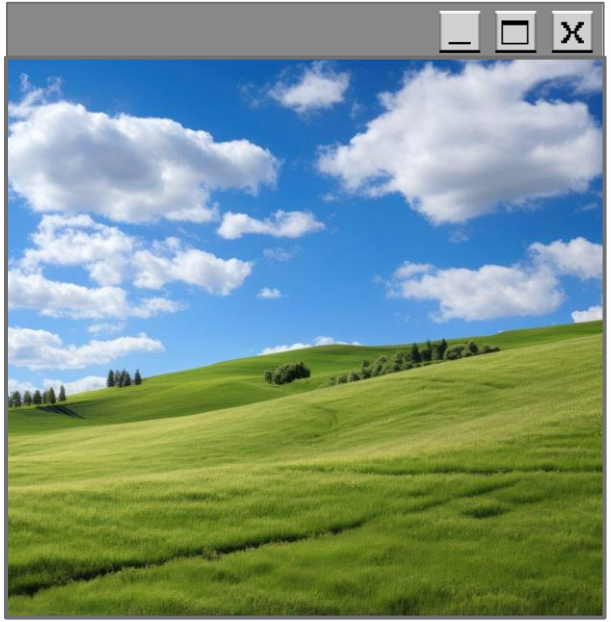
Ezedine / Samy / Othman / Eliot

Projet PARM

Processeur Polytech-ARM Cortex-M

(simulation Logisim)



Agenda & contraintes du projet

Agenda

- 1. **Slides** : compréhension globale + architecture (~4 min)
- 2. **Démo CPU Logisim** : exécution pas-à-pas (~2 min)
- 3. **Démo assembleur** : .s → .hex Logisim (~2 min)
- 4. **Démo tests** + taux de couverture (~2 min)

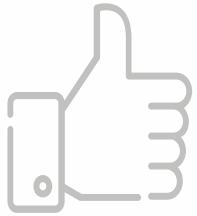
Cadre / contraintes imposées

- **ISA** : Thumb 16 bits (instructions sur 16 bits), données 32 bits
- **Mémoire** : ROM (programme) + RAM (données) dans Logisim
- **Timing** : LDR/STR = 2 cycles, le reste = 1 cycle
- **Objectif soutenance** : expliquer tout le fonctionnement, puis montrer le test fonctionnel le plus complet

Ce qu'on va prouver aujourd'hui

Que la chaîne complète fonctionne :

assembleur → ROM → CPU Logisim → résultat + tests/couverture.



Répartition des tâches

Othman

- ALU
- Shift, add, sub, mov
- Chemin des données

Eliot

- Flags APSR
- Assembleur
- Conditional

Samy

- Data Processing
- SP Address
- Tableau de synthèse

Ezedine

- Banc de Registres
- Load Store
- Décodeur

Fonctionnement global (de bout en bout)

Code → binaire → exécution

- **Fichier assembleur** .s
- **Assembleur** (parseur + encodeur) → génère ROM Logisim .hex
- **Chargement ROM** dans Logisim
- **CPU exécute** → lit/écrit **RAM** → met à jour registres + **flags**
- **Résultat observable** : registres / mémoire / flags / PC

```
PS C:\Code\PARM> python asm_parm.py TEST.s -o TEST.hex
OK: 2 instructions -> TEST.hex
PS C:\Code\PARM>
```

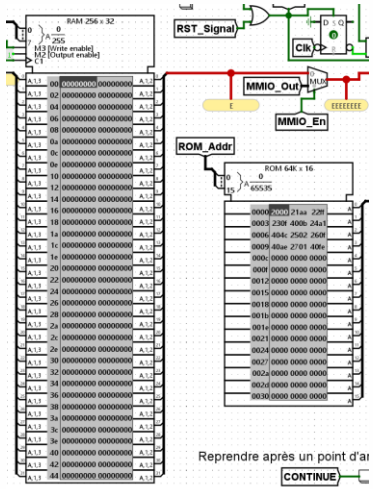
```
movs r2, #5
adds r0, r1, r2
```

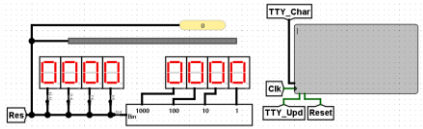
```
v2.0 raw
2205 1888
```

Tableau jeu d'instructions

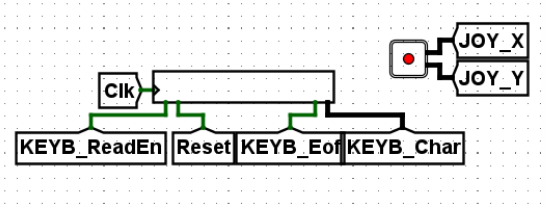
Description	UAL code				Bits																Flags			
	Instruction	operandes			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	C	V	N	Z
Logical Shift Left	LSLS	Rd	Rm	imm5	0	0	0	0	0	imm5					Rn			Rd			x		x	x
Logical Shift Right	LSRS	Rd	Rm	imm5	0	0	0	0	1	imm5					Rn			Rd			x		x	x
Arithmetic Shift Right	ASRS	Rd	Rm	imm5	0	0	0	1	0	imm5					Rn			Rd			x		x	x
Shift, add, sub, mov					S																			
Add register	ADDS	Rd	Rn	Rm	0	0	0	1	1	0	0	Rm			Rn			Rd			x	x	x	x
Substract register	SUBS	Rd	Rn	Rm	0	0	0	1	1	0	1	Rm			Rn			Rd			x	x	x	x
Add 3-bit immediate	ADDS	Rd	Rn	imm3	0	0	0	1	1	1	0	imm3			Rn			Rd			x	x	x	x
Sub stract 3-bit immediate	SUBS	Rd	Rn	imm3	0	0	0	1	1	1	1	imm3			Rn			Rd			x	x	x	x
Move	MOVS	Rd	imm8		0	0	1	0	0	Rd					imm8								x	x
Compare	CMP	Rd	imm8		0	0	1	0	1	Rd					imm8						x	x	x	x
Add 8-bit immediate	ADDS	Rdn	imm8		0	0	1	1	0	Rdn					imm8						x	x	x	x
Subtract 8-bit immediate	SUBS	Rdn	imm8		0	0	1	1	1	Rdn					imm8						x	x	x	x
Data processing					0	1	0	0	0	opcode														
Bitwise AND	ANDS	Rdn	Rm		0	1	0	0	0	0	0	0	0	0	Rm			Rdn			0		x	x
Exclusive OR	EORS	Rdn	Rm		0	1	0	0	0	0	0	0	0	1	Rm			Rdn			0		x	x
Logical Shift Left	LSLS	Rdn	Rm		0	1	0	0	0	0	0	0	1	0	Rm			Rdn			x		x	x
Logical Shift Right	LSRS	Rdn	Rm		0	1	0	0	0	0	0	0	1	1	Rm			Rdn			x		x	x
Arithmetic Shift Right	ASRS	Rdn	Rm		0	1	0	0	0	0	0	1	0	0	Rm			Rdn			x		x	x
Add with Carry	ADCS	Rdn	Rm		0	1	0	0	0	0	0	1	0	1	Rm			Rdn			x	x	x	x
Substract with Carry	SBCS	Rdn	Rm		0	1	0	0	0	0	0	1	1	0	Rm			Rdn			x	x	x	x
Rotate Right	RORS	Rdn	Rm		0	1	0	0	0	0	0	1	1	1	Rm			Rdn			x		x	x
Set Flags on bitwise AND	TST	Rn	Rm		0	1	0	0	0	0	1	0	0	0	Rm			Rn					x	x
Reverse Subtract from 0	RSBS	Rd	Rn		0	1	0	0	0	0	1	0	0	1	Rn			Rd			x	x	x	x
Compare Registers	CMP	Rn	Rm		0	1	0	0	0	0	1	0	1	0	Rm			Rn			x	x	x	x
Compare Negative	CMN	Rn	Rm		0	1	0	0	0	0	1	0	1	1	Rm			Rn			x	x	x	x
Logical OR	ORRS	Rdn	Rm		0	1	0	0	0	0	1	1	0	0	Rm			Rdn					x	x
Multiply Two Registers	MULS	Rdm	Rn		0	1	0	0	0	0	1	1	0	1	Rn			Rdm					x	x
Bit Clear	BICS	Rdn	Rm		0	1	0	0	0	0	1	1	1	0	Rm			Rdn					x	x
Bitwise NOT	MVNS	Rd	Rm		0	1	0	0	0	0	1	1	1	1	Rm			Rd					x	x
Load / Store					1	0	0	1	opcode															
Store Register	STR	Rt	SP	imm8	1	0	0	1	0	Rt					imm8									
Load Register	LDR	Rt	SP	imm8	1	0	0	1	1	Rt					imm8									
Miscellaneous 16-bit instructions					1	0	1	1	opcode															
Add Immediate to SP	ADD	SP	imm7		1	0	1	1	0	0	0	0	0	0	imm7									
Subtract Immediate from SP	SUB	SP	imm7		1	0	1	1	0	0	0	0	0	1	imm7									
Conditional Branch	B				1	1	0	1	cond						imm8									
égalité	BEQ	label			1	1	0	1	0	0	0	0	imm8											
différence	BNE	label			1	1	0	1	0	0	0	1	imm8											
retenue	BCS	label			1	1	0	1	0	0	1	0	imm8											
pas de retenue	BCC	label			1	1	0	1	0	0	1	1	imm8											
négatif	BMI	label			1	1	0	1	0	1	0	0	imm8											
positif ou nul	BPL	label			1	1	0	1	0	1	0	1	imm8											
dépassement de capacité	BVS	label			1	1	0	1	0	1	1	0	imm8											
pas de dépassement de capacité	BVC	label			1	1	0	1	0	1	1	1	imm8											
supérieur (non signé)	BHI	label			1	1	0	1	1	0	0	0	imm8											
inférieur ou égal (non signé)	BLS	label			1	1	0	1	1	0	0	1	imm8											
supérieur ou égal (signé)	BGE	label			1	1	0	1	1	0	1	0	imm8											
inférieur (signé)	BLT	label			1	1	0	1	1	0	1	1	imm8											
supérieur (signé)	BGT	label			1	1	0	1	1	1	0	0	imm8											
inférieur ou égal (signe)	BLE	label			1	1	0	1	1	1	0	1	imm8											
toujours vrai	B ou BAL	label			1	1	0	1	1	1	1	0	imm8											

Architecture CPU (datapath)





Interface	Entrées / Sorties	À quoi ça sert
ROM	adresse → instruction	lire le programme
RAM	adresse + données + write enable	lire/écrire données (load/store)
Clock/Reset	horloge / remise à zéro	cadence + init



- **Compteur d'instructions** : fournit l'adresse de la prochaine instruction
- **Mémoire programme (ROM)** : renvoie l'instruction courante
- **Décodage / contrôle** : choisit les chemins et active les écritures
- **Banc de registres + ALU** : calcule le résultat et met à jour les indicateurs
- **Interface mémoire (RAM)** : utilisée uniquement pour charger / stocker

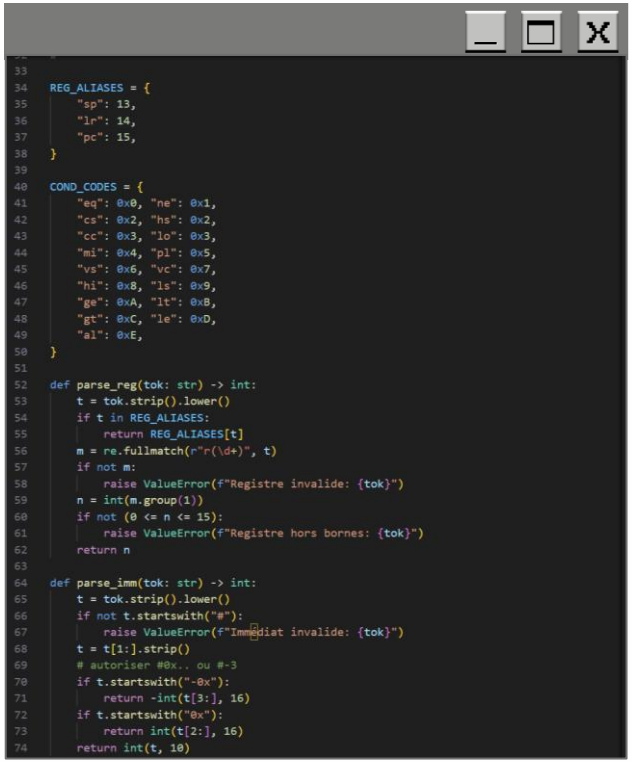
ISA supportée (Thumb 16 bits) - ce que notre CPU exécute

Déplacements & arithmétique
MOV (imm)
ADD / SUB
Shift (LSL/LSR/ASR)
CMP (selon impl.)

Traitement logique/ALU
AND / ORR / EOR
ADC / SBC
TST / CMN

Mémoire (pile)
LDR (SP + imm)
STR (SP + imm)
Adresse = SP + offset
2 cycles

Branchements
B conditionnel
offset signé (imm)
dépend des indicateurs
pas de BL



```
REG_ALIASES = {
    "sp": 13,
    "lr": 14,
    "pc": 15,
}

COND_CODES = {
    "eq": 0x0, "ne": 0x1,
    "cs": 0x2, "hs": 0x2,
    "cc": 0x3, "lo": 0x3,
    "mi": 0x4, "pl": 0x5,
    "vs": 0x6, "vc": 0x7,
    "hi": 0x8, "ls": 0x9,
    "ge": 0xA, "lt": 0xB,
    "gt": 0xC, "le": 0xD,
    "al": 0xE,
}

def parse_reg(tok: str) -> int:
    t = tok.strip().lower()
    if t in REG_ALIASES:
        return REG_ALIASES[t]
    m = re.fullmatch(r"r(\d+)", t)
    if not m:
        raise ValueError(f"Registre invalide: {tok}")
    n = int(m.group(1))
    if not (0 <= n <= 15):
        raise ValueError(f"Registre hors bornes: {tok}")
    return n

def parse_imm(tok: str) -> int:
    t = tok.strip().lower()
    if not t.startswith("#"):
        raise ValueError(f"Immédiat invalide: {tok}")
    t = t[1:].strip()
    # autoriser #0x.. ou #-3
    if t.startswith("-0x"):
        return -int(t[3:], 16)
    if t.startswith("0x"):
        return int(t[2:], 16)
    return int(t, 10)
```

Assembleur / Parseur

Transformer un programme assembleur en fichier ROM compatible Logisim pour exécuter sur notre CPU.

- Entrée : fichier .s (code assembleur)
- Analyse : lecture ligne par ligne, découpage en tokens, gestion labels
- Encodage : conversion en instructions 16 bits (et gestion des erreurs)
- Sortie : fichier .hex Logisim

ALU + Indicateurs (NZCV)

L’ALU fait les calculs (arithmétique, logique, décalages) et met à jour les indicateurs.

Entrées :

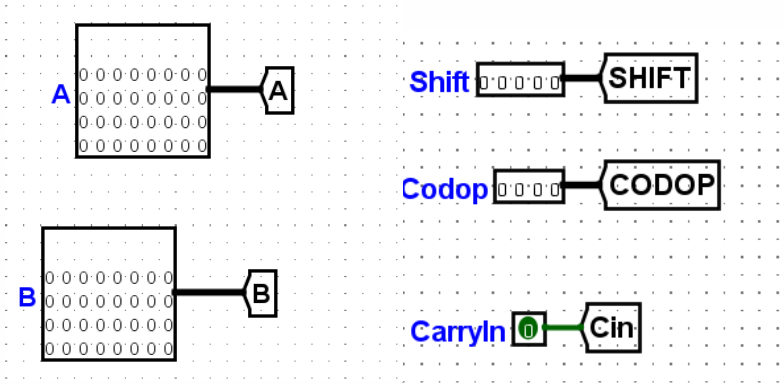
A (32 bits) : valeur venant d’un registre
B (32 bits) : valeur venant d’un registre / immédiat
Paramètres : type d’opération + décalage + retenue (si besoin)

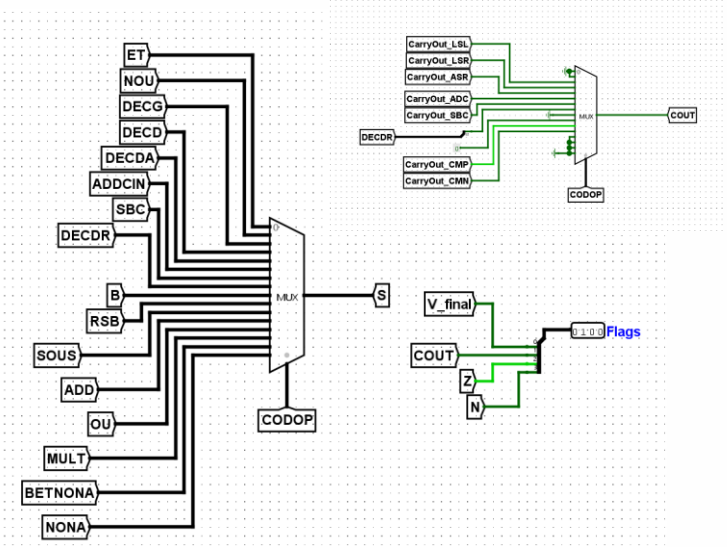
Sorties :

Résultat (32 bits)
Indicateurs NZCV
- **N** : résultat négatif
- **Z** : résultat = 0
- **C** : retenue (carry)
- **V** : dépassement (overflow)

ALU + Indicateurs (NZCV)

L'ALU fait les calculs (arithmétique, logique, décalages) et met à jour les indicateurs.





Tests & Taux de couverture

Tests unitaires :

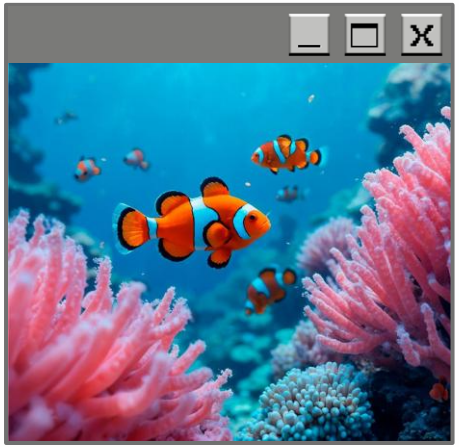
98,9%

Tests d'intégration CPU :

91%

Test shift add sub mov	100%
Test conditional	100%
Test sp address	100%
Test op code	95,3%
Test data processing	100%
Test load store	OK
Test alu	98,3%

Test fonctionnel final (démo) + Conclusion



DEMO

Axes d'amélioration

Architecture / Contrôleurs : Faire dès le début une table de contrôle et un ordre d'implémentation des contrôleurs pour limiter les dépendances et les recâblages

Tests et vérification : Augmenter la couverture avec une suite de non-régression (programmes courts) + tests de cas limites (flags, overflow, mémoire) et métriques de coverage

Organisation de groupe : Mettre en place des sprints avec livrables, revues d'intégration régulières et conventions d'interface (nom/largeurs/timing) pour sécuriser la collaboration

MERCI