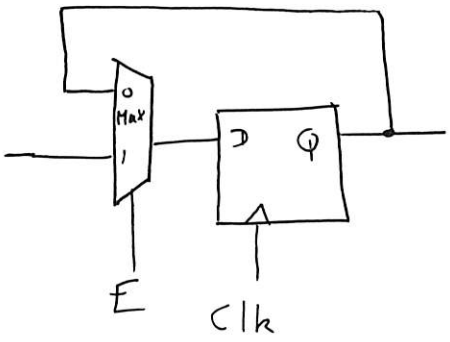
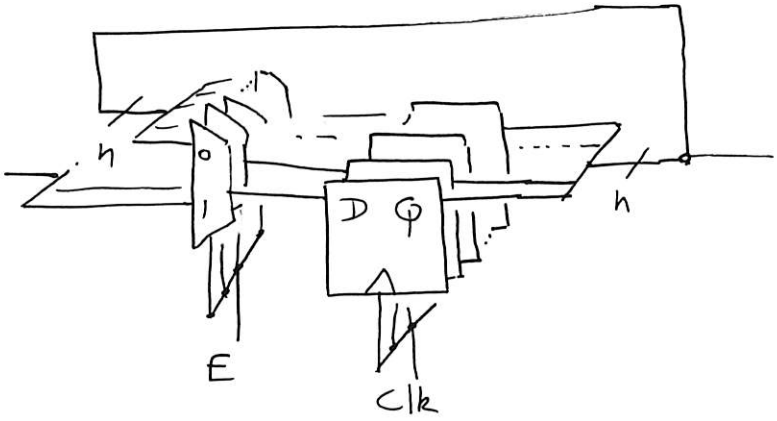
Exercice 1

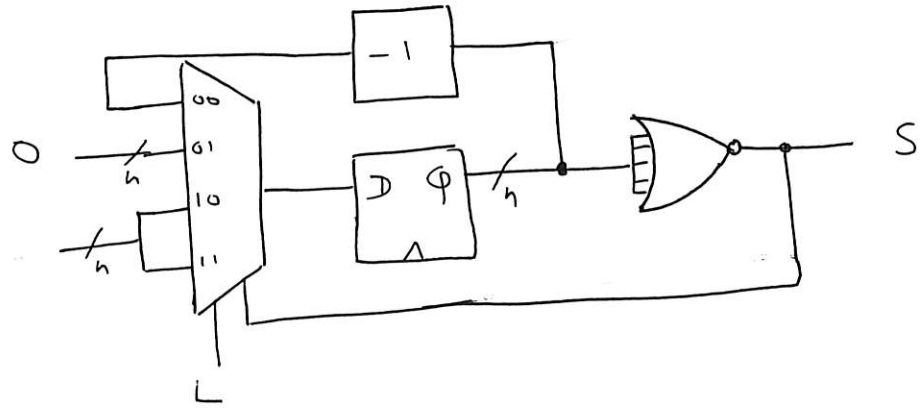
(1.1)



Il suffit de mettre en parallèle ce circuit :

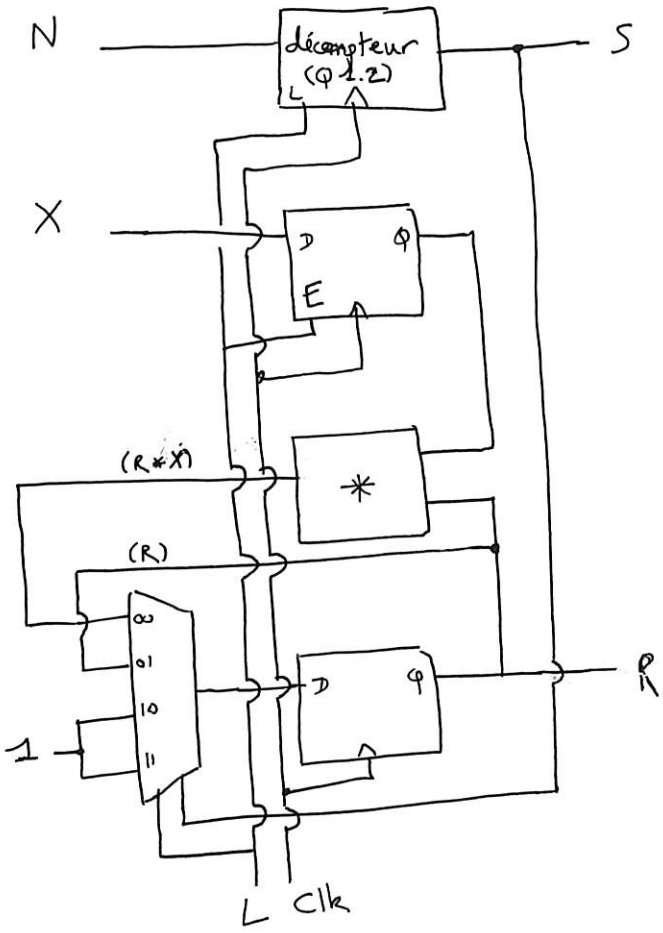


(1.2) On peut encore utiliser un multiplexeur :



(1.3)

Il suffit de combiner les circuits vus précédemment :



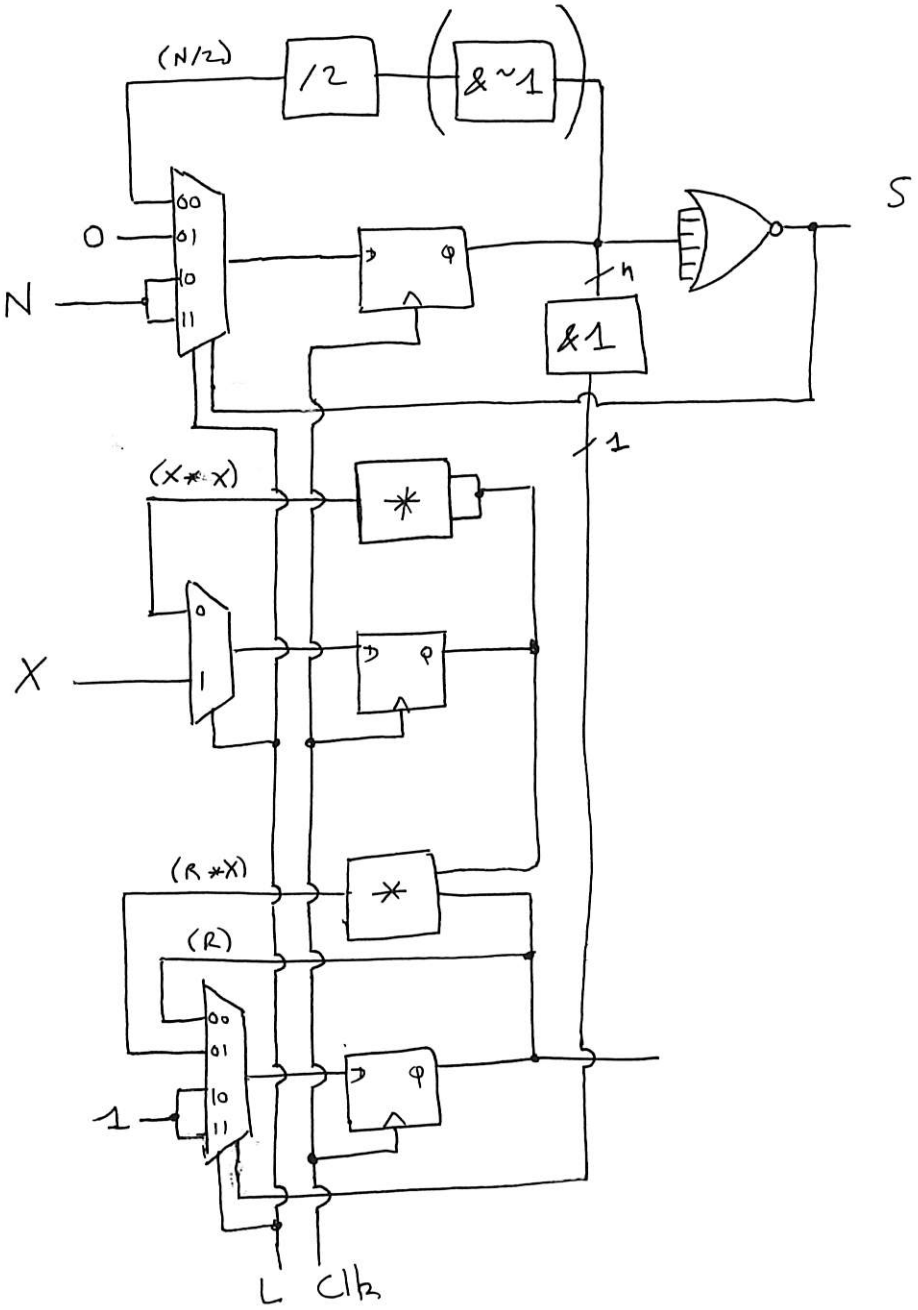
(1.4)

- Une valeur entière est impaire si son bit de poids le plus faible est à 1.
- Pour diviser un nombre entier positif par deux, il suffit de le décaler d'un cran vers la droite. En termes d'électronique, cela ne coûte rien : il suffit de garder les (n-1) fils les plus à gauche, et de mettre un nouveau fil de poids fort à la valeur 0.

(1.5)

Il faut faire évoluer en parallèle les valeurs de N, X et R.

On a donc, ici encore, besoin d'un multiplexeur devant chaque bascule D.



Le bloc $\overset{n}{\rightarrow}$ [&1] $\overset{1}{\rightarrow}$ prend la valeur du fil de bit de poids le plus faible.

Le bloc [&~1] met à zéro le bit de poids le plus faible, pour faire le (-1) dans le cas où N est impair. En fait, il est inutile, car le bloc [/2] élimine le bit de poids le plus faible.

Exercice 2

Le "data forwarding" est un mécanisme permettant de limiter l'apparition de bulles dans les architectures pipe-linées.

Par défaut, dans ces architectures, les valeurs des registres sont lues à l'étage "decode", et ré-écrites à l'étage "write-back".

De fait, si une instruction dépend des valeurs produites par une instruction en cours d'exécution, l'instruction dépendante est bloquée dans le pipe-line jusqu'à ce que l'autre instruction arrive à l'étage "write-back", donnant lieu à la création de bulles dans le pipe-line.

Le "data-forwarding" consiste à rediriger les valeurs produites par les instructions précédentes, depuis les étages "execute" et "memory", sans attendre l'étage "write-back", vers l'étage "decode", en contournant la lecture de la banque de registres. Ainsi, on peut faire progresser les instructions au-delà de l'étage "decode" dès que leurs valeurs sont disponibles.

Exercice 3

(3.1) Ces instructions sont décrites sur les diapositives du cours.

Etage	mrmovl vC(rB),rA	popl rA	call valC
Fetch	ic:if = M_1[PC] rA:rB = M_1[PC+1] valC = M_4[PC+2] valP = PC+6	ic:if = M_1[PC] rA:rB = M_1[PC+1] valP = PC+2	ic:if = M_1[PC] valC = M_4[PC+1] valP = PC+5
Decode	valB = R[rB]	valA = R[%esp] valB = R[%esp]	valB = R[%esp]
Execute	valE = valB+valC	valE = valB+4	valE = valB+(-4)
Memory	valM = M_4[valE]	valM = M_4[valA]	M_4[valE] = valP
Write-Back	r[rA] = valM	R[%esp] = valE R[rA] = valM	R[%esp] = valE
PC update	PC = valP	PC = valP	PC = valC

(3.2) On déduit de ci-dessus les ajouts suivants (seules les lignes modifiées sont indiquées) :

```
int srcA = [    icode in { POPL } : RESP;    ]

int srcB = [    icode in { MRMOVL } : rB;
                icode in { POPL, CALL } : RESP;    ]

int dstE = [    icode in { POPL, CALL } : RESP;    ]

int aluA = [    icode in { MRMOVL } : valC;
                icode in { POPL } : 4;
                icode in { CALL } : -4;            ]

int aluB = [    icode in { MRMOVL, POPL, CALL } : valB;    ]

int mem_addr = [    icode in { MRMOVL, CALL } : valE;
                    icode in { POPL } : valA;              ]
```