Nom : _____ **Prénom :** _____

université angers
FACULTÉ DES SCIENCES
Unité de formation et de recherche

Département Informatique
L3 Informatique
Jean-Michel Richer
Architecture des Ordinateurs
2023/2024[1]

jean-michel.richer@univ-angers.fr

Contrôle Continu 2023 2/2

Aucun document ni outil électronique n'est autorisé. Vous devez soigner votre écriture (la copie ne doit pas être un brouillon, écrire lisiblement) et expliquer clairement votre raisonnement. L'absence de raisonnement ou des étapes de calcul conduit à l'absence de point.

Exercice 1 - 4 pts, 20 min - Soit le code suivant pour lequel initialement **eax** = 102, **edx** = 0, **edi** = 13 et **esi** = 5.

Instruction	eax	edx	edi	esi
	102	0	13	5
`div esi ; i1`				
`lea edi, [esi + esi*4] ; i2`				
`add eax, 27 ; i3`				
`shr eax, 3 ; i4`				
`add eax, edi ; i5`				
`lea edi, [edx + esi*4] ; i6`				
`popcnt eax, edi ; i7`				
`and eax, 7 ; i8`				

Indiquez ce que contiennent chacun des registres **eax**, **edx**, **esi**, **edi** en utilisant un tableau où les colonnes représentent les registres et les lignes les instructions `i1` à `i8`. On n'indiquera que les modifications pour le registre concerné à l'exécution de chaque instruction.

Exercice 2 - (4 pts, 30 min) - Nombre premier

Soit la fonction C++ suivante qui vérifie si un nombre entier n est premier ou non.

```cpp
bool is_prime(int n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if ((n % 2) == 0) return false;
    for (int i = 3; i <= static_cast<int>(sqrt(n)); i += 2) {
        if ((n % i) == 0) return false;
    }
    return true;
}
```

src/is_prime.cpp

Donnez une traduction assembleur x86 32 bits du code C précédent. Le fait d'utiliser la fonction `sqrt` impose d'utiliser juste pour ce calcul la FPU. Le reste du code utilise les entiers. On rappelle qu'il existe une instruction `fist/fistp [mem]` qui convertit la valeur dans `st0` en un entier et stocke cette valeur en mémoire.

Exercice 3 - 8 pts, 50 min - Codage assembleur

```cpp
float prod_sum(float *x, float *y, float *z, int size, float a) {
    float sum = 0.0;

    for (int i = 0; i < size; ++i) {
      z[i] = x[i] * (y[i] - z[i]) / a;
      sum += z[i];
    }
    return sum;
}
```

src/prod_sum_1.cpp

1. donnez le code assembleur x86 32 bits de la fonction `prod_sum` sans vectorisation. On indiquera au préalable l'association variables / registres.
2. donnez le code C/C++ avec un dépliage par 4 dans le cas général (si la taille n'est pas multiple de 4)
3. donnez une version assembleur vectorisée (SSE) de la fonction en utilisant les instructions `movaps`, `mulps`, `addps`, `subps`, `divps`, `haddps`, `movss` et `shufps` ou `pshufd`. On stockera $x[i:i+3]$, $y[i:i+3]$ et $z[i:i+3]$ dans **`xmm1`**, **`xmm2`**, **`xmm3`** et **`xmm4`** contiendra quatre fois a. Le registre **`xmm0`** sera utilisé pour stocker les sommes partielles.

Note : on considère que les vecteurs `x` ,`y` et `z` sont alignés sur des adresses multiples de 16 et ont une taille multiple de 4.

Exercice 4 - 4 pts, 20 min - Répondre aux questions de cours en expliquant et détaillant votre réponse : maximum 10 lignes par réponse.

1. rappelez quelles sont les 5 étapes de traitement d'une instruction assembleur
2. qu'est ce que la prédiction de branchement et à quoi sert-elle ?
3. pourquoi observe t-on des temps de calcul plus importants que la normale lors du produit de matrice pour certaines dimensions ?
4. entre un Intel Core i3-6100 et un i5-7400, lequel de ces processeurs est le plus performant et dans quels cas ?