Projet PARM

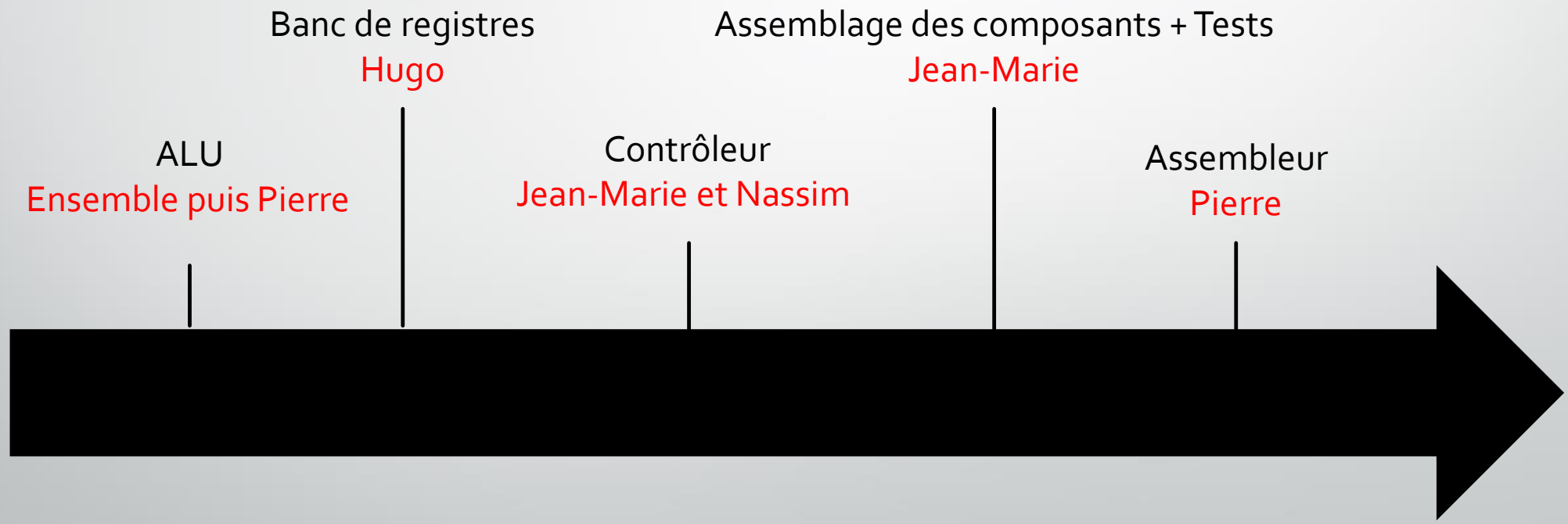
Groupe GDEB

GATTO – DORMOY – EL GAZZAH - BRICARD

Sommaire

- Organisation
- Présentation Composants
  - ALU
  - Banc de registres
  - Contrôleur
  - Assembleur
- Tests et Validation
- Conclusion

Organisation



Description des composants

ALU

- Composant qui calcule
- Entrée :
  - CODOP
  - A et B (Banc de registres ou immédiat)
  - Retenue + décalage
- Sortie :
  - Résultat
  - Drapeaux

<u>Méthode d'implémentation</u>

Utilisation de multiplexeurs pour la sortie et pour le flag C (carry) et V (overflow) avec le CODOP pour sélectionner

sous composant pour rendre le circuit plus lisible

Comparaison des bits de poids fort (bit de signe) pour détecter l'overflow

Banc de registres

- Mémoire la plus temporaire, pour manipuler les données car très accessible
- Entrées
  - DataIn (donnée à enregistrer)
  - Regdest (dans quel registre)
  - regA / regB (sélectionne les registres desquels on extrait les données pour Aout Bout)
- Sorties
  - Aout / Bout
  - R0-R7 (pour voir le contenu des registres)

<u>Méthode d'implémentation</u>

Utilisation des registres déjà existant

Regdest pour sélectionner dans le décodeur et envoyer DataIn dans le bon registre

2 multiplexeurs pour les sorties Aout et Bout sélectionnées par regA et par regB

Contrôleur

- Entrées notables: Instruction 16 bits en provenance de l'entrée ROM_In du CPU, flags en provenance de la sortie Flags de l'ALU.

Décodeur

- Le décodeur prend en argument les 6 bits de poids fort de l'instruction 16 bits
- Elle active le bloc correspondant à ces 6 bits: SASMov, Dprocessing, L/Store, SP_Address, Conditional

ShiftAddSubMov

- Ce composant prend en entrée une instruction 16 bits
- Génère les numéros de registres dans lesquels on va lire les opérandes et stocker le résultat
- Génère le codeOp_ALU
- Génère le nombre de décalage pour les instructions LSL, LSR et ASR
- Génère le flags_update_mask qui correpond aux flags qui vont être générés par l'instruction de l'ALU correspondant à codeOp_ALU
- Génère la sortie imm32 correspondant à la valeur des immédiats de Mov, Add et Sub à destination de l'ALU

Data_Processing

- Ce composant prend en entrée une instruction 16 bits
- Génère les numéros de registres dans lesquels on va lire les opérandes et stocker le résultat
- Génère le codeOp_ALU
- Génère le flags_update_mask qui correpond aux flags qui vont être générés par l'instruction de l'ALU correspondant à codeOp_ALU généré par le module Data_Processing

Flags APSR

- Entrée update_mask permet de sélectionner les drapeaux à mettre à jour
- Ce bloc conserve les drapeaux générés par la dernière instruction, pour qu'ils soient disponibles pour la prochaine instruction

Load/Store

- Objectif: déclencher LDR ou STR, et éventuellement activer PC_HOLD pour rester sur la même instruction du PC si on fait un LDR.
- Dans l'instruction 16 bits prise en entrée de ce composant, il y a 3 bits qui indiques dans quel registre on va faire le LDR ou le STR et 8 bits qui indiquent quelle adresse mémoire utiliser.

Assembleur

Le rôle de l'assembleur est de traduire un programme écrit en langage assembleur dans une représentation que le processeur saura interpréter. (en hexa)

Comment l'exécuter :

```
python assembleur.py nom_du_fichier
```

où nom_du_fichier est le nom du fichier où se trouve le code assembleur

Cet assembleur ne fonctionne qu'avec les instructrions décrites dans le projet PARM et la documentation officielle ARM

Sachant que certaines instructions génerées par gnueabi-gcc ne sont pas prise en compte par notre processeur, nous avons pris soin de créer des exemples qui conviennent a partir des codes générés.

Tests et Validation

- mini-programmes avec les instructions MOVS, LDR, STR, MULS
- code du maximum donné sur le forum, pour vérifier conditional, load/store, CMP
- Adaptation du code comparaison de multiplication écrit en C et traduit en assembleur par gnueabi-gcc, pour qu'il soit exécutable par notre cpu, pour vérifier LSLS, MULS, CMP, ADD, SUB, STR, LDR, etc.

Conclusion