IUT Villetaneuse
Université Sorbonne Paris Nord

SAE 13 – Microprocesseur 4 bits avec signaux de contrôle

R106-Architecture of computer systems/ BUT RT1 Grp 4

Réaliser par :
Rafael Maia et Riad Sadki

Table des Matières

Introduction

Dans le cadre de la SAE 13, l'objectif est de concevoir et de programmer un microprocesseur 4 bits capable d'exécuter des opérations arithmétiques et de manipuler des registres à l'aide de signaux de contrôle. Ce microprocesseur est contrôlé via une interface web, permettant une interaction en temps réel avec les signaux d'entrée et les résultats des opérations.

Les microprocesseurs 4 bits, bien qu'élémentaires, jouent un rôle essentiel dans la compréhension des architectures informatiques. Leur simplicité permet d'explorer en profondeur les interactions entre les registres, l'accumulateur, l'ALU et les signaux de contrôle. Ce projet met en pratique ces concepts fondamentaux tout en les adaptant à une application moderne via l'intégration d'une interface web.

Le projet implique l'utilisation d'une carte ESP32, connectée à un réseau Wi-Fi, et la programmation d'un code en C++ pour gérer les signaux de contrôle, les registres, l'accumulateur et l'ALU (Unité Arithmétique

et Logique). Ce rapport détaille les étapes de réalisation, les résultats obtenus, et inclut une analyse des performances.

Structure du Microprocesseur

Le microprocesseur est composé des éléments suivants :

1. **Banc de registres (Register Bank)** : Stocke temporairement les données.
2. **ALU 4 bits (ALU4)** : Effectue les opérations arithmétiques (addition, soustraction).
3. **Accumulateur (ACC)** : Stocke les résultats des opérations.
4. **Multiplexeur 2 vers 1 (MUX21)** : Sélectionne l'une des deux entrées en fonction du signal de contrôle.

Fonctionnement

Les signaux de contrôle gèrent les interactions entre ces composants. Par exemple :

- Wr_ACC active l'écriture dans l'accumulateur.
- Wr_RB active l'écriture dans le banc de registres.
- Clk déclenche les opérations à chaque cycle.

Programmation et Implémentation

1. Initialisation des Signaux et Composants

Le code initialise les signaux de contrôle et les variables nécessaires pour assurer un état stable au démarrage. Voici les parties du code correspondantes :

Code : Initialisation des variables

```
void setup()
{
  // Initialise the variables
  Wr_ACC = 0;
  I[0] = 0;
  I[1] = 0;
  I[2] = 0;
  I[3] = 0;
  op = 0;
  Wr_RB = 0;
  Adr[0] = 0;
  Adr[1] = 0;
  sel = 0;
  Clk = 0;
  // Initialise the variables Q for the registers and accumulator
  Reg0.Q[0] = 0;
  Reg0.Q[1] = 0;
  Reg0.Q[2] = 0;
  Reg0.Q[3] = 0;

  Reg0.O[0] = 0;
  Reg0.O[1] = 0;
  Reg0.O[2] = 0;
  Reg0.O[3] = 0;

  Reg1.Q[0] = 0;
  Reg1.Q[1] = 0;
  Reg1.Q[2] = 0;
  Reg1.Q[3] = 0;
```

microprocesseur circuits.h

```
Reg1.O[0] = 0;
Reg1.O[1] = 0;
Reg1.O[2] = 0;
Reg1.O[3] = 0;

Reg2.Q[0] = 0;
Reg2.Q[1] = 0;
Reg2.Q[2] = 0;
Reg2.Q[3] = 0;

Reg2.O[0] = 0;
Reg2.O[1] = 0;
Reg2.O[2] = 0;
Reg2.O[3] = 0;

Reg3.Q[0] = 0;
Reg3.Q[1] = 0;
Reg3.Q[2] = 0;
Reg3.Q[3] = 0;

Reg3.O[0] = 0;
Reg3.O[1] = 0;
Reg3.O[2] = 0;
Reg3.O[3] = 0;

// Initialise the accumulator output
ACC.Q[0] = 0;
ACC.Q[1] = 0;
ACC.Q[2] = 0;
ACC.Q[3] = 0;
ACC.O[0] = 0;
ACC.O[1] = 0;
ACC.O[2] = 0;
ACC.O[3] = 0;



// Initialize the LED variables as outputs
pinMode(LED_O0, OUTPUT);
pinMode(LED_O1, OUTPUT);
pinMode(LED_O2, OUTPUT);
pinMode(LED_O3, OUTPUT);
```

Les LEDs sont connectées aux broches GPIO 26, 27, 14 et 12, représentant les bits de sortie de l'accumulateur.

2. Connexion Wi-Fi

La carte ESP32 se connecte au réseau Wi-Fi pour permettre le contrôle via une interface web. L'adresse IP est affichée sur le moniteur série.

Code : Connexion Wi-Fi

```
  Serial.begin(115200);

  // Connect to Wi-Fi
  WiFi.begin(ssid, password);
  Serial.print("Connecting to ");
  Serial.println(ssid);
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.print("WiFi connected to ");
  Serial.println(ssid);
  // Print ESP32 Local IP Address
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());

  server.begin();
}
```

3. Interface Web

L'interface web permet de contrôler les signaux de contrôle du microprocesseur 4 bits en temps réel. Une page HTML est générée par le serveur ESP32, qui interprète les requêtes HTTP envoyées par le navigateur. Chaque requête permet d'activer ou de désactiver un signal, comme les entrées du microprocesseur, les signaux de contrôle des registres, ou les signaux d'opérations de l'ALU.

1. Réponse HTTP initiale

Le serveur ESP32 commence par répondre à chaque requête HTTP avec un en-tête confirmant le succès de la connexion. Cet en-tête informe le navigateur que le contenu envoyé sera de type HTML.

```
// and a content-type so the client knows what's coming, then a blank line:
client.println("HTTP/1.1 200 OK");
client.println("Content-type:text/html");
client.println("Connection: close");
client.println();
```

Cet en-tête est indispensable pour établir la communication entre le serveur et le navigateur web.

2. Traitement des commandes d'entrée

Une fois la réponse initiale envoyée, le serveur analyse les commandes reçues. Selon la requête HTTP, il active ou désactive les signaux d'entrée du microprocesseur. Voici un exemple de gestion des signaux I[3] à I[0] :

```
// Micro-processor Inputs
if (header.indexOf("GET /I[3]/on") >= 0)
{
  I[3] = 1;
} else if (header.indexOf("GET /I[3]/off") >= 0)
{
  I[3] = 0;
} else if (header.indexOf("GET /I[2]/on") >= 0)
{
  I[2] = 1;
} else if (header.indexOf("GET /I[2]/off") >= 0)
{
  I[2] = 0;
} else if (header.indexOf("GET /I[1]/on") >= 0)
{
  I[1] = 1;
} else if (header.indexOf("GET /I[1]/off") >= 0)
{
  I[1] = 0;
} else if (header.indexOf("GET /I[0]/on") >= 0)
{
  I[0] = 1;
} else if (header.indexOf("GET /I[0]/off") >= 0)
{
  I[0] = 0;
}
```

Ces commandes modifient les états des entrées du microprocesseur selon les interactions de l'utilisateur sur la page web.

Afin d'obtenir cette interface, on a tapé l'adresse IP donné par Arduino dans notre navigateur web.

Cette capture illustre les interactions possibles entre l'utilisateur et le microprocesseur via le navigateur.

Une fois la réponse initiale envoyée, le serveur analyse les commandes reçues via HTTP. Selon la requête reçue, il modifie les états des signaux d'entrée du microprocesseur.

3. Commandes pour le banc de registres et l'horloge

Le banc de registres nécessite plusieurs signaux pour sélectionner une adresse, écrire des données, et synchroniser les opérations. Voici le traitement de ces commandes :

```
// Address inputs (Register bank), Write input (Register bank), Clock input
if (header.indexOf("GET /Adr1/on") >= 0)
{
  Adr[1] = 1;
} else if (header.indexOf("GET /Adr1/off") >= 0)
{
  Adr[1] = 0;
} else if (header.indexOf("GET /Adr0/on") >= 0)
{
  Adr[0] = 1;
} else if (header.indexOf("GET /Adr0/off") >= 0)
{
  Adr[0] = 0;
} else if (header.indexOf("GET /Wr_RB/on") >= 0)
{
  Wr_RB = 1;
} else if (header.indexOf("GET /Wr_RB/off") >= 0)
{
  Wr_RB = 0;
} else if (header.indexOf("GET /Clk/on") >= 0)
{
  Clk = 1;
} else if (header.indexOf("GET /Clk/off") >= 0)
{
  Clk = 0;
}
```

Ces commandes permettent à l'utilisateur de contrôler les registres et l'horloge via l'interface web.

4. Commandes pour l'ALU, l'accumulateur et le multiplexeur

Les signaux de contrôle op (opération ALU), Wr_ACC (écriture dans l'accumulateur), et sel (sélection du multiplexeur) sont également gérés par des requêtes HTTP :

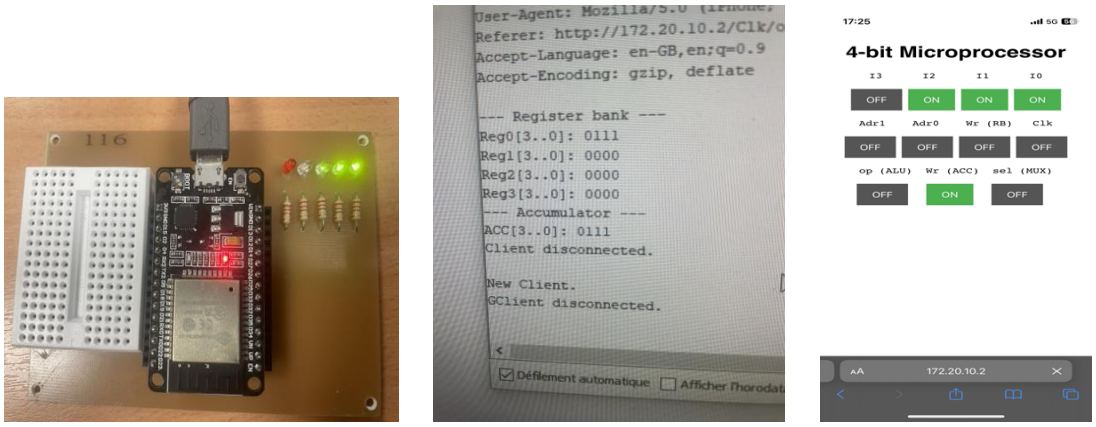
```
// Operation input (ALU), Write input (Accumulator), Selection input (MUX21)
if (header.indexOf("GET /op/on") >= 0)
{
  op = 1;
} else if (header.indexOf("GET /op/off") >= 0)
{
  op = 0;
} else if (header.indexOf("GET /Wr_ACC/on") >= 0)
{
  Wr_ACC = 1;
} else if (header.indexOf("GET /Wr_ACC/off") >= 0)
{
  Wr_ACC = 0;
} else if (header.indexOf("GET /sel/on") >= 0)
{
  sel = 1;
} else if (header.indexOf("GET /sel/off") >= 0)
{
  sel = 0;
}
```

Ces signaux contrôlent les opérations de l'ALU, la gestion des données dans l'accumulateur, et la sélection des entrées via le multiplexeur.

4.Résultats et Tests

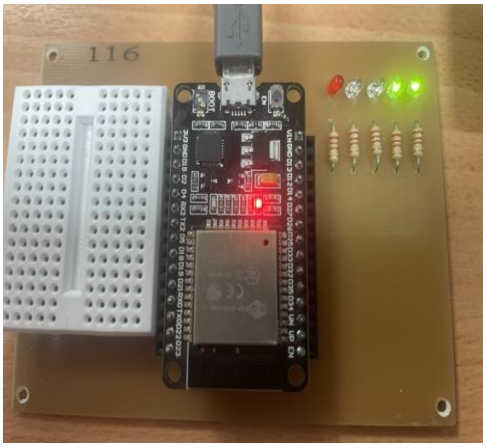
1. Test de l'Opération : 7 + 3 - 6

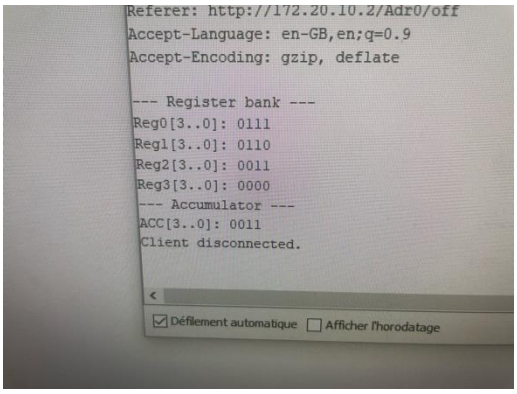
Étape 1 : Stocker 7 dans l'ACC et Copier 7 dans le registre à l'adresse 00



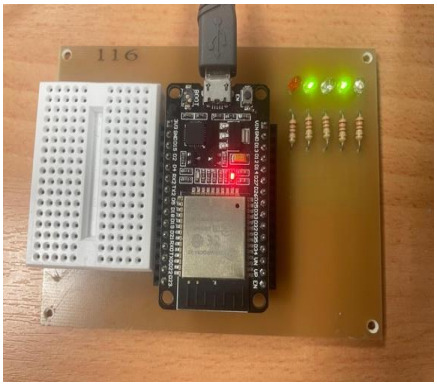
Étape 2 : Stocker 6 dans l'ACC et Copier 6 dans le registre à l'adresse 00

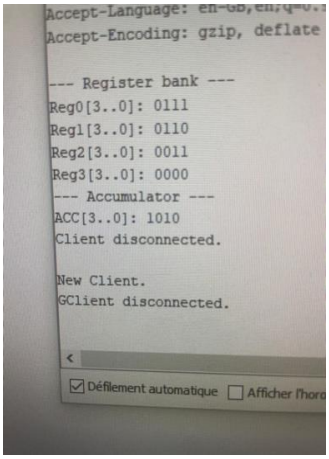
Étape 3 : Stocker 3 dans l'ACC et Copier 3 dans le registre à l'adresse 00



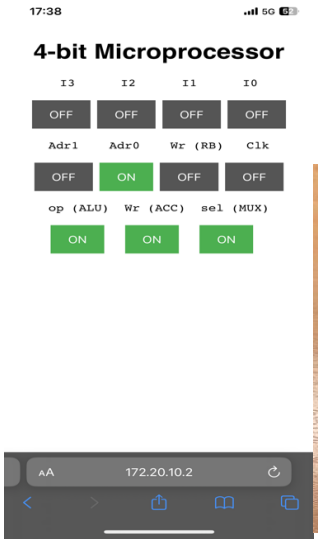


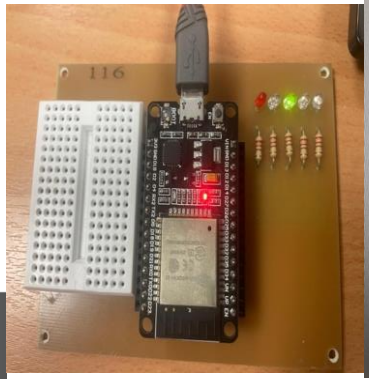
Étape 4 : Addition de 7 et 3

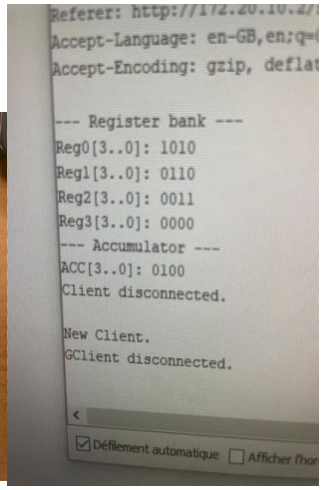




Étape 5 : Soustraction de 10 et 6







Conclusion

L Le microprocesseur 4 bits a été conçu et testé avec succès. Les résultats montrent que les signaux de contrôle et les composants fonctionnent comme prévu. L'opération complexe 7 + 3 - 6 a été réalisée correctement, confirmant la fiabilité du programme et de l'interface.

Perspectives et Améliorations Futures :

Bien que le microprocesseur fonctionne comme prévu, plusieurs améliorations peuvent être envisagées pour étendre ses capacités. Par exemple, augmenter la largeur de données à 8 bits permettrait d'exécuter des calculs plus complexes et de manipuler des données plus volumineuses. De plus, l'intégration d'une interface graphique interactive améliorerait l'expérience utilisateur en rendant le contrôle des signaux plus intuitif. Enfin, l'ajout de nouvelles instructions, comme la multiplication ou la division, pourrait enrichir les fonctionnalités du processeur.

Annexes

Extrait de Code Essentiel

Les extraits de code ci-dessous sont issus du fichier circuits.h, qui contient les définitions des composants essentiels du microprocesseur, tels que l'ALU, le multiplexeur, et l'accumulateur. Ces fonctions ont été utilisées pour implémenter les fonctionnalités décrites dans ce rapport.

1. Multiplexeur 2:1 (MUX21)

Le multiplexeur 2:1 est un composant clé utilisé pour sélectionner entre deux ensembles de données en fonction d'un signal de contrôle (sel). Voici l'implémentation utilisée dans le microprocesseur :

```
//////////////////////  MUX21  ////////////////////////////////////////////////
struct Output4 MUX21(bool sel, bool i0[], bool i1[])
{
    Output4 MUX_21;
    MUX_21.O[0] = (!sel & i0[0]) | (sel & i1[0]);
    MUX_21.O[1] = (!sel & i0[1]) | (sel & i1[1]);
    MUX_21.O[2] = (!sel & i0[2]) | (sel & i1[2]);
    MUX_21.O[3] = (!sel & i0[3]) | (sel & i1[3]);
    return MUX_21;
}
```

2. Unité Arithmétique et Logique (ALU)

L'ALU (Unité Arithmétique et Logique) gère les opérations de calcul (addition et soustraction). Elle utilise le multiplexeur MUX21 pour choisir entre les résultats de l'addition ou de la soustraction en fonction du signal op.

```
//////////////////////  ALU4  ////////////////////////////////////////////
struct Output4 ALU_4(bool A[], bool B[], bool op)
{
    bool C, Bo;
    ADD_SUB ADD4, SUB4;
    Output4 ALU4;
    SUB4 = Sub_4(A, B);
    ADD4 = Add_4(A, B);
    ALU4 = MUX21(op, ADD4.S, SUB4.D);
    return ALU4;
}
```

3. Accumulateur

L'accumulateur est une mémoire temporaire essentielle pour stocker les résultats intermédiaires des calculs. Il est mis à jour en fonction des signaux de contrôle Clk et Wr.

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

4. Banc de Registres

Le banc de registres stocke les données de manière temporaire et permet leur accès en fonction des signaux d'adresse (Adr) et d'écriture (Wr). La sortie est ensuite sélectionnée via le multiplexeur MUX41.

```
//////////////////////  Register bank  ////////////////////////////////////////////

struct Output4 Register_bank(bool I[], bool Adr[], bool Clk, bool Wr)
{
    bool Wr_reg[4];
    Output4 DEC24;

    DEC24 = DECOD24(Adr);

    Wr_reg[0] = DEC24.O[0] & Wr;                          // Operation between DEC24.O[0] and Wr
    Wr_reg[1] = DEC24.O[1] & Wr;
    Wr_reg[2] = DEC24.O[2] & Wr;
    Wr_reg[3] = DEC24.O[3] & Wr;

    Reg0 = Register0(I, Clk, Wr_reg[0]);
    Reg1 = Register1(I, Clk, Wr_reg[1]);
    Reg2 = Register2(I, Clk, Wr_reg[2]);
    Reg3 = Register3(I, Clk, Wr_reg[3]);

    RB = MUX41(Adr, Reg0.O, Reg1.O, Reg2.O, Reg3.O);

    return RB;
}
```

5. Fonction Addition (ADD4)

Voici une implémentation bit à bit de l'addition avec report, utilisée dans l'ALU pour effectuer des calculs binaires.

```
//////////////////////  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);    // A compléter
    ADD4.S[1] = Full_Adder(A[1], B[1], Carry);    // A compléter
    ADD4.S[2] = Full_Adder(A[2], B[2], Carry);    // A compléter
    ADD4.S[3] = Full_Adder(A[3], B[3], Carry);    // A compléter
    ADD4.C = Carry;
    return ADD4;
}
```