ChatGPT

Cours d'architecture informatique : module LC-3

Objectifs pédagogiques :
- Comprendre le modèle de Von Neumann et ses composants principaux (PC, IR, ALU, mémoire, unité de contrôle).
- Distinguer l'**état architectural** (visible par le programmeur) de l'**état microarchitectural** (interne) [1] [2].
- Décrire le cycle d'instruction (fetch – decode – execute) et ses phases en pseudocode [3] [4].
- Expliquer la structure du *datapath* LC-3 (bus global, registres, ALU, MAR/MDR) et le rôle du contrôleur (FSM ou microcode) [5] [6].
- Illustrer l'exécution d'instructions LC-3 typiques (ADD, LD, BR, etc.) par des algorithmes (pseudocode) intégrés.

1. Le modèle de Von Neumann

Objectif : Définir les caractéristiques de l'architecture de Von Neumann.

Dans l'architecture de Von Neumann, les **instructions** et les **données** du programme sont stockées dans une même mémoire unifiée [7]. La CPU se compose de trois blocs principaux :
- une **unité de traitement** (Arithmetic Logic Unit, ALU + registres généraux) ;
- une **unité de contrôle** (contenant au minimum le *compteur de programme* PC et le *registre d'instruction* IR) [8] ;
- la **mémoire principale**, qui contient à la fois les instructions et les données.

Cette organisation permet un *programme stocké* (le CPU lit séquentiellement les instructions en mémoire) et une exécution *séquentielle* (une instruction à la fois) [7]. L'avantage est la simplicité de programmation : le programmeur n'a pas à gérer le flux d'instruction ni le chargement manuel du code. L'inconvénient est le **goulot d'étranglement de Von Neumann** : toutes les opérations mémoire (lecture d'instruction ou de données) passent par un même bus partagé [7]. En pratique, les processeurs modernes ajoutent des caches et du parallélisme pour atténuer ce problème, mais l'idée fondamentale demeure.

2. État architectural vs. état microarchitectural

Objectif : Cerner ce que le programmeur peut voir et contrôler.

L'**état architectural** d'un processeur est l'ensemble des éléments définis par l'ISA et directement accessibles au programmeur [1]. Cela inclut notamment :
- la **mémoire principale** (où sont stockés code et données) ;
- les registres architecturaux (registre général R0–R7, *PC*, registres de drapeaux NZP, etc.) ;
- d'autres registres de contrôle (drapeaux de conditions, etc.) [1].

Le programmeur peut lire/écrire ces registres via les instructions de l'ISA. Par exemple, le texte de l'ISA LC-3 définit que le programme peut charger (LD) ou stocker (ST) des valeurs en mémoire, déplacer des données entre registres, modifier le PC via un saut, etc.

À l'inverse, l'**état microarchitectural** regroupe les éléments internes au CPU qui ne sont pas visibles directement par l'ISA [2]. Exemples : registres intermédiaires (buffers, pipeline registers), balises de cache, état du prédicteur de branches, signaux internes, etc. Ces éléments évoluent au cours de l'exécution d'une instruction, mais le programmeur ne peut pas les manipuler directement. L'existence de cet état caché permet des optimisations internes (pipeline, caches, microinstructions) tout en restant compatible au niveau architectural.

3. Le cycle d'instruction (Fetch – Decode – Execute)

Objectifs :
- Décrire les phases *fetch*, *decode*, *execute* d'une instruction.
- Montrer comment le PC, MAR, MDR et IR interagissent lors du fetch.
- Donner le pseudocode correspondant au fetch.

Le CPU suit en permanence un cycle d'instruction en trois étapes :
- **Fetch (lecture)** : on lit en mémoire l'instruction pointée par le PC. On copie PC dans le registre d'adresse *MAR*, on incrémente PC, on déclenche une lecture mémoire, et on dépose la valeur lue dans *IR* [4] [9].
- **Decode (décodage)** : le contenu de *IR* (opcode + opérandes) est interprété par l'unité de contrôle. Cette dernière détermine l'opération à effectuer et calcule éventuellement les adresses effectives (par exemple un offset PC-relatif ou registre+offset). Les signaux de commande (gates, registres *LD.X*, mode ALU…) sont préparés [3].
- **Execute (exécution)** : l'opération est réalisée – par exemple l'ALU effectue un calcul, la mémoire est lue/écrite, ou le PC est modifié en cas de saut. On met alors à jour les registres cibles et les drapeaux de condition NZP selon le résultat [10].

Ces étapes sont illustrées dans le diagramme suivant (figure générique d'un pipeline 3 étapes, avec PC, MAR/MDR, IR, ALU, etc.). À chaque cycle d'horloge complet, on passe d'une étape à la suivante. Dans un design LC-3 simple, chaque phase dure exactement un cycle.

L'algorithme suivant décrit la phase **fetch** d'une instruction (lecture en mémoire) :

```
Procédure Lecture_Instruction():
    MAR ← PC            // Charger l'adresse de l'instruction dans MAR
    PC  ← PC + 1        // Incrémenter le compteur (point vers la prochaine
instruction)
    MDR ← Mémoire[MAR] // Lire le mot mémoire à l'adresse MAR
    IR  ← MDR           // Charger l'instruction dans le registre IR
Fin Procédure
```

Cette séquence garantit qu'après le fetch, *IR* contient l'instruction courante et *PC* pointe vers la suivante. Les phases *decode* et *execute* qui suivent s'appuient sur cet état pour finir d'exécuter l'instruction.

4. Architecture LC-3 (Datapath et unité de contrôle)

Objectifs :
- Présenter les spécificités de l'ISA LC-3 (taille, registres, opcode, modes d'adressage).
- Détailler les composants du datapath LC-3 (bus, registres, ALU, mémoire).
- Expliquer le rôle de l'unité de contrôle (hardwired vs microprogrammée).

Le LC-3 est un processeur pédagogique à mots de 16 bits. L'espace d'adressage est de 2^{16} mots mémoire de 16 bits [11]. Il possède huit registres généraux **R0–R7** de 16 bits chacun [12], un PC 16 bits, et 3 bits de condition (N, Z, P) qui indiquent le signe du dernier résultat (N=1 si négatif, Z si nul, P si positif) [13]. L'ISA LC-3 définit 15 opcodes (opérations) principales : opérations **ADD**, **AND**, **NOT** (arithmétiques logiques), instructions de transfert de données (*LD, LDI, LDR, LEA, ST, STR, STI*), et instructions de contrôle (*BR, JSR/JSRR, JMP, RTI, TRAP*) [14].

Le **datapath** du LC-3 repose sur un **bus global** de 16 bits reliant les composants principaux. Les différents modules (registres, ALU, PC, MDR, etc.) peuvent mettre leurs données sur ce bus, mais un seul à la fois grâce à des pilotes *tri-états* [5] [15]. Par exemple, lorsqu'on active l'écriture d'un registre particulier (*LD.REG* par le contrôleur), ce registre place sa valeur sur le bus pendant que les autres restent désactivés. Les flèches pleines dans le schéma représentent le flux de données sur le bus, et les flèches creuses indiquent les signaux de contrôle de l'unité de commande [16].

Les **registres clés** du datapath sont :
- **PC (Program Counter)** : contient l'adresse de l'instruction courante. Pendant le fetch, PC est transmis à MAR et immédiatement incrémenté [17] [4]. Le PC peut aussi être modifié en cours d'exécution (par branchement) par chargement de nouvelles données dans PC depuis le bus.
- **IR (Instruction Register)** : stocke l'instruction de 16 bits lue en mémoire. Ses champs (opcode et opérandes) commandent les opérations suivantes.
- **Registre d'Adresse Mémoire (MAR)** : reçoit les adresses pour les opérations mémoire. Durant *fetch*, on y place la valeur de PC. Pour un chargement de données (LD), on y place l'adresse effective calculée (par exemple PC+offset).
- **Registre de Données Mémoire (MDR)** : tampon 16 bits pour la donnée échangée avec la mémoire. En lecture, il reçoit le mot lu (depuis l'adresse dans MAR) ; en écriture, on y place le mot à écrire, puis on déclenche l'écriture mémoire via le signal *MEM* et *RW*.

Le **Register File** (les registres généraux R0–R7) a deux ports de lecture (SR1, SR2) et un port d'écriture (DR) [18]. À chaque instruction, on peut lire simultanément deux registres sources et écrire un registre destination. Par exemple, pour ADD on lit R[SR1] et R[SR2] (ou une valeur immédiate sign-étendue) et on écrit dans R[DR]. Chaque écriture de registre se fait via le bus global et l'activation du signal *LD.REG*. Les drapeaux NZP sont mis à jour à partir du résultat écrit (N=1 si le bit 15 est à 1, etc.) [13].

L'**ALU (Arithmetic Logic Unit)** 16 bits effectue les opérations arithmétiques et logiques (addition, AND, NOT, etc.). Elle prend **toujours** comme premier opérande A le contenu de R[SR1], et comme second opérande B soit R[SR2] soit une valeur immédiate sign-étendue extraite de l'IR, selon le type d'instruction (ex. bit 'mode' dans ADD/AND) [19]. Le résultat produit par l'ALU est remis sur le bus via un pilote *gateALU* lorsque le contrôleur active *LD.ALU*, puis peut être stocké dans un registre cible.

Le **contrôleur** du LC-3 est une FSM (machine à états finis) qui pilote ces composants en fonction de l'état courant du cycle d'instruction. À chaque état de la FSM, un ensemble de signaux de contrôle est généré (p. ex. quels registres chargés, quelle opération ALU, lecture/écriture mémoire). Cette FSM passe successivement par des états « Fetch1, Fetch2, …, Decode, EvalAddr, FetchOperands, Execute, Store » pour chaque instruction. En pratique, on peut implémenter cette FSM soit « hardwired » (logique combinatoire fixe) soit via un microprogramme (ROM de micro-instructions) [6]. Le LC-3 académique utilise typiquement une implémentation hardwired simple (logique câblée) pour expliciter les signaux aux étudiants.

5. Exemples d'instructions LC-3 et pseudocode

5.1 Instructions arithmétiques : ADD, AND, NOT

- **ADD** (addition) : somme arithmétique. Deux formats existent, selon le bit d'extension : registre ou immédiat. Le pseudocode est :

```
Procédure ADD:
    si IR[5] = 0 alors  // format registre
        R[DR] ← R[SR1] + R[SR2]
    sinon               // format immédiat
        imm5 ← SEXT(IR[4..0], 16)  // étendre 5 bits signé à 16 bits
        R[DR] ← R[SR1] + imm5
    Fin si
    // Mise à jour des drapeaux de condition
    N ← (R[DR] < 0),  Z ← (R[DR] = 0),  P ← (R[DR] > 0)
Fin Procédure
```

  Ainsi, si le bit 5 de l'instruction vaut 1, on prend un operande immédiat sign-étendu, sinon un second registre. Les signaux de contrôle (par ex. *ALUK* pour addition, *LD.REG*) sont activés par le contrôleur pendant cette phase. L'instruction **AND** (ET bit-à-bit) se traduit de la même façon (remplacer «+» par «∧» dans le pseudocode). L'instruction **NOT** n'a qu'un seul opérande source SR, et exécute « `R[DR] ← bitwise_not(R[SR])` » puis met à jour NZP.

5.2 Accès mémoire : LD, ST, etc.

- **LD** (load) : charge en DR la donnée mémoire à l'adresse `PC + offset9`. Le pseudocode :

```
Procédure LD:
    adr ← PC + SEXT(IR[8..0], 16)  // calcul de l'adresse cible
    MAR ← adr;                     // placer dans MAR
    MEM ← 1 (lecture activée)
    MDR ← Mémoire[MAR];            // lecture mémoire
    R[DR] ← MDR;                   // charger dans le registre
destination
    MEP ← 0 (lecture désactivée)
Fin Procédure
```

- **ST** (store) : inverse de LD. On calcule aussi `adr ← PC + SEXT(offset9)`, on met **LD.MDR**=1 pour charger R[SR] dans MDR, puis **MEM=1 (écriture)** pour stocker MDR en mémoire. Les signaux *LD.REG*, *LD.MDR*, *MEM*, *RW* sont activés dans les états FSM correspondants.
  Le LC-3 propose plusieurs modes d'adressage (immédiat, PC-relatif, indirect, base+offset) [20] . Par exemple, LDI/LST utilisent l'indirection (2 accès mémoire successifs). Le contrôleur gère la séquence complète d'accès mémoire (calcul d'adresse, MAR/MDR, activation de *MEM*).

5.3 Instructions de contrôle : BR, JMP, JSR, TRAP

- **BR** (Branch) : branchement conditionnel PC-relatif. Les bits [11..9] spécifient la condition NZP à tester. Par exemple, **BRz** teste *Z=1*. En pseudocode :

```
Procédure BR(offset9, NZP_tests):
    // NZP_tests contient les bits de test (p. ex. 010 pour BRz)
    if ( (N et NZP_N) ou (Z et NZP_Z) ou (P et NZP_P) ) alors
        PC ← PC + SEXT(IR[8..0], 16)
    Fin si
Fin Procédure
```

Si aucun bit de condition n'est vrai, le PC reste inchangé (il pointe déjà sur l'instruction suivante).

- **JMP BaseR** : saut inconditionnel à l'adresse contenue dans le registre *BaseR*. Pseudocode :

```
Procédure JMP(BaseR):
    PC ← R[BaseR]  // on charge le PC avec la valeur du registre BaseR
Fin Procédure
```

Cette instruction est toujours prise (c'est un branch inconditionnel) [21].

- **JSR/JSRR** : saut vers sous-programme. Elles stockent la valeur de retour (PC+1) dans R7 puis font un saut (JSR utilise un offset PC, JSRR utilise BaseR).
- **TRAP** : appel de service système. Le PC est chargé avec une adresse fixe dans la routine système (table TRAP).

6. Conclusion et perspectives

Ce cours a exposé les fondements de l'architecture de Von Neumann appliqués au processeur pédagogique LC-3. Nous avons couvert :
- les composants essentiels (PC, IR, registres, ALU, mémoire) et la différence entre état architectural et microarchitectural [1] [8] ;
- le cycle d'instruction détaillé (*fetch/decode/execute*) et son implémentation dans le datapath [4] [16] ;
- la structure du datapath LC-3 (bus global, tri-états, MAR/MDR, registre de banque, ALU) et les signaux de contrôle.

Le contenu inclut aussi le pseudocode des opérations clés (ADD, LD, BR, JMP) pour illustrer le déroulement interne de l'exécution. En poursuivant l'étude, on peut approfondir la conception du contrôleur (FSM ou microprogramme) et explorer les optimisations modernes (pipeline, parallélisme, caches).

Sources : Synthèse basée sur une transcription pédagogique d'architecture informatique complétée par des références académiques (notamment sur le modèle de Von Neumann [8], les définitions d'état architectural [1] [2], le cycle d'instruction [22] [4], et les spécifications ISA du LC-3 [11] [23]).

---

[1] [2] Architectural state - Wikipedia
https://en.wikipedia.org/wiki/Architectural_state

[3] [4] [10] [22] Instruction cycle - Wikipedia
https://en.wikipedia.org/wiki/Instruction_cycle

[5] [15] [16] [18] [19] LC3-datapath
https://cs2461-2020.github.io/lectures/Datapath.pdf

6 Lecture24(AppC)
https://www.cs.colostate.edu/~cs270/.Fall17/slides/Lecture24(AppC).pdf

7 8 Von Neumann architecture - Wikipedia
https://en.wikipedia.org/wiki/Von_Neumann_architecture

9 17 transcription.txt
file://file-PvBmwwxh9ff3cYwKSWjaFv

11 12 13 14 20 21 23 LC3-ISA - Compatibility Mode
https://cs2461-2020.github.io/lectures/lc3ISA.pdf