
L'Architecture Codex

De la Porte Logique au Système d'Exploitation
32-bits

Tahiry RAZAFINDRALAMBO

2026

Table des matières

1 L'Architecture Codex : Guide de l'Étudiant	1
1.1 Table des Matières	1
1.2 Comment utiliser ce guide	2
2 Introduction	3
2.1 Le Mystère de l'Ordinateur	3
2.2 Pourquoi Construire un Ordinateur ?	3
2.2.1 Le problème de la "boîte noire"	3
2.2.2 L'approche "Du NAND au Tetris"	4
2.3 Ce que Vous Allez Construire	4
2.3.1 La beauté de l'abstraction	5
2.4 L'Architecture Codex A32	5
2.4.1 Pourquoi ces changements ?	5
2.5 Ce que Vous Allez Apprendre	6
2.6 Vos Outils	6
2.6.1 Les Outils en Ligne de Commande	6
2.6.2 Le Simulateur Web (Recommandé)	6
2.6.3 Le CPU Visualizer	7
2.7 Comment Utiliser ce Livre	7
2.7.1 L'approche recommandée	7
2.7.2 Si vous êtes bloqué	7
2.8 La Grande Aventure Commence	8
3 Logique Booléenne	9
3.1 Où en sommes-nous ?	10
3.2 Pourquoi le Binaire ?	10
3.2.1 La question fondamentale	10
3.2.2 Du voltage au bit	11
3.2.3 L'abstraction qui libère	11
3.3 La Porte NAND : Notre Axiome	12
3.3.1 Pourquoi partir du NAND ?	12
3.3.2 Table de Vérité NAND	12
3.3.3 Symbole graphique	13
3.4 Construction des Portes Élémentaires	13
3.4.1 A. NOT (Inverseur) — L'inversion de la réalité	13
3.4.2 B. AND (Et) — La conjonction	14

3.4.3 C. OR (Ou) — L'alternative	15
3.4.4 D. XOR (Ou Exclusif) — La différence	15
3.4.5 E. Multiplexeur (Mux) — L'aiguilleur	16
3.4.6 F. Démultiplexeur (DMux) — L'inverse de l'aiguilleur	17
3.5 Le Lien avec l'Ordinateur Complet	17
3.5.1 Du NAND au CPU : La feuille de route	17
3.6 Description Matérielle (Codex HDL)	18
3.6.1 Pourquoi un langage de description ?	18
3.6.2 Structure d'un fichier .hdl	18
3.6.3 Vocabulaire essentiel	19
3.6.4 Règles de connexion	20
3.6.5 Bus (Vecteurs de bits)	20
3.6.6 Exemple complet : XOR en HDL	20
3.7 Exercices Pratiques	21
3.7.1 Exercices sur le Simulateur Web	21
3.7.2 Comment lancer le simulateur web ?	22
3.7.3 Alternative : Tests en ligne de commande	22
3.8 Défis Supplémentaires	22
3.8.1 Défi 1 : Minimiser le nombre de NAND	22
3.8.2 Défi 2 : Implémenter IMPLIES	22
3.8.3 Défi 3 : Mux à 4 entrées	23
3.9 Ce qu'il faut retenir	23
3.10 Auto-évaluation	24
3.10.1 Questions de compréhension	24
3.10.2 Mini-défi pratique	24
3.10.3 Checklist de validation	24
4 Arithmétique Binaire	25
4.1 Où en sommes-nous ?	26
4.2 Pourquoi l'Arithmétique est-elle si Importante ?	26
4.2.1 Au cœur de tout calcul	26
4.2.2 Ce que nous allons construire	27
4.3 Représentation des Nombres	27
4.3.1 Le Système Binaire (Base 2)	27
4.3.2 Taille des nombres dans Codex	28
4.3.3 Les Nombres Négatifs : Le Complément à 2	28
4.3.4 Comment obtenir le complément à 2 (la valeur négative) ?	28
4.3.5 Pourquoi le complément à 2 est-il génial ?	29
4.4 L'Addition Binaire	29
4.4.1 Les règles de base (sur 1 bit)	29
4.4.2 Exemple d'addition sur 4 bits	29
4.5 Le Demi-Additionneur (Half Adder)	30
4.5.1 Table de vérité	30

4.5.2	L'insight clé	30
4.5.3	Schéma du circuit	31
4.5.4	Limitation	31
4.6	L'Additionneur Complet (Full Adder)	31
4.6.1	Interface	31
4.6.2	Table de vérité	31
4.6.3	Comment le construire ?	32
4.7	L'Additionneur 32-bits (Ripple Carry Adder)	32
4.7.1	Schéma simplifié	33
4.7.2	Le compromis du Ripple Carry	33
4.8	L'ALU (Arithmetic Logic Unit)	33
4.8.1	Pourquoi combiner arithmétique et logique ?	33
4.8.2	Interface de l'ALU Codex	34
4.8.3	Les Opérations de l'ALU	34
4.8.4	Comment implémenter la soustraction ?	34
4.8.5	Les Drapeaux (Flags)	35
4.8.6	Comment calculer les drapeaux ?	35
4.9	Architecture de l'ALU	36
4.10	Exercices Pratiques	36
4.10.1	Exercices sur le Simulateur Web	36
4.10.2	Conseils pour l'ALU	37
4.10.3	Tests en ligne de commande	37
4.11	Défis Supplémentaires	38
4.11.1	Défi 1 : Carry Lookahead	38
4.11.2	Défi 2 : Multiplicateur	38
4.11.3	Défi 3 : Comparateur	38
4.12	Le Lien avec le CPU	38
4.13	Ce qu'il faut retenir	39
4.14	Auto-évaluation	39
4.14.1	Questions de compréhension	39
4.14.2	Mini-défi pratique	40
4.14.3	Checklist de validation	40
5	Logique Séquentielle et Mémoire	41
5.1	Où en sommes-nous ?	42
5.2	Pourquoi la Mémoire est-elle Fondamentale ?	42
5.2.1	Le problème de l'état	42
5.2.2	Ce que stocke la mémoire	43
5.2.3	Combinatoire vs Séquentiel	43
5.3	Le Temps et l'Horloge (Clock)	43
5.3.1	Le problème de la synchronisation	43
5.3.2	Front montant (Rising Edge)	44
5.3.3	Fréquence d'horloge	44

5.4	La Bascule D (D Flip-Flop / DFF)	44
5.4.1	Interface	44
5.4.2	Comportement	45
5.4.3	Pourquoi est-ce utile ?	45
5.4.4	Comment fonctionne une DFF en interne ?	45
5.5	Le Registre 1-bit (Bit)	45
5.5.1	Le problème	46
5.5.2	La solution : la rétroaction	46
5.5.3	C'est magique !	46
5.6	Le Registre 32-bits	46
5.6.1	Du bit au mot	46
5.6.2	Le rôle des registres dans le CPU	47
5.7	La RAM (Random Access Memory)	47
5.7.1	Du registre à la mémoire	47
5.7.2	Interface de la RAM	48
5.7.3	Comment ça marche ?	48
5.7.4	Exemple : RAM8 (8 mots de 32 bits)	49
5.7.5	Construction hiérarchique de grandes RAMs	49
5.8	Le Compteur de Programme (PC)	49
5.8.1	Pourquoi est-il spécial ?	50
5.8.2	Les trois modes du PC	50
5.8.3	Schéma simplifié	50
5.8.4	Le lien avec l'exécution du programme	50
5.9	Les Différents Types de Mémoire	51
5.10	Exercices Pratiques	51
5.10.1	Exercices sur le Simulateur Web	51
5.10.2	Ordre de progression recommandé	52
5.10.3	Prérequis	52
5.10.4	Tests en ligne de commande	52
5.11	Défis Supplémentaires	53
5.11.1	Défi 1 : RAM avec deux ports de lecture	53
5.11.2	Défi 2 : Compteur avec valeur maximale	53
5.11.3	Défi 3 : Registre à décalage (Shift Register)	53
5.12	Le Lien avec la Suite	53
5.13	Ce qu'il faut retenir	54
5.14	Auto-évaluation	54
5.14.1	Questions de compréhension	54
5.14.2	Mini-défi pratique	55
5.14.3	Checklist de validation	55
6	Architecture Machine (ISA A32)	56
6.1	Où en sommes-nous ?	57

6.2	Qu'est-ce qu'une Architecture ?	58
6.2.1	Le contrat fondamental	58
6.2.2	Codex A32 : Une architecture RISC moderne	58
6.3	Pourquoi RISC ? L'architecture Load/Store	58
6.3.1	CISC vs RISC	58
6.3.2	La règle d'or Load/Store	58
6.4	Le Cycle de Vie d'une Instruction	59
6.5	Les Registres : Le Plan de Travail	60
6.5.1	Vue d'ensemble	60
6.5.2	Rôles des registres	60
6.5.3	Le cas spécial de R15 (PC)	60
6.6	La Carte Mémoire (Memory Map)	61
6.6.1	Organisation de la mémoire Codex	61
6.6.2	Le Memory-Mapped I/O (MMIO)	61
6.7	Le Format des Instructions	62
6.7.1	Les bits de condition (31-28)	62
6.7.2	Les classes d'instructions (27-25)	63
6.8	Les Instructions en Détail	63
6.8.1	A. Opérations Arithmétiques et Logiques	63
6.8.2	B. Accès Mémoire (Load/Store)	64
6.8.3	C. Branchements	64
6.9	La Pile (Stack)	65
6.9.1	Fonctionnement	65
6.9.2	Push et Pop (manuel)	66
6.10	Exemples de Programmes	66
6.10.1	Exemple 1 : Somme de 1 à N	66
6.10.2	Exemple 2 : Maximum de deux nombres (avec prédication)	66
6.10.3	Exemple 3 : Dessiner un pixel	67
6.11	Gestion des Erreurs (Traps)	67
6.12	Exercices Pratiques	68
6.12.1	Exercices sur le Simulateur Web	68
6.12.2	Tests en ligne de commande	68
6.13	Ce qu'il faut retenir	68
6.14	Auto-évaluation	69
6.14.1	Questions de compréhension	69
6.14.2	Mini-défi pratique	69
6.14.3	Checklist de validation	70
7	Le Processeur (CPU)	71
7.1	Où en sommes-nous ?	72
7.2	Deux Implémentations du CPU	72
7.2.1	CPU Mono-cycle (Simulateur Rust)	72
7.2.2	CPU Pipeline (HDL)	73

7.2.3 Pourquoi deux implémentations ?	74
7.3 Qu'est-ce qu'un CPU ?	74
7.3.1 Le chef d'orchestre	74
7.3.2 Ce que nous avons construit jusqu'ici	75
7.3.3 Ce qu'il reste à construire	75
7.4 Architecture du CPU (Data Path)	75
7.4.1 Les flux de données	76
7.5 Les Composants du CPU	76
7.5.1 1. Le Compteur de Programme (PC)	76
7.5.2 2. Le Décodeur (Decoder)	77
7.5.3 3. L'Unité de Contrôle (Control)	78
7.5.4 4. Le Vérificateur de Condition (CondCheck)	78
7.5.5 5. Le Banc de Registres (RegFile)	79
7.5.6 6. Les Multiplexeurs	79
7.6 Le Cycle d'Exécution en Détail	80
7.6.1 Phase 1 : Fetch (Récupération)	80
7.6.2 Phase 2 : Decode (Décodage)	80
7.6.3 Phase 3 : Register Read (Lecture des registres)	80
7.6.4 Phase 4 : Execute (Exécution)	80
7.6.5 Phase 5 : Memory (Accès mémoire)	80
7.6.6 Phase 6 : Writeback (Écriture)	81
7.6.7 Phase 7 : PC Update	81
7.7 Implémentation du CPU en HDL	81
7.8 Exercices Pratiques	82
7.8.1 Exercices sur le Simulateur Web	82
7.8.2 Ordre de progression	82
7.8.3 Tests en ligne de commande	83
7.9 CPU Visualizer : L'Outil Interactif	83
7.9.1 Accéder au Visualizer	83
7.9.2 Fonctionnalités du Visualizer	83
7.9.3 Les Démonstrations Intégrées	85
7.9.4 Contrôles	85
7.9.5 Charger Votre Propre Code	85
7.9.6 Exercice Pratique	86
7.10 Conseils de Débogage	86
7.10.1 Le PC reste à 0 ?	86
7.10.2 Les branchements ne marchent pas ?	86
7.10.3 Rien ne s'écrit dans les registres ?	86
7.10.4 LDR/STR ne fonctionne pas ?	86
7.11 Aller Plus Loin : Le CPU Pipeline	87
7.11.1 Pourquoi le CPU Single-Cycle est Lent	87
7.11.2 Le Pipeline à 5 Étages	88
7.11.3 Visualisation du Pipeline en Action	91

7.11.4	Les Registres de Pipeline	92
7.11.5	Les Aléas (Hazards)	93
7.11.6	Architecture Complète du CPU Pipeline	97
7.11.7	Exercices Pratiques : Projet 6	97
7.11.8	Comment Tester le CPU Pipeline HDL	99
7.11.9	Résumé : Pipeline vs Single-Cycle	101
7.11.10	Pour Aller Encore Plus Loin	102
7.12	Le Lien avec la Suite	102
7.12.1	Le parcours complet	102
7.13	Ce qu'il faut retenir	103
7.14	Auto-évaluation	103
7.14.1	Questions de compréhension	103
7.14.2	Mini-défi pratique	103
7.14.3	Checklist de validation	104
8	L'Assembleur	105
8.1	Où en sommes-nous ?	106
8.2	Le Rôle de l'Assembleur	106
8.2.1	Du texte au binaire	106
8.2.2	Les trois tâches de l'assembleur	107
8.3	La Stratégie des Deux Passes	107
8.3.1	Pourquoi deux passes ?	107
8.3.2	Passe 1 : Construction de la Table des Symboles	107
8.3.3	Passe 2 : Génération du Code	108
8.4	Sections et Directives	108
8.4.1	Les Sections	108
8.4.2	Les Directives	108
8.5	Exemple d'Encodage	109
8.5.1	Étape 1 : Identifier l'instruction	109
8.5.2	Étape 2 : Déterminer la classe	109
8.5.3	Étape 3 : Assembler les bits	109
8.6	La Gestion des Grandes Constantes	109
8.6.1	Le problème	109
8.6.2	La solution : Le Literal Pool	109
8.7	Exercices Pratiques	110
8.7.1	Exercices sur le Simulateur Web	110
8.7.2	Exercice manuel : Encodage	110
8.7.3	Exercice : Table des symboles	110
8.7.4	Utilisation de l'outil CLI	111
8.8	Ce qu'il faut retenir	111
8.9	Auto-évaluation	112
8.9.1	Questions de compréhension	112
8.9.2	Mini-défi pratique	112

8.9.3 Checklist de validation	112
9 Construction du Compilateur	113
9.1 Où en sommes-nous ?	113
9.2 Le Rôle du Compilateur	114
9.2.1 Pourquoi un compilateur ?	114
9.2.2 Les étapes de compilation	114
9.3 Les Phases du Compilateur	114
9.3.1 Phase 1 : Lexer (Analyse Lexicale)	114
9.3.2 Phase 2 : Parser (Analyse Syntaxique)	115
9.3.3 Phase 3 : Génération de Code	115
9.4 Compilation des Structures de Contrôle	115
9.4.1 Variables locales	115
9.4.2 Expressions	116
9.4.3 If / Else	116
9.4.4 Boucle While	117
9.4.5 Boucle For	117
9.5 Compilation des Fonctions	117
9.5.1 Convention d'appel	117
9.5.2 Prologue et Épilogue	118
9.5.3 Appel de fonction	118
9.6 Le Compilateur C32	118
9.6.1 Utilisation	119
9.6.2 Exemple complet	119
9.7 Construisez Votre Propre Compilateur !	119
9.7.1 Phase 1 : Lexer (Analyse Lexicale)	119
9.7.2 Phase 2 : Parser (Analyse Syntaxique)	120
9.7.3 Phase 3 : Émission ASM (Génération de Code)	120
9.7.4 Phase 4 : CodeGen Expressions	120
9.7.5 Phase 5 : Structures de Contrôle	121
9.7.6 Phase 6 : Fonctions	121
9.7.7 Phase 7 : Projet Final	121
9.7.8 Techniques Clés	121
9.8 Exercices Pratiques	122
9.8.1 Exercices sur le Simulateur Web	122
9.8.2 Exercice : Traduire manuellement	122
9.9 Ce qu'il faut retenir	122
9.10 Auto-évaluation	123
9.10.1 Questions de compréhension	123
9.10.2 Mini-défi pratique	123
9.10.3 Checklist de validation	123
10 Langage de Haut Niveau (C32)	124
10.1 Où en sommes-nous ?	125

10.2 Pourquoi un Langage de Haut Niveau ?	125
10.2.1 Le problème de l'assembleur	125
10.2.2 L'abstraction	126
10.3 Spécification du Langage C32	126
10.3.1 Les Types de Données	126
10.3.2 Variables	127
10.3.3 Portée des variables	127
10.4 Opérateurs	127
10.4.1 Arithmétiques	127
10.4.2 Comparaison	127
10.4.3 Logiques	128
10.4.4 Binaires	128
10.5 Structures de Contrôle	128
10.5.1 If / Else	128
10.5.2 While	128
10.5.3 For	129
10.5.4 Do-While	129
10.6 Fonctions	129
10.6.1 Définition	129
10.6.2 Appel	129
10.6.3 Récursion	130
10.7 Pointeurs et Tableaux	130
10.7.1 Pointeurs	130
10.7.2 Tableaux	130
10.7.3 Lien entre pointeurs et tableaux	130
10.8 Accès au Matériel (MMIO)	131
10.8.1 L'écran	131
10.8.2 Le clavier	131
10.9 Exemple Complet	131
10.10 Exercices Pratiques	132
10.10.1 Exercices sur le Simulateur Web	132
10.10.2 Défis suggérés	133
10.11 Les Structures (Structs)	133
10.11.1 Définition d'une structure	133
10.11.2 Utilisation	133
10.11.3 Pointeur vers structure	134
10.11.4 Exemple complet	134
10.11.5 Structures imbriquées	134
10.12 Limitations du C32	135
10.13 Ce qu'il faut retenir	135
10.14 Auto-évaluation	135
10.14.1 Questions de compréhension	136
10.14.2 Mini-défi pratique	136

10.14.	Checklist de validation	136
11	Système d'Exploitation	137
11.1	Où en sommes-nous ?	138
11.2	Qu'est-ce qu'un Système d'Exploitation ?	139
11.2.1	La hiérarchie logicielle	139
11.2.2	Ce que fait un OS	139
11.3	Gestion de la Mémoire (Le Tas / Heap)	139
11.3.1	Le problème	139
11.3.2	L'allocateur "Bump" (Simple)	140
11.3.3	L'allocateur par Liste Chaînée (Avancé)	140
11.4	Bibliothèque Graphique	140
11.4.1	Le problème	140
11.4.2	Les fonctions graphiques	141
11.4.3	L'algorithme de Bresenham	141
11.5	Entrées / Sorties	142
11.5.1	Printf simplifié	142
11.5.2	Lecture du clavier	142
11.6	Interruptions et Timer (Concepts)	143
11.6.1	Le problème du polling	143
11.6.2	Les interruptions	143
11.7	Applications Démo	143
11.7.1	Compiler et exécuter une démo	143
11.8	Exercices Pratiques	144
11.8.1	Exercices sur le Simulateur Web	144
11.8.2	Défis suggérés	144
11.9	Le Parcours Complet	145
11.10	Ce qu'il faut retenir	145
11.11	Félicitations !	145
11.12	Auto-évaluation	146
11.12.1	Questions de compréhension	146
11.12.2	Réflexion finale	146
11.12.3	Checklist de validation finale	147
12	Annexe : Tous les Exercices	148
12.1A.	Exercices HDL (Portes Logiques)	148
12.1.1	Projet 1 : Portes de Base	148
12.1.2	Projet 2 : Arithmétique	148
12.1.3	Projet 3 : Mémoire	149
12.1.4	Projet 5 : CPU	149
12.1.5	Projet 6 : CPU Pipeline (Avancé)	149
12.1.6	Projet 7 : Cache L1	150
12.2B.	Exercices Assembleur A32	150
12.2.1	Bases	150

12.2.2	Controle de Flux	150
12.2.3	Memoire	151
12.2.4	Structures	151
12.2.5	Fonctions	151
12.2.6	Entrees/Sorties	151
12.2.7	Ecran (320x240, 1 bit/pixel)	152
12.2.8	Jeux Interactifs	152
12.2.9	Cache (Patterns d'Acces Memoire)	152
12.3	C. Exercices C32	153
12.3.1	Bases	153
12.3.2	Conditions	153
12.3.3	Boucles	153
12.3.4	Fonctions	154
12.3.5	Tableaux	154
12.3.6	Pointeurs	154
12.3.7	Operations Binaires	154
12.3.8	Recursion	155
12.3.9	Algorithmes Avances	155
12.3.10	Structures	155
12.3.11	Cache (Patterns d'Acces Memoire)	156
12.3.12	Entrees/Sorties	156
12.3.13	Projets Avances	156
12.4	D. Construction du Compilateur	157
12.4.1	Phase 1 : Lexer	157
12.4.2	Phase 2 : Parser	157
12.4.3	Phase 3 : Emission ASM	157
12.4.4	Phase 4 : CodeGen Expressions	157
12.4.5	Phase 5 : Structures de Controle	158
12.4.6	Phase 6 : Fonctions	158
12.4.7	Phase 7 : Projet Final	158
12.5	E. Systeme d'Exploitation	158
12.5.1	Initialisation	158
12.5.2	Gestion Memoire	159
12.5.3	Drivers	159
12.5.4	Console et Clavier	159
12.5.5	Shell et Applications	159
12.5.6	Multitache	160
12.5.7	Projets OS	160
12.6	Conseils pour les Exercices	160
13	L'Art du Débogage	161
13.1	Pourquoi ce Chapitre ?	161

13.21. Comprendre les Messages d'Erreur	161
13.2.1 Codes d'Erreur du Projet Codex	161
13.2.2 Erreurs d'Assemblage (E1xxx)	162
13.2.3 Erreurs de Compilation (E2xxx)	162
13.2.4 Erreurs de Liaison (E3xxx)	163
13.32. Utiliser le Simulateur Pas-à-Pas	163
13.3.1 Le Visualiseur CPU	163
13.3.2 Méthodologie de Débogage Pas-à-Pas	164
13.3.3 Exemple Pratique	164
13.43. Erreurs Courantes et Solutions	164
13.4.13.1 Assembleur	164
13.4.23.2 Compilateur C32	165
13.4.33.3 HDL	166
13.54. Méthodologie Systématique	167
13.5.1 La Méthode Scientifique du Débogage	167
13.5.2 La Technique de la Bisection	168
13.5.3 Ajouter des Traces	168
13.65. Erreurs de Traps et leur Diagnostic	169
13.6.1 Types de Traps	169
13.6.2 Déboguer un MEM_FAULT	169
13.76. Checklist de Débogage	169
13.7.1 Assembleur	169
13.7.2 C32	170
13.7.3 HDL	170
13.87. Exercices de Débogage	170
13.8.1 Exercice 1 : Trouvez le bug	170
13.8.2 Exercice 2 : Trouvez le bug	171
13.8.3 Exercice 3 : Trouvez le bug	171
13.9 Ce qu'il faut retenir	172
14 Le Cache : Pourquoi Votre Ordinateur Semble Rapide	173
14.1 Le Problème : La RAM est Lente !	173
14.1.1 La Hiérarchie Mémoire	173
14.1.2 L'écart de vitesse visualisé	174
14.2 La Solution : Le Principe de Localité	174
14.2.1 Localité Temporelle	174
14.2.2 Localité Spatiale	175
14.3 Comment Fonctionne le Cache ?	175
14.3.1 Les Lignes de Cache	175
14.3.2 Structure d'une Ligne de Cache	176
14.3.3 Découpage d'une Adresse	176
14.3.4 Structure Complète du Cache	177

14.4	Hit ou Miss : Que se passe-t-il ?	178
14.4.1	Scénario 1 : Cache HIT (Succès)	178
14.4.2	Scénario 2 : Cache MISS (Échec)	179
14.4.3	Diagramme de Flux Complet	180
14.5	Politiques d'Écriture	181
14.5.1	Write-Through (Écriture Directe)	181
14.5.2	Write-Back (Écriture Différée)	181
14.6	Impact sur vos Programmes	182
14.6.1	Parcours de Tableaux 2D : L'Ordre Compte !	182
14.6.2	Parcours Row-Major (En Ligne) - EFFICACE	182
14.6.3	Parcours Column-Major (En Colonne) - INEFFICACE	183
14.6.4	Comparaison Visuelle	183
14.6.5	Technique du Blocking	184
14.7	Implémentation HDL du Cache	185
14.7.1	Architecture Globale	185
14.7.2	Machine à États du Contrôleur	186
14.7.3	WordSelect : Sélection du Mot	186
14.8	Statistiques et Performance	187
14.8.1	Calcul du Hit Rate	187
14.9	Visualiser le Cache avec le CPU Visualizer	187
14.9.1	Accéder au Visualizer	187
14.9.2	La Démo "Cache"	187
14.9.3	Ce que vous verrez	188
14.9.4	Exercice Pratique	188
14.10	Exercices	188
14.10.1	Exercices HDL	188
14.10.2	Exercices Assembleur A32	189
14.10.3	Exercices C32	189
14.11	Résumé Visuel	190
14.12	Points Clés à Retenir	190
14.13	Auto-évaluation	190
14.13.1	Questions de compréhension	191
14.13.2	Mini-défi pratique	191
14.13.3	Checklist de validation	191
15	Solutions des Quiz d'Auto-évaluation	192
15.1	Chapitre 1 : Logique Booléenne	192
15.1.1	Q1. Pourquoi les ordinateurs utilisent-ils le binaire plutôt que le décimal ?	192
15.1.2	Q2. Pourquoi dit-on que NAND est une porte "universelle" ?	192
15.1.3	Q3. Quelle est la sortie de $XOR(1, 1)$? Et de $XOR(0, 1)$?	192

15.1.4Q4. Un multiplexeur (MUX) a 2 entrées de données a et b, et un signal de sélection sel. Si sel = 1, quelle entrée est transmise en sortie ?	193
15.1.5Q5. Combien de portes NAND faut-il au minimum pour construire un inverseur (NOT) ?	193
15.1.6Mini-défi : Table de vérité de $F(a, b) = \text{AND}(\text{OR}(a, b), \text{NOT}(a))$.	193
15.2Chapitre 2 : Arithmétique Binaire	193
15.2.1Q1. En complément à deux sur 8 bits, quelle est la représentation de -1 ?	193
15.2.2Q2. Pourquoi le complément à deux est-il préféré au "signe + magnitude" ?	194
15.2.3Q3. Quel est le rôle de la retenue entrante (Cin) dans un additionneur complet ?	194
15.2.4Q4. Une ALU reçoit les opérandes A=5 et B=3, avec l'opération SUB. Quel est le résultat et quels flags sont activés ?	194
15.2.5Q5. Quelle opération ALU permet de mettre à zéro les bits 4-7 d'un registre tout en préservant les autres ?	194
15.2.6Mini-défi : $0x7F + 0x01$ en 8 bits signés	194
15.3Chapitre 3 : Mémoire	195
15.3.1Q1. Quelle est la différence fondamentale entre un circuit combinatoire et un circuit séquentiel ?	195
15.3.2Q2. Pourquoi le signal d'horloge (clk) est-il essentiel dans les circuits séquentiels ?	195
15.3.3Q3. Un registre 32 bits est construit à partir de combien de bascules D ?	195
15.3.4Q4. Quelle est la capacité totale d'une RAM de 1024 mots de 32 bits, en kilooctets ?	195
15.3.5Q5. Dans une RAM8, combien de bits d'adresse sont nécessaires et pourquoi ?	195
15.3.6Mini-défi : Registre avec Enable	195
15.4Chapitre 4 : Architecture	196
15.4.1Q1. Quelle est la différence entre l'architecture von Neumann et Harvard ?	196
15.4.2Q2. Pourquoi utilise-t-on des registres plutôt que d'accéder directement à la RAM ?	196
15.4.3Q3. À quoi sert le registre PC (Program Counter) ?	196
15.4.4Q4. Qu'est-ce que le MMIO (Memory-Mapped I/O) ?	196
15.4.5Q5. Dans le cycle fetch-decode-execute, que se passe-t-il pendant la phase "decode" ?	197
15.4.6Mini-défi : Pourquoi 16 registres avec 4 bits ?	197
15.5Chapitre 5 : CPU	197
15.5.1Q1. Quelles sont les 5 étapes du cycle d'instruction dans un pipeline classique ?	197

15.5.2Q2. Qu'est-ce qu'un aléa de données (data hazard) et comment le résoudre ?	197
15.5.3Q3. Pourquoi le Program Counter est-il incrémenté de 4 (et non de 1) sur une architecture 32 bits ?	198
15.5.4Q4. Quel composant décide si un branchement conditionnel doit être pris ?	198
15.5.5Q5. Quelle est la différence entre les instructions LDR et STR ?	198
15.5.6Mini-défi : Temps d'exécution avec pipeline	198
15.6Chapitre 6 : Assembleur	198
15.6.1Q1. Quelle est la différence entre une instruction et une directive en assembleur ?	198
15.6.2Q2. Pourquoi doit-on sauvegarder LR avant d'appeler une sous-fonction avec BL ?	199
15.6.3Q3. Comment charger une valeur 32 bits arbitraire (ex: 0x12345678) dans un registre ?	199
15.6.4Q4. À quoi sert la directive .ltorg ?	199
15.6.5Q5. Quelle est la convention pour les registres "callee-saved" vs "caller-saved" ?	199
15.6.6Mini-défi : Valeur finale de R0	199
15.7Chapitre 7 : Compilateur	200
15.7.1Q1. Quelles sont les principales phases d'un compilateur ? . . .	200
15.7.2Q2. Qu'est-ce qu'un AST et à quoi sert-il ?	200
15.7.3Q3. Comment le compilateur gère-t-il les variables locales d'une fonction ?	200
15.7.4Q4. Quelle est la différence entre une erreur syntaxique et sémantique ?	201
15.7.5Q5. Pourquoi le compilateur génère-t-il parfois du code qui semble inefficace ?	201
15.7.6Mini-défi : Code assembleur généré	201
15.8Chapitre 8 : Le Langage C32	201
15.8.1Q1. Quelles sont les principales différences entre C32 et C standard ?	201
15.8.2Q2. Comment passer un tableau à une fonction en C32 ?	202
15.8.3Q3. Quelle est la différence entre *p et &x ?	202
15.8.4Q4. Comment accéder au framebuffer pour dessiner un pixel ?	202
15.8.5Q5. Pourquoi faut-il toujours terminer une chaîne par '\0' ? . .	202
15.8.6Mini-défi : Fonction de longueur de chaîne	203
15.9Chapitre 9 : Système d'Exploitation	203
15.9.1Q1. Quels sont les trois rôles principaux d'un système d'exploitation ?	203
15.9.2Q2. Comment fonctionne un appel système (syscall) ?	203
15.9.3Q3. Qu'est-ce que le memory mapping et pourquoi est-il utile ?	203
15.9.4Q4. Pourquoi l'OS a-t-il besoin d'un mode privilégié séparé ? .	204

15.9.5Q5. Comment l'OS gère-t-il plusieurs programmes en même temps ?	204
15.9.6Mini-défi : Allocation mémoire simple	204
15.10Chapitre 11 : Mémoire Cache	204
15.10.Q1. Pourquoi a-t-on besoin d'une mémoire cache ?	204
15.10.Q2. Qu'est-ce que la localité spatiale et temporelle ?	205
15.10.Q3. Quelle est la différence entre un cache direct-mapped et associatif ?	205
15.10.Q4. Que se passe-t-il lors d'un "cache miss" ?	205
15.10.Q5. Quelle est la différence entre write-through et write-back ?	205
15.10Mini-défi : Optimisation de parcours de matrice	205
15.11Chapitre 10bis : Débogage	206
16 Livre des Solutions	207
16.1A. Solutions HDL (Portes Logiques)	207
16.1.1Inv	207
16.1.2And2	207
16.1.3Or2	208
16.1.4Xor2	209
16.1.5Mux	210
16.1.6DMux	210
16.1.7Inv16	211
16.1.8And16	212
16.1.9Or16	213
16.1.10Mux16	213
16.1.11Or8Way	214
16.1.12Mux4Way16	215
16.1.13Mux8Way16	215
16.1.14DMux4Way	216
16.1.15DMux8Way	217
16.1.16HalfAdder	218
16.1.17FullAdder	218
16.1.18Add16	219
16.1.19Inc16	220
16.1.20Sub16	221
16.1.21ALU	221
16.1.22DFF1	223
16.1.23BitReg	223
16.1.24Register16	224
16.1.25PC	225
16.1.26RAM8	226
16.1.27RAM64	227
16.1.28RegFile	228

16.1.2	Decoder	229
16.1.3	CondCheck	230
16.1.3	Control	231
16.1.3	CPU	232
16.1.3	EF_ID_Reg	234
16.1.3	HazardDetect	235
16.1.3	ForwardUnit	236
16.1.3	CPU_Pipeline	237
16.1.3	Projet 7 : Cache L1	238
16.1.3	CacheLine	238
16.1.3	TagCompare	239
16.1.4	WordSelect	240
16.1.4	CacheController	240
16.2	B. Solutions Assembleur A32	242
16.2.1	Hello World	242
16.2.2	Addition	242
16.2.3	Soustraction	243
16.2.4	Logique	243
16.2.5	Doubler	243
16.2.6	Conditions	244
16.2.7	Valeur Absolue	244
16.2.8	Boucles	245
16.2.9	Multiplication	245
16.2.10	Fibonacci	246
16.2.11	Tableaux	246
16.2.12	Maximum Tableau	247
16.2.13	Mémoire	248
16.2.14	Structure Simple	248
16.2.15	Initialiser Structure	249
16.2.16	Structure Rectangle	250
16.2.17	Tableau de Structures	251
16.2.18	Somme x+y Structures	252
16.2.19	Fonctions	253
16.2.20	Fonction Add3	253
16.2.21	Écrire Caractère	253
16.2.22	Hello String	254
16.2.23	Print Loop	254
16.2.24	Pixel	255
16.2.25	Ligne Horizontale	255
16.2.26	Ligne Verticale	255
16.2.27	Rectangle	256
16.2.28	Damier	257
16.2.29	Lire un Caractère	257

16.2.3	Dire un Nombre à 2 Chiffres	258
16.2.3	Deviner le Nombre	259
16.2.3	Dégradé (Dithering)	260
16.2.3	Dégradé Plein Écran	261
16.2.3	Recherche Dichotomique	262
16.2.3	Cache: Accés Séquentiel	263
16.2.3	Cache: Accés avec Stride	264
16.2.3	Cache: Réutilisation Registre	265
16.3	C. Solutions C32	266
16.3.1	Variables	266
16.3.2	Expressions	266
16.3.3	Modulo	267
16.3.4	Incrémentations	267
16.3.5	Conditions	267
16.3.6	Else-If	268
16.3.7	Opérateurs Logiques	268
16.3.8	Maximum de 3	269
16.3.9	Boucle For	269
16.3.1	Boucle While	270
16.3.1	Boucles Imbriquées	270
16.3.1	Multiplication	270
16.3.1	Fonctions	271
16.3.1	Paramètres Multiples	271
16.3.1	Valeur Absolue	272
16.3.1	Min et Max	272
16.3.1	Tableaux	273
16.3.1	Maximum Tableau	273
16.3.1	Compter Éléments	274
16.3.2	Pointeurs	274
16.3.2	Swap	275
16.3.2	Pointeurs et Tableaux	275
16.3.2	Opérations Binaires	276
16.3.2	Puissance de 2	276
16.3.2	Factorielle	276
16.3.2	Fibonacci	277
16.3.2	Somme Récursive	277
16.3.2	PGCD (Euclide)	278
16.3.2	Puissance	278
16.3.3	Test Primalité	278
16.3.3	Tri à Bulles	279
16.3.3	Recherche Binaire	280
16.3.3	Inverser Tableau	281
16.3.3	Somme des Chiffres	281

16.3.35	Nombre Palindrome	282
16.3.36	Définition Struct	282
16.3.37	Pointeur Struct	283
16.3.38	Struct et Fonctions	283
16.3.39	Structs Imbriquées	283
16.3.40	Tableau de Structs	284
16.3.41	Sizeof Struct	284
16.3.42	Parcours en Ligne	285
16.3.43	Parcours en Colonne	286
16.3.44	Traitement par Blocs	286
16.3.45	Localité Temporelle	287
16.3.46	Écrire un Caractère	288
16.3.47	Afficher une Chaîne	289
16.3.48	Afficher un Nombre	289
16.3.49	Dessiner un Pixel	290
16.3.50	Ligne Horizontale	290
16.3.51	Dessiner un Rectangle	291
16.3.52	Écrivable d	291
16.3.53	Suite de Collatz	292
16.3.54	Projet Final	292
16.4	D. Solutions Construction du Compilateur	293
16.4.11.1	Reconnaître un Chiffre	293
16.4.21.2	Lire un Nombre	293
16.4.31.3	Identifier les Tokens	294
16.4.42.1	Évaluer $a + b$	295
16.4.52.2	Évaluer $a + b + c$	296
16.4.62.3	Respecter la Précédence	297
16.4.72.4	Gérer les Parenthèses	298
16.4.83.1	Émettre MOV	300
16.4.93.2	Émettre ADD/SUB/MUL	301
16.4.10.3	Émettre PUSH/POP	302
16.4.11.1	Compiler une Constante	304
16.4.12.2	Compiler $a + b$	305
16.4.13.3	Compiler Expressions Complètes	307
16.4.14.1	Générer des Labels	309
16.4.15.2	Générer Comparaisons	311
16.4.16.3	Compiler if/else	312
16.4.17.4	Compiler while	313
16.4.18.1	Prologue et Épilogue	315
16.4.19.2	Appels de Fonction	316
16.4.20.1	Mini-Compilateur Complet	317

16.5E. Solutions Systeme d'Exploitation	320
16.5.1Bootstrap	320
16.5.2Bump Allocator	321
16.5.3Free List	322
16.5.4Driver Écran	323
16.5.5Police Bitmap	325
16.5.6Console	328
16.5.7Driver Clavier	330
16.5.8Shell	332
16.5.9Calculatrice	335
16.5.10Variables Shell	337
16.5.11Compte à Rebours	340
16.5.12Interruptions	342
16.5.13Coroutines	345
16.5.14Scheduler	349
16.5.15Projet 1: Mini-OS Shell	354
16.5.16Projet 2: Task Manager	359

1 L'Architecture Codex : Guide de l'Étudiant

Ce livre vous guide pas à pas dans la construction d'un ordinateur complet, moderne et 32-bits, à partir de rien : des portes logiques jusqu'au système d'exploitation.

1.1 Table des Matières

1. Introduction

- La philosophie du projet.
- L'architecture Codex A32.
- Présentation des outils.

2. Chapitre 1 : Logique Booléenne

- L'abstraction binaire.
- Les portes logiques fondamentales (Nand, And, Or, Xor).
- *Exercices* : Implémentation dans `hdl_lib/01_gates`.

3. Chapitre 2 : Arithmétique Binaire (*À venir*)

- Représentation des nombres (Complément à 2).
- Additionneurs et ALU (Arithmetic Logic Unit).
- *Exercices* : `hdl_lib/03_arith`.

4. Chapitre 3 : Logique Séquentielle et Mémoire (*À venir*)

- Le temps et l'horloge.
- Flip-Flops, Registres et RAM.
- *Exercices* : `hdl_lib/04_seq & 02_multibit`.

5. Chapitre 4 : Architecture Machine (ISA A32) (*À venir*)

- Le jeu d'instructions A32-Lite.
- Registres, adressage et structure.
- *Référence* : `SPECS.md`.

6. Chapitre 5 : Le Processeur (CPU) (*À venir*)

- Implémentation du chemin de données (Data Path).

- Logique de contrôle.
- *Exercices* : hdl_lib/05_cpu.

7. **Chapitre 6 : L'Assembleur** (À venir)

- Traduction symbolique vers binaire.
- Gestion des symboles et des labels.
- *Outils* : a32_asm.

8. **Chapitre 7 : Construction du Compilateur** (À venir)

- Analyse lexicale et syntaxique (Parsing).
- Génération de code pour pile.
- *Outils* : c32_cli.

9. **Chapitre 8 : Langage de Haut Niveau (C32)** (À venir)

- Syntaxe du langage C32.
- Structures de contrôle et types.

10. **Chapitre 9 : Système d'Exploitation** (À venir)

- Gestion de la mémoire (Heap).
- Entrées/Sorties (Clavier, Écran).
- *Exercices* : os_lib.

1.2 Comment utiliser ce guide

Chaque chapitre commence par une **Théorie**, expliquant les concepts fondamentaux. Ensuite, la section **Implémentation** détaille ce que vous devez construire, en faisant référence aux dossiers du projet.

Note : Ce projet utilise Rust pour l'outillage, mais votre travail consistera principalement à écrire du HDL (Hardware Description Language), de l'assembleur A32 et du C32.

2 Introduction

Bienvenue dans le projet **Codex**. Si vous lisez ceci, c'est que vous avez l'ambition de comprendre comment fonctionnent les ordinateurs — non pas en lisant des théories abstraites, mais en en construisant un vous-même, de zéro.

2.1 Le Mystère de l'Ordinateur

Prenez un instant pour réfléchir à ce qui se passe quand vous tapez une lettre sur votre clavier :

1. Vos doigts appuient sur une touche physique
2. Un signal électrique est envoyé
3. Ce signal est transformé en code numérique
4. Le processeur le détecte et l'interprète
5. Un programme décide quoi faire de cette information
6. Des pixels s'allument sur votre écran pour afficher la lettre

Entre votre doigt et le pixel, il y a des **dizaines de couches d'abstraction**. Chaque couche fait confiance à celle en dessous et simplifie la vie de celle au-dessus.

Ce livre va vous faire traverser **toutes ces couches**, de la plus basse (les portes logiques) à la plus haute (les applications).

2.2 Pourquoi Construire un Ordinateur ?

2.2.1 Le problème de la “boîte noire”

Dans notre vie quotidienne, l'ordinateur est une “boîte noire”. Nous tapons sur un clavier, touchons un écran, et la magie opère. En tant qu'ingénieurs logiciels, nous travaillons souvent sur des couches d'abstraction très élevées (Python, Java, React).

Cette abstraction est une bénédiction pour la productivité, mais elle crée un **fossé de compréhension**. Combien de développeurs savent vraiment : - Comment le processeur exécute leur code ? - Pourquoi certaines opérations sont rapides et d'autres lentes ? - Ce qui se passe quand on écrit $x = 5$? - Comment une image apparaît à l'écran ?

2.2.2 L'approche "Du NAND au Tetris"

Ce projet a pour but de **briser l'abstraction**. Nous allons descendre au niveau le plus bas — la porte logique — et remonter couche par couche jusqu'à pouvoir jouer à un jeu vidéo écrit dans un langage de haut niveau sur notre propre machine.

À la fin de ce parcours, quand vous verrez du code s'exécuter, vous saurez **exactement** ce qui se passe dans la machine. Ce n'est plus de la magie — c'est de l'ingénierie que vous maîtrisez.

2.3 Ce que Vous Allez Construire

Voici les couches que nous allons traverser, de bas en haut :

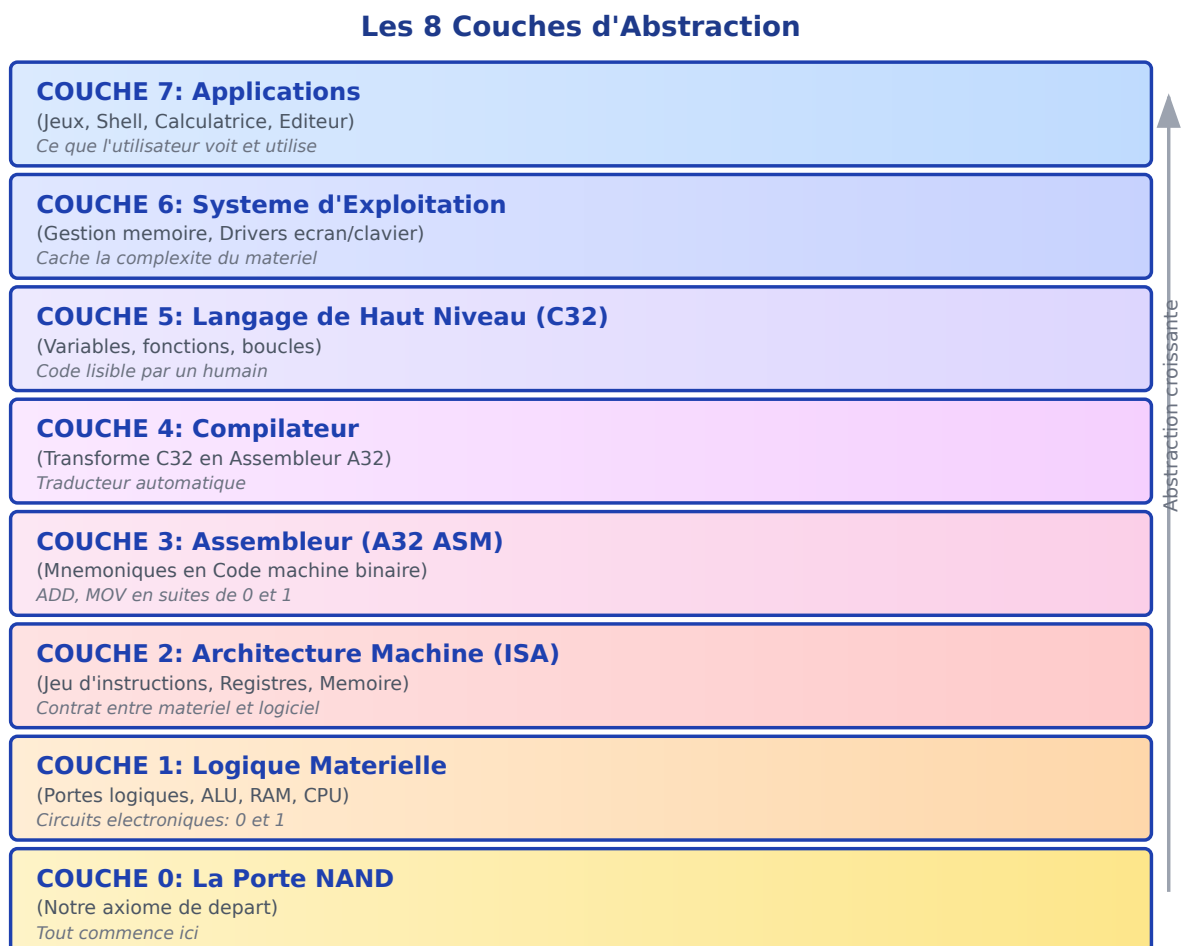


Figure 2.1: Les 8 couches d'abstraction

2.3.1 La beauté de l'abstraction

Chaque couche a une propriété remarquable : **elle n'a besoin de connaître que la couche juste en dessous.**

- Le programmeur C32 n'a pas besoin de savoir comment fonctionne l'ALU
- L'ALU n'a pas besoin de savoir qu'elle va être utilisée pour un jeu vidéo
- La porte NAND ne "sait" pas qu'elle fait partie d'un ordinateur

Cette séparation des préoccupations est ce qui rend possible la construction de systèmes complexes.

2.4 L'Architecture Codex A32

L'ordinateur **Codex** est conçu pour être à la fois pédagogique et réaliste. Il s'inspire des architectures ARM modernes tout en restant accessible.

Comparé aux architectures pédagogiques classiques plus simples (comme Hack), Codex offre :

Caractéristique	Hack (Original)	Codex (Ce projet)
Architecture	16-bits	32-bits
Registres	2 (A et D)	16 (R0-R15) style ARM
Mémoire	Séparée (Harvard)	Unifiée (Von Neumann)
Instructions	Simple, propriétaire	RISC moderne (Load/Store)
Écran	Monochrome fixe	320×240 monochrome

2.4.1 Pourquoi ces changements ?

1. **32 bits** : C'est la taille standard des machines modernes (avant 64 bits). Cela permet d'adresser 4 Go de mémoire et de manipuler des nombres plus grands.
2. **16 registres** : Les processeurs ARM (smartphones, Raspberry Pi) utilisent aussi des registres R0-R15. Comprendre Codex, c'est comprendre ARM.
3. **Architecture RISC** : Les instructions sont simples et régulières. Le CPU fait une chose à la fois, mais le fait vite.
4. **Load/Store** : Le CPU ne calcule jamais directement en mémoire. Il charge d'abord les données dans des registres, calcule, puis stocke le résultat. C'est plus simple à implémenter et à comprendre.

2.5 Ce que Vous Allez Apprendre

À la fin de ce livre, vous saurez :

Au niveau matériel : - Comment construire des portes logiques à partir de NAND
- Comment un additionneur transforme des bits en nombres - Comment la mémoire “se souvient” des données - Comment le CPU orchestre tout cela

Au niveau logiciel : - Comment l'assembleur traduit les mnémoniques en binaire
- Comment un compilateur transforme du code lisible en instructions - Comment un OS simplifie l'accès au matériel - Comment une application utilise toutes ces couches

Au niveau conceptuel : - Pourquoi les ordinateurs utilisent le binaire - Comment l'abstraction permet de gérer la complexité - Pourquoi certaines opérations sont “coûteuses” - Comment le même matériel peut faire des choses très différentes

2.6 Vos Outils

Le projet est fourni avec une suite d'outils performants :

2.6.1 Les Outils en Ligne de Commande

Outil	Rôle	Exemple
hdl_cli	Simule vos circuits HDL	hdl_cli test Not.hdl
a32_cli	Assemble le code A32 → binaire	a32_cli prog.s -o prog.bin
c32_cli	Compile le code C32 → assembleur	c32_cli prog.c -o prog.s
a32_runner	Exécute le code binaire	a32_runner prog.bin

2.6.2 Le Simulateur Web (Recommandé)

Pour une expérience plus visuelle et interactive, utilisez le **Simulateur Web**. Il vous permet de : - Écrire et tester votre HDL directement dans le navigateur - Voir l'état des signaux en temps réel avec des chronogrammes - Compiler et exécuter du code C et Assembleur - Visualiser l'écran, les registres et la mémoire

Pour le lancer :

```
cd web
npm install
npm run dev
```

Ouvrez ensuite votre navigateur à l'adresse indiquée (généralement <http://localhost:5173>).

2.6.3 Le CPU Visualizer

Le **CPU Visualizer** est un outil pédagogique spécialement conçu pour comprendre le fonctionnement du processeur. Il affiche en temps réel :

- **Le pipeline** : Les 5 étapes d'exécution (Fetch, Decode, Execute, Memory, Writeback) s'illuminent au fur et à mesure
- **Les registres** : R0-R15 avec mise en évidence des modifications
- **Les flags** : N, Z, C, V avec animations lors des changements
- **Le code source** : Coloration syntaxique et surlignage de la ligne en cours d'exécution
- **Le cache** : Statistiques (hits/misses) et contenu des lignes cache

Accédez-y depuis le menu principal du Simulateur Web ou directement via </visualizer.html>.

7 démos intégrées vous permettent d'explorer différents concepts : 1. Addition simple 2. Boucles et branchements 3. Accès mémoire (LDR/STR) 4. Conditions et prédication 5. Tableaux 6. Flags CPU 7. Comportement du cache

2.7 Comment Utiliser ce Livre

2.7.1 L'approche recommandée

1. **Lisez chaque chapitre en entier** avant de commencer les exercices
2. **Faites les exercices dans l'ordre** — chaque exercice prépare le suivant
3. **Ne regardez pas les solutions** avant d'avoir vraiment essayé
4. **Utilisez le simulateur web** pour visualiser ce qui se passe
5. **Reliez toujours à l'ensemble** — demandez-vous "où cela s'insère-t-il ?"

2.7.2 Si vous êtes bloqué

- Relisez la section correspondante du chapitre
- Vérifiez que vous avez bien compris les exercices précédents
- Utilisez le débogueur visuel du simulateur web
- Les erreurs les plus fréquentes sont des problèmes de câblage (mauvaises connexions)

2.8 La Grande Aventure Commence

Vous êtes sur le point d'entreprendre un voyage fascinant. Chaque chapitre vous rapprochera un peu plus de la compréhension totale de la machine.

Quand vous aurez terminé, vous regarderez votre ordinateur différemment. Ce ne sera plus une boîte noire mystérieuse, mais une symphonie d'abstractions que vous pouvez comprendre, modifier, et même reconstruire.

Prêt ? Passons à la première brique élémentaire : la logique booléenne.

Rappel important : Chaque chapitre de ce livre construit sur les précédents. Résistez à la tentation de sauter des étapes — la compréhension profonde vient de la construction progressive.

3 Logique Booléenne

“Au commencement était le NAND.”

Tout ordinateur numérique, aussi complexe soit-il, est construit à partir de concepts incroyablement simples : le Vrai (1) et le Faux (0). Ce chapitre traite de la construction de portes logiques élémentaires à partir d’une brique fondamentale : la porte NAND.

3.1 Où en sommes-nous ?

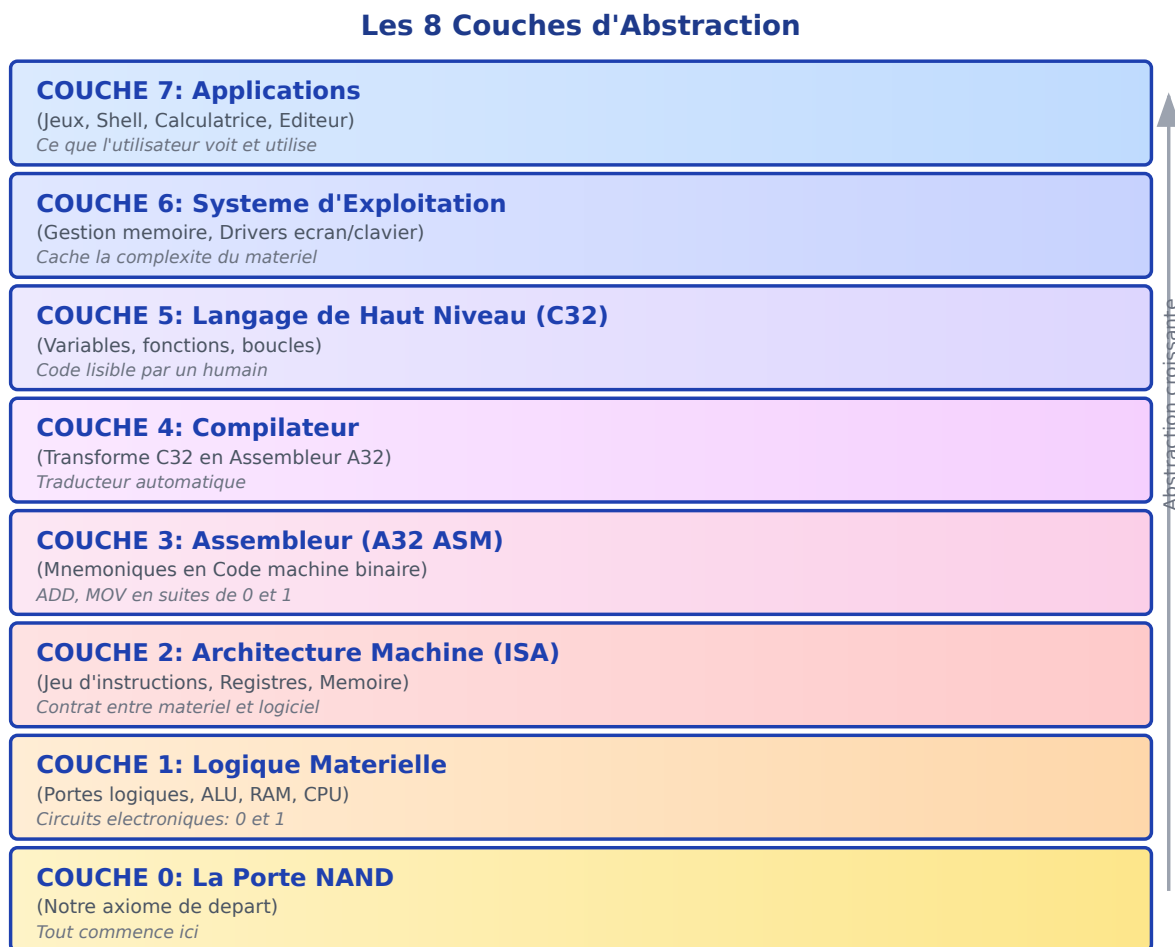


Figure 3.1: Position dans l'architecture

Nous sommes à la Couche 0 : La Porte NAND - Notre axiome de départ

Nous commençons tout en bas de la pyramide. C'est ici que nous posons les fondations de tout l'édifice. Chaque porte que vous construirez dans ce chapitre sera utilisée dans les chapitres suivants pour construire des circuits de plus en plus complexes, jusqu'au CPU complet.

3.2 Pourquoi le Binaire ?

3.2.1 La question fondamentale

Avant de construire des portes logiques, posons-nous une question essentielle : **pourquoi les ordinateurs utilisent-ils le binaire (0 et 1) plutôt que le**

systeme decimal (0-9) que nous utilisons au quotidien ?

La réponse est d'ordre **physique et pratique** :

1. **Fiabilité** : Distinguer entre deux états (tension haute/basse, courant/pas courant) est beaucoup plus fiable que de distinguer entre dix niveaux différents. Le bruit électrique peut facilement transformer un "7" en "8", mais rarement un "1" en "0".
2. **Simplicité** : Les circuits qui ne gèrent que deux états sont beaucoup plus simples à concevoir et à fabriquer. Un transistor peut facilement agir comme un interrupteur (on/off).
3. **Universalité** : George Boole a prouvé au 19ème siècle que toute la logique peut être exprimée avec seulement deux valeurs : Vrai et Faux.

3.2.2 Du voltage au bit

Dans le monde physique, nos ordinateurs utilisent des tensions électriques :

Tension	Signification logique
0V - 0.8V	0 (Faux)
2.4V - 3.3V	1 (Vrai)

La zone entre 0.8V et 2.4V est une "zone interdite" — les circuits sont conçus pour ne jamais s'y trouver de manière stable. C'est cette séparation nette qui rend le binaire si robuste.

3.2.3 L'abstraction qui libère

En tant qu'architectes système, nous n'avons pas besoin de penser aux voltages, aux électrons, ou aux transistors. Nous travaillons avec des **abstractions** : - Un **bit** peut valoir 0 ou 1 - Une **fonction booléenne** transforme des bits en d'autres bits

C'est la première de nombreuses couches d'abstraction que nous allons traverser. Chaque couche nous permet de penser à un niveau supérieur sans nous soucier des détails de la couche inférieure.

3.3 La Porte NAND : Notre Axiome

3.3.1 Pourquoi partir du NAND ?

Il existe de nombreuses portes logiques (AND, OR, NOT, XOR...), mais nous choisissons de tout construire à partir d'une seule : la porte **NAND** (Not-AND).

Pourquoi ce choix ?

1. **Complétude fonctionnelle** : Le NAND est dit "fonctionnellement complet" — on peut construire TOUTES les autres portes logiques à partir de NAND uniquement. C'est mathématiquement prouvé !
2. **Réalité industrielle** : La technologie CMOS (Complementary Metal-Oxide-Semiconductor), utilisée dans tous les processeurs modernes, implémente naturellement les portes NAND de manière très efficace — seulement 4 transistors.
3. **Pédagogie** : Partir d'une seule brique force à comprendre comment les abstractions se construisent les unes sur les autres.

3.3.2 Table de Vérité NAND

La porte NAND prend deux entrées (A et B) et produit une sortie. Son comportement :

A	B	NAND(A, B)
0	0	1
0	1	1
1	0	1
1	1	0

Règle simple : Le résultat est 0 *seulement si* A **et** B sont tous les deux à 1. Dans tous les autres cas, c'est 1.

Le nom "NAND" vient de "Not-AND" : c'est l'inverse exact d'une porte AND.

3.3.3 Symbole graphique

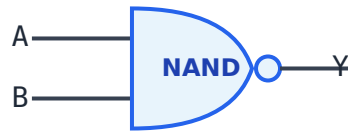


Figure 3.2: Porte NAND

Le petit cercle (O) indique l'inversion

3.4 Construction des Portes Élémentaires

Votre mission dans ce chapitre est de construire les portes suivantes, en utilisant **uniquement** des NAND (et les portes que vous aurez déjà créées).

3.4.1 A. NOT (Inverseur) — L'inversion de la réalité

Rôle dans l'ordinateur : L'inverseur est fondamental. Il permet de créer des conditions négatives ("si PAS égal..."), de complémenter des nombres (pour la soustraction), et d'implémenter des bascules (pour la mémoire).

Spécification :

in	out
0	1
1	0

Le défi : Comment utiliser une porte NAND (qui prend 2 entrées) pour n'en traiter qu'une seule ?

L'astuce : Connectez le même signal aux deux entrées !



Figure 3.3: NOT construit à partir de NAND

Vérifions avec la table de vérité :

- Si $in = 0$: $NAND(0, 0) = 1$ OK
- Si $in = 1$: $NAND(1, 1) = 0$ OK

C'est exactement le comportement NOT !

3.4.2 B. AND (Et) — La conjonction

Rôle dans l'ordinateur : La porte AND est essentielle pour : - Extraire des bits spécifiques d'un nombre (masquage) - Vérifier que TOUTES les conditions sont vraies - Contrôler le passage de données (avec un signal d'activation)

Spécification :

A	B	AND(A, B)
0	0	0
0	1	0
1	0	0
1	1	1

La sortie vaut 1 seulement si A ET B valent 1.

L'insight : Le NAND est un "Not-AND". Si on inverse le résultat d'un NAND... on obtient un AND !

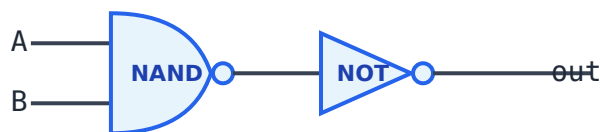


Figure 3.4: AND construit à partir de NAND et NOT

Formule : $AND(A, B) = NOT(NAND(A, B))$

3.4.3 C. OR (Ou) — L'alternative

Rôle dans l'ordinateur : La porte OR permet de : - Combiner plusieurs signaux de contrôle - Vérifier qu'AU MOINS UNE condition est vraie - Créer des priorités (interruptions, erreurs)

Spécification :

A	B	OR(A, B)
0	0	0
0	1	1
1	0	1
1	1	1

La sortie vaut 1 si au moins une entrée vaut 1.

Le Théorème de De Morgan : Ce théorème fondamental nous donne la clé :

$$A \text{ OR } B = \text{NOT} ((\text{NOT } A) \text{ AND } (\text{NOT } B))$$

En utilisant uniquement des NAND :

$$A \text{ OR } B = (\text{NOT } A) \text{ NAND } (\text{NOT } B)$$

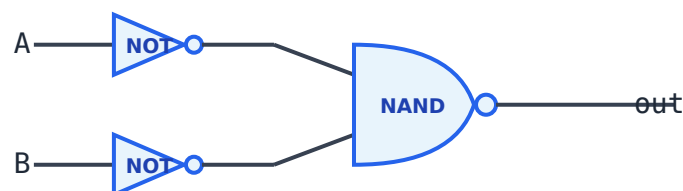


Figure 3.5: OR construit à partir de NOT et NAND

3.4.4 D. XOR (Ou Exclusif) — La différence

Rôle dans l'ordinateur : XOR est crucial pour : - **L'addition binaire** : XOR calcule la somme de deux bits (sans la retenue) - **La comparaison** : XOR détecte si deux bits sont différents - **Le cryptage** : XOR est au cœur de nombreux algorithmes de chiffrement - **La détection d'erreurs** : Calcul de parité

Spécification :

A	B	XOR(A, B)
0	0	0
0	1	1
1	0	1
1	1	0

La sortie vaut 1 si les entrées sont *différentes*.

Formule algébrique :

$$\text{XOR}(A, B) = (A \text{ AND NOT } B) \text{ OR } (\text{NOT } A \text{ AND } B)$$

En mots : “A est vrai et B est faux” OU “A est faux et B est vrai”.

3.4.5 E. Multiplexeur (Mux) — L'aiguilleur

Rôle dans l'ordinateur : Le multiplexeur est l'un des composants les plus importants ! Il permet de :

- **Choisir entre plusieurs sources de données :** Comme un aiguillage de train
- **Implémenter les conditions :** if (sel) then b else a
- **Construire la mémoire :** Sélectionner quelle cellule lire

Spécification :

- Si sel == 0 alors out = a
- Si sel == 1 alors out = b

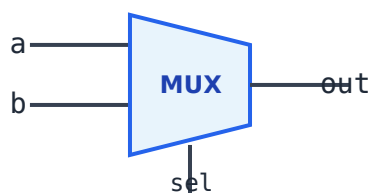


Figure 3.6: Multiplexeur

L'insight : On peut voir le Mux comme : “soit (a ET non-sel) soit (b ET sel)”.

$$\text{out} = (a \text{ AND NOT } \text{sel}) \text{ OR } (b \text{ AND } \text{sel})$$

Pourquoi le Mux est-il si important ?

Imaginez que vous construisez un CPU. À chaque cycle, le CPU doit choisir :

- D'où vient l'opérande ? De la mémoire ou d'un registre ?
- Où va le résultat ? Vers la mémoire ou un registre ?
- Quelle instruction exécuter ?

Chacun de ces choix est implémenté par un multiplexeur !

3.4.6 F. Démultiplexeur (DMux) — L'inverse de l'aiguilleur

Rôle dans l'ordinateur : Le DMux fait l'inverse du Mux — il prend UNE entrée et la dirige vers UNE des sorties possibles. Utile pour :

- **L'adressage mémoire** : Activer la bonne cellule
- **La distribution de signaux** : Envoyer une commande au bon périphérique

Spécification :

- Si $sel == 0$ alors $a = in$, $b = 0$
- Si $sel == 1$ alors $a = 0$, $b = in$

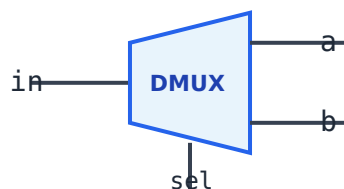


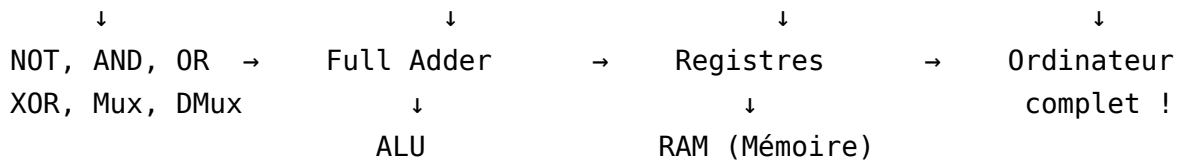
Figure 3.7: Démultiplexeur

3.5 Le Lien avec l'Ordinateur Complet

Prenons du recul. Pourquoi construisons-nous ces portes ?

3.5.1 Du NAND au CPU : La feuille de route

CHAPITRE 1 (Ici)	CHAPITRE 2	CHAPITRE 3	CHAPITRE 5
↓	↓	↓	↓
NAND	Half Adder	DFF (Flip-Flop)	CPU



Chaque porte a un rôle précis :

- **NOT** : Permet la soustraction (via le complément à 2)
- **AND** : Masquage de bits, conditions ET
- **OR** : Combinaison de signaux, conditions OU
- **XOR** : Addition bit à bit, comparaisons
- **Mux** : Tous les choix du CPU (quelle donnée ? quelle opération ?)
- **DMux** : Adressage mémoire, routage des résultats

Quand vous jouerez à un jeu sur votre ordinateur Codex au Chapitre 9, chaque pixel affiché à l'écran aura été calculé par des millions d'opérations utilisant ces portes élémentaires !

3.6 Description Matérielle (Codex HDL)

Pour décrire nos circuits, nous utilisons un langage appelé **HDL** (Hardware Description Language). C'est un langage **déclaratif** : on décrit QUELS composants existent et COMMENT ils sont connectés, pas DANS QUEL ORDRE les exécuter.

3.6.1 Pourquoi un langage de description ?

En électronique réelle, on dessine des schémas. Mais les schémas deviennent illisibles pour des circuits complexes (un CPU contient des millions de portes !). Le HDL permet de :

1. **Décrire** des circuits de manière textuelle
2. **Simuler** leur comportement avant fabrication
3. **Synthétiser** le circuit vers du matériel réel

Notre HDL est inspiré du **VHDL** (Very High-Speed Integrated Circuit HDL), utilisé dans l'industrie.

3.6.2 Structure d'un fichier .hdl

Un fichier HDL se compose de deux parties :

```

-- =====
-- 1. L'ENTITÉ : L'interface externe (les "broches" de la puce)
-- =====
entity NomDeLaPuce is
  port(
    a : in bit;      -- Entrée de 1 bit
    b : in bit;      -- Autre entrée
    y : out bit      -- Sortie (pas de point-virgule sur la dernière)
  );
end entity;

-- =====
-- 2. L'ARCHITECTURE : Le contenu interne (le câblage)
-- =====
architecture rtl of NomDeLaPuce is
  -- Déclaration des composants utilisés
  component Nand
    port(a : in bit; b : in bit; y : out bit);
  end component;

  -- Déclaration des signaux internes (fils)
  signal fil_interne : bit;

begin
  -- Instanciation et connexion des composants
  u1: Nand port map (a => a, b => b, y => fil_interne);

  -- Assignment directe (optionnel)
  y <= fil_interne;

end architecture;

```

3.6.3 Vocabulaire essentiel

Terme	Signification
entity	L'interface externe — quelles sont les entrées/sorties
architecture	Le contenu — comment c'est câblé à l'intérieur
component	Une puce qu'on veut utiliser (il faut la déclarer)
signal	Un fil interne (pour connecter des composants entre eux)
port map	"Brancher" les fils aux broches d'un composant
=>	"est connecté à" (broche => signal)
<=	Assignment (sortie <= signal)

3.6.4 Règles de connexion

1. **À gauche du =>** : Le nom de la broche du composant que vous utilisez
2. **À droite du =>** : Le signal de VOTRE architecture que vous y connectez
3. **Les signaux doivent être déclarés** avant le begin
4. **Chaque instance a un nom unique** (u1, u2, etc.)

3.6.5 Bus (Vecteurs de bits)

Pour manipuler plusieurs bits simultanément (ex: un nombre de 32 bits), on utilise bits(MSB downto LSB) :

```
-- Un bus de 32 bits
data_bus : in bits(31 downto 0);

-- Accéder à un bit spécifique
data_bus(0) -- Le bit de poids faible (LSB)
data_bus(31) -- Le bit de poids fort (MSB)

-- Extraire une tranche
data_bus(7 downto 0) -- Les 8 bits de poids faible
```

3.6.6 Exemple complet : XOR en HDL

```
-- La porte XOR : sortie = 1 si les entrées sont différentes
-- XOR(a,b) = (a AND NOT b) OR (NOT a AND b)

entity Xor2 is
  port(
    a : in bit;
    b : in bit;
    y : out bit
  );
end entity;

architecture rtl of Xor2 is
  -- Composants utilisés
  component Inv
    port(a : in bit; y : out bit);
  end component;
  component And2
    port(a : in bit; b : in bit; y : out bit);
  end component;
  component Or2
    port(a : in bit; b : in bit; y : out bit);
  end component;
```

```

end component;

-- Signaux internes
signal not_a, not_b : bit; -- Les inversions de a et b
signal w1, w2 : bit;      -- Résultats intermédiaires

begin
  -- NOT a et NOT b
  u_nota: Inv port map (a => a, y => not_a);
  u_notb: Inv port map (a => b, y => not_b);

  -- (a AND NOT b)
  u_and1: And2 port map (a => a, b => not_b, y => w1);

  -- (NOT a AND b)
  u_and2: And2 port map (a => not_a, b => b, y => w2);

  -- Résultat final : OR des deux termes
  u_or: Or2 port map (a => w1, b => w2, y => y);

end architecture;

```

3.7 Exercices Pratiques

3.7.1 Exercices sur le Simulateur Web

Le **Simulateur Web** vous permet de construire et tester vos portes de manière interactive. Lancez-le et allez dans la section **HDL Progression**.

Exercice	Description	Difficulté
Inv	Inverseur (NOT) — Votre première porte	[*]
And2	Porte AND à 2 entrées	[*]
Or2	Porte OR à 2 entrées	[*]
Xor2	Porte XOR (Ou exclusif)	[**]
Mux	Multiplexeur 2 vers 1	[**]
DMux	Démultiplexeur 1 vers 2	[**]

Pour chaque exercice :

1. Lisez la spécification et la table de vérité

2. Réfléchissez à comment combiner les portes disponibles
3. Écrivez votre code HDL
4. Testez — le simulateur vous montrera un chronogramme des signaux
5. Si un test échoue, analysez quelle entrée produit le mauvais résultat

3.7.2 Comment lancer le simulateur web ?

```
cd web
npm install    # (première fois uniquement)
npm run dev
```

Puis ouvrez votre navigateur à l'adresse indiquée (généralement `http://localhost:5173`).

3.7.3 Alternative : Tests en ligne de commande

Si vous préférez travailler dans les fichiers du répertoire `hdl_lib/01_gates/` :

```
# Tester une porte spécifique
cargo run -p hdl_cli -- test hdl_lib/01_gates/Not.hdl

# Tester toutes les portes du projet 1
cargo run -p hdl_cli -- test hdl_lib/01_gates/
```

3.8 Défis Supplémentaires

3.8.1 Défi 1 : Minimiser le nombre de NAND

Construisez la porte XOR en utilisant **seulement 4 portes NAND** (c'est le minimum théorique !). La solution classique en utilise 5.

3.8.2 Défi 2 : Implémenter IMPLIES

La fonction "implication" ($A \rightarrow B$) vaut FAUX seulement si A est VRAI et B est FAUX.

A	B	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

Construisez cette porte en utilisant les portes élémentaires.

3.8.3 Défi 3 : Mux à 4 entrées

Construisez un Mux qui choisit parmi 4 entrées (a, b, c, d) avec 2 bits de sélection (sel[1:0]).

3.9 Ce qu'il faut retenir

1. **Le binaire simplifié** : Deux états sont plus fiables que dix
2. **NAND est universel** : Toutes les portes peuvent être construites à partir de NAND
3. **L'abstraction est puissante** : On construit des couches les unes sur les autres
4. **Chaque porte a un rôle** :
 - NOT → Inversion, complément
 - AND → Masquage, condition "et"
 - OR → Combinaison, condition "ou"
 - XOR → Addition, comparaison
 - Mux → Choix, sélection
 - DMux → Routage, adressage

Prochaine étape : Au Chapitre 2, nous utiliserons ces portes pour construire des circuits qui font de l'**arithmétique** — addition, soustraction, et une ALU (Unité Arithmétique et Logique) complète.

Conseil : Ne passez pas au chapitre suivant avant d'avoir réussi tous les exercices de ce chapitre. Chaque porte que vous construisez sera réutilisée dans les chapitres suivants !

3.10 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

3.10.1 Questions de compréhension

- Q1.** Pourquoi les ordinateurs utilisent-ils le binaire plutôt que le décimal ?
- Q2.** Pourquoi dit-on que NAND est une porte “universelle” ?
- Q3.** Quelle est la sortie de $XOR(1, 1)$? Et de $XOR(0, 1)$?
- Q4.** Un multiplexeur (MUX) a 2 entrées de données a et b, et un signal de sélection sel. Si $sel = 1$, quelle entrée est transmise en sortie ?
- Q5.** Combien de portes NAND faut-il au minimum pour construire un inverseur (NOT) ?

3.10.2 Mini-défi pratique

Complétez cette table de vérité pour la fonction $F(a, b) = AND(OR(a, b), NOT(a))$:

a	b	OR(a,b)	NOT(a)	F
0	0	?	?	?
0	1	?	?	?
1	0	?	?	?
1	1	?	?	?

Les solutions se trouvent dans le document **Codex_Solutions**.

3.10.3 Checklist de validation

Avant de passer au chapitre 2, assurez-vous de pouvoir :

- ☐ Expliquer pourquoi NAND est universel
- ☐ Construire NOT, AND, OR à partir de NAND
- ☐ Lire et écrire une table de vérité
- ☐ Comprendre le rôle du MUX (sélection) et du DMUX (routage)
- ☐ Écrire du code HDL simple avec port map

4 Arithmétique Binaire

“Les mathématiques sont le langage avec lequel Dieu a écrit l’univers.” —
Galilée

Dans le chapitre précédent, nous avons appris à manipuler des bits individuels avec des portes logiques. Mais un ordinateur doit savoir compter ! Comment passer de simples portes logiques à une calculatrice capable d’additionner des nombres à 32 bits ?

4.1 Où en sommes-nous ?

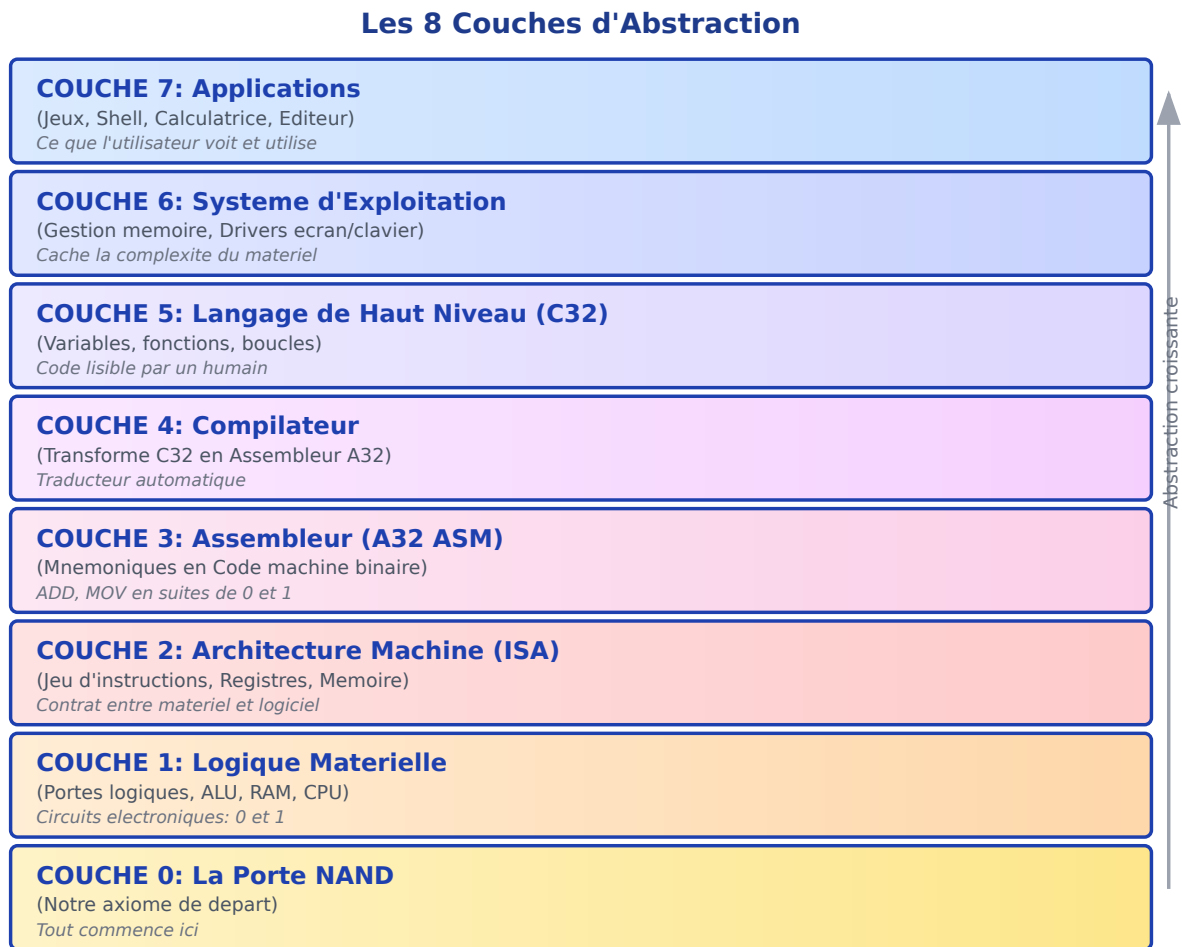


Figure 4.1: Position dans l'architecture

Nous sommes à la Couche 1 : Logique Matérielle (Portes logiques - ALU, RAM, CPU)

Nous sommes toujours dans la couche matérielle, mais nous montons d'un niveau. Nous allons combiner les portes logiques du Chapitre 1 pour construire des circuits arithmétiques, culminant avec l'**ALU** (Arithmetic Logic Unit) — le composant qui effectue TOUS les calculs du processeur.

4.2 Pourquoi l'Arithmétique est-elle si Importante ?

4.2.1 Au cœur de tout calcul

Regardez ce que fait un ordinateur :

- **Afficher une image** : Calculer la couleur de chaque pixel (additions, multiplications)
- **Jouer un son** : Mélanger des formes d'onde (additions)
- **Naviguer sur le web** : Calculer des checksums, décompresser des données
- **Exécuter un programme** : Calculer l'adresse de la prochaine instruction (addition)

Même les opérations les plus “abstraites” se réduisent finalement à des opérations arithmétiques sur des nombres binaires. L'ALU que vous allez construire est le moteur qui fait tourner TOUT.

4.2.2 Ce que nous allons construire

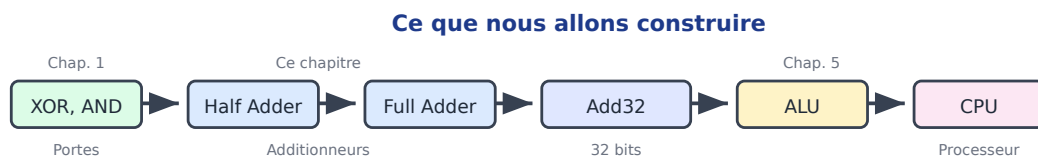


Figure 4.2: Feuille de route : des portes à l'ALU

À la fin de ce chapitre, vous aurez construit une ALU capable d'effectuer : - Addition et soustraction - ET, OU, XOR logiques - Comparaisons (via les drapeaux)

4.3 Représentation des Nombres

4.3.1 Le Système Binaire (Base 2)

Avant de construire des additionneurs, comprenons comment les nombres sont représentés.

En décimal (base 10), chaque position représente une puissance de 10 :

$$\begin{array}{ccc}
 4 & 2 & 7 \\
 \downarrow & \downarrow & \downarrow \\
 10^2 & 10^1 & 10^0 \rightarrow 4 \times 100 + 2 \times 10 + 7 \times 1 = 427
 \end{array}$$

En binaire (base 2), chaque position représente une puissance de 2 :

Position :	3	2	1	0
Poids :	2^3	2^2	2^1	2^0

Valeur : 8 4 2 1

Exemple : $1011_2 = 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 = 11_{10}$

4.3.2 Taille des nombres dans Codex

Notre ordinateur Codex travaille sur **32 bits**. Cela signifie : - **Plage non-signée** : 0 à $2^{32} - 1 = 4\,294\,967\,295$ (≈ 4 milliards) - **Plage signée** : -2 147 483 648 à 2 147 483 647 ($\approx \pm 2$ milliards)

C'est suffisant pour : - Adresser 4 Go de mémoire (chaque octet a une adresse unique) - Représenter des coordonnées d'écran, des scores de jeux, des compteurs

4.3.3 Les Nombres Négatifs : Le Complément à 2

Comment représenter des nombres négatifs avec seulement des 0 et des 1 ?

Le problème : On pourrait utiliser un bit de signe (0 = positif, 1 = négatif), mais alors on aurait besoin de circuits différents pour l'addition et la soustraction, et on aurait deux représentations du zéro (+0 et -0).

La solution brillante : Le **Complément à 2**.

Le bit le plus à gauche (bit 31, le MSB) est le "bit de signe" : - 0 \rightarrow le nombre est positif ou nul - 1 \rightarrow le nombre est négatif

Mais attention, ce n'est pas un simple bit de signe ! Le système est conçu pour que **l'addition fonctionne de la même manière** que le nombre soit positif ou négatif.

4.3.4 Comment obtenir le complément à 2 (la valeur négative) ?

Pour obtenir -X à partir de X : 1. **Inverser** tous les bits de X (0 \rightarrow 1, 1 \rightarrow 0) 2. **Ajouter 1** au résultat

Exemple sur 4 bits : Calculons -5

5 en binaire :	0101
Inversion :	1010
Ajouter 1 :	+ 0001
	<hr/>
-5 :	1011

Vérification : 5 + (-5) devrait donner 0

$$\begin{array}{r} 0101 \quad (5) \\ + 1011 \quad (-5) \\ \hline \end{array}$$

10000 → Les 4 bits de poids faible sont 0000 OK
(La retenue "1" est ignorée car on travaille sur 4 bits)

4.3.5 Pourquoi le complément à 2 est-il génial ?

1. **Un seul zéro** : 0000 est le seul zéro (pas de +0 et -0)
2. **L'addition fonctionne universellement** : Le même circuit additionne les positifs et les négatifs
3. **La soustraction devient une addition** : $A - B = A + (-B) = A + \text{NOT}(B) + 1$

C'est grâce au complément à 2 que notre ALU peut être relativement simple !

4.4 L'Addition Binaire

L'addition binaire suit les mêmes règles que l'addition décimale qu'on apprend à l'école : on additionne colonne par colonne, de droite à gauche, en propageant les retenues.

4.4.1 Les règles de base (sur 1 bit)

$0 + 0 = 0$ (pas de retenue)
 $0 + 1 = 1$ (pas de retenue)
 $1 + 0 = 1$ (pas de retenue)
 $1 + 1 = 10$ (c'est-à-dire 0 avec une retenue de 1)

4.4.2 Exemple d'addition sur 4 bits

Calculons $5 + 3 = 8$:

$$\begin{array}{r} \text{Retenues :} \quad 1 \ 1 \ 1 \\ \hline \begin{array}{r} 5 \quad : \quad 0 \ 1 \ 0 \ 1 \\ + \ 3 \quad : \quad + \ 0 \ 0 \ 1 \ 1 \\ \hline 8 \quad : \quad 1 \ 0 \ 0 \ 0 \end{array} \end{array}$$

Détail colonne par colonne (de droite à gauche) :

- Colonne 0 : $1 + 1 = 0$, retenue 1
- Colonne 1 : $0 + 1 + 1(\text{retenue}) = 0$, retenue 1
- Colonne 2 : $1 + 0 + 1(\text{retenue}) = 0$, retenue 1
- Colonne 3 : $0 + 0 + 1(\text{retenue}) = 1$

Résultat : $1000_2 = 8_{10}$ OK

4.5 Le Demi-Additionneur (Half Adder)

Le demi-additionneur est le circuit le plus simple pour additionner deux bits. Il produit : - **sum** : La somme (bit de poids faible) - **carry** : La retenue (bit de poids fort)

4.5.1 Table de vérité

a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

4.5.2 L'insight clé

Regardez attentivement les colonnes : - **sum** correspond exactement à **XOR(a, b)** — différent = 1, identique = 0 - **carry** correspond exactement à **AND(a, b)** — les deux à 1 = retenue

C'est pour cela que nous avons construit XOR et AND au Chapitre 1 !

4.5.3 Schéma du circuit

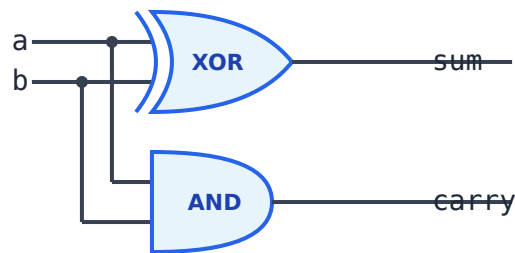


Figure 4.3: Demi-additionneur

4.5.4 Limitation

Le demi-additionneur ne peut pas recevoir de retenue d'une colonne précédente. Il ne fonctionne donc que pour le bit de poids faible (la première colonne).

4.6 L'Additionneur Complet (Full Adder)

Pour additionner des nombres de plusieurs bits, chaque colonne (sauf la première) doit pouvoir accepter une retenue venant de la colonne précédente.

4.6.1 Interface



Figure 4.4: Additionneur complet

4.6.2 Table de vérité

a	b	cin	sum	cout
0	0	0	0	0
0	0	1	1	0

a	b	cin	sum	cout
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

4.6.3 Comment le construire ?

Un Full Adder peut être construit avec **deux Half Adders et une porte OR** :

1. Le premier Half Adder additionne a et b
2. Le second Half Adder additionne le résultat avec cin
3. Si l'un des deux Half Adders produit une retenue, on a une retenue finale

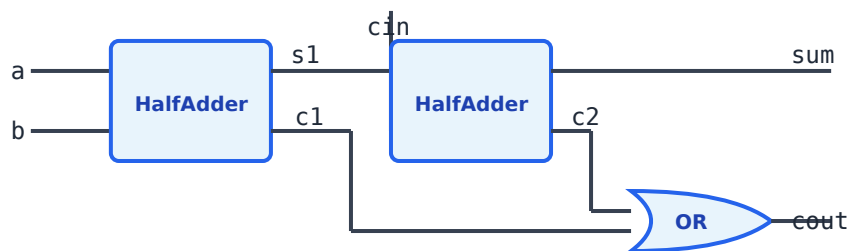


Figure 4.5: Construction du Full Adder

Formules :

- $s1 = \text{XOR}(a, b)$
- $\text{sum} = \text{XOR}(s1, \text{cin})$
- $c1 = \text{AND}(a, b)$
- $c2 = \text{AND}(s1, \text{cin})$
- $\text{cout} = \text{OR}(c1, c2)$

4.7 L'Additionneur 32-bits (Ripple Carry Adder)

Pour additionner des nombres de 32 bits, nous connectons 32 Full Adders en cascade. La retenue de sortie de chaque additionneur devient la retenue d'entrée

du suivant.

4.7.1 Schéma simplifié

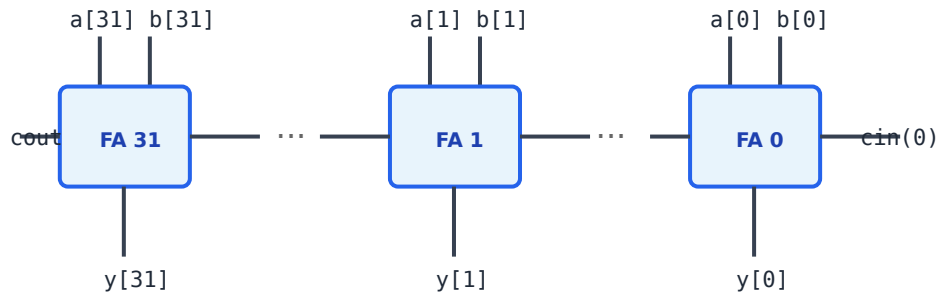


Figure 4.6: Additionneur 32 bits à propagation de retenue

Note : Le premier Full Adder (position 0) a une retenue d'entrée de 0 pour une addition normale. Mais on peut y injecter un 1 pour implémenter la soustraction ($A + \text{NOT}(B) + 1$).

4.7.2 Le compromis du Ripple Carry

Avantage : Très simple à comprendre et à implémenter.

Inconvénient : Les retenues se propagent d'un bout à l'autre. Pour 32 bits, la retenue doit traverser 32 étages. C'est lent !

Dans les vrais processeurs, on utilise des techniques comme le "Carry Lookahead Adder" pour accélérer la propagation. Mais pour notre projet pédagogique, le Ripple Carry est parfait.

4.8 L'ALU (Arithmetic Logic Unit)

L'ALU est le **cœur calculatoire** du processeur. C'est elle qui effectue TOUTES les opérations arithmétiques et logiques.

4.8.1 Pourquoi combiner arithmétique et logique ?

Plutôt que d'avoir des circuits séparés pour l'addition, la soustraction, le AND, le OR, etc., l'ALU combine tout en un seul composant. Un signal de contrôle (op) lui dit quelle opération effectuer.

4.8.2 Interface de l'ALU Codex

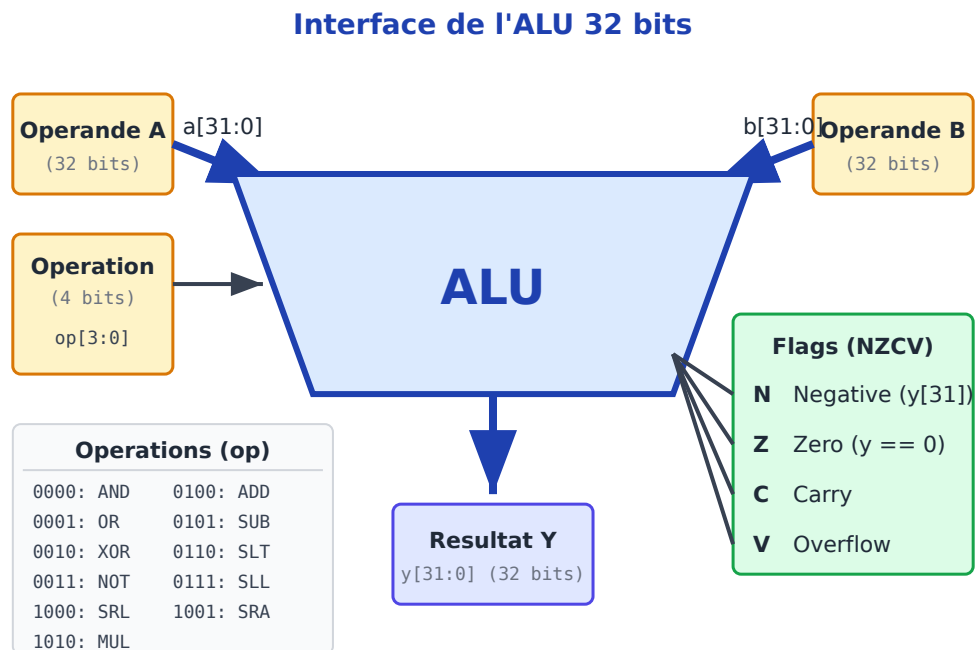


Figure 4.7: Interface de l'ALU

4.8.3 Les Opérations de l'ALU

Le signal op (4 bits) définit l'opération à effectuer :

op (binaire)	Nom	Opération	Description
0000	AND	$a \& b$	ET logique bit à bit
0001	EOR	$a \wedge b$	OU exclusif bit à bit
0010	SUB	$a - b$	Soustraction
0011	ADD	$a + b$	Addition
0100	ORR	$a \setminus b$	OU logique bit à bit
0101	MOV	b	Copie de b (ignore a)
0110	MVN	$\sim b$	Inversion de b (NOT)

4.8.4 Comment implémenter la soustraction ?

Grâce au complément à 2, la soustraction devient une addition :

$$A - B = A + (-B) = A + (\text{NOT } B) + 1$$

En pratique :

1. Inverser tous les bits de B (avec des portes NOT)
2. Additionner A et NOT(B) avec une retenue d'entrée de 1

C'est pour cela que notre additionneur a une entrée cin !

4.8.5 Les Drapeaux (Flags)

Les drapeaux sont des informations supplémentaires sur le résultat :

Drapeau	Nom	Signification
N	Negative	1 si le résultat est négatif (bit 31 = 1)
Z	Zero	1 si le résultat est exactement 0
C	Carry	1 s'il y a eu une retenue (dépassement non-signé)
V	Overflow	1 s'il y a eu un dépassement signé

À quoi servent ces drapeaux ?

Ils permettent au CPU de prendre des décisions :

- BEQ (Branch if Equal) teste si Z = 1
- BLT (Branch if Less Than) teste une combinaison de N et V
- BCS (Branch if Carry Set) teste si C = 1

Sans les drapeaux, il serait impossible d'implémenter les conditions if, les boucles while, etc. !

4.8.6 Comment calculer les drapeaux ?

- **N (Negative)** : C'est simplement le bit 31 du résultat
- **Z (Zero)** : NOR de tous les bits du résultat (tous à 0 ?)
- **C (Carry)** : La retenue de sortie de l'additionneur
- **V (Overflow)** : Se produit quand :
 - Deux positifs donnent un négatif
 - Deux négatifs donnent un positif
 - Formule : $V = (a[31] == b[31]) \text{ AND } (a[31] != y[31])$ (pour l'addition)

4.9 Architecture de l'ALU

Voici comment l'ALU est structurée en interne :

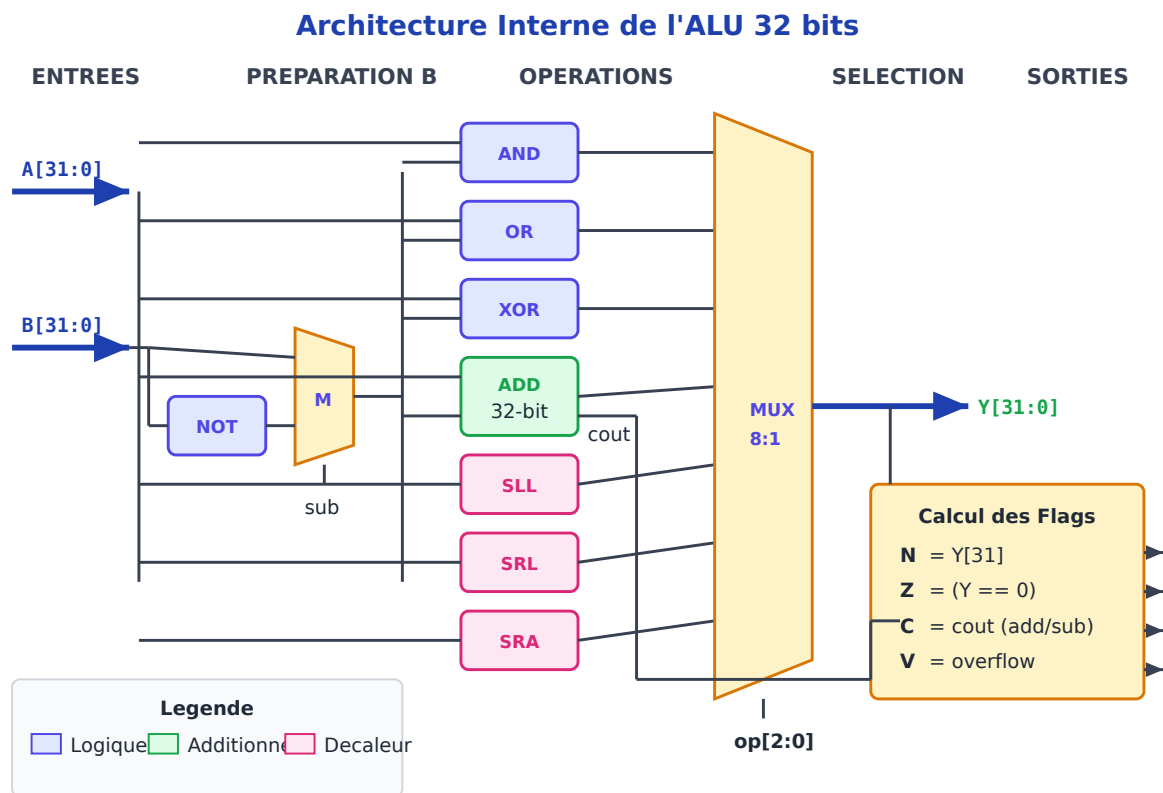


Figure 4.8: Architecture de l'ALU

L'idée clé : calculer TOUS les résultats possibles en parallèle, puis utiliser un multiplexeur pour sélectionner le bon selon op.

4.10 Exercices Pratiques

4.10.1 Exercices sur le Simulateur Web

Lancez le **Simulateur Web** et allez dans **HDL Progression** → **Projet 3 : Arithmétique**.

Exercice	Description	Difficulté
HalfAdder	Demi-additionneur (XOR + AND)	[*]
FullAdder	Additionneur complet (2 Half Adders + OR)	[**]

Exercice	Description	Difficulté
Add16	Additionneur 16 bits en cascade	[**]
Inc16	Incrémenteur (+1) — cas spécial utile	[*]
Sub16	Soustracteur (via complément à 2)	[**]
ALU	L'ALU complète avec drapeaux	[***]

4.10.2 Conseils pour l'ALU

1. **Commencez par les opérations simples** : AND, OR, XOR sont juste des portes appliquées bit à bit

2. **Pour la soustraction** :

$$\text{sub} = a + (\text{NOT } b) + 1$$

Utilisez un inverseur sur b et une retenue d'entrée de 1

3. **Utilisez les Mux** : Un Mux4Way ou Mux8Way sélectionne parmi plusieurs résultats

4. **Les drapeaux** :

- N : bit 31 du résultat
- Z : tous les bits sont à 0 ? (utilisez un grand OR puis NOT)
- C : retenue de sortie de l'additionneur
- V : comparez les signes des entrées et du résultat

4.10.3 Tests en ligne de commande

```
# Tester le Half Adder
cargo run -p hdl_cli -- test hdl_lib/03_arith/HalfAdder.hdl

# Tester l'ALU complète
cargo run -p hdl_cli -- test hdl_lib/03_arith/ALU.hdl
```

4.11 Défis Supplémentaires

4.11.1 Défi 1 : Carry Lookahead

Le Ripple Carry est lent car les retenues se propagent séquentiellement. Implémentez un "Carry Lookahead Adder" sur 4 bits qui calcule les retenues en parallèle.

4.11.2 Défi 2 : Multiplicateur

Construisez un circuit qui multiplie deux nombres de 4 bits. Indice : la multiplication est une série d'additions décalées.

4.11.3 Défi 3 : Comparateur

Construisez un circuit qui compare deux nombres 32 bits et produit trois sorties :

/bin/bash: ligne 1: q : commande introuvable - eq : a == b - gt : a > b

Indice : Vous pouvez utiliser la soustraction et regarder les drapeaux !

4.12 Le Lien avec le CPU

L'ALU que vous venez de construire sera utilisée à CHAQUE cycle d'horloge du CPU :

Instruction	Utilisation de l'ALU
ADD R1, R2, R3	Additionne R2 et R3, stocke dans R1
SUB R1, R2, R3	Soustrait R3 de R2
CMP R1, R2	Soustrait et met à jour les drapeaux (sans stocker)
AND R1, R2, R3	ET logique
LDR R1, [R2]	Calcule l'adresse mémoire (R2 + offset)
B label	Calcule la nouvelle adresse (PC + offset)

Même les sauts conditionnels (BEQ, BNE, etc.) dépendent des drapeaux produits par l'ALU !

4.13 Ce qu'il faut retenir

1. **Le binaire est naturel pour les circuits** : Addition = XOR pour le bit, AND pour la retenue
2. **Le complément à 2 est magique** : Un seul additionneur pour addition ET soustraction
3. **L'ALU est le cœur du calcul** : Toutes les opérations passent par elle
4. **Les drapeaux permettent les décisions** : Sans eux, pas de if, pas de boucles
5. **La hiérarchie continue** :
 - Portes → Half Adder → Full Adder → Additionneur 32-bits → ALU

Prochaine étape : Au Chapitre 3, nous aborderons la **mémoire**. Comment l'ordinateur peut-il "se souvenir" de données ? Nous construirons des flip-flops, des registres, et de la RAM.

Conseil : L'ALU est l'un des composants les plus complexes du projet. Prenez le temps de bien comprendre chaque étape. Si vous avez réussi l'ALU, le reste du projet sera beaucoup plus accessible !

4.14 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

4.14.1 Questions de compréhension

- Q1.** Dans un Half Adder, quelle porte logique calcule le bit de somme ? Et la retenue ?
- Q2.** Quelle est la représentation en complément à 2 de -1 sur 8 bits ?
- Q3.** Comment l'ALU effectue-t-elle une soustraction $A - B$?
- Q4.** Que signifient les drapeaux N, Z, C, V ?
- Q5.** Pourquoi `CMP R1, R2` n'a pas besoin de registre destination ?

4.14.2 Mini-défi pratique

Calculez mentalement le résultat de ces opérations sur 8 bits :

Opération	A (hex)	B (hex)	Résultat	Drapeaux
ADD	0x7F	0x01	?	N=? Z=? V=?
ADD	0xFF	0x01	?	N=? Z=? C=?
SUB	0x05	0x05	?	Z=?

Les solutions se trouvent dans le document **Codex_Solutions**.

4.14.3 Checklist de validation

Avant de passer au chapitre 3, assurez-vous de pouvoir :

- ☐ Expliquer la différence entre Half Adder et Full Adder
- ☐ Convertir un nombre négatif en complément à 2
- ☐ Expliquer comment la soustraction utilise l'addition
- ☐ Décrire le rôle de chaque drapeau (N, Z, C, V)
- ☐ Implémenter un additionneur 32 bits en chaînant des Full Adders

5 Logique Séquentielle et Mémoire

“Le temps est ce qui empêche tout d’arriver en même temps.” — John Wheeler

Jusqu’à présent, nos circuits étaient **combinatoires** : la sortie dépendait uniquement des entrées instantanées, comme une fonction mathématique pure $y = f(x)$. Si vous coupez le courant, ils “oublient” tout.

Pour construire un véritable ordinateur, nous devons pouvoir **stocker de l’information** et la récupérer plus tard. C’est le rôle de la **logique séquentielle**.

5.1 Où en sommes-nous ?

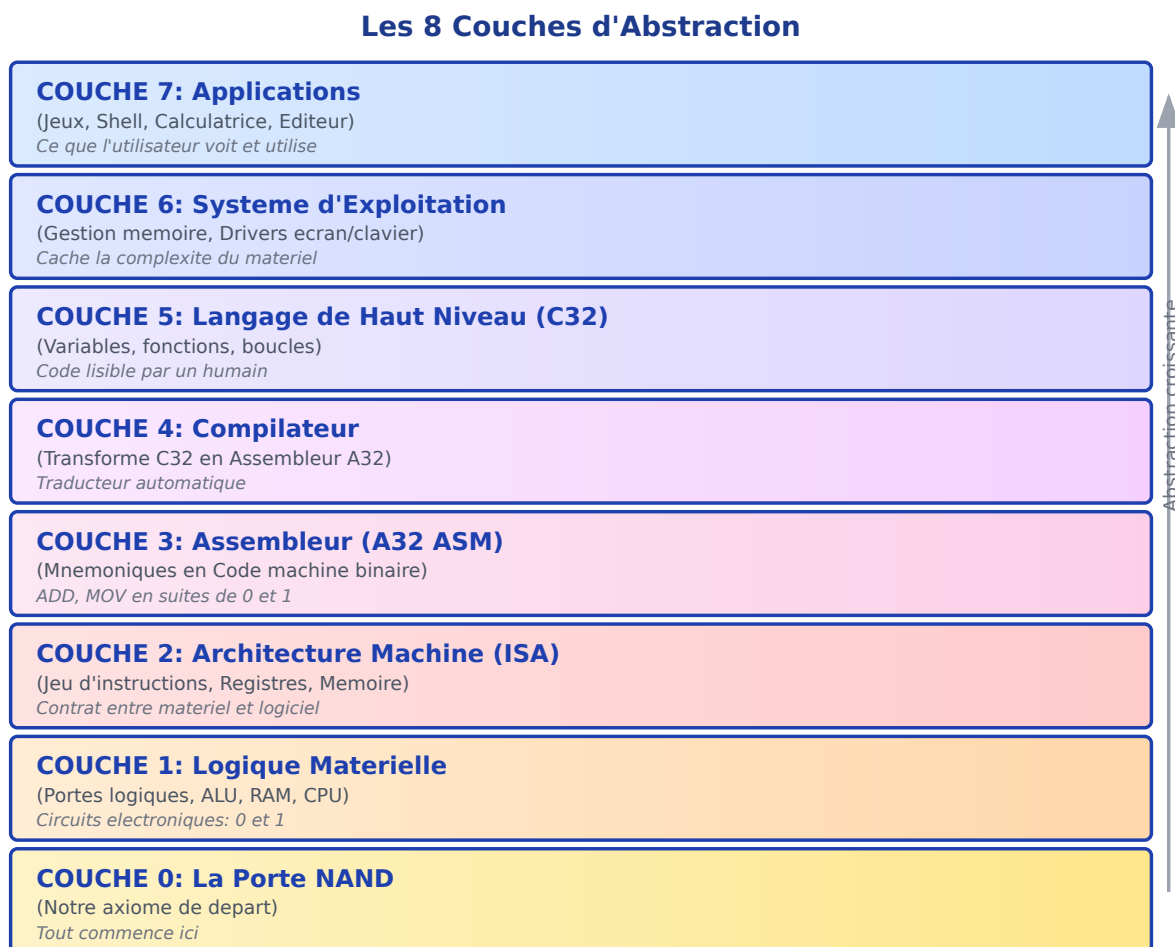


Figure 5.1: Position dans l'architecture

Nous sommes à la Couche 1 : Logique Matérielle (Portes - ALU - MEMOIRE - CPU)

Nous continuons à construire la couche matérielle. Après les portes logiques (Chapitre 1) et l'ALU (Chapitre 2), nous abordons maintenant la **mémoire** — le composant qui permet à l'ordinateur de “se souvenir”.

5.2 Pourquoi la Mémoire est-elle Fondamentale ?

5.2.1 Le problème de l'état

Imaginez un programme simple :

```
x = x + 1;
```

Pour exécuter cette instruction, l'ordinateur doit : 1. **Lire** la valeur actuelle de x 2. **Calculer** $x + 1$ (avec l'ALU) 3. **Écrire** le résultat dans x

Sans mémoire, il n'y a pas de "valeur actuelle de x" à lire. Sans mémoire, le résultat du calcul disparaît immédiatement après avoir été produit.

5.2.2 Ce que stocke la mémoire

Un ordinateur en fonctionnement stocke : - **Le programme** : Les instructions à exécuter (le code machine) - **Les données** : Les variables, les tableaux, les objets - **L'état du CPU** : Les registres, le compteur de programme - **La pile d'appels** : Pour les fonctions et les retours

Toutes ces informations vivent dans différentes formes de mémoire.

5.2.3 Combinatoire vs Séquentiel

Circuits Combinatoires	Circuits Séquentiels
Sortie = $f(\text{entrées})$	Sortie = $f(\text{entrées}, \text{état précédent})$
Pas de mémoire	A de la mémoire
Pas d'horloge	Synchronisé par une horloge
Exemples : AND, OR, ALU	Exemples : Registres, RAM, CPU

5.3 Le Temps et l'Horloge (Clock)

5.3.1 Le problème de la synchronisation

Dans un circuit combinatoire, les signaux se propagent à travers les portes avec un léger délai. Si on essaie de lire un résultat avant qu'il soit stable, on obtient des valeurs incorrectes.

La solution : L'**horloge** (clk).

L'horloge est un signal qui oscille perpétuellement entre 0 et 1 à une fréquence fixe :

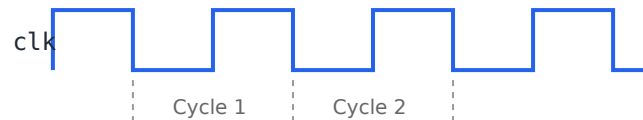


Figure 5.2: Signal d'horloge

5.3.2 Front montant (Rising Edge)

Le moment crucial est le **front montant** : le passage de 0 à 1.

Dans le système Codex, les changements d'état se produisent sur le front montant. Cela signifie :

- Pendant que l'horloge est à 0, les circuits combinatoires calculent
- Quand l'horloge passe à 1, les résultats sont capturés dans les registres

C'est comme dire : "Tout le monde calcule... et maintenant, on fige les résultats !"

5.3.3 Fréquence d'horloge

La fréquence de l'horloge détermine la vitesse du processeur :

- Un processeur à 1 GHz = 1 milliard de cycles par seconde
- À chaque cycle, le CPU peut exécuter (une partie d')une instruction

Plus vite bat l'horloge, plus l'ordinateur est rapide — mais aussi plus il chauffe !

5.4 La Bascule D (D Flip-Flop / DFF)

La **DFF** (Data Flip-Flop) est l'atome de la mémoire. C'est le plus petit circuit capable de mémoriser un bit.

5.4.1 Interface



Figure 5.3: Bascule D (DFF)

- d : La donnée à mémoriser (entrée)
- q : La donnée mémorisée (sortie)
- clk : L'horloge (parfois implicite)

5.4.2 Comportement

Règle fondamentale : $q(t) = d(t-1)$

La sortie à l'instant t est égale à ce qu'était l'entrée au cycle précédent.

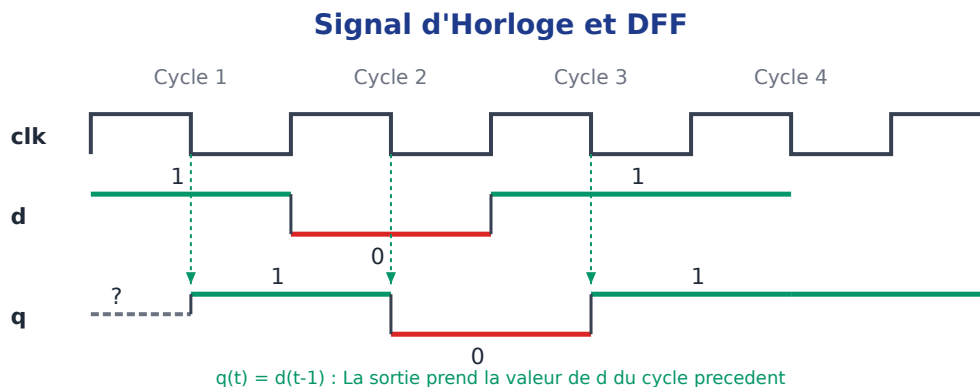


Figure 5.4: Signal d'horloge et DFF

5.4.3 Pourquoi est-ce utile ?

La DFF introduit un **délai d'un cycle**. Ce délai permet : 1. De stocker une valeur pour le prochain cycle 2. De casser les boucles (éviter les oscillations infinies) 3. De synchroniser tous les composants sur l'horloge

5.4.4 Comment fonctionne une DFF en interne ?

Une DFF peut être construite à partir de deux verrous (latches) en série, eux-mêmes construits à partir de portes NAND. C'est un sujet fascinant mais hors de notre scope — nous considérons la DFF comme une primitive fournie par le simulateur.

5.5 Le Registre 1-bit (Bit)

La DFF mémorise pendant UN cycle, puis elle prend la nouvelle valeur d'entrée. Comment garder une valeur **indéfiniment** ?

5.5.1 Le problème

On veut un circuit qui :

- Si $\text{load} = 1$: stocke la nouvelle valeur in
- Si $\text{load} = 0$: conserve l'ancienne valeur

5.5.2 La solution : la rétroaction

On utilise un **Mux** pour choisir entre :

- L'ancienne valeur (sortie de la DFF)
- La nouvelle valeur (in)

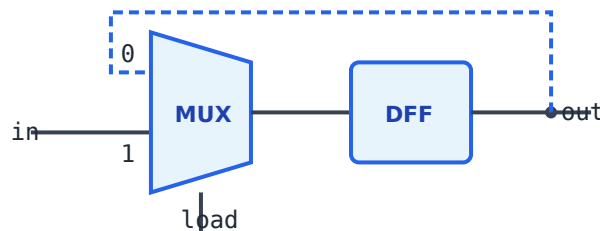


Figure 5.5: Registre 1-bit avec rétroaction

Fonctionnement :

- Si $\text{load} = 0$: Le Mux sélectionne la sortie de la DFF. La DFF ré-enregistre sa propre valeur. La valeur est **maintenue**.
- Si $\text{load} = 1$: Le Mux sélectionne in . La DFF enregistre la nouvelle valeur.

5.5.3 C'est magique !

Cette petite boucle de rétroaction transforme un délai d'un cycle en une mémoire permanente. C'est le principe fondamental de toute mémoire électronique.

5.6 Le Registre 32-bits

5.6.1 Du bit au mot

Un registre 32-bits est simplement **32 registres 1-bit en parallèle**, partageant le même signal load .

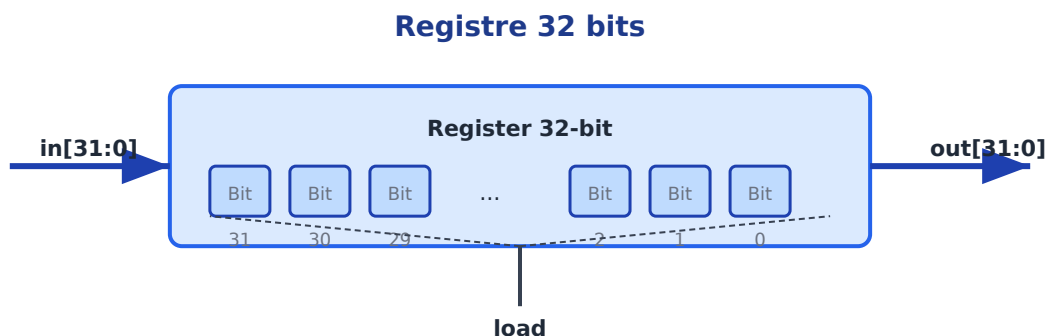


Figure 5.6: Registre 32 bits

Quand $\text{load} = 1$, les 32 bits sont capturés simultanément. C'est atomique.

5.6.2 Le rôle des registres dans le CPU

Le CPU Codex dispose de **16 registres** nommés R0 à R15 :

Registre	Rôle typique
R0-R12	Registres généraux (calculs, variables)
R13 (SP)	Stack Pointer (pointeur de pile)
R14 (LR)	Link Register (adresse de retour)
R15 (PC)	Program Counter (adresse de l'instruction courante)

Les registres sont la mémoire la plus rapide du CPU — ils sont directement connectés à l'ALU et peuvent être lus/écrits en un seul cycle.

5.7 La RAM (Random Access Memory)

5.7.1 Du registre à la mémoire

Un registre stocke UN mot de 32 bits. Pour stocker des millions de mots, nous construisons une **RAM** (Random Access Memory).

“Random Access” signifie qu'on peut accéder à n'importe quelle cellule directement, sans parcourir les autres. Contrairement à une bande magnétique où il faut rembobiner !

5.7.2 Interface de la RAM



Figure 5.7: Interface de la RAM

- **in** : La donnée à écrire
- **address** : L'adresse de la cellule à lire/écrire
- **load** : Si 1, écrire in à address. Si 0, ne rien écrire.
- **out** : La valeur stockée à address (toujours disponible en lecture)

5.7.3 Comment ça marche ?

La RAM utilise les composants que nous avons construits :

1. **DMux** (Démultiplexeur) : Route le signal load vers UN SEUL registre — celui correspondant à l'adresse
2. **Registres** : Stockent les données
3. **Mux** (Multiplexeur) : Sélectionne la sortie du registre correspondant à l'adresse

5.7.4 Exemple : RAM8 (8 mots de 32 bits)

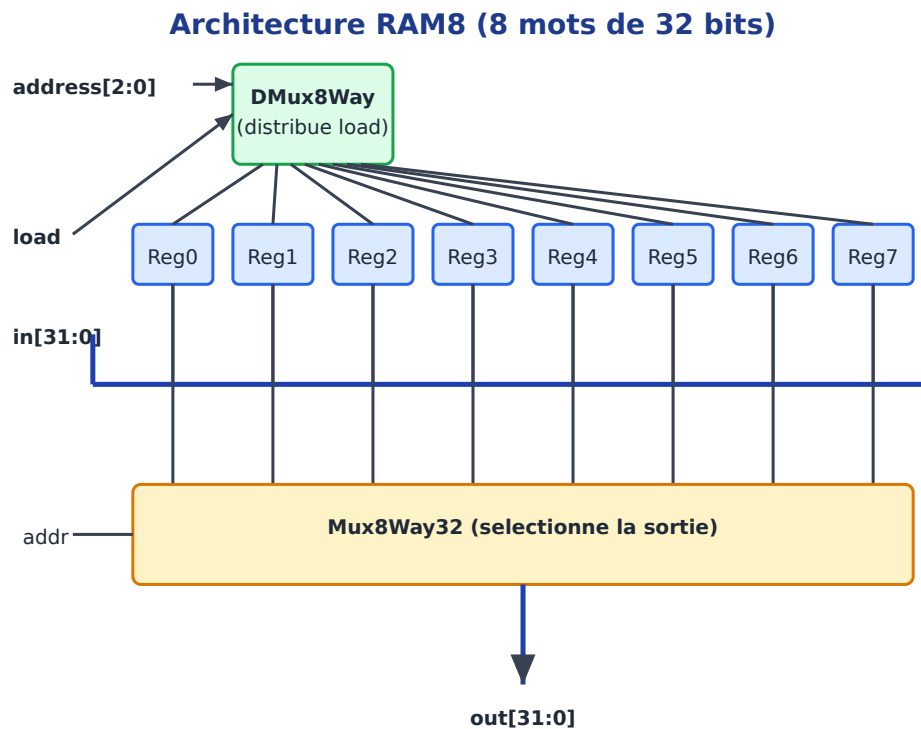


Figure 5.8: Architecture RAM8

5.7.5 Construction hiérarchique de grandes RAMs

Comment construire une RAM64 (64 mots) à partir de RAM8 ?

```
address[5:0] = address[5:3] (3 bits supérieurs) + address[2:0] (3 bits inférieurs)
```

↓ ↓

Sélectionne laquelle des 8 RAM8 Sélectionne le mot dans cette RAM8

On utilise 8 RAM8 :

- Les 3 bits de poids fort (address[5:3]) choisissent **quelle RAM8**
- Les 3 bits de poids faible (address[2:0]) choisissent **quel mot dans la RAM8**

C'est un pattern récursif qui permet de construire des mémoires de n'importe quelle taille !

5.8 Le Compteur de Programme (PC)

Le **Program Counter** (PC) est peut-être le registre le plus important du CPU. Il contient l'adresse de la prochaine instruction à exécuter.

5.8.1 Pourquoi est-il spécial ?

Après chaque instruction, le PC doit passer à l'instruction suivante. Mais parfois :

- On veut **sauter** à une autre adresse (boucles, conditions)
- On veut **revenir à 0** (redémarrage)

5.8.2 Les trois modes du PC

Priorité	Mode	Condition	Action
1	Reset	reset = 1	$PC \leftarrow 0$
2	Jump	load = 1	$PC \leftarrow in$
3	Increment	inc = 1	$PC \leftarrow PC + 1$
4	Hold	sinon	$PC \leftarrow PC$

5.8.3 Schéma simplifié

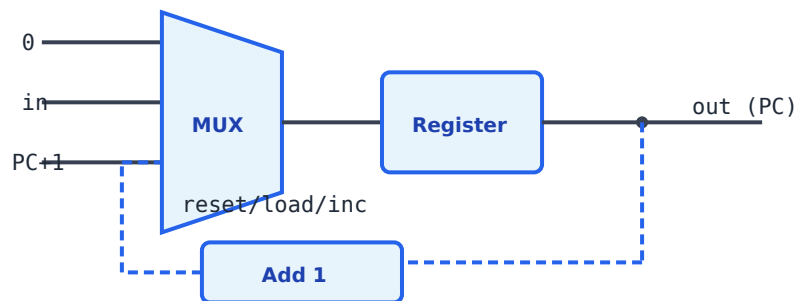


Figure 5.9: Compteur de programme (PC)

5.8.4 Le lien avec l'exécution du programme

À chaque cycle d'horloge :

1. Le CPU lit l'instruction à l'adresse PC
2. Il décode et exécute l'instruction
3. Il met à jour le PC (incrément ou saut)
4. Le cycle recommence

C'est le cœur battant de l'ordinateur !

5.9 Les Différents Types de Mémoire

Dans un vrai ordinateur, il y a plusieurs niveaux de mémoire :

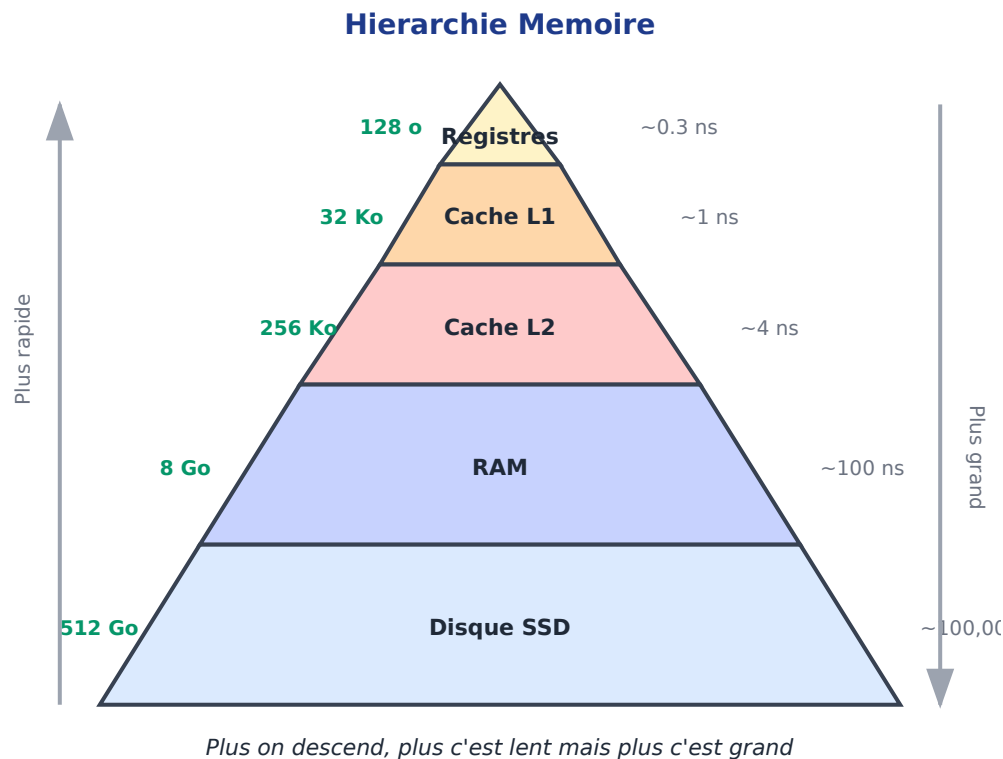


Figure 5.10: Pyramide de la hiérarchie mémoire

Plus on monte dans la pyramide :

- Plus c'est rapide
- Plus c'est cher par octet
- Plus la capacité est faible

Dans le Codex, nous implémentons les registres et la RAM. Les caches et les disques sont des concepts plus avancés.

5.10 Exercices Pratiques

5.10.1 Exercices sur le Simulateur Web

Lancez le **Simulateur Web** et allez dans **HDL Progression** → **Projet 4 : Séquentiel**.

Exercice	Description	Difficulté
DFF1	Bascule D (fournie comme primitive)	—
BitReg	Registre 1-bit (Mux + DFF)	[*]
Register16	Registre 16-bits (16 BitReg en parallèle)	[*]
PC	Compteur de programme avec reset/load/inc	[**]
RAM8	RAM de 8 mots (DMux + 8 Registres + Mux)	[**]
RAM64	RAM de 64 mots (8 RAM8)	[**]
RegFile	Banc de registres (lecture double, écriture simple)	[***]

5.10.2 Ordre de progression recommandé

1. **BitReg** : La brique de base. Un Mux et une DFF.
2. **Register16** : 16 BitReg en parallèle. Vérifiez que tous les bits sont synchronisés.
3. **PC** : Attention à la priorité ! reset > load > inc > hold
4. **RAM8** : Utilisez DMux8Way et Mux8Way16
5. **RAM64** : Composition de 8 RAM8

5.10.3 Prérequis

Avant de construire les RAMs, assurez-vous d'avoir terminé les composants multi-bits du Projet 2 : - Mux8Way16 : Sélectionne parmi 8 entrées de 16 bits - DMux8Way : Distribue 1 entrée vers 8 sorties

5.10.4 Tests en ligne de commande

```
# Tester le registre 1-bit
cargo run -p hdl_cli -- test hdl_lib/04_seq/BitReg.hdl

# Tester la RAM8
cargo run -p hdl_cli -- test hdl_lib/04_seq/RAM8.hdl

# Tester le PC
```

```
cargo run -p hdl_cli -- test hdl_lib/04_seq/PC.hdl
```

5.11 Défis Supplémentaires

5.11.1 Défi 1 : RAM avec deux ports de lecture

Construisez une RAM qui permet de lire DEUX adresses différentes simultanément (utile pour le CPU qui doit lire deux opérandes).

5.11.2 Défi 2 : Compteur avec valeur maximale

Modifiez le PC pour qu'il s'arrête à une valeur maximale au lieu de continuer à compter (overflow protection).

5.11.3 Défi 3 : Registre à décalage (Shift Register)

Construisez un registre où les bits se décalent d'une position à chaque cycle. Utilisé pour : - Les communications série - La multiplication/division par 2 - Les générateurs de nombres aléatoires

5.12 Le Lien avec la Suite

La mémoire que vous venez de construire sera utilisée partout dans l'ordinateur :

Composant	Utilisation de la mémoire
Registres R0-R15	16 registres 32-bits pour les calculs
RAM	Stockage du programme et des données
PC	Adresse de l'instruction courante
Pile (Stack)	Zone de RAM pour les appels de fonction
Écran	Zone de RAM mappée aux pixels (MMIO)
Clavier	Registre mappé en mémoire (MMIO)

Au Chapitre 4, nous verrons comment le CPU accède à la mémoire et aux périphériques via le **Memory-Mapped I/O** (MMIO).

5.13 Ce qu'il faut retenir

1. **L'horloge synchronise tout** : Les changements se font sur le front montant
2. **La DFF est l'atome de mémoire** : $q(t) = d(t-1)$
3. **La rétroaction crée la persistance** : Mux + DFF = mémoire permanente
4. **La RAM est un tableau de registres** : Accès par adresse
5. **Le PC guide l'exécution** : Il pointe vers l'instruction courante
6. **Hierarchie de mémoire** :
 - Registres → Cache → RAM → Disque
 - Rapidité vs Capacité

Prochaine étape : Au Chapitre 4, nous définissons l'**Architecture Machine** (ISA). C'est le "contrat" entre le matériel et le logiciel : quelles instructions le CPU comprend-il ? Comment accède-t-il à la mémoire ?

Conseil : Prenez le temps de bien comprendre la boucle de rétroaction du registre 1-bit. C'est un concept fondamental qui revient constamment en informatique !

5.14 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

5.14.1 Questions de compréhension

- Q1.** Quelle est la différence fondamentale entre un circuit combinatoire et un circuit séquentiel ?
- Q2.** Pourquoi le signal load est-il nécessaire dans un registre ?
- Q3.** Comment fonctionne l'adressage dans une RAM de 8 mots ?
- Q4.** Le PC (Program Counter) a plusieurs modes : reset, load, inc, hold. Dans quel ordre de priorité fonctionnent-ils ?
- Q5.** Pourquoi la RAM est-elle plus lente que les registres ?

5.14.2 Mini-défi pratique

Dessinez le schéma bloc d'un registre 1-bit avec les composants suivants : - 1 MUX 2-vers-1 - 1 DFF

Indice : La sortie du DFF retourne vers une entrée du MUX.

*Les solutions se trouvent dans le document **Codex_Solutions**.*

5.14.3 Checklist de validation

Avant de passer au chapitre 4, assurez-vous de pouvoir :

- ☐ Expliquer pourquoi l'horloge est nécessaire (synchronisation)
- ☐ Décrire le comportement d'une DFF : $q(t) = d(t-1)$
- ☐ Construire un registre 1-bit avec MUX + DFF
- ☐ Expliquer comment RAM8 utilise DMux et Mux pour l'adressage
- ☐ Décrire les priorités du PC (reset > load > inc > hold)

6 Architecture Machine (ISA A32)

“Le langage est la limite de mon monde.” — Wittgenstein

Nous avons maintenant des portes logiques, des additionneurs, et de la mémoire. Mais comment **commander** tout cela ? C’est le rôle de l’**Architecture de Jeu d’Instructions** (ISA - Instruction Set Architecture).

L’ISA est le **contrat** entre le matériel et le logiciel. C’est la liste de toutes les instructions que le CPU sait exécuter.

6.1 Où en sommes-nous ?

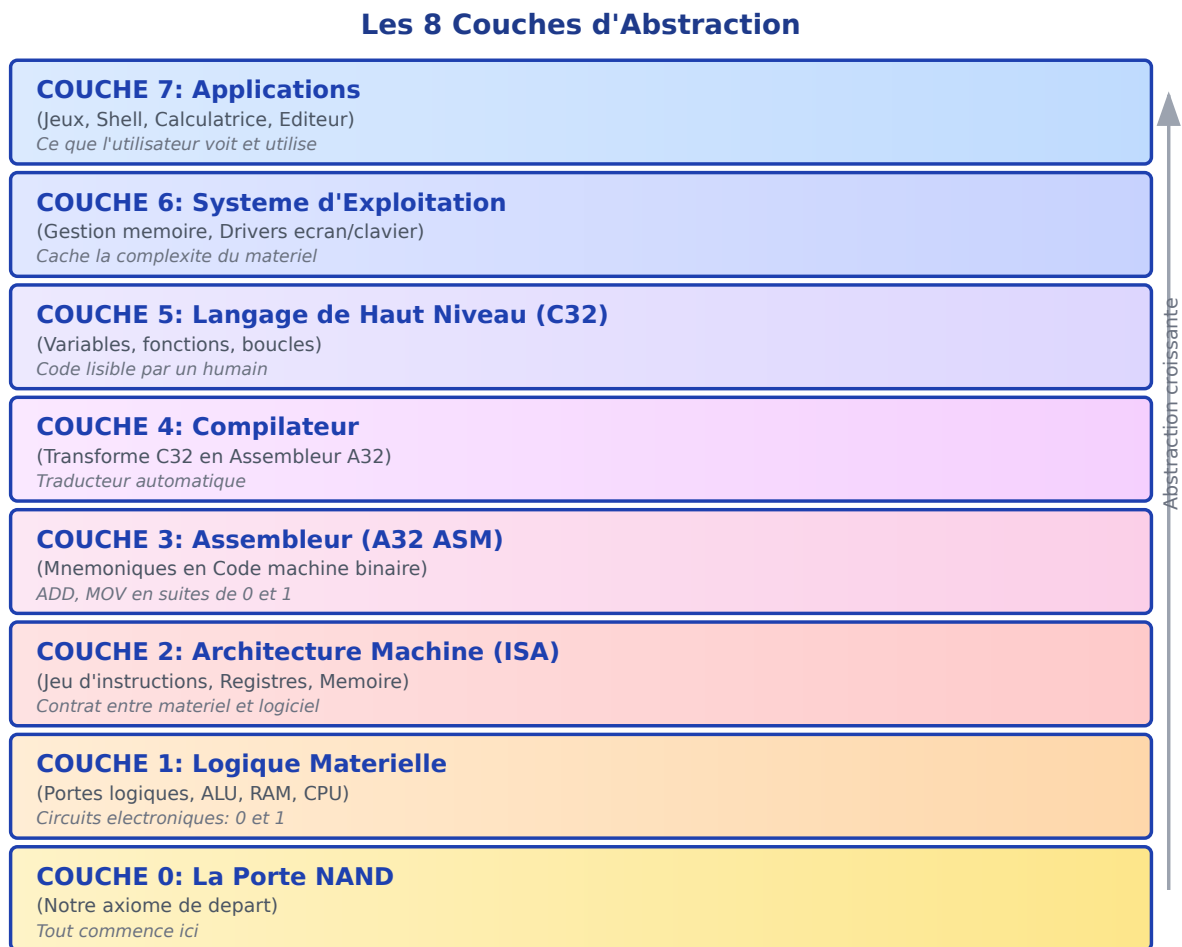


Figure 6.1: Position dans l'architecture

Nous sommes à la Couche 2 : Architecture Machine (ISA) - Le contrat entre matériel et logiciel

Ce chapitre marque une transition importante. Nous quittons temporairement le monde du matériel pour définir **l'interface** entre matériel et logiciel. L'ISA que nous définissons ici sera :

- **Implémentée** par le CPU au Chapitre 5
- **Utilisée** par l'assembleur au Chapitre 6
- **Ciblée** par le compilateur au Chapitre 7

6.2 Qu'est-ce qu'une Architecture ?

6.2.1 Le contrat fondamental

L'architecture d'un processeur définit :

1. **Les registres** : Combien ? Quelle taille ? Quel rôle ?
2. **Les instructions** : Quelles opérations le CPU peut-il faire ?
3. **L'encodage** : Comment les instructions sont-elles représentées en binaire ?
4. **Le modèle mémoire** : Comment le CPU voit-il la mémoire ?

C'est un **contrat** :

- Le matériel **promet** d'exécuter les instructions comme spécifié
- Le logiciel **s'engage** à n'utiliser que les instructions définies

6.2.2 Codex A32 : Une architecture RISC moderne

L'architecture **Codex A32** est inspirée de ARM, l'architecture qui équipe la plupart des smartphones et le Raspberry Pi. Elle est :

- **RISC** (Reduced Instruction Set Computer) : Instructions simples et rapides
- **32 bits** : Registres et adresses sur 32 bits
- **Load/Store** : Le CPU ne calcule jamais directement en mémoire

6.3 Pourquoi RISC ? L'architecture Load/Store

6.3.1 CISC vs RISC

CISC (x86, Intel)	RISC (ARM, Codex)
Instructions complexes	Instructions simples
ADD [mem], reg possible	Calcul uniquement entre registres
Vitesse variable par instruction	1 instruction \approx 1 cycle
Plus facile à programmer directement	Plus facile à implémenter en matériel

6.3.2 La règle d'or Load/Store

En architecture RISC, le CPU ne peut **jamais** calculer directement sur la mémoire. Toute opération suit le schéma :

1. **LOAD** : Charger les données de la mémoire vers les registres
2. **COMPUTE** : Effectuer le calcul dans les registres
3. **STORE** : Stocker le résultat de retour en mémoire

Exemple : Incrémenter une variable en mémoire

```
LDR R0, [R1]    ; 1. Charger la valeur depuis l'adresse R1
ADD R0, R0, #1   ; 2. Ajouter 1
STR R0, [R1]    ; 3. Stocker le résultat
```

En x86 (CISC), on pourrait écrire `ADD [mem], 1` en une seule instruction. Mais le CPU RISC est plus simple à construire et peut aller plus vite.

6.4 Le Cycle de Vie d'une Instruction

Chaque instruction traverse trois phases :



Figure 6.2: Cycle Fetch-Decode-Execute

1. **Fetch** (Récupération)

- Le CPU lit l'instruction à l'adresse contenue dans PC
- PC est incrémenté pour pointer vers l'instruction suivante

2. **Decode** (Décodage)

- Les 32 bits de l'instruction sont analysés
- Le CPU identifie : quel opération ? quels registres ? quelle valeur immédiate ?

3. **Execute** (Exécution)

- L'ALU effectue le calcul
 - Le résultat est stocké dans le registre de destination
 - Si c'est un branchement, PC peut être modifié
-

6.5 Les Registres : Le Plan de Travail

Le CPU dispose de **16 registres** de 32 bits, nommés R0 à R15.

6.5.1 Vue d'ensemble



Figure 6.3: Banc de registres

6.5.2 Rôles des registres

Registre	Alias	Rôle conventionnel
R0-R3	—	Arguments des fonctions et valeurs de retour
R4-R11	—	Variables locales (préservées par les fonctions)
R12	IP	Registre temporaire (Intra-Procedure call)
R13	SP	Stack Pointer — Pointe vers le sommet de la pile
R14	LR	Link Register — Adresse de retour après BL
R15	PC	Program Counter — Adresse de l'instruction courante

6.5.3 Le cas spécial de R15 (PC)

Le PC est accessible comme n'importe quel registre. Si vous écrivez dedans, vous forcez un saut !

```
MOV PC, R14    ; Équivalent à "return" : saute à l'adresse dans LR
ADD PC, PC, #8 ; Saute de 8 octets plus loin
```

C'est puissant mais dangereux — une erreur de calcul et le CPU saute n'importe où !

6.6 La Carte Mémoire (Memory Map)

La mémoire est un espace linéaire de 4 Go (2^{32} octets), mais toutes les adresses ne sont pas utilisables.

6.6.1 Organisation de la mémoire Codex

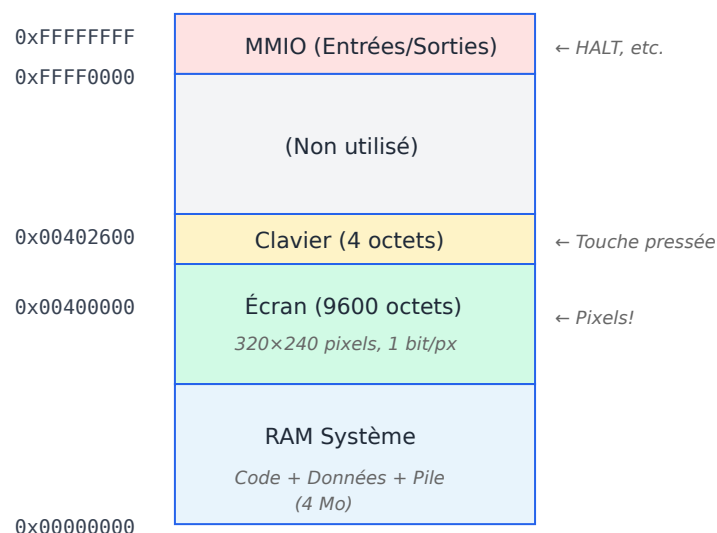


Figure 6.4: Carte mémoire Codex

6.6.2 Le Memory-Mapped I/O (MMIO)

En Codex (comme en ARM), les périphériques sont accessibles **comme de la mémoire**. Il n'y a pas d'instructions spéciales IN/OUT.

L'écran :

- Adresse : 0x00400000 à 0x00402580
- Format : 1 bit par pixel, 40 octets par ligne
- Écrire un 1 à un bit = pixel blanc

Le clavier :

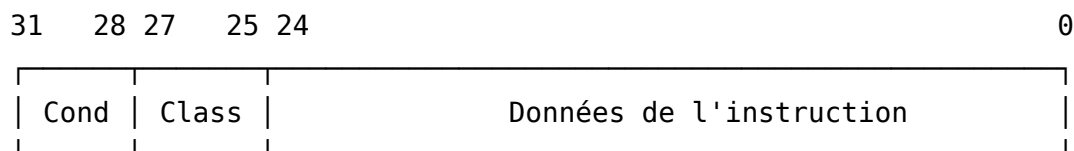
- Adresse : 0x00402600
- Lire cette adresse donne le code ASCII de la touche pressée (ou 0)

Exemple : Allumer le premier pixel

```
LDR R0, =0x00400000 ; Adresse de l'écran
MOV R1, #0x80       ; Bit 7 = premier pixel de la ligne
STRB R1, [R0]       ; Écrire un octet
```

6.7 Le Format des Instructions

Chaque instruction est encodée sur **32 bits**. La structure générale :



6.7.1 Les bits de condition (31-28)

Fonctionnalité unique de ARM/Codex : Toute instruction peut être conditionnelle !

Au lieu de :

```
CMP R0, #0
BNE skip
MOV R1, #1
skip:
```

On peut écrire :

```
CMP R0, #0
MOV.EQ R1, #1 ; Exécuté SEULEMENT si Z=1 (égal)
```

Code	Suffixe	Condition	Signification
0000	EQ	Z = 1	Égal (Equal)
0001	NE	Z = 0	Différent (Not Equal)

Code	Suffixe	Condition	Signification
0010	CS/HS	$C = 1$	Retenue (Carry Set)
0011	CC/LO	$C = 0$	Pas de retenue (Carry Clear)
1010	GE	$N = V$	Plus grand ou égal (signé)
1011	LT	$N \neq V$	Plus petit (signé)
1100	GT	$Z=0$ et $N=V$	Plus grand (signé)
1101	LE	$Z=1$ ou $N \neq V$	Plus petit ou égal (signé)
1110	AL	(toujours)	Toujours exécuter (défaut)

6.7.2 Les classes d'instructions (27-25)

Bits 27-25	Classe	Description
000	Data Processing (reg)	Opérations ALU avec registre
001	Data Processing (imm)	Opérations ALU avec immédiat
010	Load/Store	Accès mémoire (LDR, STR)
011	Branch	Branchements (B, BL)
100	Block Transfer	Push/Pop multiple (LDM, STM)
111	System	Instructions système (SVC, HALT)

6.8 Les Instructions en Détail

6.8.1 A. Opérations Arithmétiques et Logiques

Format général : $OP\{cond\}\{S\} \text{ Rd, Rn, Operand2}$

Instruction	Opération	Exemple
ADD	Addition	ADD R1, R2, R3 $\rightarrow R1 = R2 + R3$
SUB	Soustraction	SUB R1, R2, #5 $\rightarrow R1 = R2 - 5$
AND	ET logique	AND R1, R2, R3 $\rightarrow R1 = R2 \& R3$
ORR	OU logique	ORR R1, R2, R3 $\rightarrow R1 = R2 R3$

Instruction	Opération	Exemple
EOR	XOR	EOR R1, R2, R3 → $R1 = R2 \oplus R3$
MOV	Copie	MOV R1, R2 → $R1 = R2$
MVN	Copie inversée	MVN R1, R2 → $R1 = \sim R2$
CMP	Comparaison	CMP R1, R2 → met à jour les flags
TST	Test bits	TST R1, R2 → AND sans stocker

Le suffixe S : Ajouter S met à jour les drapeaux NZCV.

```
ADDS R1, R2, R3 ; Met à jour les flags
ADD R1, R2, R3 ; Ne touche pas aux flags
```

6.8.2 B. Accès Mémoire (Load/Store)

Format général : LDR/STR{B} Rd, [Rn, #offset]

Instruction	Action	Exemple
LDR	Charger 32 bits	LDR R0, [R1] → $R0 = \text{MEM}[R1]$
STR	Stocker 32 bits	STR R0, [R1] → $\text{MEM}[R1] = R0$
LDRB	Charger 8 bits	LDRB R0, [R1] → $R0 = \text{MEM}[R1]$ (1 octet)
STRB	Stocker 8 bits	STRB R0, [R1] → $\text{MEM}[R1] = R0$ (1 octet)

Modes d'adressage :

```
LDR R0, [R1] ; Simple : adresse = R1
LDR R0, [R1, #4] ; Offset : adresse = R1 + 4
LDR R0, [R1, R2] ; Registre : adresse = R1 + R2
```

6.8.3 C. Branchements

Format : B{cond} label ou BL{cond} label

Instruction	Action
B label	Saut inconditionnel
BL label	Branch with Link (appel de fonction)

Instruction	Action
B.EQ label	Saut si égal (Z=1)
B.NE label	Saut si différent (Z=0)
B.LT label	Saut si plus petit (signé)
B.GT label	Saut si plus grand (signé)

Le mystère de BL :

```
main:
    BL ma_fonction    ; 1. Sauvegarde PC+4 dans LR
                     ; 2. Saute à ma_fonction
    ; ... (on revient ici après le retour)

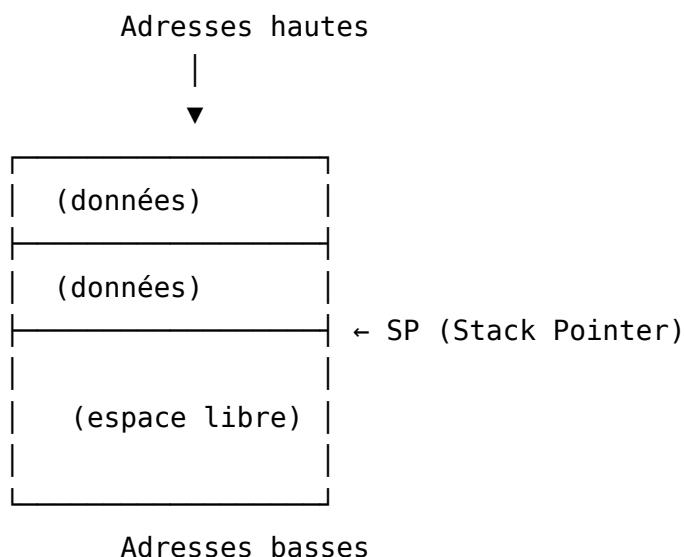
ma_fonction:
    ; ... faire quelque chose ...
    MOV PC, LR        ; Retour : saute à l'adresse dans LR
```

6.9 La Pile (Stack)

La pile est une zone de mémoire utilisée pour : - Sauvegarder les registres - Stocker les variables locales - Passer des arguments aux fonctions

6.9.1 Fonctionnement

La pile **grandit vers le bas** (des adresses hautes vers les basses) :



6.9.2 Push et Pop (manuel)

Codex n'a pas d'instructions PUSH/POP natives. On les simule :

```
; PUSH R0 (empiler R0)
SUB SP, SP, #4      ; Faire de la place (pile descend)
STR R0, [SP]        ; Stocker R0 au sommet

; POP R0 (dépiler dans R0)
LDR R0, [SP]        ; Lire depuis le sommet
ADD SP, SP, #4      ; Libérer l'espace
```

6.10 Exemples de Programmes

6.10.1 Exemple 1 : Somme de 1 à N

```
; Calcule 1 + 2 + ... + 10
.text
.global _start

_start:
    MOV R0, #0      ; sum = 0
    MOV R1, #1      ; i = 1

loop:
    CMP R1, #10
    BGT done        ; si i > 10, sortir
    ADD R0, R0, R1   ; sum += i
    ADD R1, R1, #1   ; i++
    B loop

done:
    ; R0 contient 55
    HALT
```

6.10.2 Exemple 2 : Maximum de deux nombres (avec prédication)

```
; R2 = max(R0, R1) sans branchement
CMP R0, R1
MOV.GE R2, R0      ; Si R0 >= R1, R2 = R0
MOV.LT R2, R1      ; Si R0 < R1, R2 = R1
```

6.10.3 Exemple 3 : Dessiner un pixel

```
; Allumer le pixel (10, 20)
; Adresse = 0x00400000 + (y * 40) + (x / 8)
; Bit = 7 - (x % 8)

LDR R0, =0x00400000 ; Base de l'écran
MOV R1, #20         ; y = 20
MOV R2, #40         ; octets par ligne
MUL R1, R1, R2       ; offset_y = y * 40
ADD R0, R0, R1       ; adresse_ligne

MOV R1, #10         ; x = 10
MOV R2, R1, LSR #3   ; x / 8
ADD R0, R0, R2       ; adresse finale

AND R1, R1, #7       ; x % 8
RSB R1, R1, #7       ; 7 - (x % 8)
MOV R2, #1
LSL R2, R2, R1       ; masque = 1 << bit_pos

LDRB R3, [R0]        ; Lire l'octet actuel
ORR R3, R3, R2        ; Allumer le bit
STRB R3, [R0]        ; Écrire l'octet
```

6.11 Gestion des Erreurs (Traps)

Si le CPU rencontre une situation invalide, il déclenche une **trap** :

Trap	Cause
ILLEGAL	Instruction invalide (opcode inconnu)
MEM_FAULT	Accès à une adresse non mappée
MISALIGNED	Accès 32-bits à une adresse non alignée (ex: 0x3)
DIV_ZERO	Division par zéro

6.12 Exercices Pratiques

6.12.1 Exercices sur le Simulateur Web

Lancez le **Simulateur Web** et allez dans **A32 Assembly**.

Exercice	Description	Difficulté
Hello World	Afficher du texte	[*]
Addition	Additionner deux registres	[*]
Soustraction	Soustraire avec le drapeau	[*]
Logique	Opérations AND, OR, XOR	[*]
Conditions	Utiliser les branches conditionnelles	[**]
Boucles	Implémenter une boucle while	[**]
Multiplication	Multiplier par additions successives	[**]
Fibonacci	Calculer la suite de Fibonacci	[**]
Tableaux	Parcourir un tableau en mémoire	[**]
Maximum Tableau	Trouver le max dans un tableau	[***]
Fonctions	Appeler des fonctions avec BL	[***]
Pixel	Allumer un pixel à l'écran	[**]
Ligne	Dessiner une ligne	[***]
Rectangle	Dessiner un rectangle	[***]
Lire un Caractère	Lire le clavier	[**]

6.12.2 Tests en ligne de commande

```
# Assembler un fichier
cargo run -p a32_cli -- assemble mon_prog.s -o mon_prog.bin

# Exécuter un binaire
cargo run -p a32_runner -- mon_prog.bin
```

6.13 Ce qu'il faut retenir

1. **L'ISA est un contrat** : Entre le matériel et le logiciel

2. **RISC = Simple** : Load, Compute, Store — jamais de calcul direct en mémoire
3. **16 registres** : R0-R12 généraux, R13=SP, R14=LR, R15=PC
4. **Tout est conditionnel** : ADD.EQ, MOV.GT évitent les branchements
5. **Memory-Mapped I/O** : L'écran et le clavier sont des adresses mémoire
6. **Le cycle Fetch-Decode-Execute** : Le cœur battant du CPU

Prochaine étape : Au Chapitre 5, nous construirons le CPU qui **implémente** cette architecture. Vous verrez comment les circuits du Chapitre 1-3 sont combinés pour exécuter ces instructions.

Conseil : Passez du temps sur le simulateur web à écrire des programmes en assembleur. Comprendre l'assembleur vous aidera énormément à comprendre le compilateur plus tard !

6.14 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

6.14.1 Questions de compréhension

- Q1.** Qu'est-ce qu'une architecture RISC et pourquoi A32 en est une ?
- Q2.** Pourquoi y a-t-il 16 registres (R0-R15) et pas 8 ou 32 ?
- Q3.** Que font les registres spéciaux R13, R14, R15 ?
- Q4.** Comment fonctionne l'exécution conditionnelle (ex: ADD.EQ) ?
- Q5.** Qu'est-ce que le Memory-Mapped I/O (MMIO) ?

6.14.2 Mini-défi pratique

Écrivez un programme assembleur qui : 1. Met 10 dans R0 2. Met 3 dans R1 3. Calcule R0 - R1 et stocke dans R2 4. Si le résultat est positif, met 1 dans R3, sinon 0

*Les solutions se trouvent dans le document **Codex_Solutions**.*

6.14.3 Checklist de validation

Avant de passer au chapitre 5, assurez-vous de pouvoir :

- ☐ Expliquer la différence entre RISC et CISC
- ☐ Décrire le rôle de chaque registre spécial (SP, LR, PC)
- ☐ Écrire un programme simple en assembleur A32
- ☐ Utiliser les conditions (EQ, NE, GT, LT, etc.)
- ☐ Expliquer le cycle Fetch-Decode-Execute

7 Le Processeur (CPU)

“Si vous ne pouvez pas le construire, vous ne le comprenez pas.” — Richard Feynman

C’est le grand moment. Nous allons assembler toutes les pièces du puzzle — portes logiques, ALU, registres, mémoire — pour construire le **cœur de l’ordinateur** : le CPU A32.

7.1 Où en sommes-nous ?

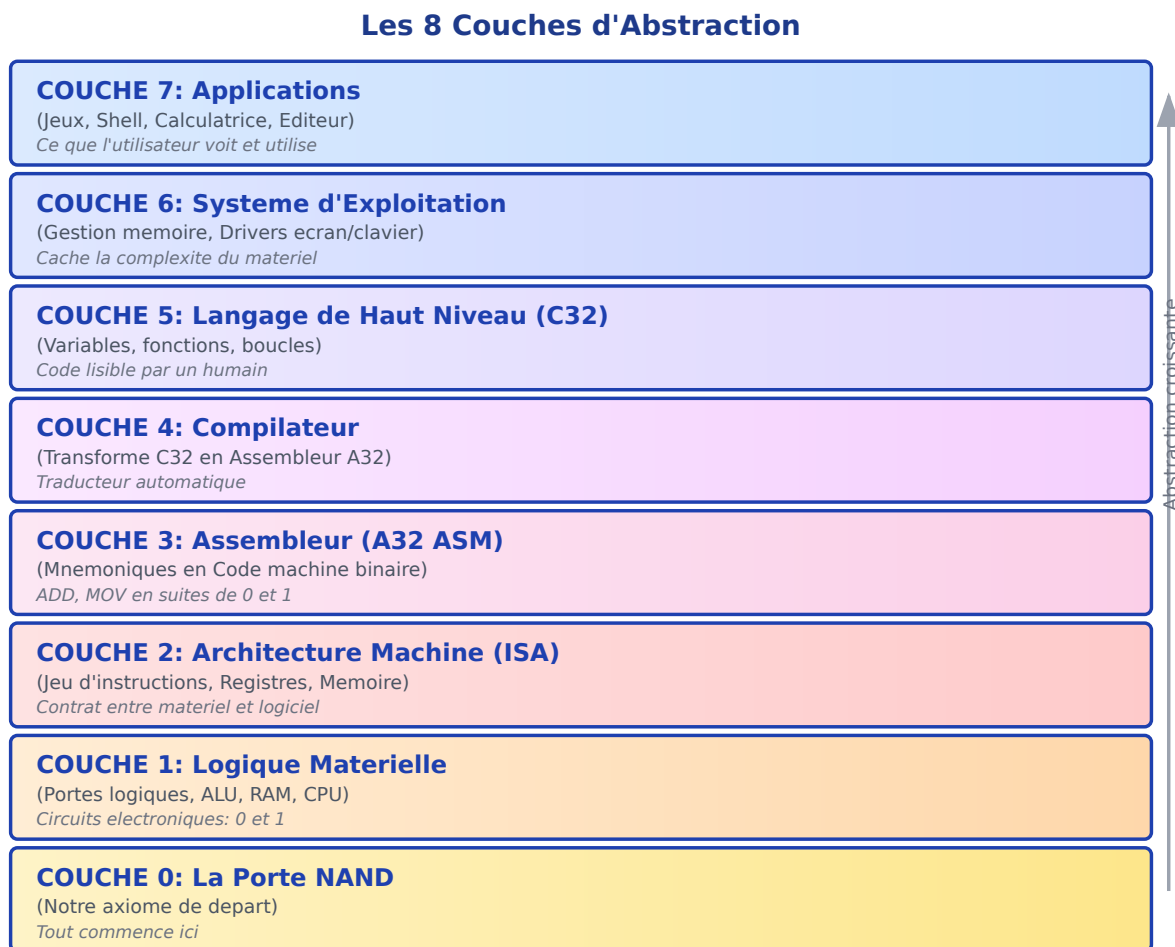


Figure 7.1: Position dans l'architecture

Nous sommes à la Couche 1 : CPU - L'Aboutissement (ALU + RAM + Registres)

Ce chapitre est le **point culminant** de tout le travail matériel. Après ce chapitre, vous aurez construit un ordinateur complet capable d'exécuter du vrai code !

7.2 Deux Implémentations du CPU

Le projet Codex propose **deux implémentations** du CPU A32, chacune avec un objectif pédagogique différent :

7.2.1 CPU Mono-cycle (Simulateur Rust)

Le **simulateur Rust** (a32_core) implémente un CPU **mono-cycle** :

CPU MONO-CYCLE



Figure 7.2: CPU Mono-cycle

Chaque instruction traverse TOUTES les étapes en UN cycle

Utilisé par :

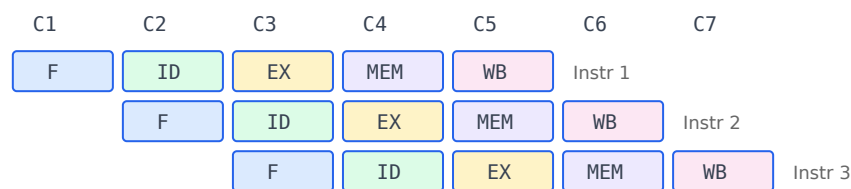
- Le **CPU Visualizer** (interface web)
- Le **runner** (a32_runner)
- L'**IDE web** (exécution des programmes)
- Les **tests** C32 et A32

Avantages : Simple à comprendre, facile à déboguer, comportement prévisible.

7.2.2 CPU Pipeline (HDL)

Le **CPU en HDL** (hdl_lib/05_cpu/CPU_Pipeline.hdl) implémente un vrai **pipeline 5 étages** :

CPU PIPELINE 5 ÉTAGES



Jusqu'à 5 instructions en vol simultanément!

Figure 7.3: CPU Pipeline 5 étages

Composants HDL :

Fichier	Rôle
IF_ID_Reg.hdl	Registre pipeline IF→ID
ID_EX_Reg.hdl	Registre pipeline ID→EX
EX_MEM_Reg.hdl	Registre pipeline EX→MEM
MEM_WB_Reg.hdl	Registre pipeline MEM→WB
HazardDetect.hdl	Détection des hazards (stall)

Fichier	Rôle
ForwardUnit.hdl	Bypass/forwarding des données
CPU_Pipeline.hdl	CPU complet assemblé

Utilisé par :

- Les **exercices HDL** (apprentissage hardware)
- Le **simulateur HDL** (hdl_cli)

Avantages : Réaliste, montre les vrais défis du design CPU (hazards, forwarding, stalls).

7.2.3 Pourquoi deux implémentations ?

Aspect	Mono-cycle (Rust)	Pipeline (HDL)
Objectif	Exécuter des programmes	Apprendre le hardware
Complexité	Simple	Réaliste
Performance	1 instr/cycle	Jusqu'à 1 instr/cycle (avec hazards)
Hazards	Aucun	Gérés (stall + forward)
Utilisation	IDE, tests, visualizer	Exercices HDL

Note : Le CPU Visualizer affiche les étapes (Fetch, Decode, Execute, Memory, Writeback) de manière **pédagogique**, mais l'exécution sous-jacente est mono-cycle. Pour voir un vrai pipeline avec hazards et forwarding, utilisez le simulateur HDL avec CPU_Pipeline.hdl.

7.3 Qu'est-ce qu'un CPU ?

7.3.1 Le chef d'orchestre

Le CPU (Central Processing Unit) est le composant qui :

1. **Lit** les instructions depuis la mémoire
2. **Décode** ces instructions pour comprendre quoi faire

3. **Exécute** les opérations (calculs, accès mémoire, branchements)
4. **Répète** à l'infini (jusqu'à HALT)

C'est une machine à états qui exécute une instruction après l'autre, inlassablement.

7.3.2 Ce que nous avons construit jusqu'ici

Chapitre	Composant	Rôle dans le CPU
1	Portes logiques	Briques de base de tout circuit
2	ALU	Effectue les calculs (ADD, SUB, AND...)
3	Registres	Stockent les données du CPU (R0-R15)
3	PC	Pointe vers l'instruction courante
3	RAM	Stocke le programme et les données
4	ISA	Définit les instructions à supporter

7.3.3 Ce qu'il reste à construire

- **Décodeur** : Analyse les bits de l'instruction
- **Unité de contrôle** : Décide quoi activer
- **Multiplexeurs de données** : Routent les données entre composants
- **Le CPU lui-même** : L'assemblage final

7.4 Architecture du CPU (Data Path)

Voici le schéma complet du CPU. Chaque flèche est un fil (ou un bus de 32 fils). Chaque boîte est un composant que vous avez construit ou que vous allez construire.

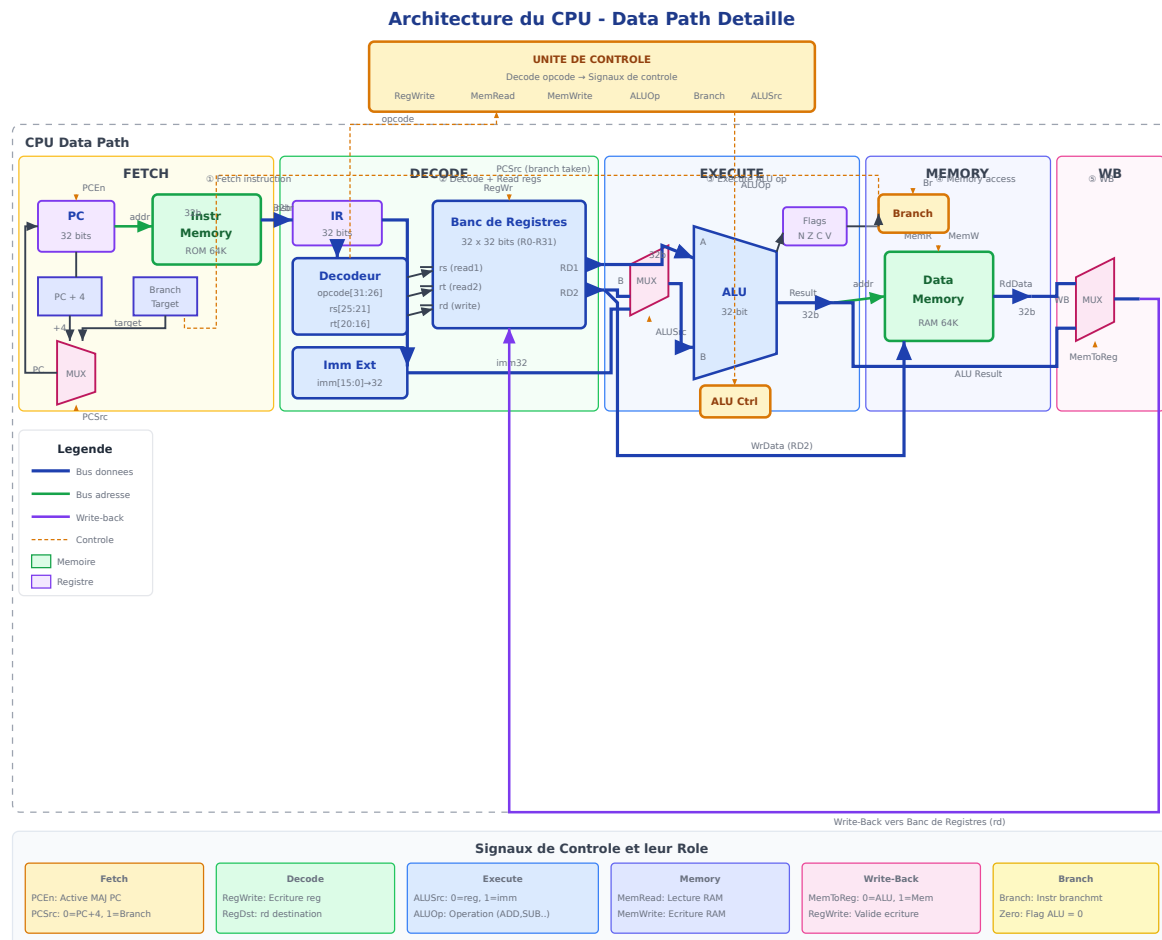


Figure 7.4: Architecture du CPU (Data Path)

7.4.1 Les flux de données

1. **Fetch** : PC → Mémoire Instructions → Instruction (32 bits)
2. **Decode** : Instruction → Décodeur → (Rd, Rn, Rm, Imm, opcode)
3. **Register Read** : Rn, Rm → Banc de Registres → Valeurs
4. **Execute** : Valeurs → ALU → Résultat
5. **Memory** : Résultat → Mémoire Données (si LDR/STR)
6. **Writeback** : Résultat → Registre Rd

7.5 Les Composants du CPU

7.5.1 1. Le Compteur de Programme (PC)

Vous l'avez déjà construit au Chapitre 3 ! Le PC contient l'adresse de l'instruction courante.

Modes de fonctionnement :

- `inc = 1` : $PC \leftarrow PC + 4$ (instruction suivante)
- `load = 1` : $PC \leftarrow$ adresse de branchement
- `reset = 1` : $PC \leftarrow 0$ (redémarrage)

7.5.2 2. Le Décodeur (Decoder)

Le décodeur est un circuit **purement combinatoire** qui “découpe” les 32 bits de l’instruction.

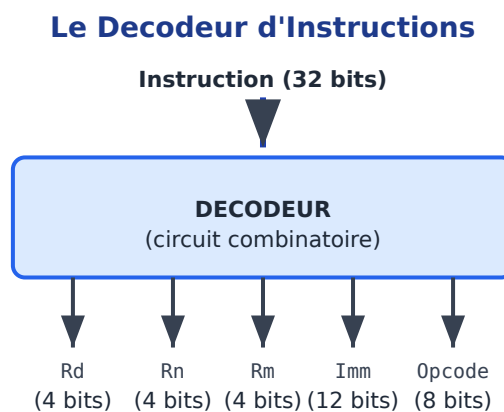


Figure 7.5: Le décodeur d’instructions

Sorties du décodeur :

Signal	Bits	Description
cond	31-28	Code de condition (EQ, NE, LT...)
class	27-25	Classe d’instruction (ALU, MEM, BRANCH)
op	24-21	Opération ALU (ADD, SUB, AND...)
S	20	Mettre à jour les drapeaux ?
Rn	19-16	Registre source 1
Rd	15-12	Registre destination
Rm	3-0	Registre source 2
imm12	11-0	Valeur immédiate (12 bits)
imm24	23-0	Offset de branchement (24 bits)

Le décodeur ne fait que du **câblage** — il ne calcule rien, il ne fait que router les bits vers les bonnes sorties.

7.5.3 3. L'Unité de Contrôle (Control)

L'unité de contrôle est le **chef d'orchestre**. Elle regarde la classe et l'opcode, et décide quels signaux activer.

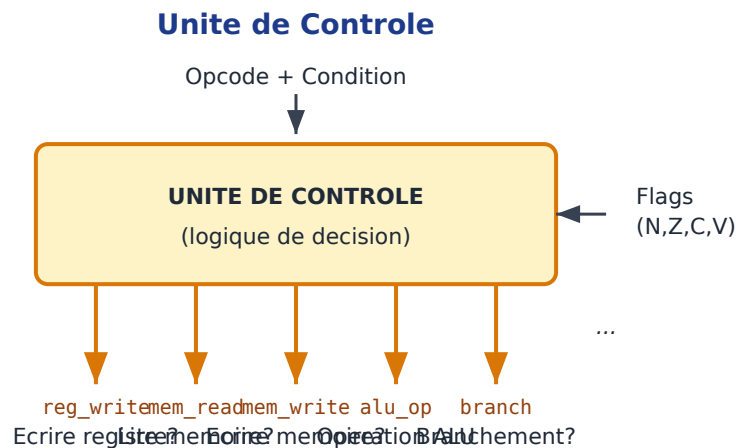


Figure 7.6: Unité de contrôle

Exemples de signaux de contrôle :

Instruction	reg_write	mem_read	mem_write	alu_src	branch
ADD	1	0	0	0 (reg)	0
ADD #imm	1	0	0	1 (imm)	0
LDR	1	1	0	1 (imm)	0
STR	0	0	1	1 (imm)	0
B	0	0	0	X	1
CMP	0	0	0	0	0

7.5.4 4. Le Vérificateur de Condition (CondCheck)

Ce petit circuit vérifie si la condition est satisfaite.

Entrées :

- cond : Le code de condition (4 bits, ex: 0000 = EQ)
- flags : Les drapeaux NZCV

Sortie :

- ok : 1 si la condition est vraie, 0 sinon

cond = 0000 (EQ) et Z = 1 → ok = 1
 cond = 0000 (EQ) et Z = 0 → ok = 0
 cond = 1110 (AL) → ok = 1 (toujours)

Pourquoi est-ce important ?

Si ok = 0, l'instruction est "annulée" — on n'écrit pas dans le registre, on ne fait pas le branchement. C'est la **prédication** en action !

7.5.5 5. Le Banc de Registres (RegFile)

Vous l'avez construit au Chapitre 3 (RAM8, RAM16...). Le banc de registres est une RAM spéciale avec :

- **2 ports de lecture** : Lire Rn ET Rm simultanément
- **1 port d'écriture** : Écrire dans Rd

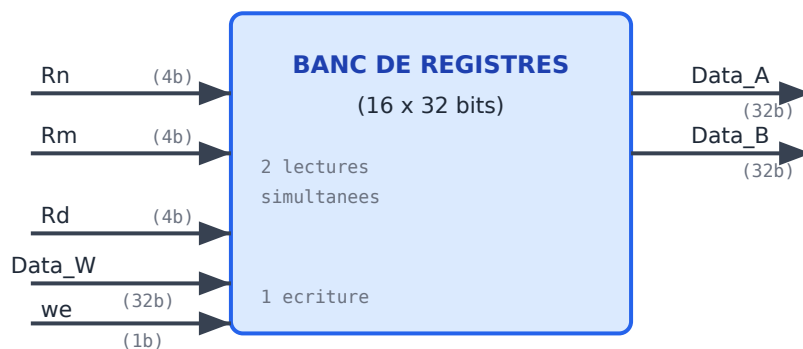


Figure 7.7: Interface du banc de registres

7.5.6 6. Les Multiplexeurs

Les multiplexeurs routent les données entre les composants :

Mux	Choix	Signification
ALU_src	0: Rm, 1: Imm	Deuxième opérande de l'ALU
Writeback	0: ALU, 1: MEM	Source de la valeur à écrire
PC_src	0: PC+4, 1: Branch	Prochaine valeur du PC

7.6 Le Cycle d'Exécution en Détail

Notre CPU est **single-cycle** : chaque instruction s'exécute en un seul cycle d'horloge.

7.6.1 Phase 1 : Fetch (Récupération)

PC \rightarrow Mémoire Instructions \rightarrow instruction (32 bits)

Le PC envoie son adresse à la mémoire d'instructions. La mémoire renvoie les 32 bits de l'instruction.

7.6.2 Phase 2 : Decode (Décodage)

```

instruction → Décodeur → cond, class, op, Rn, Rd, Rm, imm
                |
                └→ Unité de Contrôle → signaux

```

Le décodeur découpe l'instruction. L'unité de contrôle décide quoi activer.

7.6.3 Phase 3 : Register Read (Lecture des registres)

Rn, Rm \longrightarrow Banc de Registres \longrightarrow Data_A, Data_B

Les valeurs des registres sources sont lues.

7.6.4 Phase 4 : Execute (Exécution)

```

graph LR
    DA[Data_A] --> MUX
    DBI[Data_B ou Imm] --> MUX
    MUX --> ALU
    ALU --> RF[Résultat, Flags]

```

L'ALU effectue l'opération. Les drapeaux sont mis à jour (si S=1).

7.6.5 Phase 5 : Memory (Accès mémoire)

Résultat (adresse) \rightarrow $\begin{cases} \text{Si LDR : MEM[addr] } \rightarrow \text{valeur} \\ \text{Si STR : valeur } \rightarrow \text{MEM[addr]} \end{cases}$

Pour les instructions LDR/STR, on accède à la mémoire de données.

7.6.6 Phase 6 : Writeback (Écriture)

Résultat (ALU ou MEM) → MUX → Banc de Registres → Rd

Si `reg_write = 1` ET `cond_ok = 1`, on écrit dans le registre destination.

7.6.7 Phase 7 : PC Update

PC+4 ————┐
 ├── Si branch : PC + (offset × 4)
 └── Sinon : PC + 4

Le PC est mis à jour pour le prochain cycle.

7.7 Implémentation du CPU en HDL

Voici un squelette de l'architecture du CPU :

```
entity CPU is
  port(
    clk      : in bit;
    reset    : in bit;
    -- Interface mémoire
    instr    : in bits(31 downto 0);
    mem_in   : in bits(31 downto 0);
    pc_out   : out bits(31 downto 0);
    mem_addr : out bits(31 downto 0);
    mem_out  : out bits(31 downto 0);
    mem_we   : out bit
  );
end entity;

architecture rtl of CPU is
  -- Signaux internes
  signal pc, pc_next : bits(31 downto 0);
  signal cond, op : bits(3 downto 0);
  signal rd, rn, rm : bits(3 downto 0);
  signal imm12 : bits(11 downto 0);
  signal data_a, data_b, alu_result : bits(31 downto 0);
  signal reg_write, mem_read, mem_write, alu_src, branch : bit;
  signal n_flag, z_flag, c_flag, v_flag, cond_ok : bit;

begin
```

```
-- Instanciation des composants
u_decoder: Decoder port map (...);
u_control: Control port map (...);
u_condcheck: CondCheck port map (...);
u_regfile: RegFile port map (...);
u_alu: ALU port map (...);
u_pc: PC port map (...);

-- Multiplexeurs
alu_b <= imm12 when alu_src = '1' else data_b;
writeback <= mem_in when mem_read = '1' else alu_result;
pc_next <= branch_addr when (branch and cond_ok) = '1' else pc_plus_4;

end architecture;
```

7.8 Exercices Pratiques

7.8.1 Exercices sur le Simulateur Web

Lancez le **Simulateur Web** et allez dans **HDL Progression → Projet 5 : CPU**.

Exercice	Description	Difficulté
Decoder	Découper l'instruction en champs	[**]
CondCheck	Vérifier les conditions (EQ, NE, LT...)	[**]
Control	Générer les signaux de contrôle	[***]
CPU	L'assemblage final !	[****]

7.8.2 Ordre de progression

1. **Decoder** : Commencez par là. C'est du pur câblage.
 - Utilisez la syntaxe `instr(31 downto 28)` pour extraire les bits
2. **CondCheck** : Table de vérité des conditions
 - EQ : $Z = 1$
 - NE : $Z = 0$
 - LT : $N \neq V$
 - etc.
3. **Control** : La logique de commande

- Pour chaque classe d'instruction, décidez les signaux
- Attention aux cas spéciaux (CMP ne fait pas reg_write)

4. CPU : L'assemblage final

- Suivez le schéma du data path
- N'oubliez pas les multiplexeurs !

7.8.3 Tests en ligne de commande

```
# Tester le décodeur
cargo run -p hdl_cli -- test hdl_lib/05_cpu/Decoder.hdl

# Tester le CPU complet
cargo run -p hdl_cli -- test hdl_lib/05_cpu/CPU.hdl
```

7.9 CPU Visualizer : L'Outil Interactif

Pour mieux comprendre comment le CPU exécute les instructions, le projet inclut un **CPU Visualizer** interactif. C'est un outil web qui vous permet de voir en temps réel le fonctionnement du processeur.

7.9.1 Accéder au Visualizer

1. Lancez le serveur web :

```
cd web
npm install
npm run dev
```

2. Ouvrez votre navigateur à l'adresse indiquée (généralement `http://localhost:5173`)
3. Cliquez sur **CPU Visualizer** dans la barre de navigation

7.9.2 Fonctionnalités du Visualizer

7.9.2.1 Vue Pipeline

Le Visualizer affiche les **5 étapes du cycle d'exécution** :

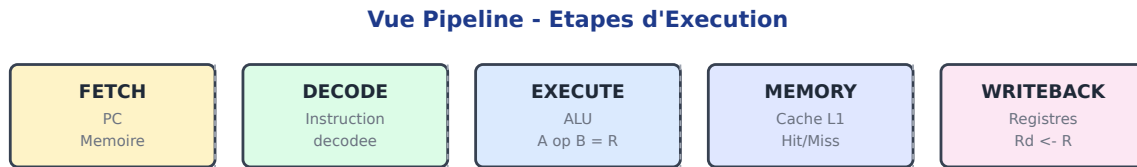


Figure 7.8: Vue Pipeline - Étapes d'exécution

Chaque étape s'illumine en jaune quand elle est active, vous permettant de suivre la progression de l'instruction.

7.9.2.2 Panneau Registres

Affiche les **16 registres** (R0-R15) avec les alias :

- **SP** (R13) : Stack Pointer
- **LR** (R14) : Link Register
- **PC** (R15) : Program Counter

Les registres modifiés s'illuminent en vert pendant un instant.

7.9.2.3 Panneau Flags (CPSR)

Les 4 drapeaux du processeur sont affichés :

- **N** (Negative) : Le résultat est négatif
- **Z** (Zero) : Le résultat est zéro
- **C** (Carry) : Retenue/emprunt
- **V** (Overflow) : Débordement signé

Les flags changent de couleur quand ils sont actifs.

7.9.2.4 Panneau Code Source

Affiche le code assembleur avec :

- **Coloration syntaxique** : Instructions, registres, nombres, commentaires
- **Surlignage de la ligne courante** : La ligne en cours d'exécution est mise en évidence en jaune
- **Défilement automatique** : Le code défile pour suivre l'exécution

7.9.2.5 Panneau Mémoire et Cache

Affiche :

- **Vue mémoire** : Les octets en mémoire autour du PC
- **Statistiques cache** : Hits, Misses, Taux de réussite
- **Contenu du cache L1** : Lignes valides avec tag et données
- **Indicateur HIT/MISS** : Flash vert pour hit, rouge pour miss

7.9.3 Les Démonstrations Intégrées

Le Visualizer inclut 7 démonstrations prêtes à l'emploi :

Demo	Description	Concept illustré
1. Addition	$5 + 3 = 8$	Instructions ALU basiques
2. Boucle	Somme 1-5	Branchements conditionnels
3. Mémoire	LDR/STR	Accès mémoire
4. Condition	Valeur absolue	Prédication
5. Tableau	Somme tableau	Boucle + mémoire
6. Flags	N, Z, C, V	Drapeaux CPU
7. Cache	Parcours mémoire	Cache hits/misses

7.9.4 Contrôles

Bouton	Raccourci	Action
Reset	Ctrl+R	Remet le CPU à zéro
Step	N, F10	Exécute une instruction
Play/Pause	Espace	Lance/arrête l'exécution continue
Vitesse	Slider	Ajuste la vitesse d'exécution

7.9.5 Charger Votre Propre Code

1. Cliquez sur **Charger fichier**
2. Sélectionnez un fichier .asm, .a32 ou .a32b
3. Le code est assemblé et chargé automatiquement

7.9.6 Exercice Pratique

Utilisez le Visualizer pour observer ces comportements :

1. **Suivez une addition** : Chargez la démo "Addition" et observez comment ADD lit deux registres et écrit le résultat
 2. **Observez un branchement** : Chargez la démo "Boucle" et regardez comment B.LE revient au début de la boucle
 3. **Analysez le cache** : Chargez la démo "Cache" et observez les miss au premier parcours, puis les hits au second
-

7.10 Conseils de Débogage

7.10.1 Le PC reste à 0 ?

- Vérifiez que inc = 1 par défaut
- Vérifiez que le reset n'est pas bloqué

7.10.2 Les branchements ne marchent pas ?

- L'offset dans l'instruction est en mots ($\times 4$ pour avoir des octets)
- Vérifiez que cond_ok est correct
- Vérifiez le calcul de l'adresse de branchement

7.10.3 Rien ne s'écrit dans les registres ?

- reg_write doit être à 1
- cond_ok doit être à 1
- Le registre destination ne doit pas être R15 (géré à part)

7.10.4 LDR/STR ne fonctionne pas ?

- Vérifiez le calcul de l'adresse (base + offset)
 - Vérifiez les signaux mem_read et mem_write
 - Attention à l'alignement (adresses multiples de 4)
-

7.11 Aller Plus Loin : Le CPU Pipeline

Le CPU single-cycle que nous avons construit est simple et pédagogique. Mais dans le monde réel, il serait **très lent**. Les vrais processeurs utilisent une technique appelée **pipeline** pour être beaucoup plus rapides.

Cette section explique en détail ce qu'est un pipeline, pourquoi il est nécessaire, et comment le construire.

7.11.1 Pourquoi le CPU Single-Cycle est Lent

7.11.1.1 Le problème de la chaîne critique

Dans notre CPU single-cycle, une instruction doit traverser **tous** les composants en un seul cycle :

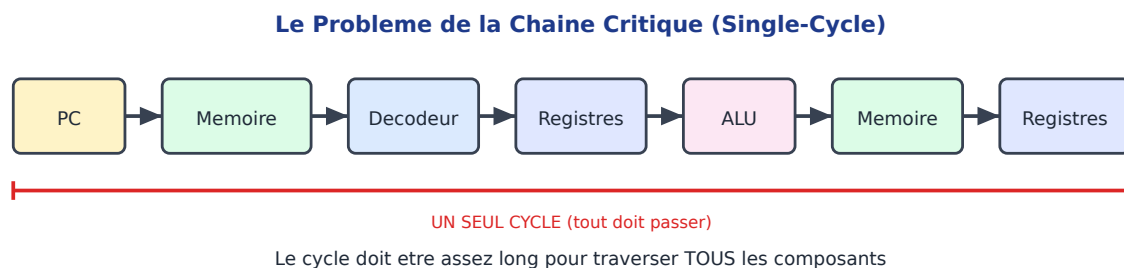


Figure 7.9: Le problème de la chaîne critique

Le cycle d'horloge doit être assez **long** pour que le signal traverse tout ce chemin. Si chaque étape prend 1 nanoseconde, le cycle doit faire au minimum 6 ns.

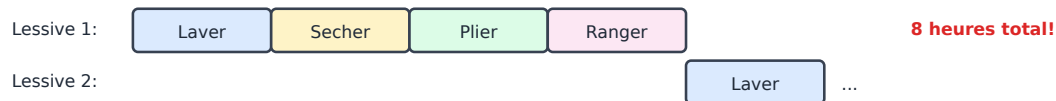
Résultat : Même si certaines instructions n'ont pas besoin de la mémoire de données (comme ADD), elles prennent quand même 6 ns.

7.11.1.2 Une analogie : La laverie

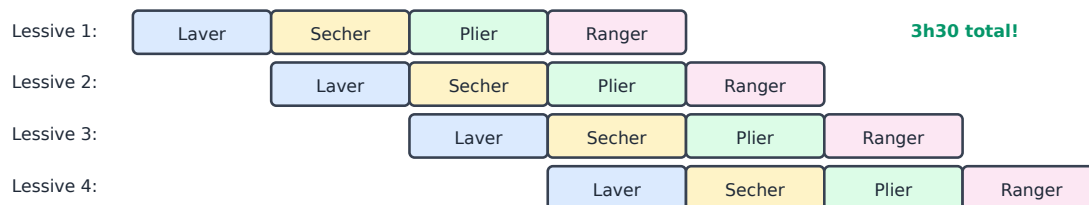
Imaginez que vous avez 4 lessives à faire. Chaque lessive a 4 étapes : laver (30 min), sécher (30 min), plier (30 min), ranger (30 min).

Analogie : La Laverie

Approche "Single-Cycle" : une lessive a la fois



Approche "Pipeline" : parallelisation



Le pipeline traite plus de lessives par heure (debit), meme si chaque lessive prend toujours 2h

Figure 7.10: Analogie de la laverie

Le pipeline ne rend pas une lessive individuelle plus rapide, mais il permet de traiter **plus de lessives par heure** !

7.11.2 Le Pipeline à 5 Étages

Notre CPU pipeliné divise l'exécution en **5 étapes**, chacune prenant exactement 1 cycle d'horloge :

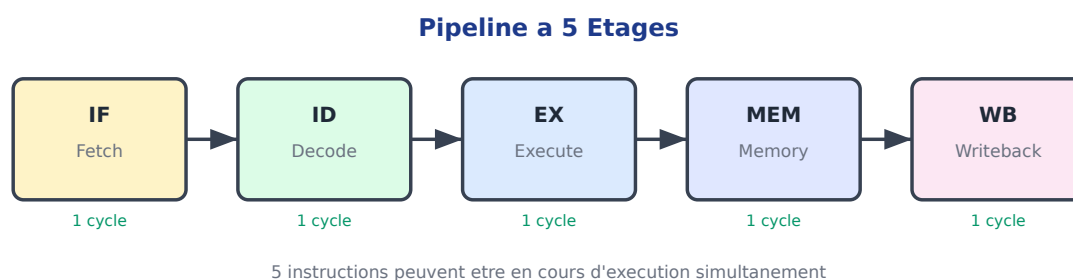
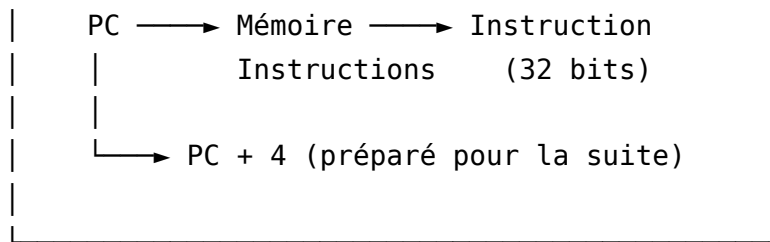


Figure 7.11: Pipeline à 5 étages

7.11.2.1 Étape 1 : IF (Instruction Fetch)

But : Aller chercher l'instruction en mémoire.

IF - Instruction Fetch



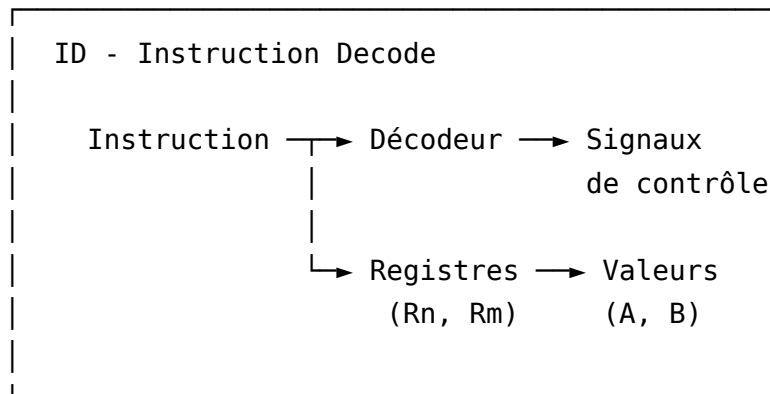
Ce qui se passe :

1. Le PC (Program Counter) envoie son adresse à la mémoire
2. La mémoire renvoie l'instruction (32 bits)
3. On calcule PC + 4 pour l'instruction suivante

Sortie : L'instruction et PC+4 sont stockés dans le registre IF/ID.

7.11.2.2 Étape 2 : ID (Instruction Decode)

But : Comprendre l'instruction et lire les registres sources.



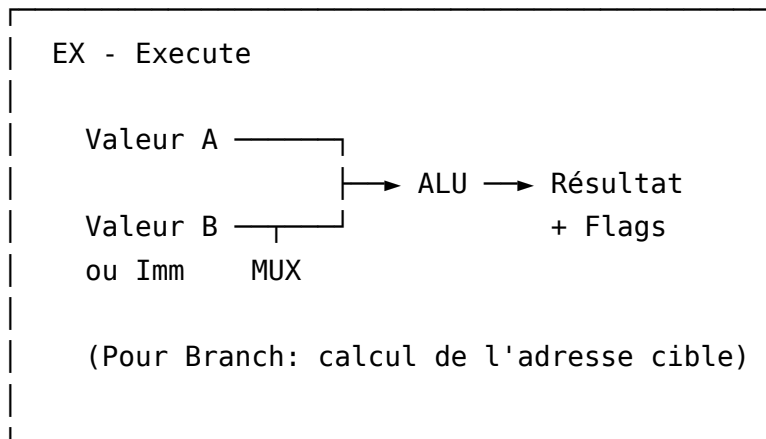
Ce qui se passe :

1. Le décodeur extrait les champs (Rd, Rn, Rm, opcode, etc.)
2. L'unité de contrôle génère les signaux (reg_write, mem_read, etc.)
3. On lit les valeurs des registres Rn et Rm
4. On détecte les éventuels aléas (hazards)

Sortie : Tout est stocké dans le registre ID/EX.

7.11.2.3 Étape 3 : EX (Execute)

But : Effectuer le calcul.



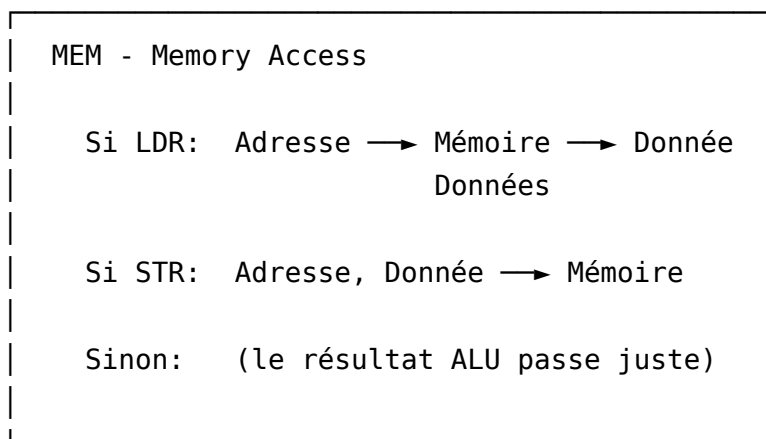
Ce qui se passe :

1. L'ALU effectue l'opération (ADD, SUB, AND, etc.)
2. Les flags (N, Z, C, V) sont calculés
3. Pour les branchements, on calcule l'adresse cible
4. Le forwarding peut injecter des valeurs ici (on verra plus tard)

Sortie : Le résultat ALU est stocké dans le registre EX/MEM.

7.11.2.4 Étape 4 : MEM (Memory Access)

But : Lire ou écrire en mémoire (pour LDR/STR seulement).



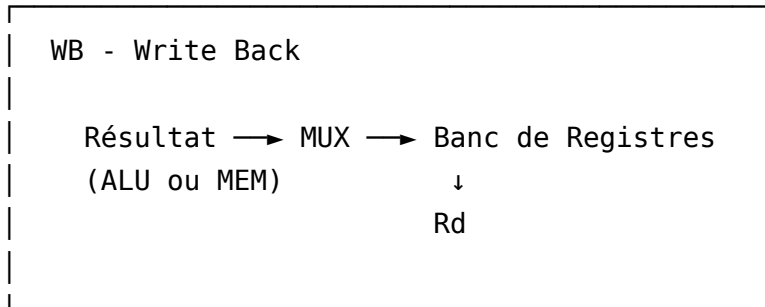
Ce qui se passe :

1. Pour LDR : on lit la mémoire à l'adresse calculée
2. Pour STR : on écrit la valeur en mémoire
3. Pour les autres instructions : rien (le résultat ALU est juste transmis)

Sortie : Le résultat (ALU ou mémoire) est stocké dans le registre MEM/WB.

7.11.2.5 Étape 5 : WB (Write Back)

But : Écrire le résultat dans le registre destination.



Ce qui se passe :

1. On choisit le résultat à écrire (ALU ou mémoire)
2. Si `reg_write = 1`, on écrit dans le registre `Rd`

7.11.3 Visualisation du Pipeline en Action

Voici comment 5 instructions traversent le pipeline :

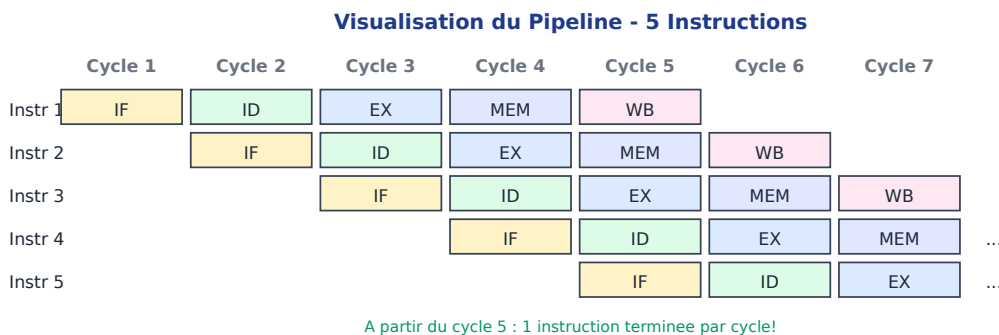


Figure 7.12: Visualisation du timing du pipeline

Observation clé : À partir du cycle 5, le pipeline est “rempli” et on termine **une instruction par cycle** !

Comparaison des performances :

CPU	100 instructions	Temps (si cycle = 1ns)
Single-cycle	100 cycles	100 ns

CPU	100 instructions	Temps (si cycle = 1ns)
Pipeline 5 étages	104 cycles*	104 ns

*Attendez... le pipeline n'est pas plus rapide ?

C'est parce que le cycle du pipeline est **5× plus court** ! Chaque étage ne fait qu'une partie du travail.

CPU	Durée cycle	100 instructions	Temps réel
Single-cycle	5 ns	100 cycles	500 ns
Pipeline	1 ns	104 cycles	104 ns

Le pipeline est **~5× plus rapide** !

7.11.4 Les Registres de Pipeline

Pour que le pipeline fonctionne, il faut **stocker** les résultats intermédiaires entre chaque étage. C'est le rôle des **registres de pipeline**.

7.11.4.1 Le registre IF/ID

Stocke :

- L'instruction (32 bits)
- PC+4 (32 bits)

Signaux spéciaux :

- stall : Si 1, garder les mêmes valeurs (ne pas avancer)
- flush : Si 1, mettre l'instruction à NOP (annuler)

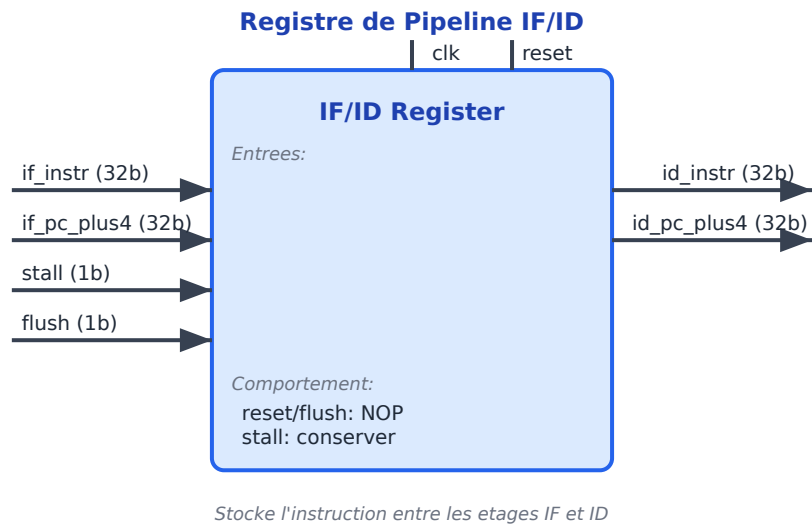


Figure 7.13: Registre de pipeline IF/ID

7.11.5 Les Aléas (Hazards)

Le pipeline crée de nouveaux problèmes. Quand une instruction dépend du résultat d'une instruction précédente qui n'est pas encore terminée, on a un **aléa**.

7.11.5.1 Aléa de Données (Data Hazard)

Exemple problématique :

```
ADD R1, R2, R3    ; Instruction 1: R1 = R2 + R3
SUB R4, R1, R5    ; Instruction 2: R4 = R1 - R5 (utilise R1!)
```

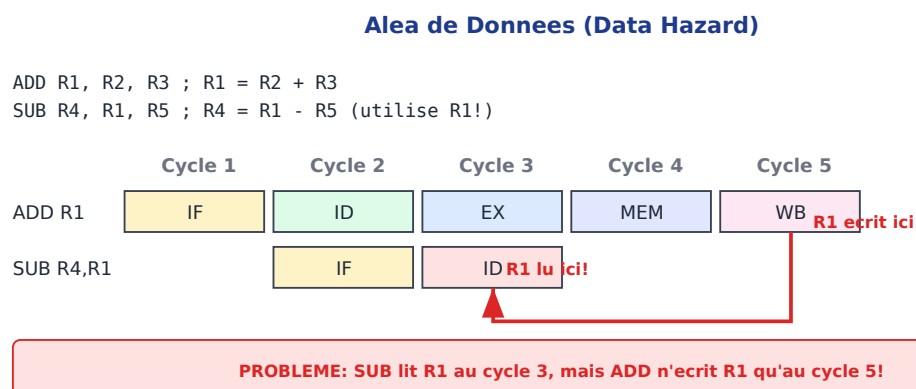
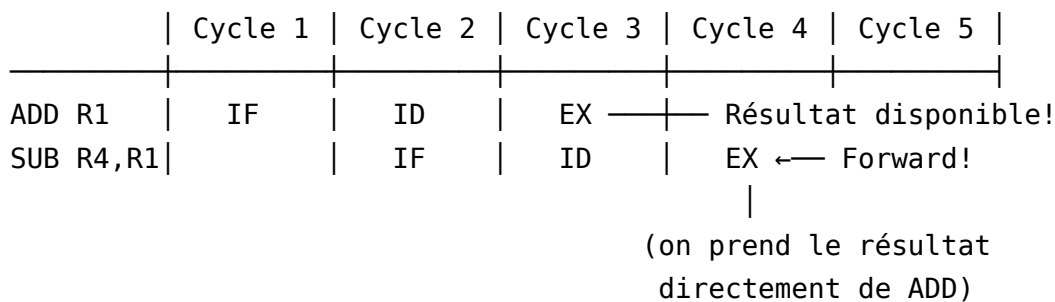


Figure 7.14: Aléa de données (Data Hazard)

Le problème : SUB lit R1 au cycle 3 (étage ID), mais ADD n'écrit R1 qu'au cycle 5 (étage WB). SUB va lire l'**ancienne** valeur de R1 !

7.11.5.2 Solution 1 : Le Forwarding (Bypass)

Au lieu d'attendre que R1 soit écrit dans le banc de registres, on peut **transférer** le résultat directement depuis l'étage EX ou MEM vers l'étage où on en a besoin.



Le ForwardUnit détecte ces situations et redirige les données :

ForwardUnit

Entrées:

ex_rn, ex_rm : registres sources de l'instruction en EX
 mem_rd : registre destination en MEM
 mem_reg_write : MEM va écrire un registre?
 wb_rd : registre destination en WB
 wb_reg_write : WB va écrire un registre?

Logique:

Si MEM.rd = EX.rn et MEM écrit → forward depuis MEM
 Sinon si WB.rd = EX.rn et WB écrit → forward depuis WB
 Sinon → pas de forwarding

Sorties (2 bits chacune):

forward_a : 00=rien, 01=depuis MEM, 10=depuis WB
 forward_b : 00=rien, 01=depuis MEM, 10=depuis WB

7.11.5.3 Solution 2 : Le Stall (pour Load-Use)

Le forwarding ne résout pas tous les cas. Considérons :

```
LDR R1, [R2]      ; Charge R1 depuis la mémoire
ADD R4, R1, R5     ; Utilise R1 immédiatement!
```

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
LDR R1	IF	ID	EX	MEM ← R1 disponible ici	
ADD R4,R1		IF	ID	EX ← Besoin de R1 ici!	

Problème : ADD a besoin de R1 dans son étage EX (cycle 4), mais LDR ne lit la mémoire qu'à l'étage MEM (aussi cycle 4). On ne peut pas faire de forwarding vers le passé !

Solution : Insérer une **bulle** (stall) pour retarder ADD d'un cycle.

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6
LDR R1	IF	ID	EX	MEM → Forward possible!		
ADD R4,R1		IF	ID	STALL	EX ← Forward OK!	
Instr 3			IF	STALL	ID	EX

Le HazardDetect détecte ces situations :

HazardDetect

Entrées:

```
id_rn, id_rm      : registres sources en ID
id_rn_used        : Rn est utilisé par l'instruction?
id_rm_used        : Rm est utilisé par l'instruction?
ex_rd             : registre destination en EX
ex_mem_read       : instruction en EX est un LDR?
```

Logique:

```
Si EX est un load (ex_mem_read = 1)
ET ID utilise ce registre (id_rn = ex_rd ou id_rm = ex_rd)
→ Déclencher un STALL
```

Sortie:

```
stall : 1 = bloquer IF et ID, insérer NOP en EX
```

7.11.5.4 Aléa de Contrôle (Control Hazard)

Les branchements posent un autre problème :

```
BEQ label      ; Si égal, sauter à label
ADD R1, R2, R3 ; Cette instruction est-elle exécutée?
SUB R4, R5, R6 ; Et celle-ci?
label:
MOV R7, #42
```

	Cycle 1	Cycle 2	Cycle 3	Cycle 4
BEQ	IF	ID	EX ← On sait si on branche	
ADD		IF	ID	???
SUB			IF	???

Problème : Quand on exécute BEQ, on a déjà commencé à chercher les instructions suivantes ! Si le branchement est pris, ADD et SUB n'auraient jamais dû être exécutées.

Solution : Le **Flush**

Si le branchement est pris, on **annule** les instructions qui n'auraient pas dû être chargées :

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
BEQ	IF	ID	EX	MEM	WB
ADD		IF	ID	FLUSH	
SUB			IF	FLUSH	
MOV R7				IF	ID ...

Le signal flush met les registres de pipeline à NOP (instruction qui ne fait rien).

7.11.6 Architecture Complète du CPU Pipeline

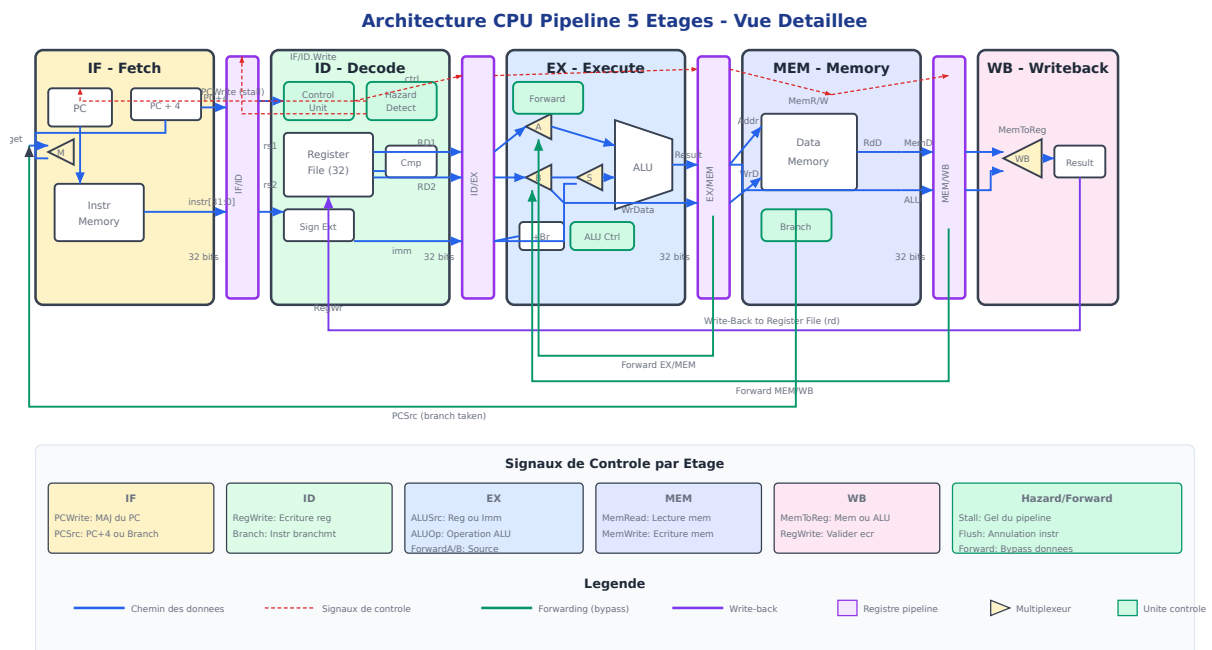


Figure 7.15: Architecture du CPU Pipeline

7.11.7 Exercices Pratiques : Projet 6

Le **Projet 6 : CPU Pipeline** vous permet de construire ces composants.

7.11.7.1 Exercice 1 : IF_ID_Reg

Objectif : Implémenter le registre de pipeline IF/ID.

Comportement :

1. Sur reset='1' OU flush='1' : mettre l'instruction à NOP (0xE0000000)
2. Sur stall='1' : garder les valeurs actuelles
3. Sinon : capturer les nouvelles valeurs

Squelette :

```
architecture rtl of IF_ID_Reg is
    signal instr_reg : bits(31 downto 0);
    signal pc_plus4_reg : bits(31 downto 0);
begin
    process(clk)
    begin
```

```

    if rising_edge(clk) then
        if (reset = '1') or (flush = '1') then
            instr_reg <= x"E0000000"; -- NOP
            pc_plus4_reg <= x"00000000";
        elsif stall = '0' then
            instr_reg <= if_instr;
            pc_plus4_reg <= if_pc_plus4;
        end if;
        -- Si stall='1', on ne fait rien (garde les valeurs)
    end if;
end process;

id_instr <= instr_reg;
id_pc_plus4 <= pc_plus4_reg;
end architecture;

```

7.11.7.2 Exercice 2 : HazardDetect

Objectif : Détecter les aléas load-use.

Logique :

```

rn_hazard = ex_mem_read AND id_rn_used AND (id_rn = ex_rd)
rm_hazard = ex_mem_read AND id_rm_used AND (id_rm = ex_rd)
stall = rn_hazard OR rm_hazard

```

En HDL (attention : pas de when...else, utiliser la logique booléenne) :

```

architecture rtl of HazardDetect is
    signal rn_hazard : bit;
    signal rm_hazard : bit;
begin
    rn_hazard <= ex_mem_read and id_rn_used and (id_rn = ex_rd);
    rm_hazard <= ex_mem_read and id_rm_used and (id_rm = ex_rd);
    stall <= rn_hazard or rm_hazard;
end architecture;

```

7.11.7.3 Exercice 3 : ForwardUnit

Objectif : Générer les signaux de forwarding.

Encodage :

- 00 : Pas de forwarding
- 01 : Forward depuis MEM
- 10 : Forward depuis WB

Logique :

```

mem_fwd_a = mem_reg_write AND (mem_rd = ex_rn)
wb_fwd_a = wb_reg_write AND (wb_rd = ex_rn) AND (NOT mem_fwd_a)
forward_a = wb_fwd_a & mem_fwd_a    (concaténation de bits)

```

En HDL :

```

architecture rtl of ForwardUnit is
    signal mem_fwd_a, wb_fwd_a : bit;
    signal mem_fwd_b, wb_fwd_b : bit;
begin
    mem_fwd_a <= mem_reg_write and (mem_rd = ex_rn);
    wb_fwd_a <= wb_reg_write and (wb_rd = ex_rn) and (not mem_fwd_a);

    mem_fwd_b <= mem_reg_write and (mem_rd = ex_rm);
    wb_fwd_b <= wb_reg_write and (wb_rd = ex_rm) and (not mem_fwd_b);

    forward_a <= wb_fwd_a & mem_fwd_a;
    forward_b <= wb_fwd_b & mem_fwd_b;
end architecture;

```

7.11.7.4 Exercice 4 : CPU_Pipeline (Projet Final)

C'est le grand défi ! Assembler tous les composants en un CPU pipeliné complet.

Conseil : L'implémentation de référence est dans `hdl_lib/05_cpu/CPU_Pipeline.hdl` (~450 lignes).

7.11.8 Comment Tester le CPU Pipeline HDL

Pour tester le CPU Pipeline en HDL, créez un script de test `.tst` :

Fichier CPU_Pipeline_test.tst :

```

-- Test du CPU Pipeline
load CPU_Pipeline.hdl;

-- Charger les composants requis
load IF_ID_Reg.hdl;
load ID_EX_Reg.hdl;
load EX_MEM_Reg.hdl;
load MEM_WB_Reg.hdl;

```



```
load HazardDetect.hdl;
load ForwardUnit.hdl;
load Decoder.hdl;
load Control.hdl;
load CondCheck.hdl;
load RegFile16.hdl;
load ALU32.hdl;
load Shifter32.hdl;
load Add32.hdl;
load PC.hdl;
load Mux32.hdl;

-- Initialiser
set reset 1;
tick; tock;
set reset 0;

-- Charger une instruction ADD R1, R0, #5 en mémoire
set instr_data 0x22100005; -- ADD R1, R0, #5

-- Exécuter plusieurs cycles (5 cycles pour traverser le pipeline)
tick; tock; -- IF
tick; tock; -- ID
tick; tock; -- EX
tick; tock; -- MEM
tick; tock; -- WB

-- Vérifier l'état
expect halted 0;
```

Exécuter le test :

```
cargo run -p hdl_cli -- CPU_Pipeline_test.tst
```

Structure des fichiers HDL pour le pipeline :

```
hdl_lib/05_cpu/
├─ CPU_Pipeline.hdl      # CPU complet assemblé
├─ IF_ID_Reg.hdl         # Registre pipeline IF→ID
├─ ID_EX_Reg.hdl         # Registre pipeline ID→EX
├─ EX_MEM_Reg.hdl        # Registre pipeline EX→MEM
├─ MEM_WB_Reg.hdl        # Registre pipeline MEM→WB
├─ HazardDetect.hdl      # Détection load-use hazards
```

```

└─ ForwardUnit.hdl      # Bypass/forwarding
└─ Decoder.hdl          # Décodage instruction
└─ Control.hdl          # Signaux de contrôle
└─ CondCheck.hdl        # Vérification conditions

```

Observer le pipeline en action :

Pour voir les hazards et le forwarding, testez avec des instructions dépendantes :

```

-- Test de forwarding (EX→EX)
-- ADD R1, R0, #5      ; R1 = 5
-- ADD R2, R1, #3      ; R2 = R1 + 3 = 8 (forward depuis EX)
set instr_data 0x22100005; tick; tock;
set instr_data 0x22210003; tick; tock;
-- Le ForwardUnit détecte que R1 est produit par l'instruction précédente
-- et bypass la valeur directement sans attendre le writeback

-- Test de stall (load-use hazard)
-- LDR R1, [R0]        ; R1 = mem[R0]
-- ADD R2, R1, #3      ; R2 = R1 + 3 (doit attendre le load)
set instr_data 0x51100000; tick; tock;
set instr_data 0x22210003; tick; tock;
-- Le HazardDetect insère un stall car le LDR n'a pas encore
-- la valeur disponible au moment où ADD en a besoin

```

7.11.9 Résumé : Pipeline vs Single-Cycle

Aspect	Single-Cycle	Pipeline
Instructions en parallèle	1	Jusqu'à 5
Throughput	1 instr / 5 unités de temps	1 instr / 1 unité de temps
Complexité	Simple	Plus complexe
Aléas	Aucun	Data hazards, Control hazards

Aspect	Single-Cycle	Pipeline
Composants supplémentaires	Aucun	Registres pipeline, Hazard Detect, Forward Unit

7.11.10 Pour Aller Encore Plus Loin

Les vrais processeurs modernes vont bien au-delà :

- **Superscalaire** : Plusieurs pipelines en parallèle
- **Exécution dans le désordre** : Réorganiser les instructions
- **Prédiction de branchement** : Deviner si un branchement sera pris
- **Cache** : Mémoire ultra-rapide proche du CPU

Mais ces concepts dépassent le cadre de ce livre. Le pipeline à 5 étages reste la base sur laquelle tout le reste est construit !

7.12 Le Lien avec la Suite

Félicitations ! Vous venez de construire un ordinateur complet.

Ce CPU que vous avez construit peut maintenant :

- Exécuter des programmes écrits en assembleur (Chapitre 6)
- Exécuter des programmes compilés depuis C32 (Chapitre 7-8)
- Faire tourner un système d'exploitation minimal (Chapitre 9)

7.12.1 Le parcours complet

Chapitre 1-5 : MATÉRIEL

NAND → Portes → ALU → Mémoire → CPU

↓

Chapitre 6-9 : LOGICIEL

Assembleur → Compilateur → Langage C32 → OS

À partir de maintenant, nous passons du côté **logiciel**. Le matériel est terminé !

7.13 Ce qu'il faut retenir

1. **Le CPU orchestre tout** : Fetch → Decode → Execute → Memory → Writeback
2. **Le décodeur analyse** : 32 bits → signaux individuels
3. **L'unité de contrôle décide** : Quels composants activer
4. **Les multiplexeurs routent** : Les données entre composants
5. **Les drapeaux permettent les conditions** : NZCV → CondCheck → ok/pas ok
6. **Single-cycle = simple** : Tout en un cycle (mais lent en vrai)

Prochaine étape : Au Chapitre 6, nous construirons l'**Assembleur** — le programme qui traduit le code assembleur en binaire exécutable par votre CPU.

Conseil : Si vous avez réussi le CPU, vous avez accompli quelque chose de remarquable. Prenez le temps de savourer : vous avez construit un ordinateur complet, de la porte NAND au processeur fonctionnel !

7.14 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

7.14.1 Questions de compréhension

- Q1.** Quelles sont les 5 étapes du cycle d'exécution d'une instruction ?
- Q2.** Quel est le rôle du décodeur d'instructions ?
- Q3.** Pourquoi les branchements conditionnels dépendent-ils des drapeaux NZCV ?
- Q4.** Dans un CPU pipeline 5 étages, qu'est-ce qu'un hazard de données ?
- Q5.** Quelle est la différence entre un CPU single-cycle et un CPU pipeliné ?

7.14.2 Mini-défi pratique

Tracez l'exécution de cette instruction dans le CPU :

```
ADD R1, R2, R3
```

Décrivez ce qui se passe à chaque étape (Fetch, Decode, Execute, Memory, Writeback).

*Les solutions se trouvent dans le document **Codex_Solutions**.*

7.14.3 Checklist de validation

Avant de passer au chapitre 6, assurez-vous de pouvoir :

- ☐ Décrire les 5 étapes d'exécution d'une instruction
- ☐ Expliquer le rôle du décodeur et de l'unité de contrôle
- ☐ Tracer le chemin des données pour ADD, LDR, et B
- ☐ Comprendre pourquoi les MUX sont essentiels (choix des sources)
- ☐ Expliquer le concept de pipeline et ses avantages

8 L'Assembleur

“Traduire, c’est trahir ?” — Pas ici.

Dans les chapitres précédents, nous avons conçu le matériel capable d’exécuter des instructions 32 bits. Mais écrire un programme en hexadécimal (comme 0xE2801001) est extrêmement pénible et source d’erreurs.

L'**Assembleur** est l’outil logiciel qui fait le pont entre le programmeur et la machine. Il traduit un fichier texte contenant des mnémoniques lisibles (ex: ADD R1, R1, #1) en un fichier binaire exécutable par le CPU.

8.1 Où en sommes-nous ?

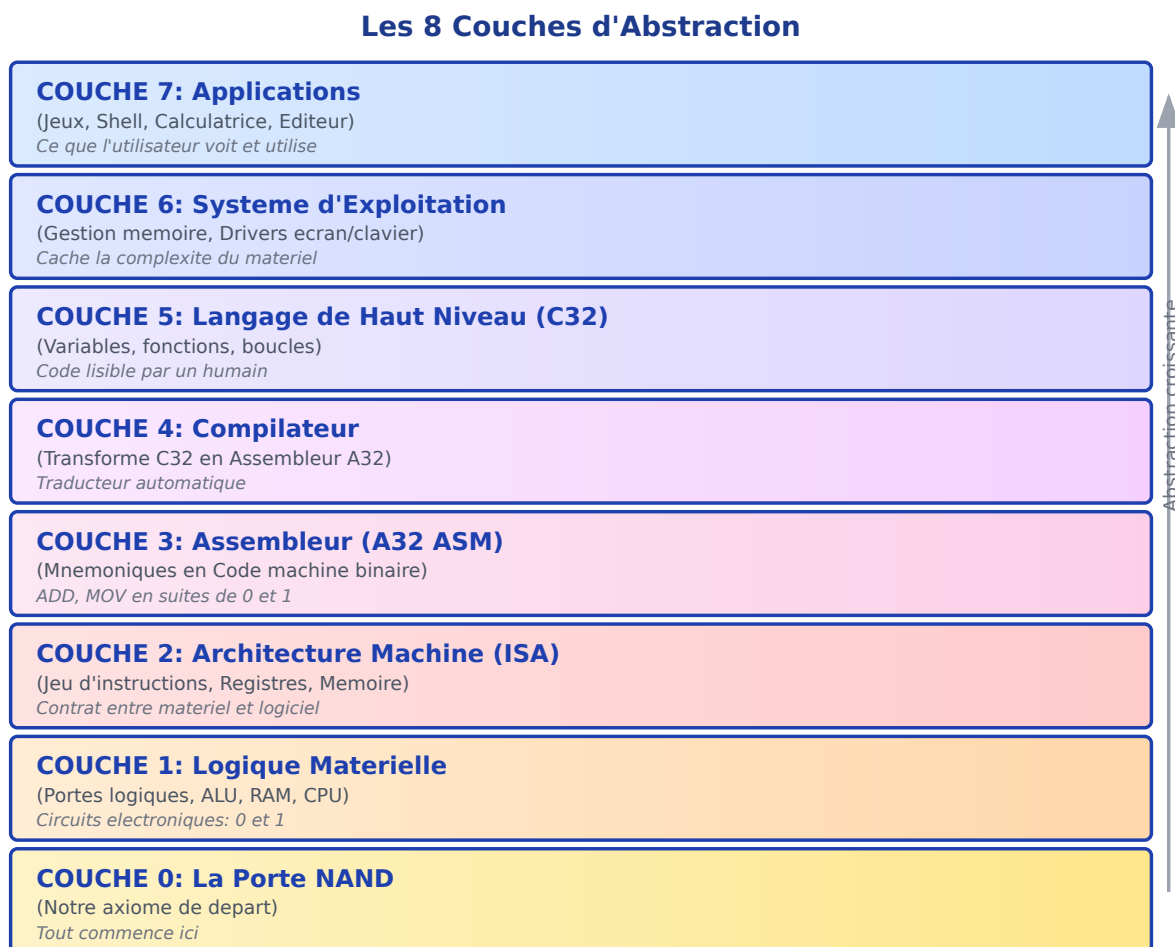


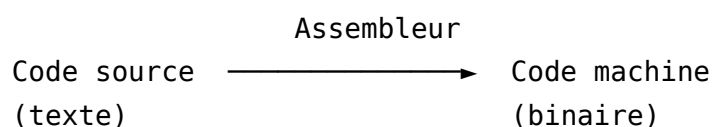
Figure 8.1: Position dans l'architecture

Nous sommes à la Couche 3 : Assembleur - Du texte au binaire

Nous entrons maintenant dans le monde du **logiciel** ! L'assembleur est le premier programme que nous construisons pour notre machine.

8.2 Le Rôle de l'Assembleur

8.2.1 Du texte au binaire



```
ADD R1, R2, #10    →    0xE2821010
B loop             →    0xEAFFFFFEE
```

8.2.2 Les trois tâches de l'assembleur

1. **Analyse (Parsing)** : Lire le code source et comprendre les instructions, les opérandes, les labels.
2. **Résolution des Symboles** : Transformer les étiquettes (labels) comme loop : en adresses numériques.
3. **Encodage** : Transformer chaque instruction en son équivalent binaire de 32 bits selon la spécification de l'ISA.

8.3 La Stratégie des Deux Passes

8.3.1 Pourquoi deux passes ?

Regardez ce code :

```
B suite    ; Où est 'suite' ? On ne le sait pas encore !
MOV R0, #1
suite:
ADD R0, R0, #1
```

À la ligne 1, l'assembleur ne sait pas encore où est suite. C'est le problème des **références vers l'avant**.

8.3.2 Passe 1 : Construction de la Table des Symboles

L'assembleur parcourt le fichier et note l'adresse de chaque label :

```
Adresse 0x0000 : B suite           (4 octets)
Adresse 0x0004 : MOV R0, #1        (4 octets)
Adresse 0x0008 : suite:            ← On note : suite = 0x0008
Adresse 0x0008 : ADD R0, R0, #1
```

Table des symboles : { "suite": 0x00000008 }

8.3.3 Passe 2 : Génération du Code

L'assembleur reparcourt le fichier. Quand il voit `B suite`, il regarde dans sa table et génère l'offset correct.

`B suite` → `offset = (0x0008 - 0x0000 - 8) / 4 = -2 → 0xEFFFFFFE`

8.4 Sections et Directives

8.4.1 Les Sections

Un programme n'est pas fait que d'instructions. Il contient aussi des données.

Section	Contenu
<code>.text</code>	Le code (les instructions) — généralement en lecture seule
<code>.data</code>	Les variables globales initialisées
<code>.bss</code>	Les variables globales non initialisées (mises à zéro)

8.4.2 Les Directives

Les directives (commençant par `.`) guident l'assembleur :

Directive	Signification
<code>.text</code>	Début de la section code
<code>.data</code>	Début de la section données
<code>.global _start</code>	Exporte le symbole <code>_start</code>
<code>.word 123</code>	Réserve 4 octets avec la valeur 123
<code>.asciz "Hello"</code>	Chaîne terminée par un zéro
<code>.align 2</code>	Aligne sur un multiple de 4 octets
<code>.ltorg</code>	Force l'émission du literal pool

8.5 Exemple d'Encodage

Comment l'assembleur encode-t-il `ADD R1, R2, #10` ?

8.5.1 Étape 1 : Identifier l'instruction

- **Mnémonique** : `ADD` → opcode = 0011
- **Registres** : Rd = R1, Rn = R2
- **Immédiat** : #10

8.5.2 Étape 2 : Déterminer la classe

- Classe 001 car on utilise un immédiat

8.5.3 Étape 3 : Assembler les bits

31-28	27-25	24-21	20	19-16	15-12	11-0
Cond	Class	Op	S	Rn	Rd	Imm12
1110	001	0011	0	0010	0001	000000001010

= 0xE2821010

8.6 La Gestion des Grandes Constantes

8.6.1 Le problème

Une instruction fait 32 bits. Un immédiat fait 12 bits maximum. Comment charger 0xDEADBEEF (32 bits) dans un registre ?

8.6.2 La solution : Le Literal Pool

L'assembleur offre une syntaxe magique : `LDR R0, =0xDEADBEEF`

Ce n'est **pas** une vraie instruction LDR — c'est une **pseudo-instruction** que l'assembleur transforme :

1. La valeur 0xDEADBEEF est stockée dans le **literal pool** (une zone de données après le code)
2. L'instruction est remplacée par `LDR R0, [PC, #offset]` qui va chercher la valeur

```
; Code source
    LDR R0, =0xDEADBEEF

; Ce que l'assembleur génère
    LDR R0, [PC, #8]    ; Va chercher la valeur 8 octets plus loin
    ...
literal_pool:
    .word 0xDEADBEEF    ; La valeur est stockée ici
```

8.7 Exercices Pratiques

8.7.1 Exercices sur le Simulateur Web

Tous les exercices de la section **A32 Assembly** du simulateur web vous font pratiquer l'écriture d'assembleur. L'assembleur intégré traduit votre code en binaire automatiquement.

Catégorie	Exercices
Basique	Hello World, Addition, Soustraction, Logique
Contrôle	Conditions, Boucles, Multiplication, Fibonacci
Mémoire	Tableaux, Maximum Tableau, Fonctions
Graphique	Pixel, Ligne, Rectangle, Damier
Avancé	Recherche Dichotomique, Dégradé

8.7.2 Exercice manuel : Encodage

Traduisez ces instructions en binaire (32 bits) :

1. MOV R0, #5
2. SUB R1, R1, #1
3. B -2 (saut de 2 instructions en arrière)

8.7.3 Exercice : Table des symboles

Calculez l'adresse de chaque label :

```
.text
start:
    MOV R0, #0
loop:
    CMP R0, #10
    BEQ end
    ADD R0, R0, #1
    B loop
end:
    HALT
```

8.7.4 Utilisation de l'outil CLI

```
# Assembler un fichier
cargo run -p a32_cli -- assemble mon_prog.s -o mon_prog.bin

# Examiner le binaire généré
hexdump -C mon_prog.bin
```

8.8 Ce qu'il faut retenir

1. **L'assembleur traduit** : Texte lisible → Binaire exécutable
2. **Deux passes** : D'abord collecter les symboles, puis générer le code
3. **Les directives organisent** : .text, .data, .word, .asciz
4. **Le literal pool résout** : Les constantes 32 bits via LDR R0, =value
5. **Un symbole = une adresse** : Les labels deviennent des nombres

Prochaine étape : Au Chapitre 7, nous construirons un **Compilateur** qui traduit du code C32 en assembleur. C'est l'étape suivante vers l'abstraction !

Conseil : Faites beaucoup d'exercices en assembleur. Plus vous serez à l'aise avec l'assembleur, mieux vous comprendrez ce que fait le compilateur.

8.9 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

8.9.1 Questions de compréhension

- Q1.** Pourquoi l'assembleur fait-il deux passes sur le code source ?
- Q2.** Quelle est la différence entre `.text` et `.data` ?
- Q3.** Comment l'assembleur gère-t-il `LDR R0, =0xDEADBEEF` ?
- Q4.** Que se passe-t-il si un branchement est trop loin (offset > 24 bits) ?
- Q5.** Qu'est-ce qu'un fichier binaire A32B contient exactement ?

8.9.2 Mini-défi pratique

Encodez manuellement cette instruction en binaire 32 bits :

```
ADD R1, R2, R3
```

Indice : Format ALU registre = [cond:4][000][opcode:4][S][Rn:4][Rd:4][00000000][Rm:4]

*Les solutions se trouvent dans le document **Codex_Solutions**.*

8.9.3 Checklist de validation

Avant de passer au chapitre 7, assurez-vous de pouvoir :

- ☐ Expliquer le rôle des deux passes de l'assembleur
- ☐ Utiliser les directives `.text`, `.data`, `.word`, `.asciz`
- ☐ Comprendre le fonctionnement du literal pool
- ☐ Encoder une instruction simple à la main (format binaire)
- ☐ Lire et interpréter un hexdump de fichier binaire

9 Construction du Compilateur

“Pour comprendre la récursivité, il faut d’abord comprendre la récursivité.”

Dans ce chapitre, nous allons construire le **pont** entre le langage de haut niveau (C32) et l’assembleur (A32). Le compilateur est l’outil qui permet aux humains d’écrire du code lisible tout en profitant de la vitesse du code machine.

9.1 Où en sommes-nous ?

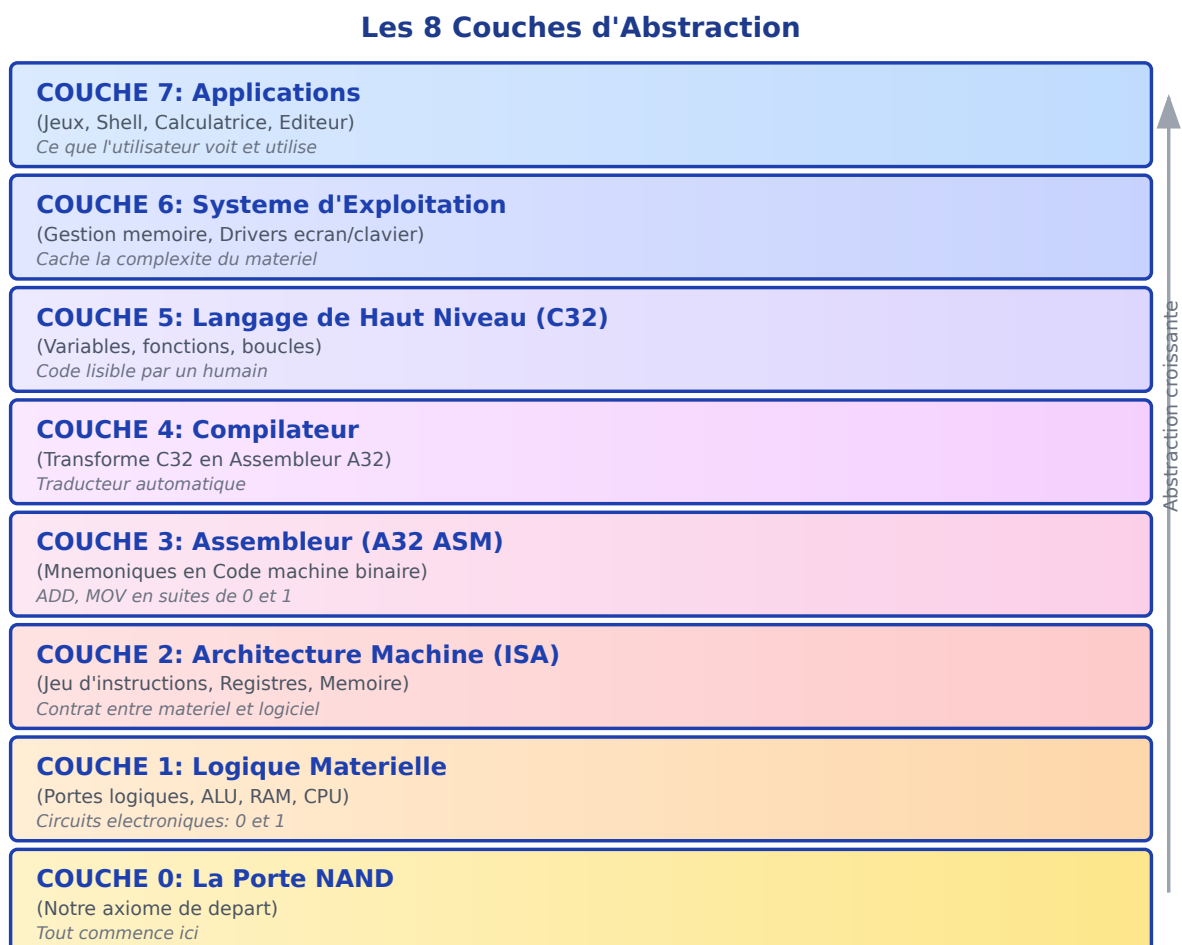


Figure 9.1: Position dans l’architecture

Nous sommes à la Couche 4 : Compilateur - Transforme C32 en Assembleur A32

Le compilateur est le **traducteur automatique** qui transforme du code lisible par les humains en code exécutable par la machine.

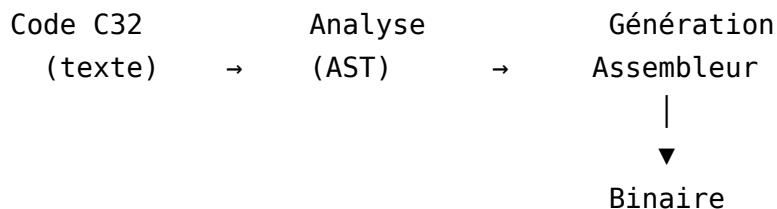
9.2 Le Rôle du Compilateur

9.2.1 Pourquoi un compilateur ?

Assembleur	C32 (haut niveau)
ADD R0, R0, #1	x = x + 1;
Gestion manuelle des registres	Variables nommées
Sauts et labels	if, while, for
Appels manuels avec conventions	return fonction();

Le compilateur traduit le second en premier, automatiquement.

9.2.2 Les étapes de compilation



1. **Analyse lexicale** : Découpe le texte en tokens (int, x, =, 5, ;)
2. **Analyse syntaxique** : Construit un arbre (AST) représentant la structure
3. **Analyse sémantique** : Vérifie les types, les portées, etc.
4. **Génération de code** : Produit l'assembleur équivalent

9.3 Les Phases du Compilateur

9.3.1 Phase 1 : Lexer (Analyse Lexicale)

Le lexer transforme le flux de caractères en tokens :

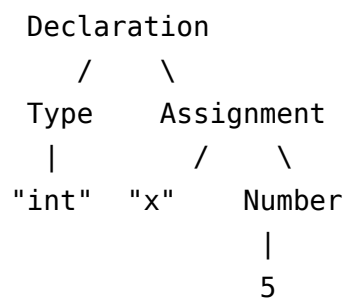
```
int x = 5;
```

Devient :

```
[INT] [ID:"x"] [EQUAL] [NUMBER:5] [SEMICOLON]
```

9.3.2 Phase 2 : Parser (Analyse Syntaxique)

Le parser construit un **AST** (Abstract Syntax Tree) :



9.3.3 Phase 3 : Génération de Code

Le générateur parcourt l'AST et produit l'assembleur :

```
; int x = 5;
MOV R0, #5
STR R0, [SP, #-4]! ; Push x sur la pile
```

9.4 Compilation des Structures de Contrôle

9.4.1 Variables locales

Les variables locales vivent sur la **pile** :

```
int a = 10;
int b = 20;
```



```
; Prologue
SUB SP, SP, #8      ; Réserve 8 octets pour a et b

; a = 10
MOV R0, #10
STR R0, [SP, #4]    ; a est à SP+4

; b = 20
MOV R0, #20
STR R0, [SP, #0]    ; b est à SP+0
```

9.4.2 Expressions

Pour $a + b * 2$:

```
; Évaluation de b * 2
LDR R0, [SP, #0]    ; R0 = b
MOV R1, #2
MUL R0, R0, R1      ; R0 = b * 2

; Évaluation de a + (b * 2)
LDR R1, [SP, #4]    ; R1 = a
ADD R0, R1, R0      ; R0 = a + b*2
```

9.4.3 If / Else

```
if (x > 0) {
    y = 1;
} else {
    y = 0;
}
```

```
LDR R0, [SP, #x_offset]
CMP R0, #0
BLE else_label

; Then branch
MOV R0, #1
STR R0, [SP, #y_offset]
B endif_label

else_label:
; Else branch
MOV R0, #0
```

```
    STR R0, [SP, #y_offset]

endif_label:
```

9.4.4 Boucle While

```
while (i < 10) {
    i = i + 1;
}
```

```
while_start:
    LDR R0, [SP, #i_offset]
    CMP R0, #10
    BGE while_end

    ; Corps de la boucle
    ADD R0, R0, #1
    STR R0, [SP, #i_offset]

    B while_start

while_end:
```

9.4.5 Boucle For

for (init; cond; incr) { body } est équivalent à :

```
init;
while (cond) {
    body;
    incr;
}
```

9.5 Compilation des Fonctions

9.5.1 Convention d'appel

Comment passe-t-on les arguments ? Comment retourne-t-on une valeur ?

Registre	Rôle
R0-R3	Arguments 1-4, valeur de retour en R0
R4-R11	Sauvegardés par l'appelé (callee-saved)
R13 (SP)	Pointeur de pile
R14 (LR)	Adresse de retour

9.5.2 Prologue et Épilogue

```
int add(int a, int b) {  
    return a + b;  
}
```

```
add:  
    ; Prologue  
    SUB SP, SP, #4      ; Place pour LR  
    STR LR, [SP]        ; Sauvegarde LR  
  
    ; Corps : a est dans R0, b dans R1  
    ADD R0, R0, R1      ; Résultat dans R0  
  
    ; Épilogue  
    LDR LR, [SP]        ; Restaure LR  
    ADD SP, SP, #4  
    MOV PC, LR          ; Retour
```

9.5.3 Appel de fonction

```
result = add(5, 3);
```

```
MOV R0, #5          ; Premier argument  
MOV R1, #3          ; Deuxième argument  
BL add              ; Appel (sauve PC+4 dans LR)  
STR R0, [SP, #result_offset] ; Sauve le résultat
```

9.6 Le Compilateur C32

Le compilateur C32 du projet Codex (c32_cli) implémente toutes ces transformations.

9.6.1 Utilisation

```
# Compiler un fichier C32 en assembleur
cargo run -p c32_cli -- mon_fichier.c -o mon_fichier.s

# Compiler directement en binaire
cargo run -p c32_cli -- mon_fichier.c -o mon_fichier.bin
```

9.6.2 Exemple complet

```
// fibonacci.c
int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}

int main() {
    return fib(10);
}
```

Produit de l'assembleur avec : - Gestion automatique de la pile - Appels récursifs - Sauvegarde/restauration des registres

9.7 Construisez Votre Propre Compilateur !

Le simulateur web contient une section **Compilateur: Construction** avec **18 exercices progressifs** organisés en 7 phases. Vous construirez un mini-compilateur qui génère du vrai code assembleur A32.

9.7.1 Phase 1 : Lexer (Analyse Lexicale)

Le lexer transforme le texte en tokens.

Exercice	Description
1.1 Reconnaître un Chiffre	Implémenter <code>is_digit(c)</code> pour détecter '0'-'9'
1.2 Lire un Nombre	Parser un entier depuis une chaîne

Exercice	Description
1.3 Compter les Tokens	Compter les tokens dans "12 + 34 * 5"

9.7.2 Phase 2 : Parser (Analyse Syntaxique)

Le parser construit une représentation structurée et évalue les expressions.

Exercice	Description
2.1 Précédence des Opérateurs	Déterminer la priorité ($*$ > $+$)
2.2 Évaluer une Opération	Évaluer $3 + 4$ ou $6 * 7$
2.3 Parser avec Précédence	Descente récursive pour $2 + 3 * 4$
2.4 Parenthèses	Supporter $(2 + 3) * 4$

9.7.3 Phase 3 : Émission ASM (Génération de Code)

Générer des instructions A32 sous forme de chaînes.

Exercice	Description
3.1 Générer MOV	Produire "MOV R0, #42"
3.2 Opération Binaire	Mapper $+$ → ADD, $*$ → MUL
3.3 Comparaison	Générer CMP et codes de condition

9.7.4 Phase 4 : CodeGen Expressions

Générer du code A32 complet pour des expressions.

Exercice	Description
4.1 Constante → A32	Générer code pour charger une constante
4.2 Addition → A32	$a + b$ → MOV R0, #a / MOV R1, #b / ADD R0, R0, R1
4.3 Expression → A32	Expression complète avec précedence

9.7.5 Phase 5 : Structures de Contrôle

Générer du code pour if/else et while.

Exercice	Description
5.1 If/Else → A32	Générer les sauts conditionnels et labels
5.2 While → A32	Générer les boucles avec labels

9.7.6 Phase 6 : Fonctions

Gérer les appels de fonction et la pile.

Exercice	Description
6.1 Prologue/Épilogue	Sauvegarder LR, réserver la pile
6.2 Appel de Fonction	Passer les arguments, appeler avec BL

9.7.7 Phase 7 : Projet Final

Exercice	Description
7.1 Mini-Compilateur Complet	Compiler une expression en A32 exécutable

Le projet final combine toutes les phases : lexer → parser → codegen pour produire du code assembleur A32 fonctionnel.

9.7.8 Techniques Clés

Ces exercices utilisent la technique de **descente récursive** :

- `parse_expr()` gère + et - (basse priorité)
- `parse_term()` gère * et / (haute priorité)
- `parse_factor()` gère les nombres et les parenthèses

9.8 Exercices Pratiques

9.8.1 Exercices sur le Simulateur Web

La section **C32** du simulateur web vous permet de compiler et exécuter du C32.

Catégorie	Exercices
Bases	Variables, Expressions, Modulo, Incrémentation
Contrôle	Conditions, Else-If, Maximum de 3
Boucles	For, While, Imbriquées, Multiplication
Fonctions	Appels, Paramètres, Valeur Absolue, Min/Max
Tableaux	Accès, Maximum, Comptage
Pointeurs	Adresses, Swap, Tableaux via pointeurs
Récursion	Factorielle, Fibonacci, PGCD
Algorithmes	Tri à Bulles, Recherche Binaire

9.8.2 Exercice : Traduire manuellement

Traduisez ce code C32 en assembleur à la main :

```
int sum = 0;
for (int i = 1; i <= 10; i = i + 1) {
    sum = sum + i;
}
```

Comparez avec la sortie du compilateur !

9.9 Ce qu'il faut retenir

1. **Le compilateur traduit** : C32 (lisible) → Assembleur (exécutable)
2. **Trois phases** : Lexer → Parser → Générateur de code
3. **Les variables locales vivent sur la pile** : Accès via [SP, #offset]
4. **Les structures de contrôle deviennent des sauts** : if → CMP + B
5. **Les fonctions suivent une convention** : Arguments en R0-R3, retour en R0

Prochaine étape : Au Chapitre 8, nous explorerons le langage C32 en détail — sa syntaxe, ses types, et ses possibilités.

Conseil : Pour vraiment comprendre le compilateur, écrivez du C32 et regardez l'assembleur généré. Cherchez à prédire ce que le compilateur va produire !

9.10 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

9.10.1 Questions de compréhension

Q1. Quelles sont les trois phases principales d'un compilateur ?

Q2. Comment le compilateur gère-t-il les variables locales ?

Q3. Comment un if-else est-il traduit en assembleur ?

Q4. Quelle est la convention d'appel pour les arguments de fonction ?

Q5. Pourquoi le compilateur génère-t-il un prologue et un épilogue pour chaque fonction ?

9.10.2 Mini-défi pratique

Prédisez l'assembleur généré pour ce code C32 :

```
int double_it(int x) {  
    return x + x;  
}
```

*Les solutions se trouvent dans le document **Codex_Solutions**.*

9.10.3 Checklist de validation

Avant de passer au chapitre 8, assurez-vous de pouvoir :

- ☐ Décrire le rôle du lexer, parser, et codegen
- ☐ Expliquer comment les variables locales sont stockées
- ☐ Traduire un if-else simple en assembleur
- ☐ Connaître la convention d'appel (R0-R3, pile, LR)
- ☐ Comprendre le prologue/épilogue d'une fonction

10 Langage de Haut Niveau (C32)

“Le logiciel est l’esprit qui anime la machine.”

Jusqu’à présent, nous avons construit le matériel et appris à lui parler en assembleur. Mais écrire des applications complexes en assembleur est laborieux. C’est ici qu’intervient le **C32** — un langage de haut niveau qui vous permet de vous concentrer sur la **logique** de votre programme.

10.1 Où en sommes-nous ?

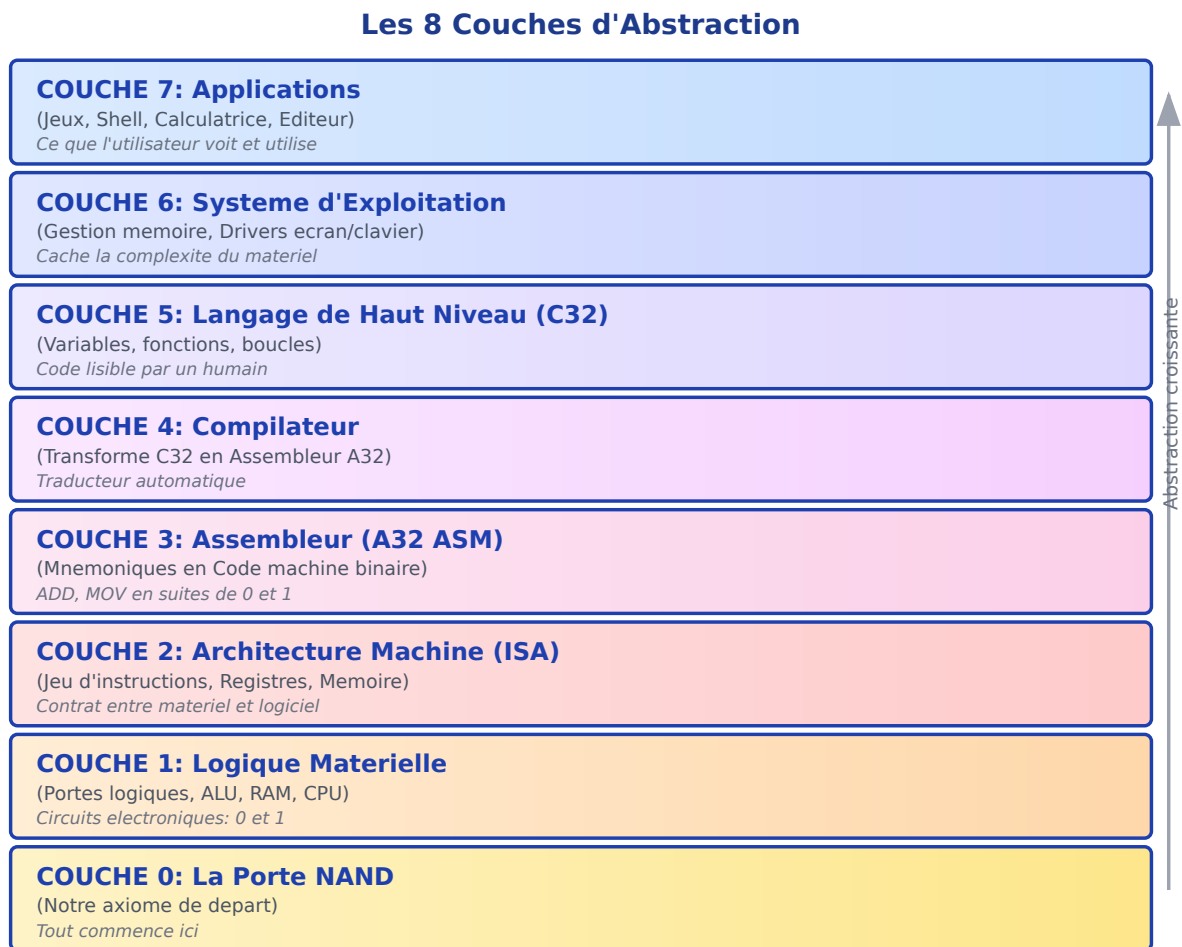


Figure 10.1: Position dans l'architecture

Nous sommes à la Couche 5 : Langage de Haut Niveau (C32) - Variables, fonctions, boucles

Le C32 est un sous-ensemble du langage C. Si vous connaissez le C, le Java ou le C++, vous vous sentirez chez vous.

10.2 Pourquoi un Langage de Haut Niveau ?

10.2.1 Le problème de l'assembleur

Assembleur

C32

```

MOV R0, #0                int sum = 0;
MOV R1, #1                for (int i = 1; i <= 10; i = i + 1) {
loop:                     sum = sum + i;
    CMP R1, #10           }
    BGT done
    ADD R0, R0, R1
    ADD R1, R1, #1
    B loop
done:

```

Le C32 est : - **Plus lisible** : Variables nommées, structures de contrôle - **Plus maintenable** : Moins de code, moins de bugs - **Portable** : Le même code peut cibler différentes architectures

10.2.2 L'abstraction

```

[ Pensée Humaine ] → "Calculer la moyenne"
      ↓
[ Langage C32 ]    → sum = sum + tab[i];
      ↓
[ Assembleur A32 ] → LDR R0, [R1, R2]; ADD R3, R3, R0...
      ↓
[ Code Machine ]   → 0xE0833000...

```

10.3 Spécification du Langage C32

10.3.1 Les Types de Données

Type	Taille	Description
int	32 bits	Entier signé (complément à 2)
uint	32 bits	Entier non-signé
char	8 bits	Caractère ASCII
bool	1 bit	true ou false
void	—	Pour les fonctions sans retour
type*	32 bits	Pointeur (adresse mémoire)

10.3.2 Variables

```
int x = 42;           // Variable globale ou locale
int tab[10];          // Tableau de 10 entiers
int* p = &x;          // Pointeur vers x
```

10.3.3 Portée des variables

- **Globales** : Déclarées hors des fonctions, accessibles partout
 - **Locales** : Déclarées dans une fonction, vivent sur la pile
-

10.4 Opérateurs

10.4.1 Arithmétiques

Opérateur	Signification
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo (reste)

10.4.2 Comparaison

Opérateur	Signification
==	Égal
!=	Différent
<	Inférieur
>	Supérieur
<=	Inférieur ou égal
>=	Supérieur ou égal

10.4.3 Logiques

Opérateur	Signification
&&	ET logique
\ \	OU logique
!	NON logique

10.4.4 Binaires

Opérateur	Signification
&	ET bit à bit
\	OU bit à bit
^	XOR bit à bit
~	Inversion
<<	Décalage gauche
>>	Décalage droite

10.5 Structures de Contrôle

10.5.1 If / Else

```
if (score > 100) {  
    win();  
} else if (score > 50) {  
    try_again();  
} else {  
    game_over();  
}
```

10.5.2 While

```
while (x < 10) {  
    x = x + 1;  
}
```

10.5.3 For

```
for (int i = 0; i < 10; i = i + 1) {  
    sum = sum + i;  
}
```

10.5.4 Do-While

```
do {  
    x = x - 1;  
} while (x > 0);
```

10.6 Fonctions

10.6.1 Définition

```
int add(int a, int b) {  
    return a + b;  
}  
  
void greet() {  
    putchar('H');  
    putchar('i');  
}
```

10.6.2 Appel

```
int result = add(5, 3);  
greet();
```

10.6.3 Récursion

```
int factorial(int n) {  
    if (n <= 1) return 1;  
    return n * factorial(n - 1);  
}
```

10.7 Pointeurs et Tableaux

10.7.1 Pointeurs

Un pointeur contient une **adresse mémoire** :

```
int x = 42;  
int* p = &x;    // p contient l'adresse de x  
*p = 100;       // x vaut maintenant 100
```

10.7.2 Tableaux

Un tableau est une suite de valeurs consécutives en mémoire :

```
int scores[5];  
scores[0] = 10;  
scores[4] = 50;
```

10.7.3 Lien entre pointeurs et tableaux

```
int tab[10];  
int* p = tab;    // Équivalent à &tab[0]  
p[3] = 42;       // Équivalent à tab[3] = 42  
*(p + 3) = 42;   // Même chose !
```

10.8 Accès au Matériel (MMIO)

10.8.1 L'écran

```
// L'écran commence à 0x00400000
// 320x240 pixels, 1 bit par pixel

void set_pixel(int x, int y) {
    uint* screen = (uint*)0x00400000;
    int offset = y * 10 + (x / 32);
    uint mask = 1 << (31 - (x % 32));
    screen[offset] = screen[offset] | mask;
}

void clear_screen() {
    uint* screen = (uint*)0x00400000;
    for (int i = 0; i < 2400; i = i + 1) {
        screen[i] = 0;
    }
}
```

10.8.2 Le clavier

```
// Le clavier est à 0x00402600

int get_key() {
    int* keyboard = (int*)0x00402600;
    return *keyboard;
}

void wait_key() {
    while (get_key() == 0) {
        // Attendre
    }
}
```

10.9 Exemple Complet


```
// Programme qui dessine un rectangle et attend une touche

extern void putchar(char c);

int main() {
    // Dessiner un rectangle 10x5 à la position (20, 30)
    uint* screen = (uint*)0x00400000;

    for (int y = 30; y < 35; y = y + 1) {
        for (int x = 20; x < 30; x = x + 1) {
            int offset = y * 10 + (x / 32);
            uint mask = 1 << (31 - (x % 32));
            screen[offset] = screen[offset] | mask;
        }
    }

    // Afficher un message
    putchar('D');
    putchar('o');
    putchar('\n');
    putchar('e');
    putchar('!');

    // Attendre une touche
    int* kbd = (int*)0x00402600;
    while (*kbd == 0) {}

    return 0;
}
```

10.10 Exercices Pratiques

10.10.1 Exercices sur le Simulateur Web

La section **C32** contient de nombreux exercices progressifs :

Catégorie	Exercices clés
Bases	Variables, Expressions, Modulo
Contrôle	Conditions, Else-If, Opérateurs Logiques
Boucles	For, While, Imbriquées

Catégorie	Exercices clés
Fonctions	Paramètres, Valeur Absolue, Min/Max
Tableaux	Accès, Maximum, Comptage
Pointeurs	Adresses, Swap
Récursion	Factorielle, Fibonacci, PGCD
Algorithmes	Tri à Bulles, Recherche Binaire
Graphique	Pixel, Ligne, Rectangle, Damier

10.10.2 Défis suggérés

1. **Hello World** : Affichez votre nom à l'écran
2. **Jeu de devinette** : Le programme choisit un nombre, l'utilisateur devine
3. **Calculatrice** : Lisez deux nombres et affichez leur somme

10.11 Les Structures (Structs)

Le C32 supporte les structures (structs) pour regrouper plusieurs variables :

10.11.1 Définition d'une structure

```
struct Point {  
    int x;  
    int y;  
};
```

10.11.2 Utilisation

```
struct Point p;    // Déclarer une variable de type struct  
p.x = 10;          // Accéder aux champs avec .  
p.y = 20;
```

10.11.3 Pointeur vers structure

```
struct Point *ptr = &p;
ptr->x = 30;          // Accéder via pointeur avec ->
ptr->y = 40;
```

10.11.4 Exemple complet

```
struct Point { int x; int y; };

int distance_sq(struct Point *p) {
    return p->x * p->x + p->y * p->y;
}

int main() {
    struct Point p;
    p.x = 3;
    p.y = 4;
    return distance_sq(&p); // Retourne 25
}
```

10.11.5 Structures imbriquées

```
struct Point { int x; int y; };
struct Rectangle {
    struct Point corner;
    int width;
    int height;
};

int main() {
    struct Rectangle r;
    r.corner.x = 0;
    r.corner.y = 0;
    r.width = 100;
    r.height = 50;
    return r.width * r.height; // Retourne 5000
}
```

10.12 Limitations du C32

Pour rester simple et pédagogique, le C32 a quelques limites :

Fonctionnalité	État
struct	Supporté
float, double	Non supporté
malloc/free	Via OS uniquement
Chaînes de caractères	Basique
Préprocesseur	Minimal

10.13 Ce qu'il faut retenir

1. **C32 simplifie la programmation** : Variables nommées, structures de contrôle
2. **Les types de base** : int, char, bool, void, pointeurs
3. **Pointeurs = adresses** : Accès direct à la mémoire
4. **MMIO** : Écran à 0x00400000, clavier à 0x00402600
5. **Fonctions** : Modularité et réutilisation du code

Prochaine étape : Au Chapitre 9, nous construirons un **Système d'Exploitation** minimal — gestion mémoire, graphiques, entrées/sorties.

Conseil : Le C32 est proche du C. Si vous voulez aller plus loin, apprenez le C — c'est le langage de base de Linux, Windows, et de presque tous les systèmes embarqués !

10.14 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

10.14.1 Questions de compréhension

- Q1.** Quelle est la différence entre `int` et `uint` en C32 ?
- Q2.** Qu'est-ce qu'un pointeur et comment l'utilise-t-on ?
- Q3.** Comment accède-t-on à l'écran en C32 ?
- Q4.** Quelle est la syntaxe d'une boucle `for` en C32 ?
- Q5.** Comment définit-on et utilise-t-on un tableau en C32 ?

10.14.2 Mini-défi pratique

Écrivez une fonction C32 qui calcule la somme des éléments d'un tableau :

```
int sum(int *arr, int len) {  
    // À compléter  
}
```

*Les solutions se trouvent dans le document **Codex_Solutions**.*

10.14.3 Checklist de validation

Avant de passer au chapitre 9, assurez-vous de pouvoir :

- ☐ Utiliser les types de base : `int`, `uint`, `char`, `bool`
- ☐ Déclarer et utiliser des pointeurs
- ☐ Accéder à la mémoire mappée (écran, clavier)
- ☐ Écrire des boucles `for` et `while`
- ☐ Manipuler des tableaux et des chaînes de caractères

11 Système d'Exploitation

“Un OS est ce qui reste quand on a enlevé tout ce qui est utile.” — Ken Thompson

Félicitations ! Vous avez construit le matériel, l'assembleur et le compilateur. Votre machine Codex est fonctionnelle. Mais pour l'instant, chaque programmeur doit réinventer la roue : comment dessiner un cercle ? comment lire une chaîne de caractères ?

Dans ce dernier chapitre, nous allons construire une **Bibliothèque Système** qui simplifie l'accès au matériel.

11.1 Où en sommes-nous ?

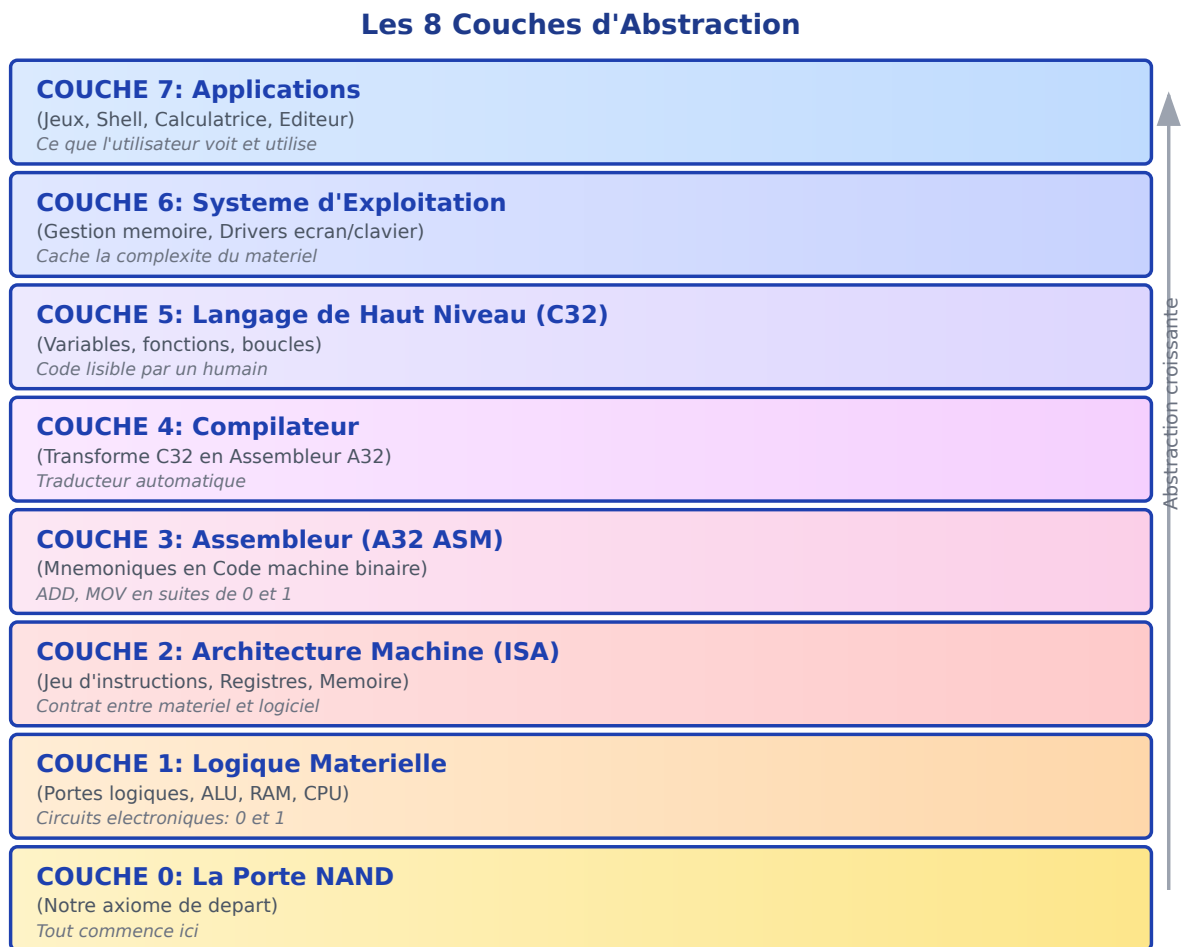


Figure 11.1: Position dans l'architecture

Nous sommes à la Couche 6 : Système d'Exploitation - Le sommet de la pyramide logicielle !

C'est le **sommet de la pyramide** logicielle ! L'OS cache la complexité du matériel et offre des services de haut niveau aux applications.

11.2 Qu'est-ce qu'un Système d'Exploitation ?

11.2.1 La hiérarchie logicielle

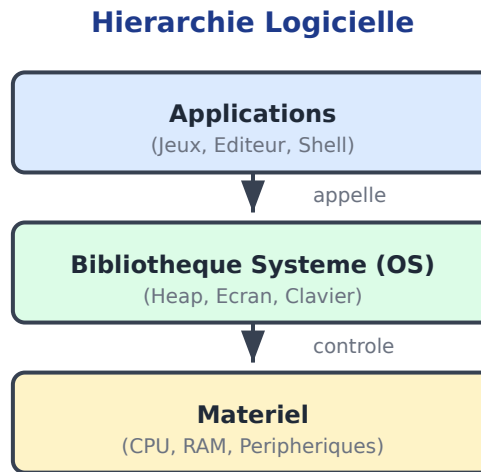


Figure 11.2: Hiérarchie logicielle

11.2.2 Ce que fait un OS

Fonction	Description
Gestion mémoire	Allouer/libérer de la mémoire dynamiquement
Graphiques	Dessiner des formes, du texte
Entrées/Sorties	Lire le clavier, afficher à l'écran
Fichiers	Sauvegarder/charger des données (avancé)
Multitâche	Exécuter plusieurs programmes (avancé)

Notre OS Codex implémente les trois premiers points.

11.3 Gestion de la Mémoire (Le Tas / Heap)

11.3.1 Le problème

Jusqu'à présent, nous utilisons :

- **Variables globales** : Taille fixée à la compilation

- **Variables locales** : Sur la pile, libérées automatiquement

Mais que faire si on veut allouer une taille **inconnue à l'avance** ?

11.3.2 L'allocateur "Bump" (Simple)

L'allocateur le plus simple : un pointeur qui avance à chaque allocation.

```
char* heap_ptr = (char*)HEAP_START;

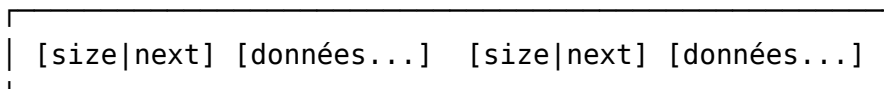
char* malloc(int size) {
    char* result = heap_ptr;
    heap_ptr = heap_ptr + size;
    return result;
}

void free(char* ptr) {
    // Ne fait rien ! La mémoire n'est jamais récupérée.
}
```

Avantage : Très simple, très rapide. **Inconvénient** : On ne peut pas réutiliser la mémoire libérée.

11.3.3 L'allocateur par Liste Chaînée (Avancé)

Pour réutiliser la mémoire, chaque bloc contient : - Sa taille - Un pointeur vers le bloc libre suivant



Quand on libère un bloc, on l'ajoute à la liste des blocs libres. Quand on alloue, on cherche un bloc de taille suffisante.

11.4 Bibliothèque Graphique

11.4.1 Le problème

Pour allumer un pixel, il faut :

1. Calculer l'adresse de l'octet
2. Calculer la position du bit
3. Faire un OR pour allumer (ou AND + NOT pour éteindre)

C'est fastidieux et source d'erreurs.

11.4.2 Les fonctions graphiques

```
void screen_init();
void screen_clear();
void screen_set_color(int color);

void screen_draw_pixel(int x, int y);
void screen_draw_line(int x1, int y1, int x2, int y2);
void screen_draw_rect(int x, int y, int w, int h);
void screen_draw_circle(int cx, int cy, int r);
void screen_print(char* text, int x, int y);
```

11.4.3 L'algorithme de Bresenham

Pour dessiner des lignes droites avec uniquement des additions et des comparaisons:

```
void screen_draw_line(int x1, int y1, int x2, int y2) {
    int dx = abs(x2 - x1);
    int dy = abs(y2 - y1);
    int sx = x1 < x2 ? 1 : -1;
    int sy = y1 < y2 ? 1 : -1;
    int err = dx - dy;

    while (1) {
        screen_draw_pixel(x1, y1);
        if (x1 == x2 && y1 == y2) break;
        int e2 = 2 * err;
        if (e2 > -dy) { err = err - dy; x1 = x1 + sx; }
        if (e2 < dx) { err = err + dx; y1 = y1 + sy; }
    }
}
```

Pas de multiplication, pas de division — uniquement des opérations que notre CPU fait rapidement !

11.5 Entrées / Sorties

11.5.1 Printf simplifié

```
void putchar(char c) {
    // Appel système SVC pour afficher un caractère
}

void puts(char* s) {
    while (*s != 0) {
        putchar(*s);
        s = s + 1;
    }
}

void print_int(int n) {
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n >= 10) {
        print_int(n / 10);
    }
    putchar('0' + (n % 10));
}
```

11.5.2 Lecture du clavier

```
int keyboard_read() {
    int* kbd = (int*)0x00402600;
    return *kbd;
}

int keyboard_wait() {
    int key;
    while ((key = keyboard_read()) == 0) {
        // Attendre
    }
    return key;
}
```

11.6 Interruptions et Timer (Concepts)

11.6.1 Le problème du polling

Dans notre approche actuelle, le CPU vérifie constamment le clavier :

```
while (keyboard_read() == 0) {} // CPU occupe à ne rien faire !
```

11.6.2 Les interruptions

Avec les interruptions, le matériel **signale** au CPU qu'un événement s'est produit:

1. Le CPU exécute le programme
2. Une touche est pressée → Le matériel déclenche une **interruption**
3. Le CPU s'arrête et saute vers le **handler d'interruption**
4. Le handler traite l'événement
5. Le CPU reprend le programme là où il s'était arrêté

C'est la base du multitâche et des systèmes réactifs !

11.7 Applications Démo

Le répertoire demos/ contient des exemples complets :

Démo	Description
01_hello	Hello World classique
02_counter	Compteur avec affichage
03_graphics	Dessins géométriques
04_snake	Jeu du serpent complet !
05_shell	Interface en ligne de commande

11.7.1 Compiler et exécuter une démo

```
# Compiler
cargo run -p c32_cli -- demos/04_snake/main.c -o snake.bin

# Exécuter
cargo run -p a32_runner -- snake.bin
```

Ou utilisez le **Simulateur Web** pour une expérience visuelle !

11.8 Exercices Pratiques

11.8.1 Exercices sur le Simulateur Web

La section **OS** contient des exercices interactifs :

Exercice	Description
Calculatrice	Calculatrice interactive avec clavier
Variables	Shell pour définir des variables
Timer	Affichage d'un compteur animé
Scheduler	Simulation d'un ordonnanceur
Projet Mini-OS	Shell multi-applications
Task Manager	Gestionnaire de tâches visuel

11.8.2 Défis suggérés

1. **Étendre l'OS** : Ajoutez `screen_draw_triangle()` en utilisant trois appels à `screen_draw_line()`.
 2. **Gestion mémoire** : Testez l'allocateur avec plusieurs `malloc()` et observez le comportement.
 3. **Projet final** : Créez votre propre application dans `demos/` en utilisant tout ce que vous avez appris !
-

11.9 Le Parcours Complet

Vous avez parcouru tout le chemin :

Chapitre 0 : Introduction

↓

Chapitre 1-5 : MATÉRIEL

NAND → Portes → ALU → Mémoire → CPU

↓

Chapitre 6-9 : LOGICIEL

Assembleur → Compilateur → C32 → OS

↓

Applications : Jeux, Calculatrices, Shell...

Vous comprenez maintenant que l'ordinateur n'est pas une boîte magique, mais une **pyramide d'abstractions** magnifiquement ordonnées.

11.10 Ce qu'il faut retenir

1. **L'OS cache le matériel** : `draw_circle()` au lieu de manipuler des bits
2. **Gestion mémoire dynamique** : `malloc()` et `free()`
3. **Bibliothèque graphique** : Lignes, rectangles, cercles, texte
4. **Entrées/Sorties** : `putchar()`, `keyboard_read()`
5. **Interruptions** : Le matériel peut signaler des événements au CPU

11.11 Félicitations !

Vous avez parcouru tout le chemin, de la **porte NAND** au **système d'exploitation**.
Vous comprenez maintenant :

- Comment les bits deviennent des calculs (ALU)
- Comment les calculs deviennent de la mémoire (RAM)
- Comment la mémoire devient un programme (CPU)
- Comment un programme devient une application (Compilateur + OS)

Quand vous verrez du code s'exécuter, vous saurez **exactement** ce qui se passe dans la machine. Ce n'est plus de la magie — c'est de l'ingénierie que vous maîtrisez.

Et maintenant ?

- Apprenez le C pour approfondir la programmation système
 - Étudiez le noyau Linux pour voir un vrai OS
 - Explorez les architectures ARM/RISC-V modernes
 - Construisez vos propres projets sur la base de Codex !
-

11.12 Auto-évaluation

Testez votre compréhension de l'ensemble du parcours.

11.12.1 Questions de compréhension

- Q1.** Qu'est-ce qu'un allocateur mémoire et pourquoi en a-t-on besoin ?
- Q2.** Comment fonctionne l'affichage graphique via MMIO ?
- Q3.** Quelle est la différence entre polling et interruptions ?
- Q4.** Pourquoi sépare-t-on le code en bibliothèques (Math, String, Screen, etc.) ?
- Q5.** Résumez le chemin complet du code source à l'exécution.

*Les solutions se trouvent dans le document **Codex_Solutions**.*

11.12.2 Réflexion finale

Vous avez maintenant une vision complète de comment fonctionne un ordinateur :

Couche	Ce que vous avez appris
Portes logiques	NAND → NOT, AND, OR, XOR, MUX
Arithmétique	Half Adder → Full Adder → ALU
Mémoire	DFP → Registre → RAM → PC
CPU	Fetch-Decode-Execute, pipeline

Couche	Ce que vous avez appris
ISA	Instructions A32, encodage binaire
Assembleur	Texte → Binaire (2 passes)
Compilateur	C32 → Assembleur (lexer, parser, codegen)
OS	Allocation mémoire, graphiques, I/O

Vous n'êtes plus un utilisateur passif de la technologie — vous comprenez comment elle fonctionne !

11.12.3 Checklist de validation finale

- ☐ Je peux expliquer comment NAND permet de construire toutes les portes
- ☐ Je comprends le complément à 2 et les drapeaux NZCV
- ☐ Je sais tracer l'exécution d'une instruction dans le CPU
- ☐ Je peux écrire et déboguer un programme en assembleur A32
- ☐ Je comprends comment le compilateur traduit C32 en assembleur
- ☐ Je sais utiliser le MMIO pour l'écran et le clavier
- ☐ Je peux expliquer la hiérarchie mémoire (registres → cache → RAM)

12 Annexe : Tous les Exercices

Cette annexe contient les énoncés de tous les exercices disponibles sur le simulateur web.

12.1 A. Exercices HDL (Portes Logiques)

Ces exercices construisent progressivement un ordinateur à partir de la porte NAND.

12.1.1 Projet 1 : Portes de Base

Exercice	Description
Inv	Inverseur (NOT) : $\text{Inv}(a) = \text{Nand}(a, a)$
And2	Porte AND : $\text{And2}(a, b) = \text{Inv}(\text{Nand}(a, b))$
Or2	Porte OR : $\text{Or2}(a, b) = \text{Nand}(\text{Inv}(a), \text{Inv}(b))$
Xor2	Porte XOR : $\text{Xor2}(a, b) = \text{Or2}(\text{And2}(a, \text{Inv}(b)), \text{And2}(\text{Inv}(a), b))$
Mux	Multiplexeur : si $\text{sel}=0$ alors $y=a$ sinon $y=b$
DMux	Demultiplexeur : distribue x vers a ou b selon sel

12.1.2 Projet 2 : Arithmétique

Exercice	Description
HalfAdder	Demi-additionneur : $\text{sum} = a \text{ XOR } b$, $\text{carry} = a \text{ AND } b$
FullAdder	Additionneur complet avec retenue entrante

Exercice	Description
Add32	Additionneur 32 bits (ripple carry)
Inc32	Incrementeur : ajoute 1 a l'entree
Alu32	ALU 32 bits avec 6 operations

12.1.3 Projet 3 : Memoire

Exercice	Description
DFF	D Flip-Flop (fourni)
Bit	Registre 1 bit avec load
Register	Registre 32 bits
RAM8	8 registres avec adressage 3 bits
RAM64	64 registres avec adressage 6 bits
PC	Compteur de programme avec inc/load/reset

12.1.4 Projet 5 : CPU

Exercice	Description
CPU	Processeur complet A32

12.1.5 Projet 6 : CPU Pipeline (Avance)

Ces exercices construisent un CPU pipeline 5 etages avec gestion des aleas.

Exercice	Description
IF_ID_Reg	Registre pipeline IF/ID avec stall et flush
HazardDetect	Detection des aleas load-use
ForwardUnit	Bypass des donnees (forwarding)
CPU_Pipeline	CPU pipeline 5 etages complet

12.1.6 Projet 7 : Cache L1

Ces exercices implementent un cache memoire direct-mapped.

Exercice	Description
CacheLine	Ligne de cache (valid, dirty, tag, data)
TagCompare	Comparateur de tags pour detecter hit/miss
WordSelect	Selecteur de mot dans une ligne de 128 bits
CacheController	Machine a etats (IDLE, FETCH, WRITEBACK)

12.2 B. Exercices Assembleur A32

Ces exercices enseignent la programmation en assembleur A32.

12.2.1 Bases

Exercice	Objectif	Resultat
Hello World	Charger 42 dans R0	R0 = 42
Addition	Calculer 15 + 27	R0 = 42
Soustraction	Calculer 100 - 58	R0 = 42
Logique	0xFF AND 0x0F	R0 = 15
Doubler	21 * 2 sans MUL	R0 = 42

12.2.2 Controle de Flux

Exercice	Objectif	Resultat
Conditions	Maximum de 25 et 17	R0 = 25
Valeur Absolue	Calculer	-42
Boucles	Somme 1 + 2 + ... + 10	R0 = 55
Multiplication	6 * 7 par additions	R0 = 42
Fibonacci	Calculer F(10)	R0 = 55

12.2.3 Memoire

Exercice	Objectif	Resultat
Tableaux	Somme de {10, 20, 30, 40, 50}	R0 = 150
Maximum Tableau	Max de {12, 45, 7, 89, 23}	R0 = 89
Memoire	Store puis Load	R0 = 30

12.2.4 Structures

Ces exercices preparent aux structs en C en montrant comment les donnees structurees sont organisees en memoire.

Exercice	Objectif	Resultat
Structure Simple	Lire champs d'un Point (x+y)	R0 = 42
Initialiser Structure	Ecrire dans les champs d'un Point	R0 = 42
Structure Rectangle	Structure avec 4 champs, calcul aire	R0 = 42
Tableau de Structures	Parcourir tableau de Points, somme des x	R0 = 33
Somme x+y Structures	Acceder aux deux champs de chaque Point	R0 = 42

12.2.5 Fonctions

Exercice	Objectif	Resultat
Fonctions	Fonction double(21)	R0 = 42
Fonction Add3	add3(10, 15, 17)	R0 = 42

12.2.6 Entrees/Sorties

Exercice	Objectif	Resultat
Ecrire Caractere	Ecrire 'A' a PUTC	R0 = 65
Hello String	Afficher "Hi"	R0 = 2
Print Loop	Afficher "ABCD" avec boucle	R0 = 4

12.2.7 Ecran (320x240, 1 bit/pixel)

Exercice	Objectif	Resultat
Pixel	Allumer pixel (0,0)	R0 = 0x80
Ligne Horizontale	8 pixels horizontaux	R0 = 0xFF
Ligne Verticale	8 pixels verticaux	R0 = 8
Rectangle	Carre 8x8	R0 = 8
Damier	Motif en damier	R0 = 8

12.2.8 Jeux Interactifs

Exercice	Objectif
Lire Caractere	Lire clavier et convertir ASCII
Lire 2 Chiffres	Former un nombre a 2 chiffres
Deviner Nombre	Jeu de devinette (secret = 7)
Degrade	Effet dithering sur ecran
Recherche Dichotomique	Trouver 42 en 7 essais

12.2.9 Cache (Patterns d'Acces Memoire)

Ces exercices illustrent l'impact des patterns d'accès memoire sur les performances cache.

Exercice	Objectif	Resultat
Acces Sequentiel	Parcours cache-friendly (adresses consecutives)	R0 = 100

Exercice	Objectif	Resultat
Acces avec Stride	Parcours avec sauts de 16 bytes (moins efficace)	R0 = 28
Reutilisation Registre	Charger une fois, reutiliser plusieurs fois	R0 = 91

12.3 C. Exercices C32

Ces exercices enseignent la programmation en C32.

12.3.1 Bases

Exercice	Objectif	Resultat
Variables	x=10, y=32, retourner x+y	42
Expressions	(5+3)*(10-4)/2	24
Modulo	(100%7) + (45%8)	7
Incrementation	5 -> +3 -> *2 -> -1	15

12.3.2 Conditions

Exercice	Objectif	Resultat
Conditions	Maximum de 25 et 17	25
Else-If	Classifier note 75	3
Operateurs Logiques	15 dans [10,20] ?	1
Maximum de 3	Max de 15, 42, 27	42

12.3.3 Boucles

Exercice	Objectif	Resultat
Boucle For	Somme 1 a 10	55
Boucle While	Compter chiffres de 12345	5
Boucles Imbriquees	Double boucle i*j	60
Multiplication	<i>78 sans operateur</i>	56

12.3.4 Fonctions

Exercice	Objectif	Resultat
Fonctions	square(7)	49
Parametres Multiples	add3(10, 20, 12)	42
Valeur Absolue	abs(-15) + abs(10)	25
Min et Max	max(10,25) - min(10,25)	15

12.3.5 Tableaux

Exercice	Objectif	Resultat
Tableaux	Somme de {3,7,2,9,5}	26
Maximum Tableau	Max de 6 elements	56
Compter Elements	Nombres pairs	4

12.3.6 Pointeurs

Exercice	Objectif	Resultat
Pointeurs	Modifier via *p	42
Swap	Echanger x et y	20
Pointeurs et Tableaux	Somme avec *(p+i)	50

12.3.7 Operations Binaires

Exercice	Objectif	Resultat
Operations Binaires	(10&12)	(10^12)
Puissance de 2	is_pow2(16)+is_pow2(15)+is_pow2(32)	2

12.3.8 Recursion

Exercice	Objectif	Resultat
Factorielle	fact(5)	120
Fibonacci	fib(10)	55
Somme Recursive	sum(10)	55

12.3.9 Algorithmes Avances

Exercice	Objectif	Resultat
PGCD (Euclide)	gcd(48, 18)	6
Puissance	power(2, 10)	1024
Test Primalite	Premiers <= 20	8
Tri a Bulles	Trier et retourner min	12
Recherche Binaire	Trouver 23 dans tableau	5
Inverser Tableau	Inverser {1,2,3,4,5}	35
Somme Chiffres	digit_sum(12345)	15
Palindrome	12321 + 1221 + 123	2

12.3.10 Structures

Les structures permettent de regrouper plusieurs variables liees.

Exercice	Objectif	Resultat
Definition Struct	struct Point, p.x + p.y	42
Pointeur Struct	Utiliser l'operateur ->	42
Struct et Fonctions	distance_sq(Point*)	25

Exercice	Objectif	Resultat
Structs Imbriquees	Rectangle avec Point	42
Tableau de Structs	Point[3], somme des x	33
Sizeof Struct	Taille des structures	16

12.3.11 Cache (Patterns d'Acces Memoire)

Ces exercices illustrent l'impact de la localite sur les performances cache.

Exercice	Objectif	Resultat
Parcours en Ligne	Parcours row-major (cache-friendly)	120
Parcours en Colonne	Parcours column-major (moins efficace)	120
Traitement par Blocs	Technique de blocking	120
Localite Temporelle	Reutiliser les donnees en cache	30

12.3.12 Entrees/Sorties

Exercice	Objectif	Resultat
Ecrire Caractere	putchar(65)	65
Afficher Chaine	print("HI")	2
Afficher Nombre	print_int(42)	42
Dessiner Pixel	Pixel (0,0)	128
Ligne Horizontale	16 pixels	16
Dessiner Rectangle	Carre 8x8	64

12.3.13 Projets Avances

Exercice	Objectif	Resultat
Crible Eratosthene	Premiers ≤ 50	15
Suite Collatz	Longueur pour $n=27$	112
Projet Final	Diviseurs de 28	28

12.4 D. Construction du Compilateur

Ces exercices construisent progressivement un compilateur C -> A32.

12.4.1 Phase 1 : Lexer

Exercice	Description
1.1 Reconnaître Chiffre	<code>is_digit(c)</code> : retourne 1 si '0'-'9'
1.2 Lire Nombre	<code>parse_number(s, pos)</code> : extrait un entier
1.3 Identifier Tokens	<code>next_token()</code> : retourne type du token

12.4.2 Phase 2 : Parser

Exercice	Description
2.1 Evaluer $a + b$	Parser et calculer une operation
2.2 Evaluer $a + b + c$	Chaine d'operations gauche a droite
2.3 Precedence	Descente recursive : * avant +
2.4 Parentheses	Supporter $(2 + 3) * 4$

12.4.3 Phase 3 : Emission ASM

Exercice	Description
3.1 Generer MOV	Produire "MOV R0, #42"
3.2 Operation Binaire	Mapper + -> ADD, * -> MUL
3.3 Comparaison	Generer CMP et conditions

12.4.4 Phase 4 : CodeGen Expressions

Exercice	Description
4.1 Constante -> A32	Code pour charger une constante
4.2 Addition -> A32	a + b -> MOV/MOV/ADD
4.3 Expression -> A32	Expression complete avec precedence

12.4.5 Phase 5 : Structures de Controle

Exercice	Description
5.1 If/Else -> A32	Sauts conditionnels et labels
5.2 While -> A32	Boucles avec labels

12.4.6 Phase 6 : Fonctions

Exercice	Description
6.1 Prologue/Epilogue	Sauvegarder LR, reserver pile
6.2 Appel de Fonction	Arguments et BL

12.4.7 Phase 7 : Projet Final

Exercice	Description
7.1 Mini-Compilateur	Compiler expression en A32 executable

12.5 E. Systeme d'Exploitation

Ces exercices introduisent les concepts OS.

12.5.1 Initialisation

Exercice	Description	Resultat
Bootstrap	Initialiser SP, effacer BSS, appeler main	42

12.5.2 Gestion Memoire

Exercice	Description	Resultat
Bump Allocator	Allocation simple par incrementation	100
Free List	Allocateur avec liberation	1

12.5.3 Drivers

Exercice	Description	Resultat
Driver Ecran	Fonctions set_pixel, clear_screen	4
Police Bitmap	Dessiner caracteres 8x8	51

12.5.4 Console et Clavier

Exercice	Description	Resultat
Console	Console texte avec curseur (40x30)	1
Driver Clavier	Lire les touches, buffer clavier	3

12.5.5 Shell et Applications

Exercice	Description	Resultat
Shell	Interpreteur de commandes basique	1
Calculatrice	Evaluer expressions arithmetiques	42
Variables Shell	Variables dans le shell (\$x, \$y)	15
Compte a Rebours	Timer avec affichage	0

12.5.6 Multitache

Exercice	Description	Resultat
Interruptions	Gestion des interruptions timer	10
Coroutines	Changement de contexte manuel	2
Scheduler	Ordonnanceur round-robin	6

12.5.7 Projets OS

Exercice	Description
Projet 1: Mini-OS Shell	Shell complet avec commandes
Projet 2: Task Manager	Gestionnaire de taches multiples

12.6 Conseils pour les Exercices

1. **Commencez simple** : Les premiers exercices de chaque section sont accessibles
 2. **Lisez l'énoncé** : Chaque exercice contient des indices
 3. **Testez souvent** : Le simulateur exécute votre code instantanément
 4. **Consultez les solutions** : Après avoir essayé, comparez avec la solution
 5. **Progresssez** : Chaque exercice prépare le suivant
-

Bon courage !

13 L'Art du Débogage

“Le débogage, c’est comme être détective dans un film policier où vous êtes aussi le meurtrier.” — Filipe Fortes

Le débogage est une compétence fondamentale en programmation. Ce chapitre vous apprend à trouver et corriger les erreurs de manière méthodique.

13.1 Pourquoi ce Chapitre ?

Vous avez appris à construire des circuits, écrire de l’assembleur, et utiliser un compilateur. Mais que faire quand **ça ne marche pas** ?

Ce chapitre couvre : 1. Comment lire les messages d’erreur 2. Comment utiliser le simulateur pas-à-pas 3. Les erreurs les plus courantes et leurs solutions 4. Une méthodologie systématique de débogage

13.2 1. Comprendre les Messages d’Erreur

13.2.1 Codes d’Erreur du Projet Codex

Les outils du projet utilisent des codes d’erreur standardisés :

Préfixe	Source	Description
E1xxx	Assembleur	Erreurs d’assemblage
E2xxx	Compilateur C32	Erreurs de compilation
E3xxx	Linker	Erreurs de liaison

13.2.2 Erreurs d'Assemblage (E1xxx)

Code	Signification	Solution
E1001	Mnémonique inconnu	Vérifiez l'orthographe de l'instruction
E1002	Opérande invalide	Vérifiez le format des registres/immédiats
E1003	Label non défini	Ajoutez le label ou corrigez son nom
E1004	Immédiat hors plage	Utilisez LDR R0, =value pour les grandes valeurs
E1005	Registre invalide	Utilisez R0-R15 uniquement
E1008	Literal pool overflow	Placez .ltorg plus tôt dans le code

Exemple : Erreur E1004

```
; ERREUR : L'immédiat 0x12345678 ne tient pas sur 12 bits
MOV R0, #0x12345678

; SOLUTION : Utiliser le literal pool
LDR R0, =0x12345678
```

13.2.3 Erreurs de Compilation (E2xxx)

Code	Signification	Solution
E2001	Erreur de syntaxe	Vérifiez les parenthèses, points-virgules
E2002	Type incompatible	Vérifiez les types des opérandes
E2003	Variable non déclarée	Déclarez la variable avant utilisation
E2004	Fonction non définie	Définissez la fonction ou incluez le header

Code	Signification	Solution
E2008	Return manquant	Ajoutez return à la fin de la fonction

Exemple : Erreur E2003

```
// ERREUR : 'x' non déclaré
int main() {
    x = 5; // E2003: Variable 'x' non déclarée
    return 0;
}

// SOLUTION : Déclarer la variable
int main() {
    int x;
    x = 5;
    return 0;
}
```

13.2.4 Erreurs de Liaison (E3xxx)

Code	Signification	Solution
E3001	Symbole non résolu	Définissez le symbole ou liez la bibliothèque
E3002	Point d'entrée manquant	Ajoutez _start: ou main()
E3004	Débordement mémoire	Réduisez la taille du programme

13.3 2. Utiliser le Simulateur Pas-à-Pas

13.3.1 Le Visualiseur CPU

Le visualiseur web (web/visualizer.html) est votre meilleur ami pour déboguer.

Fonctionnalités clés : - **Step (N ou F10)** : Exécute une instruction - **Registres** : Voir R0-R15, SP, LR, PC et les flags - **Mémoire** : Inspecter la RAM - **Code** : Ligne courante surlignée

13.3.2 Méthodologie de Débogage Pas-à-Pas

1. **Chargez votre programme** dans le visualiseur
2. **Identifiez le symptôme** : Qu'est-ce qui ne va pas ?
3. **Formulez une hypothèse** : Où le bug pourrait-il être ?
4. **Placez-vous avant** la zone suspecte
5. **Exécutez pas-à-pas** en vérifiant les registres
6. **Trouvez la divergence** : Où le comportement diffère-t-il de l'attendu ?

13.3.3 Exemple Pratique

Programme bugué :

```
; Censé calculer 5 + 3 = 8
MOV R0, #5
MOV R1, #3
SUB R2, R0, R1 ; BUG : devrait être ADD !
SVC #0
```

Débogage : 1. Après MOV R0, #5 : R0 = 5 □ 2. Après MOV R1, #3 : R1 = 3 □ 3. Après SUB R2, R0, R1 : R2 = 2 □ (attendu : 8)

→ **Bug trouvé** : SUB au lieu de ADD

13.4 3. Erreurs Courantes et Solutions

13.4.1 3.1 Assembleur

13.4.1.1 Oubli de sauvegarder LR

Symptôme : Le programme ne retourne pas correctement d'une fonction.

```
; BUGUÉ
my_func:
    BL other_func ; LR est écrasé !
    BX LR        ; Retourne... où ?

; CORRECT
my_func:
    PUSH {LR}    ; Sauvegarder LR
    BL other_func
    POP {LR}     ; Restaurer LR
    BX LR
```

13.4.1.2 Off-by-one dans les boucles

Symptôme : La boucle s'exécute une fois de trop ou de moins.

```
; BUGUÉ : S'arrête à 9 au lieu de 10
    MOV R0, #0
loop:
    CMP R0, #10
    BEQ done      ; BEQ : si R0 == 10, sauter
    ; ... corps de la boucle ...
    ADD R0, R0, #1
    B loop
done:

; Vérifiez toujours : combien d'itérations attendues ?
; R0 va de 0 à 9 = 10 itérations ✓
```

13.4.1.3 Mauvais alignement mémoire

Symptôme : Erreur MISALIGNED ou données corrompues.

```
; BUGUÉ : Adresse non alignée sur 4 octets
    LDR R0, [R1, #3]    ; 3 n'est pas multiple de 4 !

; CORRECT
    LDR R0, [R1, #4]    ; OK : 4 est multiple de 4
```

13.4.2 3.2 Compilateur C32

13.4.2.1 Oubli du return

Symptôme : Valeur de retour incorrecte ou aléatoire.

```
// BUGUÉ
int add(int a, int b) {
    int c = a + b;
    // Oops, pas de return !
}

// CORRECT
int add(int a, int b) {
    int c = a + b;
    return c;
}
```

13.4.2.2 Confusion pointeur/valeur

Symptôme : Crash ou données incorrectes.

```
// BUGUÉ
void increment(int x) {
    x = x + 1; // Modifie la COPIE, pas l'original !
}

// CORRECT
void increment(int *x) {
    *x = *x + 1; // Modifie la valeur POINTÉE
}

// Appel
int n = 5;
increment(&n); // Passer l'ADRESSE
```

13.4.2.3 Division par zéro

Symptôme : Trap DIV_ZERO ou résultat infini.

```
// BUGUÉ
int divide(int a, int b) {
    return a / b; // Et si b == 0 ?
}

// CORRECT
int divide(int a, int b) {
    if (b == 0) {
        return 0; // Ou gérer l'erreur autrement
    }
    return a / b;
}
```

13.4.3 3.3 HDL

13.4.3.1 Signal non connecté

Symptôme : Sortie toujours à 0 ou X (indéfini).

```
-- BUGUÉ : 'result' n'est jamais assigné
entity Adder is
    port(a, b : in bit; result : out bit);
end entity;
```

```
architecture rtl of Adder is
begin
    -- Oops, rien ici !
end architecture;

-- CORRECT
architecture rtl of Adder is
begin
    result <= a xor b; -- Connexion explicite
end architecture;
```

13.4.3.2 Boucle combinatoire

Symptôme : Le simulateur ne converge pas ou donne des résultats instables.

```
-- BUGUÉ : 'x' dépend de lui-même instantanément
signal x : bit;
x <= not x; -- Boucle infinie !

-- CORRECT : Utiliser une DFF pour la rétroaction
-- x(t+1) <= not x(t) -- Via une DFF
```

13.5 4. Méthodologie Systématique

13.5.1 La Méthode Scientifique du Débogage

1. **Observer** : Quel est le symptôme exact ?
 - Message d'erreur ?
 - Mauvais résultat ?
 - Crash ?
2. **Hypothétiser** : Quelle pourrait être la cause ?
 - Erreur de logique ?
 - Mauvaise valeur ?
 - Condition incorrecte ?
3. **Tester** : Vérifier l'hypothèse
 - Ajouter des prints/traces
 - Exécuter pas-à-pas

- Simplifier le code

4. **Conclure** : L'hypothèse était-elle correcte ?

- Si oui → corriger
- Si non → nouvelle hypothèse

13.5.2 La Technique de la Bissection

Quand le bug est difficile à trouver :

1. **Divisez** le code en deux moitiés
2. **Testez** chaque moitié indépendamment
3. **Identifiez** quelle moitié contient le bug
4. **Répétez** jusqu'à isoler le problème

Code complet (bug quelque part)

```
|— Première moitié
|   |— ✓ OK
|— Deuxième moitié
|   |— x Bug ici !
|       |— Premier quart → ✓ OK
|       |— Deuxième quart → x Bug trouvé !
```

13.5.3 Ajouter des Traces

En C32 :

```
void debug_print(int value) {
    // Afficher la valeur pour debug
    putc('[');
    // ... afficher value ...
    putc(']');
    putc('\n');
}

int main() {
    int x = compute_something();
    debug_print(x); // Voir la valeur
    // ...
}
```

En assembleur :

```
; Afficher R0 pour debug (caractère ASCII)
ADD R0, R0, #'0' ; Convertir en ASCII
LDR R1, =0xFFFF0000
STR R0, [R1] ; Afficher
```

13.6 5. Erreurs de Traps et leur Diagnostic

13.6.1 Types de Traps

Trap	Cause	Solution
DIV_ZERO	Division par 0	Vérifier le diviseur avant
MEM_FAULT	Accès mémoire invalide	Vérifier les pointeurs
MISALIGNED	Adresse non alignée	Aligner sur 4 octets
ILLEGAL	Instruction invalide	Vérifier l'encodage

13.6.2 Débuguer un MEM_FAULT

1. **Notez l'adresse** du PC quand le trap se produit
2. **Trouvez l'instruction** à cette adresse
3. **Examinez les registres** utilisés pour l'adressage
4. **Questions à se poser** :
 - Le pointeur est-il initialisé ?
 - Est-il dans la plage valide (0 - RAM_SIZE) ?
 - A-t-il été corrompu par une autre partie du code ?

13.7 6. Checklist de Débogage

Avant de chercher un bug complexe, vérifiez ces points simples :

13.7.1 Assembleur

- ☐ Toutes les instructions sont correctement orthographiées ?

- ☐ Les registres sont dans la plage R0-R15 ?
- ☐ Les immédiats sont dans la plage autorisée ?
- ☐ LR est sauvegardé avant les appels de fonction ?
- ☐ La pile est équilibrée (PUSH = POP) ?
- ☐ Les accès mémoire sont alignés sur 4 octets ?

13.7.2 C32

- ☐ Toutes les variables sont déclarées ?
- ☐ Toutes les fonctions ont un return ?
- ☐ Les pointeurs sont initialisés avant déréférencement ?
- ☐ Pas de division par zéro possible ?
- ☐ Les indices de tableau sont dans les bornes ?

13.7.3 HDL

- ☐ Toutes les sorties sont assignées ?
 - ☐ Pas de boucles combinatoires ?
 - ☐ Les signaux sont de la bonne largeur (bit vs bits) ?
 - ☐ Les port maps connectent les bons signaux ?
-

13.8 7. Exercices de Débogage

13.8.1 Exercice 1 : Trouvez le bug

```
; Programme censé afficher "Hello"
    LDR R0, =message
    LDR R1, =0xFFFF0000
loop:
    LDRB R2, [R0]
    CMP R2, #0
    BEQ done
    STR R2, [R1]
    ADD R0, R0, #1
    B done                ; <-- Bug ici !
done:
    SVC #0
message: .asciz "Hello"
```

Voir la solution

Le bug est B done qui devrait être B loop. Le programme n'affiche que le premier caractère car il saute directement à done au lieu de boucler.

```
B loop ; CORRECT : retourner au début de la boucle
```

13.8.2 Exercice 2 : Trouvez le bug

```
int factorial(int n) {  
    if (n == 0) {  
        return 1;  
    }  
    return n * factorial(n); // <-- Bug ici !  
}
```

Voir la solution

Le bug est factorial(n) qui devrait être factorial(n - 1). Sans décrémenter n, la récursion est infinie → stack overflow.

```
return n * factorial(n - 1); // CORRECT
```

13.8.3 Exercice 3 : Trouvez le bug

```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}  
  
int main() {  
    int x = 5;  
    int y = 10;  
    swap(x, y);  
    // x est toujours 5, y toujours 10 !  
    return 0;  
}
```

Voir la solution

Le bug est le passage par valeur au lieu de passage par pointeur. Les modifications de a et b n'affectent pas x et y.


```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 5;
    int y = 10;
    swap(&x, &y); // Passer les adresses
    return 0;
}
```

13.9 Ce qu'il faut retenir

1. **Lisez les messages d'erreur** : Ils contiennent souvent la solution
2. **Utilisez le pas-à-pas** : Le visualiseur est votre meilleur outil
3. **Formulez des hypothèses** : Approche scientifique
4. **Connaissez les erreurs courantes** : Elles représentent 90% des bugs
5. **Simplifiez** : Réduisez le problème au minimum reproductible

Le débogage est une compétence qui s'améliore avec la pratique. Chaque bug que vous trouvez vous rend meilleur !

14 Le Cache : Pourquoi Votre Ordinateur Semble Rapide

Imaginez que vous travaillez dans un bureau et que vous avez besoin de consulter des documents. Vous avez deux options : - **Votre bureau** : les documents sont juste devant vous, vous pouvez les lire instantanément - **Les archives au sous-sol** : il faut descendre 5 étages, chercher le bon dossier, remonter... cela prend plusieurs minutes

Le **cache** est exactement comme votre bureau : une petite zone de stockage très rapide où l'on garde les documents (données) les plus utilisés, pour éviter d'aller constamment aux archives (la mémoire RAM).

14.1 Le Problème : La RAM est Lente !

14.1.1 La Hiérarchie Mémoire

Voici comment est organisée la mémoire dans un ordinateur, du plus rapide au plus lent :

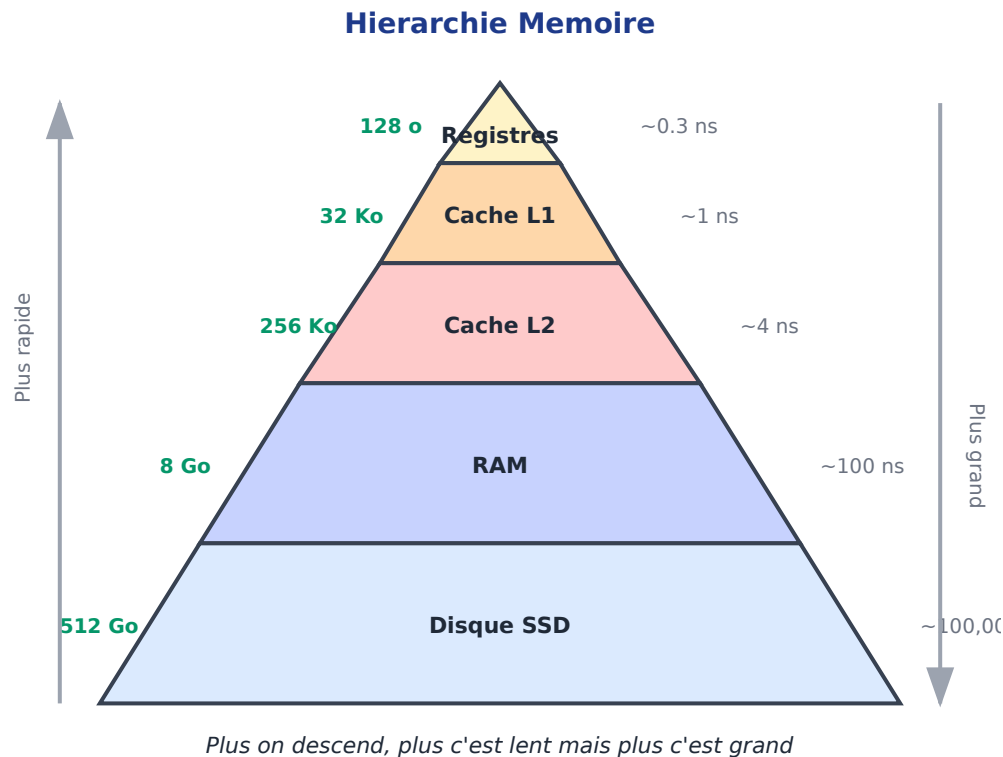


Figure 14.1: Hiérarchie mémoire

14.1.2 L'écart de vitesse visualisé

Si on convertissait ces temps en échelle humaine, les registres seraient comme cligner des yeux (0.3 seconde), le cache L1 comme une seconde, le cache L2 comme 4 secondes, la RAM comme 1 minute 40 secondes, et accéder au SSD prendrait... **27 heures !**

Sans cache, le processeur passerait **99% de son temps à attendre !**

14.2 La Solution : Le Principe de Localité

Les programmes ne lisent pas la mémoire au hasard. Ils suivent des **patterns** prévisibles.

14.2.1 Localité Temporelle

“Si j'utilise une donnée maintenant, je vais probablement la réutiliser bientôt”

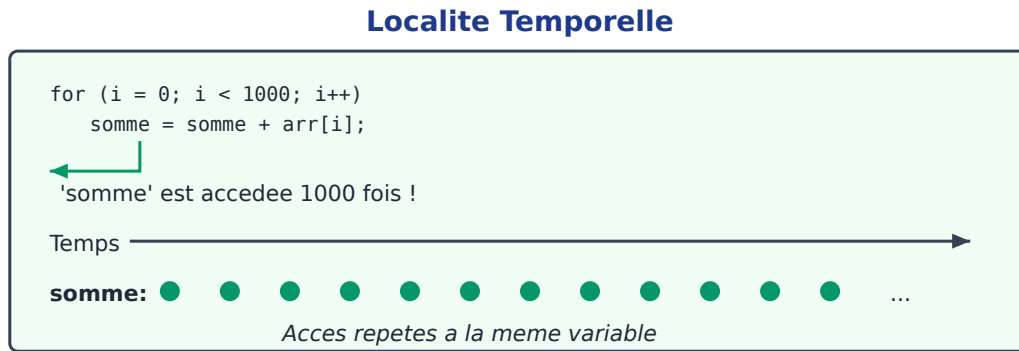


Figure 14.2: Localité temporelle

14.2.2 Localité Spatiale

“Si j’utilise une donnée, je vais probablement utiliser ses voisines”

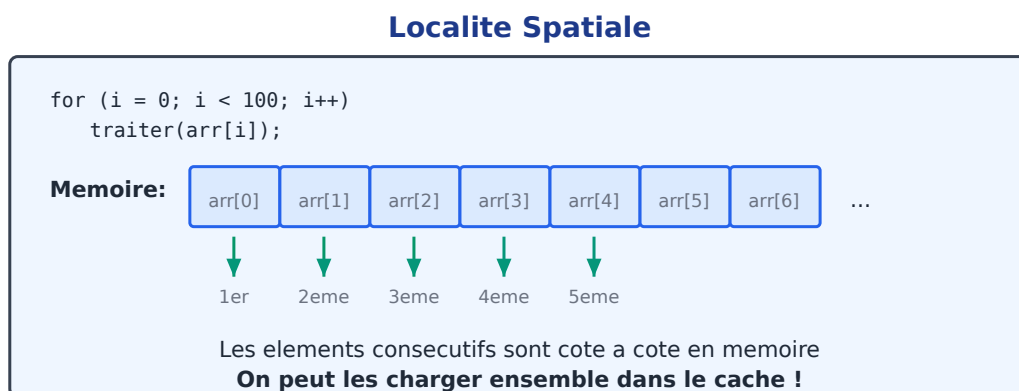


Figure 14.3: Localité spatiale

14.3 Comment Fonctionne le Cache ?

14.3.1 Les Lignes de Cache

Le cache stocke des **blocs** de données appelés **lignes de cache**, pas des octets individuels.

Quand vous demandez l’adresse 100, le cache charge **16 octets ensemble** (adresses 100 à 115). Ensuite, si vous demandez 101, 102, 103... c’est gratuit car déjà en cache !

14.3.2 Structure d'une Ligne de Cache

Structure d'une Ligne de Cache

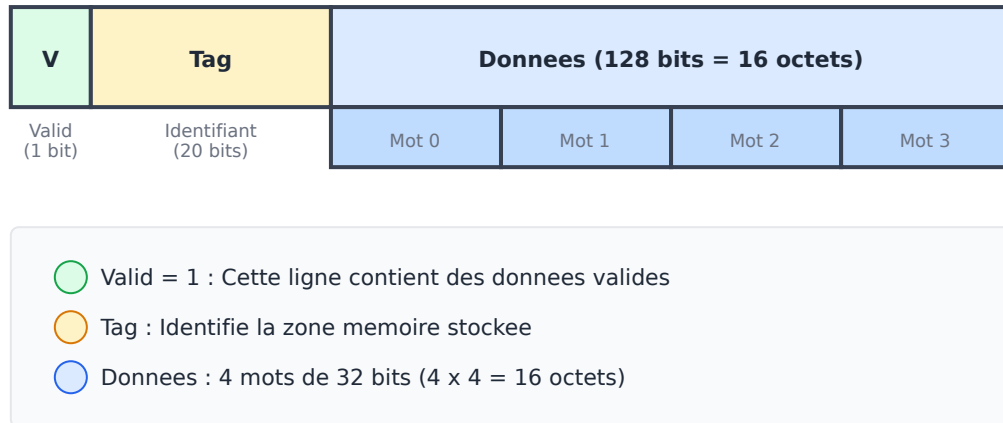


Figure 14.4: Structure d'une ligne de cache

14.3.3 Découpage d'une Adresse

Quand le CPU demande l'adresse 0x00001234, comment le cache la trouve-t-il ?

Decomposition d'une Adresse 32 bits

Adresse: 0x00001234

0000 0000 0000 0000 0001 0010 0011 0100

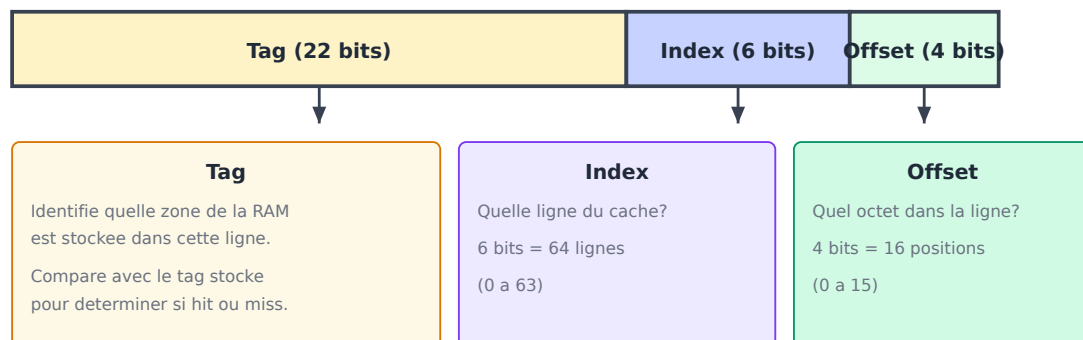


Figure 14.5: Décomposition d'une adresse

14.3.4 Structure Complète du Cache

Cache Direct-Mapped (64 lignes)

Index	Valid	Tag	Donnees (128 bits)	
0	1	0x00012	[Mot0] [Mot1] [Mot2] [Mot3]	
1	0	-----	-----	
2	1	0x00045	[Mot0] [Mot1] [Mot2] [Mot3]	
3	1	0x00123	[Mot0] [Mot1] [Mot2] [Mot3]	Ligne ac
4	0	-----	-----	
5	1	0x00067	[Mot0] [Mot1] [Mot2] [Mot3]	
...				
63	1	0x00ABC	[Mot0] [Mot1] [Mot2] [Mot3]	

Chaque ligne = 1 bit Valid + 20 bits Tag + 128 bits Data

Valid=1 Valid=0 Tag Donnees

Figure 14.6: Cache direct-mapped

14.4 Hit ou Miss : Que se passe-t-il ?

14.4.1 Scénario 1 : Cache HIT (Succès)

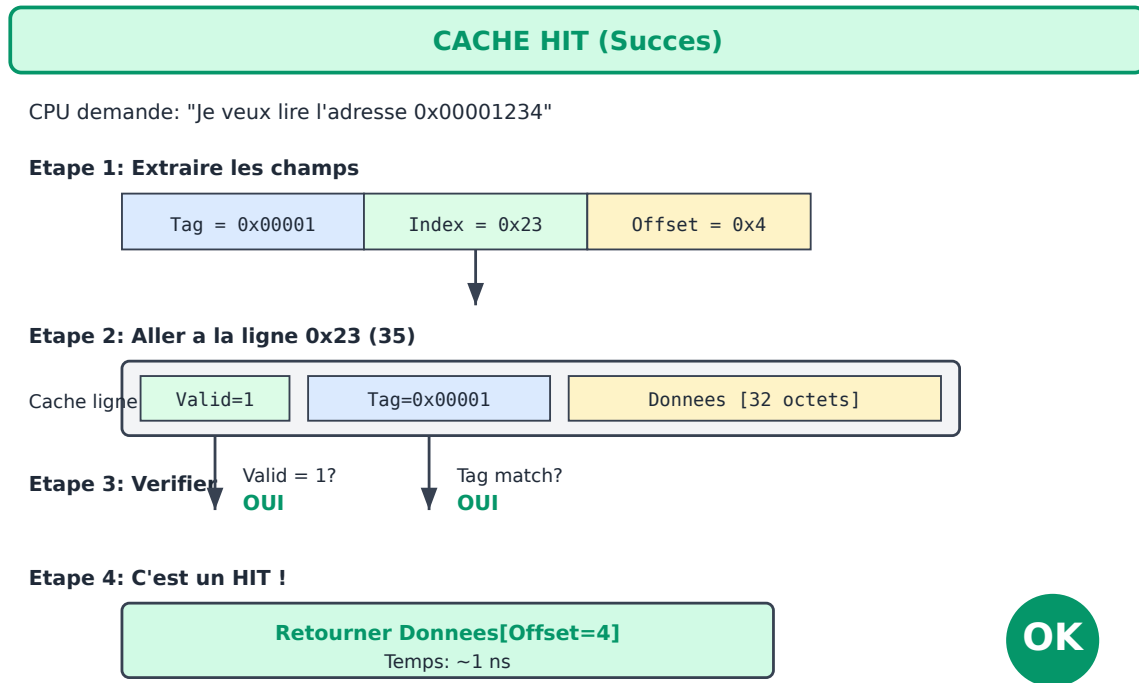


Figure 14.7: Scénario Cache HIT

14.4.2 Scénario 2 : Cache MISS (Échec)

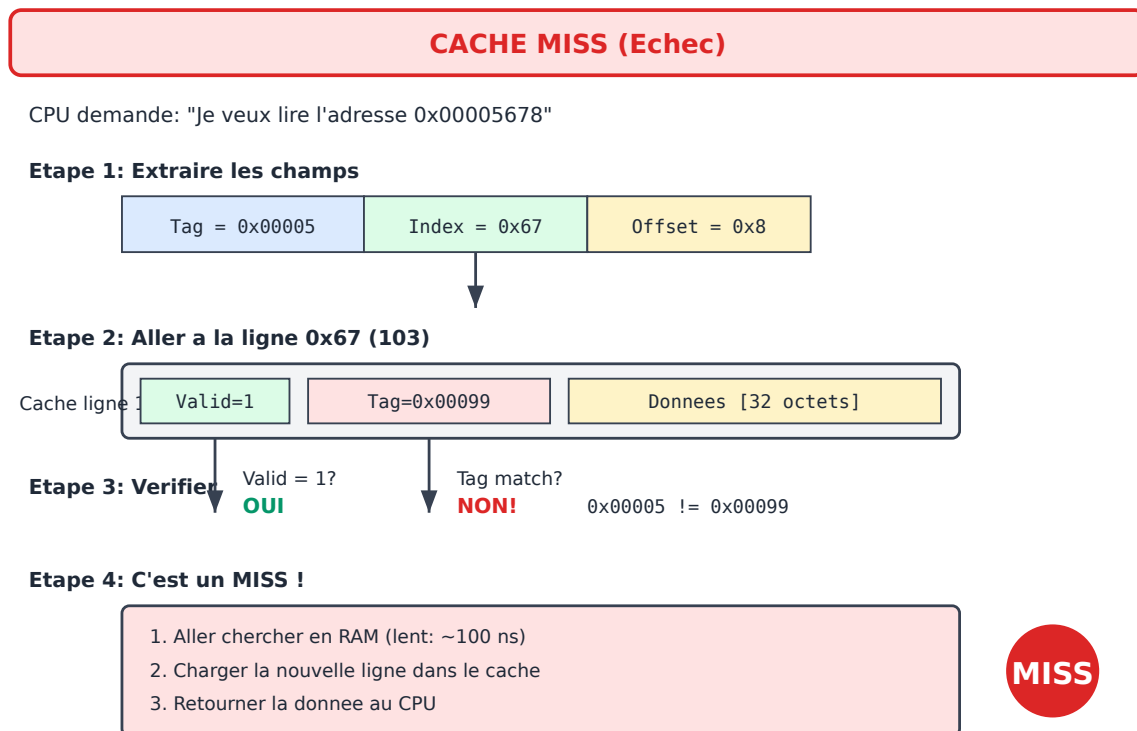


Figure 14.8: Scénario Cache MISS

14.4.3 Diagramme de Flux Complet

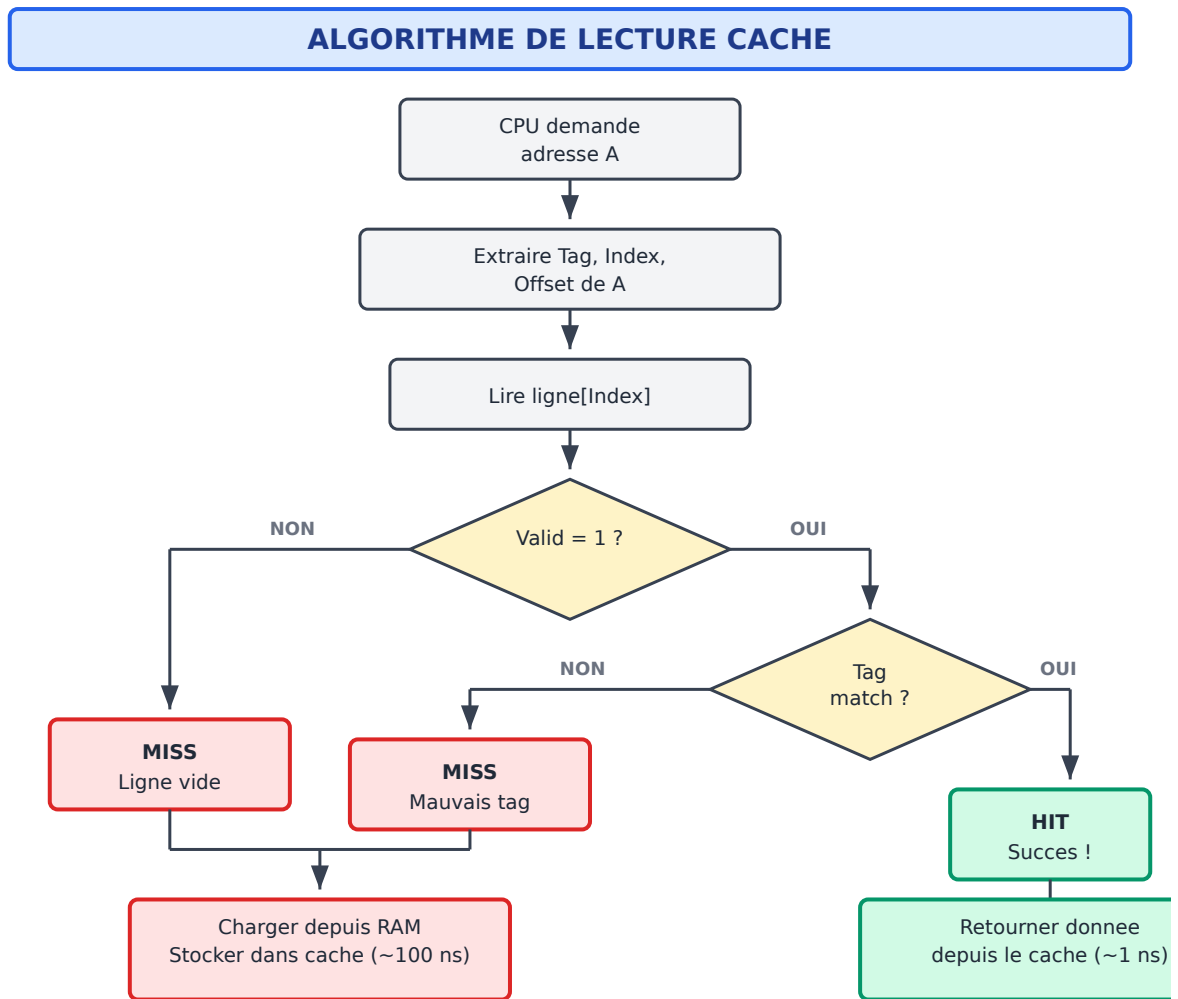


Figure 14.9: Algorithme de lecture cache

14.5 Politiques d'Écriture

14.5.1 Write-Through (Écriture Directe)

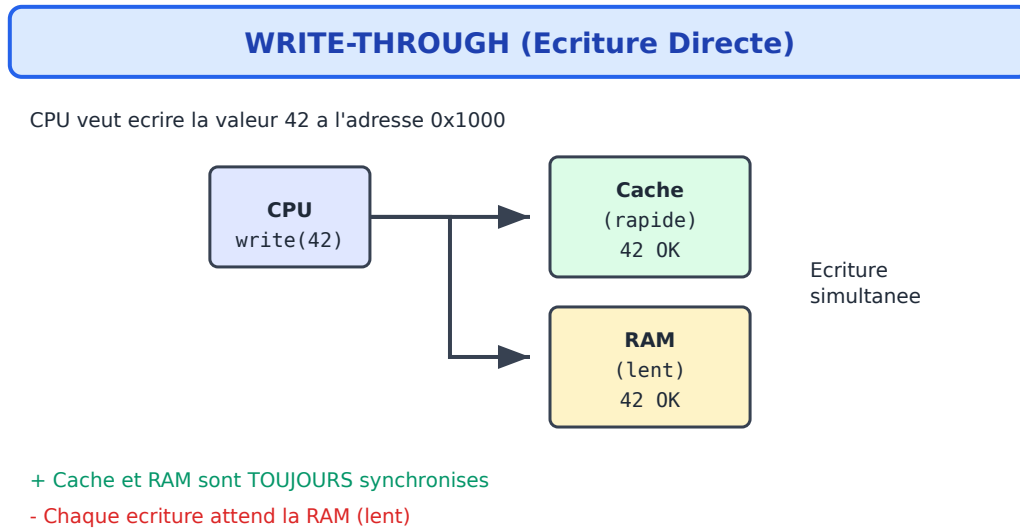


Figure 14.10: Write-Through

14.5.2 Write-Back (Écriture Différée)

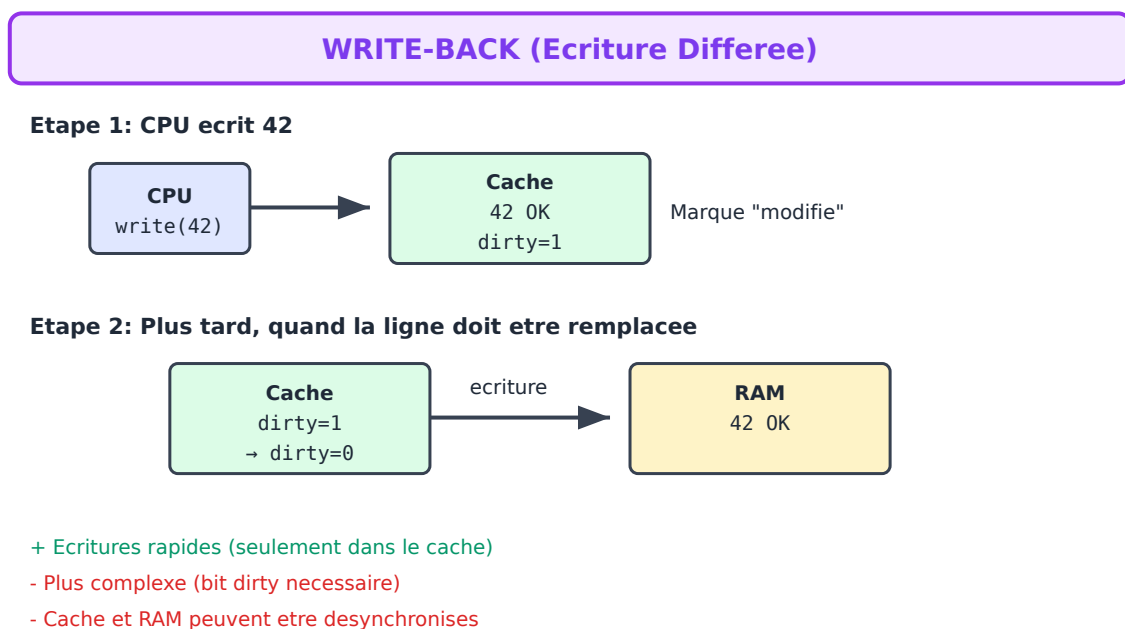


Figure 14.11: Write-Back

14.6 Impact sur vos Programmes

14.6.1 Parcours de Tableaux 2D : L'Ordre Compte !

Une matrice 4x4 est stockée **ligne par ligne** en mémoire (row-major) :

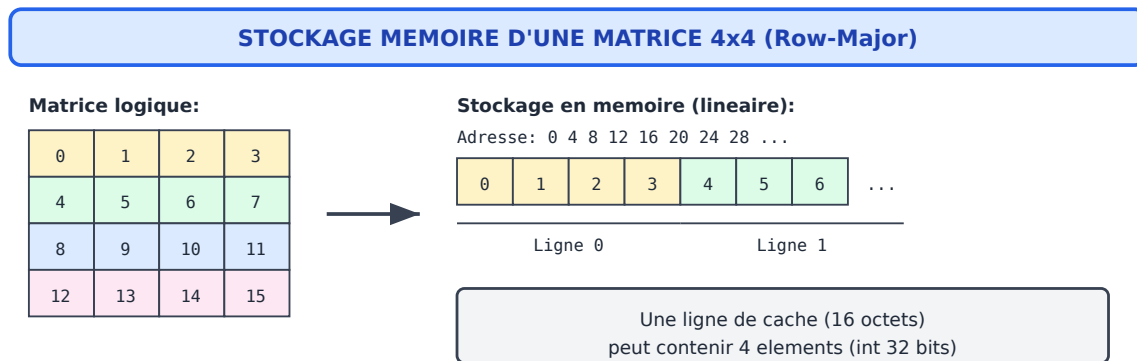


Figure 14.12: Stockage mémoire d'une matrice

14.6.2 Parcours Row-Major (En Ligne) - EFFICACE

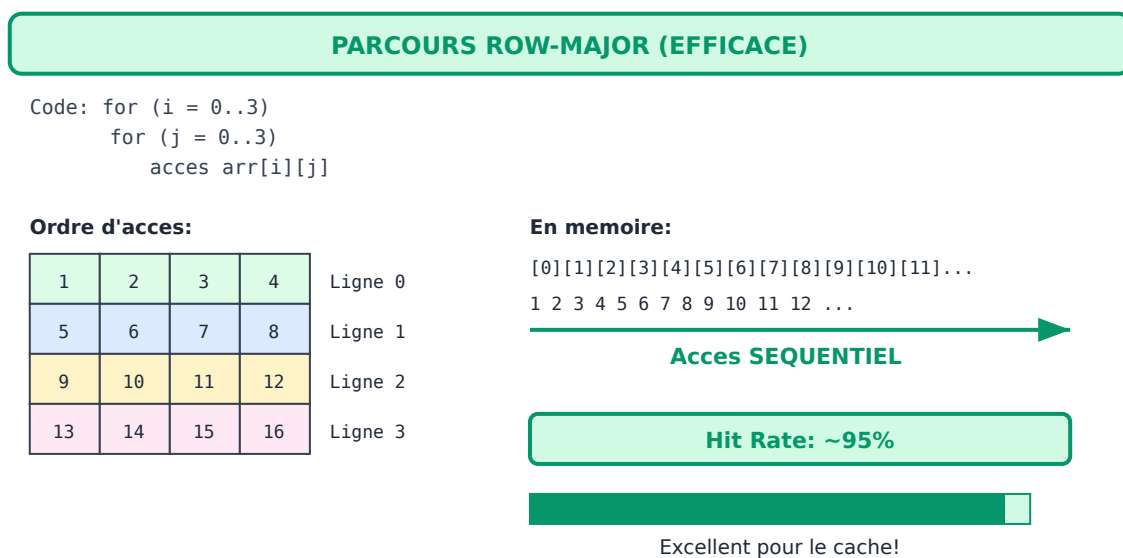


Figure 14.13: Parcours Row-Major

14.6.3 Parcours Column-Major (En Colonne) - INEFFICACE

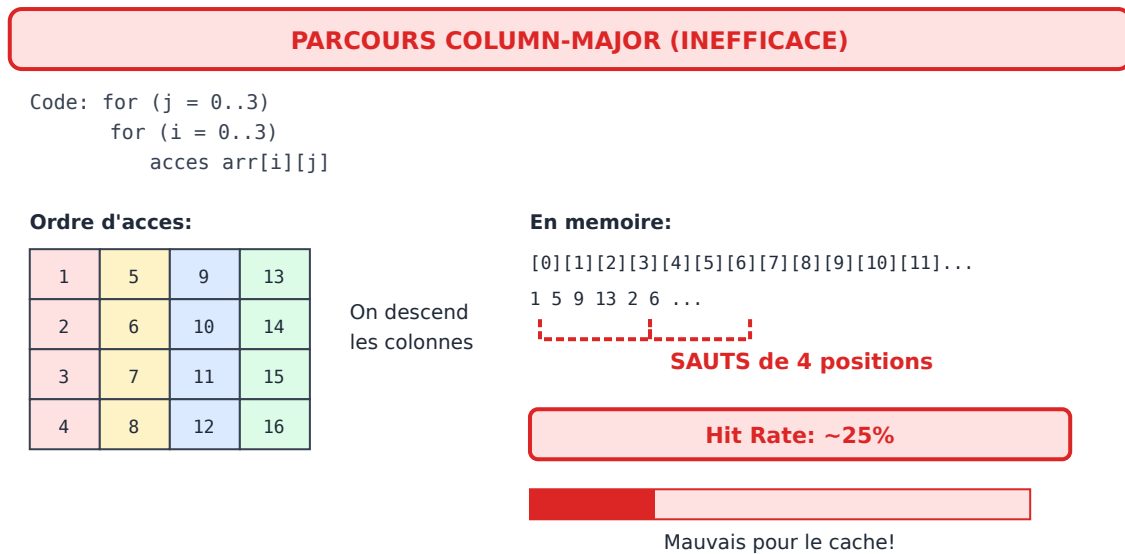


Figure 14.14: Parcours Column-Major

14.6.4 Comparaison Visuelle

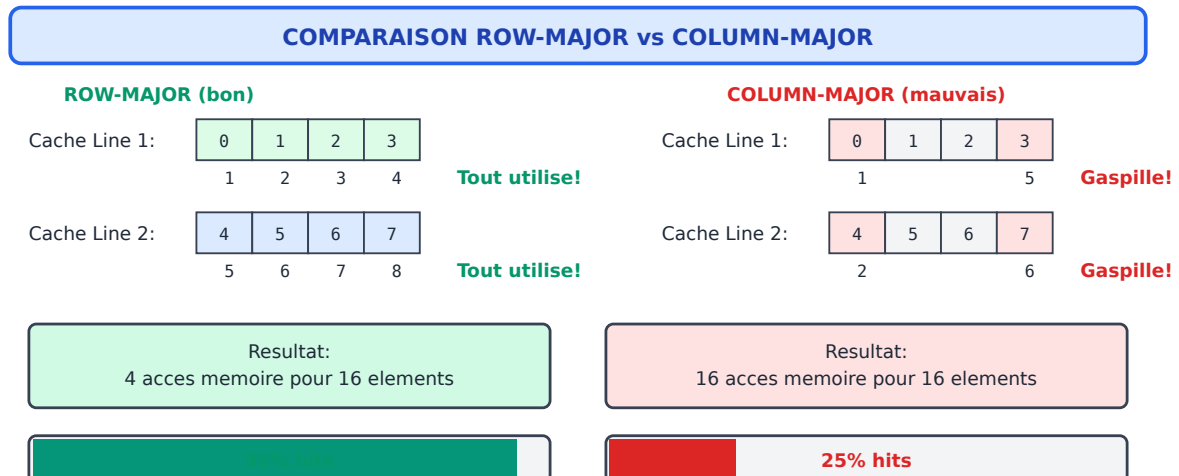


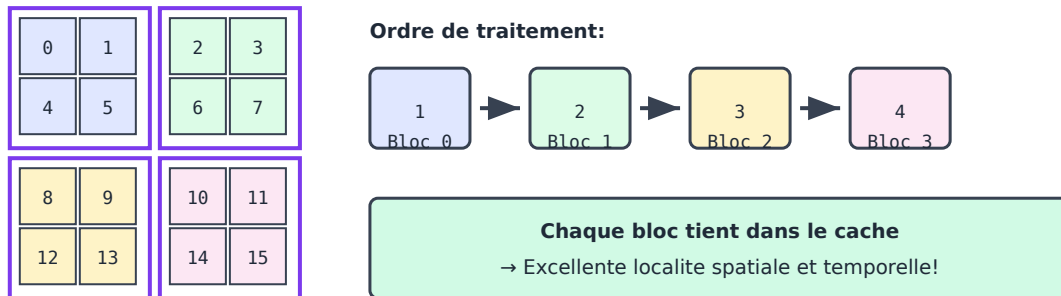
Figure 14.15: Comparaison Row-Major vs Column-Major

14.6.5 Technique du Blocking

TECHNIQUE DE BLOCKING

Au lieu de parcourir toute la matrice, on traite par BLOCS qui tiennent dans le cache.

Matrice 4x4 divisee en blocs 2x2:



Note: Le blocking est utilise dans les bibliotheques d'algebre lineaire (BLAS, LAPACK) pour optimiser les multiplications de matrices.

Figure 14.16: Technique de Blocking

14.7 Implémentation HDL du Cache

14.7.1 Architecture Globale

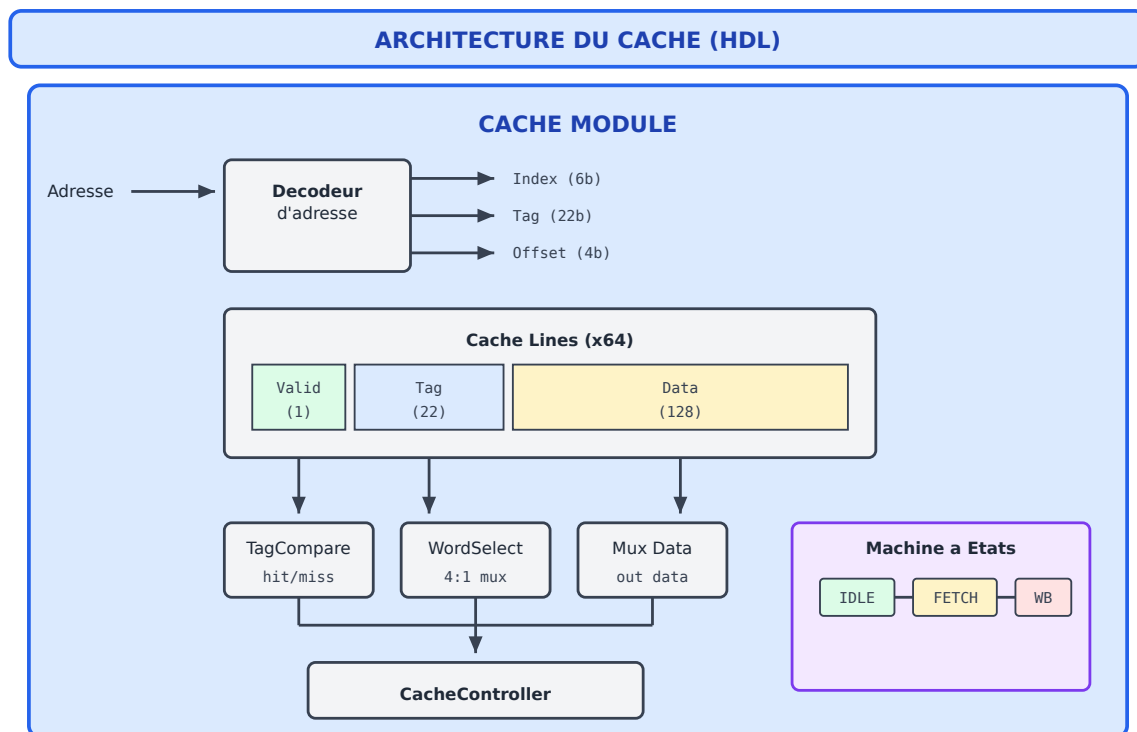


Figure 14.17: Architecture du cache HDL

14.7.2 Machine à États du Contrôleur

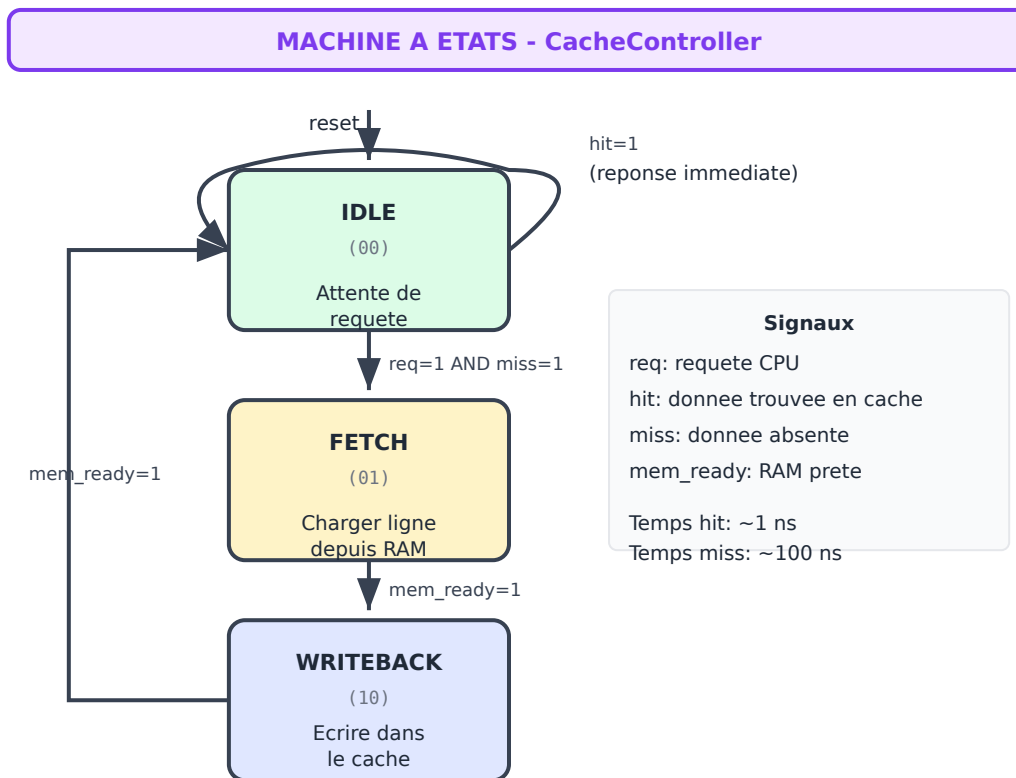


Figure 14.18: Machine à états du CacheController

14.7.3 WordSelect : Sélection du Mot

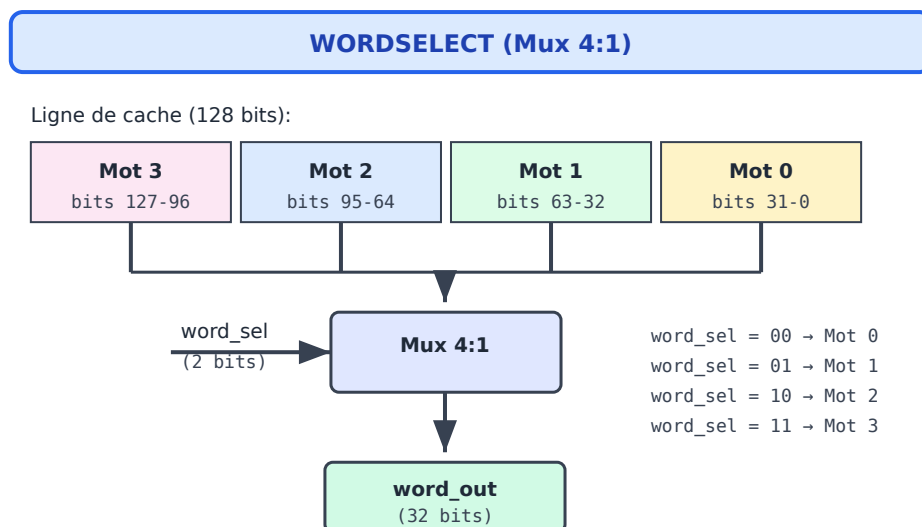


Figure 14.19: WordSelect

14.8 Statistiques et Performance

14.8.1 Calcul du Hit Rate

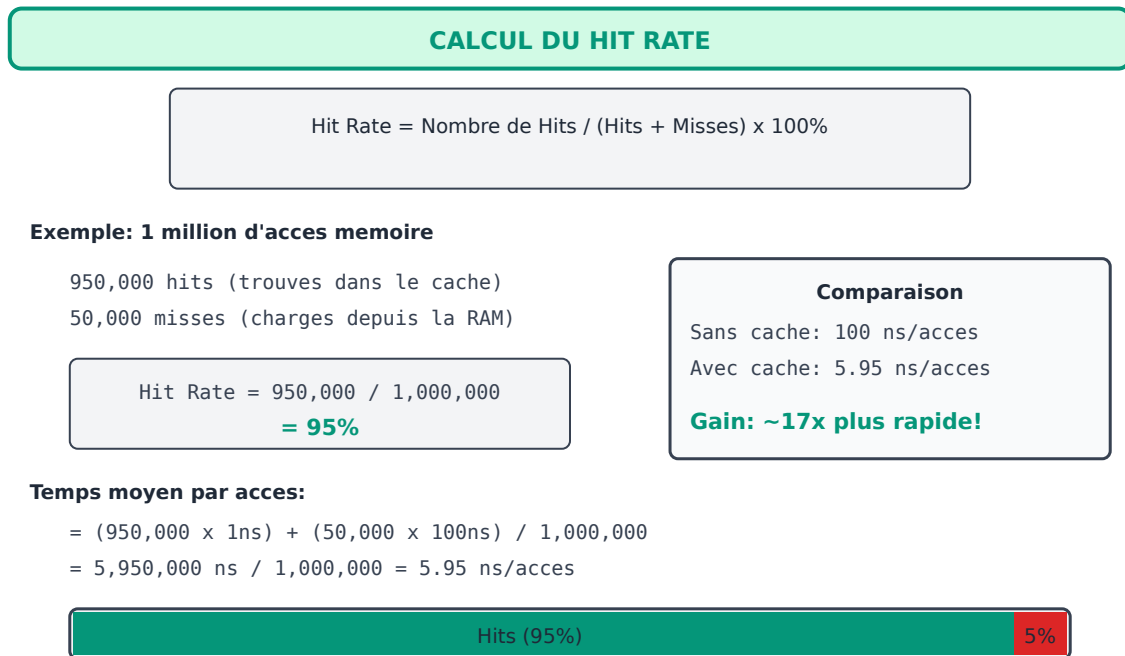


Figure 14.20: Calcul du Hit Rate

14.9 Visualiser le Cache avec le CPU Visualizer

Le **CPU Visualizer** vous permet d'observer le comportement du cache en temps réel pendant l'exécution d'un programme.

14.9.1 Accéder au Visualizer

```
cd web
npm run dev
# Ouvrir http://localhost:5173 -> CPU Visualizer
```

14.9.2 La Démo "Cache"

Chargez la démo **"7. Cache"** dans le menu déroulant. Ce programme : 1. Parcourt un tableau de 16 éléments une première fois (cache misses) 2. Parcourt le même tableau une seconde fois (cache hits)

14.9.3 Ce que vous verrez

Panneau “Cache L1” : - **Hits** : Nombre d'accès trouvés dans le cache - **Misses** : Nombre d'accès qui ont dû aller en RAM - **Taux** : Pourcentage de hits (ex: “94.2%”) - **Indicateur HIT/MISS** : Flash vert pour hit, rouge pour miss

Contenu du cache : - **Ligne** : Numéro de la ligne (0-63) - **Valid** : 1 si la ligne contient des données valides - **Tag** : Identifie quelle zone mémoire est stockée - **Données** : Le mot stocké dans la ligne

14.9.4 Exercice Pratique

1. Lancez la démo “Cache” et observez :
 - Au premier parcours : beaucoup de **MISS** (flash rouge)
 - Au second parcours : beaucoup de **HIT** (flash vert)
2. Regardez le taux de hits évoluer :
 - Début : ~0% (cache vide)
 - Après premier parcours : ~50%
 - Fin : ~85-95%
3. Observez les lignes de cache se remplir :
 - Les bits Valid passent de 0 à 1
 - Les Tags s'affichent
 - Les données apparaissent

14.10 Exercices

14.10.1 Exercices HDL

Exercice	Description
CacheLine	Implémenter une ligne de cache avec valid, tag, data
TagCompare	Comparateur de tags pour détecter hit/miss
WordSelect	Sélecteur de mot (4:1) dans une ligne 128 bits
CacheController	Machine à états (IDLE, FETCH, WRITEBACK)

14.10.2 Exercices Assembleur A32

Exercice	Description	Résultat
Accès Séquentiel	Parcours cache-friendly d'un tableau	R0 = 100
Accès avec Stride	Parcours avec sauts (moins efficace)	R0 = 28
Réutilisation Registre	Garder les données en registre	R0 = 91

14.10.3 Exercices C32

Exercice	Description	Résultat
Parcours en Ligne	Accès row-major (cache-friendly)	120
Parcours en Colonne	Accès column-major (moins efficace)	120
Traitement par Blocs	Technique de blocking	120
Localité Temporelle	Réutiliser les données	30

14.11 Résumé Visuel

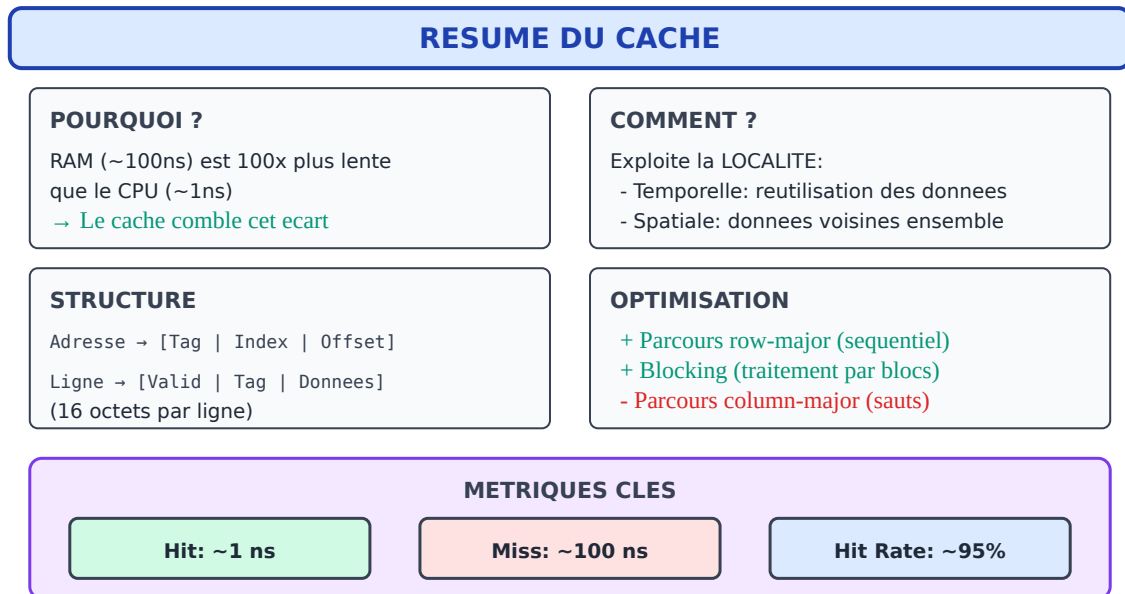


Figure 14.21: Résumé du cache

14.12 Points Clés à Retenir

1. **La RAM est lente** : ~100x plus lente que le cache
2. **Le cache exploite la localité** : temporelle et spatiale
3. **L'ordre d'accès compte** : row-major » column-major
4. **Réutilisez les données** : gardez-les en registre ou en cache
5. **Pensez en blocs** : traitez des données qui tiennent dans le cache

Ces principes s'appliquent à tous les niveaux de programmation, du code assembleur aux applications modernes !

14.13 Auto-évaluation

Testez votre compréhension du cache.

14.13.1 Questions de compréhension

- Q1.** Qu'est-ce que la localité temporelle et la localité spatiale ?
- Q2.** Pourquoi le parcours en ligne (row-major) est-il plus efficace que le parcours en colonne ?
- Q3.** Qu'est-ce qu'un cache hit et un cache miss ?
- Q4.** Comment fonctionne un cache direct-mapped ?
- Q5.** Qu'est-ce que la technique de "blocking" ?

14.13.2 Mini-défi pratique

Calculez le hit rate pour ce pattern d'accès avec un cache de 4 lignes de 16 octets :

Accès séquentiels aux adresses : 0, 4, 8, 12, 16, 20, 24, 28, 0, 4, 8, 12

*Les solutions se trouvent dans le document **Codex_Solutions**.*

14.13.3 Checklist de validation

- ☐ Expliquer la différence entre localité temporelle et spatiale
- ☐ Comprendre pourquoi le cache existe (hiérarchie mémoire)
- ☐ Calculer un hit rate simple
- ☐ Optimiser un parcours de tableau pour le cache
- ☐ Appliquer la technique de blocking sur une matrice

15 Solutions des Quiz d'Auto-évaluation

Ce document contient les réponses aux questions d'auto-évaluation de chaque chapitre. Essayez de répondre par vous-même avant de consulter les solutions !

Note : Pour les solutions des exercices pratiques HDL, assembleur et C32 du simulateur web, consultez le document **Solutions.md**.

15.1 Chapitre 1 : Logique Booléenne

15.1.1 Q1. Pourquoi les ordinateurs utilisent-ils le binaire plutôt que le décimal ?

Trois raisons principales : 1. **Fiabilité** : Distinguer 2 états (haut/bas) est plus robuste que 10 niveaux 2. **Simplicité** : Les transistors fonctionnent naturellement comme des interrupteurs on/off 3. **Universalité** : Toute logique peut s'exprimer avec Vrai/Faux (algèbre de Boole)

15.1.2 Q2. Pourquoi dit-on que NAND est une porte “universelle” ?

Parce que **toutes les autres portes** (NOT, AND, OR, XOR, MUX, etc.) peuvent être construites uniquement à partir de portes NAND. C'est notre “axiome” de départ — tout le reste en découle.

15.1.3 Q3. Quelle est la sortie de $XOR(1, 1)$? Et de $XOR(0, 1)$?

- $XOR(1, 1) = 0$ (les entrées sont identiques)
- $XOR(0, 1) = 1$ (les entrées sont différentes)

XOR vaut 1 **si et seulement si** les entrées sont différentes.

15.1.4 Q4. Un multiplexeur (MUX) a 2 entrées de données a et b, et un signal de sélection sel. Si sel = 1, quelle entrée est transmise en sortie ?

Si sel = 1, la sortie est b. Si sel = 0, la sortie est a.

Le MUX “choisit” entre ses entrées selon le signal de sélection.

15.1.5 Q5. Combien de portes NAND faut-il au minimum pour construire un inverseur (NOT) ?

Une seule porte NAND suffit : $\text{NOT}(a) = \text{NAND}(a, a)$

Quand les deux entrées du NAND sont identiques, on obtient l'inverse.

15.1.6 Mini-défi : Table de vérité de $F(a, b) = \text{AND}(\text{OR}(a, b), \text{NOT}(a))$

a	b	OR(a,b)	NOT(a)	F
0	0	0	1	0
0	1	1	1	1
1	0	1	0	0
1	1	1	0	0

Cette fonction est équivalente à $\text{AND}(\text{NOT}(a), b)$, soit “b ET NON a”.

15.2 Chapitre 2 : Arithmétique Binaire

15.2.1 Q1. En complément à deux sur 8 bits, quelle est la représentation de -1 ?

0xFF (ou 11111111 en binaire).

En complément à deux, -1 est représenté par tous les bits à 1, quelle que soit la largeur.

15.2.2 Q2. Pourquoi le complément à deux est-il préféré au “signe + magnitude” ?

Trois avantages majeurs : 1. **Un seul zéro** (pas de +0 et -0) 2. **L'addition fonctionne identiquement** pour les nombres signés et non-signés 3. **La soustraction devient une addition** : $A - B = A + (\sim B + 1)$

15.2.3 Q3. Quel est le rôle de la retenue entrante (Cin) dans un additionneur complet ?

Elle permet de **propager la retenue** de la colonne précédente. C'est essentiel pour chaîner plusieurs additionneurs et créer un additionneur multi-bits. Sans Cin, on ne pourrait additionner que des bits individuels.

15.2.4 Q4. Une ALU reçoit les opérandes A=5 et B=3, avec l'opération SUB. Quel est le résultat et quels flags sont activés ?

- **Résultat** : $5 - 3 = 2$
- **Flags** : Aucun flag actif
 - Z = 0 (résultat non nul)
 - N = 0 (résultat positif)
 - C = 1 (pas d'emprunt en soustraction non-signée)
 - V = 0 (pas de débordement signé)

15.2.5 Q5. Quelle opération ALU permet de mettre à zéro les bits 4-7 d'un registre tout en préservant les autres ?

AND avec un masque : $R = R \text{ AND } 0x0F$

Le masque 0x0F (00001111 en binaire) préserve les bits 0-3 et force les bits 4-7 à zéro.

15.2.6 Mini-défi : 0x7F + 0x01 en 8 bits signés

- $0x7F = 127$ (le plus grand positif en 8 bits signés)
- $0x7F + 0x01 = 0x80 = 128$ en non-signé, mais **-128 en signé**

C'est un **débordement (overflow)** ! Le flag V serait activé car on passe de positif à négatif.

15.3 Chapitre 3 : Mémoire

15.3.1 Q1. Quelle est la différence fondamentale entre un circuit combinatoire et un circuit séquentiel ?

Un circuit **combinatoire** : la sortie dépend uniquement des entrées actuelles. Un circuit **séquentiel** : la sortie dépend des entrées ET de l'état précédent (il a une "mémoire").

La bascule D (DFF) est l'élément qui introduit la notion de temps et de mémoire.

15.3.2 Q2. Pourquoi le signal d'horloge (clk) est-il essentiel dans les circuits séquentiels ?

L'horloge **synchronise** toutes les mises à jour de l'état. Sans elle : - Les données se propageraient de manière chaotique - Des "courses" (race conditions) créeraient des résultats imprévisibles - On ne pourrait pas garantir un comportement déterministe

15.3.3 Q3. Un registre 32 bits est construit à partir de combien de bascules D ?

32 bascules D, une par bit. Chaque DFF mémorise un bit, donc un registre de N bits nécessite N DFF.

15.3.4 Q4. Quelle est la capacité totale d'une RAM de 1024 mots de 32 bits, en kilooctets ?

- $1024 \text{ mots} \times 32 \text{ bits/mot} = 32768 \text{ bits}$
- $32768 \text{ bits} \div 8 = 4096 \text{ octets}$
- $4096 \text{ octets} \div 1024 = 4 \text{ Ko}$

15.3.5 Q5. Dans une RAM8, combien de bits d'adresse sont nécessaires et pourquoi ?

3 bits d'adresse, car $2^3 = 8$.

Avec 3 bits, on peut adresser 8 emplacements distincts (000 à 111).

15.3.6 Mini-défi : Registre avec Enable

Pour ajouter un signal enable à une DFF :


```
-- Solution : MUX devant la DFF
signal d_mux : bit;
d_mux <= d when enable = '1' else q; -- Si enable=0, reboucler la sortie
u_dff: DFF port map (d => d_mux, clk => clk, q => q);
```

Le MUX choisit entre la nouvelle donnée (si enable=1) ou la valeur actuelle (si enable=0).

15.4 Chapitre 4 : Architecture

15.4.1 Q1. Quelle est la différence entre l'architecture von Neumann et Harvard ?

- **Von Neumann** : Une seule mémoire pour les instructions et les données. Plus simple, mais un seul accès mémoire par cycle (goulot d'étranglement).
- **Harvard** : Mémoires séparées pour instructions et données. Permet de charger une instruction pendant qu'on accède aux données (plus rapide).

15.4.2 Q2. Pourquoi utilise-t-on des registres plutôt que d'accéder directement à la RAM ?

Les registres sont **beaucoup plus rapides** que la RAM : - Accès en 1 cycle vs plusieurs cycles pour la RAM - Intégrés dans le CPU, pas de bus externe - Nombre limité (16-32) mais suffisant pour les calculs immédiats

Les registres sont le "bureau de travail" du CPU, la RAM est "l'armoire de rangement".

15.4.3 Q3. À quoi sert le registre PC (Program Counter) ?

Le PC contient l'**adresse de la prochaine instruction** à exécuter. Après chaque instruction : - Il est incrémenté automatiquement (PC = PC + 4 pour des instructions 32 bits) - Ou modifié par un branchement (saut, appel de fonction)

15.4.4 Q4. Qu'est-ce que le MMIO (Memory-Mapped I/O) ?

Le MMIO permet d'accéder aux **périphériques** (écran, clavier, etc.) comme s'ils étaient de la mémoire. Chaque périphérique a une adresse réservée : - Écrire à

0xFFFF0000 → envoie un caractère à la sortie - Lire à 0xFFFF0004 → récupère un caractère du clavier

Avantage : pas besoin d'instructions spéciales, les mêmes LDR/STR fonctionnent.

15.4.5 Q5. Dans le cycle fetch-decode-execute, que se passe-t-il pendant la phase "decode" ?

Pendant le **décodage** : 1. L'instruction est analysée pour identifier son type (ALU, mémoire, branchement) 2. Les registres sources sont identifiés et lus 3. L'unité de contrôle génère les signaux appropriés pour l'exécution

15.4.6 Mini-défi : Pourquoi 16 registres avec 4 bits ?

Avec 4 bits, on peut encoder $2^4 = 16$ valeurs différentes (0000 à 1111), donc 16 registres (R0 à R15).

Pour 32 registres, il faudrait 5 bits ($2^5 = 32$).

15.5 Chapitre 5 : CPU

15.5.1 Q1. Quelles sont les 5 étapes du cycle d'instruction dans un pipeline classique ?

1. **IF** (Instruction Fetch) : Charger l'instruction depuis la mémoire
2. **ID** (Instruction Decode) : Décoder et lire les registres
3. **EX** (Execute) : Effectuer le calcul dans l'ALU
4. **MEM** (Memory Access) : Accéder à la mémoire si nécessaire
5. **WB** (Write Back) : Écrire le résultat dans le registre destination

15.5.2 Q2. Qu'est-ce qu'un aléa de données (data hazard) et comment le résoudre ?

Un **aléa de données** se produit quand une instruction a besoin du résultat d'une instruction précédente qui n'est pas encore terminée.

Solutions : - **Forwarding** : Transmettre le résultat directement depuis EX/MEM vers l'instruction suivante - **Stall** : Insérer des bulles (attendre) si le forwarding n'est pas possible - **Réordonnement** : Le compilateur réorganise le code pour éviter les dépendances

15.5.3 Q3. Pourquoi le Program Counter est-il incrémenté de 4 (et non de 1) sur une architecture 32 bits ?

Parce que chaque instruction occupe **4 octets** (32 bits). La mémoire est adressée par octet, donc passer à l'instruction suivante nécessite d'avancer de 4 adresses.

15.5.4 Q4. Quel composant décide si un branchement conditionnel doit être pris ?

L'**unité de contrôle** utilise les **flags de l'ALU** (Z, N, C, V) pour décider. Par exemple :
- BEQ (Branch if Equal) vérifie si $Z = 1$
- BNE (Branch if Not Equal) vérifie si $Z = 0$
- BGT (Branch if Greater Than) vérifie si $Z = 0$ ET $N = V$

15.5.5 Q5. Quelle est la différence entre les instructions LDR et STR ?

- **LDR** (Load Register) : Lit une valeur **de la mémoire vers un registre**
 - LDR R0, [R1] $\rightarrow R0 = \text{Mem}[R1]$
- **STR** (Store Register) : Écrit une valeur **d'un registre vers la mémoire**
 - STR R0, [R1] $\rightarrow \text{Mem}[R1] = R0$

15.5.6 Mini-défi : Temps d'exécution avec pipeline

Sans pipeline : $5 \times 10 = 50$ cycles

Avec pipeline (après remplissage initial) :
- Remplissage : 4 cycles
- Puis 1 instruction complétée par cycle : 10 cycles
- Total : **14 cycles** (ou $10 + 4 = 14$)

Gain : $50/14 \approx 3.6\times$ plus rapide

15.6 Chapitre 6 : Assembleur

15.6.1 Q1. Quelle est la différence entre une instruction et une directive en assembleur ?

- **Instruction** : Traduite en code machine, exécutée par le CPU (ex: ADD R0, R1, R2)
- **Directive** : Commande pour l'assembleur lui-même, pas de code généré (ex: .data, .word 42, .global main)

Les directives contrôlent l'organisation du programme, pas son exécution.

15.6.2 Q2. Pourquoi doit-on sauvegarder LR avant d'appeler une sous-fonction avec BL ?

Parce que BL (Branch and Link) **écrase LR** avec l'adresse de retour. Si on appelle une autre fonction sans sauvegarder, l'adresse de retour originale est perdue.

```
func:
    PUSH {LR}      ; Sauvegarder AVANT le BL
    BL autre_func  ; LR est écrasé ici
    POP {LR}       ; Restaurer
    BX LR          ; Retour correct
```

15.6.3 Q3. Comment charger une valeur 32 bits arbitraire (ex: 0x12345678) dans un registre ?

Utiliser LDR Rd, =value qui utilise le **literal pool** :

```
LDR R0, =0x12345678 ; L'assembleur place la constante en mémoire
```

L'instruction MOV ne peut charger que des immédiats sur 12 bits (0-4095 ou valeurs rotées).

15.6.4 Q4. À quoi sert la directive .ltorg ?

.ltorg force l'assembleur à **placer le literal pool** (les constantes utilisées par LDR Rd, =value) à cet endroit.

C'est nécessaire car le PC-relative offset est limité ($\pm 4\text{KB}$). Si les constantes sont trop loin, l'assembleur génère une erreur E1008.

15.6.5 Q5. Quelle est la convention pour les registres "callee-saved" vs "caller-saved" ?

- **Caller-saved** (R0-R3) : L'appelant doit les sauvegarder s'il veut les préserver. La fonction appelée peut les modifier librement.
- **Callee-saved** (R4-R11) : La fonction appelée doit les restaurer avant de retourner. L'appelant peut compter sur leur préservation.

15.6.6 Mini-défi : Valeur finale de R0

```
MOV R0, #10
MOV R1, #3
loop:
SUBS R0, R0, R1
BPL loop
```

Trace : - R0 = 10, R0 - 3 = 7 (positif, N=0) → boucle - R0 = 7, R0 - 3 = 4 (positif, N=0) → boucle - R0 = 4, R0 - 3 = 1 (positif, N=0) → boucle - R0 = 1, R0 - 3 = -2 (négatif, N=1) → sort

R0 = -2 (ou 0xFFFFFFFEE en non-signé)

15.7 Chapitre 7 : Compilateur

15.7.1 Q1. Quelles sont les principales phases d'un compilateur ?

1. **Analyse lexicale** : Transforme le texte en tokens (mots-clés, identifiants, opérateurs)
2. **Analyse syntaxique** : Construit l'arbre syntaxique (AST) selon la grammaire
3. **Analyse sémantique** : Vérifie les types, la portée des variables
4. **Génération de code** : Produit le code assembleur/machine

15.7.2 Q2. Qu'est-ce qu'un AST et à quoi sert-il ?

L'**AST** (Abstract Syntax Tree) est une représentation arborescente du programme qui capture sa structure logique sans les détails syntaxiques (parenthèses, points-virgules).

Il sert de représentation intermédiaire entre le code source et le code généré, facilitant l'analyse et les transformations.

15.7.3 Q3. Comment le compilateur gère-t-il les variables locales d'une fonction ?

Les variables locales sont stockées sur la **pile** (stack) : - À l'entrée de la fonction, l'espace est réservé (SUB SP, SP, #n) - Chaque variable a un offset par rapport à SP ou FP - À la sortie, l'espace est libéré (ADD SP, SP, #n)

15.7.4 Q4. Quelle est la différence entre une erreur syntaxique et sémantique ?

- **Erreur syntaxique** : Le code ne respecte pas la grammaire (ex: parenthèse manquante, point-virgule oublié)
- **Erreur sémantique** : Le code est syntaxiquement correct mais n'a pas de sens (ex: `int x = "hello"`, variable non déclarée)

15.7.5 Q5. Pourquoi le compilateur génère-t-il parfois du code qui semble inefficace ?

Sans optimisation, le compilateur génère du code **direct et prévisible** qui suit fidèlement la structure du source. Par exemple : - Chaque variable a son emplacement mémoire - Chaque opération génère ses instructions

Les optimisations (niveau -O2, -O3) réduisent ces inefficacités mais compliquent le débogage.

15.7.6 Mini-défi : Code assembleur généré

Pour `int x = a + b * c;` :

```
; Supposons a, b, c dans R0, R1, R2
MUL R3, R1, R2    ; R3 = b * c (multiplication d'abord)
ADD R4, R0, R3    ; R4 = a + (b * c)
; R4 contient x
```

Le compilateur respecte la **priorité des opérateurs** : multiplication avant addition.

15.8 Chapitre 8 : Le Langage C32

15.8.1 Q1. Quelles sont les principales différences entre C32 et C standard ?

- Pas d'opérateurs `++` et `--` (utiliser `x = x + 1`)
- Pas de `float/double` (entiers uniquement)
- Pas d'`enum`
- Préprocesseur minimal
- Pas de bibliothèque standard (juste `putc`, `getc`, `exit`)

15.8.2 Q2. Comment passer un tableau à une fonction en C32 ?

Les tableaux sont passés par **pointeur** (l'adresse du premier élément) :

```
void process(int *arr, int len) {
    for (int i = 0; i < len; i = i + 1) {
        arr[i] = arr[i] * 2; // Modifie le tableau original
    }
}

int main() {
    int data[5];
    process(data, 5); // 'data' est converti en &data[0]
}
```

15.8.3 Q3. Quelle est la différence entre *p et &x ?

- &x : L'**adresse** de la variable x (opérateur "adresse de")
- *p : La **valeur** pointée par p (opérateur de déréférencement)

```
int x = 42;
int *p = &x; // p contient l'adresse de x
int y = *p; // y = 42 (valeur lue à l'adresse p)
```

15.8.4 Q4. Comment accéder au framebuffer pour dessiner un pixel ?

Via **MMIO** (Memory-Mapped I/O) :

```
uint *screen = (uint*)0x00400000; // Adresse du framebuffer
screen[y * 320 + x] = 0xFFFFFFFF; // Pixel blanc à (x, y)
```

Chaque mot de 32 bits représente les couleurs de 32 pixels (1 bit par pixel en monochrome).

15.8.5 Q5. Pourquoi faut-il toujours terminer une chaîne par '\0' ?

Le caractère '\0' (valeur 0) marque la **fin de la chaîne**. Sans lui, les fonctions de traitement de chaînes (affichage, copie, comparaison) ne sauraient pas où s'arrêter et liraient au-delà de la chaîne.

```
char *msg = "Hello"; // Le compilateur ajoute '\0' automatiquement
// En mémoire : 'H' 'e' 'l' 'l' 'o' '\0'
```

15.8.6 Mini-défi : Fonction de longueur de chaîne

```
int strlen(char *s) {  
    int len = 0;  
    while (*s != '\0') {  
        len = len + 1;  
        s = s + 1;  
    }  
    return len;  
}
```

15.9 Chapitre 9 : Système d'Exploitation

15.9.1 Q1. Quels sont les trois rôles principaux d'un système d'exploitation ?

1. **Abstraction du matériel** : Fournir une interface unifiée (fichiers, processus) au-dessus du matériel varié
2. **Gestion des ressources** : Allouer CPU, mémoire, périphériques entre les programmes
3. **Protection et isolation** : Empêcher les programmes de se perturber mutuellement

15.9.2 Q2. Comment fonctionne un appel système (syscall) ?

1. Le programme place le numéro de syscall et les arguments dans les registres
2. L'instruction SVC (Supervisor Call) déclenche une exception
3. Le CPU passe en mode privilégié et saute au gestionnaire de syscall
4. L'OS exécute le service demandé
5. Le contrôle retourne au programme avec le résultat dans R0

15.9.3 Q3. Qu'est-ce que le memory mapping et pourquoi est-il utile ?

Le **memory mapping** associe des régions de l'espace d'adressage à des ressources : - **RAM physique** : Pour le code et les données du programme - **Périphériques (MMIO)** : Pour communiquer avec le matériel - **Fichiers** : Pour accéder aux fichiers comme de la mémoire

Avantage : Une interface uniforme (load/store) pour tout accès.

15.9.4 Q4. Pourquoi l'OS a-t-il besoin d'un mode privilégié séparé ?

Pour **protéger le système** : - Seul l'OS peut accéder au matériel directement - Seul l'OS peut modifier les tables de pages mémoire - Les programmes utilisateur ne peuvent pas crasher le système entier

Sans cette séparation, un bug dans un programme pourrait corrompre tout le système.

15.9.5 Q5. Comment l'OS gère-t-il plusieurs programmes en même temps ?

Par le **multitâche préemptif** : 1. Un timer génère des interruptions régulières 2. À chaque interruption, l'OS peut changer de processus 3. L'état (registres, PC) du processus courant est sauvegardé 4. L'état d'un autre processus est restauré 5. L'exécution reprend dans le nouveau processus

Chaque processus croit avoir le CPU pour lui seul.

15.9.6 Mini-défi : Allocation mémoire simple

```
// Allocateur "bump pointer" minimal
int heap_ptr = 0x200000; // Début du tas

void* malloc(int size) {
    int ptr = heap_ptr;
    heap_ptr = heap_ptr + size;
    // Aligner sur 4 octets
    heap_ptr = (heap_ptr + 3) & ~3;
    return (void*)ptr;
}

// Note : pas de free() dans cette version simple !
```

15.10 Chapitre 11 : Mémoire Cache

15.10.1 Q1. Pourquoi a-t-on besoin d'une mémoire cache ?

Pour combler l'**écart de vitesse** entre le CPU et la RAM : - CPU : peut traiter une opération par cycle (< 1 ns) - RAM : temps d'accès de 50-100 ns (50-100 cycles perdus !)

Le cache stocke les données récemment utilisées dans une mémoire très rapide (SRAM) proche du CPU.

15.10.2 Q2. Qu'est-ce que la localité spatiale et temporelle ?

- **Localité temporelle** : Une donnée accédée récemment sera probablement réaccédée bientôt (ex: variables de boucle)
- **Localité spatiale** : Les données proches en mémoire sont souvent accédées ensemble (ex: éléments consécutifs d'un tableau)

Le cache exploite ces deux principes pour prédire quelles données garder.

15.10.3 Q3. Quelle est la différence entre un cache direct-mapped et associatif ?

- **Direct-mapped** : Chaque adresse mémoire ne peut aller qu'à **un seul** emplacement du cache. Simple mais conflits fréquents.
- **Associatif (N-way)** : Chaque adresse peut aller dans **N emplacements** différents. Moins de conflits mais plus complexe.
- **Totalement associatif** : N'importe où dans le cache. Optimal mais coûteux.

15.10.4 Q4. Que se passe-t-il lors d'un "cache miss" ?

1. La donnée demandée n'est pas dans le cache
2. Le CPU est mis en attente (stall)
3. La ligne de cache complète est chargée depuis la mémoire principale
4. Si le cache est plein, une ligne existante est évincée (et écrite en RAM si modifiée)
5. L'accès est réessayé (cache hit cette fois)

15.10.5 Q5. Quelle est la différence entre write-through et write-back ?

- **Write-through** : Chaque écriture va **immédiatement** en cache ET en RAM. Simple mais lent.
- **Write-back** : Les écritures vont **seulement** au cache. La RAM est mise à jour seulement quand la ligne est évincée. Plus rapide mais plus complexe.

15.10.6 Mini-défi : Optimisation de parcours de matrice

```
// Version optimisée (parcours par lignes)
for (int i = 0; i < N; i = i + 1) {
    for (int j = 0; j < N; j = j + 1) {
        sum = sum + matrix[i][j]; // Accès séquentiels en mémoire
    }
}
```

En C, les matrices sont stockées **ligne par ligne** (row-major). Parcourir par lignes (i puis j) accède à des adresses consécutives → excellent pour le cache (localité spatiale).

Parcourir par colonnes (j puis i) saute de N éléments à chaque accès → très mauvais pour le cache.

15.11 Chapitre 10bis : Débogage

Les solutions des exercices de débogage sont dans le chapitre lui-même, car ils font partie intégrante de l'apprentissage du débogage. Consultez les balises <details> dans le chapitre 10bis.

16 Livre des Solutions

Ce document contient les solutions complètes de tous les exercices du simulateur web Codex.

Note: Ces solutions sont fournies pour reference apres avoir tente les exercices.

16.1 A. Solutions HDL (Portes Logiques)

16.1.1 Inv

```
-- Inverter (NOT gate)
-- Inv(a) = Nand(a, a)

entity Inv is
  port(
    a : in bit;
    y : out bit
  );
end entity;

architecture rtl of Inv is
  component nand2
    port(a : in bit; b : in bit; y : out bit);
  end component;
begin
  u0: nand2 port map (a => a, b => a, y => y);
end architecture;
```

16.1.2 And2

```
-- AND gate
-- And2(a,b) = Inv(Nand(a,b))

entity And2 is
  port(
    a : in bit;
    b : in bit;
    y : out bit
  );
end entity;

architecture rtl of And2 is
  component nand2
    port(a : in bit; b : in bit; y : out bit);
  end component;
  component Inv
    port(a : in bit; y : out bit);
  end component;
  signal t : bit;
begin
  u0: nand2 port map (a => a, b => b, y => t);
  u1: Inv port map (a => t, y => y);
end architecture;
```

16.1.3 Or2

```
-- OR gate
-- Or2(a,b) = Nand(Inv(a), Inv(b))

entity Or2 is
  port(
    a : in bit;
    b : in bit;
    y : out bit
  );
end entity;

architecture rtl of Or2 is
  component nand2
    port(a : in bit; b : in bit; y : out bit);
  end component;
  component Inv
    port(a : in bit; y : out bit);
  end component;
```

```
end component;
signal na, nb : bit;
begin
u0: Inv port map (a => a, y => na);
u1: Inv port map (a => b, y => nb);
u2: nand2 port map (a => na, b => nb, y => y);
end architecture;
```

16.1.4 Xor2

```
-- XOR gate
-- Xor2(a,b) = Or2(And2(a, Inv(b)), And2(Inv(a), b))

entity Xor2 is
  port(
    a : in bit;
    b : in bit;
    y : out bit
  );
end entity;

architecture rtl of Xor2 is
  component Inv
    port(a : in bit; y : out bit);
  end component;
  component And2
    port(a : in bit; b : in bit; y : out bit);
  end component;
  component Or2
    port(a : in bit; b : in bit; y : out bit);
  end component;
  signal na, nb, t1, t2 : bit;
begin
u0: Inv port map (a => a, y => na);
u1: Inv port map (a => b, y => nb);
u2: And2 port map (a => a, b => nb, y => t1);
u3: And2 port map (a => na, b => b, y => t2);
u4: Or2 port map (a => t1, b => t2, y => y);
end architecture;
```

16.1.5 Mux

```
-- 2-way Multiplexer
-- if sel=0 then y=a else y=b

entity Mux is
  port(
    a   : in bit;
    b   : in bit;
    sel : in bit;
    y   : out bit;
  );
end entity;

architecture rtl of Mux is
  component Inv
    port(a : in bit; y : out bit);
  end component;
  component And2
    port(a : in bit; b : in bit; y : out bit);
  end component;
  component Or2
    port(a : in bit; b : in bit; y : out bit);
  end component;
  signal nsel, t1, t2 : bit;
begin
  u0: Inv port map (a => sel, y => nsel);
  u1: And2 port map (a => a, b => nsel, y => t1);
  u2: And2 port map (a => b, b => sel, y => t2);
  u3: Or2 port map (a => t1, b => t2, y => y);
end architecture;
```

16.1.6 DMux

```
-- Demultiplexer
-- if sel=0 then {a,b}={x,0} else {a,b}={0,x}

entity DMux is
  port(
    x   : in bit;
    sel : in bit;
    a   : out bit;
    b   : out bit;
  );
end entity;
```

```

);
end entity;

architecture rtl of DMux is
    component Inv
        port(a : in bit; y : out bit);
    end component;
    component And2
        port(a : in bit; b : in bit; y : out bit);
    end component;
    signal nsel : bit;
begin
    u0: Inv port map (a => sel, y => nsel);
    u1: And2 port map (a => x, b => nsel, y => a);
    u2: And2 port map (a => x, b => sel, y => b);
end architecture;

```

16.1.7 Inv16

```

-- 16-bit Inverter
entity Inv16 is
    port(
        a : in bits(15 downto 0);
        y : out bits(15 downto 0)
    );
end entity;

architecture rtl of Inv16 is
    component Inv
        port(a : in bit; y : out bit);
    end component;
begin
    u0: Inv port map (a => a(0), y => y(0));
    u1: Inv port map (a => a(1), y => y(1));
    u2: Inv port map (a => a(2), y => y(2));
    u3: Inv port map (a => a(3), y => y(3));
    u4: Inv port map (a => a(4), y => y(4));
    u5: Inv port map (a => a(5), y => y(5));
    u6: Inv port map (a => a(6), y => y(6));
    u7: Inv port map (a => a(7), y => y(7));
    u8: Inv port map (a => a(8), y => y(8));
    u9: Inv port map (a => a(9), y => y(9));
    u10: Inv port map (a => a(10), y => y(10));
    u11: Inv port map (a => a(11), y => y(11));

```



```
u12: Inv port map (a => a(12), y => y(12));
u13: Inv port map (a => a(13), y => y(13));
u14: Inv port map (a => a(14), y => y(14));
u15: Inv port map (a => a(15), y => y(15));
end architecture;
```

16.1.8 And16

```
-- 16-bit AND
entity And16 is
  port(
    a : in bits(15 downto 0);
    b : in bits(15 downto 0);
    y : out bits(15 downto 0)
  );
end entity;

architecture rtl of And16 is
  component And2
    port(a : in bit; b : in bit; y : out bit);
  end component;
begin
  u0: And2 port map (a => a(0), b => b(0), y => y(0));
  u1: And2 port map (a => a(1), b => b(1), y => y(1));
  u2: And2 port map (a => a(2), b => b(2), y => y(2));
  u3: And2 port map (a => a(3), b => b(3), y => y(3));
  u4: And2 port map (a => a(4), b => b(4), y => y(4));
  u5: And2 port map (a => a(5), b => b(5), y => y(5));
  u6: And2 port map (a => a(6), b => b(6), y => y(6));
  u7: And2 port map (a => a(7), b => b(7), y => y(7));
  u8: And2 port map (a => a(8), b => b(8), y => y(8));
  u9: And2 port map (a => a(9), b => b(9), y => y(9));
  u10: And2 port map (a => a(10), b => b(10), y => y(10));
  u11: And2 port map (a => a(11), b => b(11), y => y(11));
  u12: And2 port map (a => a(12), b => b(12), y => y(12));
  u13: And2 port map (a => a(13), b => b(13), y => y(13));
  u14: And2 port map (a => a(14), b => b(14), y => y(14));
  u15: And2 port map (a => a(15), b => b(15), y => y(15));
end architecture;
```

16.1.9 Or16

```
-- 16-bit OR
entity Or16 is
  port(
    a : in bits(15 downto 0);
    b : in bits(15 downto 0);
    y : out bits(15 downto 0)
  );
end entity;

architecture rtl of Or16 is
  component Or2
    port(a : in bit; b : in bit; y : out bit);
  end component;
begin
  u0: Or2 port map (a => a(0), b => b(0), y => y(0));
  u1: Or2 port map (a => a(1), b => b(1), y => y(1));
  u2: Or2 port map (a => a(2), b => b(2), y => y(2));
  u3: Or2 port map (a => a(3), b => b(3), y => y(3));
  u4: Or2 port map (a => a(4), b => b(4), y => y(4));
  u5: Or2 port map (a => a(5), b => b(5), y => y(5));
  u6: Or2 port map (a => a(6), b => b(6), y => y(6));
  u7: Or2 port map (a => a(7), b => b(7), y => y(7));
  u8: Or2 port map (a => a(8), b => b(8), y => y(8));
  u9: Or2 port map (a => a(9), b => b(9), y => y(9));
  u10: Or2 port map (a => a(10), b => b(10), y => y(10));
  u11: Or2 port map (a => a(11), b => b(11), y => y(11));
  u12: Or2 port map (a => a(12), b => b(12), y => y(12));
  u13: Or2 port map (a => a(13), b => b(13), y => y(13));
  u14: Or2 port map (a => a(14), b => b(14), y => y(14));
  u15: Or2 port map (a => a(15), b => b(15), y => y(15));
end architecture;
```

16.1.10 Mux16

```
-- 16-bit 2-way Multiplexer
entity Mux16 is
  port(
    a : in bits(15 downto 0);
    b : in bits(15 downto 0);
    sel : in bit;
    y : out bits(15 downto 0)
  );
end entity;
```

```

);
end entity;

architecture rtl of Mux16 is
    component Mux
        port(a : in bit; b : in bit; sel : in bit; y : out bit);
    end component;
begin
    u0: Mux port map (a => a(0), b => b(0), sel => sel, y => y(0));
    u1: Mux port map (a => a(1), b => b(1), sel => sel, y => y(1));
    u2: Mux port map (a => a(2), b => b(2), sel => sel, y => y(2));
    u3: Mux port map (a => a(3), b => b(3), sel => sel, y => y(3));
    u4: Mux port map (a => a(4), b => b(4), sel => sel, y => y(4));
    u5: Mux port map (a => a(5), b => b(5), sel => sel, y => y(5));
    u6: Mux port map (a => a(6), b => b(6), sel => sel, y => y(6));
    u7: Mux port map (a => a(7), b => b(7), sel => sel, y => y(7));
    u8: Mux port map (a => a(8), b => b(8), sel => sel, y => y(8));
    u9: Mux port map (a => a(9), b => b(9), sel => sel, y => y(9));
    u10: Mux port map (a => a(10), b => b(10), sel => sel, y => y(10));
    u11: Mux port map (a => a(11), b => b(11), sel => sel, y => y(11));
    u12: Mux port map (a => a(12), b => b(12), sel => sel, y => y(12));
    u13: Mux port map (a => a(13), b => b(13), sel => sel, y => y(13));
    u14: Mux port map (a => a(14), b => b(14), sel => sel, y => y(14));
    u15: Mux port map (a => a(15), b => b(15), sel => sel, y => y(15));
end architecture;

```

16.1.11 Or8Way

```

-- 8-way OR
entity Or8Way is
    port(
        a : in bits(7 downto 0);
        y : out bit
    );
end entity;

architecture rtl of Or8Way is
    component Or2
        port(a : in bit; b : in bit; y : out bit);
    end component;
    signal t1, t2, t3, t4, t5, t6 : bit;
begin
    u0: Or2 port map (a => a(0), b => a(1), y => t1);
    u1: Or2 port map (a => a(2), b => a(3), y => t2);

```

```

u2: 0r2 port map (a => a(4), b => a(5), y => t3);
u3: 0r2 port map (a => a(6), b => a(7), y => t4);
u4: 0r2 port map (a => t1, b => t2, y => t5);
u5: 0r2 port map (a => t3, b => t4, y => t6);
u6: 0r2 port map (a => t5, b => t6, y => y);
end architecture;

```

16.1.12 Mux4Way16

```

-- 4-way 16-bit Multiplexer
entity Mux4Way16 is
  port(
    a  : in bits(15 downto 0);
    b  : in bits(15 downto 0);
    c  : in bits(15 downto 0);
    d  : in bits(15 downto 0);
    sel : in bits(1 downto 0);
    y   : out bits(15 downto 0)
  );
end entity;

architecture rtl of Mux4Way16 is
  component Mux16
    port(a : in bits(15 downto 0); b : in bits(15 downto 0); sel : in bit; y :
      out bits(15 downto 0));
  end component;
  signal ab, cd : bits(15 downto 0);
begin
  u0: Mux16 port map (a => a, b => b, sel => sel(0), y => ab);
  u1: Mux16 port map (a => c, b => d, sel => sel(0), y => cd);
  u2: Mux16 port map (a => ab, b => cd, sel => sel(1), y => y);
end architecture;

```

16.1.13 Mux8Way16

```

-- 8-way 16-bit Multiplexer
entity Mux8Way16 is
  port(
    a  : in bits(15 downto 0);

```

```

    b : in bits(15 downto 0);
    c : in bits(15 downto 0);
    d : in bits(15 downto 0);
    e : in bits(15 downto 0);
    f : in bits(15 downto 0);
    g : in bits(15 downto 0);
    h : in bits(15 downto 0);
    sel : in bits(2 downto 0);
    y : out bits(15 downto 0)
);
end entity;

architecture rtl of Mux8Way16 is
    component Mux4Way16
        port(a,b,c,d : in bits(15 downto 0); sel : in bits(1 downto 0); y : out
            ⇨ bits(15 downto 0));
    end component;
    component Mux16
        port(a : in bits(15 downto 0); b : in bits(15 downto 0); sel : in bit; y :
            ⇨ out bits(15 downto 0));
    end component;
    signal lo, hi : bits(15 downto 0);
begin
    u0: Mux4Way16 port map (a => a, b => b, c => c, d => d, sel => sel(1 downto
        ⇨ 0), y => lo);
    u1: Mux4Way16 port map (a => e, b => f, c => g, d => h, sel => sel(1 downto
        ⇨ 0), y => hi);
    u2: Mux16 port map (a => lo, b => hi, sel => sel(2), y => y);
end architecture;

```

16.1.14 DMux4Way

```

-- 4-way Demultiplexer
entity DMux4Way is
    port(
        x : in bit;
        sel : in bits(1 downto 0);
        a : out bit;
        b : out bit;
        c : out bit;
        d : out bit
    );
end entity;

```

```

architecture rtl of DMux4Way is
    component DMux
        port(x : in bit; sel : in bit; a : out bit; b : out bit);
    end component;
    signal lo, hi : bit;
begin
    u0: DMux port map (x => x, sel => sel(1), a => lo, b => hi);
    u1: DMux port map (x => lo, sel => sel(0), a => a, b => b);
    u2: DMux port map (x => hi, sel => sel(0), a => c, b => d);
end architecture;

```

16.1.15 DMux8Way

```

-- 8-way Demultiplexer
entity DMux8Way is
    port(
        x : in bit;
        sel : in bits(2 downto 0);
        a, b, c, d, e, f, g, h : out bit
    );
end entity;

architecture rtl of DMux8Way is
    component DMux
        port(x : in bit; sel : in bit; a : out bit; b : out bit);
    end component;
    component DMux4Way
        port(x : in bit; sel : in bits(1 downto 0); a,b,c,d : out bit);
    end component;
    signal lo, hi : bit;
begin
    u0: DMux port map (x => x, sel => sel(2), a => lo, b => hi);
    u1: DMux4Way port map (x => lo, sel => sel(1 downto 0), a => a, b => b, c =>
        c, d => d);
    u2: DMux4Way port map (x => hi, sel => sel(1 downto 0), a => e, b => f, c =>
        g, d => h);
end architecture;

```

16.1.16 HalfAdder

```
-- Half Adder
entity HalfAdder is
  port(
    a      : in bit;
    b      : in bit;
    sum    : out bit;
    carry  : out bit
  );
end entity;

architecture rtl of HalfAdder is
  component Xor2
    port(a : in bit; b : in bit; y : out bit);
  end component;
  component And2
    port(a : in bit; b : in bit; y : out bit);
  end component;
begin
  u0: Xor2 port map (a => a, b => b, y => sum);
  u1: And2 port map (a => a, b => b, y => carry);
end architecture;
```

16.1.17 FullAdder

```
-- Full Adder
entity FullAdder is
  port(
    a      : in bit;
    b      : in bit;
    cin    : in bit;
    sum    : out bit;
    cout   : out bit
  );
end entity;

architecture rtl of FullAdder is
  component HalfAdder
    port(a : in bit; b : in bit; sum : out bit; carry : out bit);
  end component;
  component Or2
    port(a : in bit; b : in bit; y : out bit);
  end component;
```

```

    end component;
    signal s1, c1, c2 : bit;
begin
    u0: HalfAdder port map (a => a, b => b, sum => s1, carry => c1);
    u1: HalfAdder port map (a => s1, b => cin, sum => sum, carry => c2);
    u2: Or2 port map (a => c1, b => c2, y => cout);
end architecture;

```

16.1.18 Add16

```

-- 16-bit Adder
entity Add16 is
    port(
        a      : in bits(15 downto 0);
        b      : in bits(15 downto 0);
        cin    : in bit;
        sum    : out bits(15 downto 0);
        cout   : out bit
    );
end entity;

architecture rtl of Add16 is
    component FullAdder
        port(a,b,cin : in bit; sum,cout : out bit);
    end component;
    signal c : bits(16 downto 0);
begin
    u0: FullAdder port map (a => a(0), b => b(0), cin => cin, sum => sum(0), cout
        => c(1));
    u1: FullAdder port map (a => a(1), b => b(1), cin => c(1), sum => sum(1), cout
        => c(2));
    u2: FullAdder port map (a => a(2), b => b(2), cin => c(2), sum => sum(2), cout
        => c(3));
    u3: FullAdder port map (a => a(3), b => b(3), cin => c(3), sum => sum(3), cout
        => c(4));
    u4: FullAdder port map (a => a(4), b => b(4), cin => c(4), sum => sum(4), cout
        => c(5));
    u5: FullAdder port map (a => a(5), b => b(5), cin => c(5), sum => sum(5), cout
        => c(6));
    u6: FullAdder port map (a => a(6), b => b(6), cin => c(6), sum => sum(6), cout
        => c(7));
    u7: FullAdder port map (a => a(7), b => b(7), cin => c(7), sum => sum(7), cout
        => c(8));
    u8: FullAdder port map (a => a(8), b => b(8), cin => c(8), sum => sum(8), cout
        => c(9));

```



```

u9: FullAdder port map (a => a(9), b => b(9), cin => c(9), sum => sum(9), cout
    => c(10));
u10: FullAdder port map (a => a(10), b => b(10), cin => c(10), sum => sum(10),
    => cout => c(11));
u11: FullAdder port map (a => a(11), b => b(11), cin => c(11), sum => sum(11),
    => cout => c(12));
u12: FullAdder port map (a => a(12), b => b(12), cin => c(12), sum => sum(12),
    => cout => c(13));
u13: FullAdder port map (a => a(13), b => b(13), cin => c(13), sum => sum(13),
    => cout => c(14));
u14: FullAdder port map (a => a(14), b => b(14), cin => c(14), sum => sum(14),
    => cout => c(15));
u15: FullAdder port map (a => a(15), b => b(15), cin => c(15), sum => sum(15),
    => cout => cout);
end architecture;

```

16.1.19 Inc16

```

-- 16-bit Incrementer
entity Inc16 is
    port(
        a : in bits(15 downto 0);
        y : out bits(15 downto 0)
    );
end entity;

architecture rtl of Inc16 is
    component Add16
        port(a,b : in bits(15 downto 0); cin : in bit; sum : out bits(15 downto 0);
            => cout : out bit);
    end component;
    signal zero16 : bits(15 downto 0);
    signal unused_cout : bit;
begin
    zero16 <= X"0000";
    u0: Add16 port map (a => a, b => zero16, cin => '1', sum => y, cout =>
        unused_cout);
end architecture;

```

16.1.20 Sub16

```
-- 16-bit Subtractor
entity Sub16 is
  port(
    a      : in bits(15 downto 0);
    b      : in bits(15 downto 0);
    diff   : out bits(15 downto 0)
  );
end entity;

architecture rtl of Sub16 is
  component Add16
    port(a,b : in bits(15 downto 0); cin : in bit; sum : out bits(15 downto 0);
      ↪ cout : out bit);
  end component;
  component Inv16
    port(a : in bits(15 downto 0); y : out bits(15 downto 0));
  end component;
  signal nb : bits(15 downto 0);
  signal unused_cout : bit;
begin
  u0: Inv16 port map (a => b, y => nb);
  u1: Add16 port map (a => a, b => nb, cin => '1', sum => diff, cout =>
    ↪ unused_cout);
end architecture;
```

16.1.21 ALU

```
-- 16-bit ALU
-- op: 0=AND, 1=OR, 2=ADD, 3=SUB

entity ALU is
  port(
    a      : in bits(15 downto 0);
    b      : in bits(15 downto 0);
    op     : in bits(1 downto 0);
    y      : out bits(15 downto 0);
    zero   : out bit;
    neg    : out bit
  );
end entity;
```

```

architecture rtl of ALU is
  component Add16
    port(a,b : in bits(15 downto 0); cin : in bit; sum : out bits(15 downto 0);
      ↪ cout : out bit);
  end component;
  component And16
    port(a,b : in bits(15 downto 0); y : out bits(15 downto 0));
  end component;
  component Or16
    port(a,b : in bits(15 downto 0); y : out bits(15 downto 0));
  end component;
  component Inv16
    port(a : in bits(15 downto 0); y : out bits(15 downto 0));
  end component;
  component Mux4Way16
    port(a,b,c,d : in bits(15 downto 0); sel : in bits(1 downto 0); y : out
      ↪ bits(15 downto 0));
  end component;
  component Or8Way
    port(a : in bits(7 downto 0); y : out bit);
  end component;
  signal r_and, r_or, r_add, r_sub, nb, result : bits(15 downto 0);
  signal unused_cout1, unused_cout2 : bit;
  signal or_lo, or_hi : bit;
begin
  -- Compute all operations
  u_and: And16 port map (a => a, b => b, y => r_and);
  u_or: Or16 port map (a => a, b => b, y => r_or);
  u_add: Add16 port map (a => a, b => b, cin => '0', sum => r_add, cout =>
    ↪ unused_cout1);

  -- SUB: a - b = a + (~b) + 1
  u_inv: Inv16 port map (a => b, y => nb);
  u_sub: Add16 port map (a => a, b => nb, cin => '1', sum => r_sub, cout =>
    ↪ unused_cout2);

  -- Select result based on op
  u_mux: Mux4Way16 port map (a => r_and, b => r_or, c => r_add, d => r_sub, sel
    ↪ => op, y => result);

  -- Output result
  y <= result;

  -- Zero flag: result == 0
  u_or_lo: Or8Way port map (a => result(7 downto 0), y => or_lo);
  u_or_hi: Or8Way port map (a => result(15 downto 8), y => or_hi);
  zero <= not (or_lo or or_hi);

  -- Negative flag: MSB of result

```

```
    neg <= result(15);  
end architecture;
```

16.1.22 DFF1

```
-- D Flip-Flop  
-- Uses the built-in dff primitive  
  
entity DFF1 is  
    port(  
        clk : in bit;  
        d   : in bit;  
        q   : out bit  
    );  
end entity;  
  
architecture rtl of DFF1 is  
    component dff  
        port(clk : in bit; d : in bit; q : out bit);  
    end component;  
begin  
    u0: dff port map (clk => clk, d => d, q => q);  
end architecture;
```

16.1.23 BitReg

```
-- 1-bit Register  
entity BitReg is  
    port(  
        clk : in bit;  
        d   : in bit;  
        load : in bit;  
        q   : out bit  
    );  
end entity;  
  
architecture rtl of BitReg is  
    component dff  
        port(clk : in bit; d : in bit; q : out bit);  
    end component;
```

```

end component;
component Mux
  port(a,b : in bit; sel : in bit; y : out bit);
end component;
signal mux_out, q_int : bit;
begin
  u_mux: Mux port map (a => q_int, b => d, sel => load, y => mux_out);
  u_dff: dff port map (clk => clk, d => mux_out, q => q_int);
  q <= q_int;
end architecture;

```

16.1.24 Register16

```

-- 16-bit Register
entity Register16 is
  port(
    clk  : in bit;
    d    : in bits(15 downto 0);
    load : in bit;
    q    : out bits(15 downto 0)
  );
end entity;

architecture rtl of Register16 is
  component BitReg
    port(clk : in bit; d : in bit; load : in bit; q : out bit);
  end component;
begin
  u0: BitReg port map (clk => clk, d => d(0), load => load, q => q(0));
  u1: BitReg port map (clk => clk, d => d(1), load => load, q => q(1));
  u2: BitReg port map (clk => clk, d => d(2), load => load, q => q(2));
  u3: BitReg port map (clk => clk, d => d(3), load => load, q => q(3));
  u4: BitReg port map (clk => clk, d => d(4), load => load, q => q(4));
  u5: BitReg port map (clk => clk, d => d(5), load => load, q => q(5));
  u6: BitReg port map (clk => clk, d => d(6), load => load, q => q(6));
  u7: BitReg port map (clk => clk, d => d(7), load => load, q => q(7));
  u8: BitReg port map (clk => clk, d => d(8), load => load, q => q(8));
  u9: BitReg port map (clk => clk, d => d(9), load => load, q => q(9));
  u10: BitReg port map (clk => clk, d => d(10), load => load, q => q(10));
  u11: BitReg port map (clk => clk, d => d(11), load => load, q => q(11));
  u12: BitReg port map (clk => clk, d => d(12), load => load, q => q(12));
  u13: BitReg port map (clk => clk, d => d(13), load => load, q => q(13));
  u14: BitReg port map (clk => clk, d => d(14), load => load, q => q(14));
  u15: BitReg port map (clk => clk, d => d(15), load => load, q => q(15));

```

```
end architecture;
```

16.1.25 PC

```
-- Program Counter
entity PC is
  port(
    clk    : in bit;
    d      : in bits(15 downto 0);
    inc    : in bit;
    load   : in bit;
    reset  : in bit;
    q      : out bits(15 downto 0)
  );
end entity;

architecture rtl of PC is
  component Register16
    port(clk : in bit; d : in bits(15 downto 0); load : in bit; q : out bits(15
      ↪ downto 0));
  end component;
  component Inc16
    port(a : in bits(15 downto 0); y : out bits(15 downto 0));
  end component;
  component Mux16
    port(a,b : in bits(15 downto 0); sel : in bit; y : out bits(15 downto 0));
  end component;
  signal q_int, inc_out, mux1_out, mux2_out, mux3_out : bits(15 downto 0);
  signal zero16 : bits(15 downto 0);
  signal do_load : bit;
begin
  zero16 <= X"0000";

  -- Increment current value
  u_inc: Inc16 port map (a => q_int, y => inc_out);

  -- Priority: reset > load > inc
  -- First mux: inc or hold
  u_mux1: Mux16 port map (a => q_int, b => inc_out, sel => inc, y => mux1_out);
  -- Second mux: load overrides
  u_mux2: Mux16 port map (a => mux1_out, b => d, sel => load, y => mux2_out);
  -- Third mux: reset overrides all
  u_mux3: Mux16 port map (a => mux2_out, b => zero16, sel => reset, y =>
    ↪ mux3_out);
```

```

-- Always load the register
do_load <= inc or load or reset;
u_reg: Register16 port map (clk => clk, d => mux3_out, load => do_load, q =>
    ↪ q_int);

q <= q_int;
end architecture;

```

16.1.26 RAM8

```

-- 8-word RAM
entity RAM8 is
    port(
        clk : in bit;
        din : in bits(15 downto 0);
        addr : in bits(2 downto 0);
        we : in bit;
        dout : out bits(15 downto 0)
    );
end entity;

architecture rtl of RAM8 is
    component Register16
        port(clk : in bit; d : in bits(15 downto 0); load : in bit; q : out bits(15
            ↪ downto 0));
    end component;
    component DMux8Way
        port(x : in bit; sel : in bits(2 downto 0); a,b,c,d,e,f,g,h : out bit);
    end component;
    component Mux8Way16
        port(a,b,c,d,e,f,g,h : in bits(15 downto 0); sel : in bits(2 downto 0); y :
            ↪ out bits(15 downto 0));
    end component;
    signal load0,load1,load2,load3,load4,load5,load6,load7 : bit;
    signal r0,r1,r2,r3,r4,r5,r6,r7 : bits(15 downto 0);
begin
    u_dmux: DMux8Way port map (x => we, sel => addr, a => load0, b => load1, c =>
        ↪ load2, d => load3, e => load4, f => load5, g => load6, h => load7);

    u_r0: Register16 port map (clk => clk, d => din, load => load0, q => r0);
    u_r1: Register16 port map (clk => clk, d => din, load => load1, q => r1);
    u_r2: Register16 port map (clk => clk, d => din, load => load2, q => r2);
    u_r3: Register16 port map (clk => clk, d => din, load => load3, q => r3);

```

```

u_r4: Register16 port map (clk => clk, d => din, load => load4, q => r4);
u_r5: Register16 port map (clk => clk, d => din, load => load5, q => r5);
u_r6: Register16 port map (clk => clk, d => din, load => load6, q => r6);
u_r7: Register16 port map (clk => clk, d => din, load => load7, q => r7);

u_mux: Mux8Way16 port map (a => r0, b => r1, c => r2, d => r3, e => r4, f =>
    ↪ r5, g => r6, h => r7, sel => addr, y => dout);
end architecture;

```

16.1.27 RAM64

```

-- 64-word RAM
entity RAM64 is
    port(
        clk : in bit;
        din : in bits(15 downto 0);
        addr : in bits(5 downto 0);
        we : in bit;
        dout : out bits(15 downto 0)
    );
end entity;

architecture rtl of RAM64 is
    component RAM8
        port(clk : in bit; din : in bits(15 downto 0); addr : in bits(2 downto 0);
            ↪ we : in bit; dout : out bits(15 downto 0));
    end component;
    component DMux8Way
        port(x : in bit; sel : in bits(2 downto 0); a,b,c,d,e,f,g,h : out bit);
    end component;
    component Mux8Way16
        port(a,b,c,d,e,f,g,h : in bits(15 downto 0); sel : in bits(2 downto 0); y :
            ↪ out bits(15 downto 0));
    end component;
    signal we0,we1,we2,we3,we4,we5,we6,we7 : bit;
    signal r0,r1,r2,r3,r4,r5,r6,r7 : bits(15 downto 0);
begin
    u_dmux: DMux8Way port map (x => we, sel => addr(5 downto 3), a => we0, b =>
        ↪ we1, c => we2, d => we3, e => we4, f => we5, g => we6, h => we7);

    u_ram0: RAM8 port map (clk => clk, din => din, addr => addr(2 downto 0), we =>
        ↪ we0, dout => r0);
    u_ram1: RAM8 port map (clk => clk, din => din, addr => addr(2 downto 0), we =>
        ↪ we1, dout => r1);

```



```

u_ram2: RAM8 port map (clk => clk, din => din, addr => addr(2 downto 0), we =>
    ↪ we2, dout => r2);
u_ram3: RAM8 port map (clk => clk, din => din, addr => addr(2 downto 0), we =>
    ↪ we3, dout => r3);
u_ram4: RAM8 port map (clk => clk, din => din, addr => addr(2 downto 0), we =>
    ↪ we4, dout => r4);
u_ram5: RAM8 port map (clk => clk, din => din, addr => addr(2 downto 0), we =>
    ↪ we5, dout => r5);
u_ram6: RAM8 port map (clk => clk, din => din, addr => addr(2 downto 0), we =>
    ↪ we6, dout => r6);
u_ram7: RAM8 port map (clk => clk, din => din, addr => addr(2 downto 0), we =>
    ↪ we7, dout => r7);

u_mux: Mux8Way16 port map (a => r0, b => r1, c => r2, d => r3, e => r4, f =>
    ↪ r5, g => r6, h => r7, sel => addr(5 downto 3), y => dout);
end architecture;

```

16.1.28 RegFile

```

-- 16-Register File using RAM primitive
entity RegFile is
    port(
        clk    : in bit;
        we     : in bit;
        waddr  : in bits(3 downto 0);
        wdata  : in bits(15 downto 0);
        raddr1 : in bits(3 downto 0);
        raddr2 : in bits(3 downto 0);
        rdata1 : out bits(15 downto 0);
        rdata2 : out bits(15 downto 0)
    );
end entity;

architecture rtl of RegFile is
    component ram
        port(clk : in bit; we : in bit; addr : in bits(3 downto 0);
            din : in bits(15 downto 0); dout : out bits(15 downto 0));
    end component;
begin
    -- Use two RAM instances for dual read ports
    u_ram1: ram port map (clk => clk, we => we, addr => waddr, din => wdata, dout
        ↪ => rdata1);
    u_ram2: ram port map (clk => clk, we => we, addr => waddr, din => wdata, dout
        ↪ => rdata2);

```

```
-- Note: Simplified - real implementation would need proper read port
-- addressing
end architecture;
```

16.1.29 Decoder

```
-- Instruction Decoder
-- Decodes opcode into control signals
-- Opcodes: 0000=ALU, 0100=LOAD, 0101=STORE, 1000=BRANCH

entity Decoder is
  port(
    opcode      : in bits(3 downto 0);
    alu_op      : out bits(1 downto 0);
    reg_write   : out bit;
    mem_read    : out bit;
    mem_write   : out bit;
    branch      : out bit
  );
end entity;

architecture rtl of Decoder is
  component Inv
    port(a : in bit; y : out bit);
  end component;
  component And2
    port(a, b : in bit; y : out bit);
  end component;
  signal not_op3, not_op2, not_op1, not_op0 : bit;
  signal is_alu, is_load, is_store, is_branch : bit;
begin
  -- Invert opcode bits
  inv3: Inv port map (a => opcode(3), y => not_op3);
  inv2: Inv port map (a => opcode(2), y => not_op2);
  inv1: Inv port map (a => opcode(1), y => not_op1);
  inv0: Inv port map (a => opcode(0), y => not_op0);

  -- Decode: ALU = 0000
  alu_a: And2 port map (a => not_op3, b => not_op2, y => is_alu);

  -- Decode: LOAD = 0100
  ld_a: And2 port map (a => not_op3, b => opcode(2), y => is_load);

  -- Decode: STORE = 0101
```

```

st_a: And2 port map (a => is_load, b => opcode(0), y => is_store);

-- Decode: BRANCH = 1xxx
br_a: Inv port map (a => not_op3, y => is_branch);

-- Control signals - pass through lower opcode bits for ALU operation
alu_op <= opcode(1 downto 0);
reg_write <= is_alu;
mem_read <= is_load;
mem_write <= is_store;
branch <= is_branch;
end architecture;

```

16.1.30 CondCheck

```

-- Condition Checker
-- Checks ALU flags against condition code
-- cond: 0000=EQ (zero), 0001=NE (!zero), 0010=LT (neg), 0011=GE (!neg)

entity CondCheck is
  port(
    cond : in bits(3 downto 0);
    zero : in bit;
    neg : in bit;
    carry: in bit;
    ovf : in bit;
    take : out bit
  );
end entity;

architecture rtl of CondCheck is
  component Inv
    port(a : in bit; y : out bit);
  end component;
  component Mux
    port(a, b : in bit; sel : in bit; y : out bit);
  end component;
  signal not_zero, not_neg : bit;
  signal eq_result, ne_result, lt_result, ge_result : bit;
  signal sel0, sel1 : bit;
begin
  -- Invert flags for NE and GE conditions
  inv_z: Inv port map (a => zero, y => not_zero);
  inv_n: Inv port map (a => neg, y => not_neg);

```

```

-- Condition results
eq_result <= zero;      -- EQ: zero=1
ne_result <= not_zero;  -- NE: zero=0
lt_result <= neg;       -- LT: neg=1
ge_result <= not_neg;   -- GE: neg=0

-- 4-way mux using cond(1:0)
mux0: Mux port map (a => eq_result, b => ne_result, sel => cond(0), y => sel0);
mux1: Mux port map (a => lt_result, b => ge_result, sel => cond(0), y => sel1);
mux2: Mux port map (a => sel0, b => sel1, sel => cond(1), y => take);
end architecture;

```

16.1.31 Control

```

-- Control Unit
-- Generates all control signals

entity Control is
    port(
        clk      : in bit;
        opcode    : in bits(3 downto 0);
        cond      : in bits(3 downto 0);
        zero      : in bit;
        neg       : in bit;
        alu_op     : out bits(1 downto 0);
        reg_write  : out bit;
        mem_read   : out bit;
        mem_write  : out bit;
        pc_src     : out bit
    );
end entity;

architecture rtl of Control is
    component Decoder
        port(opcode : in bits(3 downto 0); alu_op : out bits(1 downto 0);
              reg_write, mem_read, mem_write, branch : out bit);
    end component;
    component CondCheck
        port(cond : in bits(3 downto 0); zero, neg, carry, ovf : in bit; take : out
              bit);
    end component;
    component And2
        port(a, b : in bit; y : out bit);
    end component;

```

```

    end component;
    signal branch_sig, cond_take : bit;
begin
    -- Decoder generates base control signals
    u_dec: Decoder port map (
        opcode => opcode,
        alu_op => alu_op,
        reg_write => reg_write,
        mem_read => mem_read,
        mem_write => mem_write,
        branch => branch_sig
    );

    -- CondCheck evaluates branch condition
    u_cond: CondCheck port map (
        cond => cond,
        zero => zero,
        neg => neg,
        carry => '0',
        ovf => '0',
        take => cond_take
    );

    -- pc_src = branch AND condition_met
    u_and: And2 port map (a => branch_sig, b => cond_take, y => pc_src);
end architecture;

```

16.1.32 CPU

```

-- A32-Lite CPU
-- Simple 16-bit RISC processor
-- Instruction format: [15:12]=opcode, [11:8]=rd, [7:4]=rs1, [3:0]=rs2/imm

entity CPU is
    port(
        clk      : in bit;
        reset    : in bit;
        instr     : in bits(15 downto 0);
        mem_in    : in bits(15 downto 0);
        mem_out   : out bits(15 downto 0);
        mem_addr  : out bits(15 downto 0);
        mem_we    : out bit;
        pc_out    : out bits(15 downto 0)
    );

```

```

end entity;

architecture rtl of CPU is
  component RegFile
    port(clk,we : in bit; waddr,raddr1,raddr2 : in bits(3 downto 0);
         wdata : in bits(15 downto 0); rdata1,rdata2 : out bits(15 downto 0));
  end component;
  component ALU
    port(a,b : in bits(15 downto 0); op : in bits(1 downto 0);
         y : out bits(15 downto 0); zero,neg : out bit);
  end component;
  component PC
    port(clk : in bit; d : in bits(15 downto 0); inc,load,reset : in bit; q :
         out bits(15 downto 0));
  end component;
  component Control
    port(clk : in bit; opcode,cond : in bits(3 downto 0); zero,neg : in bit;
         alu_op : out bits(1 downto 0); reg_write,mem_read,mem_write,pc_src :
         out bit);
  end component;
  component Mux16
    port(a,b : in bits(15 downto 0); sel : in bit; y : out bits(15 downto 0));
  end component;

  -- Instruction decode
  signal opcode, rd, rs1, rs2 : bits(3 downto 0);
  -- Control signals
  signal alu_op : bits(1 downto 0);
  signal reg_write, mem_rd, mem_wr, pc_src : bit;
  -- Datapath signals
  signal pc_val, branch_target : bits(15 downto 0);
  signal reg_data1, reg_data2, alu_result : bits(15 downto 0);
  signal write_data : bits(15 downto 0);
  signal zero_flag, neg_flag : bit;
  signal not_reset : bit;
  component Inv
    port(a : in bit; y : out bit);
  end component;
begin
  -- Instruction decode using slices
  opcode <= instr(15 downto 12);
  rd <= instr(11 downto 8);
  rs1 <= instr(7 downto 4);
  rs2 <= instr(3 downto 0);

  -- Control unit
  u_ctrl: Control port map (
    clk => clk, opcode => opcode, cond => rs2,
    zero => zero_flag, neg => neg_flag,

```

```

    alu_op => alu_op, reg_write => reg_write,
    mem_read => mem_rd, mem_write => mem_wr, pc_src => pc_src
);

-- Register file
u_regs: RegFile port map (
    clk => clk, we => reg_write,
    waddr => rd, raddr1 => rs1, raddr2 => rs2,
    wdata => write_data, rdata1 => reg_data1, rdata2 => reg_data2
);

-- ALU
u_alu: ALU port map (
    a => reg_data1, b => reg_data2, op => alu_op,
    y => alu_result, zero => zero_flag, neg => neg_flag
);

-- Write back mux (ALU result or memory)
u_wb_mux: Mux16 port map (
    a => alu_result, b => mem_in, sel => mem_rd, y => write_data
);

-- Program counter
inv_reset: Inv port map (a => reset, y => not_reset);
branch_target <= reg_data1;
u_pc: PC port map (
    clk => clk, d => branch_target,
    inc => not_reset, load => pc_src, reset => reset, q => pc_val
);

-- Outputs
pc_out <= pc_val;
mem_addr <= alu_result;
mem_out <= reg_data2;
mem_we <= mem_wr;
end architecture;

```

16.1.33 IF_ID_Reg

```

-- IF/ID Pipeline Register

entity IF_ID_Reg is
    port(
        clk : in bit;

```

```

    reset : in bit;
    stall : in bit;
    flush : in bit;
    if_instr : in bits(31 downto 0);
    if_pc_plus4 : in bits(31 downto 0);
    id_instr : out bits(31 downto 0);
    id_pc_plus4 : out bits(31 downto 0)
);
end entity;

architecture rtl of IF_ID_Reg is
    signal instr_reg : bits(31 downto 0);
    signal pc_plus4_reg : bits(31 downto 0);
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if (reset = '1') or (flush = '1') then
                instr_reg <= x"E0000000"; -- NOP
                pc_plus4_reg <= x"00000000";
            elsif stall = '0' then
                instr_reg <= if_instr;
                pc_plus4_reg <= if_pc_plus4;
            end if;
        end if;
    end process;

    id_instr <= instr_reg;
    id_pc_plus4 <= pc_plus4_reg;
end architecture;

```

16.1.34 HazardDetect

```

-- Hazard Detection Unit

entity HazardDetect is
    port(
        id_rn : in bits(3 downto 0);
        id_rm : in bits(3 downto 0);
        id_rn_used : in bit;
        id_rm_used : in bit;
        ex_rd : in bits(3 downto 0);
        ex_mem_read : in bit;
        stall : out bit
    );
end entity;

```



```

    );
end entity;

architecture rtl of HazardDetect is
    signal rn_hazard : bit;
    signal rm_hazard : bit;
begin
    -- Hazard if: load in EX AND register used in ID AND same register
    rn_hazard <= ex_mem_read and id_rn_used and (id_rn = ex_rd);
    rm_hazard <= ex_mem_read and id_rm_used and (id_rm = ex_rd);
    stall <= rn_hazard or rm_hazard;
end architecture;

```

16.1.35 ForwardUnit

```

-- Forwarding Unit

entity ForwardUnit is
    port(
        ex_rn : in bits(3 downto 0);
        ex_rm : in bits(3 downto 0);
        mem_rd : in bits(3 downto 0);
        mem_reg_write : in bit;
        wb_rd : in bits(3 downto 0);
        wb_reg_write : in bit;
        forward_a : out bits(1 downto 0);
        forward_b : out bits(1 downto 0)
    );
end entity;

architecture rtl of ForwardUnit is
    signal mem_fwd_a : bit;
    signal wb_fwd_a : bit;
    signal mem_fwd_b : bit;
    signal wb_fwd_b : bit;
begin
    -- Detect forwarding conditions
    mem_fwd_a <= mem_reg_write and (mem_rd = ex_rn);
    wb_fwd_a <= wb_reg_write and (wb_rd = ex_rn) and (not mem_fwd_a);
    mem_fwd_b <= mem_reg_write and (mem_rd = ex_rm);
    wb_fwd_b <= wb_reg_write and (wb_rd = ex_rm) and (not mem_fwd_b);

    -- Encode output: 00=none, 01=MEM, 10=WB
    forward_a <= wb_fwd_a & mem_fwd_a;

```

```
forward_b <= wb_fwd_b & mem_fwd_b;  
end architecture;
```

16.1.36 CPU_Pipeline

```
-- 5-Stage Pipelined CPU  
-- See hdl_lib/05_cpu/CPU_Pipeline.hdl for full implementation  
-- This exercise is a capstone project  
  
entity CPU_Pipeline is  
  port(  
    clk : in bit;  
    reset : in bit;  
    instr_addr : out bits(31 downto 0);  
    instr_data : in bits(31 downto 0);  
    mem_addr : out bits(31 downto 0);  
    mem_wdata : out bits(31 downto 0);  
    mem_rdata : in bits(31 downto 0);  
    mem_read : out bit;  
    mem_write : out bit;  
    halted : out bit  
  );  
end entity;  
  
architecture rtl of CPU_Pipeline is  
begin  
  -- Implementation uses IF_ID_Reg, HazardDetect, ForwardUnit  
  -- and additional pipeline registers EX_MEM, MEM_WB  
  -- See hdl_lib/05_cpu/CPU_Pipeline.hdl for complete code  
  instr_addr <= x"00000000";  
  mem_addr <= x"00000000";  
  mem_wdata <= x"00000000";  
  mem_read <= '0';  
  mem_write <= '0';  
  halted <= '0';  
end architecture;
```

16.1.37 Projet 7 : Cache L1

16.1.38 CacheLine

```
-- Cache Line

entity CacheLine is
  port(
    clk : in bit;
    write_enable : in bit;
    write_tag : in bits(19 downto 0);
    write_data : in bits(127 downto 0);
    write_word : in bits(31 downto 0);
    write_word_sel : in bits(1 downto 0);
    write_word_en : in bit;
    set_dirty : in bit;
    clear_dirty : in bit;
    invalidate : in bit;
    valid : out bit;
    dirty : out bit;
    tag : out bits(19 downto 0);
    data : out bits(127 downto 0)
  );
end entity;

architecture rtl of CacheLine is
  signal valid_reg : bit;
  signal dirty_reg : bit;
  signal tag_reg : bits(19 downto 0);
  signal data_reg : bits(127 downto 0);
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if invalidate = '1' then
        valid_reg <= '0';
        dirty_reg <= '0';
      elsif write_enable = '1' then
        valid_reg <= '1';
        tag_reg <= write_tag;
        data_reg <= write_data;
        dirty_reg <= '0';
      elsif write_word_en = '1' then
        if write_word_sel = b"00" then
          data_reg(31 downto 0) <= write_word;
        elsif write_word_sel = b"01" then
          data_reg(63 downto 32) <= write_word;
        elsif write_word_sel = b"10" then
```

```
        data_reg(95 downto 64) <= write_word;
    else
        data_reg(127 downto 96) <= write_word;
    end if;
end if;

if set_dirty = '1' then
    dirty_reg <= '1';
elsif clear_dirty = '1' then
    dirty_reg <= '0';
end if;
end if;
end process;

valid <= valid_reg;
dirty <= dirty_reg;
tag <= tag_reg;
data <= data_reg;
end architecture;
```

16.1.39 TagCompare

```
-- Tag Comparator

entity TagCompare is
    port(
        valid : in bit;
        addr_tag : in bits(19 downto 0);
        stored_tag : in bits(19 downto 0);
        hit : out bit
    );
end entity;

architecture rtl of TagCompare is
begin
    -- Hit when valid AND all 20 tag bits match
    hit <= valid and (addr_tag = stored_tag);
end architecture;
```

16.1.40 WordSelect

```
-- Word Selector

entity WordSelect is
  port(
    line_data : in bits(127 downto 0);
    word_sel  : in bits(1 downto 0);
    word_out  : out bits(31 downto 0)
  );
end entity;

architecture rtl of WordSelect is
  component Mux4Way16
    port(a,b,c,d : in bits(15 downto 0); sel : in bits(1 downto 0); y : out
      bits(15 downto 0));
  end component;
begin
  -- Use two 16-bit 4-way muxes for lower and upper halves
  lo: Mux4Way16 port map(
    a => line_data(15 downto 0),
    b => line_data(47 downto 32),
    c => line_data(79 downto 64),
    d => line_data(111 downto 96),
    sel => word_sel,
    y => word_out(15 downto 0)
  );
  hi: Mux4Way16 port map(
    a => line_data(31 downto 16),
    b => line_data(63 downto 48),
    c => line_data(95 downto 80),
    d => line_data(127 downto 112),
    sel => word_sel,
    y => word_out(31 downto 16)
  );
end architecture;
```

16.1.41 CacheController

```
-- Cache Controller FSM

entity CacheController is
  port(
```

```

    clk : in bit;
    reset : in bit;
    cpu_read : in bit;
    cpu_write : in bit;
    cache_hit : in bit;
    mem_ready : in bit;
    state : out bits(1 downto 0);
    cpu_ready : out bit;
    mem_read : out bit;
    mem_write : out bit;
    fill_line : out bit
);
end entity;

architecture rtl of CacheController is
    signal state_reg : bits(1 downto 0);
    signal pending_write : bit;
    signal fill_line_reg : bit;
    signal is_idle, is_fetch, is_wb : bit;
    signal miss, req : bit;
begin
    -- Precompute conditions
    is_idle <= (state_reg = 0b00);
    is_fetch <= (state_reg = 0b01);
    is_wb <= (state_reg = 0b10);
    miss <= not cache_hit;
    req <= cpu_read or cpu_write;

    process(clk)
    begin
        if rising_edge(clk) then
            if reset = '1' then
                state_reg <= 0b00;
                pending_write <= '0';
                fill_line_reg <= '0';
            elsif (is_idle = '1') and (req = '1') and (miss = '1') then
                state_reg <= 0b01;
                pending_write <= cpu_write;
                fill_line_reg <= '0';
            elsif (is_idle = '1') and (cpu_write = '1') and (cache_hit = '1') then
                state_reg <= 0b10;
                fill_line_reg <= '0';
            elsif (is_fetch = '1') and (mem_ready = '1') and (pending_write = '1') then
                state_reg <= 0b10;
                fill_line_reg <= '1';
            elsif (is_fetch = '1') and (mem_ready = '1') and (pending_write = '0') then
                state_reg <= 0b00;
                fill_line_reg <= '1';
            elsif (is_wb = '1') and (mem_ready = '1') then

```

```
        state_reg <= 0b00;
        pending_write <= '0';
        fill_line_reg <= '0';
    else
        fill_line_reg <= '0';
    end if;
end if;
end process;

state <= state_reg;
cpu_ready <= (is_idle and cache_hit) or (is_fetch and mem_ready and (not
↪ pending_write)) or (is_wb and mem_ready);
mem_read <= is_fetch;
mem_write <= is_wb;
fill_line <= fill_line_reg;
end architecture;
```

16.2 B. Solutions Assembleur A32

16.2.1 Hello World

```
; Hello World - Solution

.text
.global _start
_start:
    MOV R0, #42
    HALT
```

16.2.2 Addition

```
; Addition - Solution

.text
.global _start
_start:
    MOV R0, #15
    ADD R0, R0, #27
    HALT
```

16.2.3 Soustraction

```
; Soustraction - Solution

.text
.global _start
_start:
    MOV R0, #100
    SUB R0, R0, #58
    HALT
```

16.2.4 Logique

```
; Logique - Solution

.text
.global _start
_start:
    MOV R1, #0xFF
    MOV R2, #0x0F
    AND R0, R1, R2
    HALT
```

16.2.5 Doubler

```
; Doubler - Solution

.text
.global _start
_start:
    MOV R0, #21
    ADD R0, R0, R0    ; R0 = R0 + R0 = 2 * R0
    HALT
```

16.2.6 Conditions

```
; Conditions - Solution

.text
.global _start
_start:
    MOV R1, #25
    MOV R2, #17

    CMP R1, R2
    B.GT .r1_bigger
    MOV R0, R2
    B .done

.r1_bigger:
    MOV R0, R1

.done:
    HALT
```

16.2.7 Valeur Absolue

```
; Valeur Absolue - Solution

.text
.global _start
_start:
    MOV R1, #0
    SUB R1, R1, #42 ; R1 = -42

    CMP R1, #0
    B.GE .positive
    ; Négatif: R0 = 0 - R1
    MOV R0, #0
    SUB R0, R0, R1
    B .done

.positive:
    MOV R0, R1

.done:
    HALT
```

16.2.8 Boucles

```
; Boucles - Solution

.text
.global _start
_start:
    MOV R0, #0      ; somme = 0
    MOV R1, #1      ; compteur = 1

.loop:
    ADD R0, R0, R1   ; somme += compteur
    ADD R1, R1, #1   ; compteur++
    CMP R1, #10
    B.LE .loop

    HALT
```

16.2.9 Multiplication

```
; Multiplication - Solution

.text
.global _start
_start:
    MOV R0, #0      ; résultat = 0
    MOV R1, #6      ; multiplicande
    MOV R2, #7      ; compteur (multiplicateur)

.loop:
    ADD R0, R0, R1   ; résultat += multiplicande
    SUB R2, R2, #1   ; compteur--
    CMP R2, #0
    B.GT .loop

    HALT
```

16.2.10 Fibonacci

```
; Fibonacci - Solution

.text
.global _start
_start:
    MOV R0, #1      ; F(1) = 1
    MOV R1, #1      ; F(2) = 1
    MOV R2, #2      ; compteur = 2

.loop:
    ADD R3, R0, R1   ; R3 = F(n-2) + F(n-1)
    MOV R0, R1       ; F(n-2) = ancien F(n-1)
    MOV R1, R3       ; F(n-1) = nouveau F(n)
    ADD R2, R2, #1   ; compteur++
    CMP R2, #10
    B.LT .loop

    MOV R0, R1       ; résultat dans R0
    HALT
```

16.2.11 Tableaux

```
; Tableaux - Solution

.data
data:
    .word 10
    .word 20
    .word 30
    .word 40
    .word 50

.text
.global _start
_start:
    MOV R0, #0       ; somme = 0
    LDR R1, =data     ; adresse du tableau
    MOV R2, #5       ; compteur = 5

.loop:
    LDR R3, [R1]      ; charge élément
```

```
ADD R0, R0, R3      ; somme += élément
ADD R1, R1, #4      ; adresse suivante
SUB R2, R2, #1      ; compteur--
CMP R2, #0
B.GT .loop

HALT
```

16.2.12 Maximum Tableau

```
; Maximum Tableau - Solution

.data
data:
    .word 12
    .word 45
    .word 7
    .word 89
    .word 23

.text
.global _start
_start:
    LDR R1, =data      ; adresse du tableau
    LDR R0, [R1]       ; max = premier élément
    ADD R1, R1, #4     ; passer au suivant
    MOV R2, #4         ; compteur = 4 (reste)

.loop:
    LDR R3, [R1]       ; charge élément
    CMP R3, R0         ; compare avec max
    B.LE .skip
    MOV R0, R3         ; nouveau max

.skip:
    ADD R1, R1, #4     ; adresse suivante
    SUB R2, R2, #1     ; compteur--
    CMP R2, #0
    B.GT .loop

HALT
```

16.2.13 Mémoire

```
; Mémoire - Solution

.data
data:
    .word 0
    .word 0

.text
.global _start
_start:
    MOV R1, #10
    MOV R2, #20

    ; Sauvegarder
    LDR R4, =data
    STR R1, [R4]      ; data[0] = R1
    ADD R4, R4, #4
    STR R2, [R4]      ; data[1] = R2

    ; Effacer
    MOV R1, #0
    MOV R2, #0

    ; Recharger
    LDR R4, =data
    LDR R1, [R4]      ; R1 = data[0]
    ADD R4, R4, #4
    LDR R2, [R4]      ; R2 = data[1]

    ; Calculer
    ADD R0, R1, R2

    HALT
```

16.2.14 Structure Simple

```
; Structure Simple - Solution

.data
; Structure Point { int x; int y; }
point:
```

```
.word 10      ; x = 10 (offset 0)
.word 32      ; y = 32 (offset 4)

.text
.global _start
_start:
    ; Charger l'adresse de la structure
    LDR R1, =point

    ; Charger p.x (offset 0)
    LDR R2, [R1]      ; R2 = p.x = 10

    ; Charger p.y (offset 4)
    LDR R3, [R1, #4]   ; R3 = p.y = 32

    ; Calculer la somme
    ADD R0, R2, R3     ; R0 = 10 + 32 = 42

    HALT
```

16.2.15 Initialiser Structure

```
; Initialiser Structure - Solution

.data
; Structure Point (non initialisée)
point:
    .word 0      ; x (offset 0)
    .word 0      ; y (offset 4)

.text
.global _start
_start:
    LDR R4, =point ; adresse de la structure

    ; Initialiser p.x = 20
    MOV R1, #20
    STR R1, [R4]    ; p.x = 20

    ; Initialiser p.y = 22
    MOV R2, #22
    STR R2, [R4, #4] ; p.y = 22

    ; Relire et additionner
```

```
LDR R1, [R4]      ; R1 = p.x
LDR R2, [R4, #4]   ; R2 = p.y
ADD R0, R1, R2     ; R0 = 42

HALT
```

16.2.16 Structure Rectangle

```
; Structure Rectangle - Solution

.data
; Structure Rectangle
rect:
    .word 5      ; x (offset 0)
    .word 10     ; y (offset 4)
    .word 6      ; width (offset 8)
    .word 7      ; height (offset 12)

.text
.global _start
_start:
    LDR R4, =rect

    ; Charger width et height
    LDR R1, [R4, #8]    ; R1 = width = 6
    LDR R2, [R4, #12]   ; R2 = height = 7

    ; Multiplier par additions: 6 * 7
    MOV R0, #0          ; résultat = 0
.mult:
    CMP R2, #0
    B.EQ .done
    ADD R0, R0, R1       ; résultat += width
    SUB R2, R2, #1       ; height--
    B .mult

.done:
    HALT
```

16.2.17 Tableau de Structures

```
; Tableau de Structures - Solution

.data
; Tableau de 3 Points (8 octets chacun)
points:
    ; points[0]: x=10, y=2
    .word 10
    .word 2
    ; points[1]: x=15, y=5
    .word 15
    .word 5
    ; points[2]: x=8, y=7
    .word 8
    .word 7

.text
.global _start
_start:
    LDR R1, =points      ; adresse du tableau
    MOV R0, #0           ; somme = 0
    MOV R2, #3           ; compteur = 3

.loop:
    CMP R2, #0
    B.EQ .done

    ; Charger x du Point courant
    LDR R3, [R1]         ; R3 = points[i].x
    ADD R0, R0, R3       ; somme += x

    ; Passer au Point suivant (8 octets)
    ADD R1, R1, #8

    ; Décrémenter compteur
    SUB R2, R2, #1
    B .loop

.done:
    HALT
```


16.2.18 Somme x+y Structures

```
; Somme x+y de Structures - Solution

.data
points:
    ; points[0]: x=5, y=3
    .word 5
    .word 3
    ; points[1]: x=10, y=4
    .word 10
    .word 4
    ; points[2]: x=12, y=8
    .word 12
    .word 8

.text
.global _start
_start:
    LDR R1, =points      ; adresse du tableau
    MOV R0, #0           ; somme totale = 0
    MOV R4, #3           ; compteur = 3

.loop:
    CMP R4, #0
    B.EQ .done

    ; Charger x et y du Point courant
    LDR R2, [R1]          ; R2 = x
    LDR R3, [R1, #4]      ; R3 = y

    ; Ajouter x+y à la somme
    ADD R0, R0, R2        ; somme += x
    ADD R0, R0, R3        ; somme += y

    ; Point suivant (8 octets)
    ADD R1, R1, #8
    SUB R4, R4, #1
    B .loop

.done:
    HALT
```

16.2.19 Fonctions

```
; Fonctions - Solution

.text
.global _start
_start:
    MOV R0, #21
    BL double
    HALT

double:
    ADD R0, R0, R0 ; R0 = R0 + R0 = 2 * R0
    MOV PC, LR    ; retour
```

16.2.20 Fonction Add3

```
; Fonction Add3 - Solution

.text
.global _start
_start:
    MOV R0, #10
    MOV R1, #15
    MOV R2, #17
    BL add3
    HALT

add3:
    ADD R0, R0, R1
    ADD R0, R0, R2
    MOV PC, LR
```

16.2.21 Écrire Caractère

```
; Écrire Caractère - Solution

.text
```

```
.global _start
_start:
    MOV R0, #65          ; 'A' = 65
    LDR R1, =0xFFFF0000 ; adresse PUTC
    STRB R0, [R1]        ; écrire le caractère
    HALT
```

16.2.22 Hello String

```
; Hello String - Solution

.text
.global _start
_start:
    LDR R1, =0xFFFF0000 ; adresse PUTC
    MOV R0, #0           ; compteur

    MOV R2, #72          ; 'H'
    STRB R2, [R1]
    ADD R0, R0, #1

    MOV R2, #105         ; 'i'
    STRB R2, [R1]
    ADD R0, R0, #1

    HALT
```

16.2.23 Print Loop

```
; Print Loop - Solution

.text
.global _start
_start:
    LDR R1, =0xFFFF0000 ; adresse PUTC
    MOV R0, #0           ; compteur
    MOV R2, #65          ; caractère courant = 'A'

.loop:
```

```
STRB R2, [R1]      ; écrire caractère
ADD R0, R0, #1     ; compteur++
ADD R2, R2, #1     ; caractère suivant
CMP R2, #69        ; 'E' = 69
B.LT .loop

HALT
```

16.2.24 Pixel

```
; Pixel - Solution

.text
.global _start
_start:
    LDR R1, =0x00400000 ; adresse écran
    MOV R0, #0x80       ; bit 7 = pixel 0
    STRB R0, [R1]       ; écrire le byte
    HALT
```

16.2.25 Ligne Horizontale

```
; Ligne Horizontale - Solution

.text
.global _start
_start:
    LDR R1, =0x00400000 ; adresse écran
    MOV R0, #0xFF       ; 8 pixels allumés
    STRB R0, [R1]       ; écrire le byte
    HALT
```

16.2.26 Ligne Verticale

```
; Ligne Verticale - Solution

.text
.global _start
_start:
    LDR R1, =0x00400000 ; adresse écran
    MOV R2, #0x80       ; bit du pixel (colonne 0)
    MOV R0, #0          ; compteur

.loop:
    STRB R2, [R1]       ; dessiner pixel
    ADD R1, R1, #40     ; ligne suivante (320/8 = 40)
    ADD R0, R0, #1      ; compteur++
    CMP R0, #8
    B.LT .loop

    HALT
```

16.2.27 Rectangle

```
; Rectangle - Solution

.text
.global _start
_start:
    LDR R1, =0x00400000 ; adresse écran
    MOV R2, #0xFF       ; ligne de 8 pixels
    MOV R0, #0          ; compteur de lignes

.loop:
    STRB R2, [R1]       ; dessiner ligne
    ADD R1, R1, #40     ; ligne suivante
    ADD R0, R0, #1      ; compteur++
    CMP R0, #8
    B.LT .loop

    HALT
```

16.2.28 Damier

```
; Damier - Solution

.text
.global _start
_start:
    LDR R1, =0x00400000 ; adresse écran
    MOV R2, #0xAA       ; motif courant
    MOV R0, #0           ; compteur de lignes

.loop:
    STRB R2, [R1]        ; colonne 0
    ADD R4, R1, #1
    EOR R5, R2, #0xFF    ; inverser pour colonne 1
    STRB R5, [R4]        ; colonne 1

    ADD R1, R1, #40      ; ligne suivante
    EOR R2, R2, #0xFF    ; alterner le motif
    ADD R0, R0, #1
    CMP R0, #8
    B.LT .loop

    HALT
```

16.2.29 Lire un Caractère

```
; Lire un Caractère - Solution

.text
.global _start
_start:
    LDR R1, =0x00402600 ; adresse KEYBOARD (temps réel)
    LDR R4, =0xFFFF0000 ; adresse PUTC

    ; Attendre une touche
.wait:
    LDR R2, [R1]         ; lire clavier
    CMP R2, #0
    B.EQ .wait           ; boucler si pas de touche

    ; Convertir ASCII → nombre
    SUB R0, R2, #0x30    ; R0 = chiffre (0-9)
```

```
; Afficher le chiffre tapé (écho)
STR R2, [R4]          ; afficher le caractère

HALT
```

16.2.30 Lire un Nombre à 2 Chiffres

```
; Lire un Nombre à 2 Chiffres - Solution

.text
.global _start
_start:
    LDR R5, =0x00402600 ; adresse KEYBOARD (temps réel)
    LDR R6, =0xFFFF0000 ; adresse PUTC

    ; Attendre premier chiffre
.wait1:
    LDR R1, [R5]
    CMP R1, #0
    B.EQ .wait1

    ; Écho du premier chiffre
    STR R1, [R6]

    ; Convertir en nombre
    SUB R1, R1, #0x30 ; R1 = premier chiffre

    ; Multiplier par 10: x → 2x → 4x → 5x → 10x
    ADD R2, R1, R1 ; R2 = 2x
    ADD R2, R2, R2 ; R2 = 4x
    ADD R2, R2, R1 ; R2 = 5x
    ADD R1, R2, R2 ; R1 = 10x

    ; Attendre relâche de la touche
.release1:
    LDR R2, [R5]
    CMP R2, #0
    B.NE .release1

    ; Attendre deuxième chiffre
.wait2:
    LDR R2, [R5]
    CMP R2, #0
```

```
B.EQ .wait2

; Écho du deuxième chiffre
STR R2, [R6]

; Convertir et ajouter
SUB R2, R2, #0x30 ; R2 = deuxième chiffre
ADD R0, R1, R2    ; R0 = nombre complet

HALT
```

16.2.31 Deviner le Nombre

```
; Deviner le Nombre - Solution

.text
.global _start
_start:
    MOV R7, #55 ; nombre secret: '7' en ASCII (0x37)
    LDR R5, =0x00402600 ; KEYBOARD
    LDR R6, =0xFFFF0000 ; PUTC
    MOV R4, #0 ; dernière touche vue

.game_loop:
    ; Lire clavier
    LDR R0, [R5]

    ; Ignorer si pas de touche ou même touche que avant
    CMP R0, #0
    B.EQ .game_loop
    CMP R0, R4
    B.EQ .game_loop

    ; Nouvelle touche détectée
    MOV R4, R0 ; sauvegarder

    ; Écho de la touche
    STR R0, [R6]

    ; Comparer au secret
    CMP R0, R7
    B.EQ .win
    B.LT .too_small
```



```
    ; Trop grand -> afficher '-'
    MOV R1, #45          ; '-'
    STR R1, [R6]
    MOV R1, #10          ; newline
    STR R1, [R6]
    B .wait_release

.too_small:
    ; Trop petit -> afficher '+'
    MOV R1, #43          ; '+'
    STR R1, [R6]
    MOV R1, #10          ; newline
    STR R1, [R6]

.wait_release:
    ; Attendre relâche (R4 != 0, donc on attend que R0 change)
    LDR R0, [R5]
    CMP R0, R4
    B.EQ .wait_release
    MOV R4, #0           ; reset
    B .game_loop

.win:
    ; Gagné! Afficher '*'
    MOV R1, #42          ; '*'
    STR R1, [R6]
    MOV R1, #10          ; newline
    STR R1, [R6]
    MOV R0, #7           ; résultat dans R0

    HALT
```

16.2.32 Dégradé (Dithering)

```
; Dégradé (Dithering) - Solution

.text
.global _start
_start:
    LDR R1, =0x00400000 ; adresse écran
    MOV R0, #0          ; compteur lignes

.line_loop:
    ; Écrire les 5 motifs de dégradé
```

```
MOV R2, #0x00      ; noir
STRB R2, [R1]

MOV R2, #0x11      ; 12.5% blanc
ADD R3, R1, #1
STRB R2, [R3]

MOV R2, #0x55      ; 50% blanc
ADD R3, R1, #2
STRB R2, [R3]

MOV R2, #0xBB      ; 75% blanc
ADD R3, R1, #3
STRB R2, [R3]

MOV R2, #0xFF      ; blanc
ADD R3, R1, #4
STRB R2, [R3]

; Ligne suivante
ADD R1, R1, #40     ; 40 bytes par ligne
ADD R0, R0, #1
CMP R0, #240
B.LT .line_loop

HALT
```

16.2.33 Dégradé Plein Écran

```
; Dégradé Plein Écran - Solution
; 8 bandes avec motifs croissants

.text
.global _start
_start:
    LDR R1, =0x00400000 ; adresse écran
    MOV R0, #0          ; compteur bandes
    MOV R2, #0x00       ; motif initial (noir)

next_band:
    LDR R3, =1200        ; 30 lignes * 40 bytes

fill_band:
    STRB R2, [R1]
```

```
ADD R1, R1, #1
SUB R3, R3, #1
CMP R3, #0
B.GT fill_band

; Passer au motif suivant (approximation)
; 0x00 -> 0x11 -> 0x22 -> 0x33 -> ...
ADD R2, R2, #0x11
ADD R0, R0, #1
CMP R0, #8
B.LT next_band

HALT
```

16.2.34 Recherche Dichotomique

```
; Recherche Dichotomique - Solution

.text
.global _start
_start:
    MOV R5, #42          ; nombre secret
    MOV R1, #0           ; low
    MOV R2, #100         ; high
    MOV R0, #0           ; compteur d'essais

.loop:
    ADD R0, R0, #1       ; compteur++

    ; mid = (low + high) / 2
    ADD R3, R1, R2
    MOV R4, R3
    ; Division par 2 avec shift (simulé par soustraction successive)
    MOV R3, #0
.div2:
    CMP R4, #2
    B.LT .div_done
    SUB R4, R4, #2
    ADD R3, R3, #1
    B .div2
.div_done:
    ; R3 = mid

    CMP R3, R5
```

```
B.EQ .found
B.LT .too_low

; mid > secret: high = mid - 1
SUB R2, R3, #1
B .loop

.too_low:
; mid < secret: low = mid + 1
ADD R1, R3, #1
B .loop

.found:
HALT
```

16.2.35 Cache: Accés Séquentiel

```
; Accès Séquentiel - Solution
; Parcours cache-friendly d'un tableau

.data
arr:
    .word 10
    .word 20
    .word 30
    .word 40

.text
.global _start
_start:
    MOV R0, #0          ; somme = 0
    LDR R1, =arr         ; adresse du tableau
    MOV R2, #4          ; compteur

.loop:
    CMP R2, #0
    B.EQ .done

    LDR R3, [R1]         ; charger arr[i]
    ADD R0, R0, R3       ; somme += arr[i]
    ADD R1, R1, #4       ; adresse += 4 (accès séquentiel!)
    SUB R2, R2, #1       ; compteur--
    B .loop
```

```
.done:
    HALT
```

Explication: L'accès séquentiel (adresses 0, 4, 8, 12...) est optimal pour le cache car il exploite la localité spatiale : les données proches en mémoire sont chargées ensemble dans une ligne de cache.

16.2.36 Cache: Accés avec Stride

```
; Accès avec Stride - Solution
; Parcours avec sauts de 16 bytes (4 mots)

.data
matrix:
    ; Ligne 0
    .word 1
    .word 2
    .word 3
    .word 4
    ; Ligne 1
    .word 5
    .word 6
    .word 7
    .word 8
    ; Ligne 2
    .word 9
    .word 10
    .word 11
    .word 12
    ; Ligne 3
    .word 13
    .word 14
    .word 15
    .word 16

.text
.global _start
_start:
    MOV R0, #0          ; somme = 0
    LDR R1, =matrix      ; adresse colonne 0
    MOV R2, #4           ; compteur lignes

.loop:
```

```
CMP R2, #0
B.EQ .done

LDR R3, [R1]      ; charger matrix[i][0]
ADD R0, R0, R3     ; somme += valeur
ADD R1, R1, #16    ; stride = 16 bytes (saute une ligne)
SUB R2, R2, #1
B .loop

.done:
HALT
```

Explication: Le stride de 16 bytes signifie qu'on saute 4 éléments à chaque accès (1→5→9→13). C'est moins efficace pour le cache car chaque accès peut nécessiter une nouvelle ligne de cache.

16.2.37 Cache: Reutilisation Registre

```
; Réutilisation des Registres - Solution
; Charger une fois, réutiliser plusieurs fois

.data
a:
    .word 10
b:
    .word 3

.text
.global _start
_start:
    ; Charger une seule fois depuis la mémoire
    LDR R6, =a
    LDR R1, [R6]      ; R1 = a = 10
    LDR R6, =b
    LDR R2, [R6]      ; R2 = b = 3

    ; Calculer avec les registres (pas d'accès mémoire!)
    ADD R3, R1, R2     ; sum = a + b = 13
    SUB R4, R1, R2     ; diff = a - b = 7

    ; Multiplier sum * diff par additions
    MOV R0, #0         ; résultat
    MOV R5, R4         ; compteur = diff = 7
```

```
.mult:
    CMP R5, #0
    B.EQ .done
    ADD R0, R0, R3      ; résultat += sum
    SUB R5, R5, #1
    B .mult

.done:
    HALT
```

Explication: Les registres sont $\sim 100x$ plus rapides que la RAM. En chargeant a et b une seule fois et en gardant les valeurs en registre pour tous les calculs, on évite les accès mémoire coûteux.

16.3 C. Solutions C32

16.3.1 Variables

```
// Variables - Solution

int main() {
    int x = 10;
    int y = 32;
    int result = x + y;
    return result;
}
```

16.3.2 Expressions

```
// Expressions - Solution

int main() {
    int result = (5 + 3) * (10 - 4) / 2;
    return result;
}
```

16.3.3 Modulo

```
// Modulo - Solution

int main() {
    int result = (100 % 7) + (45 % 8);
    return result;
}
```

16.3.4 Incrémentation

```
// Incrémentation - Solution

int main() {
    int x = 5;

    x = x + 3;    // x = 8
    x = x * 2;    // x = 16
    x = x - 1;    // x = 15

    return x;
}
```

16.3.5 Conditions

```
// Conditions - Solution

int main() {
    int a = 25;
    int b = 17;
    int max;

    if (a > b) {
        max = a;
    } else {
        max = b;
    }
}
```



```
    return max;
}
```

16.3.6 Else-If

```
// Else-If - Solution

int main() {
    int score = 75;
    int grade;

    if (score >= 90) {
        grade = 5;
    } else if (score >= 80) {
        grade = 4;
    } else if (score >= 70) {
        grade = 3;
    } else if (score >= 60) {
        grade = 2;
    } else {
        grade = 1;
    }

    return grade;
}
```

16.3.7 Opérateurs Logiques

```
// Opérateurs Logiques - Solution

int main() {
    int x = 15;
    int result;

    if (x >= 10 && x <= 20) {
        result = 1;
    } else {
        result = 0;
    }
}
```

```
    return result;
}
```

16.3.8 Maximum de 3

```
// Maximum de 3 - Solution

int main() {
    int a = 15;
    int b = 42;
    int c = 27;
    int max;

    if (a >= b && a >= c) {
        max = a;
    } else if (b >= c) {
        max = b;
    } else {
        max = c;
    }

    return max;
}
```

16.3.9 Boucle For

```
// Boucle For - Solution

int main() {
    int sum = 0;

    for (int i = 1; i <= 10; i = i + 1) {
        sum = sum + i;
    }

    return sum;
}
```

16.3.10 Boucle While

```
// Boucle While - Solution

int main() {
    int n = 12345;
    int count = 0;

    while (n > 0) {
        count = count + 1;
        n = n / 10;
    }

    return count;
}
```

16.3.11 Boucles Imbriquées

```
// Boucles Imbriquées - Solution

int main() {
    int sum = 0;

    for (int i = 1; i <= 3; i = i + 1) {
        for (int j = 1; j <= 4; j = j + 1) {
            sum = sum + i * j;
        }
    }

    return sum;
}
```

16.3.12 Multiplication

```
// Multiplication - Solution

int main() {
    int a = 7;
```

```
int b = 8;
int result = 0;

for (int i = 0; i < b; i = i + 1) {
    result = result + a;
}

return result;
}
```

16.3.13 Fonctions

```
// Fonctions - Solution

int square(int n) {
    return n * n;
}

int main() {
    return square(7);
}
```

16.3.14 Paramètres Multiples

```
// Paramètres Multiples - Solution

int add3(int a, int b, int c) {
    return a + b + c;
}

int main() {
    return add3(10, 20, 12);
}
```

16.3.15 Valeur Absolue

```
// Valeur Absolue - Solution

int abs(int x) {
    if (x < 0) {
        return -x;
    }
    return x;
}

int main() {
    return abs(-15) + abs(10);
}
```

16.3.16 Min et Max

```
// Min et Max - Solution

int min(int a, int b) {
    if (a < b) {
        return a;
    }
    return b;
}

int max(int a, int b) {
    if (a > b) {
        return a;
    }
    return b;
}

int main() {
    return max(10, 25) - min(10, 25);
}
```

16.3.17 Tableaux

```
// Tableaux - Solution

int main() {
    int arr[5];
    arr[0] = 3;
    arr[1] = 7;
    arr[2] = 2;
    arr[3] = 9;
    arr[4] = 5;

    int sum = 0;

    for (int i = 0; i < 5; i = i + 1) {
        sum = sum + arr[i];
    }

    return sum;
}
```

16.3.18 Maximum Tableau

```
// Maximum Tableau - Solution

int main() {
    int arr[6];
    arr[0] = 12;
    arr[1] = 45;
    arr[2] = 7;
    arr[3] = 23;
    arr[4] = 56;
    arr[5] = 34;

    int max = arr[0];

    for (int i = 1; i < 6; i = i + 1) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }

    return max;
}
```

16.3.19 Compter Éléments

```
// Compter Éléments - Solution

int main() {
    int arr[8];
    arr[0] = 3;
    arr[1] = 8;
    arr[2] = 2;
    arr[3] = 7;
    arr[4] = 4;
    arr[5] = 9;
    arr[6] = 6;
    arr[7] = 1;

    int count = 0;

    for (int i = 0; i < 8; i = i + 1) {
        if (arr[i] % 2 == 0) {
            count = count + 1;
        }
    }

    return count;
}
```

16.3.20 Pointeurs

```
// Pointeurs - Solution

int main() {
    int x = 10;
    int *p = &x;
    *p = 42;
    return x;
}
```

16.3.21 Swap

```
// Swap - Solution

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 10;
    int y = 20;

    swap(&x, &y);

    return x;
}
```

16.3.22 Pointeurs et Tableaux

```
// Pointeurs et Tableaux - Solution

int main() {
    int arr[4];
    arr[0] = 5;
    arr[1] = 10;
    arr[2] = 15;
    arr[3] = 20;

    int sum = 0;
    int *p = arr;

    for (int i = 0; i < 4; i = i + 1) {
        sum = sum + *(p + i);
    }

    return sum;
}
```

16.3.23 Opérations Binaires

```
// Opérations Binaires - Solution

int main() {
    int x = 10;
    int y = 12;

    int result = (x & y) | (x ^ y);
    return result;
}
```

16.3.24 Puissance de 2

```
// Puissance de 2 - Solution

int is_pow2(int n) {
    if (n <= 0) {
        return 0;
    }
    return (n & (n - 1)) == 0;
}

int main() {
    return is_pow2(16) + is_pow2(15) + is_pow2(32);
}
```

16.3.25 Factorielle

```
// Factorielle - Solution

int fact(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * fact(n - 1);
}
```

```
int main() {  
    return fact(5);  
}
```

16.3.26 Fibonacci

```
// Fibonacci - Solution  
  
int fib(int n) {  
    if (n <= 0) {  
        return 0;  
    }  
    if (n == 1) {  
        return 1;  
    }  
    return fib(n - 1) + fib(n - 2);  
}  
  
int main() {  
    return fib(10);  
}
```

16.3.27 Somme Récursive

```
// Somme Récursive - Solution  
  
int sum(int n) {  
    if (n <= 0) {  
        return 0;  
    }  
    return n + sum(n - 1);  
}  
  
int main() {  
    return sum(10);  
}
```

16.3.28 PGCD (Euclide)

```
// PGCD (Euclide) - Solution

int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    return gcd(b, a % b);
}

int main() {
    return gcd(48, 18);
}
```

16.3.29 Puissance

```
// Puissance - Solution

int power(int x, int n) {
    if (n == 0) {
        return 1;
    }
    if (n % 2 == 0) {
        return power(x * x, n / 2);
    }
    return x * power(x, n - 1);
}

int main() {
    return power(2, 10);
}
```

16.3.30 Test Primalité

```
// Test Primalité - Solution

int is_prime(int n) {
```

```
    if (n < 2) {
        return 0;
    }
    for (int i = 2; i * i <= n; i = i + 1) {
        if (n % i == 0) {
            return 0;
        }
    }
    return 1;
}

int main() {
    int count = 0;
    for (int n = 2; n <= 20; n = n + 1) {
        if (is_prime(n)) {
            count = count + 1;
        }
    }
    return count;
}
```

16.3.31 Tri à Bulles

```
// Tri à Bulles - Solution

int main() {
    int arr[5];
    arr[0] = 64;
    arr[1] = 34;
    arr[2] = 25;
    arr[3] = 12;
    arr[4] = 22;

    for (int i = 0; i < 5; i = i + 1) {
        for (int j = 0; j < 4 - i; j = j + 1) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    return arr[0];
}
```

```
}
```

16.3.32 Recherche Binaire

```
// Recherche Binaire - Solution

int binary_search(int *arr, int size, int target) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = (left + right) / 2;
        if (arr[mid] == target) {
            return mid;
        }
        if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}

int main() {
    int arr[10];
    arr[0] = 2;
    arr[1] = 5;
    arr[2] = 8;
    arr[3] = 12;
    arr[4] = 16;
    arr[5] = 23;
    arr[6] = 38;
    arr[7] = 56;
    arr[8] = 72;
    arr[9] = 91;

    return binary_search(arr, 10, 23);
}
```

16.3.33 Inverser Tableau

```
// Inverser Tableau - Solution

int main() {
    int arr[5];
    arr[0] = 1;
    arr[1] = 2;
    arr[2] = 3;
    arr[3] = 4;
    arr[4] = 5;

    int left = 0;
    int right = 4;
    while (left < right) {
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
        left = left + 1;
        right = right - 1;
    }

    int sum = 0;
    for (int i = 0; i < 5; i = i + 1) {
        sum = sum + arr[i] * (i + 1);
    }
    return sum;
}
```

16.3.34 Somme des Chiffres

```
// Somme des Chiffres - Solution

int digit_sum(int n) {
    int sum = 0;
    while (n > 0) {
        sum = sum + (n % 10);
        n = n / 10;
    }
    return sum;
}

int main() {
```

```
    return digit_sum(12345);  
}
```

16.3.35 Nombre Palindrome

```
// Nombre Palindrome - Solution  
  
int is_palindrome(int n) {  
    int original = n;  
    int reversed = 0;  
  
    while (n > 0) {  
        reversed = reversed * 10 + (n % 10);  
        n = n / 10;  
    }  
  
    if (reversed == original) {  
        return 1;  
    }  
    return 0;  
}  
  
int main() {  
    return is_palindrome(12321) + is_palindrome(1221) + is_palindrome(123);  
}
```

16.3.36 Définition Struct

```
// Définition Struct - Solution  
  
struct Point { int x; int y; };  
  
int main() {  
    struct Point p;  
    p.x = 17;  
    p.y = 25;  
    return p.x + p.y;  
}
```

16.3.37 Pointeur Struct

```
// Pointeur Struct - Solution

struct Point { int x; int y; };

int main() {
    struct Point p;
    p.x = 10;
    p.y = 32;

    struct Point *ptr = &p;
    return ptr->x + ptr->y;
}
```

16.3.38 Struct et Fonctions

```
// Struct et Fonctions - Solution

struct Point { int x; int y; };

int distance_sq(struct Point *p) {
    return p->x * p->x + p->y * p->y;
}

int main() {
    struct Point p;
    p.x = 3;
    p.y = 4;
    return distance_sq(&p);
}
```

16.3.39 Structs Imbriquées

```
// Structs Imbriquées - Solution

struct Point { int x; int y; };
struct Rectangle { struct Point corner; int width; int height; };
```



```
int main() {
    struct Rectangle r;
    r.corner.x = 0;
    r.corner.y = 0;
    r.width = 6;
    r.height = 7;

    return r.width * r.height;
}
```

16.3.40 Tableau de Structs

```
// Tableau de Structs - Solution

struct Point { int x; int y; };

int main() {
    struct Point points[3];

    points[0].x = 10;
    points[0].y = 2;
    points[1].x = 15;
    points[1].y = 5;
    points[2].x = 8;
    points[2].y = 2;

    int sum = 0;
    for (int i = 0; i < 3; i = i + 1) {
        sum = sum + points[i].x;
    }
    return sum;
}
```

16.3.41 Sizeof Struct

```
// Sizeof Struct - Solution

struct S1 { int a; };
```

```
struct S2 { int x; int y; char c; };

int main() {
    return sizeof(struct S1) + sizeof(struct S2);
}
```

16.3.42 Parcours en Ligne

```
// Parcours en Ligne - Solution
// Utilise un tableau 1D avec indexation manuelle: arr[i * 4 + j]

int arr[16];

int main() {
    int i;
    int j;
    int sum;

    // Initialiser (parcours en ligne: cache-friendly)
    // Accès: 0, 1, 2, 3, 4, 5, 6, 7... (séquentiel)
    for (i = 0; i < 4; i = i + 1) {
        for (j = 0; j < 4; j = j + 1) {
            arr[i * 4 + j] = i * 4 + j;
        }
    }

    // Calculer la somme (accès séquentiel = cache-friendly)
    sum = 0;
    for (i = 0; i < 4; i = i + 1) {
        for (j = 0; j < 4; j = j + 1) {
            sum = sum + arr[i * 4 + j];
        }
    }

    return sum;
}
```

Explication: Le parcours row-major (i puis j) génère des accès séquentiels en mémoire (indices 0, 1, 2, 3, 4...). C'est optimal pour le cache car une ligne de cache chargée est entièrement utilisée avant de passer à la suivante.

16.3.43 Parcours en Colonne

```
// Parcours en Colonne - Solution
// Même calcul, mais accès non-séquentiels (cache-unfriendly)

int arr[16];

int main() {
    int i;
    int j;
    int sum;

    // Initialiser (row-major comme d'habitude)
    for (i = 0; i < 4; i = i + 1) {
        for (j = 0; j < 4; j = j + 1) {
            arr[i * 4 + j] = i * 4 + j;
        }
    }

    // Somme en colonne: j d'abord, puis i
    // Accès: 0, 4, 8, 12, 1, 5, 9, 13... (sauts de 4!)
    sum = 0;
    for (j = 0; j < 4; j = j + 1) {
        for (i = 0; i < 4; i = i + 1) {
            sum = sum + arr[i * 4 + j];
        }
    }

    return sum;
}
```

Explication: Le parcours column-major (j puis i) génère des accès avec des sauts de 4 positions (indices 0, 4, 8, 12, puis 1, 5, 9, 13...). Chaque accès peut charger une nouvelle ligne de cache, gaspillant les données déjà chargées.

16.3.44 Traitement par Blocs

```
// Traitement par Blocs - Solution
// Technique de blocking pour améliorer la localité

int arr[16];

int main() {
```

```
int i;
int j;
int bi;
int bj;
int sum;

// Initialiser
for (i = 0; i < 4; i = i + 1) {
    for (j = 0; j < 4; j = j + 1) {
        arr[i * 4 + j] = i * 4 + j;
    }
}

sum = 0;

// Parcours par blocs 2x2
// On traite entièrement chaque bloc avant de passer au suivant
for (bi = 0; bi < 4; bi = bi + 2) {
    for (bj = 0; bj < 4; bj = bj + 2) {
        for (i = bi; i < bi + 2; i = i + 1) {
            for (j = bj; j < bj + 2; j = j + 1) {
                sum = sum + arr[i * 4 + j];
            }
        }
    }
}

return sum;
}
```

Explication: Le blocking divise la matrice en petits blocs (2x2 ici) qui tiennent dans le cache. On traite chaque bloc entièrement avant de passer au suivant, maximisant la réutilisation des données chargées.

16.3.45 Localité Temporelle

```
// Localité Temporelle - Solution
// Réutiliser les données pendant qu'elles sont en cache

int arr[4];

int main() {
    int i;
```

```
int sum;
int factor;

arr[0] = 1;
arr[1] = 2;
arr[2] = 3;
arr[3] = 4;

sum = 0;
factor = 3;

// Un seul parcours: chaque élément est lu une seule fois
// et utilisé immédiatement (bonne localité temporelle)
for (i = 0; i < 4; i = i + 1) {
    sum = sum + arr[i] * factor;
}

return sum;
}
```

Explication: La localité temporelle consiste à réutiliser rapidement les données récemment accédées. Ici, on lit chaque élément une seule fois et on effectue tout le calcul immédiatement, plutôt que de faire plusieurs passes sur le tableau.

16.3.46 Écrire un Caractère

```
// Écrire un Caractère - Solution

void putchar(int c) {
    int *port = (int*)0xFFFF0000;
    *port = c;
}

int main() {
    putchar(65); // Affiche 'A'
    return 65;
}
```

16.3.47 Afficher une Chaîne

```
// Afficher une Chaîne - Solution

void putchar(int c) {
    int *port = (int*)0xFFFF0000;
    *port = c;
}

int print(char *s) {
    int count = 0;
    while (*s) {
        putchar(*s);
        s = s + 1;
        count = count + 1;
    }
    return count;
}

int main() {
    return print("HI");
}
```

16.3.48 Afficher un Nombre

```
// Afficher un Nombre - Solution

void putchar(int c) {
    int *port = (int*)0xFFFF0000;
    *port = c;
}

void print_int(int n) {
    char buf[12];
    int i = 0;

    if (n == 0) {
        putchar(48);
        return;
    }

    while (n > 0) {
        buf[i] = 48 + (n % 10);
    }
}
```

```
        n = n / 10;
        i = i + 1;
    }

    while (i > 0) {
        i = i - 1;
        putchar(buf[i]);
    }
}

int main() {
    print_int(42);
    return 42;
}
```

16.3.49 Dessiner un Pixel

```
// Dessiner un Pixel - Solution

int main() {
    char *screen = (char*)0x00400000;
    *screen = 0x80; // 0b10000000 - allume le pixel 0
    return 128;
}
```

16.3.50 Ligne Horizontale

```
// Ligne Horizontale - Solution

int main() {
    char *screen = (char*)0x00400000;

    screen[0] = 0xFF; // 8 premiers pixels
    screen[1] = 0xFF; // 8 pixels suivants

    return 16;
}
```

16.3.51 Dessiner un Rectangle

```
// Dessiner un Rectangle - Solution

int main() {
    char *screen = (char*)0x00400000;

    for (int y = 0; y < 8; y = y + 1) {
        screen[y * 40] = 0xFF; // 40 bytes par ligne
    }

    return 64;
}
```

16.3.52 Crible d

```
// Crible d'Ératosthène - Solution

int main() {
    int is_prime[51];
    int i;
    int j;
    int count;

    for (i = 0; i <= 50; i = i + 1) {
        is_prime[i] = 1;
    }
    is_prime[0] = 0;
    is_prime[1] = 0;

    for (i = 2; i * i <= 50; i = i + 1) {
        if (is_prime[i]) {
            for (j = i * i; j <= 50; j = j + i) {
                is_prime[j] = 0;
            }
        }
    }

    count = 0;
    for (i = 2; i <= 50; i = i + 1) {
        if (is_prime[i]) {
            count = count + 1;
        }
    }
}
```



```
    }  
    return count;  
}
```

16.3.53 Suite de Collatz

```
// Suite de Collatz - Solution  
  
int collatz_length(int n) {  
    int count;  
    count = 1;  
    while (n != 1) {  
        if (n % 2 == 0) {  
            n = n / 2;  
        } else {  
            n = 3 * n + 1;  
        }  
        count = count + 1;  
    }  
    return count;  
}  
  
int main() {  
    return collatz_length(27); // attendu: 112  
}
```

16.3.54 Projet Final

```
// Projet Final - Solution  
  
int sum_divisors(int n) {  
    int sum = 0;  
    for (int i = 1; i < n; i = i + 1) {  
        if (n % i == 0) {  
            sum = sum + i;  
        }  
    }  
    return sum;  
}
```

```
int main() {  
    return sum_divisors(28);  
}
```

16.4 D. Solutions Construction du Compilateur

16.4.1 1.1 Reconnaître un Chiffre

```
int is_digit(char c) {  
    return c >= '0' && c <= '9';  
}  
  
int main() {  
    int score = 0;  
    if (is_digit('0') == 1) score = score + 1;  
    if (is_digit('5') == 1) score = score + 1;  
    if (is_digit('9') == 1) score = score + 1;  
    if (is_digit('a') == 0) score = score + 1;  
    if (is_digit(' ') == 0) score = score + 1;  
    return score;  
}
```

16.4.2 1.2 Lire un Nombre

```
int is_digit(char c) { return c >= '0' && c <= '9'; }  
  
int parse_number(char* s, int* pos) {  
    int result = 0;  
    while (is_digit(s[*pos])) {  
        result = result * 10 + (s[*pos] - '0');  
        *pos = *pos + 1;  
    }  
    return result;  
}  
  
int main() {  
    int score = 0;
```

```
int p;

p = 0;
if (parse_number("42", &p) == 42 && p == 2) score = score + 1;

p = 0;
if (parse_number("123+45", &p) == 123 && p == 3) score = score + 1;

p = 4;
if (parse_number("123+45", &p) == 45 && p == 6) score = score + 1;

p = 0;
if (parse_number("7", &p) == 7 && p == 1) score = score + 1;

return score;
}
```

16.4.3 1.3 Identifier les Tokens

```
int is_digit(char c) { return c >= '0' && c <= '9'; }

int next_token(char* s, int* pos) {
    while (s[*pos] == ' ') *pos = *pos + 1;

    char c = s[*pos];
    if (c == 0) return 0;

    if (is_digit(c)) {
        while (is_digit(s[*pos])) *pos = *pos + 1;
        return 1;
    }

    *pos = *pos + 1;
    if (c == '+') return 2;
    if (c == '-') return 3;
    if (c == '*') return 4;
    if (c == '/') return 5;
    if (c == '(') return 6;
    if (c == ')') return 7;

    return 0;
}

int main() {
```

```
int score = 0;
int p;

p = 0;
if (next_token("42", &p) == 1) score = score + 1;

p = 0;
if (next_token("+", &p) == 2) score = score + 1;

p = 0;
if (next_token("3 + 5", &p) == 1) score = score + 1;
if (next_token("3 + 5", &p) == 2) score = score + 1;
if (next_token("3 + 5", &p) == 1) score = score + 1;
if (next_token("3 + 5", &p) == 0) score = score + 1;

return score;
}
```

16.4.4 2.1 Évaluer a + b

```
int is_digit(char c) { return c >= '0' && c <= '9'; }

int parse_num(char* s, int* p) {
    while (s[*p] == ' ') *p = *p + 1;
    int v = 0;
    while (is_digit(s[*p])) {
        v = v * 10 + (s[*p] - '0');
        *p = *p + 1;
    }
    return v;
}

int eval_simple(char* s) {
    int pos = 0;
    int a = parse_num(s, &pos);

    while (s[pos] == ' ') pos = pos + 1;
    char op = s[pos];
    pos = pos + 1;

    int b = parse_num(s, &pos);

    if (op == '+') return a + b;
    if (op == '-') return a - b;
}
```

```
    if (op == '*') return a * b;
    if (op == '/') return a / b;
    return 0;
}

int main() {
    int score = 0;
    if (eval_simple("3 + 5") == 8) score = score + 1;
    if (eval_simple("10 - 4") == 6) score = score + 1;
    if (eval_simple("6 * 7") == 42) score = score + 1;
    if (eval_simple("20 / 4") == 5) score = score + 1;
    return score;
}
```

16.4.5 2.2 Évaluer a + b + c

```
int is_digit(char c) { return c >= '0' && c <= '9'; }

int parse_num(char* s, int* p) {
    while (s[*p] == ' ') *p = *p + 1;
    int v = 0;
    while (is_digit(s[*p])) { v = v * 10 + (s[*p] - '0'); *p = *p + 1; }
    return v;
}

int is_op(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

int eval_chain(char* s) {
    int pos = 0;
    int result = parse_num(s, &pos);

    while (1) {
        while (s[pos] == ' ') pos = pos + 1;
        char op = s[pos];
        if (!is_op(op)) break;
        pos = pos + 1;
        int b = parse_num(s, &pos);

        if (op == '+') result = result + b;
        else if (op == '-') result = result - b;
        else if (op == '*') result = result * b;
        else if (op == '/') result = result / b;
    }
}
```

```
    }

    return result;
}

int main() {
    int score = 0;
    if (eval_chain("5") == 5) score = score + 1;
    if (eval_chain("3 + 5") == 8) score = score + 1;
    if (eval_chain("1 + 2 + 3") == 6) score = score + 1;
    if (eval_chain("10 - 2 - 3") == 5) score = score + 1;
    if (eval_chain("2 * 3 * 4") == 24) score = score + 1;
    return score;
}
```

16.4.6 2.3 Respecter la Précédence

```
int is_digit(char c) { return c >= '0' && c <= '9'; }
char* input;
int pos;

void skip() { while (input[pos] == ' ') pos = pos + 1; }

int parse_num() {
    skip();
    int v = 0;
    while (is_digit(input[pos])) { v = v * 10 + (input[pos] - '0'); pos = pos + 1; }
    return v;
}

int parse_factor() {
    return parse_num();
}

int parse_term() {
    int left = parse_factor();
    while (1) {
        skip();
        char op = input[pos];
        if (op != '*' && op != '/') break;
        pos = pos + 1;
        int right = parse_factor();
        if (op == '*') left = left * right;
```

```
        else left = left / right;
    }
    return left;
}

int parse_expr() {
    int left = parse_term();
    while (1) {
        skip();
        char op = input[pos];
        if (op != '+' && op != '-') break;
        pos = pos + 1;
        int right = parse_term();
        if (op == '+') left = left + right;
        else left = left - right;
    }
    return left;
}

int eval(char* s) {
    input = s;
    pos = 0;
    return parse_expr();
}

int main() {
    int score = 0;
    if (eval("42") == 42) score = score + 1;
    if (eval("3 + 5") == 8) score = score + 1;
    if (eval("3 * 4") == 12) score = score + 1;
    if (eval("2 + 3 * 4") == 14) score = score + 1;
    if (eval("2 * 3 + 4") == 10) score = score + 1;
    if (eval("10 - 2 * 3") == 4) score = score + 1;
    return score;
}
```

16.4.7 2.4 Gérer les Parenthèses

```
int is_digit(char c) { return c >= '0' && c <= '9'; }
char* input;
int pos;

void skip() { while (input[pos] == ' ') pos = pos + 1; }
```

```
int parse_expr();

int parse_num() {
    skip();
    int v = 0;
    while (is_digit(input[pos])) { v = v * 10 + (input[pos] - '0'); pos = pos + 1; }
    return v;
}

int parse_factor() {
    skip();
    if (input[pos] == '(') {
        pos = pos + 1;
        int v = parse_expr();
        skip();
        pos = pos + 1; // ')'
        return v;
    }
    return parse_num();
}

int parse_term() {
    int left = parse_factor();
    while (1) {
        skip();
        char op = input[pos];
        if (op != '*' && op != '/') break;
        pos = pos + 1;
        int right = parse_factor();
        if (op == '*') left = left * right;
        else left = left / right;
    }
    return left;
}

int parse_expr() {
    int left = parse_term();
    while (1) {
        skip();
        char op = input[pos];
        if (op != '+' && op != '-') break;
        pos = pos + 1;
        int right = parse_term();
        if (op == '+') left = left + right;
        else left = left - right;
    }
    return left;
}
```



```
int eval(char* s) { input = s; pos = 0; return parse_expr(); }

int main() {
    int score = 0;
    if (eval("(5)") == 5) score = score + 1;
    if (eval("(2 + 3)") == 5) score = score + 1;
    if (eval("(2 + 3) * 4") == 20) score = score + 1;
    if (eval("2 * (3 + 4)") == 14) score = score + 1;
    if (eval("(1 + 2) * (3 + 4)") == 21) score = score + 1;
    if (eval("((2))") == 2) score = score + 1;
    return score;
}
```

16.4.8 3.1 Émettre MOV

```
int strlen(char* s) { int i = 0; while (s[i]) i = i + 1; return i; }

void append(char* buf, char* s) {
    int i = strlen(buf);
    int j = 0;
    while (s[j]) { buf[i] = s[j]; i = i + 1; j = j + 1; }
    buf[i] = 0;
}

void append_num(char* buf, int n) {
    char tmp[12];
    int i = 0;
    if (n == 0) { tmp[i] = '0'; i = i + 1; }
    else {
        int rev[12]; int r = 0;
        while (n > 0) { rev[r] = n % 10; r = r + 1; n = n / 10; }
        while (r > 0) { r = r - 1; tmp[i] = '0' + rev[r]; i = i + 1; }
    }
    tmp[i] = 0;
    append(buf, tmp);
}

void emit_mov(char* buf, int reg, int val) {
    append(buf, "MOV R");
    append_num(buf, reg);
    append(buf, ", #");
    append_num(buf, val);
    append(buf, "\\n");
}
```

```
}

int check(char* a, char* b) {
    int i = 0;
    while (a[i] && b[i]) { if (a[i] != b[i]) return 0; i = i + 1; }
    return a[i] == b[i];
}

int main() {
    char buf[100];
    int score = 0;

    buf[0] = 0;
    emit_mov(buf, 0, 42);
    if (check(buf, "MOV R0, #42\\n")) score = score + 1;

    buf[0] = 0;
    emit_mov(buf, 1, 5);
    if (check(buf, "MOV R1, #5\\n")) score = score + 1;

    buf[0] = 0;
    emit_mov(buf, 0, 0);
    if (check(buf, "MOV R0, #0\\n")) score = score + 1;

    return score;
}
```

16.4.9 3.2 Émettre ADD/SUB/MUL

```
int strlen(char* s) { int i = 0; while (s[i]) i = i + 1; return i; }
void append(char* buf, char* s) {
    int i = strlen(buf); int j = 0;
    while (s[j]) { buf[i] = s[j]; i = i + 1; j = j + 1; }
    buf[i] = 0;
}

void append_num(char* buf, int n) {
    if (n == 0) { append(buf, "0"); return; }
    char tmp[12]; int i = 11; tmp[11] = 0;
    while (n > 0) { i = i - 1; tmp[i] = '0' + (n % 10); n = n / 10; }
    append(buf, tmp + i);
}

void emit_op(char* buf, char op, int rd, int rn, int rm) {
    if (op == '+') append(buf, "ADD R");
}
```

```

    else if (op == '-') append(buf, "SUB R");
    else if (op == '*') append(buf, "MUL R");
    append_num(buf, rd);
    append(buf, ", R");
    append_num(buf, rn);
    append(buf, ", R");
    append_num(buf, rm);
    append(buf, "\\n");
}

int check(char* a, char* b) {
    int i = 0;
    while (a[i] && b[i]) { if (a[i] != b[i]) return 0; i = i + 1; }
    return a[i] == b[i];
}

int main() {
    char buf[100];
    int score = 0;

    buf[0] = 0;
    emit_op(buf, '+', 0, 1, 2);
    if (check(buf, "ADD R0, R1, R2\\n")) score = score + 1;

    buf[0] = 0;
    emit_op(buf, '-', 0, 0, 1);
    if (check(buf, "SUB R0, R0, R1\\n")) score = score + 1;

    buf[0] = 0;
    emit_op(buf, '*', 2, 3, 4);
    if (check(buf, "MUL R2, R3, R4\\n")) score = score + 1;

    return score;
}

```

16.4.10 3.3 Émettre PUSH/POP

```

int strlen(char* s) { int i = 0; while (s[i]) i = i + 1; return i; }
void append(char* buf, char* s) {
    int i = strlen(buf); int j = 0;
    while (s[j]) { buf[i] = s[j]; i = i + 1; j = j + 1; }
    buf[i] = 0;
}

void append_num(char* buf, int n) {

```

```
    if (n == 0) { append(buf, "0"); return; }
    char tmp[12]; int i = 11; tmp[11] = 0;
    while (n > 0) { i = i - 1; tmp[i] = '0' + (n % 10); n = n / 10; }
    append(buf, tmp + i);
}

void emit_push(char* buf, int reg) {
    append(buf, "STR R");
    append_num(buf, reg);
    append(buf, ", [SP, #-4]!\n");
}

void emit_pop(char* buf, int reg) {
    append(buf, "LDR R");
    append_num(buf, reg);
    append(buf, ", [SP], #4\n");
}

int check(char* a, char* b) {
    int i = 0;
    while (a[i] && b[i]) { if (a[i] != b[i]) return 0; i = i + 1; }
    return a[i] == b[i];
}

int main() {
    char buf[100];
    int score = 0;

    buf[0] = 0;
    emit_push(buf, 0);
    if (check(buf, "STR R0, [SP, #-4]!\n")) score = score + 1;

    buf[0] = 0;
    emit_pop(buf, 1);
    if (check(buf, "LDR R1, [SP], #4\n")) score = score + 1;

    buf[0] = 0;
    emit_push(buf, 0);
    emit_pop(buf, 1);
    if (check(buf, "STR R0, [SP, #-4]!\nLDR R1, [SP], #4\n")) score = score +
        1;

    return score;
}
```

16.4.11 4.1 Compiler une Constante

```
int strlen(char* s) { int i = 0; while (s[i]) i = i + 1; return i; }
void append(char* buf, char* s) {
    int i = strlen(buf); int j = 0;
    while (s[j]) { buf[i] = s[j]; i = i + 1; j = j + 1; }
    buf[i] = 0;
}
void append_num(char* buf, int n) {
    if (n == 0) { append(buf, "0"); return; }
    char tmp[12]; int i = 11; tmp[11] = 0;
    while (n > 0) { i = i - 1; tmp[i] = '0' + (n % 10); n = n / 10; }
    append(buf, tmp + i);
}
void emit_mov(char* buf, int reg, int val) {
    append(buf, "MOV R"); append_num(buf, reg);
    append(buf, ", #"); append_num(buf, val); append(buf, "\\n");
}

int is_digit(char c) { return c >= '0' && c <= '9'; }
char* input;
int pos;

int parse_num() {
    int v = 0;
    while (is_digit(input[pos])) { v = v * 10 + (input[pos] - '0'); pos = pos + 1; }
    return v;
}

void codegen_const(char* buf) {
    int n = parse_num();
    emit_mov(buf, 0, n);
}

int check(char* a, char* b) {
    int i = 0;
    while (a[i] && b[i]) { if (a[i] != b[i]) return 0; i = i + 1; }
    return a[i] == b[i];
}

int main() {
    char buf[100];
    int score = 0;

    buf[0] = 0; input = "42"; pos = 0;
    codegen_const(buf);
    if (check(buf, "MOV R0, #42\\n")) score = score + 1;
}
```

```

    buf[0] = 0; input = "5"; pos = 0;
    codegen_const(buf);
    if (check(buf, "MOV R0, #5\\n")) score = score + 1;

    buf[0] = 0; input = "123"; pos = 0;
    codegen_const(buf);
    if (check(buf, "MOV R0, #123\\n")) score = score + 1;

    return score;
}

```

16.4.12 4.2 Compiler a + b

```

int strlen(char* s) { int i = 0; while (s[i]) i = i + 1; return i; }
void append(char* buf, char* s) {
    int i = strlen(buf); int j = 0;
    while (s[j]) { buf[i] = s[j]; i = i + 1; j = j + 1; } buf[i] = 0;
}
void append_num(char* buf, int n) {
    if (n == 0) { append(buf, "0"); return; }
    char tmp[12]; int i = 11; tmp[11] = 0;
    while (n > 0) { i = i - 1; tmp[i] = '0' + (n % 10); n = n / 10; }
    append(buf, tmp + i);
}
void emit_mov(char* buf, int reg, int val) {
    append(buf, "MOV R"); append_num(buf, reg);
    append(buf, ", #"); append_num(buf, val); append(buf, "\\n");
}
void emit_push(char* buf, int reg) {
    append(buf, "STR R"); append_num(buf, reg); append(buf, ", [SP, #-4]\\n");
}
void emit_pop(char* buf, int reg) {
    append(buf, "LDR R"); append_num(buf, reg); append(buf, ", [SP], #4\\n");
}
void emit_op(char* buf, char op, int rd, int rn, int rm) {
    if (op == '+') append(buf, "ADD R");
    else if (op == '-') append(buf, "SUB R");
    else if (op == '*') append(buf, "MUL R");
    append_num(buf, rd); append(buf, ", R"); append_num(buf, rn);
    append(buf, ", R"); append_num(buf, rm); append(buf, "\\n");
}
int is_digit(char c) { return c >= '0' && c <= '9'; }

```

```
char* input;
int pos;
void skip() { while (input[pos] == ' ') pos = pos + 1; }
int parse_num() {
    skip();
    int v = 0;
    while (is_digit(input[pos])) { v = v * 10 + (input[pos] - '0'); pos = pos +
        ↪ 1; }
    return v;
}

void codegen_binop(char* buf) {
    int a = parse_num();
    emit_mov(buf, 0, a);
    emit_push(buf, 0);

    skip();
    char op = input[pos];
    pos = pos + 1;

    int b = parse_num();
    emit_mov(buf, 0, b);
    emit_pop(buf, 1);
    emit_op(buf, op, 0, 1, 0);
}

int contains(char* buf, char* sub) {
    int i = 0;
    while (buf[i]) {
        int j = 0;
        while (sub[j] && buf[i+j] == sub[j]) j = j + 1;
        if (sub[j] == 0) return 1;
        i = i + 1;
    }
    return 0;
}

int main() {
    char buf[200];
    int score = 0;

    buf[0] = 0; input = "3 + 5"; pos = 0;
    codegen_binop(buf);
    if (contains(buf, "MOV R0, #3") && contains(buf, "MOV R0, #5") &&
        contains(buf, "ADD R0")) score = score + 1;

    buf[0] = 0; input = "10 - 4"; pos = 0;
    codegen_binop(buf);
    if (contains(buf, "MOV R0, #10") && contains(buf, "SUB R0")) score = score +
        ↪ 1;
```

```

    buf[0] = 0; input = "6 * 7"; pos = 0;
    codegen_binop(buf);
    if (contains(buf, "MUL R0")) score = score + 1;

    return score;
}

```

16.4.13 4.3 Compiler Expressions Complètes

```

int strlen(char* s) { int i = 0; while (s[i]) i = i + 1; return i; }
void append(char* buf, char* s) {
    int i = strlen(buf); int j = 0;
    while (s[j]) { buf[i] = s[j]; i = i + 1; j = j + 1; } buf[i] = 0;
}
void append_num(char* buf, int n) {
    if (n == 0) { append(buf, "0"); return; }
    char tmp[12]; int i = 11; tmp[11] = 0;
    while (n > 0) { i = i - 1; tmp[i] = '0' + (n % 10); n = n / 10; }
    append(buf, tmp + i);
}
void emit_mov(char* buf, int reg, int val) {
    append(buf, "MOV R"); append_num(buf, reg);
    append(buf, ", #"); append_num(buf, val); append(buf, "\\n");
}
void emit_push(char* buf, int reg) {
    append(buf, "STR R"); append_num(buf, reg); append(buf, ", [SP, #-4]\\n");
}
void emit_pop(char* buf, int reg) {
    append(buf, "LDR R"); append_num(buf, reg); append(buf, ", [SP], #4\\n");
}
void emit_op(char* buf, char op, int rd, int rn, int rm) {
    if (op == '+') append(buf, "ADD R");
    else if (op == '-') append(buf, "SUB R");
    else if (op == '*') append(buf, "MUL R");
    else if (op == '/') append(buf, "SDIV R");
    append_num(buf, rd); append(buf, ", R"); append_num(buf, rn);
    append(buf, ", R"); append_num(buf, rm); append(buf, "\\n");
}

int is_digit(char c) { return c >= '0' && c <= '9'; }
char* input;
int pos;
char* out;

```



```
void skip() { while (input[pos] == ' ') pos = pos + 1; }

void codegen_expr();

void codegen_factor() {
    skip();
    int v = 0;
    while (is_digit(input[pos])) { v = v * 10 + (input[pos] - '0'); pos = pos + 1; }
    emit_mov(out, 0, v);
}

void codegen_term() {
    codegen_factor();
    while (1) {
        skip();
        char op = input[pos];
        if (op != '*' && op != '/') break;
        pos = pos + 1;
        emit_push(out, 0);
        codegen_factor();
        emit_pop(out, 1);
        emit_op(out, op, 0, 1, 0);
    }
}

void codegen_expr() {
    codegen_term();
    while (1) {
        skip();
        char op = input[pos];
        if (op != '+' && op != '-') break;
        pos = pos + 1;
        emit_push(out, 0);
        codegen_term();
        emit_pop(out, 1);
        emit_op(out, op, 0, 1, 0);
    }
}

void compile(char* buf, char* src) {
    out = buf;
    input = src;
    pos = 0;
    buf[0] = 0;
    codegen_expr();
}
```

```
int contains(char* buf, char* sub) {
    int i = 0;
    while (buf[i]) {
        int j = 0; while (sub[j] && buf[i+j] == sub[j]) j = j + 1;
        if (sub[j] == 0) return 1;
        i = i + 1;
    }
    return 0;
}

int count(char* buf, char* sub) {
    int c = 0, i = 0;
    while (buf[i]) {
        int j = 0; while (sub[j] && buf[i+j] == sub[j]) j = j + 1;
        if (sub[j] == 0) c = c + 1;
        i = i + 1;
    }
    return c;
}

int main() {
    char buf[500];
    int score = 0;

    compile(buf, "42");
    if (contains(buf, "MOV R0, #42")) score = score + 1;

    compile(buf, "3 + 5");
    if (contains(buf, "ADD R0")) score = score + 1;

    compile(buf, "2 + 3 * 4");
    if (contains(buf, "MUL R0") && contains(buf, "ADD R0")) score = score + 1;

    compile(buf, "1 + 2 + 3");
    if (count(buf, "ADD R0") == 2) score = score + 1;

    return score;
}
```

16.4.14 5.1 Générer des Labels

```
int strlen(char* s) { int i = 0; while (s[i]) i = i + 1; return i; }
void append(char* buf, char* s) {
    int i = strlen(buf); int j = 0;
```

```
    while (s[j]) { buf[i] = s[j]; i = i + 1; j = j + 1; } buf[i] = 0;
}

void append_num(char* buf, int n) {
    if (n == 0) { append(buf, "0"); return; }
    char tmp[12]; int i = 11; tmp[11] = 0;
    while (n > 0) { i = i - 1; tmp[i] = '0' + (n % 10); n = n / 10; }
    append(buf, tmp + i);
}

void emit_label(char* buf, char* prefix, int num) {
    append(buf, ".L");
    append(buf, prefix);
    append(buf, "_");
    append_num(buf, num);
    append(buf, ":\n");
}

void emit_branch(char* buf, int cond, char* prefix, int num) {
    if (cond == 0) append(buf, "B .L");
    else if (cond == 1) append(buf, "BEQ .L");
    else if (cond == 2) append(buf, "BNE .L");
    else if (cond == 3) append(buf, "BLT .L");
    else if (cond == 4) append(buf, "BGE .L");
    append(buf, prefix);
    append(buf, "_");
    append_num(buf, num);
    append(buf, "\n");
}

int check(char* a, char* b) {
    int i = 0;
    while (a[i] && b[i]) { if (a[i] != b[i]) return 0; i = i + 1; }
    return a[i] == b[i];
}

int main() {
    char buf[100];
    int score = 0;

    buf[0] = 0;
    emit_label(buf, "if", 1);
    if (check(buf, ".Lif_1:\n")) score = score + 1;

    buf[0] = 0;
    emit_branch(buf, 0, "end", 2);
    if (check(buf, "B .Lend_2:\n")) score = score + 1;

    buf[0] = 0;
    emit_branch(buf, 1, "else", 3);
}
```

```

    if (check(buf, "BEQ .Lelse_3\\n")) score = score + 1;

    buf[0] = 0;
    emit_branch(buf, 3, "loop", 0);
    if (check(buf, "BLT .Lloop_0\\n")) score = score + 1;

    return score;
}

```

16.4.15 5.2 Générer Comparaisons

```

int strlen(char* s) { int i = 0; while (s[i]) i = i + 1; return i; }
void append(char* buf, char* s) {
    int i = strlen(buf); int j = 0;
    while (s[j]) { buf[i] = s[j]; i = i + 1; j = j + 1; } buf[i] = 0;
}
void append_num(char* buf, int n) {
    if (n == 0) { append(buf, "0"); return; }
    char tmp[12]; int i = 11; tmp[11] = 0;
    while (n > 0) { i = i - 1; tmp[i] = '0' + (n % 10); n = n / 10; }
    append(buf, tmp + i);
}

void emit_cmp(char* buf, int rn, int rm) {
    append(buf, "CMP R");
    append_num(buf, rn);
    append(buf, ", R");
    append_num(buf, rm);
    append(buf, "\\n");
}

int get_branch_cond(char op) {
    if (op == '<') return 4; // BGE
    if (op == '>') return 3; // BLT (simplifié, >= aussi)
    if (op == '=') return 2; // BNE
    if (op == '!') return 1; // BEQ
    return 0;
}

int check(char* a, char* b) {
    int i = 0;
    while (a[i] && b[i]) { if (a[i] != b[i]) return 0; i = i + 1; }
    return a[i] == b[i];
}

```

```

int main() {
    char buf[100];
    int score = 0;

    buf[0] = 0;
    emit_cmp(buf, 0, 1);
    if (check(buf, "CMP R0, R1\\n")) score = score + 1;

    if (get_branch_cond('<') == 4) score = score + 1;
    if (get_branch_cond('=') == 2) score = score + 1;
    if (get_branch_cond('!=') == 1) score = score + 1;
    if (get_branch_cond('>') == 3) score = score + 1;

    return score;
}

```

16.4.16 5.3 Compiler if/else

```

int strlen(char* s) { int i = 0; while (s[i]) i = i + 1; return i; }
void append(char* buf, char* s) {
    int i = strlen(buf); int j = 0;
    while (s[j]) { buf[i] = s[j]; i = i + 1; j = j + 1; } buf[i] = 0;
}
void append_num(char* buf, int n) {
    if (n == 0) { append(buf, "0"); return; }
    char tmp[12]; int i = 11; tmp[11] = 0;
    while (n > 0) { i = i - 1; tmp[i] = '0' + (n % 10); n = n / 10; }
    append(buf, tmp + i);
}
void emit_cmp(char* buf, int rn, int rm) {
    append(buf, "CMP R"); append_num(buf, rn);
    append(buf, ", R"); append_num(buf, rm); append(buf, "\\n");
}
void emit_label(char* buf, char* prefix, int num) {
    append(buf, ".L"); append(buf, prefix); append(buf, "_");
    append_num(buf, num); append(buf, ":\n");
}
void emit_branch(char* buf, int cond, char* prefix, int num) {
    if (cond == 0) append(buf, "B .L");
    else if (cond == 4) append(buf, "BGE .L");
    append(buf, prefix); append(buf, "_");
    append_num(buf, num); append(buf, "\\n");
}

```

```
void emit_mov(char* buf, int reg, int val) {
    append(buf, "MOV R"); append_num(buf, reg);
    append(buf, ", #"); append_num(buf, val); append(buf, "\\n");
}

void codegen_if(char* buf, int n) {
    emit_cmp(buf, 0, 1);
    emit_branch(buf, 4, "else", n);
    emit_mov(buf, 2, 1);
    emit_branch(buf, 0, "end", n);
    emit_label(buf, "else", n);
    emit_mov(buf, 2, 0);
    emit_label(buf, "end", n);
}

int contains(char* buf, char* sub) {
    int i = 0;
    while (buf[i]) {
        int j = 0; while (sub[j] && buf[i+j] == sub[j]) j = j + 1;
        if (sub[j] == 0) return 1;
        i = i + 1;
    }
    return 0;
}

int main() {
    char buf[300];
    int score = 0;

    buf[0] = 0;
    codegen_if(buf, 1);

    if (contains(buf, "CMP R0, R1")) score = score + 1;
    if (contains(buf, "BGE .Lelse_1")) score = score + 1;
    if (contains(buf, "B .Lend_1")) score = score + 1;
    if (contains(buf, ".Lelse_1:")) score = score + 1;
    if (contains(buf, ".Lend_1:")) score = score + 1;

    return score;
}
```

16.4.17 5.4 Compiler while

```

int strlen(char* s) { int i = 0; while (s[i]) i = i + 1; return i; }
void append(char* buf, char* s) {
    int i = strlen(buf); int j = 0;
    while (s[j]) { buf[i] = s[j]; i = i + 1; j = j + 1; } buf[i] = 0;
}
void append_num(char* buf, int n) {
    if (n == 0) { append(buf, "0"); return; }
    char tmp[12]; int i = 11; tmp[11] = 0;
    while (n > 0) { i = i - 1; tmp[i] = '0' + (n % 10); n = n / 10; }
    append(buf, tmp + i);
}
void emit_cmp(char* buf, int rn, int rm) {
    append(buf, "CMP R"); append_num(buf, rn);
    append(buf, ", R"); append_num(buf, rm); append(buf, "\\n");
}
void emit_label(char* buf, char* prefix, int num) {
    append(buf, ".L"); append(buf, prefix); append(buf, "_");
    append_num(buf, num); append(buf, ":\n");
}
void emit_branch(char* buf, int cond, char* prefix, int num) {
    if (cond == 0) append(buf, "B .L");
    else if (cond == 4) append(buf, "BGE .L");
    append(buf, prefix); append(buf, "_");
    append_num(buf, num); append(buf, "\\n");
}
void emit_add_imm(char* buf, int rd, int rn, int imm) {
    append(buf, "ADD R"); append_num(buf, rd);
    append(buf, ", R"); append_num(buf, rn);
    append(buf, ", #"); append_num(buf, imm); append(buf, "\\n");
}

void codegen_while(char* buf, int n) {
    emit_label(buf, "while", n);
    emit_cmp(buf, 0, 1);
    emit_branch(buf, 4, "end", n);
    emit_add_imm(buf, 0, 0, 1);
    emit_branch(buf, 0, "while", n);
    emit_label(buf, "end", n);
}

int contains(char* buf, char* sub) {
    int i = 0;
    while (buf[i]) {
        int j = 0; while (sub[j] && buf[i+j] == sub[j]) j = j + 1;
        if (sub[j] == 0) return 1;
        i = i + 1;
    }
    return 0;
}

```

```

}

int main() {
    char buf[300];
    int score = 0;

    buf[0] = 0;
    codegen_while(buf, 2);

    if (contains(buf, ".Lwhile_2:")) score = score + 1;
    if (contains(buf, "CMP R0, R1")) score = score + 1;
    if (contains(buf, "BGE .Lend_2")) score = score + 1;
    if (contains(buf, "B .Lwhile_2")) score = score + 1;
    if (contains(buf, ".Lend_2:")) score = score + 1;

    return score;
}

```

16.4.18 6.1 Prologue et Épilogue

```

int strlen(char* s) { int i = 0; while (s[i]) i = i + 1; return i; }
void append(char* buf, char* s) {
    int i = strlen(buf); int j = 0;
    while (s[j]) { buf[i] = s[j]; i = i + 1; j = j + 1; } buf[i] = 0;
}

void emit_prologue(char* buf, char* name) {
    append(buf, name);
    append(buf, ":\n    STR LR, [SP, #-4]!\n");
}

void emit_epilogue(char* buf) {
    append(buf, "    LDR LR, [SP], #4\n    BX LR\n");
}

int contains(char* buf, char* sub) {
    int i = 0;
    while (buf[i]) {
        int j = 0; while (sub[j] && buf[i+j] == sub[j]) j = j + 1;
        if (sub[j] == 0) return 1;
        i = i + 1;
    }
    return 0;
}

```



```

int main() {
    char buf[200];
    int score = 0;

    buf[0] = 0;
    emit_prologue(buf, "add");
    if (contains(buf, "add:") && contains(buf, "STR LR")) score = score + 1;

    buf[0] = 0;
    emit_epilogue(buf);
    if (contains(buf, "LDR LR") && contains(buf, "BX LR")) score = score + 1;

    buf[0] = 0;
    emit_prologue(buf, "main");
    emit_epilogue(buf);
    if (contains(buf, "main:") && contains(buf, "BX LR")) score = score + 1;

    return score;
}

```

16.4.19 6.2 Appels de Fonction

```

int strlen(char* s) { int i = 0; while (s[i]) i = i + 1; return i; }
void append(char* buf, char* s) {
    int i = strlen(buf); int j = 0;
    while (s[j]) { buf[i] = s[j]; i = i + 1; j = j + 1; } buf[i] = 0;
}
void append_num(char* buf, int n) {
    if (n == 0) { append(buf, "0"); return; }
    char tmp[12]; int i = 11; tmp[11] = 0;
    while (n > 0) { i = i - 1; tmp[i] = '0' + (n % 10); n = n / 10; }
    append(buf, tmp + i);
}
void emit_mov(char* buf, int reg, int val) {
    append(buf, "MOV R"); append_num(buf, reg);
    append(buf, ", #"); append_num(buf, val); append(buf, "\\n");
}
void emit_call(char* buf, char* name) {
    append(buf, "BL ");
    append(buf, name);
    append(buf, "\\n");
}

```

```

int contains(char* buf, char* sub) {
    int i = 0;
    while (buf[i]) {
        int j = 0; while (sub[j] && buf[i+j] == sub[j]) j = j + 1;
        if (sub[j] == 0) return 1;
        i = i + 1;
    }
    return 0;
}

int main() {
    char buf[200];
    int score = 0;

    buf[0] = 0;
    emit_call(buf, "putchar");
    if (contains(buf, "BL putchar")) score = score + 1;

    buf[0] = 0;
    emit_mov(buf, 0, 65);
    emit_call(buf, "putchar");
    if (contains(buf, "MOV R0, #65") && contains(buf, "BL putchar")) score =
        score + 1;

    buf[0] = 0;
    emit_mov(buf, 0, 3);
    emit_mov(buf, 1, 5);
    emit_call(buf, "add");
    if (contains(buf, "MOV R0, #3") && contains(buf, "MOV R1, #5") &&
        contains(buf, "BL add")) score = score + 1;

    return score;
}

```

16.4.20 7.1 Mini-Compilateur Complet

```

// === UTILITAIRES ===
int strlen(char* s) { int i = 0; while (s[i]) i = i + 1; return i; }
void append(char* buf, char* s) {
    int i = strlen(buf); int j = 0;
    while (s[j]) { buf[i] = s[j]; i = i + 1; j = j + 1; } buf[i] = 0;
}
void append_num(char* buf, int n) {

```

```

    if (n == 0) { append(buf, "0"); return; }
    char tmp[12]; int i = 11; tmp[11] = 0;
    while (n > 0) { i = i - 1; tmp[i] = '0' + (n % 10); n = n / 10; }
    append(buf, tmp + i);
}

// === ÉMETTEURS ===
void emit_mov(char* buf, int reg, int val) {
    append(buf, "MOV R"); append_num(buf, reg);
    append(buf, ", #"); append_num(buf, val); append(buf, "\\n");
}
void emit_push(char* buf, int reg) {
    append(buf, "STR R"); append_num(buf, reg); append(buf, ", [SP, #-4]\\n");
}
void emit_pop(char* buf, int reg) {
    append(buf, "LDR R"); append_num(buf, reg); append(buf, ", [SP], #4\\n");
}
void emit_op(char* buf, char op, int rd, int rn, int rm) {
    if (op == '+') append(buf, "ADD R");
    else if (op == '-') append(buf, "SUB R");
    else if (op == '*') append(buf, "MUL R");
    append_num(buf, rd); append(buf, ", R"); append_num(buf, rn);
    append(buf, ", R"); append_num(buf, rm); append(buf, "\\n");
}

// === PARSER + CODEGEN ===
int is_digit(char c) { return c >= '0' && c <= '9'; }
char* input;
int pos;
char* out;

void skip() { while (input[pos] == ' ') pos = pos + 1; }

void codegen_expr();

void codegen_factor() {
    skip();
    if (input[pos] == '(') {
        pos = pos + 1;
        codegen_expr();
        skip();
        pos = pos + 1; // ')'
    } else {
        int v = 0;
        while (is_digit(input[pos])) {
            v = v * 10 + (input[pos] - '0');
            pos = pos + 1;
        }
        emit_mov(out, 0, v);
    }
}

```

```
    }
}

void codegen_term() {
    codegen_factor();
    while (1) {
        skip();
        char op = input[pos];
        if (op != '*' && op != '/') break;
        pos = pos + 1;
        emit_push(out, 0);
        codegen_factor();
        emit_pop(out, 1);
        emit_op(out, op, 0, 1, 0);
    }
}

void codegen_expr() {
    codegen_term();
    while (1) {
        skip();
        char op = input[pos];
        if (op != '+' && op != '-') break;
        pos = pos + 1;
        emit_push(out, 0);
        codegen_term();
        emit_pop(out, 1);
        emit_op(out, op, 0, 1, 0);
    }
}

void compile(char* buf, char* src) {
    out = buf; input = src; pos = 0; buf[0] = 0;
    codegen_expr();
}

int contains(char* buf, char* sub) {
    int i = 0;
    while (buf[i]) {
        int j = 0; while (sub[j] && buf[i+j] == sub[j]) j = j + 1;
        if (sub[j] == 0) return 1;
        i = i + 1;
    }
    return 0;
}

int count(char* buf, char* sub) {
    int c = 0, i = 0;
    while (buf[i]) {
```

```
        int j = 0; while (sub[j] && buf[i+j] == sub[j]) j = j + 1;
        if (sub[j] == 0) c = c + 1;
        i = i + 1;
    }
    return c;
}

int main() {
    char buf[500];
    int score = 0;

    compile(buf, "42");
    if (contains(buf, "MOV R0, #42")) score = score + 1;

    compile(buf, "(5)");
    if (contains(buf, "MOV R0, #5")) score = score + 1;

    compile(buf, "2 + 3 * 4");
    if (contains(buf, "MUL") && contains(buf, "ADD")) score = score + 1;

    compile(buf, "(2 + 3) * 4");
    if (count(buf, "ADD") == 1 && count(buf, "MUL") == 1) score = score + 1;

    compile(buf, "(1 + 2) * (3 + 4)");
    if (count(buf, "ADD") == 2 && count(buf, "MUL") == 1) score = score + 1;

    return score;
}
```

16.5 E. Solutions Systeme d'Exploitation

16.5.1 Bootstrap

```
// Bootstrap - Solution

int bss_start = 0x00402800;
int bss_end = 0x00403000;

int main() {
    return 42;
}

// Note: En vrai, _start serait en ASM
```

```
// Ici on simule le concept
int _start() {
    int ptr;
    int result;

    // Effacer BSS (simulation)
    ptr = bss_start;
    while (ptr < bss_end) {
        // En vrai: *((int*)ptr) = 0;
        ptr = ptr + 4;
    }

    // Appeler main
    result = main();

    return result;
}
```

16.5.2 Bump Allocator

```
// Bump Allocator - Solution

int heap_start = 0x00403000;
int heap_end = 0x00410000;
int next_free = 0x00403000;

int bump_alloc(int size) {
    int addr;

    // Vérifier qu'il reste de la place
    if (next_free + size > heap_end) {
        return 0;
    }

    // Sauvegarder l'adresse actuelle
    addr = next_free;

    // Avancer le pointeur
    next_free = next_free + size;

    return addr;
}

int main() {
```

```
int ptr1;
int ptr2;

ptr1 = bump_alloc(100);
ptr2 = bump_alloc(200);

return ptr2 - ptr1;
}
```

16.5.3 Free List

```
// Free List - Solution simplifiée
// Note: Version conceptuelle, pas de vrais pointeurs

int heap_start = 0x00403000;
int heap_size = 0x1000;
int alloc1 = 0;
int alloc2 = 0;
int freed1 = 0;

int my_malloc(int size) {
    if (alloc1 == 0) {
        alloc1 = 1;
        return heap_start + 100;
    }
    if (alloc2 == 0) {
        alloc2 = 1;
        return heap_start + 200;
    }
    if (freed1 == 1) {
        freed1 = 0;
        return heap_start + 100;
    }
    return 0;
}

void my_free(int ptr) {
    if (ptr == heap_start + 100) {
        freed1 = 1;
        alloc1 = 0;
    }
}

int main() {
```

```
int p1;
int p2;
int p3;

p1 = my_malloc(50);
p2 = my_malloc(50);
my_free(p1);
p3 = my_malloc(30);

if (p3 == p1) {
    return 1;
}
return 0;
}
```

16.5.4 Driver Écran

```
// Driver Écran - Solution (optimisé)

int *SCREEN = (int*)0x00400000;

// Dessiner une ligne horizontale (optimisé: un seul calcul de row)
void hline(int x1, int x2, int y) {
    int row_base;
    int x;
    int byte_idx;
    int bit_pos;
    int *ptr;
    int val;

    // y * 10 calculé une seule fois
    row_base = (y << 3) + (y << 1); // y*8 + y*2 = y*10

    for (x = x1; x <= x2; x = x + 1) {
        ptr = SCREEN + row_base + (x >> 5);
        byte_idx = (x >> 3) & 3;
        bit_pos = byte_idx * 8 + 7 - (x & 7);
        val = *ptr;
        *ptr = val | (1 << bit_pos);
    }
}

// Dessiner une ligne verticale
void vline(int x, int y1, int y2) {
```



```
int y;
int row_base;
int byte_idx;
int bit_pos;
int *ptr;
int val;
int mask;

byte_idx = (x >> 3) & 3;
bit_pos = byte_idx * 8 + 7 - (x & 7);
mask = 1 << bit_pos;

for (y = y1; y <= y2; y = y + 1) {
    row_base = (y << 3) + (y << 1);
    ptr = SCREEN + row_base + (x >> 5);
    val = *ptr;
    *ptr = val | mask;
}
}

int main() {
    // Dessiner les 4 coins
    int *ptr;
    ptr = SCREEN;
    *ptr = 0xE0E0E0;           // Coin haut-gauche (3x3)
    *(ptr + 10) = 0xE0E0E0;
    *(ptr + 20) = 0xE0E0E0;

    ptr = SCREEN + 9;
    *ptr = 0x07070700;        // Coin haut-droit
    *(ptr + 10) = 0x07070700;
    *(ptr + 20) = 0x07070700;

    ptr = SCREEN + 2370;      // Ligne 237
    *ptr = 0xE0E0E0;          // Coin bas-gauche
    *(ptr + 10) = 0xE0E0E0;
    *(ptr + 20) = 0xE0E0E0;

    ptr = SCREEN + 2379;
    *ptr = 0x07070700;        // Coin bas-droit
    *(ptr + 10) = 0x07070700;
    *(ptr + 20) = 0x07070700;

    // Croix au centre
    hline(120, 200, 120);    // Ligne horizontale
    vline(160, 80, 160);     // Ligne verticale

    return 4;
}
```

16.5.5 Police Bitmap

```
// Police Bitmap - Solution (affichage réel)
//
// Table des caractères ASCII 8x8 (hex: ligne0-ligne7)
// =====
// ' ' (32): 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
// '!' (33): 0x18,0x18,0x18,0x18,0x18,0x00,0x18,0x00
// '"' (34): 0x6C,0x6C,0x00,0x00,0x00,0x00,0x00,0x00
// '#' (35): 0x24,0x7E,0x24,0x24,0x7E,0x24,0x00,0x00
// '$' (36): 0x18,0x3E,0x58,0x3C,0x1A,0x7C,0x18,0x00
// '%' (37): 0x62,0x64,0x08,0x10,0x26,0x46,0x00,0x00
// '&' (38): 0x30,0x48,0x30,0x56,0x88,0x76,0x00,0x00
// ''' (39): 0x18,0x18,0x30,0x00,0x00,0x00,0x00,0x00
// '(' (40): 0x0C,0x18,0x30,0x30,0x30,0x18,0x0C,0x00
// ')' (41): 0x30,0x18,0x0C,0x0C,0x0C,0x18,0x30,0x00
// '*' (42): 0x00,0x24,0x18,0x7E,0x18,0x24,0x00,0x00
// '+' (43): 0x00,0x18,0x18,0x7E,0x18,0x18,0x00,0x00
// ',' (44): 0x00,0x00,0x00,0x00,0x00,0x18,0x18,0x30
// '-' (45): 0x00,0x00,0x00,0x7E,0x00,0x00,0x00,0x00
// '.' (46): 0x00,0x00,0x00,0x00,0x00,0x18,0x18,0x00
// '/' (47): 0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x00
// '0' (48): 0x3C,0x46,0x4A,0x52,0x62,0x3C,0x00,0x00
// '1' (49): 0x18,0x38,0x18,0x18,0x18,0x3C,0x00,0x00
// '2' (50): 0x3C,0x42,0x02,0x1C,0x20,0x7E,0x00,0x00
// '3' (51): 0x3C,0x42,0x0C,0x02,0x42,0x3C,0x00,0x00
// '4' (52): 0x04,0x0C,0x14,0x24,0x7E,0x04,0x00,0x00
// '5' (53): 0x7E,0x40,0x7C,0x02,0x42,0x3C,0x00,0x00
// '6' (54): 0x1C,0x20,0x7C,0x42,0x42,0x3C,0x00,0x00
// '7' (55): 0x7E,0x02,0x04,0x08,0x10,0x10,0x00,0x00
// '8' (56): 0x3C,0x42,0x3C,0x42,0x42,0x3C,0x00,0x00
// '9' (57): 0x3C,0x42,0x42,0x3E,0x04,0x38,0x00,0x00
// ':' (58): 0x00,0x18,0x18,0x00,0x18,0x18,0x00,0x00
// ';' (59): 0x00,0x18,0x18,0x00,0x18,0x18,0x30,0x00
// '<' (60): 0x06,0x18,0x60,0x60,0x18,0x06,0x00,0x00
// '=' (61): 0x00,0x00,0x7E,0x00,0x7E,0x00,0x00,0x00
// '>' (62): 0x60,0x18,0x06,0x06,0x18,0x60,0x00,0x00
// '?' (63): 0x3C,0x42,0x04,0x08,0x00,0x08,0x00,0x00
// '@' (64): 0x3C,0x42,0x5E,0x5E,0x40,0x3C,0x00,0x00
// 'A' (65): 0x18,0x24,0x42,0x7E,0x42,0x42,0x42,0x00
// 'B' (66): 0x7C,0x42,0x7C,0x42,0x42,0x7C,0x00,0x00
// 'C' (67): 0x3C,0x42,0x40,0x40,0x42,0x3C,0x00,0x00
// 'D' (68): 0x78,0x44,0x42,0x42,0x44,0x78,0x00,0x00
```

```
// 'E' (69): 0x7E,0x40,0x7C,0x40,0x40,0x7E,0x00,0x00
// 'F' (70): 0x7E,0x40,0x7C,0x40,0x40,0x40,0x00,0x00
// 'G' (71): 0x3C,0x42,0x40,0x4E,0x42,0x3C,0x00,0x00
// 'H' (72): 0x42,0x42,0x7E,0x42,0x42,0x42,0x00,0x00
// 'I' (73): 0x3E,0x08,0x08,0x08,0x08,0x3E,0x00,0x00
// 'J' (74): 0x1E,0x04,0x04,0x04,0x44,0x38,0x00,0x00
// 'K' (75): 0x42,0x44,0x78,0x48,0x44,0x42,0x00,0x00
// 'L' (76): 0x40,0x40,0x40,0x40,0x40,0x7E,0x00,0x00
// 'M' (77): 0x42,0x66,0x5A,0x42,0x42,0x42,0x00,0x00
// 'N' (78): 0x42,0x62,0x52,0x4A,0x46,0x42,0x00,0x00
// 'O' (79): 0x3C,0x42,0x42,0x42,0x42,0x3C,0x00,0x00
// 'P' (80): 0x7C,0x42,0x7C,0x40,0x40,0x40,0x00,0x00
// 'Q' (81): 0x3C,0x42,0x42,0x4A,0x44,0x3A,0x00,0x00
// 'R' (82): 0x7C,0x42,0x7C,0x48,0x44,0x42,0x00,0x00
// 'S' (83): 0x3C,0x40,0x3C,0x02,0x42,0x3C,0x00,0x00
// 'T' (84): 0x7E,0x18,0x18,0x18,0x18,0x18,0x00,0x00
// 'U' (85): 0x42,0x42,0x42,0x42,0x42,0x3C,0x00,0x00
// 'V' (86): 0x42,0x42,0x42,0x42,0x24,0x18,0x00,0x00
// 'W' (87): 0x42,0x42,0x42,0x5A,0x66,0x42,0x00,0x00
// 'X' (88): 0x42,0x24,0x18,0x18,0x24,0x42,0x00,0x00
// 'Y' (89): 0x42,0x42,0x24,0x18,0x18,0x18,0x00,0x00
// 'Z' (90): 0x7E,0x04,0x08,0x10,0x20,0x7E,0x00,0x00
// '[' (91): 0x3C,0x30,0x30,0x30,0x30,0x3C,0x00,0x00
// '\\ ' (92): 0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x00
// ']' (93): 0x3C,0x0C,0x0C,0x0C,0x0C,0x3C,0x00,0x00
// '^' (94): 0x18,0x24,0x00,0x00,0x00,0x00,0x00,0x00
// '_' (95): 0x00,0x00,0x00,0x00,0x00,0x00,0x7E,0x00
// 'a' (97): 0x00,0x3C,0x02,0x3E,0x42,0x3E,0x00,0x00
// 'b' (98): 0x40,0x40,0x7C,0x42,0x42,0x7C,0x00,0x00
// 'c' (99): 0x00,0x3C,0x40,0x40,0x40,0x3C,0x00,0x00
// 'd' (100): 0x02,0x02,0x3E,0x42,0x42,0x3E,0x00,0x00
// 'e' (101): 0x00,0x3C,0x42,0x7E,0x40,0x3C,0x00,0x00
// 'f' (102): 0x0C,0x10,0x3C,0x10,0x10,0x10,0x00,0x00
// 'g' (103): 0x00,0x3E,0x42,0x3E,0x02,0x3C,0x00,0x00
// 'h' (104): 0x40,0x40,0x7C,0x42,0x42,0x42,0x00,0x00
// 'i' (105): 0x18,0x00,0x38,0x18,0x18,0x3C,0x00,0x00
// 'j' (106): 0x04,0x00,0x04,0x04,0x04,0x44,0x38,0x00
// 'k' (107): 0x40,0x44,0x48,0x70,0x48,0x44,0x00,0x00
// 'l' (108): 0x38,0x18,0x18,0x18,0x18,0x3C,0x00,0x00
// 'm' (109): 0x00,0x76,0x5A,0x5A,0x42,0x42,0x00,0x00
// 'n' (110): 0x00,0x7C,0x42,0x42,0x42,0x42,0x00,0x00
// 'o' (111): 0x00,0x3C,0x42,0x42,0x42,0x3C,0x00,0x00
// 'p' (112): 0x00,0x7C,0x42,0x7C,0x40,0x40,0x00,0x00
// 'q' (113): 0x00,0x3E,0x42,0x3E,0x02,0x02,0x00,0x00
// 'r' (114): 0x00,0x5C,0x60,0x40,0x40,0x40,0x00,0x00
// 's' (115): 0x00,0x3E,0x40,0x3C,0x02,0x7C,0x00,0x00
// 't' (116): 0x10,0x3C,0x10,0x10,0x10,0x0C,0x00,0x00
// 'u' (117): 0x00,0x42,0x42,0x42,0x42,0x3E,0x00,0x00
// 'v' (118): 0x00,0x42,0x42,0x42,0x24,0x18,0x00,0x00
```

```
// 'w' (119): 0x00,0x42,0x42,0x5A,0x5A,0x66,0x00,0x00
// 'x' (120): 0x00,0x42,0x24,0x18,0x24,0x42,0x00,0x00
// 'y' (121): 0x00,0x42,0x42,0x3E,0x02,0x3C,0x00,0x00
// 'z' (122): 0x00,0x7E,0x04,0x18,0x20,0x7E,0x00,0x00
// '{' (123): 0x0E,0x18,0x30,0x18,0x18,0x0E,0x00,0x00
// '|' (124): 0x18,0x18,0x18,0x18,0x18,0x18,0x00,0x00
// '}' (125): 0x70,0x18,0x0C,0x18,0x18,0x70,0x00,0x00
// '~' (126): 0x00,0x32,0x4C,0x00,0x00,0x00,0x00,0x00

int *SCREEN = (int*)0x00400000;
int pixels_drawn = 0;

void set_pixel(int x, int y) {
    int row_base;
    int byte_idx;
    int bit_pos;
    int *ptr;

    row_base = (y << 3) + (y << 1); // y * 10
    ptr = SCREEN + row_base + (x >> 5);
    byte_idx = (x >> 3) & 3;
    bit_pos = byte_idx * 8 + 7 - (x & 7);
    *ptr = *ptr | (1 << bit_pos);
    pixels_drawn = pixels_drawn + 1;
}

void draw_line(int x, int y, int line_data) {
    int bit;

    for (bit = 0; bit < 8; bit = bit + 1) {
        if ((line_data >> (7 - bit)) & 1) {
            set_pixel(x + bit, y);
        }
    }
}

void draw_char(int x, int y, int line0, int line1, int line2, int line3,
               int line4, int line5, int line6, int line7) {
    draw_line(x, y, line0);
    draw_line(x, y + 1, line1);
    draw_line(x, y + 2, line2);
    draw_line(x, y + 3, line3);
    draw_line(x, y + 4, line4);
    draw_line(x, y + 5, line5);
    draw_line(x, y + 6, line6);
    draw_line(x, y + 7, line7);
}

int main() {
```

```
// Dessiner "HI" en grand (position centrale)
// H: lignes verticales + barre horizontale
draw_char(100, 100, 0x42, 0x42, 0x42, 0x7E, 0x42, 0x42, 0x42, 0x00);

// I: barre verticale centrée
draw_char(112, 100, 0x3E, 0x08, 0x08, 0x08, 0x08, 0x08, 0x3E, 0x00);

// A
draw_char(124, 100, 0x18, 0x24, 0x42, 0x7E, 0x42, 0x42, 0x42, 0x00);

return pixels_drawn;
}
```

16.5.6 Console

```
// Console - Solution (affichage réel optimisé)

int *SCREEN = (int*)0x00400000;
int cursor_x = 0;
int cursor_y = 0;

void set_pixel(int x, int y) {
    int row_base;
    int *ptr;
    int bit_pos;

    row_base = (y << 3) + (y << 1);
    ptr = SCREEN + row_base + (x >> 5);
    bit_pos = ((x >> 3) & 3) * 8 + 7 - (x & 7);
    *ptr = *ptr | (1 << bit_pos);
}

// Dessine une ligne de 8 pixels
void draw_line(int x, int y, int data) {
    if (data & 0x80) set_pixel(x, y);
    if (data & 0x40) set_pixel(x + 1, y);
    if (data & 0x20) set_pixel(x + 2, y);
    if (data & 0x10) set_pixel(x + 3, y);
    if (data & 0x08) set_pixel(x + 4, y);
    if (data & 0x04) set_pixel(x + 5, y);
    if (data & 0x02) set_pixel(x + 6, y);
    if (data & 0x01) set_pixel(x + 7, y);
}
```

```
// Dessine H à (x, y)
void draw_H(int x, int y) {
    draw_line(x, y, 0x42);
    draw_line(x, y + 1, 0x42);
    draw_line(x, y + 2, 0x7E);
    draw_line(x, y + 3, 0x42);
    draw_line(x, y + 4, 0x42);
    draw_line(x, y + 5, 0x42);
}

// Dessine e à (x, y)
void draw_e(int x, int y) {
    draw_line(x, y + 2, 0x3C);
    draw_line(x, y + 3, 0x42);
    draw_line(x, y + 4, 0x7E);
    draw_line(x, y + 5, 0x40);
    draw_line(x, y + 6, 0x3C);
}

// Dessine l à (x, y)
void draw_l(int x, int y) {
    draw_line(x, y, 0x38);
    draw_line(x, y + 1, 0x18);
    draw_line(x, y + 2, 0x18);
    draw_line(x, y + 3, 0x18);
    draw_line(x, y + 4, 0x18);
    draw_line(x, y + 5, 0x3C);
}

// Dessine o à (x, y)
void draw_o(int x, int y) {
    draw_line(x, y + 2, 0x3C);
    draw_line(x, y + 3, 0x42);
    draw_line(x, y + 4, 0x42);
    draw_line(x, y + 5, 0x42);
    draw_line(x, y + 6, 0x3C);
}

// Dessine W à (x, y)
void draw_W(int x, int y) {
    draw_line(x, y, 0x42);
    draw_line(x, y + 1, 0x42);
    draw_line(x, y + 2, 0x42);
    draw_line(x, y + 3, 0x5A);
    draw_line(x, y + 4, 0x66);
    draw_line(x, y + 5, 0x42);
}

// Dessine r à (x, y)
```

```
void draw_r(int x, int y) {
    draw_line(x, y + 2, 0x5C);
    draw_line(x, y + 3, 0x60);
    draw_line(x, y + 4, 0x40);
    draw_line(x, y + 5, 0x40);
    draw_line(x, y + 6, 0x40);
}

// Dessine d à (x, y)
void draw_d(int x, int y) {
    draw_line(x, y, 0x02);
    draw_line(x, y + 1, 0x02);
    draw_line(x, y + 2, 0x3E);
    draw_line(x, y + 3, 0x42);
    draw_line(x, y + 4, 0x42);
    draw_line(x, y + 5, 0x3E);
}

int main() {
    // Hello (ligne 0, y=0)
    draw_H(0, 0);
    draw_e(8, 0);
    draw_l(16, 0);
    draw_l(24, 0);
    draw_o(32, 0);

    // World (ligne 1, y=8)
    draw_W(0, 8);
    draw_o(8, 8);
    draw_r(16, 8);
    draw_l(24, 8);
    draw_d(32, 8);

    cursor_y = 1;
    return cursor_y;
}
```

16.5.7 Driver Clavier

```
// Driver Clavier - Solution Interactive
// Activez "Capturer clavier" et appuyez sur des touches!

int *SCREEN = (int*)0x00400000;
int *KEYBOARD = (int*)0x00402600;
```

```
int key_count = 0;
int cursor_x = 0;

void set_pixel(int x, int y) {
    int row_base;
    int *ptr;
    int bit_pos;
    row_base = (y << 3) + (y << 1);
    ptr = SCREEN + row_base + (x >> 5);
    bit_pos = ((x >> 3) & 3) * 8 + 7 - (x & 7);
    *ptr = *ptr | (1 << bit_pos);
}

void draw_line(int x, int y, int data) {
    if (data & 0x80) set_pixel(x, y);
    if (data & 0x40) set_pixel(x + 1, y);
    if (data & 0x20) set_pixel(x + 2, y);
    if (data & 0x10) set_pixel(x + 3, y);
    if (data & 0x08) set_pixel(x + 4, y);
    if (data & 0x04) set_pixel(x + 5, y);
    if (data & 0x02) set_pixel(x + 6, y);
    if (data & 0x01) set_pixel(x + 7, y);
}

// Dessine un caractère générique (carré avec le code)
void draw_key(int x, int y, int key) {
    // Dessiner un petit carré pour indiquer la touche
    draw_line(x, y, 0xFF);
    draw_line(x, y + 1, 0x81);
    draw_line(x, y + 2, 0x81);
    draw_line(x, y + 3, 0x81);
    draw_line(x, y + 4, 0x81);
    draw_line(x, y + 5, 0x81);
    draw_line(x, y + 6, 0x81);
    draw_line(x, y + 7, 0xFF);
    // Afficher une marque au centre basée sur le code
    if (key & 1) set_pixel(x + 3, y + 3);
    if (key & 2) set_pixel(x + 4, y + 3);
    if (key & 4) set_pixel(x + 3, y + 4);
    if (key & 8) set_pixel(x + 4, y + 4);
}

int read_key() {
    return *KEYBOARD;
}

int main() {
    int key;
    int last_key;
```



```
int timeout;

last_key = 0;
timeout = 0;

// Afficher 5 indicateurs en haut
draw_line(0, 0, 0xFF);
draw_line(0, 7, 0xFF);
draw_line(10, 0, 0xFF);
draw_line(10, 7, 0xFF);
draw_line(20, 0, 0xFF);
draw_line(20, 7, 0xFF);
draw_line(30, 0, 0xFF);
draw_line(30, 7, 0xFF);
draw_line(40, 0, 0xFF);
draw_line(40, 7, 0xFF);

// Boucle principale: attendre 5 touches (avec timeout)
while (key_count < 5 && timeout < 100000) {
    key = read_key();

    // Nouvelle touche pressée?
    if (key != 0 && key != last_key) {
        draw_key(cursor_x, 16, key);
        cursor_x = cursor_x + 10;
        key_count = key_count + 1;
        timeout = 0; // Reset timeout quand touche pressée
    }

    last_key = key;
    timeout = timeout + 1;
}

// Si timeout atteint sans 5 touches, retourner quand même key_count
return 0; // Test visuel uniquement
}
```

16.5.8 Shell

```
// Shell Interactif - Solution
// Cochez "Capturer clavier" et tapez des chiffres!

int *SCREEN = (int*)0x00400000;
int *KEYBOARD = (int*)0x00402600;
```

```
int number = 0;
int cursor_x = 16;

void set_pixel(int x, int y) {
    int row_base;
    int *ptr;
    int bit_pos;
    row_base = (y << 3) + (y << 1);
    ptr = SCREEN + row_base + (x >> 5);
    bit_pos = ((x >> 3) & 3) * 8 + 7 - (x & 7);
    *ptr = *ptr | (1 << bit_pos);
}

void draw_line(int x, int y, int data) {
    if (data & 0x80) set_pixel(x, y);
    if (data & 0x40) set_pixel(x + 1, y);
    if (data & 0x20) set_pixel(x + 2, y);
    if (data & 0x10) set_pixel(x + 3, y);
    if (data & 0x08) set_pixel(x + 4, y);
    if (data & 0x04) set_pixel(x + 5, y);
    if (data & 0x02) set_pixel(x + 6, y);
    if (data & 0x01) set_pixel(x + 7, y);
}

// Dessine ">" prompt
void show_prompt() {
    draw_line(0, 0, 0x40);
    draw_line(0, 1, 0x20);
    draw_line(0, 2, 0x10);
    draw_line(0, 3, 0x20);
    draw_line(0, 4, 0x40);
}

// Dessine un chiffre simplifié
void draw_digit(int x, int d) {
    if (d == 0) {
        draw_line(x, 0, 0x3C); draw_line(x, 1, 0x42);
        draw_line(x, 2, 0x42); draw_line(x, 3, 0x42);
        draw_line(x, 4, 0x42); draw_line(x, 5, 0x3C);
    }
    if (d == 1) {
        draw_line(x, 0, 0x18); draw_line(x, 1, 0x38);
        draw_line(x, 2, 0x18); draw_line(x, 3, 0x18);
        draw_line(x, 4, 0x18); draw_line(x, 5, 0x3C);
    }
    if (d == 2) {
        draw_line(x, 0, 0x3C); draw_line(x, 1, 0x42);
        draw_line(x, 2, 0x04); draw_line(x, 3, 0x18);
        draw_line(x, 4, 0x20); draw_line(x, 5, 0x7E);
    }
}
```

```
}
if (d == 3) {
    draw_line(x, 0, 0x3C); draw_line(x, 1, 0x42);
    draw_line(x, 2, 0x0C); draw_line(x, 3, 0x02);
    draw_line(x, 4, 0x42); draw_line(x, 5, 0x3C);
}
if (d == 4) {
    draw_line(x, 0, 0x04); draw_line(x, 1, 0x0C);
    draw_line(x, 2, 0x14); draw_line(x, 3, 0x24);
    draw_line(x, 4, 0x7E); draw_line(x, 5, 0x04);
}
if (d == 5) {
    draw_line(x, 0, 0x7E); draw_line(x, 1, 0x40);
    draw_line(x, 2, 0x7C); draw_line(x, 3, 0x02);
    draw_line(x, 4, 0x42); draw_line(x, 5, 0x3C);
}
if (d == 6) {
    draw_line(x, 0, 0x1C); draw_line(x, 1, 0x20);
    draw_line(x, 2, 0x7C); draw_line(x, 3, 0x42);
    draw_line(x, 4, 0x42); draw_line(x, 5, 0x3C);
}
if (d == 7) {
    draw_line(x, 0, 0x7E); draw_line(x, 1, 0x02);
    draw_line(x, 2, 0x04); draw_line(x, 3, 0x08);
    draw_line(x, 4, 0x10); draw_line(x, 5, 0x10);
}
if (d == 8) {
    draw_line(x, 0, 0x3C); draw_line(x, 1, 0x42);
    draw_line(x, 2, 0x3C); draw_line(x, 3, 0x42);
    draw_line(x, 4, 0x42); draw_line(x, 5, 0x3C);
}
if (d == 9) {
    draw_line(x, 0, 0x3C); draw_line(x, 1, 0x42);
    draw_line(x, 2, 0x42); draw_line(x, 3, 0x3E);
    draw_line(x, 4, 0x04); draw_line(x, 5, 0x38);
}
}

int read_key() {
    return *KEYBOARD;
}

int main() {
    int key;
    int last_key;
    int timeout;

    last_key = 0;
    timeout = 0;
```

```
show_prompt();

while (timeout < 50000) {
    key = read_key();

    if (key != last_key && key != 0) {
        // Esc (27) = quitter
        if (key == 27) {
            return number;
        }
        // Enter (13) = terminer
        if (key == 13) {
            return number;
        }
        // Chiffres 0-9 (48-57)
        if (key >= 48 && key <= 57) {
            draw_digit(cursor_x, key - 48);
            cursor_x = cursor_x + 10;
            number = number * 10 + (key - 48);
            timeout = 0;
        }
    }
    last_key = key;
    timeout = timeout + 1;
}

return number;
}
```

16.5.9 Calculatrice

```
// Calculatrice Interactive - Solution
// Cochez "Capturer clavier", tapez: 3+5 Enter

int *SCREEN = (int*)0x00400000;
int *KEYBOARD = (int*)0x00402600;
int cursor_x = 4;

void set_pixel(int x, int y) {
    int *ptr;
    int bit_pos;
    ptr = SCREEN + (y << 3) + (y << 1) + (x >> 5);
    bit_pos = ((x >> 3) & 3) * 8 + 7 - (x & 7);
```

```

    *ptr = *ptr | (1 << bit_pos);
}

// Chiffre 3x5 pixels
void draw_digit(int x, int y, int d) {
    if (d != 1 && d != 4) { set_pixel(x, y); set_pixel(x+1, y); set_pixel(x+2,
        ↪ y); }
    if (d == 0 || d == 4 || d == 5 || d == 6 || d == 8 || d == 9) { set_pixel(x,
        ↪ y+1); }
    if (d != 5 && d != 6) { set_pixel(x+2, y+1); }
    if (d != 0 && d != 1 && d != 7) { set_pixel(x, y+2); set_pixel(x+1, y+2);
        ↪ set_pixel(x+2, y+2); }
    if (d == 0 || d == 2 || d == 6 || d == 8) { set_pixel(x, y+3); }
    if (d != 2) { set_pixel(x+2, y+3); }
    if (d != 1 && d != 4 && d != 7) { set_pixel(x, y+4); set_pixel(x+1, y+4);
        ↪ set_pixel(x+2, y+4); }
    if (d == 1) { set_pixel(x+1, y); set_pixel(x+1, y+1); set_pixel(x+1, y+2);
        ↪ set_pixel(x+1, y+3); set_pixel(x+1, y+4); }
    if (d == 7) { set_pixel(x, y); set_pixel(x+1, y); set_pixel(x+2, y); }
}

void draw_plus(int x, int y) {
    set_pixel(x+1, y); set_pixel(x+1, y+1); set_pixel(x+1, y+2);
    set_pixel(x, y+1); set_pixel(x+2, y+1);
}

void draw_minus(int x, int y) { set_pixel(x, y+1); set_pixel(x+1, y+1);
    ↪ set_pixel(x+2, y+1); }
void draw_times(int x, int y) { set_pixel(x,y); set_pixel(x+2,y);
    ↪ set_pixel(x+1,y+1); set_pixel(x,y+2); set_pixel(x+2,y+2); }
void draw_equal(int x, int y) { set_pixel(x,y); set_pixel(x+1,y);
    ↪ set_pixel(x+2,y); set_pixel(x,y+2); set_pixel(x+1,y+2); set_pixel(x+2,y+2);
    ↪ }

int main() {
    int key; int lk; int a; int b; int op; int st; int r; int t; int tens; int u;
    a = 0; b = 0; op = 0; st = 0; lk = 0; t = 0;

    while (t < 50000) {
        key = *KEYBOARD;
        if (key != 0 && key != lk) {
            t = 0;
            if (key == 13 && st == 2) {
                if (op == 1) r = a + b;
                if (op == 2) r = a - b;
                if (op == 3) r = a * b;
                draw_equal(cursor_x, 10); cursor_x = cursor_x + 5;
                tens = 0; u = r;
                while (u >= 10) { u = u - 10; tens = tens + 1; }
                if (tens > 0) { draw_digit(cursor_x, 10, tens); cursor_x =
                    ↪ cursor_x + 5; }
            }
        }
    }
}

```

```

        draw_digit(cursor_x, 10, u);
        return r;
    }
    if (key >= 48 && key <= 57) {
        draw_digit(cursor_x, 10, key - 48);
        cursor_x = cursor_x + 5;
        if (st == 0) { a = key - 48; st = 1; }
        else if (st == 2) { b = key - 48; }
    }
    if (key == 43 && st == 1) { draw_plus(cursor_x, 10); cursor_x =
        ↪ cursor_x + 5; op = 1; st = 2; }
    if (key == 45 && st == 1) { draw_minus(cursor_x, 10); cursor_x =
        ↪ cursor_x + 5; op = 2; st = 2; }
    if (key == 42 && st == 1) { draw_times(cursor_x, 10); cursor_x =
        ↪ cursor_x + 5; op = 3; st = 2; }
    }
    lk = key; t = t + 1;
}
return 0;
}

```

16.5.10 Variables Shell

```

// Variables Shell Interactive - Solution
// Cochez "Capturer clavier", tapez: a=5 b=3 Enter

int *SCREEN = (int*)0x00400000;
int *KEYBOARD = (int*)0x00402600;
int cursor_x = 4;

int var_names[8];
int var_values[8];
int var_count = 0;

void set_pixel(int x, int y) {
    int *ptr;
    int bit_pos;
    ptr = SCREEN + (y << 3) + (y << 1) + (x >> 5);
    bit_pos = ((x >> 3) & 3) * 8 + 7 - (x & 7);
    *ptr = *ptr | (1 << bit_pos);
}

void draw_char(int x, int y, int c) {
    int i;

```

```

// Lettre = ligne verticale + petit trait
if (c >= 97 && c <= 122) {
    for (i = 0; i < 5; i = i + 1) { set_pixel(x, y + i); }
    set_pixel(x + 1, y + 2);
    set_pixel(x + 2, y + (c - 97) % 3);
}

// Chiffre 3x5
if (c >= 48 && c <= 57) {
    int d;
    d = c - 48;
    if (d != 1 && d != 4) { set_pixel(x, y); set_pixel(x+1, y);
        ↪ set_pixel(x+2, y); }
    if (d == 0 || d == 4 || d == 5 || d == 6 || d == 8 || d == 9) {
        ↪ set_pixel(x, y+1); }
    if (d != 5 && d != 6) { set_pixel(x+2, y+1); }
    if (d != 0 && d != 1 && d != 7) { set_pixel(x, y+2); set_pixel(x+1, y+2);
        ↪ set_pixel(x+2, y+2); }
    if (d == 0 || d == 2 || d == 6 || d == 8) { set_pixel(x, y+3); }
    if (d != 2) { set_pixel(x+2, y+3); }
    if (d != 1 && d != 4 && d != 7) { set_pixel(x, y+4); set_pixel(x+1, y+4);
        ↪ set_pixel(x+2, y+4); }
    if (d == 1) { set_pixel(x+1, y); set_pixel(x+1, y+1); set_pixel(x+1,
        ↪ y+2); set_pixel(x+1, y+3); set_pixel(x+1, y+4); }
}

// = sign
if (c == 61) { set_pixel(x,y+1); set_pixel(x+1,y+1); set_pixel(x+2,y+1);
    ↪ set_pixel(x,y+3); set_pixel(x+1,y+3); set_pixel(x+2,y+3); }
// + sign
if (c == 43) { set_pixel(x+1,y); set_pixel(x+1,y+1); set_pixel(x+1,y+2);
    ↪ set_pixel(x,y+1); set_pixel(x+2,y+1); }
}

int find_var(int name) {
    int i;
    for (i = 0; i < var_count; i = i + 1) {
        if (var_names[i] == name) return i;
    }
    return 0 - 1;
}

void set_var(int name, int value) {
    int idx;
    idx = find_var(name);
    if (idx >= 0) { var_values[idx] = value; return; }
    if (var_count < 8) {
        var_names[var_count] = name;
        var_values[var_count] = value;
        var_count = var_count + 1;
    }
}

```

```
}

int get_var(int name) {
    int idx;
    idx = find_var(name);
    if (idx >= 0) return var_values[idx];
    return 0;
}

int main() {
    int key; int lk; int t; int st; int cur_name; int result;
    lk = 0; t = 0; st = 0; cur_name = 0; result = 0;

    while (t < 50000) {
        key = *KEYBOARD;
        if (key != 0 && key != lk) {
            t = 0;
            // Enter = calculer résultat
            if (key == 13) {
                if (var_count >= 2) {
                    result = var_values[0] + var_values[1];
                    draw_char(cursor_x, 10, 61); cursor_x = cursor_x + 5;
                    draw_char(cursor_x, 10, 48 + result); cursor_x = cursor_x + 5;
                }
                return result;
            }
            // Lettre a-z
            if (key >= 97 && key <= 122 && st == 0) {
                draw_char(cursor_x, 10, key); cursor_x = cursor_x + 5;
                cur_name = key;
                st = 1;
            }
            // = après lettre
            if (key == 61 && st == 1) {
                draw_char(cursor_x, 10, 61); cursor_x = cursor_x + 5;
                st = 2;
            }
            // Chiffre après =
            if (key >= 48 && key <= 57 && st == 2) {
                draw_char(cursor_x, 10, key); cursor_x = cursor_x + 5;
                set_var(cur_name, key - 48);
                st = 0;
                cursor_x = cursor_x + 3; // espace
            }
        }
        lk = key; t = t + 1;
    }
    return 0;
}
```


16.5.11 Compte à Rebours

```
// Compte à Rebours Interactif - Solution
// Cochez "Capturer clavier", tapez 3 pour 3 secondes

int *SCREEN = (int*)0x00400000;
int *KEYBOARD = (int*)0x00402600;

void set_pixel(int x, int y) {
    int *ptr;
    int bit_pos;
    ptr = SCREEN + (y << 3) + (y << 1) + (x >> 5);
    bit_pos = ((x >> 3) & 3) * 8 + 7 - (x & 7);
    *ptr = *ptr | (1 << bit_pos);
}

// Dessine chiffre 3x5
void draw_digit(int x, int y, int d, int on) {
    int i; int j; int *ptr; int bit_pos; int px; int py;
    // Efface d'abord la zone
    for (i = 0; i < 4; i = i + 1) {
        for (j = 0; j < 6; j = j + 1) {
            px = x + i; py = y + j;
            ptr = SCREEN + (py << 3) + (py << 1) + (px >> 5);
            bit_pos = ((px >> 3) & 3) * 8 + 7 - (px & 7);
            *ptr = *ptr & (0xFFFFFFFF ^ (1 << bit_pos));
        }
    }
    if (on == 0) return;
    // Dessine le chiffre
    if (d != 1 && d != 4) { set_pixel(x, y); set_pixel(x+1, y); set_pixel(x+2,
        ↪ y); }
    if (d == 0 || d == 4 || d == 5 || d == 6 || d == 8 || d == 9) { set_pixel(x,
        ↪ y+1); }
    if (d != 5 && d != 6) { set_pixel(x+2, y+1); }
    if (d != 0 && d != 1 && d != 7) { set_pixel(x, y+2); set_pixel(x+1, y+2);
        ↪ set_pixel(x+2, y+2); }
    if (d == 0 || d == 2 || d == 6 || d == 8) { set_pixel(x, y+3); }
    if (d != 2) { set_pixel(x+2, y+3); }
    if (d != 1 && d != 4 && d != 7) { set_pixel(x, y+4); set_pixel(x+1, y+4);
        ↪ set_pixel(x+2, y+4); }
    if (d == 1) { set_pixel(x+1, y); set_pixel(x+1, y+1); set_pixel(x+1, y+2);
        ↪ set_pixel(x+1, y+3); set_pixel(x+1, y+4); }
```

```
}

// Dessine barre horizontale
void draw_hbar(int x, int y, int w) {
    int i; int j;
    for (i = 0; i < w; i = i + 1) {
        for (j = 0; j < 4; j = j + 1) {
            set_pixel(x + i, y + j);
        }
    }
}

// Efface colonne de barre (XOR avec 0xFFFFFFFF au lieu de ~)
void clear_col(int x, int y) {
    int j; int *ptr; int bit_pos; int py;
    for (j = 0; j < 4; j = j + 1) {
        py = y + j;
        ptr = SCREEN + (py << 3) + (py << 1) + (x >> 5);
        bit_pos = ((x >> 3) & 3) * 8 + 7 - (x & 7);
        *ptr = *ptr & (0xFFFFFFFF ^ (1 << bit_pos));
    }
}

// Flash rectangle
void flash(int x, int y, int w, int h) {
    int i; int j;
    for (i = 0; i < w; i = i + 1) {
        for (j = 0; j < h; j = j + 1) {
            set_pixel(x + i, y + j);
        }
    }
}

int state = 0;
int secs = 0;
int bar_x = 0;
int step = 0;
int tick = 0;

int main() {
    int key; int lk; int t; int bar_w;
    lk = 0; t = 0;

    while (t < 500000) {
        key = *KEYBOARD;

        if (state == 0) {
            if (key >= 49 && key <= 57 && key != lk) {
                secs = key - 48;
            }
        }
    }
}
```

```
        draw_digit(4, 4, secs, 1);
        bar_w = secs << 3;
        draw_hbar(4, 15, bar_w);
        bar_x = 3 + bar_w;
        step = 0;
        tick = 0;
        state = 1;
    }
}

if (state == 1) {
    tick = tick + 1;
    if (tick >= 800) {
        tick = 0;
        clear_col(bar_x, 15);
        bar_x = bar_x - 1;
        step = step + 1;
        if (step >= 8) {
            step = 0;
            secs = secs - 1;
            draw_digit(4, 4, secs, 1);
            if (secs <= 0) {
                state = 2;
            }
        }
    }
}

if (state == 2) {
    flash(4, 4, 60, 20);
    return 1;
}

lk = key;
t = t + 1;
}
return 0;
}
```

16.5.12 Interruptions

```
// Interruptions Visuelles - Solution
// Cochez "Capturer clavier", appuyez T/K/S puis Enter
```

```
int *SCREEN = (int*)0x00400000;
int *KEYBOARD = (int*)0x00402600;

int irq_count[3];

void set_pixel(int x, int y) {
    int *ptr; int bit_pos;
    ptr = SCREEN + (y << 3) + (y << 1) + (x >> 5);
    bit_pos = ((x >> 3) & 3) * 8 + 7 - (x & 7);
    *ptr = *ptr | (1 << bit_pos);
}

void clear_rect(int x, int y, int w, int h) {
    int i; int j; int px; int py; int *ptr; int bit_pos;
    for (i = 0; i < w; i = i + 1) {
        for (j = 0; j < h; j = j + 1) {
            px = x + i; py = y + j;
            ptr = SCREEN + (py << 3) + (py << 1) + (px >> 5);
            bit_pos = ((px >> 3) & 3) * 8 + 7 - (px & 7);
            *ptr = *ptr & (0xFFFFFFFF ^ (1 << bit_pos));
        }
    }
}

void fill_rect(int x, int y, int w, int h) {
    int i; int j;
    for (i = 0; i < w; i = i + 1) {
        for (j = 0; j < h; j = j + 1) {
            set_pixel(x + i, y + j);
        }
    }
}

void draw_box(int x, int y, int w, int h) {
    int i;
    for (i = 0; i < w; i = i + 1) { set_pixel(x + i, y); set_pixel(x + i, y + h -
    ↵ 1); }
    for (i = 0; i < h; i = i + 1) { set_pixel(x, y + i); set_pixel(x + w - 1, y +
    ↵ i); }
}

void draw_digit(int x, int y, int d) {
    clear_rect(x, y, 4, 6);
    if (d != 1 && d != 4) { set_pixel(x, y); set_pixel(x+1, y); set_pixel(x+2,
    ↵ y); }
    if (d == 0 || d == 4 || d == 5 || d == 6 || d == 8 || d == 9) set_pixel(x,
    ↵ y+1);
    if (d != 5 && d != 6) set_pixel(x+2, y+1);
    if (d != 0 && d != 1 && d != 7) { set_pixel(x, y+2); set_pixel(x+1, y+2);
    ↵ set_pixel(x+2, y+2); }
```

```

    if (d == 0 || d == 2 || d == 6 || d == 8) set_pixel(x, y+3);
    if (d != 2) set_pixel(x+2, y+3);
    if (d != 1 && d != 4 && d != 7) { set_pixel(x, y+4); set_pixel(x+1, y+4);
        ↪ set_pixel(x+2, y+4); }
    if (d == 1) { set_pixel(x+1, y); set_pixel(x+1, y+1); set_pixel(x+1, y+2);
        ↪ set_pixel(x+1, y+3); set_pixel(x+1, y+4); }
}

// Dessine lettre T, K ou S
void draw_letter(int x, int y, int c) {
    if (c == 84) { // T
        set_pixel(x,y); set_pixel(x+1,y); set_pixel(x+2,y);
        set_pixel(x+1,y+1); set_pixel(x+1,y+2); set_pixel(x+1,y+3);
        ↪ set_pixel(x+1,y+4);
    }
    if (c == 75) { // K
        set_pixel(x,y); set_pixel(x,y+1); set_pixel(x,y+2); set_pixel(x,y+3);
        ↪ set_pixel(x,y+4);
        set_pixel(x+2,y); set_pixel(x+1,y+1); set_pixel(x+1,y+3);
        ↪ set_pixel(x+2,y+4);
        set_pixel(x+1,y+2);
    }
    if (c == 83) { // S
        set_pixel(x,y); set_pixel(x+1,y); set_pixel(x+2,y);
        set_pixel(x,y+1);
        set_pixel(x,y+2); set_pixel(x+1,y+2); set_pixel(x+2,y+2);
        set_pixel(x+2,y+3);
        set_pixel(x,y+4); set_pixel(x+1,y+4); set_pixel(x+2,y+4);
    }
}

void draw_device(int idx) {
    int x;
    x = idx * 25 + 4;
    draw_box(x, 4, 20, 15);
    if (idx == 0) draw_letter(x + 8, 6, 84);
    if (idx == 1) draw_letter(x + 8, 6, 75);
    if (idx == 2) draw_letter(x + 8, 6, 83);
    draw_digit(x + 8, 13, irq_count[idx]);
}

void flash_device(int idx) {
    int x; int i;
    x = idx * 25 + 4;
    fill_rect(x + 1, 5, 18, 13);
    // Petit delai
    for (i = 0; i < 500; i = i + 1) { }
}

```

```
void irq_handler(int type) {
    if (type >= 0 && type < 3) {
        irq_count[type] = irq_count[type] + 1;
        flash_device(type);
        draw_device(type);
    }
}

int main() {
    int key; int lk; int t; int total;
    lk = 0; t = 0;
    irq_count[0] = 0; irq_count[1] = 0; irq_count[2] = 0;

    draw_device(0);
    draw_device(1);
    draw_device(2);

    while (t < 100000) {
        key = *KEYBOARD;
        if (key != 0 && key != lk) {
            t = 0;
            if (key == 13) {
                total = irq_count[0] + irq_count[1] + irq_count[2];
                return total;
            }
            if (key == 116 || key == 84) irq_handler(0);
            if (key == 107 || key == 75) irq_handler(1);
            if (key == 115 || key == 83) irq_handler(2);
        }
        lk = key;
        t = t + 1;
    }
    return 0;
}
```

16.5.13 Coroutines

```
// Coroutines Visuelles - Solution
// Cochez "Capturer clavier", appuyez Espace pour chaque step

int *SCREEN = (int*)0x00400000;
int *KEYBOARD = (int*)0x00402600;

int current_task = 0;
```

```
int task_a_val = 0;
int task_b_val = 0;
int steps = 0;

void set_pixel(int x, int y) {
    int *ptr; int bit_pos;
    ptr = SCREEN + (y << 3) + (y << 1) + (x >> 5);
    bit_pos = ((x >> 3) & 3) * 8 + 7 - (x & 7);
    *ptr = *ptr | (1 << bit_pos);
}

void clear_rect(int x, int y, int w, int h) {
    int i; int j; int px; int py; int *ptr; int bit_pos;
    for (i = 0; i < w; i = i + 1) {
        for (j = 0; j < h; j = j + 1) {
            px = x + i; py = y + j;
            ptr = SCREEN + (py << 3) + (py << 1) + (px >> 5);
            bit_pos = ((px >> 3) & 3) * 8 + 7 - (px & 7);
            *ptr = *ptr & (0xFFFFFFFF ^ (1 << bit_pos));
        }
    }
}

void fill_rect(int x, int y, int w, int h) {
    int i; int j;
    for (i = 0; i < w; i = i + 1) {
        for (j = 0; j < h; j = j + 1) {
            set_pixel(x + i, y + j);
        }
    }
}

void draw_box(int x, int y, int w, int h) {
    int i;
    for (i = 0; i < w; i = i + 1) { set_pixel(x + i, y); set_pixel(x + i, y + h - 1); }
    for (i = 0; i < h; i = i + 1) { set_pixel(x, y + i); set_pixel(x + w - 1, y + i); }
}

void draw_digit(int x, int y, int d) {
    clear_rect(x, y, 4, 6);
    if (d != 1 && d != 4) { set_pixel(x, y); set_pixel(x+1, y); set_pixel(x+2, y); }
    if (d == 0 || d == 4 || d == 5 || d == 6 || d == 8 || d == 9) set_pixel(x, y+1);
    if (d != 5 && d != 6) set_pixel(x+2, y+1);
    if (d != 0 && d != 1 && d != 7) { set_pixel(x, y+2); set_pixel(x+1, y+2); set_pixel(x+2, y+2); }
```

```
if (d == 0 || d == 2 || d == 6 || d == 8) set_pixel(x, y+3);
if (d != 2) set_pixel(x+2, y+3);
if (d != 1 && d != 4 && d != 7) { set_pixel(x, y+4); set_pixel(x+1, y+4);
    ↪ set_pixel(x+2, y+4); }
if (d == 1) { set_pixel(x+1, y); set_pixel(x+1, y+1); set_pixel(x+1, y+2);
    ↪ set_pixel(x+1, y+3); set_pixel(x+1, y+4); }
}

void draw_letter_A(int x, int y) {
    set_pixel(x+1, y);
    set_pixel(x, y+1); set_pixel(x+2, y+1);
    set_pixel(x, y+2); set_pixel(x+1, y+2); set_pixel(x+2, y+2);
    set_pixel(x, y+3); set_pixel(x+2, y+3);
    set_pixel(x, y+4); set_pixel(x+2, y+4);
}

void draw_letter_B(int x, int y) {
    set_pixel(x, y); set_pixel(x+1, y);
    set_pixel(x, y+1); set_pixel(x+2, y+1);
    set_pixel(x, y+2); set_pixel(x+1, y+2);
    set_pixel(x, y+3); set_pixel(x+2, y+3);
    set_pixel(x, y+4); set_pixel(x+1, y+4);
}

void draw_arrow(int x, int y) {
    set_pixel(x, y+2);
    set_pixel(x+1, y+1); set_pixel(x+1, y+2); set_pixel(x+1, y+3);
    set_pixel(x+2, y); set_pixel(x+2, y+2); set_pixel(x+2, y+4);
    set_pixel(x+3, y+1); set_pixel(x+3, y+2); set_pixel(x+3, y+3);
    set_pixel(x+4, y+2);
}

void draw_task(int idx, int active) {
    int x;
    x = idx * 35 + 4;
    clear_rect(x, 4, 30, 20);
    if (active) {
        fill_rect(x, 4, 30, 20);
        // Dessiner en inverse (effacer les pixels pour la lettre/chiffre)
        clear_rect(x + 12, 7, 6, 6);
        clear_rect(x + 12, 15, 5, 6);
    } else {
        draw_box(x, 4, 30, 20);
    }
    if (idx == 0) {
        if (active) clear_rect(x + 13, 7, 4, 5);
        else draw_letter_A(x + 13, 7);
        draw_digit(x + 13, 15, task_a_val);
    } else {
```



```
        if (active) clear_rect(x + 13, 7, 4, 5);
        else draw_letter_B(x + 13, 7);
        draw_digit(x + 13, 15, task_b_val);
    }
}

void draw_all() {
    draw_task(0, current_task == 0);
    draw_task(1, current_task == 1);
    // Fleche entre les taches
    if (current_task == 0) {
        clear_rect(32, 12, 8, 6);
        draw_arrow(32, 12);
    } else {
        clear_rect(32, 12, 8, 6);
        // Fleche inversee
        set_pixel(36, 14);
        set_pixel(35, 13); set_pixel(35, 14); set_pixel(35, 15);
        set_pixel(34, 12); set_pixel(34, 14); set_pixel(34, 16);
        set_pixel(33, 13); set_pixel(33, 14); set_pixel(33, 15);
        set_pixel(32, 14);
    }
}

int step() {
    if (current_task == 0) {
        task_a_val = task_a_val + 1;
        if (task_a_val > 3) return 0;
        current_task = 1;
    } else {
        task_b_val = task_b_val + 1;
        if (task_b_val > 3) return 0;
        current_task = 0;
    }
    steps = steps + 1;
    return 1;
}

int main() {
    int key; int lk; int t; int running;
    lk = 0; t = 0; running = 1;

    draw_all();

    while (t < 100000 && running) {
        key = *KEYBOARD;
        if (key != 0 && key != lk) {
            t = 0;
            if (key == 32) {
```

```

        running = step();
        draw_all();
    }
    if (key == 13) {
        return steps;
    }
}
lk = key;
t = t + 1;
}
// Flash final
fill_rect(0, 0, 80, 30);
return steps;
}

```

16.5.14 Scheduler

```

// Scheduler Round-Robin Visuel - Solution
// Cochez "Capturer clavier", Espace pour chaque tick

int *SCREEN = (int*)0x00400000;
int *KEYBOARD = (int*)0x00402600;

int proc_time[3];
int proc_state[3];
int current_proc = 0;
int quantum_left = 2;
int switches = 0;
int ticks = 0;

void set_pixel(int x, int y) {
    int *ptr; int bit_pos;
    ptr = SCREEN + (y << 3) + (y << 1) + (x >> 5);
    bit_pos = ((x >> 3) & 3) * 8 + 7 - (x & 7);
    *ptr = *ptr | (1 << bit_pos);
}

void clear_rect(int x, int y, int w, int h) {
    int i; int j; int px; int py; int *ptr; int bit_pos;
    for (i = 0; i < w; i = i + 1) {
        for (j = 0; j < h; j = j + 1) {
            px = x + i; py = y + j;
            ptr = SCREEN + (py << 3) + (py << 1) + (px >> 5);
            bit_pos = ((px >> 3) & 3) * 8 + 7 - (px & 7);

```

```
        *ptr = *ptr & (0xFFFFFFFF ^ (1 << bit_pos));
    }
}

void fill_rect(int x, int y, int w, int h) {
    int i; int j;
    for (i = 0; i < w; i = i + 1) {
        for (j = 0; j < h; j = j + 1) {
            set_pixel(x + i, y + j);
        }
    }
}

void draw_box(int x, int y, int w, int h) {
    int i;
    for (i = 0; i < w; i = i + 1) { set_pixel(x + i, y); set_pixel(x + i, y + h - 1); }
    for (i = 0; i < h; i = i + 1) { set_pixel(x, y + i); set_pixel(x + w - 1, y + i); }
}

void draw_digit(int x, int y, int d) {
    clear_rect(x, y, 4, 6);
    if (d != 1 && d != 4) { set_pixel(x, y); set_pixel(x+1, y); set_pixel(x+2, y); }
    if (d == 0 || d == 4 || d == 5 || d == 6 || d == 8 || d == 9) set_pixel(x, y+1);
    if (d != 5 && d != 6) set_pixel(x+2, y+1);
    if (d != 0 && d != 1 && d != 7) { set_pixel(x, y+2); set_pixel(x+1, y+2); set_pixel(x+2, y+2); }
    if (d == 0 || d == 2 || d == 6 || d == 8) set_pixel(x, y+3);
    if (d != 2) set_pixel(x+2, y+3);
    if (d != 1 && d != 4 && d != 7) { set_pixel(x, y+4); set_pixel(x+1, y+4); set_pixel(x+2, y+4); }
    if (d == 1) { set_pixel(x+1, y); set_pixel(x+1, y+1); set_pixel(x+1, y+2); set_pixel(x+1, y+3); set_pixel(x+1, y+4); }
}

void draw_P(int x, int y) {
    set_pixel(x, y); set_pixel(x+1, y); set_pixel(x+2, y);
    set_pixel(x, y+1); set_pixel(x+2, y+1);
    set_pixel(x, y+2); set_pixel(x+1, y+2);
    set_pixel(x, y+3); set_pixel(x, y+4);
}

void draw_proc(int idx, int active) {
    int x; int y; int t; int i;
    x = idx * 32 + 4;
```

```
y = 4;

clear_rect(x, y, 28, 24);

if (active) {
    fill_rect(x, y, 28, 24);
    clear_rect(x + 2, y + 2, 24, 20);
} else {
    draw_box(x, y, 28, 24);
}

// P et numero
draw_P(x + 4, y + 4);
draw_digit(x + 10, y + 4, idx);

// Barre de temps restant
t = proc_time[idx];
if (t > 0) {
    for (i = 0; i < t; i = i + 1) {
        fill_rect(x + 4 + i * 5, y + 14, 4, 6);
    }
}

// X si termine
if (proc_state[idx] == 2) {
    set_pixel(x + 8, y + 14); set_pixel(x + 12, y + 14);
    set_pixel(x + 9, y + 15); set_pixel(x + 11, y + 15);
    set_pixel(x + 10, y + 16);
    set_pixel(x + 9, y + 17); set_pixel(x + 11, y + 17);
    set_pixel(x + 8, y + 18); set_pixel(x + 12, y + 18);
}
}

void draw_all() {
    int i;
    for (i = 0; i < 3; i = i + 1) {
        draw_proc(i, i == current_proc && proc_state[i] != 2);
    }

    // Quantum: Q=N
    clear_rect(4, 32, 20, 6);
    // Q
    set_pixel(5, 32); set_pixel(6, 32); set_pixel(7, 32);
    set_pixel(4, 33); set_pixel(8, 33);
    set_pixel(4, 34); set_pixel(8, 34);
    set_pixel(4, 35); set_pixel(6, 35); set_pixel(8, 35);
    set_pixel(5, 36); set_pixel(6, 36); set_pixel(8, 36);
    draw_digit(12, 32, quantum_left);
}
```

```
// Switches: SW=N
clear_rect(30, 32, 30, 6);
// S
set_pixel(31, 32); set_pixel(32, 32); set_pixel(33, 32);
set_pixel(30, 33);
set_pixel(31, 34); set_pixel(32, 34);
set_pixel(33, 35);
set_pixel(30, 36); set_pixel(31, 36); set_pixel(32, 36);
// W
set_pixel(36, 32); set_pixel(40, 32);
set_pixel(36, 33); set_pixel(40, 33);
set_pixel(36, 34); set_pixel(38, 34); set_pixel(40, 34);
set_pixel(36, 35); set_pixel(38, 35); set_pixel(40, 35);
set_pixel(37, 36); set_pixel(39, 36);
draw_digit(44, 32, switches);

// Ticks: T=N
clear_rect(60, 32, 20, 6);
// T
set_pixel(60, 32); set_pixel(61, 32); set_pixel(62, 32);
set_pixel(61, 33); set_pixel(61, 34); set_pixel(61, 35); set_pixel(61, 36);
draw_digit(66, 32, ticks);
}

int find_next(int from) {
    int i; int next;
    for (i = 1; i <= 3; i = i + 1) {
        next = (from + i) % 3;
        if (proc_state[next] == 0 && proc_time[next] > 0) return next;
    }
    return from;
}

int tick() {
    int next; int all_done;

    // Verifier si tous termines
    all_done = 1;
    if (proc_state[0] != 2) all_done = 0;
    if (proc_state[1] != 2) all_done = 0;
    if (proc_state[2] != 2) all_done = 0;
    if (all_done) return 0;

    ticks = ticks + 1;

    // Executer processus courant
    if (proc_time[current_proc] > 0) {
        proc_time[current_proc] = proc_time[current_proc] - 1;
        quantum_left = quantum_left - 1;
    }
}
```

```
        if (proc_time[current_proc] == 0) {
            proc_state[current_proc] = 2;
            quantum_left = 0;
        }
    }

    // Context switch si quantum epuise
    if (quantum_left <= 0) {
        next = find_next(current_proc);
        if (next != current_proc) {
            current_proc = next;
            switches = switches + 1;
        }
        quantum_left = 2;
    }

    return 1;
}

int main() {
    int key; int lk; int t; int running;

    proc_time[0] = 3; proc_time[1] = 2; proc_time[2] = 4;
    proc_state[0] = 0; proc_state[1] = 0; proc_state[2] = 0;

    lk = 0; t = 0; running = 1;
    draw_all();

    while (t < 100000) {
        key = *KEYBOARD;
        if (key != 0 && key != lk) {
            t = 0;
            if (key == 32) {
                running = tick();
                draw_all();
            }
            if (key == 13) {
                return switches;
            }
        }
        lk = key;
        t = t + 1;
    }

    return switches;
}
```

16.5.15 Projet 1: Mini-OS Shell

```
// Mini-OS Shell - Solution
// Cochez "Capturer clavier"

int *SCREEN = (int*)0x00400000;
int *KEYBOARD = (int*)0x00402600;

void set_pixel(int x, int y) {
    int *ptr; int bit_pos;
    ptr = SCREEN + (y << 3) + (y << 1) + (x >> 5);
    bit_pos = ((x >> 3) & 3) * 8 + 7 - (x & 7);
    *ptr = *ptr | (1 << bit_pos);
}

void clear_screen() {
    int i; int *ptr;
    ptr = SCREEN;
    for (i = 0; i < 2400; i = i + 1) {
        *ptr = 0;
        ptr = ptr + 1;
    }
}

void draw_digit(int x, int y, int d) {
    if (d != 1 && d != 4) { set_pixel(x, y); set_pixel(x+1, y); set_pixel(x+2,
        ↪ y); }
    if (d == 0 || d == 4 || d == 5 || d == 6 || d == 8 || d == 9) set_pixel(x,
        ↪ y+1);
    if (d != 5 && d != 6) set_pixel(x+2, y+1);
    if (d != 0 && d != 1 && d != 7) { set_pixel(x, y+2); set_pixel(x+1, y+2);
        ↪ set_pixel(x+2, y+2); }
    if (d == 0 || d == 2 || d == 6 || d == 8) set_pixel(x, y+3);
    if (d != 2) set_pixel(x+2, y+3);
    if (d != 1 && d != 4 && d != 7) { set_pixel(x, y+4); set_pixel(x+1, y+4);
        ↪ set_pixel(x+2, y+4); }
    if (d == 1) { set_pixel(x+1, y); set_pixel(x+1, y+1); set_pixel(x+1, y+2);
        ↪ set_pixel(x+1, y+3); set_pixel(x+1, y+4); }
}

void draw_plus(int x, int y) {
    set_pixel(x+1, y); set_pixel(x, y+1); set_pixel(x+1, y+1); set_pixel(x+2,
    ↪ y+1); set_pixel(x+1, y+2);
}
```

```
void draw_equal(int x, int y) {
    set_pixel(x, y); set_pixel(x+1, y); set_pixel(x+2, y);
    set_pixel(x, y+2); set_pixel(x+1, y+2); set_pixel(x+2, y+2);
}

void draw_H(int x, int y) {
    set_pixel(x, y); set_pixel(x, y+1); set_pixel(x, y+2); set_pixel(x, y+3);
    ↪ set_pixel(x, y+4);
    set_pixel(x+1, y+2);
    set_pixel(x+2, y); set_pixel(x+2, y+1); set_pixel(x+2, y+2); set_pixel(x+2,
    ↪ y+3); set_pixel(x+2, y+4);
}

void draw_I(int x, int y) {
    set_pixel(x, y); set_pixel(x+1, y); set_pixel(x+2, y);
    set_pixel(x+1, y+1); set_pixel(x+1, y+2); set_pixel(x+1, y+3);
    set_pixel(x, y+4); set_pixel(x+1, y+4); set_pixel(x+2, y+4);
}

void draw_box(int x, int y, int w, int h) {
    int i;
    for (i = 0; i < w; i = i + 1) { set_pixel(x + i, y); set_pixel(x + i, y + h -
    ↪ 1); }
    for (i = 0; i < h; i = i + 1) { set_pixel(x, y + i); set_pixel(x + w - 1, y +
    ↪ i); }
}

void draw_menu() {
    // Titre: MENU
    // M
    set_pixel(4, 4); set_pixel(4, 5); set_pixel(4, 6); set_pixel(4, 7);
    ↪ set_pixel(4, 8);
    set_pixel(5, 5); set_pixel(6, 6); set_pixel(7, 5);
    set_pixel(8, 4); set_pixel(8, 5); set_pixel(8, 6); set_pixel(8, 7);
    ↪ set_pixel(8, 8);
    // E
    set_pixel(11, 4); set_pixel(12, 4); set_pixel(13, 4);
    set_pixel(11, 5); set_pixel(11, 6); set_pixel(12, 6); set_pixel(11, 7);
    set_pixel(11, 8); set_pixel(12, 8); set_pixel(13, 8);
    // N
    set_pixel(16, 4); set_pixel(16, 5); set_pixel(16, 6); set_pixel(16, 7);
    ↪ set_pixel(16, 8);
    set_pixel(17, 5); set_pixel(18, 6); set_pixel(19, 7);
    set_pixel(20, 4); set_pixel(20, 5); set_pixel(20, 6); set_pixel(20, 7);
    ↪ set_pixel(20, 8);
    // U
    set_pixel(23, 4); set_pixel(23, 5); set_pixel(23, 6); set_pixel(23, 7);
    set_pixel(24, 8); set_pixel(25, 8);
    set_pixel(26, 4); set_pixel(26, 5); set_pixel(26, 6); set_pixel(26, 7);
}
```



```
// Option 1: CALC
draw_box(4, 14, 30, 12);
draw_digit(8, 17, 1);
// C
set_pixel(15, 17); set_pixel(16, 17); set_pixel(17, 17);
set_pixel(14, 18); set_pixel(14, 19); set_pixel(14, 20);
set_pixel(15, 21); set_pixel(16, 21); set_pixel(17, 21);

// Option 2: COUNT
draw_box(4, 28, 30, 12);
draw_digit(8, 31, 2);
// #
set_pixel(15, 31); set_pixel(17, 31);
set_pixel(14, 32); set_pixel(15, 32); set_pixel(16, 32); set_pixel(17, 32);
↪ set_pixel(18, 32);
set_pixel(15, 33); set_pixel(17, 33);
set_pixel(14, 34); set_pixel(15, 34); set_pixel(16, 34); set_pixel(17, 34);
↪ set_pixel(18, 34);
set_pixel(15, 35); set_pixel(17, 35);

// Option 3: MSG
draw_box(4, 42, 30, 12);
draw_digit(8, 45, 3);
draw_H(15, 45);
draw_I(20, 45);

// Option 0: QUIT
draw_box(4, 56, 30, 12);
draw_digit(8, 59, 0);
// X
set_pixel(15, 59); set_pixel(19, 59);
set_pixel(16, 60); set_pixel(18, 60);
set_pixel(17, 61);
set_pixel(16, 62); set_pixel(18, 62);
set_pixel(15, 63); set_pixel(19, 63);
}

void delay() {
    int i;
    for (i = 0; i < 50000; i = i + 1) { }
}

void wait_key() {
    int k;
    while (1) {
        k = *KEYBOARD;
        if (k != 0) return;
    }
}
```

```
}

void app_calc() {
    clear_screen();
    // Titre
    // C
    set_pixel(5, 4); set_pixel(6, 4); set_pixel(7, 4);
    set_pixel(4, 5); set_pixel(4, 6); set_pixel(4, 7);
    set_pixel(5, 8); set_pixel(6, 8); set_pixel(7, 8);
    // A
    set_pixel(11, 4); set_pixel(10, 5); set_pixel(12, 5);
    set_pixel(10, 6); set_pixel(11, 6); set_pixel(12, 6);
    set_pixel(10, 7); set_pixel(12, 7); set_pixel(10, 8); set_pixel(12, 8);
    // L
    set_pixel(15, 4); set_pixel(15, 5); set_pixel(15, 6); set_pixel(15, 7);
    set_pixel(15, 8); set_pixel(16, 8); set_pixel(17, 8);
    // C
    set_pixel(21, 4); set_pixel(22, 4); set_pixel(23, 4);
    set_pixel(20, 5); set_pixel(20, 6); set_pixel(20, 7);
    set_pixel(21, 8); set_pixel(22, 8); set_pixel(23, 8);

    // 3 + 5 = 8
    draw_digit(10, 20, 3);
    draw_plus(16, 20);
    draw_digit(22, 20, 5);
    draw_equal(28, 20);
    draw_digit(34, 20, 8);

    wait_key();
}

void app_count() {
    int i;
    clear_screen();
    // Titre: COUNT
    // #
    set_pixel(5, 4); set_pixel(7, 4);
    set_pixel(4, 5); set_pixel(5, 5); set_pixel(6, 5); set_pixel(7, 5);
    ↪ set_pixel(8, 5);
    set_pixel(5, 6); set_pixel(7, 6);
    set_pixel(4, 7); set_pixel(5, 7); set_pixel(6, 7); set_pixel(7, 7);
    ↪ set_pixel(8, 7);
    set_pixel(5, 8); set_pixel(7, 8);

    for (i = 0; i < 6; i = i + 1) {
        draw_digit(10 + i * 6, 20, i);
        delay();
    }
}
```

```
    wait_key();
}

void app_msg() {
    clear_screen();
    // Titre: MSG
    // M
    set_pixel(4, 4); set_pixel(4, 5); set_pixel(4, 6); set_pixel(4, 7);
    ↪ set_pixel(4, 8);
    set_pixel(5, 5); set_pixel(6, 6); set_pixel(7, 5);
    set_pixel(8, 4); set_pixel(8, 5); set_pixel(8, 6); set_pixel(8, 7);
    ↪ set_pixel(8, 8);
    // S
    set_pixel(12, 4); set_pixel(13, 4); set_pixel(11, 5);
    set_pixel(12, 6); set_pixel(13, 7);
    set_pixel(11, 8); set_pixel(12, 8);
    // G
    set_pixel(17, 4); set_pixel(18, 4); set_pixel(19, 4);
    set_pixel(16, 5); set_pixel(16, 6); set_pixel(18, 6); set_pixel(19, 6);
    set_pixel(16, 7); set_pixel(19, 7);
    set_pixel(17, 8); set_pixel(18, 8); set_pixel(19, 8);

    // Grand HI
    // H
    set_pixel(10, 18); set_pixel(10, 19); set_pixel(10, 20); set_pixel(10, 21);
    ↪ set_pixel(10, 22);
    set_pixel(10, 23); set_pixel(10, 24); set_pixel(10, 25); set_pixel(10, 26);
    ↪ set_pixel(10, 27);
    set_pixel(11, 22); set_pixel(12, 22); set_pixel(13, 22);
    set_pixel(14, 18); set_pixel(14, 19); set_pixel(14, 20); set_pixel(14, 21);
    ↪ set_pixel(14, 22);
    set_pixel(14, 23); set_pixel(14, 24); set_pixel(14, 25); set_pixel(14, 26);
    ↪ set_pixel(14, 27);
    // I
    set_pixel(18, 18); set_pixel(19, 18); set_pixel(20, 18); set_pixel(21, 18);
    ↪ set_pixel(22, 18);
    set_pixel(20, 19); set_pixel(20, 20); set_pixel(20, 21); set_pixel(20, 22);
    set_pixel(20, 23); set_pixel(20, 24); set_pixel(20, 25); set_pixel(20, 26);
    set_pixel(18, 27); set_pixel(19, 27); set_pixel(20, 27); set_pixel(21, 27);
    ↪ set_pixel(22, 27);

    wait_key();
}

int main() {
    int key; int lk; int running;
    running = 1;
    lk = 0;
```

```
while (running) {
    clear_screen();
    draw_menu();

    while (1) {
        key = *KEYBOARD;
        if (key != 0 && key != lk) {
            if (key == 49) { app_calc(); break; }
            if (key == 50) { app_count(); break; }
            if (key == 51) { app_msg(); break; }
            if (key == 48) { running = 0; break; }
        }
        lk = key;
    }
}

clear_screen();
return 0;
}
```

16.5.16 Projet 2: Task Manager

```
// Gestionnaire de Tâches - Solution
// Cochez "Capturer clavier"

int *SCREEN = (int*)0x00400000;
int *KEYBOARD = (int*)0x00402600;

int proc_work[4];
int proc_state[4];
int proc_done[4];
int current = 0;
int quantum_left = 2;
int switches = 0;
int ticks = 0;
int auto_run = 0;

void set_pixel(int x, int y) {
    int *ptr; int bit_pos;
    ptr = SCREEN + (y << 3) + (y << 1) + (x >> 5);
    bit_pos = ((x >> 3) & 3) * 8 + 7 - (x & 7);
    *ptr = *ptr | (1 << bit_pos);
}
```

```
void clear_rect(int x, int y, int w, int h) {
    int i; int j; int px; int py; int *ptr; int bit_pos;
    for (i = 0; i < w; i = i + 1) {
        for (j = 0; j < h; j = j + 1) {
            px = x + i; py = y + j;
            ptr = SCREEN + (py << 3) + (py << 1) + (px >> 5);
            bit_pos = ((px >> 3) & 3) * 8 + 7 - (px & 7);
            *ptr = *ptr & (0xFFFFFFFF ^ (1 << bit_pos));
        }
    }
}

void fill_rect(int x, int y, int w, int h) {
    int i; int j;
    for (i = 0; i < w; i = i + 1) {
        for (j = 0; j < h; j = j + 1) {
            set_pixel(x + i, y + j);
        }
    }
}

void draw_box(int x, int y, int w, int h) {
    int i;
    for (i = 0; i < w; i = i + 1) { set_pixel(x + i, y); set_pixel(x + i, y + h - 1); }
    for (i = 0; i < h; i = i + 1) { set_pixel(x, y + i); set_pixel(x + w - 1, y + i); }
}

void draw_digit(int x, int y, int d) {
    clear_rect(x, y, 4, 6);
    if (d != 1 && d != 4) { set_pixel(x, y); set_pixel(x+1, y); set_pixel(x+2, y); }
    if (d == 0 || d == 4 || d == 5 || d == 6 || d == 8 || d == 9) set_pixel(x, y+1);
    if (d != 5 && d != 6) set_pixel(x+2, y+1);
    if (d != 0 && d != 1 && d != 7) { set_pixel(x, y+2); set_pixel(x+1, y+2); set_pixel(x+2, y+2); }
    if (d == 0 || d == 2 || d == 6 || d == 8) set_pixel(x, y+3);
    if (d != 2) set_pixel(x+2, y+3);
    if (d != 1 && d != 4 && d != 7) { set_pixel(x, y+4); set_pixel(x+1, y+4); set_pixel(x+2, y+4); }
    if (d == 1) { set_pixel(x+1, y); set_pixel(x+1, y+1); set_pixel(x+1, y+2); set_pixel(x+1, y+3); set_pixel(x+1, y+4); }
}

void draw_proc(int idx) {
    int x; int w; int i; int active;
    x = idx * 40 + 4;
```

```
clear_rect(x, 4, 36, 55);

active = (idx == current && proc_state[idx] == 1);

// Cadre - double si running
draw_box(x, 4, 36, 55);
if (active) {
    draw_box(x + 2, 6, 32, 51);
}

// P et numero (grand)
fill_rect(x + 8, 10, 2, 9);
fill_rect(x + 10, 10, 4, 2);
fill_rect(x + 14, 10, 2, 5);
fill_rect(x + 10, 14, 4, 2);

draw_digit(x + 20, 12, idx);

// Etat: R=Ready *=Run B=Block X=Done
clear_rect(x + 10, 24, 16, 10);
if (proc_state[idx] == 0) {
    // R
    fill_rect(x + 12, 25, 2, 8);
    fill_rect(x + 14, 25, 4, 2);
    fill_rect(x + 18, 25, 2, 4);
    fill_rect(x + 14, 28, 4, 2);
    fill_rect(x + 16, 30, 2, 3);
}
if (proc_state[idx] == 1) {
    // * etoile
    fill_rect(x + 14, 26, 2, 6);
    fill_rect(x + 12, 28, 6, 2);
}
if (proc_state[idx] == 2) {
    // B
    fill_rect(x + 12, 25, 2, 8);
    fill_rect(x + 14, 25, 4, 2);
    fill_rect(x + 14, 28, 4, 2);
    fill_rect(x + 14, 31, 4, 2);
    fill_rect(x + 18, 26, 2, 2);
    fill_rect(x + 18, 29, 2, 2);
}
if (proc_state[idx] == 3) {
    // X
    fill_rect(x + 12, 25, 2, 2); fill_rect(x + 18, 25, 2, 2);
    fill_rect(x + 14, 27, 2, 2); fill_rect(x + 16, 27, 2, 2);
    fill_rect(x + 14, 29, 4, 2);
    fill_rect(x + 12, 31, 2, 2); fill_rect(x + 18, 31, 2, 2);
}
```

```
}

// Barre travail restant
w = proc_work[idx];
for (i = 0; i < 4; i = i + 1) {
    if (i < w) {
        fill_rect(x + 6 + i * 7, 38, 5, 6);
    } else {
        draw_box(x + 6 + i * 7, 38, 5, 6);
    }
}

// Compteur
draw_digit(x + 15, 50, proc_done[idx] % 10);
}

void draw_info() {
    clear_rect(4, 62, 156, 8);
    // Q:
    fill_rect(6, 63, 2, 5); fill_rect(8, 63, 3, 2); fill_rect(11, 63, 2, 5);
    fill_rect(8, 66, 3, 2); fill_rect(10, 67, 3, 2);
    draw_digit(16, 63, quantum_left);
    // S:
    fill_rect(30, 63, 5, 2); fill_rect(28, 65, 2, 2);
    fill_rect(30, 66, 3, 2); fill_rect(33, 67, 2, 2);
    fill_rect(28, 68, 5, 2);
    draw_digit(38, 63, switches % 10);
    // T:
    fill_rect(52, 63, 7, 2); fill_rect(54, 65, 3, 4);
    draw_digit(62, 63, ticks % 10);
}

int find_next(int from) {
    int i; int next;
    for (i = 1; i <= 4; i = i + 1) {
        next = (from + i) % 4;
        if (proc_state[next] == 0 && proc_work[next] > 0) return next;
    }
    return from;
}

int all_done() {
    int i;
    for (i = 0; i < 4; i = i + 1) {
        if (proc_state[i] != 3 && proc_state[i] != 2) {
            if (proc_work[i] > 0) return 0;
        }
    }
    return 1;
}
```

```
}

void do_tick() {
    int next;
    if (all_done()) return;
    ticks = ticks + 1;
    if (proc_state[current] == 0) proc_state[current] = 1;
    if (proc_state[current] == 1 && proc_work[current] > 0) {
        proc_work[current] = proc_work[current] - 1;
        proc_done[current] = proc_done[current] + 1;
        quantum_left = quantum_left - 1;
        if (proc_work[current] == 0) {
            proc_state[current] = 3;
            quantum_left = 0;
        }
    }
    if (quantum_left <= 0 || proc_state[current] == 2 || proc_state[current] ==
        3) {
        if (proc_state[current] == 1) proc_state[current] = 0;
        next = find_next(current);
        if (next != current) {
            current = next;
            switches = switches + 1;
        }
        quantum_left = 2;
    }
}

void toggle_block(int idx) {
    if (proc_state[idx] == 0 || proc_state[idx] == 1) {
        proc_state[idx] = 2;
        if (idx == current) quantum_left = 0;
    } else if (proc_state[idx] == 2) {
        proc_state[idx] = 0;
    }
}

void draw_all() {
    int i;
    for (i = 0; i < 4; i = i + 1) draw_proc(i);
    draw_info();
}

int main() {
    int key; int lk; int delay;
    proc_work[0] = 3; proc_work[1] = 2; proc_work[2] = 4; proc_work[3] = 3;
    proc_state[0] = 0; proc_state[1] = 0; proc_state[2] = 0; proc_state[3] = 0;
    proc_done[0] = 0; proc_done[1] = 0; proc_done[2] = 0; proc_done[3] = 0;
    lk = 0; delay = 0;
```



```
draw_all();
while (1) {
    key = *KEYBOARD;
    if (key != 0 && key != lk) {
        if (key == 32) { do_tick(); draw_all(); }
        if (key == 65) { auto_run = 1 - auto_run; }
        if (key == 48) { toggle_block(0); draw_all(); }
        if (key == 49) { toggle_block(1); draw_all(); }
        if (key == 50) { toggle_block(2); draw_all(); }
        if (key == 51) { toggle_block(3); draw_all(); }
        if (key == 13) return switches;
    }
    lk = key;
    if (auto_run) {
        delay = delay + 1;
        if (delay > 3000) { delay = 0; do_tick(); draw_all(); }
    }
}
return switches;
}
```