Pendichev Andon,

Taoufiqui Soulayman,

Grope 4

SAE13 - Découvrir un dispositif de transmission / SAE13 – IOT

Introduction

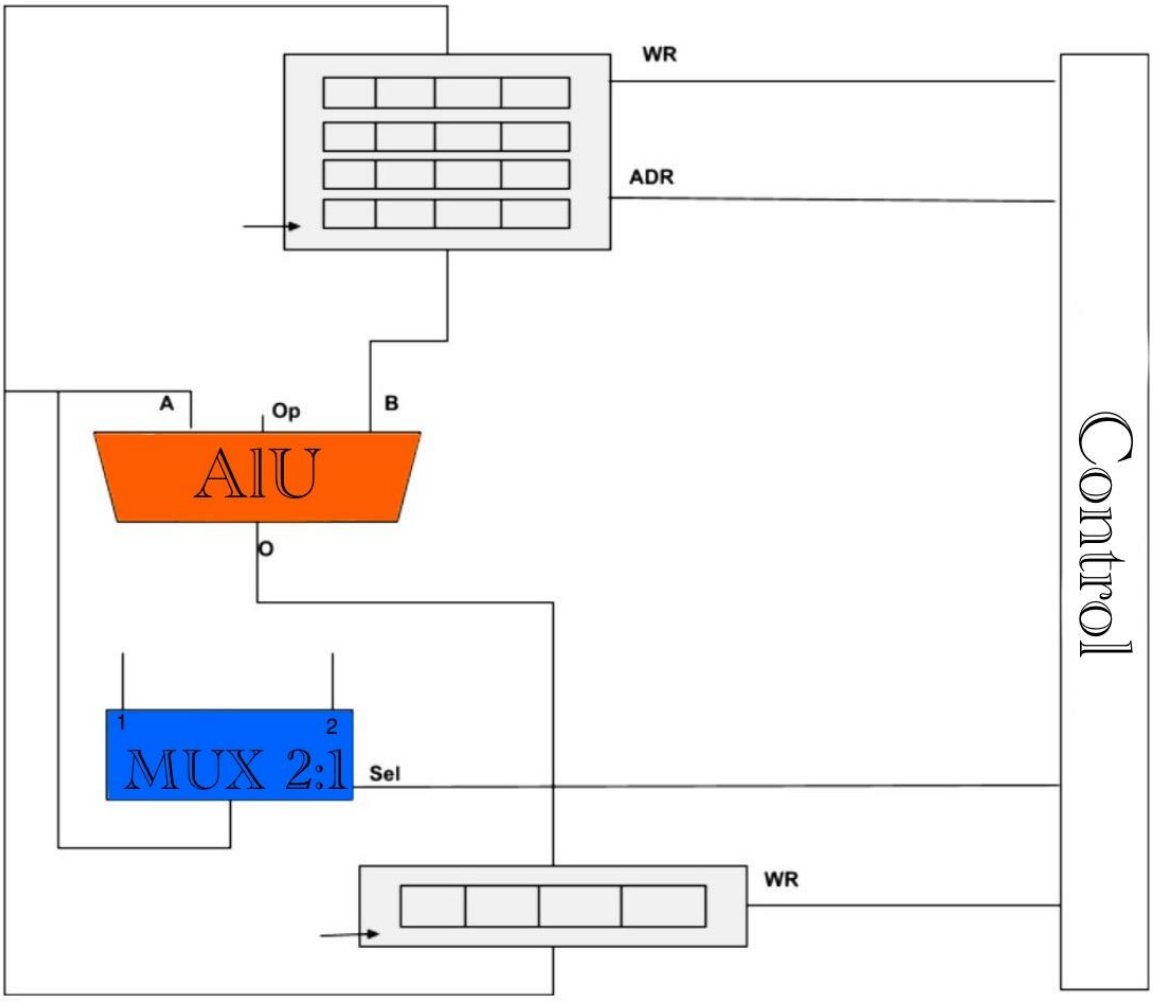
Un **microprocesseur 4 bits** constitue un composant fondamental de l'électronique numérique, conçu pour traiter des données de 4 bits simultanément. Bien que modeste en capacité comparé aux microprocesseurs plus avancés, il joue un rôle crucial dans de nombreuses applications nécessitant une logique simple et efficace, telles que les systèmes embarqués, les calculatrices ou encore les commandes de petits dispositifs électroniques. Ces microprocesseurs se distinguent par leur architecture épurée et leur capacité à exécuter des instructions élémentaires à l'aide d'une organisation interne optimisée.

Pour remplir sa mission, un microprocesseur 4 bits intègre plusieurs **composants essentiels** qui collaborent pour permettre l'exécution d'instructions, la manipulation de données et la gestion des flux de contrôle. Ces composants, bien qu'individuellement simples, forment ensemble une unité de traitement cohérente et performante. Parmi les éléments les plus importants, on retrouve :

1. **Banc de Registres (RB)** :
   Le banc de registres constitue une mémoire interne au microprocesseur, composée de plusieurs registres, chacun capable de stocker une valeur de 4 bits. Ces registres jouent un rôle clé en servant de stockage temporaire pour les données et les résultats intermédiaires lors de l'exécution des instructions. Leur accès rapide permet d'accélérer le traitement des opérations, réduisant ainsi la dépendance à une mémoire externe plus lente.
2. **Accumulateur (ACC)** :
   L'accumulateur est un registre particulier, souvent central dans l'architecture du microprocesseur. Il est spécifiquement utilisé pour les calculs arithmétiques et logiques, jouant le rôle de dépôt intermédiaire où les résultats des opérations sont stockés avant d'être éventuellement transférés vers d'autres registres ou mémoires. Son importance réside dans sa capacité à simplifier et à accélérer les interactions entre l'unité de calcul et les autres composants.
3. **Unité Arithmétique et Logique (UAL)** :
   L'UAL est le cœur computationnel du microprocesseur, responsable de l'exécution des opérations mathématiques (comme l'addition, la soustraction ou la multiplication) et logiques (telles que les opérations ET, OU et NON). En exploitant les données provenant des registres et de l'accumulateur, elle garantit la transformation et le traitement des informations nécessaires à la réalisation des programmes.
4. **Multiplexeur 2:1 (MUX 2:1)** :
   Le multiplexeur 2:1 est un circuit de commutation qui permet de sélectionner l'une des deux sources d'entrée pour l'acheminer vers une destination spécifique, comme l'UAL ou l'accumulateur. Cet élément joue un rôle essentiel dans la gestion des flux de données au sein du microprocesseur, en offrant une flexibilité dans la sélection des chemins de traitement et en optimisant l'utilisation des ressources disponibles.

Ensemble, ces composants forment une architecture compacte mais efficace, capable d'assurer les fonctions de base nécessaires à un système numérique. Cette simplicité intrinsèque en fait un outil privilégié pour les systèmes embarqués, où les contraintes de coût, de consommation énergétique et de complexité prédominent.

Schéma du microprcesseur



Au cours de cette Situation d'Apprentissage et d'évaluation, nous avons développé le codage Arduino d’un microprocesseur 4 bits capable d’exécuter quatre instructions : addition, soustraction, copie ACC/RB et initialisation d’ACC. Certains segments de code avaient déjà été élaborés lors de TPs précédents, mais chaque composant avait été programmé de manière indépendante. Nous avons donc réutilisé et ajusté ces codes pour assurer leur intégration et leur fonctionnement harmonieux au sein du microprocesseur.

Microprocesseur

Le code complété:

-->Dans la partie déclaration:

```
// Declaration of the variables of the Micro-processor

  Reg, Reg0, Reg1, Reg2, Reg3, ACC;
  Output4 ALU4, RB, MUX;
  ADD_SUB, ADD4, SUB4;
  bool Wr_RB, op, Adr[2], I[4], sel, Wr_ACC, Clk

//Assign output variables to GPIO pins
//cpnst int LED_C = 13;

const int Led_00 = 26;
const int Led_01 = 27;
const int Led_02 = 14;
const int Led_03 = 12;
```

-->Dans la partie ()setup :

```
// Initialise les variables
Wr_ACC = 0
op = 0;
Wr_RB = 0;
Adr[0] = 0;
Adr[1] = 0;
sel = 0;
Clk = 0;
```

```
// Initialise the output variables as outputs
pinMode(LED_03, OUTPUT);
pinMode(LED_02, OUTPUT);
pinMode(LED_01, OUTPUT);
pinMode(LED_00, OUTPUT);
pinMode(LED_C, OUTPUT);

// Set outputs to LOW
digitalWrite(LED_03, LOW);
digitalWrite(LED_02, LOW);
digitalWrite(LED_01, LOW);
digitalWrite(LED_00, LOW);
digitalWrite(LED_C, LOW);
```

```
// Initialise the variables Q for the registers and accumulator

  for (int i = 0; i < 4; ++i)
    {
      Reg0.Q[i]=0;
      Reg0.O[i]=0;

      Reg0.Q[i]=0;
      Reg0.O[i]=0;

      Reg0.Q[i]=0;
      Reg0.O[i]=0;

      Reg0.Q[i]=0;
      Reg0.O[i]=0;

      Reg0.Q[i]=0;
      Reg0.O[i]=0;

      ACC.Q[i]=0;
      ACC.0[i]=0;
    }

//Initialise the accumulator output
  I[i]=0;
}
```

--> Dans la partie ()loop:

```
//LED outputs
digitalWrite(LED_O3, O[3]);
digitalWrite(LED_O2, O[2]);
digitalWrite(LED_O1, O[1]);
digitalWrite(LED_O0, O[0]);
digitalWrite(LED_C, (C&!op) | (Bo&op));
```

--->Dans “circuits.h”

```
//////////////////// MUX21 ////////////////////////
    Output4 MUX21;
    MUX_21.0[0] = (sel & i1[0]) | (Isel & io[e]);
    MUX_21.0[1] = (sel & i1[1]) | (!sel & i0[1]);
    MUX_21.0[2] = (sel & i1[2]) | (!sel & i0[2]);
    MUX_21.0[3] = (sel & i1[3]) | (Isel & i0[3]);
    return MUX_21;
```

```
//////////////////// MUX41 ////////////////////////////////////////
    Output4 MUX_41;
    MUX_41.0[0] = (i0[0] & !s[1] & !s[0]) | (i1[0] & !s[1] & s[0]) | (i2[0] & s[1] & !s[0]) | (i3[0] & s
    MUX_41.0[1] = (i0[1] & !s[1] & !s[0]) | (i1[1] & !s[1] & s[0]) | (i2[1] & s[1] & !s[0]) | (i3[1] & s
    MUX_41.0[2] = (i0[2] & !s[1] & !s[0]) | (i1[2] & !s[1] & s[0]) | (i2[2] & s[1] & !s[0]) | (i3[2] & s
    MUX_41.0[3] = (i0[3] & !s[1] & !s[0]) | (i1[3] & !s[1] & s[0]) | (i2[3] & s[1] & !s[0]) | (i3[3] & s
    return MUX_41;
```

```
//////////////////////// Full Adder ////////////////////////
bool Full_Adder(bool A, bool B, bool &Carry)
{
    bool Sum = A ^ B ^ Carry;
    Carry = (A & B) | (Carry & (A | B));
    return Sum;
}


//////////////////////// Full Subtractor ////////////////////
bool Full_Sub(bool A, bool B, bool &Bo)
{
    bool Diff = A ^ B ^ Bo;
    Bo = !A&Bo | (!A&B) | B&Bo;
    return Diff;
}
```

```
//////////////////////  ADD4  ////////////////////////////////
struct ADD_SUB Add_4(bool A[], bool B[])
{
    bool Carry = 0;
    ADD_SUB ADD4;
    ADD4.S[0] = Full_Adder (A[0], B[0], Carry);
    ADD4.S[1] = Full_Adder (A[1], B[1], Carry);
    ADD4.S[2] = Full_Adder (A[2], B[2], Carry);
    ADD4.S[3] = Full_Adder (A[3], B[3], Carry);
    ADD4.C = Carry;
    return ADD4;
}
```

```
//////////////////////  SUB4  ////////////////////////////////
struct ADD_SUB Sub_4(bool A[], bool B[])
{
    bool Borrow = 0;
    ADD_SUB SUB4;
    SUB4.D[0] = Full_Subtractor(A[0], B[0], Borrow);
    SUB4.D[1] = Full_Subtractor(A[1], B[1], Borrow);
    SUB4.D[2] = Full_Subtractor(A[2], B[2], Borrow);
    SUB4.D[3] = Full_Subtractor(A[3], B[3], Borrow);
    SUB4.Bo = Borrow;
    return SUB4;
}
```

```
//////////////////////  2:4 Decoder ////////////////////////////
struct Output4 DECOD24(bool Adr[])
{
    Output4 dec;
    dec.0[0] = (!Adr[1] & !Adr[0]);
    dec.0[1] = (!Adr[1] & !Adr[0]);
    dec.0[2] = (!Adr[1] & !Adr[0]);
    dec.0[3] = (!Adr[1] & !Adr[0]);
    return dec;
}
```

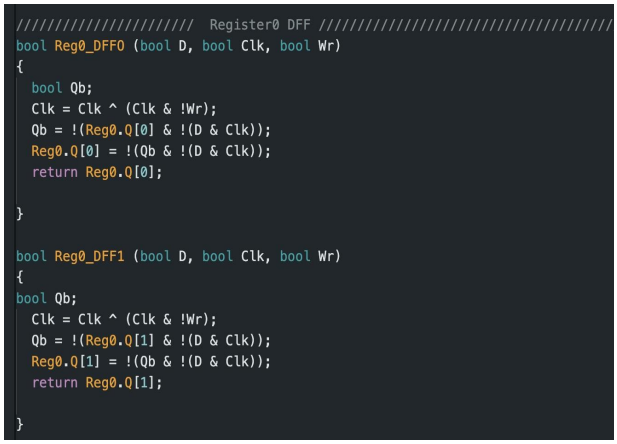
```
//////////////////////  Register bank ////////////////////////////////
struct Output4 Register_bank(bool ACC[], bool Adr[], bool clk, bool Wr)
{
    Output4 RBOut;
    Decoder_2to4(Adr, RBOut.O);
    if (RBOut.O[0]) Reg0 = Register(ACC, clk, Wr);
    if (RBOut.O[1]) Reg1 = Register(ACC, clk, Wr);
    if (RBOut.O[2]) Reg2 = Register(ACC, clk, Wr);
    if (RBOut.O[3]) Reg3 = Register(ACC, clk, Wr);

    for (int i = 0; i < 4; i++) {
        RBOut.O[i] = (Adr[1] ? (Adr[0] ? Reg3.O[i] : Reg2.O[i])
                             : (Adr[0] ? Reg1.O[i] : Reg0.O[i]));
    }
    return RBOut;
}
```

```
//////////////////////  Accumulator ////////////////////////
struct Reg Accumulator(bool I[], bool Clk, bool Wr)
{
    ACC.0[0] = ACC_DFF0(I[0], Clk, Wr);
    ACC.0[1] = ACC_DFF0(I[1], Clk, Wr);
    ACC.0[2] = ACC_DFF0(I[2], Clk, Wr);
    ACC.0[3] = ACC_DFF0(I[3], Clk, Wr);
    return ACC;
}
```

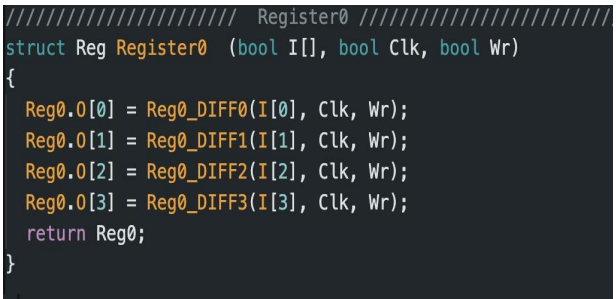
```
//////////////////////  Accumulator DFF0 ///////////
bool ACC_DFF0 (bool D, bool Clk, bool WR)
{
    bool Qb;
    Qb = !(ACC.Q[0] & !(D & Clk));
    ACC.Q[0] = !(Qb & !(D & Clk));
    return ACC.Q[0];

}
```

```
//////////////////////  Accumulator DFF1 ///////////////
bool ACC_DFF0 (bool D, bool Clk, bool WR)
{
    bool Qb;
    Qb = !(ACC.Q[1] & !(D & Clk));
    ACC.Q[1] = !(Qb & !(D & Clk));
    return ACC.Q[1];

}
```

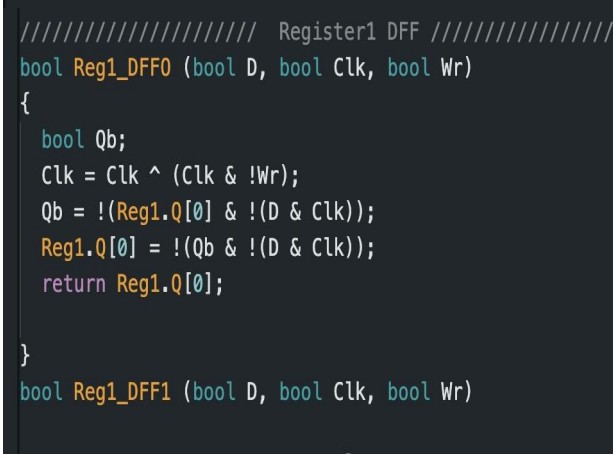
```
//////////////////////  Register1 //////////////////////////
struct Reg Register1  (bool I[], bool Clk, bool Wr)
{
  Reg1.O[0] = Reg0_DIFF0(I[0], Clk, Wr);
  Reg1.O[1] = Reg0_DIFF1(I[1], Clk, Wr);
  Reg1.O[2] = Reg0_DIFF2(I[2], Clk, Wr);
  Reg1.O[3] = Reg0_DIFF3(I[3], Clk, Wr);
  return Reg1;
}
```



```
//////////////////////  Register0 DFF ////////////////////////////////
bool Reg0_DFF0 (bool D, bool Clk, bool Wr)
{
  bool Qb;
  Clk = Clk ^ (Clk & !Wr);
  Qb = !(Reg0.Q[0] & !(D & Clk));
  Reg0.Q[0] = !(Qb & !(D & Clk));
  return Reg0.Q[0];

}

bool Reg0_DFF1 (bool D, bool Clk, bool Wr)
{
bool Qb;
  Clk = Clk ^ (Clk & !Wr);
  Qb = !(Reg0.Q[1] & !(D & Clk));
  Reg0.Q[1] = !(Qb & !(D & Clk));
  return Reg0.Q[1];

}
```



```
//////////////////////  Register0 //////////////////////////
struct Reg Register0  (bool I[], bool Clk, bool Wr)
{
  Reg0.O[0] = Reg0_DIFF0(I[0], Clk, Wr);
  Reg0.O[1] = Reg0_DIFF1(I[1], Clk, Wr);
  Reg0.O[2] = Reg0_DIFF2(I[2], Clk, Wr);
  Reg0.O[3] = Reg0_DIFF3(I[3], Clk, Wr);
  return Reg0;
}
```

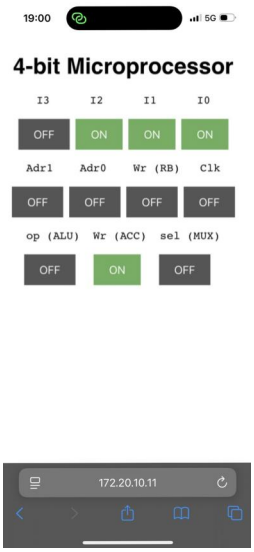


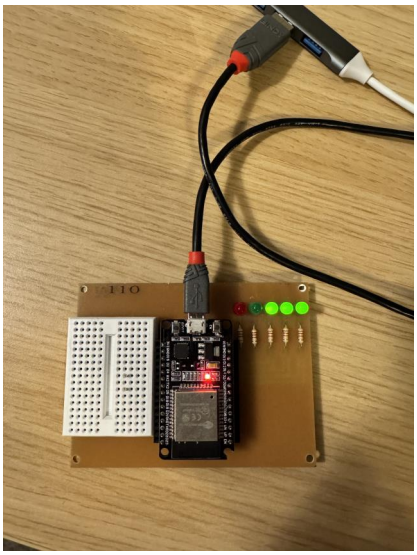
```
//////////////////////  Register1 DFF /////////////////
bool Reg1_DFF0 (bool D, bool Clk, bool Wr)
{
  bool Qb;
  Clk = Clk ^ (Clk & !Wr);
  Qb = !(Reg1.Q[0] & !(D & Clk));
  Reg1.Q[0] = !(Qb & !(D & Clk));
  return Reg1.Q[0];

}
bool Reg1_DFF1 (bool D, bool Clk, bool Wr)
```

Lorsqu'on se connecte avec notre téléphone, on obtient l'adresse IP d'une page internet contenant les instructions à suivre. Un paramètre réglé sur "off" correspond à un bit à 0, tandis qu'un paramètre sur "on" correspond à un bit à 1.

Nous avons envie de faire l’opération “7+3-6" :

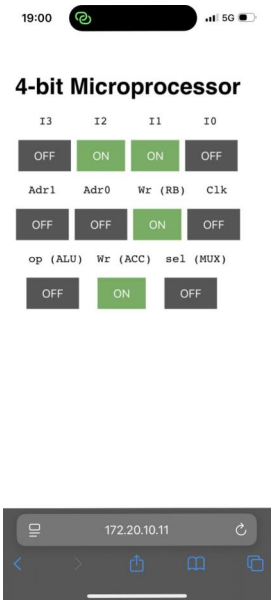
1. Il faut mettre le nombre 7 dans l’ACC:

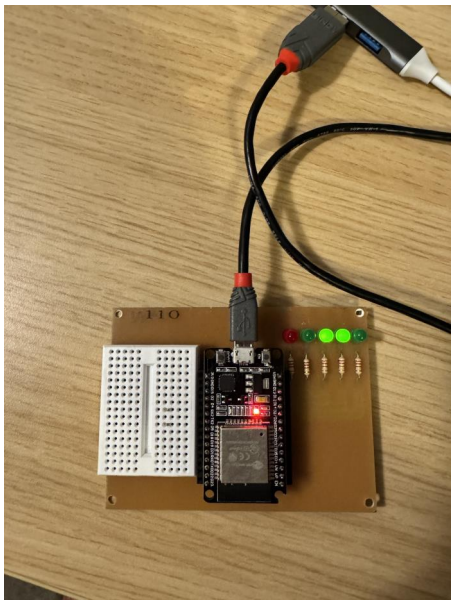




2.Il faut stocker le contenu de l'ACC dans le RB à l'Adr 00

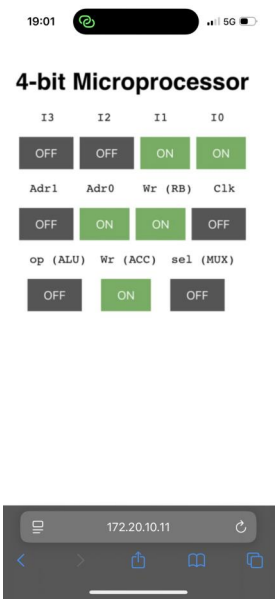
3.Il faut mettre 6 dans l'ACC

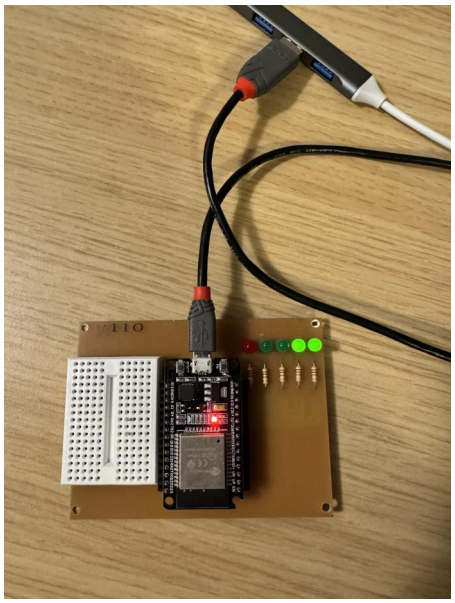




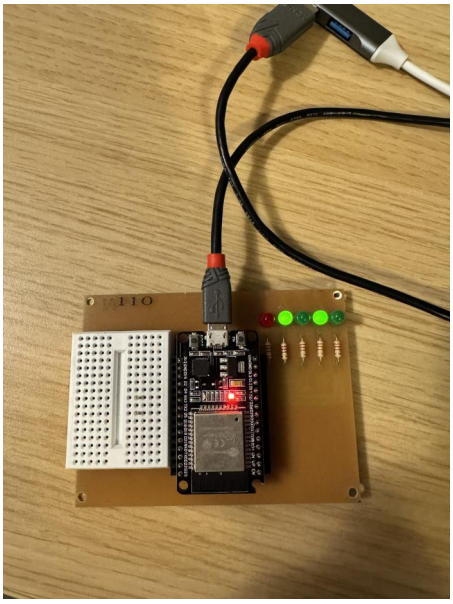
4.Il faut maintenant stocker le contenu de l'ACC dans le RB à l'Adr 01

5.Il faut mettre 3 dans ACC

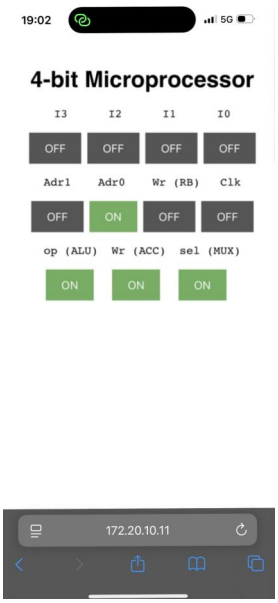


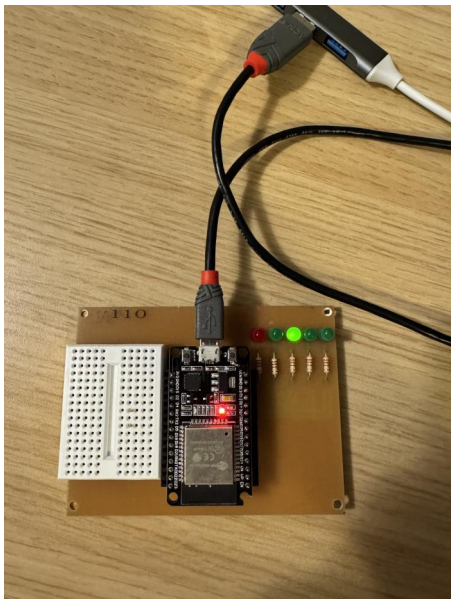


6.Il faut faire l'addition du contenu de l'ACC et du RB à l'Adr 00



7.Il faut faire la soustraction du contenu de l'ACC et du RB à l'Adr 01





Conclusion

En conclusion, nous constatons que chaque calcul s'inscrit correctement dans l'accumulateur, ce qui déclenche l'allumage des LED correspondantes. La validation du bon fonctionnement repose sur la correspondance entre le résultat attendu et l'état des LED, chaque LED représentant une puissance de 2. Par exemple, pour le calcul 7+3−6, le résultat 4 est confirmé par l'allumage de la 3ᵉ LED, qui correspond à 2x2=4. Cela prouve que le code a été exécuté avec précision et que les résultats s'affichent de manière fiable, attestant ainsi de la validité de la logique mise en œuvre

Annexe