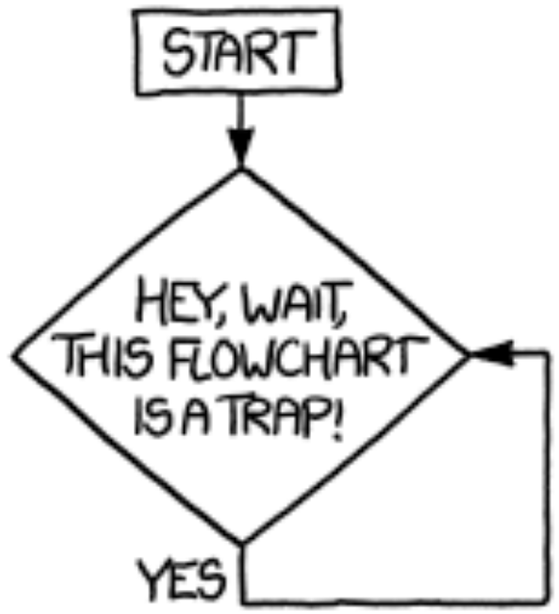
# Assembleur ARM:
# Séquence d'exécution et branchements




xkcd.com

GIF-1001 Ordinateurs: Structure et Applications, Hiver 2015

Jean-François Lalonde

Merci à Yves Roy

# Modification de la séquence d'exécution

- Par défaut, les instructions s'exécutent séquentiellement et PC est incrémenté automatiquement par le microprocesseur entre chaque instruction
  - PC = PC + 4 (taille d'une instruction)
- Dans nos programme, il arrive que l'on veuille exécuter autre chose que la prochaine instruction:
  - Saut direct à une instruction
  - Énoncé conditionnel "si"
  - Boucle: "répète N fois" ou "répète tant que"
  - Appel de fonction
- Il est possible de contrôler la séquence d'exécution, en assembleur, avec des instructions qui modifient PC.

# Sauts Absolus

- PC est un registre et peut être modifié comme les autres registres, avec la plupart des instructions
- Modifier la valeur de PC correspond à effectuer un saut absolu à une adresse.
- Exemples:

```
MOV PC, 0x80          ; PC = 0x80
MOV PC, R0            ; PC = R0
LDR PC, [R0]          ; PC = Memoire[R0]
ADD PC, R0, #0        ; PC = R0 + 0
```

# Sauts relatifs

- Dans plusieurs cas, on ne veut pas exécuter l'instruction à l'adresse X, mais exécuter l'instruction qui se trouve à N octets de l'instruction courante: on veut « déplacer » PC par rapport à sa valeur actuelle
- L'instruction B (Branch) modifie PC relativement à sa valeur actuelle:

```
B Offset
```

  - Offset est la valeur du déplacement, signé. L'adresse de la prochaine instruction est calculée comme suit
    - PC = Adresse de l'instruction B + Offset + 8*
    - Rappelez-vous: PC contient l'adresse de la prochaine instruction **+ 8**

- Comme le programmeur ne veut pas être obligé de compter l'offset, on peut remplacer « Offset » par une étiquette: l'assembleur calculera la valeur pour nous.

```
B #12                          ; Saute à l'adresse de l'instruction + 20
MonEtiquette B MonEtiquette ; Boucle infinie!
```

# Branchements conditionnels

- Un énoncé conditionnel se code habituellement avec au moins deux instructions:
  - une pour faire le test ou la comparaison
  - une pour effectuer (ou pas) le branchement en fonction de la comparaison.
- Le résultat du test ou de la comparaison est entreposé dans les drapeaux de l'ALU et l'instruction de branchement est conditionnelle. Si la condition est rencontrée, on effectue le branchement. Sinon, on exécute la ligne suivante… Pour cette raison, les conditions de branchement sont souvent inversées.
- Exemple: Si `MaVar` vaut 4, exécute une tâche:

exemple en C

```
if (MaVar == 4) {
  // exécute une tâche…
}

// le programme continue…
```

assembleur

```
LDR R0, MaVar      ; Met la valeur de MaVar dans R0
CMP R0, 4          ; Change les drapeaux comme R0-4

BNE PasEgal
; execute une tâche…

PasEgal
; le programme continue…
```

# Branchements conditionnels

- Autre exemple: if/else

exemple en C

```
if (MaVar == 4) {
  // exécute une tâche…
} else {
  // exécute une autre tâche…
}
// le programme continue…
```

assembleur

```
LDR R0, MaVar      ; Met la valeur de MaVar dans R0
CMP R0, 4          ; Change les drapeaux comme R0-4
BNE PasEgal
; execute une tâche…

B Continue
PasEgal
; exécute une autre tâche…

Continue
; le programme continue...
```

# Branchements conditionnels

- Afin d'éviter de changer les drapeaux et pour faire un énoncé conditionnel avec une seule instruction:
  - CBZ (Conditional Branch Zero)
  - CBNZ (Conditional Branch Non-Zero).

```
CBZ Rn, Offset    ; Branchement à l'Offset si Rn est égal à 0
CBNZ Rn, Offset   ; Branchement à l'Offset si Rn n'est pas égal à 0
```

# Boucles

- Une boucle (répète N fois ou tant que) est constituée de trois opérations:
  - initialiser la boucle
  - condition d'arrêt
  - opération mathématique qui mènera à la réalisation de la condition d'arrêt.
- En assembleur, le début de toutes les boucles est identifié par une étiquette qui permet de revenir au début de la boucle
- Voici un exemple de boucle qui se répète cinq fois:

exemple en C

```
for (int i = 0; i < 5; ++i) {
  // tâche à l'intérieur de la boucle
}
// le programme continue…
```

assembleur

```
InitDeBoucle
    MOV R4,#5

DebutDeBoucle
    ; tâche à l'intérieur de la boucle
    SUBS R4, R4, #1       ; R4 = R4 - 1, change les drapeaux
    BNE DebutDeBoucle     ; Condition d'arrêt

; le programme continue…
```

# Appel de Fonction et Adresse de Retour

- Appeler une fonction signifie mettre PC au début de la fonction pour exécuter les instructions constituant la fonction.
- Retourner d'une fonction signifie reprendre l'exécution d'où l'appel s'est fait. Il faut revenir à l'endroit où nous étions: c'est l'adresse de retour.
- Une fonction est un ensemble d'instructions à une adresse donnée. Cette adresse est identifiée par une étiquette.
- Mauvais exemple d'appel de fonction:

| Main | Fonction |
|---|---|
| `B MaFonction`<br>`AdresseDeRetour`<br>`MOV R0, 0` | `; Tâche dans la fonction`<br>`B AdresseDeRetour` |

- Pourquoi est-ce un mauvais exemple?
  - Il est impossible d'appeler la fonction de deux endroits différents…

# Appel de Fonction et Adresse de Retour

- Pour pouvoir revenir à plus d'un endroit, il faut sauvegarder l'adresse de retour avant de faire le branchement.
- Instruction `BL` (Branch and Link):

```
BLcc Adresse ; met l'adresse de retour dans LR
```

- Après l'exécution de la fonction, on place `PC = LR` pour continuer l'exécution après l'endroit où la fonction a été appelée, avec l'instruction `BX` (Branch and eXchange):

```
BX Rm          ; PC = Rm (Rm peut être n'importe quel registre
```

| Main | Fonction |
|---|---|
| `BL MaFonction ; LR = PC-4`<br>`MOV R0, 0`<br>`BL MaFonction ; LR = PC-4`<br>`ADD R0, R0, 1` | `; Tâche dans la fonction`<br>`BX LR  ; PC = LR` |

# Appel de Fonction et Adresse de Retour

- Que ce passe-t-il si une fonction appelle une autre fonction?

| Main | Fonction1 | Fonction2 |
|---|---|---|
| `; Debut Main`<br>`BL Fonction1`<br>`; Suite Main` | `; Debut Fonction1`<br>`BL Fonction2`<br>`; Suite 1 Fonction1`<br>`BL Fonction2`<br>`; Suite 2 Fonction1`<br>`BX LR` | `; Code Fonction 2`<br>`BX LR` |

- La séquence de gestion des adresses de retour devient:
  - `LR` = “Suite Main”
  - `LR` = “Suite 1 Fonction1”
  - `PC` = `LR`, donc `PC` = “Suite 1 Fonction1”
  - `LR` = Suite 2 Fonction1
  - `PC = LR`, donc `PC` = “Suite 2 Fonction1”
  - `PC = LR`, donc `PC` = “Suite 2 Fonction1”… Le `BX LR` de Fonction1 “retourne" au mauvais endroit!

# La pile

- La pile est une structure de données qui permet d'empiler et de dépiler des données. Le pile est un espace de la mémoire qui est géré comme une pile de feuilles: mettre ou retirer une donnée se fait toujours à partir du dessus de la pile.
- Le registre SP (Stack Pointer) devrait indiquer le dessus de la pile en tout temps.
- PUSH

```
PUSH {Rs}           ; Place le contenu de Rs sur la pile, SP = SP - 4
PUSH {R0, R1, R2}   ; Place le contenu de R0, R1, et R2 sur la pile,
                    ; SP = SP - 12
```

  - Permet de mettre une (ou plusieurs) donnée(s) sur la pile.
- POP

```
POP {Rd}            ; Place le contenu sur la pile dans Rd, SP = SP + 4
POP {R0, R1, R2}    ; Place le contenu sur la pile dans R2, R1, et R0
                    ; (dans l'ordre), SP = SP + 12
```

  - Permet de récupérer une (ou plusieurs) donnée(s) de la pile.

# Préparation d'une pile

- La pile est habituellement:
  - descendante: lors d'un PUSH, SP *diminue* de 4 octets (pourquoi 4?)
  - placée après les données en RAM, donc à une adresse supérieure
- Exemple:

```
  SECTION .text : CODE (2)
  CODE32

main
  LDR   SP, =MaPile    ; Préparons une pile (de 64 octets)
  ADD   SP, SP, #64    ; La pile descend, donc il faut commencer à la fin


  SECTION  `.noinit`:DATA(2)

MaVar       DS32    1  ; Variable
MaPile      DS32    16 ; Pile de 64 octets (16*4 = 64)
```

# Appel de fonction et adresse de retour, avec pile

- Que ce passe-t-il si une fonction appelle une autre fonction?

| Main | Fonction1 | Fonction2 |
|---|---|---|
| ; Debut Main<br>BL Fonction1<br>; Suite Main | ; Debut Fonction1<br>BL Fonction2<br>; Suite 1 Fonction1<br>BL Fonction2<br>; Suite 2 Fonction1<br>BX LR | ; Code Fonction 2<br>BX LR |

| Main | Fonction1 | Fonction2 |
|---|---|---|
| ; Debut Main<br>BL Fonction1<br>; Suite Main | **PUSH {LR}**<br>; Debut Fonction1<br>BL Fonction2<br>; Suite 1 Fonction1<br>BL Fonction2<br>; Suite 2 Fonction1<br>**POP {LR}**<br>BX LR | **PUSH {LR}**<br>; Code Fonction 2<br>**POP {LR}**<br>BX LR |

# Exemple d'appel de fonction

- L'instruction BL commande un branchement
- Le nom de la fonction suit l'instruction BL
- L'éditeur de lien convertira le nom de la fonction en adresse

Code C

```
Retour = FonctionSansParametre();
```

Code Assembleur

```
BL    FonctionSansParametre
```

# Appel de fonctions — conventions

- Paramètres:
  - On se sert de R0 à R3 lorsqu'il y en a 4 ou moins
  - S'il y en a plus?
    - On utilise la pile… et/ou la mémoire
- Valeur de retour:
  - On se sert de R0 lorsqu'il y en a 1 ou moins
  - S'il y en a plus?
    - On utilise la pile… et/ou la mémoire

# Appel de fonction: 1 paramètre et 1 retour

- On place la valeur du paramètre dans `R0` juste avant l'appel
- L'instruction `BL` commande un branchement
- `R0` contient la valeur de retour une fois la fonction exécutée

Code C

```
Retour = FonctionAUnParametre(0x12);
```

Code Assembleur

```
MOV   R0, #0x12              ; R0 contient le paramètre de la fonction
BL    FonctionAUnParametre   ; Appel de la fonction
MOV   R3, R0                 ; Récupère le résultat de la fonction
```

Code C

```
int FonctionAUnParametre (int param)
{ return param + 1; }
```

Code Assembleur

```
FonctionAUnParametre
ADD  R0, R0, #1              ; R0 contient le paramètre de la fonction
BX   LR                      ; R0 contient le résultat de la fonction
```

# Exemple

- Exemple de fonction

```
MOV  R0, #8          ; Nous voulons que R5 = Fonction(8)
BL   MaFonction
MOV  R5, R0
```

```
MaFonction           ; R0 = paramètre, R0 = valeur de retour

PUSH {LR}            ; Préserve LR

ADD R4, R0, R0       ; Calcule le double du paramètre passé en entrée
MOV R0, R4           ; R0 = valeur de retour

POP {LR}             ; Restaure R4 à sa valeur initiale
BX LR
```

- Quel est le problème?
  - Qu'arrive-t-il à R4?

# Préservation de l'environnement

- Le nombre de registres étant limité, on ne veut pas qu'une fonction remplace le contenu des registres
- Problème:
  - La fonction ne connait pas le nom des registres qui sont utilisés par le code qui fait l'appel de la fonction
  - Le code qui fait appel à la fonction ne connait pas le nom des registres qui sont utilisés par la fonction
- Solution:
  - La fonction "protège" le contenu des registres qu'elle utilise
- Méthode utilisée:
  - On protège le contenu d'un registre en le sauvegardant sur la pile avec `PUSH`
  - On récupère ce contenu avec `POP`

# Retour sur l'exemple

- Le code qui fait l'appel préserve le contenu de R0 pour ne pas le perdre lors du retour de la fonction
- La fonction préserve le contenu de R4 pour ne pas le corrompre en faisant ses calculs
- La valeur de retour est placée dans R5

```
PUSH {R0}
MOV  R0, #8                  ; Nous voulons appeler FonctionQuiPreserve(8)
BL   FonctionQuiPreserve
MOV  R5, R0
POP {R0}
```

```
FonctionQuiPreserve     ; R0 = paramètre, R0 = valeur de retour

PUSH {R4, LR}           ; Préserve LR et R4 -- nous allons nous en servir

ADD R4, R0, R0          ; Calcule le double du paramètre passé en entrée
MOV R0, R4              ; R0 = valeur de retour

POP {R4, LR}            ; Restaure R4 et LR à leur valeur initiale
BX LR
```

# Appel de fonction: 1—4 paramètres et 1 retour

- R0, R1, R2 et R3 servent à passer les paramètres
- R0 sert à retourner la valeur de retour

Code C

```
Retour = FonctionAMoinsDe5Parametres(0x12, 0x23, 0x34, 0x45);
```

Code Assembleur

```
MOV R0, #0x12                ; Paramètre 1
MOV R1, #0x23                ; Paramètre 2
MOV R2, #0x34                ; Paramètre 3
MOV R3, #0x45                ; Paramètre 4
BL FonctionAUnParametre      ; Appel de fonction
MOV R4, R0                   ; Valeur de sortie
```

Code C

```
int FonctionAMoinsDe5Parametres(int P0, int P1, int P2, int P3)
{ return P0 + P1 + P2 + P3; }
```

Code Assembleur

```
FonctionAMoinsDe5Parametres
ADD R0, R0, R1               ; Calcul de la somme
ADD R0, R0, R2
ADD R0, R0, R3
BX  LR                       ; Fonction terminée
```

# Cas à plus de 4 paramètres: par la pile

- On utilise `R0` à `R3`
- Si on en veut plus, on utilise la pile
- Il faut tenir compte des registres qu'on préserve en début de fonction avant de lire des valeurs dans la pile

```
MOV R0, #1
MOV R1, #2
MOV R2, #3
MOV R3, #4
MOV R5, #5                  ; Plaçons le 5e paramètre dans R5
PUSH {R5}
BL  FonctionA5Parametres
MOV R8, R0
POP {R5}
```

```
FonctionsA5Parametres
PUSH {R9, LR}               ; Sauvegarde LR et R9 (nous l'utiliserons)

LDR R9, [SP, #8]            ; R9 = paramètre 5
ADD R0, R0, R9              ; retour = paramètre 0 + paramètre 5

POP {R9, LR}                ; Restaure LR et R9
BX LR
```

# Autres cas à plus de 4 paramètres

- Les paramètres sont indépendants les uns des autres:
    - On les place un par un sur la pile et ils sont lus un par un
- Les paramètres font partie d'une chaîne ou d'un vecteur:
    - On place l'adresse du début des valeurs sur la pile
    - On place le nombre de valeurs sur la pile
    - La fonction peut lire en boucle
- Les paramètres font partie d'une structure dont les éléments n'occupent pas tous le même nombre d'octets:
    - On place l'adresse du début de la structure sur la pile
    - On place le nombre de mots utilisés pour mémoriser la structure
    - La fonction doit lire la pile en tenant compte des longueurs variées des valeurs de la structure
- Ça correspond à passer un pointeur ou une référence en langage plus évolué que l'assembleur (natif ou évolué)

# Cas à plusieurs retours

- R0 peut quand même servir à retourner une valeur
- Le code qui fait appel à la fonction passe des valeurs d'adresse en paramètre (par R0 jusqu'à R3 et/ou par la pile)
- Les adresses passées pointent sur des espaces mémoires où la fonction pourra écrire sans causer de problème au code qui lui fait appel
- La fonction fait ses calculs et elle utilise les valeurs d'adresses pour savoir où sauvegarder les résultats à retourner
- Le code qui a fait l'appel retrouve les valeurs retournées aux adresses qu'il a passé en paramètre
- Le principe fonctionne tant pour des variables indépendantes que pour des chaînes, des vecteurs, des structures, etc.

# Annexe: La pile, plus que pour les retours

- La pile sert à entreposer les adresses de retours comme vu précédemment.
- La pile sert aussi à passer des paramètres
- La pile sert à sauvegarder les registres utilisés dans une fonction.
- Souvent, les variables locales (dont la portée se limite à une fonction), sont de l'espace mémoire sur la pile allouée dynamiquement (le compilateur modifie SP au début de la fonction et à la fin pour réserver de l'espace mémoire). Sinon les variables locales sont des registres.
- La pile sert à effectuer des calculs mathématiques comme réaliser une chaînes d'additions et de multiplications
- La pile sert à sauvegarder le contexte d'une tâche lors d'interruptions (voir prochain cours!)
- La pile est une structure de donnée fondamentale pour les ordinateurs. Tous les processeurs ont un pointeur de pile…