
E_icode in { RRMOVL, IRMOVL } : 0;
# Other instructions don't need ALU
];

## Set the ALU function
int alufun = [
E_icode == OPL : E_ifun;
1 : ALUADD;
];

## Should the condition codes be updated?
bool set_cc = E_icode == OPL;

################ Memory Stage ######################################

## Select memory address
int mem_addr = [
M_icode in { RMMOVL, PUSHL, CALL, MRMOVL } : M_valE;
M_icode in { POPL, RET } : M_valA;
# Other instructions don't need address
];

## Set read control signal
bool mem_read = M_icode in { MRMOVL, POPL, RET };

## Set write control signal
bool mem_write = M_icode in { RMMOVL, PUSHL, CALL };
```

, STOSL

E_icode in {LOOP}: -1;

, STOSL , LOOP

M_icode in {STOSL}: M_valB;

, STOSL