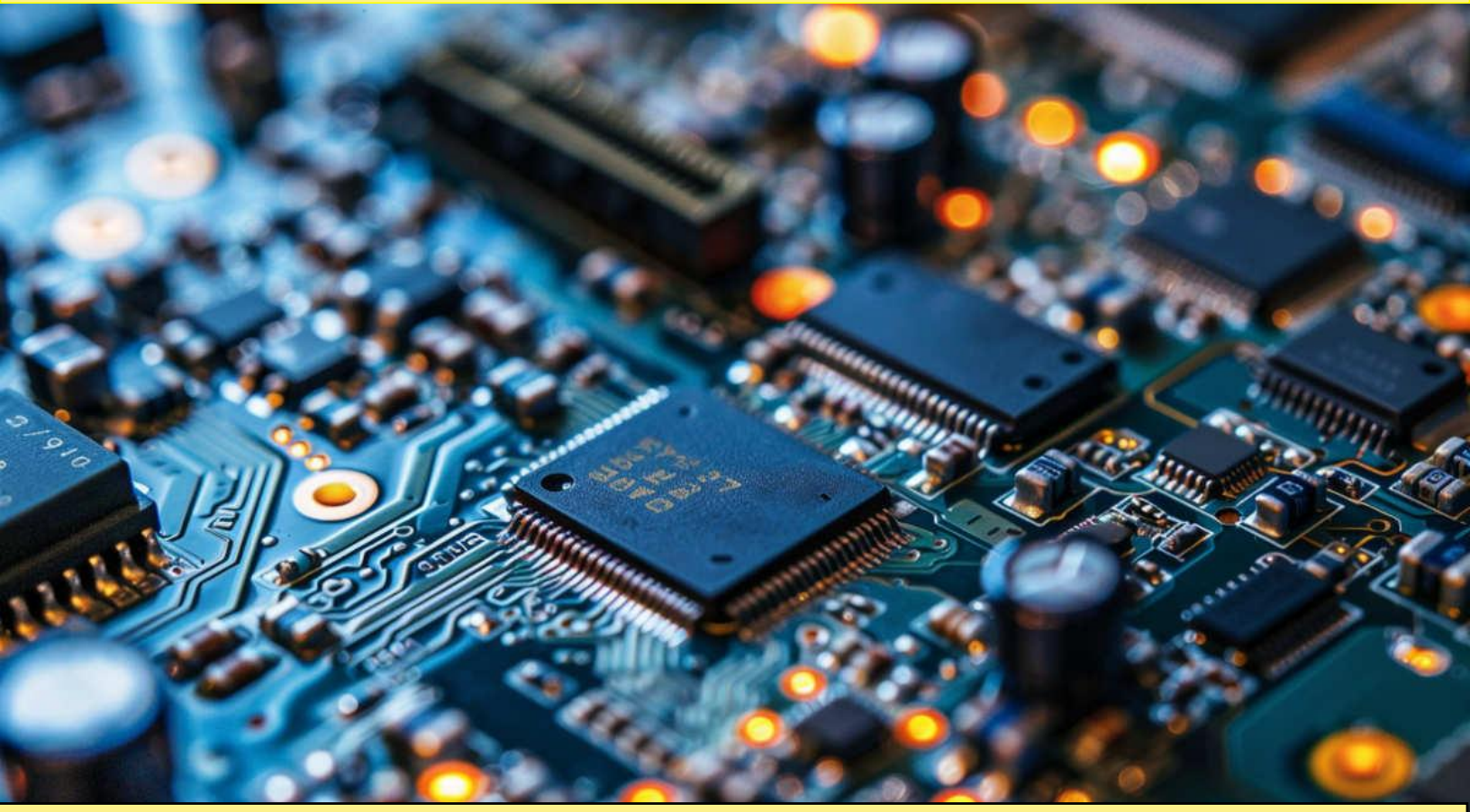
CPI A2 – PROJET SYSTEMES EMBARQUÉS

PROSIT 2 – Un registre de performance !



CONTEXTE :

INFORMATIONS IMPORTANTES :

-

MOTS INCONNUS :

- MOVE (MOV): Instruction qui transfère des données d'un emplacement mémoire ou registre à un autre. Cela ne fait qu'une copie des données, sans les modifier.

  Ex : `MOV AX, BX` (copie la valeur de BX dans AX).

- ADD : Instruction qui additionne deux valeurs et stocke le résultat dans un registre ou une mémoire. La valeur initiale dans le registre est remplacée par le résultat de l'addition.

  Ex : `ADD AX, BX` (ajoute la valeur de BX à celle de AX, et le résultat est stocké dans AX).

- Registres : Ensemble de données ou d'informations stockées, souvent pour référence ou suivi. Effectue le role de la mémoire

- ASM (Assembly): Langage de programmation de bas niveau qui communique directement avec le matériel de l'ordinateur.

- En **Assembly**, **AUL** (Arithmetic Unit Logic) n'est pas une instruction standard, mais semble se référer à l'**ALU** (Arithmetic Logic Unit). Voici une définition de l'**ALU** :

- ATM
- EEMPROM
- ALU (Arithmetic Logic Unit) : C'est la partie du processeur qui effectue les opérations arithmétiques (comme l'addition, la soustraction) et logiques (comme ET, OU). Elle est essentielle pour le traitement des données et l'exécution des instructions dans un programme.
- Data SRAM : bus qui transporte les informations
- Data bus : transport d’informations d’un composant à un autre

PROBLEMATIQUE :

Comprendre comment fonctionne un microcontrôleur pour compléter le code en C

#optimiser le traitement des données d'une station météo avec un microcontrôleur, en garantissant la précision des calculs d'algorithmes comme la suite de Fibonacci, tout en surmontant les défis liés à l'architecture logicielle et aux erreurs dans le code d'assemblage ?

PLAN D’ACTION :

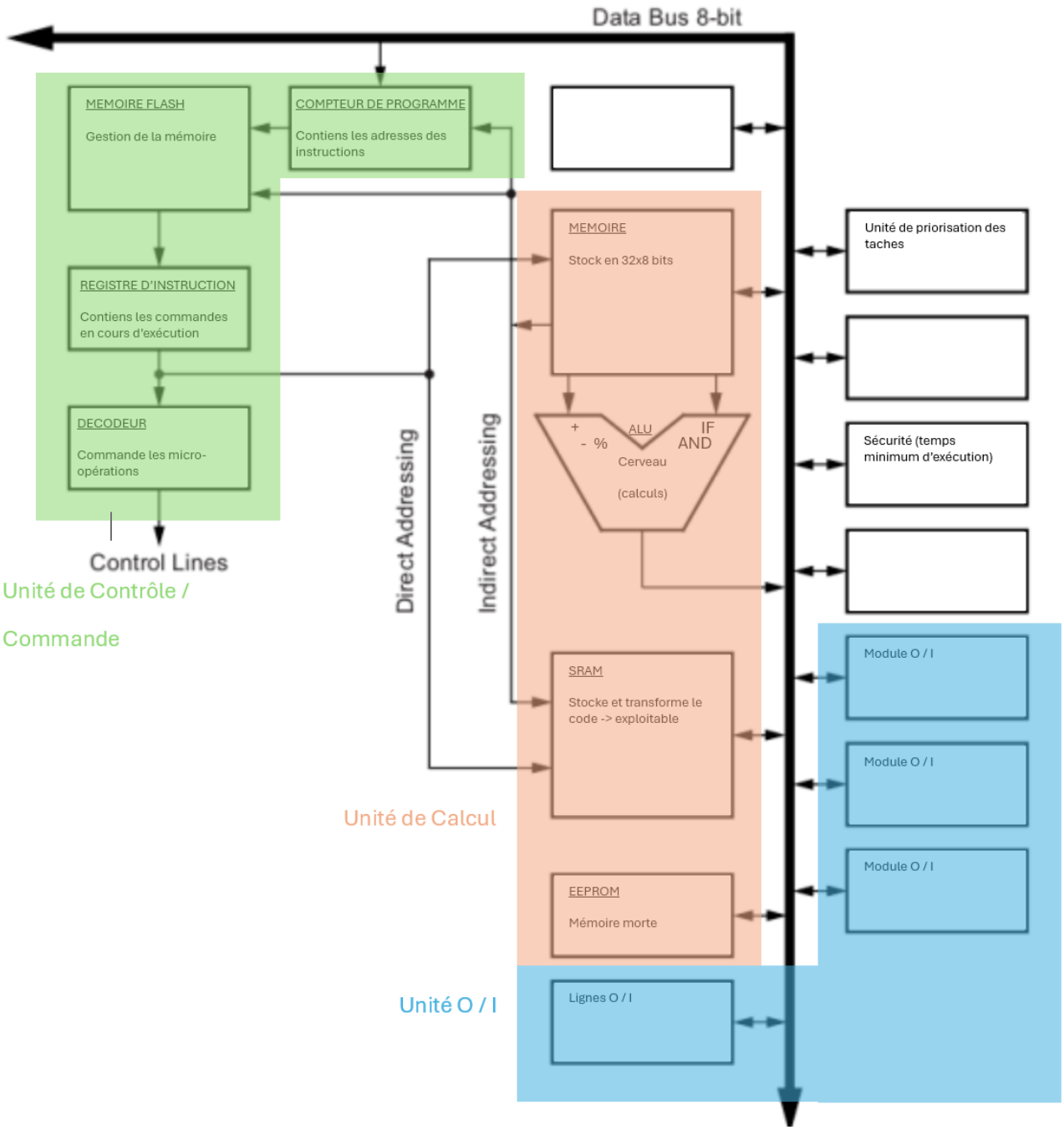
- Analyser les données et comprendre le contexte [60min].

- Comprendre le fonctionnement des microcontrôleurs et l’architecture du processeur [20min].

-Programmer en C la suite de Fibonacci [min].

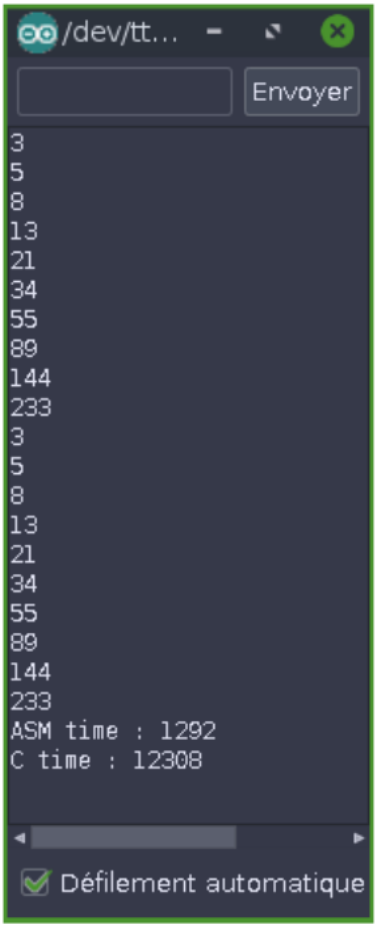
- Rendre le travail [5min].

REALISATION :


Data Bus 8-bit
MEMOIRE FLASH
Gestion de la mémoire
COMPTEUR DE PROGRAMME
Contiens les adresses des instructions
REGISTRE D'INSTRUCTION
Contiens les commandes en cours d'exécution
DECODEUR
Commande les micro-opérations
Control Lines
Unité de Contrôle / Commande
Direct Addressing
Indirect Addressing
MEMOIRE
Stock en 32x8 bits
+ - % ALU
IF AND
Cerveau
(calculs)
SRAM
Stocke et transforme le code -> exploitable
EEPROM
Mémoire morte
Unité de Calcul
Unité de priorisation des taches
Sécurité (temps minimum d'exécution)
Module O / I
Module O / I
Module O / I
Lignes O / I
Unité O / I

```
volatile uint8_t aa asm("aa")= 1;
volatile uint8_t bb asm("bb")= 2;
volatile uint8_t rASM asm("rASM") = 0;

void setup() {
  Serial.begin(9600);
  long start = micros();
  int compt = 0;
  int compt2 = 0;
  int limit = 11;



  while(compt<limit){
    fiboASM();
    compt++;
  }

  long endASM = micros();

  while(compt2<limit){
    fibo();
    compt2++;
  }

  long endN = micros();
  Serial.print("ASM time : ");  Serial.println(endASM - start);
  Serial.print("C time : ");    Serial.println(endN - endASM);
}


long fiboASM(){

  asm
  (
    "lds [0x12], 1        \n"
    "lds [0x10], (aa)     \n"
    "lds [0x11], (bb)     \n"
    "add [0x12], [0x10]   \n"
    "add [0x12], [0x11]   \n"
    "sts (aa), [0x11]     \n"
    "sts (bb), [0x12]     \n"
    "sts (rASM), [0x12]   \n"
    );

    Serial.println(rASM);
}

long a = 1;
long b = 2;
long fibo(){
  long r = a + b;
  a = b;
  b = r;
  Serial.println(r);
  }

void loop(){
```

```
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
  long start  = micros();
  int compt = 0;
  int compt2 = 0;
  int limit = 10;

  asm("LDI R16, 1");
  asm("LDI R17, 2");
  while(compt < limit){
    fiboASM();
    compt++;
  }

  long endASM = micros();

  while(compt2 < limit){
    fibo();
    compt2++;
  }

  long endN = micros();
  Serial.print("ASM time : ");Serial.println(endASM - start);
  Serial.print("C time : ");Serial.print(endN - endASM);
}



unsigned long long fiboASM(){
   long rASM = 0;
    asm("MOV R20,%0 \n\t"
        "ADD R20,R16 \n\t"
        "ADD R20,R17 \n\t"
        "MOV R16,R17 \n\t"
        "MOV R17,R20 \n\t"
        "MOV %0, R20 \n\t"
        : "+r" (rASM));

   Serial.println(rASM);
}

int a = 1;
int b = 2;
int fibo() {
  int r = a + b;
  a = b;
  b = r;
  Serial.println(r);
}
```



Détails du code

Voici une explication détaillée ligne par ligne du code que vous avez partagé. Ce code est écrit en C pour Arduino avec des sections en ASM (Assembly) afin de comparer la

vitesse d'exécution entre deux versions du calcul de la suite de Fibonacci, l'une en C et l'autre en ASM.

Partie 1 : `void setup()`

1. **`Serial.begin(9600);`**

- Initialisation de la communication série à 9600 bits par seconde. Cela permet à la carte Arduino d'envoyer des données au moniteur série pour affichage.

2. **`long start = micros();`**

- Capture le temps actuel en microsecondes juste avant de commencer la boucle ASM. `micros()` retourne le nombre de microsecondes écoulées depuis que la carte a commencé à fonctionner.

3. **`int compt = 0;` et `int compt2 = 0;`**

- Déclarations de compteurs pour les boucles de calcul Fibonacci en ASM et en C.

4. **`int limit = 10;`**

- Fixe une limite de 10 itérations pour calculer les 10 premiers termes de la suite de Fibonacci.

Partie 2 : ASM (Calcul de Fibonacci en ASM)

5. **`asm("LDI R16, 1");`**

- Chargement de la valeur `1` dans le registre R16 en ASM. Ce pourrait être le premier nombre de la séquence de Fibonacci.

6. **`asm("LDI R17, 2");`**

- Chargement de la valeur `2` dans le registre R17. Ce pourrait être le deuxième nombre de la séquence de Fibonacci.

7. **`while(compt < limit)`**

   - Boucle qui exécute le calcul en ASM pour un nombre de fois déterminé par `limit` (ici, 10 fois).

8. **`fiboASM();`**

   - Appel à la fonction `fiboASM()`, qui contient le calcul en assembleur pour la suite de Fibonacci.

9. **`compt++;`**

   - Incrémentation du compteur `compt` à chaque itération de la boucle.

10. **`long endASM = micros();`**

    - Capture du temps final après l'exécution de la partie ASM.

Partie 3 : Calcul de Fibonacci en C

11. **`while(compt2 < limit)`**

    - Boucle qui exécute le calcul de Fibonacci en C pour 10 itérations.

12. **`fibo();`**

    - Appel à la fonction `fibo()`, qui calcule la suite de Fibonacci en langage C.

13. **`compt2++;`**

- Incrémentation du compteur `compt2` à chaque itération de la boucle.

14. **`long endN = micros();`**

- Capture du temps final après l'exécution de la partie C.

15. **`Serial.print("ASM time : "); Serial.println(endASM - start);`**

- Affiche le temps pris par l'exécution de la partie ASM en soustrayant le temps initial (`start`) du temps final (`endASM`).

16. **`Serial.print("C time : "); Serial.print(endN - endASM);`**

- Affiche le temps pris par l'exécution de la partie C en soustrayant `endASM` (fin de la partie ASM) à `endN` (fin de la partie C).

Partie 4 : `fiboASM()` - Calcul de Fibonacci en ASM

17. **`long rASM = 0;`**

- Initialisation d'une variable `rASM` qui stockera le résultat du calcul en assembleur.

18. **Bloc `asm(...)`**

- Bloc de code assembleur qui effectue les opérations sur les registres pour calculer le Fibonacci :

  - **`MOV R20, R0`** : Déplace la valeur de R0 dans R20.

  - **`ADD R20, R16`** : Additionne R16 à R20.

  - **`ADD R20, R17`** : Additionne R17 à R20.

  - **`MOV R16, R17`** : Déplace la valeur de R17 dans R16.

  - **`MOV R17, R20`** : Déplace la valeur de R20 dans R17.

- **`MOV $0, R20`** : Remet à zéro R20.
- `rASM` stocke le résultat final de l'opération.

19. **`Serial.println(rASM);`**
    - Affiche le résultat du calcul en ASM sur le moniteur série.

Partie 5 : `fibo()` - Calcul de Fibonacci en C

20. **`int a = 1; int b = 2;`**
    - Initialisation des deux premiers termes de la suite de Fibonacci en C.

21. **`int fibo()`**
    - Fonction en C qui calcule la suite de Fibonacci :
      - **`int r = a + b;`** : Somme de `a` et `b` pour obtenir le terme suivant.
      - **`a = b;`** : Mise à jour de `a` pour devenir l'ancien `b`.
      - **`b = r;`** : Mise à jour de `b` pour devenir le nouveau terme calculé.
      - **`Serial.println(r);`** : Affiche le terme calculé.

Résumé :

Le code compare le temps d'exécution entre une implémentation de la suite de Fibonacci en assembleur et une implémentation en C. Il affiche les deux temps (`ASM time` et `C time`) sur le moniteur série, ainsi que les valeurs calculées pour chaque itération.

L'implémentation en ASM est optimisée pour être plus rapide que celle en C, ce qui se vérifie dans les résultats affichés :

- **ASM time : 1292 µs**

- **C time : 12308 µs**

Cela montre que l'exécution en assembleur est beaucoup plus rapide que celle en C, comme prévu.

Définition ASM Assembly :

ASM (ou **Assembly**) est un langage de programmation de bas niveau qui est très proche du code machine. Il permet d'interagir directement avec le matériel du processeur. Chaque instruction en assembleur correspond à une instruction spécifique du processeur.

Voici quelques points pour mieux comprendre **ASM** :

1. **Langage de bas niveau** : Contrairement aux langages de haut niveau comme C, Python ou Java, ASM vous permet de programmer directement des instructions qui seront exécutées par le processeur. Cela offre un contrôle très précis sur la gestion de la mémoire, des registres et des opérations internes du processeur.

2. **Spécifique à l'architecture** : Le langage assembleur est spécifique à l'architecture du processeur que vous utilisez. Par exemple, les instructions en assembleur pour un processeur ARM seront différentes de celles d'un processeur x86 ou AVR (comme sur les cartes Arduino).

3. **Optimisation et performance** : En utilisant ASM, il est possible d'écrire des programmes qui sont extrêmement optimisés et rapides, car vous pouvez éviter les surcoûts introduits par les compilateurs de langages de haut niveau. Dans votre exemple, la partie ASM est plus rapide que la version C.

4. **Registres et instructions** : Un programme en ASM manipule directement les **registres** (comme R16, R17 dans votre code) qui sont des petites zones de stockage à

l'intérieur du processeur, et effectue des opérations arithmétiques, des comparaisons ou des sauts conditionnels.

Utilisation dans votre code

Dans le contexte de votre code, **ASM** fait référence à la partie du programme qui calcule la suite de Fibonacci en utilisant des instructions en assembleur, tandis que la partie en **C** utilise des instructions plus abstraites. Les instructions assembleur permettent de réaliser ce calcul avec des opérations basiques comme des additions ou des mouvements de registres (par exemple, `MOV`, `ADD`).

Pourquoi utiliser ASM ?

1. **Performances accrues** : Comme on le voit dans votre exemple, le calcul de Fibonacci en ASM est beaucoup plus rapide qu'en C. En effet, ASM permet un contrôle total sur l'exécution et peut éviter certaines étapes superflues qui ralentissent le programme en C.

2. **Contrôle direct du matériel** : Avec ASM, vous pouvez directement contrôler et optimiser l'utilisation des registres et des opérations du processeur, ce qui peut être crucial pour des systèmes embarqués ou des applications où la performance et l'efficacité énergétique sont primordiales.

En résumé, **ASM** (Assembly) est un langage de programmation bas niveau qui permet d’écrire des instructions directement compréhensibles par le processeur, et il est utilisé lorsque des performances optimales sont nécessaires.