

Cours ECE_4ES01_TA

2026

Ver. 2026.v2

Architecture des microprocesseurs

Recueil de travaux dirigés - Travaux pratiques

Omar Hammami

École Nationale Supérieure
de **Techniques Avancées**

ENST2



INSTITUT
POLYTECHNIQUE
DE PARIS

Table des matières

<i>TD/TP1 : Evaluation de performances - programmation assembleur RISC-V</i>	<i>7</i>
Exercice 1 : Evaluation de performances analytique - Loi d'Amdahl.....	7
Exercice 2 : Evaluation de performances analytique - Loi d'Amdahl.....	7
Exercice 3 : Evaluation de performances – simulation.....	8
Exercice 4 : Commentaire d'un programme en assembleur RISC-V.....	9
Exercice 5 : Programmation en assembleur	9
Exercice 6 : Encodage en assembleur	9
Exercice 7 : Commentaire du programme en assembleur	10
Exercice 8 : Programmation en assembleur (somme d'un tableau)	10
<i>TD/TP2 : Programmation assembleur RISC-V - profiling d'applications et ISS (instruction set simulator)</i>	<i>16</i>
Exercice 1 : Switch statement.....	16
Exercice 2 : Boucles	16
Exercice 3 : Appels de fonctions	16
Exercice 4 : Appels de fonctions	17
Exercice 5 : Profiling d'applications (% d'instructions par catégorie) par simulation sur Gem5	18
<i>TD/TP3 : Processeur pipeline – superscalaire</i>	<i>20</i>
Exercice 1 : Pipeline.....	20
Exercice 2 : Pipeline - Data Forwarding	21
Exercice 3 : Impact du nombre d'unités fonctionnelles	22
Exercice 4 : Exécution dans l'ordre (in-order) VS exécution dans le désordre (out-of-order).....	23
<i>TD/TP4 : Microprocesseur superscalaire/mémoires caches - Analyse de configurations d'architectures de microprocesseurs.....</i>	<i>25</i>
Exercice 1 : Prédiction de branchement	25
Exercice 2 : Impact de la fenêtre d'instructions RUU	27
Exercice 3 : Mémoires caches - Evaluation des performances de différentes configurations de mémoires caches (instructions et données) pour 4 algorithmes de multiplication de matrices	28
Exercice 4 : Mémoires caches - Evaluation des performances de différentes configurations de mémoires caches (instructions et données).....	31
Questions	34
<i>TD/TP5 : Analyse de performances de configurations de microprocesseurs multicoeurs pour des applications parallèles</i>	<i>39</i>
Description de l'application multiplication de matrice parallèle	40
Questions	42

<i>TD/TP6 : Microprocesseurs et Energie</i>	47
Exercice 1 : Lois Physiques Energie – Puissance	47
Exercice 2 : Mode de gestion de l'énergie des microprocesseurs (ARM) et FSM	48
Exercice 3 : Microélectronique	49
Exercice 4 : Evaluation de performances – McPAT	50
<i>Solution TD/TP1 : Evaluation de performances - programmation assembleur RISC-V</i>	53
Exercice 1 : Evaluation de performances analytique - Loi d'Amdahl.....	53
Exercice 2 : Evaluation de performances analytique - Loi d'Amdahl.....	53
Exercice 3 : Evaluation de performances – simulation.....	55
Exercice 4 : Commentaire d'un programme en assembleur RISC-V.....	57
Exercice 5 : Programmation en assembleur	57
Exercice 6 : Encodage en assembleur	58
Exercice 7 : Commentaire du programme en assembleur	59
Exercice 8 : Programmation en assembleur MIPS (somme d'un tableau)	61
<i>Solution TD/TP2 : Programmation assembleur RISC-V - profiling d'applications et ISS (instruction set simulator)</i>	62
Exercice 1 : Switch statement.....	63
Exercice 2 : Boucles	65
Exercice 3 : Appels de fonctions	66
Exercice 4 : Appels de fonctions	67
Exercice 5 : Profiling d'applications (% d'instructions par catégorie) par simulation sur ISS (instruction set simulator)	68
<i>Solution TD/TP3 : Processeur pipeline – compilateur</i>	71
Exercice 1 : Pipeline.....	71
Exercice 2 : Pipeline - Data Forwarding	73
Exercice 3 : Impact du nombre d'unités fonctionnelles	76
Exercice 4 : Exécution dans l'ordre (in-order) VS exécution dans le désordre (out-of-order).....	79
<i>Solution TD/TP4 : Microprocesseur superscalaire / mémoires caches – Analyse de configurations d'architectures de microprocesseurs</i>	81
Exercice 1 : Prédiction de branchement	81
Exercice 2 : Impact de la fenêtre d'instructions RUU	83
Exercice 3 : Mémoires caches - Evaluation des performances de différentes configurations de mémoires caches (instructions et données) pour 4 algorithmes de multiplication de matrices	85
<i>Solution TD/TP6 : Microprocesseurs et Energie</i>	89
Exercice 1 : Lois Physiques Energie – Puissance	89
Exercice 3 : Microélectronique	94

Exercice 4 : Evaluation de performances – MPACT.....	97
Hors-Programme – Pour aller plus loin.....	100
Compilateur – analyse de dépendances.....	100
Impact du ré-ordonnement de code - Loop unrolling.....	100
Compilation – Optimisation automatique	103
Annexe 1 : Cache	110
Annexe 2 : Applications.....	111
PageRank	111
Betweenness Centrality Score – SSCA #2.....	111
La multiplication de polynômes	112
Secure Hash Algorithm (SHA-1)	112
Bloc cipher BlowFish.....	113
Autres applications 2020	113
• EEMBC : https://www.eembc.org/	113
Annexe 3 : gem5 – généralités	114
1. Résolution des dépendances, récupération des sources et compilation de gem5 + cross-compilateur RISC-V	114
2. (optionnel) Installation de la librairie m5threads.....	115
3. Compilation et exécution d'un programme sur le simulateur gem5	115
4. Utilisation de m5opts.h.....	116
5. Statistiques d'exécution de gem5	117
Annexe 4 : gem5 – mémo TP5-6	118
Log in sur machine salle 2013	118
Compilation de l'application de multiplication de matrice	118
Commande pour lancer une simulation.....	118
Statistiques	119
Annexe 5 : McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures.....	120
Références	121
TP2.....	121
TP3.....	121
TP4.....	121
TP5.....	122
TP6.....	122

Table des Figures

Figure 1 Evolution de la pile lors d'un appel de fonction	17
Figure 2 Processeur RISC pipeline 5 étage.....	20
Figure 3 Data forwarding a) No forwarding b) With forwarding	21
Figure 4 a) Nombre de cycles vs N vs Unités fonctionnelles b) CPI vs N vs Unités fonctionnelles	23
Figure 5 Pipeline d'exécution dans le désordre (out-of-order)	24
Figure 6 Floorplan d'un cœur de processeur Intel	25
Figure 7 Le prédicteur de branchement bimodal	26
Figure 8 Le prédicteur de branchement adaptatif à 2-niveaux	26
Figure 9 Paramètres du prédicteur de branchement	26
Figure 10 Register Update Unit (RUU).....	27
Figure 11 AMD Athlon Processor Block Diagram.....	27
Figure 12 Une configuration typique de big.LITTLE, constituée de 2 clusters, avec 2 CPUs par cluster (Source : ARM).....	31
Figure 13 Pipeline du Cortex A7	32
Figure 14 Pipeline du Cortex A15	32
Figure 15 Flot de modification de la microarchitecture d'un microprocesseur.....	33
Figure 16 a) scalaire b) superscalaire c) SMT Simultaneous Multithreading d) CMP Chip MultiProcessor	39
Figure 17 Multiplication de matrice 4x4 avec 2 threads	40
Figure 18 Architecture multicœurs à base de bus.....	41
Figure 19 Evolution de la pile lors d'un appel de fonction	66
Figure 20 Processeur RISC pipeline 5 étage.....	71
Figure 21 Data forwarding a) No forwarding b) With forwarding	73
Figure 22 Architecture finale du pipeline	75
Figure 23 a) Nombre de cycles vs N vs Unités fonctionnelles b) CPI vs N vs Unités fonctionnelles.....	78
Figure 24 Pipeline d'exécution dans le désordre (out-of-order)	79
Figure 25 Floorplan d'un cœur de processeur Intel	81
Figure 26 Le prédicteur de branchement adaptatif à 2-niveaux	82
Figure 27 Le prédicteur de branchement bimodal	82
Figure 28 Register Update Unit (RUU).....	83
Figure 29 AMD Athlon Processor Block Diagram.....	84
Figure 30 Hiérarchie mémoire vue du processeur	86
Figure 31 Options de compilation pour icc.....	104
Figure 32 Organisation de cache : exemple DECstation 3100 direct-mapped cache (64 KB, 16 Kwords, 1 word/block)	110
Figure 33 Organisation de cache : 4-way set-associative	110
Tableau 1 Evaluation de performance.....	8
Tableau 2 Jeu d'instructions RISC-V RV32I + RV32M.....	11
Tableau 3 Tableau 3 Codage des instructions assembleur du RISC-V	13
Tableau 4 Formats des instructions assembleur du RISC-V.....	14
Tableau 5 File de registres RISC-V	15
Tableau 6 Pourcentage de chaque classe d'instructions	18
Tableau 7 Configurations de caches pour 2 processeurs.....	28
Tableau 8 Paramètres de sim-cache pour chaque configuration	29
Tableau 9 Instruction Cache (il1) Miss Rate	29
Tableau 10 Data Cache (dl1) Miss Rate	29
Tableau 11 Unified Cache (ul2) Miss Rate	30
Tableau 12 Paramètres de configuration des processeurs.....	33
Tableau 13 Pourcentage de chaque classe d'instructions	69

Tableau 14 Paramètres du prédicteur de branchement	82
Tableau 15 Configurations de caches pour 2 processeurs.....	85
Tableau 16 Paramètres de sim-outorder pour chaque configuration	86
Tableau 17 Instruction Cache (il1) Miss Rate.....	87
Tableau 18 Data Cache (dl1) Miss Rate	87
Tableau 19 Unified Cache (ul2) Miss Rate	87
Tableau 20 Résultats du #pragma unroll(n)	106

TD/TP1 : Evaluation de performances - programmation assembleur RISC-V

Exercice 1 : Evaluation de performances analytique - Loi d'Amdahl

Nous souhaitons améliorer les performances d'un microprocesseur en réduisant d'un facteur 5 le temps d'exécution des instructions flottantes. Nous souhaiterions évaluer l'impact d'une telle transformation sachant que le temps d'exécution d'un benchmark avant l'amélioration prend 10 secondes et que 50% du temps est dépensé dans l'exécution de calcul flottant.

Q1 : Quel serait le speed up ?

Exercice 2 : Evaluation de performances analytique - Loi d'Amdahl

Nous souhaiterions améliorer les performances d'un microprocesseur et il existe 2 moyens :

1. Réduire le temps d'exécution des instructions de multiplication d'un facteur 4.
2. Réduire le temps d'exécution des instructions d'accès mémoire d'un facteur 2.

L'exécution du programme benchmark prend 100 secondes et utilise :

- 20 % des instructions pour la multiplication,
- 50% des instructions pour les accès mémoire,
- 30% des instructions pour les autres types d'opérations.

Q1 : Quel est le speedup si l'on améliore uniquement la multiplication ?

Q2 : Quel est le speedup si l'on améliore uniquement les accès mémoires ?

Q3 : Quel est le speedup si l'on améliore les deux ?

Exercice 3 : Evaluation de performances – simulation

Nous souhaitons mesurer les performances d'un programme en utilisant un **simulateur de processeur RISC-V** (gem5 en mode *SE – System Call Emulation*). Les informations concernant l'installation de Gem5 se trouvent en [Annexe 3 : gem5 – généralités](#).

Pour cela, nous allons écrire un programme C compilé en binaire RISC-V puis simuler son exécution.

1. Ecrire un programme en C effectuant la somme de deux vecteurs de taille $N = 50$ que vous initialiserez de manière aléatoire.
2. Compilez ce programme avec

```
riscv64-linux-gnu-gcc -O0 -static -o vadd.riscv vadd.c
```

Lancer le simulateur :

```
cd ~/gem5
build/RISCV/gem5.opt ~/TP/RISCV_se.py -b ~/TP/TP1/vadd.riscv
```

Les résultats se trouvent dans 'm5out/stats.txt' et peuvent être extraits par :

```
grep -E "simInsts|numCycles|ipc|cpi" m5out/stats.txt
```

Complétez Tableau 1 avec les valeurs suivantes :

Q1 : Quel est le nombre total d'instructions exécutées par votre programme sur ce processeur ? **simInsts**

Q2 : Quel est le nombre total de cycles processeur nécessaires pour l'exécution du programme ? **numCycles**

Q3 : Quel est le CPI de votre programme sur ce processeur ? **cpi**

Q4 : Quel est l'IPC de votre programme sur ce processeur ? **ipc**

Tableau 1 Evaluation de performance

Paramètres	Valeur
nombre total d'instructions exécutées	
nombre total de cycles processeur	
CPI	
IPC	

Q5 : Pour un processeur ayant une fréquence d'horloge de 3 GHz, quelle est la durée d'exécution de votre programme ?

Exercice 4 : Commentaire d'un programme en assembleur RISC-V

A l'aide du Tableau 2 du jeu d'instructions assembleur RISC-V, commentez le programme suivant et indiquez ce que fait la fonction $F(X, Y, Z)$.

On suppose que la variable X se trouve dans le registre $s1$, la variable Y dans $s2$ et la variable Z dans $s3$. Le résultat de la fonction $F(X, Y, Z)$ se trouve dans le registre $s7$.

```
add s7, s1, s1
add s7, s7, s7
add s7, s7, s3
add s7, s7, s3
add s7, s7, s2
```

$F(X, Y, Z) =$

Exercice 5 : Programmation en assembleur

A l'aide du Tableau 2 du jeu d'instructions RISC-V, programmez en assembleur RISC-V la fonction $F(X, Y, Z) = X + 2Y - 3Z$.

On suppose que la variable X se trouve dans le registre $s1$, la variable Y dans $s2$, la variable Z dans $s3$ et que le résultat de la fonction $F(X, Y, Z)$ se trouve dans le registre $s7$.

Exercice 6 : Encodage en assembleur

A l'aide du Tableau 3 et du Tableau 4 des codages des instructions RISC-V, codez les instructions suivantes :

```
add s5, s6, s7
and s6, s2, s5 lw
s2, 50(s4)
```

Exercice 7 : Commentaire du programme en assembleur

Commentez le programme suivant et indiquez ce qu'il fait.

Soit A un vecteur de dimension $N = 10$. L'adresse se trouve dans le registre s5.

```
    andi s3, s3, 0
    andi s1, s1, 0
    addi s1, s1, 1
    andi s11, s11, 0
    andi s2, s2, 0
    addi s2, s2, 10
Loop: lw s8, 0(s5)
    add s11, s11, s8
    addi s5, s5, 4
    sub s2, s2, s1
    bne s2, s3, Loop
```

Exercice 8 : Programmation en assembleur (somme d'un tableau)

Ecrivez un programme en assembleur qui calcule la somme des éléments pairs dans un tableau A de dimension $N = 10$. L'adresse de A se trouve dans le registre s5

Tableau 2 Jeu d'instructions RISC-V RV32I + RV32M

Category	Instruction	Example	Meaning	Comment (à corriger)
Arithmetic	add	add s1, s2, s3	$s1 = s2 + s3$	Three operands; no overflow exception
	Subtract	sub s1, s2, s3	$s1 = s2 - s3$	Three operands; no overflow exception
	Add immediate	addi s1, s2, 100	$s1 = s2 + 100$	Add immediate constant; no overflow exception
	set on less than	slt s1, s2, s3	If $(s2 < s3)$ $s1 = 1$; else $s1 = 0$	Signed comparison using two's complement
	set less than immediate	slti s1, s2, 100	If $(s2 < 100)$ $s1 = 1$; else $s1 = 0$	Signed comparison with immediate constant
	set less than unsigned	sltu s1, s2, s3	If $(s2 < s3)$ $s1 = 1$; else $s1 = 0$	Unsigned comparison
	set less than immediate unsigned	sltiu s1, s2, 100	If $(s2 < 100)$ $s1 = 1$; else $s1 = 0$	Unsigned comparison with immediate constant
	Multiply (low)	mul s1, s2, s3	$s1 = s2 \times s3$ [31:0]	Low 32 bits of signed product
	Multiply (high)	mulh s1, s2, s3	$s1 = s2 \times s3$ [63:32]	High 32 bits of signed product
	Divide	div s1, s2, s3	$s1 = s2 / s3$	Signed division; quotient truncated toward zero
	divide unsigned	divu s1, s2, s3	$s1 = s2 / s3$	Unsigned division
	Remainder	rem s1, s2, s3	$s1 = s2 \% s3$	Signed remainder
	Remainder unsiged	remu s1, s2, s3	$s1 = s2 \% s3$	Unsigned remainder
Logical	And	and s1, s2, s3	$s1 = s2 \& s3$	Logical AND
	Or	or s1, s2, s3	$s1 = s2 s3$	Logical OR
	xor	xor s1, s2, s3	$s1 = s2 \wedge s3$	Logical XOR
	and immediate	andi s1, s2, 100	$s1 = s2 \& 100$	Logical AND with constant
	or immediate	ori s1, s2, 100	$s1 = s2 100$	Logical OR with constant
	shift left logical	sll s1, s2, s3	$s1 = s2 \ll (s3[4:0])$	Logical left shift; shift amount taken from lower 5 bits of register (RV32)

	shift right logical	srl	s1, s2, s3	$s1 = s2 \gg (s3[4:0])$	Logical right shift; zero-filled
Data transfer	load word	lw	s1, 100(s2)	$s1 = \text{Memory}[s2+100]$	Word from memory to register
	store word	sw	s1, 100(s2)	$\text{Memory}[s2+100] = s1$	Word from register to memory
	Load byte signed	lb	s1, 100(s2)	$s1 = \text{Memory}[s2+100]$	Signed byte from memory to register
	load byte unsigned	lbu	s1, 100(s2)	$s1 = \text{Memory}[s2+100]$	Unsigned byte from memory to register
	store byte	sb	s1, 100(s2)	$\text{Memory}[s2+100] = s1$	Byte from register to memory
	load upper immediate	lui	s1, 100	$s1 = 100 \ll 12$	Load constant in upper 20 bits
Conditional branch	branch on equal	beq	s1, s2, offset	If $(s1 == s2)$ go to PC+ offset else PC + 4	Equal test; PC-relative branch
	branch on not equal	bne	s1, s2, offset	If $(s1 != s2)$ go to PC+ offset else PC + 4	Not Equal test; PC-relative
	branch on lower than	blt	s1, s2, offset	If $(s1 < s2)$ go to PC+ offset else PC + 4	Signed comparison
	branch on greater or equal than	bge	s1, s2, offset	If $(s1 \geq s2)$ go to PC+ offset else PC + 4	Signed comparison
	branch on unsigned lower than	bltu	s1, s2, offset	If $(s1 < s2)$ go to PC+ offset else PC + 4	Unsigned comparison
	branch on unsigned greater or equal than	bgeu	s1, s2, offset	If $(s1 \geq s2)$ go to PC+ offset else PC + 4	Unsigned comparison
Unconditional jump	jump and link (procedure call)	jal	ra, offset	$ra = PC + 4$; $PC = PC + \text{offset}$	PC-relative jump; stores return address (PC+4) in destination register
	jump to register + offset	jalr	ra, 100(s1)	$ra = PC + 4$; $PC = (s1 + 100) \& \sim 1$	Indirect jump via register + offset; return address saved

Voir documentation complète : <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.pdf>

Tableau 3 Tableau 3 Codage des instructions assembleur du RISC-V

Name	Type	Example						comments
		7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
add	R	0000000	rs2	rs1	000	rd	0110011	add rd, rs1, rs2
sub	R	0100000	rs2	rs1	000	rd	0110011	sub rd, rs1, rs2
addi	I	imm[11:0]		rs1	000	rd	0010011	addi rd, rs1, imm
slt	R	0000000	rs2	rs1	010	rd	0110011	slt rd, rs1, rs2
slti	I	imm[11:0]		rs1	010	rd	0010011	slti rd, rs1, imm
sltu	R	0000000	rs2	rs1	011	rd	0110011	sltu rd, rs1, rs2
sltiu	I	imm[11:0]		rs1	011	rd	0010011	sltiu rd, rs1, imm
mul	R	0000001	rs2	rs1	000	rd	0110011	mul rd, rs1, rs2
mulh	R	0000001	rs2	rs1	001	rd	0110011	mulh rd, rs1, rs2
div	R	0000001	rs2	rs1	100	rd	0110011	div rd, rs1, rs2
divu	R	0000001	rs2	rs1	101	rd	0110011	divu rd, rs1, rs2
rem	R	0000001	rs2	rs1	110	rd	0110011	rem rd, rs1, rs2
remu	R	0000001	rs2	rs1	111	rd	0110011	remu rd, rs1, rs2
and	R	0000000	rs2	rs1	111	rd	0110011	and rd, rs1, rs2
or	R	0000000	rs2	rs1	110	rd	0110011	or rd, rs1, rs2
xor	R	0000000	rs2	rs1	100	rd	0110011	xor rd, rs1, rs2
andi	I	imm[11:0]		rs1	111	rd	0010011	andi rd, rs1, imm
ori	I	imm[11:0]		rs1	110	rd	0010011	ori rd, rs1, imm
sll	R	0000000	rs2	rs1	001	rd	0110011	sll rd, rs1, rs2
srl	R	0000000	rs2	rs1	101	rd	0110011	srl rd, rs1, rs2
lw	I	imm[11:0]		rs1	010	rd	0000011	lw rd, 100(rs1)
sw	S	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw rd, 100(rs1)
lb	I	imm[11:0]		rs1	000	rd	0000011	lb rd, 100(rs1)
lbu	I	imm[11:0]		rs1	100	rd	0000011	lbu rd, 100(rs1)

sb	S	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	sb	rs2, imm(rs1)
lui	U	imm[31:12]				rd	0110111	lui	rd, imm
beq	SB	imm[12 10:0]	rs2	rs1	000	imm[4:1 11]	1100011	beq	rs1, rs2, label
bne	SB	imm[12 10:0]	rs2	rs1	001	imm[4:1 11]	1100011	bne	rs1, rs2, label
blt	SB	imm[12 10:0]	rs2	rs1	100	imm[4:1 11]	1100011	blt	rs1, rs2, label
bge	SB	imm[12 10:0]	rs2	rs1	101	imm[4:1 11]	1100011	bge	rs1, rs2, label
bltu	SB	imm[12 10:0]	rs2	rs1	110	imm[4:1 11]	1100011	bltu	rs1, rs2, label
bgeu	SB	imm[12 10:0]	rs2	rs1	111	imm[4:1 11]	1100011	bgeu	rs1, rs2, label
jal	UJ	imm[20 10:1 11 19:12]				rd	1101111	jal	rd, label
jalr	I	imm[11:0]		rs1	000	rd	1100111	jalr	rd, imm(rs1)

Tableau 4 Formats des instructions assembleur du RISC-V

name	Fields						Comments
Field size	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	All RISC-V instructions 32 bits
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic & logic instruction format
I-type	imm[11:0]		rs1	funct3	rd	opcode	Immediate, memory load & jalr format
S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	Memory store format
SB-type	imm[12 10:5]	rs2	rs1	funct3	imm[4:1 11]	opcode	Conditional branch format
U-type	imm[31:12]				rd	opcode	High immediate format
UJ-type	imm[20 10:1 11 19:12]				rd	opcode	Unconditional jump format

Rappel sur les conventions de l'assembleur RISC-V pour les registres :

Tableau 5 File de registres RISC-V

Name	Register	Usage	Saver
zero	x0	Valeur constante 0	--
ra	x1	Adresse de retour	Caller*
sp	x2	Pointeur de pile (stack pointer)	Callee**
gp	x3	Pointeur global (global pointer)	--
tp	x4	Pointeur de thread (thread pointer)	--
t0-2	x5-7	Variables temporaires	Caller
s0/fp	x8	Registre sauvegardé / Pointeur de frame (frame pointer)	Callee
s1	x9	Registre sauvegardé	Callee
a0-1	x10-11	Argument de fonction / valeur retour	Caller
a2-7	x12-17	Argument de fonction	Caller
s2-11	x18-27	Registre sauvegardé	Callee
t3-6	x28-31	Registres temporaires	Caller
ft0-7	f0-7	FP temporaires	Caller
fs0-1	f8-9	FP registres sauvegardés	Callee
fa0-1	f10-11	FP arguments / valeur retour	Caller
fa2-7	f12-17	FP arguments	Caller
fs2-11	f18-27	FP registres sauvegardés	Callee
ft8-11	f28-31	FP temporaires	Caller

Caller* : La fonction appelante doit sauvegarder le registre si elle veut conserver sa valeur après l'appel.

Callee** : La fonction appelée doit sauvegarder le registre si elle le modifie, et le restaurer avant de retourner.

TD/TP2 : Programmation assembleur RISC-V - profiling d'applications et ISS (instruction set simulator)

Exercice 1 : Switch statement

Veillez écrire en assembleur l'équivalent du code C suivant :

```
switch(k) {
    case 0: f = i + j ; break ; /* k = 0 */
    case 1: f = g + h ; break ; /* k = 1 */
    case 2: f = g - h ; break ; /* k = 2 */
    case 3: f = i - j ; break ; /* k = 3 */
}
```

Q1 : Proposez 2 solutions dont une version où le temps d'exécution est Indépendant du nombre de cas. On supposera que les variables f, g, h, i, j et k se trouvent dans les registres s0 -s5 respectivement.

Exercice 2 : Boucles

Veillez écrire en assembleur l'équivalent du code C suivant :

```
sum = 0 ;
for(i= 0 ; i < 1000 ; i++) {
    for(j= 500 ; j > 0 ; j--) {
        for(k = 0 ; k < 300; k = k + 10) {
            sum = sum + (i + j + k);
        }
    }
}
```

On supposera que les variables sum, i, j et k se trouvent dans les registres s1-s4 respectivement.

Exercice 3 : Appels de fonctions

Veillez écrire en assembleur l'équivalent du code C suivant :

```
int leaf(int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

On supposera que les variables g, h, i et j sont transmises par les registres a1, a2, a3, a4 lors de l'appel de fonction, et que f correspond à a0. Lors de l'exécution d'un appel de fonction, il est

nécessaire de sauvegarder dans une pile en mémoire le contenu des registres utilisés. Pour rappel, le registre sp donne l'adresse de la pile.

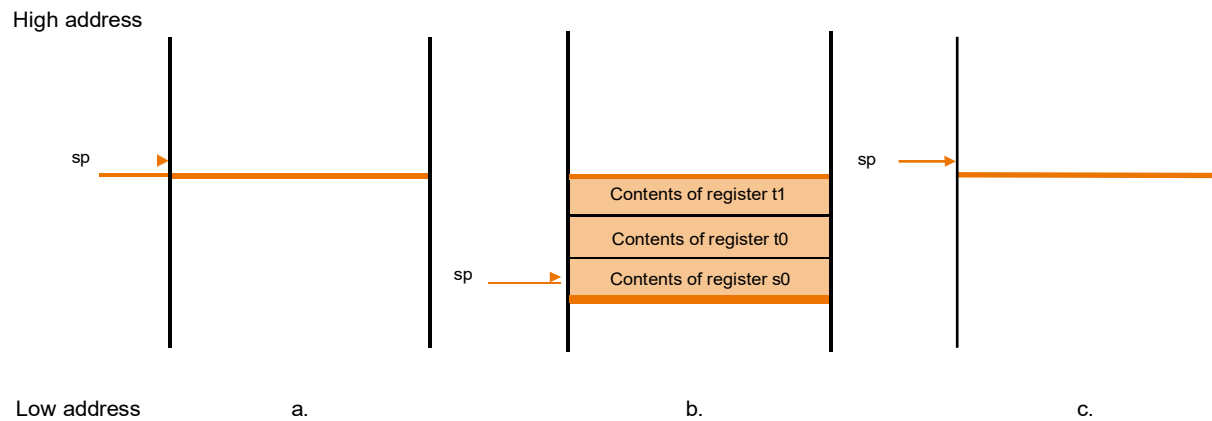


Figure 1 Evolution de la pile lors d'un appel de fonction

Q1 : Pourquoi les appels de fonctions affectent les performances des programmes ?

Q2 : Pourquoi la récursivité en programmation affecte les performances ?

Exercice 4 : Appels de fonctions

Veillez écrire en assembleur l'équivalent du code C suivant :

```
void swap(int v[], int k)
{
    int tmp;
    tmp = v[k];
    v[k] = v[k+1];
    v[k+1] = tmp;
}
```

Exercice 5 : Profiling d'applications (% d'instructions par catégorie) par simulation sur Gem5

Nous allons maintenant nous intéresser au profiling d'une application tirée de l'inférence de réseaux de neurones convolutif (**CNN**). Ce type d'application repose sur des opérations de **convolution**, utilisées pour appliquer des filtres sur des images afin d'en extraire des caractéristiques pertinentes. Trois variantes de l'opération de convolution sont proposées :

1. La **convolution d'une matrice d'entiers 320x320** représentant une image brute avec un filtre 4x4.
2. La **convolution d'une matrice de flottants 320x320** représentant une image normalisée avec un filtre 4x4.
3. La **convolution d'une matrice d'entiers 320x320** représentant une image avec un filtre 4x4 avec un déroulage de boucle permettant une première optimisation des instructions.

Q1 : Générez le pourcentage de chaque classe d'instructions utilisées par ces programmes, puis remplissez le tableau suivant

Tableau 6 Pourcentage de chaque classe d'instructions

Classe d'instructions	P1 (%)	P2 (%)	P3 (%)
Instructions de lecture (Load)			
Instructions d'écriture (Store)			
Instructions de branchement inconditionnel (Uncond branch)			
Instructions de branchement conditionnel (Cond branch)			
Instructions de calcul entier (Int computation)			
Instructions de calcul flottant (FP computation)			
Nombre total d'instructions exécutées			

Compilation (RISC-V statique) :

```
riscv64-linux-gnu-gcc -O0 -static -o conv_int.riscv conv_int.c
riscv64-linux-gnu-gcc -O0 -static -o conv_float.riscv conv_float.c
riscv64-linux-gnu-gcc -O0 -static -o conv_unrolled.riscv conv_unrolled.c
```

Attention, pour éviter que le compilateur ne suroptimise (car le résultat n'est pas utilisé) et supprime la majorité des instructions, il faut passer en O0. Cependant, en O0 on garde encore une majorité d'instructions « inutiles » ou pas optimales. Pour cela, on utilisera une initialisation des matrices de l'image et du kernel pour imposer le calcul au compilateur. On utilisera les programmes *_m5.c qui comportent des balises de ROI pour les mesures des statistiques issue de <gem5/m5ops.h>.

Ils seront compilés comme ceci :

```
riscv64-linux-gnu-gcc -O2 -static -fno-lto -I/path_to_gem5/include
conv_m5.c /path_to_gem5/util/m5/build/riscv/out/libm5.a -o
conv_m5.riscv
```

Lancez avec :

```
cd ~/gem5/
```

```
build/RISCV/gem5.opt ~/TP/RISCV_se.py -b ~/TP/TP2/conv_int.riscv
```

Vous obtiendrez les informations par la commande suivante¹ :

```
grep -E "simInsts|numLoadInsts|numStoreInsts|numBranches|numIntInsts|numFpInsts"
m5out/stats.txt
```

Pour ces instructions, il faudra chercher les lignes contenant : **commitStats0**. Le détails des calculs entiers et flottants se trouvent juste en dessous dans le fichier **stats.txt** (ex : IntAlu). Les branches sont comptabilisées dans les calculs ALU.

Q2: Quelle catégorie d'instructions nécessiterait une amélioration de performances ?

¹ Les pourcentages sont à calculés à partir du nombre total d'instructions

TD/TP3 : Processeur pipeline – superscalaire

Exercice 1 : Pipeline

Identifier toutes les dépendances de données dans le code suivant. Quelles dépendances sont des aléas de données qui seront résolus par *forwarding* ?

add \$2,\$5,\$4

add \$4,\$2,\$5

sw \$5, 100(\$2)

add \$3,\$2,\$4

Vous devez vous appuyer dans votre analyse sur la Figure 2 décrivant le pipeline du processeur.

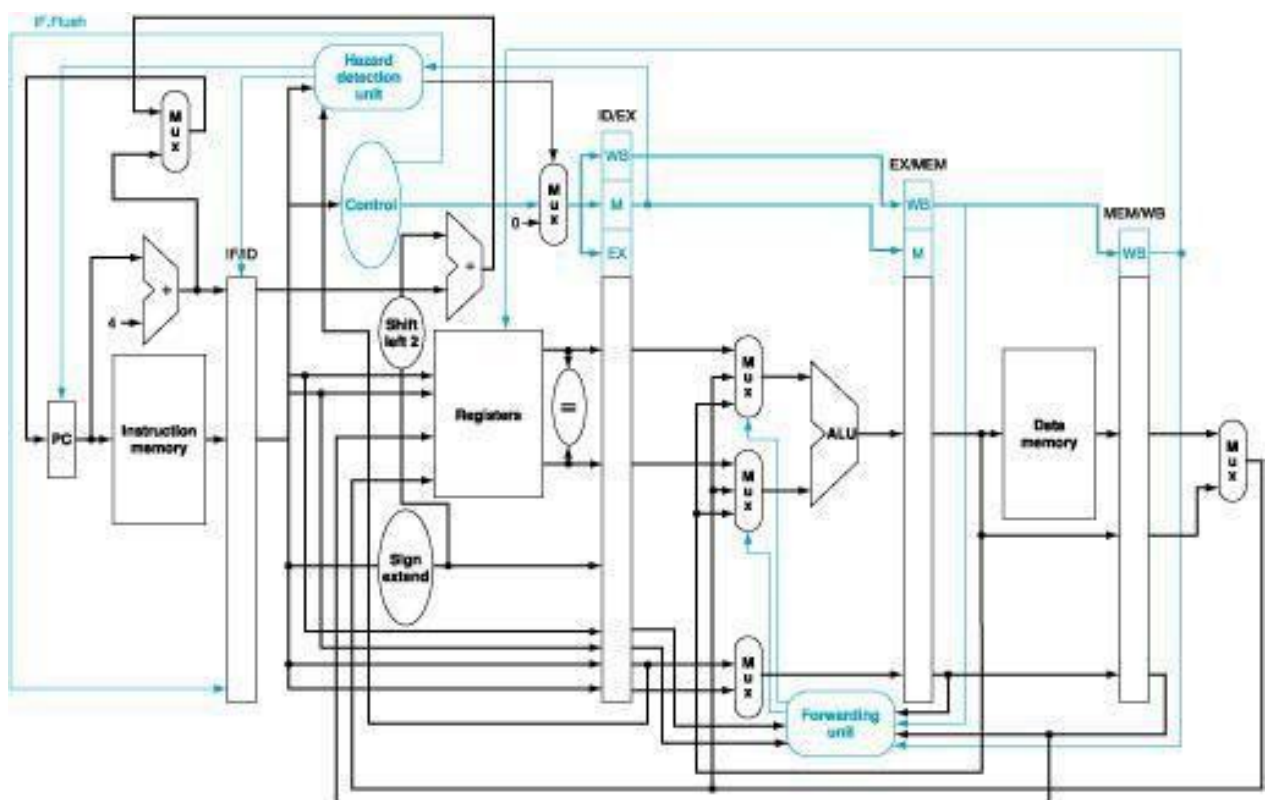


Figure 2 Processeur RISC pipeline 5 étage

Exercice 2 : Pipeline - Data Forwarding

Nous souhaitons modifier le pipeline du processeur pour lui ajouter 2 nouvelles unités fonctionnelles :

1. Une unité de multiplication
2. Une unité racine carrée

Modifiez le chemin de données du processeur et l'unité de data forwarding pour prendre en compte ces deux nouvelles unités fonctionnelles.

Vous pouvez vous appuyer dans votre analyse sur la Figure 3 décrivant le data forwarding au sein du pipeline du processeur.

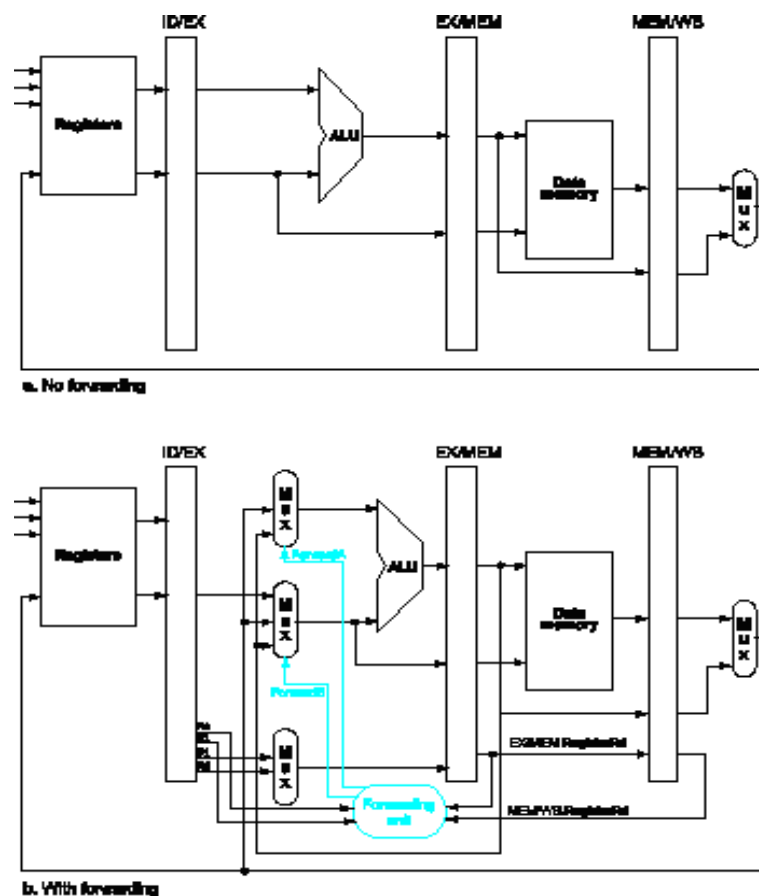


Figure 3 Data forwarding a) No forwarding b) With forwarding

Exercice 3 : Impact du nombre d'unités fonctionnelles

Dans gem5, le processeur DerivO3CPU permet de configurer finement le nombre et le type d'unités fonctionnelles à l'aide d'un pool d'unités fonctionnelles (FU pool).

--ialu	M	nombre total d'ALUs (unité arithmétique et logique) entière
--imult	M	nombre total de multiplieurs/diviseurs entiers
--fpalu	M	nombre total d'ALUs flottantes (données de type float)
--fpmult	M	nombre total de multiplieurs/diviseurs flottants (données de type float)
--mempport	M	nombre de ports mémoire, c'est-à-dire le nombre maximal d'accès mémoire (load/store) pouvant être émis par cycle

Ce nombre total M peut varier de 0 à M où M dans les processeurs actuels ne dépasse pas 8, de manière générale.

Exemple de commande correspondant à une configuration avec 4 ALUs, 1 multiplieur, 4 ALUs flottantes, 1 multiplieur flottant :

```
$GEM5 -d "out_dir" "se.py" --cmd="program.riscv" --cpu-type=O3 --caches \
--ialu=4 --imult=1 --fpalu=4 --fpmult=1 --mempport=2
```

Exemple de commande correspondant à une configuration avec 8 ALUs, 4 multiplieurs, 1 ALU flottante, 1 multiplieur flottant :

```
$GEM5 -d "out_dir" "se.py" --cmd="program.riscv" --cpu-type=O3 --caches \
--ialu=8 --imult=4 --fpalu=1 --fpmult=1 --mempport=2
```

Du point de vue applicatif, l'étude sera menée sur l'algorithme PageRank, développé par Google comme métrique mesurant l'importance d'une page web [1]. Il s'avère que cette métrique peut également être utilisée dans d'autres types de réseaux (e.g. sociaux) afin de caractériser l'importance relative d'un sommet (se reporter à l'Annexe 2 : Applications).

A l'étape initiale, chaque sommet se voit attribuer un rang égal. Chaque sommet recalcule son rang en fonction du rang de ses voisins et de ses degrés entrants/sortants. On considère que l'algorithme à convergé lorsque la variation de rang, d'une itération à l'autre, est inférieur à un seuil (voir [1] pour le détail).

Q1 : Analysez le code de l'algorithme PageRank en faisant varier la taille du graphe d'entrée (min, med, max) et en faisant varier le nombre d'unités fonctionnelles M de 1 à 8 (1, 2, 4, 8). (Voir annexe « Applications »)

Q2 : Tracez 2 figures indiquant le nombre de cycles et le CPI associés en fonction de la taille du problème traité ($N=\{\text{min}, \text{med}, \text{max}\}$) et des unités fonctionnelles. Quelle est votre analyse ?

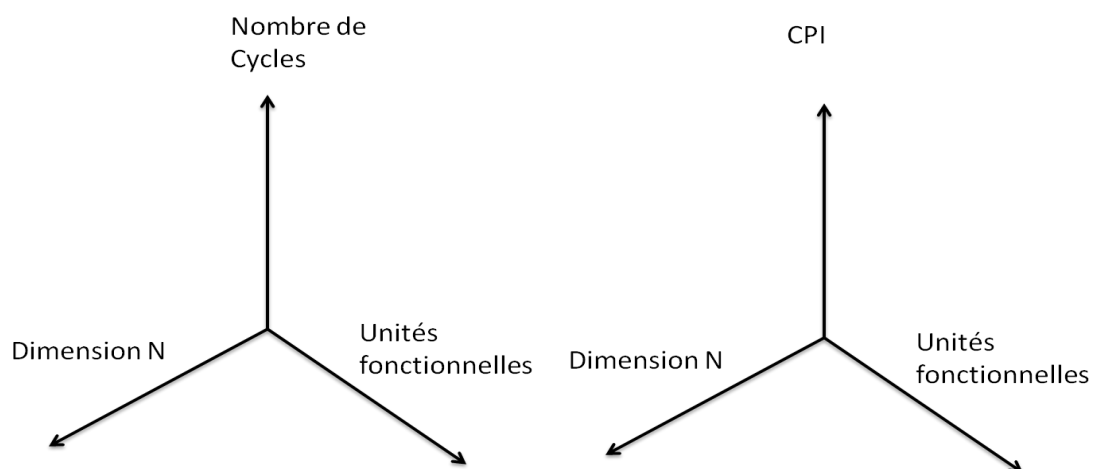


Figure 4 a) Nombre de cycles vs N vs Unités fonctionnelles b) CPI vs N vs Unités fonctionnelles

Exercice 4 : Exécution dans l'ordre (in-order) VS exécution dans le désordre (out-of-order)

Dans un processeur superscalaire à ré-ordonnancement dynamique des instructions, l'exécution des instructions peut s'effectuer dans le désordre dans l'objectif d'éviter les contraintes artificielles de la séquentialité.

L'exécution d'un programme tel que généré sans optimisations par un compilateur représente l'ordre sémantique normal du programme. Pourtant l'ordre d'exécution des instructions peut être modifié tout en préservant la sémantique. Cette exécution est appelée **out-of-order exécution**.

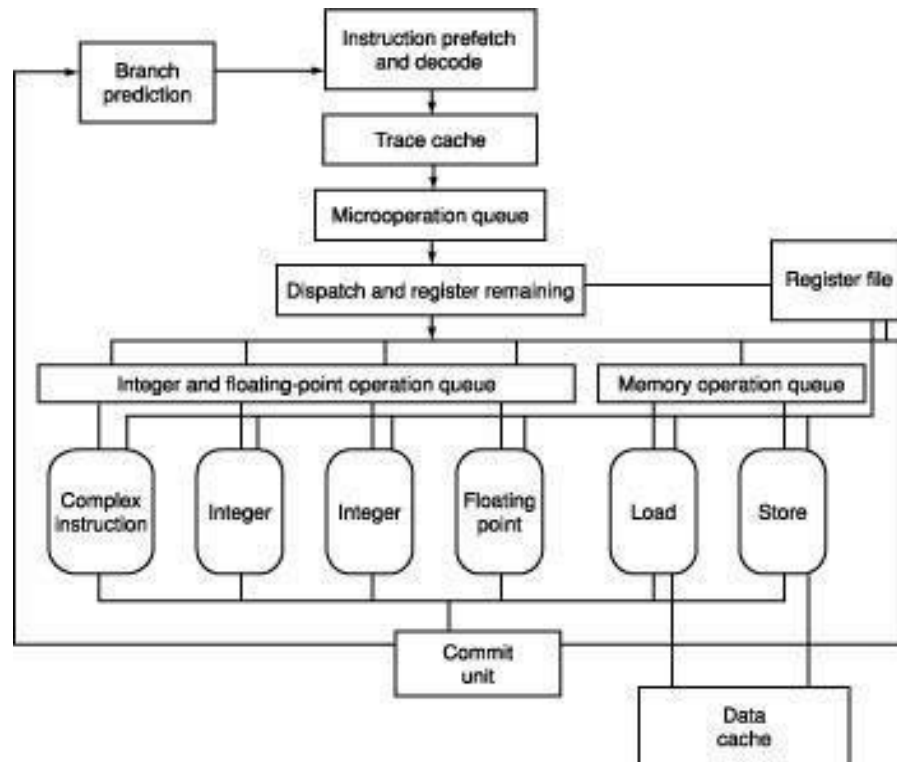


Figure 5 Pipeline d'exécution dans le désordre (out-of-order)

Le simulateur **sim-outorder** permet d'exécuter les instructions dans l'ordre ou le désordre (*out-of-order execution*). Les deux commandes suivantes permettent les 2 possibilités :

- Exécution dans l'ordre : `$GEM5 --cpu-type=MinorCPU`
- Exécution dans le désordre : `$GEM5 --cpu-type=DerivO3CPU`

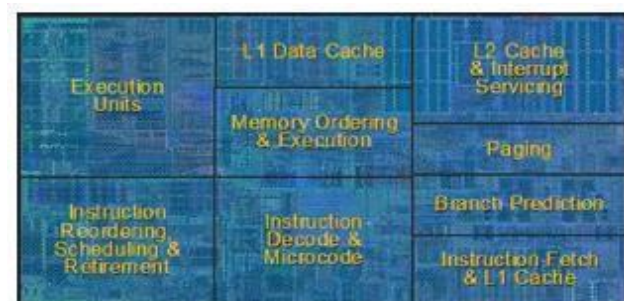
Par défaut le simulateur, ayant un modèle de processeur superscalaire avec ré-ordonnancement dynamique, exécute suivant un mode *out-of-order execution*.

Q1 : *Évaluez l'impact de l'exécution dans l'ordre par rapport à l'exécution dans le désordre pour l'algorithme PageRank (CPI, nombre de cycles).*

TD/TP4 : Microprocesseur superscalaire/mémoires caches - Analyse de configurations d'architectures de microprocesseurs

Exercice 1 : Prédiction de branchement

La prédiction de branchement permet aux processeurs d'exécuter un flot d'instructions sans pénalité de blocage dû à l'attente de l'évaluation de la condition de branchement. Tous les microprocesseurs haute-performance incluent une unité de prédiction de branchement. La Figure 6 montre la surface prise par l'unité de prédiction de branchement dans un processeur multi-cœur Intel.



Core floorplan with major units highlighted.

Figure 6 Floorplan d'un cœur de processeur Intel

Nous souhaitons dans cet exercice évaluer l'impact de différents mécanismes de prédiction de branchement sur le CPI et le nombre de cycles pour 2 applications de votre choix.

Les mécanismes de prédiction que nous souhaitons comparer sont les suivants :

- **nottaken** : ce mécanisme de prédiction de branchement statique considère toujours que le branchement n'est pas pris
- **taken** : ce mécanisme de prédiction de branchement statique considère toujours que le branchement est pris
- **perfect** : ce prédicteur « virtuel » représente la prédiction parfaite
- **bimod** : le prédicteur de branchement bimodal utilise un BTB (Branch Target Buffer) avec compteurs à 2 bits
- **2lev** : prédicteur de branchement adaptatif à 2-niveaux

Vous pouvez spécifier en utilisant le simulateur Gem5 le mécanisme de prédiction de branchement de votre choix.

La documentation de Gem5 précise les détails pour les différents mécanismes de prédiction de branchement. La Figure 7 et la Figure 8 décrivent 2 types de prédicteurs de branchement, et la Figure 9 décrit les différents paramètres en entrée pour le prédicteur de branchement.

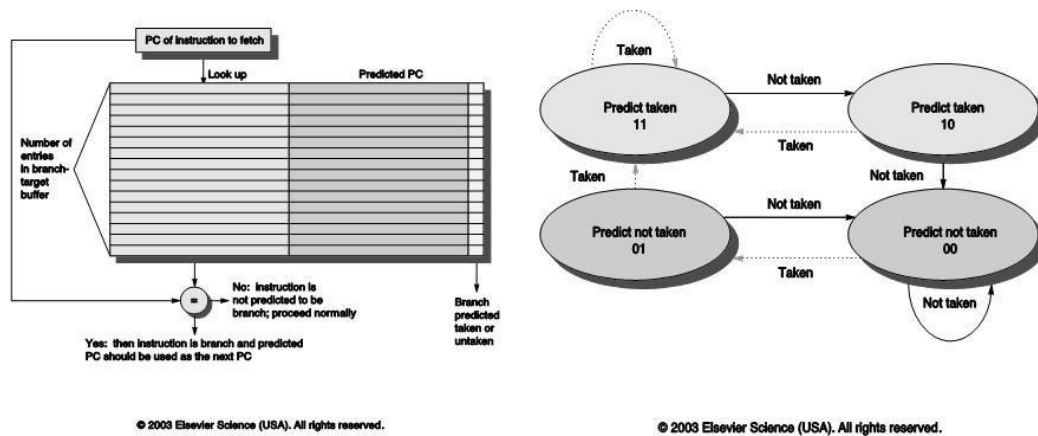


Figure 7 Le prédicteur de branchement bimodal

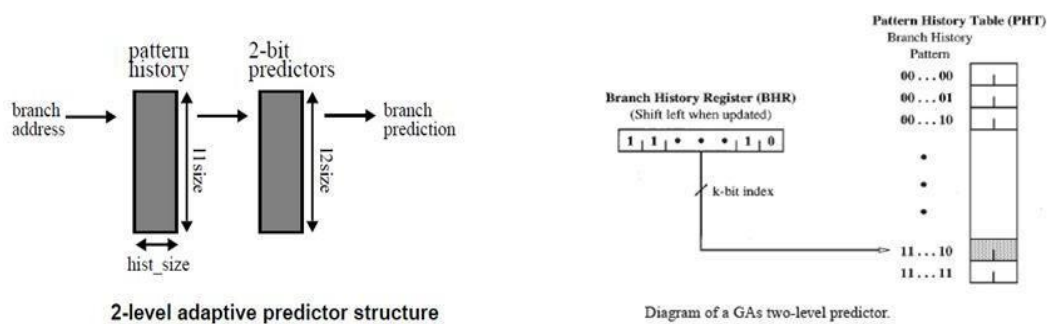


Figure 8 Le prédicteur de branchement adaptatif à 2-niveaux

predictor	l1_size	hist_size	l2_size	xor
GAg	1	W	2^W	0
GAp	1	W	$>2^W$	0
PAg	N	W	2^W	0
PAp	N	W	2^{N+W}	0
gshare	1	W	2^W	1

Branch predictor parameters

Figure 9 Paramètres du prédicteur de branchement

Q1: Etudier l'impact des différents prédicteurs de branchement sur le CPI et le nombre de Cycles pour le programme de votre choix

Exercice 2 : Impact de la fenêtre d'instructions RUU

Le modèle d'exécution superscalaire utilise un mécanisme de ré-ordonnement basé sur une partie centrale : la RUU (Register Update Unit).

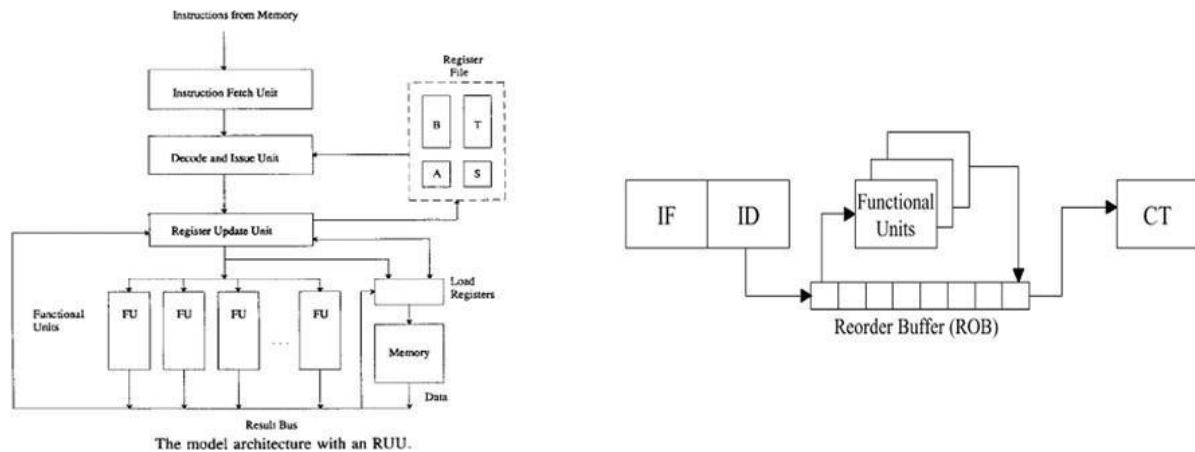


Figure 10 Register Update Unit (RUU)

L'augmentation de la taille de la RUU permet à priori de considérer davantage d'instructions potentielles pour l'exécution. Nous souhaitons analyser l'impact de cette taille sur le CPI.

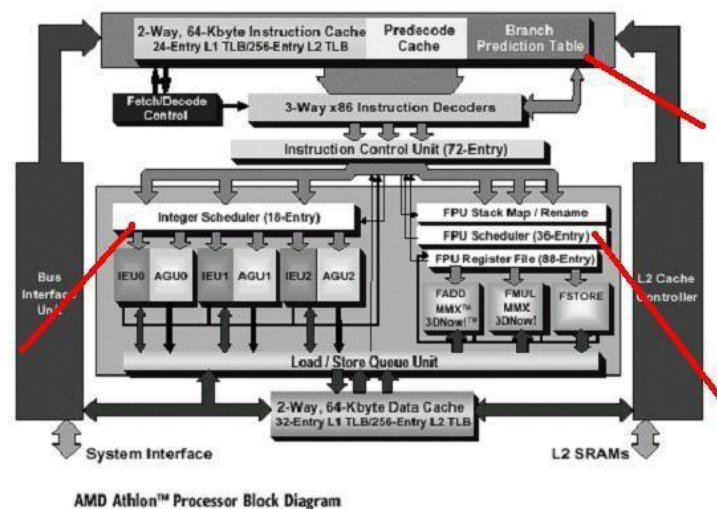


Figure 11 AMD Athlon Processor Block Diagram

Q1: Faites varier la taille de la RUU de 16 à 128 et évaluez l'impact sur le CPI et le nombre de cycles sur une application de votre choix

Exercice 3 : Mémoires caches - Evaluation des performances de différentes configurations de mémoires caches (instructions et données) pour 4 algorithmes de multiplication de matrices.

De nombreuses applications informatiques dans différents domaines font appel à des opérations sur des matrices, et en particulier la multiplication de matrices. Cette opération classique peut être répétée de nombreuses fois au cours de l'exécution de ces applications et il est donc primordial que ses performances soient optimisées. La multiplication de matrices sur des matrices de dimensions importantes est une opération dont la performance en temps d'exécution dépend beaucoup des performances de la hiérarchie mémoire, et notamment l'organisation des caches du microprocesseur sur lequel elle s'exécute. Le taux de défauts de caches (**miss rate**), qui est le rapport entre le nombre d'accès à un cache qui se traduisent par un défaut de cache (**cache miss**) sur le nombre total d'accès au cache, est le paramètre principal pour évaluer cette performance. Nous considérons ici des microprocesseurs ayant des caches d'instructions et de données séparés. De ce fait, le miss rate du cache d'instructions et du cache de données méritent une analyse.

Un simulateur de cache, permet d'effectuer une évaluation du miss rate pour une configuration de caches particulière et pour un programme en entrée. Lorsque l'on souhaite évaluer plusieurs configurations de caches il est alors nécessaire d'effectuer autant de simulations. Ce processus se répète pour le nombre de programmes que l'on considère en entrée.

Travail demandé

Nous considérons les 2 organisations de caches décrites dans le Tableau 7. On supposera que l'algorithme de remplacement est LRU pour toutes les configurations.

Tableau 7 Configurations de caches pour 2 processeurs

Configuration	Instruction cache	Data cache	L2 cache	Block size (bytes)
C1	4KB direct-mapped	4KB direct-mapped	32KB direct-mapped	32
C2	4KB direct-mapped	4KB 2-way set-asso	32KB 4-way set-asso	32

Q1 : Pour chacune des 2 configurations de mémoires cache, complétez le Tableau 8. Pour cela, vous devez déterminer les paramètres d'entrée du simulateur de cache de gem5

Tableau 8 Paramètres de sim-cache pour chaque configuration

Configuration	IL1	DL1	UL2
C1			
C2			

Q2 : Complétez les tableaux 9, 10 et 11. Pour cela vous devez simuler à l'aide de sim-cache les différentes configurations et collecter les informations suivantes :

- Le taux de défauts dans le cache d'instructions il1
- Le taux de défauts dans le cache de données dl1
- Le taux de défauts dans le cache unifié (L2) ul2

Tableau 9 Instruction Cache (il1) Miss Rate

Programmes	<i>Configurations de caches</i>	
	C1	C2
P1 (normale)		
P2 (pointeur)		
P3 (tempo)		
P4 (unrol)		

Tableau 10 Data Cache (dl1) Miss Rate

Programmes	<i>Configurations de caches</i>	
	C1	C2
P1 (normale)		
P2 (pointeur)		
P3 (tempo)		

P4 (unrol)		
------------	--	--

Tableau 11 Unified Cache (ul2) Miss Rate

Programmes	<i>Configurations de caches</i>	
	C1	C2
P1 (normale)		
P2 (pointeur)		
P3 (tempo)		
P4 (unrol)		

Q3: Les 4 algorithmes de multiplication de matrices présentent-ils une bonne localité de références pour le code ? Pourquoi ?

Exercice 4 : Mémoires caches - Evaluation des performances de différentes configurations de mémoires caches (instructions et données)

Description du problème

Le marché des processeurs généralistes pour serveurs et desktops est dominé par Intel [2] et AMD [3], tandis que le marché des processeurs embarqués est dominé par ARM [4] et MIPS [5]. Au sein des processeurs embarqués, on trouve généralement une classification supplémentaire selon le domaine applicatif (automobile, image/vidéo, militaire/espace, télécommunications, etc.).

Cependant, chez les fabricants de ces deux grandes familles de processeurs (généralistes et embarqués), la tendance commune est à l'élargissement de la gamme de processeurs disponibles : processeurs très performants mais énergivores d'un côté, processeurs légèrement moins performants mais plus efficaces de l'autre. Cela rend les frontières entre ces deux mondes de plus en plus floues, mais cela dégage un objectif commun à tous les fabricants : la recherche de l'efficacité énergétique (ratio performances pures à consommation d'énergie donnée), en témoigne la nouvelle architecture **big.LITTLE** de ARM [6].

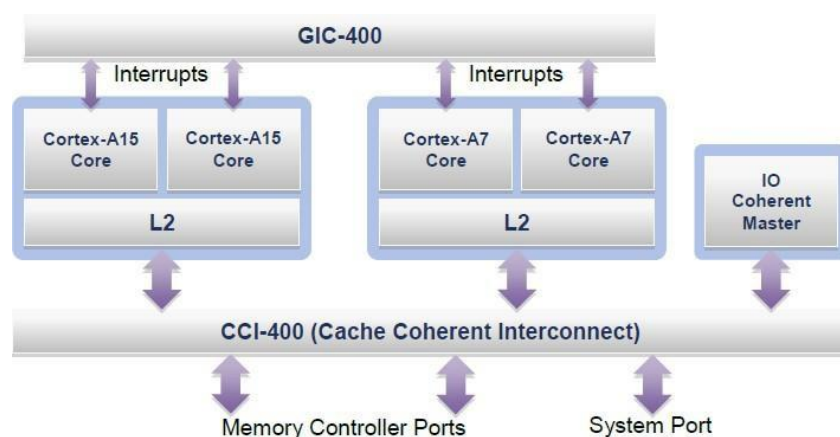


Figure 12 Une configuration typique de big.LITTLE, constituée de 2 clusters, avec 2 CPUs par cluster (Source : ARM)

Cette architecture intègre au sein du même processeur deux cœurs ARM embarqués : un **Cortex A15** (hautes performances, [4]) et un **Cortex A7** (haute efficacité énergétique, [4]). Ces deux cœurs étant entièrement compatibles au niveau du jeu d'instructions, l'architecture peut décider de reloger l'exécution sur l'un ou l'autre des cœurs en fonction des besoins en calcul de l'application. Ces deux cœurs fonctionnent donc de manière alternative (un cœur ON - un cœur OFF), afin de maximiser l'efficacité de l'architecture globale.

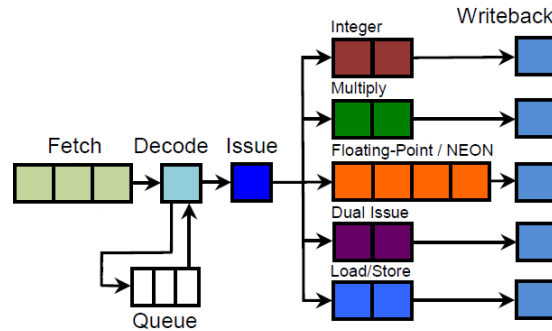


Figure 13 Pipeline du Cortex A7

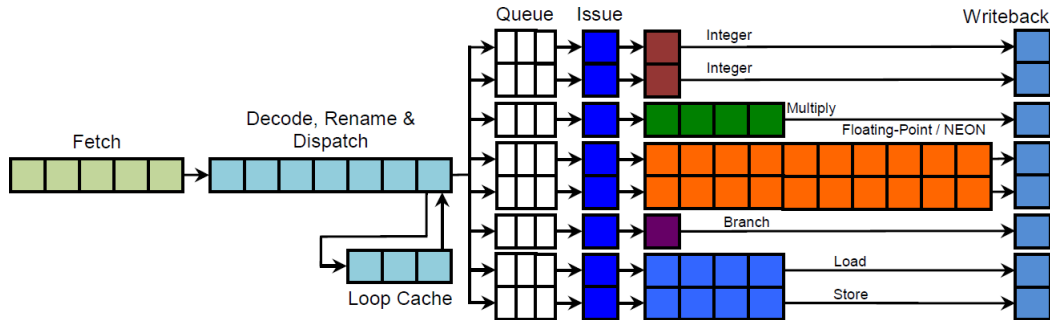


Figure 14 Pipeline du Cortex A15

Comme on a pu le voir, les critères exigés par chaque domaine applicatif (performances, énergie, surface) influencent les choix architecturaux. La microarchitecture d'un processeur spécifie : (1) le mode de traitement des instructions (pipeline, parallélisme d'instructions), (2) l'exécution dans l'ordre ou dans le désordre et enfin (3) le nombre et le type d'unités fonctionnelles (additionneurs, multiplieurs, diviseurs, unités spécialisées, etc.). Il est possible d'identifier pour un programme donné la classe (le type) des instructions sollicitées et le pourcentage d'instructions exécutées issues de cette classe. Cette phase d'identification s'appelle le **profiling**, et a déjà été abordée dans les TD/TP précédents. Il est clair que si le profiling d'une application montre qu'il existe un pourcentage élevé d'utilisation d'une classe particulière, il est alors souhaitable, pour accroître les performances, d'augmenter le nombre d'unités fonctionnelles correspondant à ce type. A contrario, les classes d'instructions ayant un faible taux d'utilisation pourraient voir leur nombre d'unités associées réduites, voire éliminées.

Ce processus consiste à faire une **exploration manuelle/automatique** de la microarchitecture en modifiant les paramètres de configuration de l'architecture et à évaluer l'impact sur les performances et la surface. Néanmoins, ce processus s'applique plutôt à des processeurs en cours de conception. Dans le cas d'une offre existante, il s'agit plutôt d'estimer les performances potentielles en modélisant des processeurs existants.

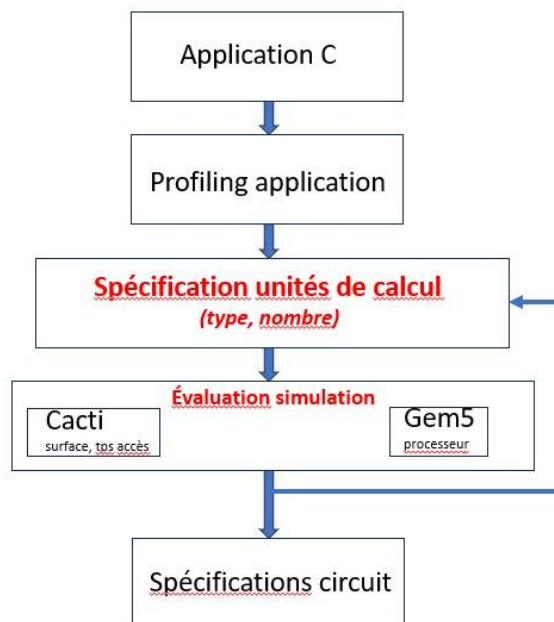


Figure 15 Flot de modification de la microarchitecture d'un microprocesseur

Le but de ce TD/TP est d'analyser les performances des 2 cœurs ARM décrits en introduction : **Cortex A7** et **Cortex A15**, sur des applications type traitement de graphes. Les configurations de ces cœurs ainsi qu'une configuration classique de leurs architectures de caches sont décrites dans le tableau suivant

Tableau 12 Paramètres de configuration des processeurs

Cœur	I-L1\$ (cache/ bloc/ assoc.)	D-L1\$ (cache/ bloc/ assoc.)	L2\$ (cache/ bloc/ assoc.)	Prédicteur de branchement		Fetch queue	Decode/ Issue/ Commit	RUU/ LSQ	# ALU entiers/ flottants	# multip. entiers/ flottants
				Type	Latence mis-préd. (cycles)					
Cortex A15	32KB/ 64/ 2	32KB/ 64/ 2	512KB/ 64/ 16	2 level BTB=256	15	8	4/8/4	16/16	5/1	1/1
Cortex A7	32KB/ 32/ 2	32KB/ 32/ 2	512KB/ 32/ 8	bimodal BTB=256	8	4	2/4/2	2/8	1/1	1/1

Pour les applications, la suite de benchmarks pour l'embarqué **MiBench** sera utilisée et notamment l'applcatif Dijkstra qui recherche les plus court chemins d'un graphe. Cette suite est téléchargeable sur <https://github.com/embecosm/mibench> Une description du benchmark est disponible ici : <https://vhosts.eecs.umich.edu/mibench/Publications/MiBench.pdf>

- Compilez l'application **Dijkstra** et **BlowFish** avec **sslittle-na-sstrix-gcc** en modifiant au besoin les *Makefile* associés.

Questions

1. Profiling de l'application :

Effectuez un profiling des applications sélectionnées en utilisant le simulateur **gem5**

Q1 : Générez le pourcentage de chaque classe d'instructions de ces applications et remplissez les valeurs dans un tableau.

Q2 : Quelle catégorie d'instructions nécessiterait une amélioration de performances ? Expliquez en quelques lignes (max 5 lignes).

Q3 : Au regard des résultats obtenus lors du TP2, pouvez-vous justifier d'éventuelles similitudes/divergences comportementales entre dijkstra, BlowFish, SSCA2-BCS, SHA-1 et le produit de polynômes ?

2. Evaluation des performances :

- Cortex A7 : A taille de cache L2 fixe (512KB) et en faisant varier simultanément la taille des caches L1 d'instructions et de données de 1KB à 16KB (i.e. : 1KB, 2KB, 4KB, 8KB, 16KB), effectuez une évaluation de performances en utilisant le simulateur **Gem5 en out-of-order**.

Q4 : Générez les figures de performances détaillées (performance générale, IPC, hiérarchie mémoire, prédiction de branchement, etc.) en fonction de la taille du cache L1 pour les configurations testées. Analysez les résultats. Quelle configuration de L1 donne les meilleures performances pour le Cortex A7 pour les applications sélectionnées ?

N.B. : Mentionnez les paramètres d'exécution de Gem5 que vous avez utilisés.

- Cortex A15 : A taille de cache L2 fixe (512KB) et en faisant varier simultanément la taille des caches L1 d'instructions et de données de 2KB à 32KB (i.e. : 2KB, 4KB, 8KB, 16KB, 32KB), effectuez une évaluation de performances en utilisant le simulateur **Gem5**

Q5 : Générez les figures de performances détaillées (performance générale, IPC, hiérarchie mémoire, prédiction de branchement, etc.) en fonction de la taille du cache L1 pour les configurations testées. Analysez les résultats. Quelle configuration de L1 donne les meilleures performances pour le Cortex A15 pour les applications sélectionnées ?

N.B. : Mentionnez les paramètres d'exécution de Gem5 que vous avez utilisés.

3. Efficacité surfacique :

La surface du Cortex A15 de l'énoncé (Tableau 13) et de ses caches L1 est de **2 mm²** pour une technologie de **28 nm**. De même, la surface du Cortex A7 de l'énoncé et de ses caches L1 est de **0.45 mm²** dans cette même technologie de **28 nm**. Utilisez l'outil CACTI de HP avec la technologie convenable pour estimer la surface des caches pour chaque type de processeur. Il est recommandé de télécharger la dernière version de CACTI (CACTI 7.0 : <https://github.com/HewlettPackard/cacti>) sur vos machines, car il s'agit de la version la plus récente, et génère donc les chiffres d'estimation de caches les plus précis. L'outil CACTI est exécuté avec la commande suivante :

`./cacti -infile cache.cfg`

N.B. : *Si votre version ne supporte pas la technologie 28nm, ciblez la technologie 32nm.*

Q6 : Observez le fichier de configuration de cache « cache.cfg ». Quels sont les paramètres de cache (taille de cache, de bloc, associativité) et la technologie en **nm** utilisés par défaut ?

Q7 : Quelle est la surface des caches L1 du Tableau 12 Paramètres de configuration des processeurs (instructions et données) en **mm²** ? Quel pourcentage de la surface totale des cœurs A7 et A15 est occupé par les caches L1 ? En déduire la taille des deux cœurs (hors caches L1). Donnez votre analyse.

Q8 : Faire varier la taille des caches L1 dans l'intervalle de valeurs possibles pour le Cortex A7 et le Cortex A15, et donner (en **mm²**) les surfaces des caches L1 obtenues. Avec la configuration de cache L2 utilisée précédemment (512KB), en déduire pour chaque valeur les nouvelles surfaces totales du Cortex A7 et du Cortex A15, caches L2 inclus (présentez les résultats sous forme de graphes).

Q9 : En prenant en compte les deux dimensions (performance et surface) pour les deux processeurs considérés, donnez pour chaque configuration de L1 l'efficacité surfacique de chaque processeur.

N.B. :
$$Efficacité_{surfacique} = \frac{IPC}{surface(mm^2)}$$

4. Efficacité énergétique :

D'après [6], les consommations énergétiques du Cortex A7 et du Cortex A15 sont de **0.10 mW/MHz** et **0.20 mW/MHz** respectivement. De plus, en technologie 28 nm, les fréquences maximales du Cortex A7 et du Cortex A15 sont de **1.0 GHz** et **2.5 GHz** respectivement.

Q10 : Quelle puissance en **mW** consomme chaque processeur à la fréquence maximale ?

Q11 : Avec le même protocole que précédemment, et en prenant en compte les deux dimensions (énergie et surface) pour les deux processeurs considérés, donnez pour chaque configuration de L1 l'efficacité énergétique de chaque processeur (à fréquence maximale).

N.B. :
$$\text{Efficacité énergétique} = \frac{IPC}{\text{consommation énergie (mW)}}$$

5. Architecture système big.LITTLE :

Q12 : Avec un esprit de concepteur de système, et en se basant sur les résultats de Q10 et Q11, proposez la meilleure configuration du cache L1 du processeur big.LITTLE pour les applications Dijkstra et BlowFish **individuellement**.

6. Facultatif :

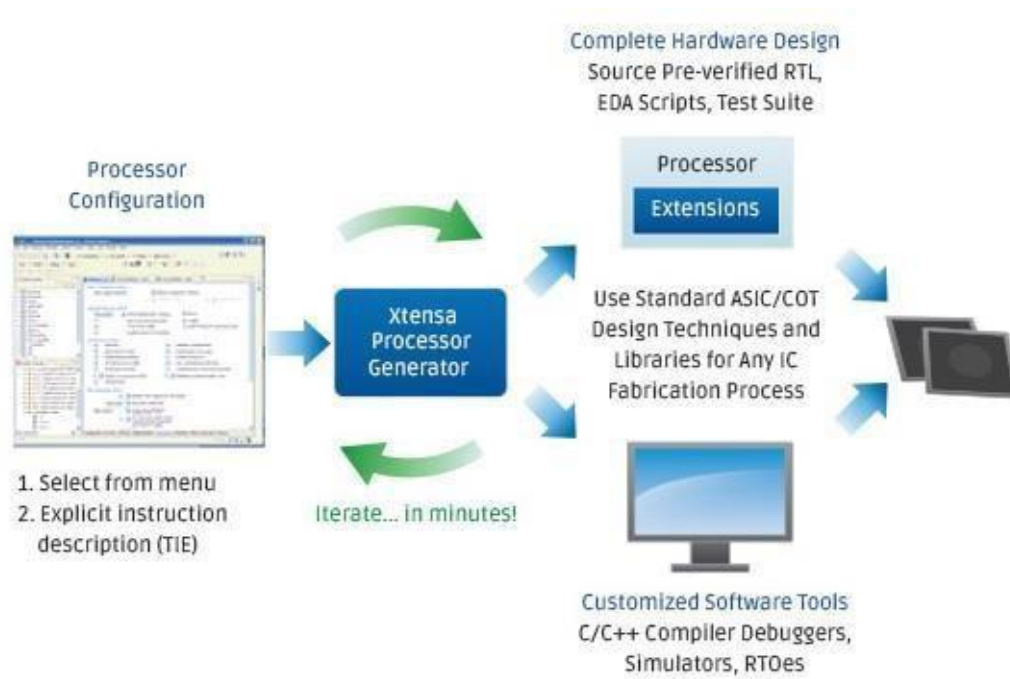
Q13 : Les configurations proposées sont-elles équivalentes ? Proposer éventuellement un compromis et conclure sur les applications étudiées.

Q14 : Proposez une approche pour la spécification d'une architecture avec plusieurs applications dans un domaine spécifique.

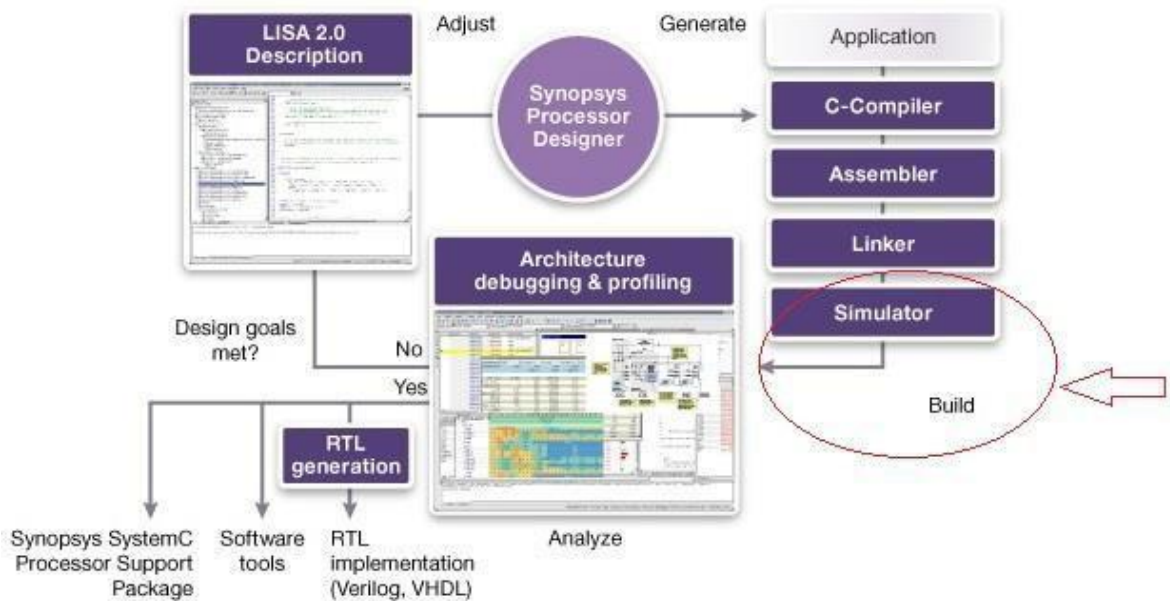
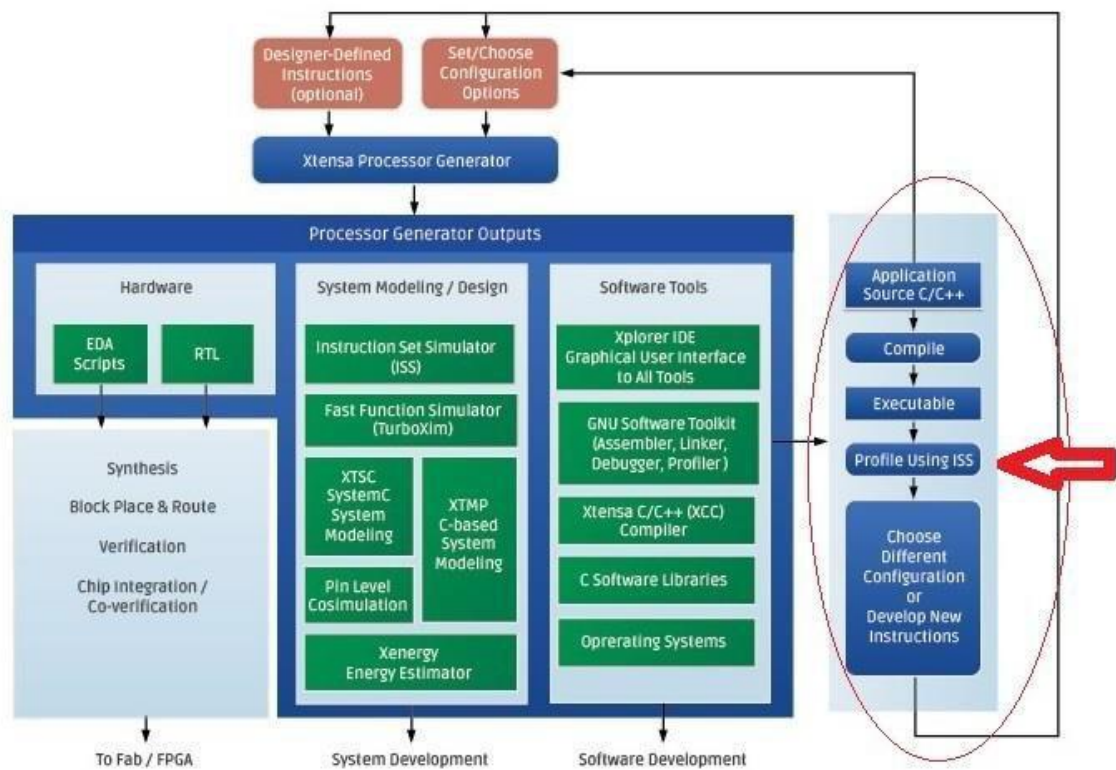
Remettre un rapport de TP par **quadrinôme** en version électronique **au format PDF (TP4-nom1-nom2-nom3-nom4.pdf)** incluant l'ensemble des réponses aux questions précédentes pour le **23/02/2026** aux adresses emails hammami@ensta.fr et celle de votre chargé de TD. avec en sujet de message **ECE_4ES01_TA/TP4**.

La note du rapport comptera pour 15% de la note globale.

N.B. : *La lisibilité et le format font partie de la note de votre rapport.*



Automated Customization Processor Overview



TD/TP5 : Analyse de performances de configurations de microprocesseurs multicoeurs pour des applications parallèles.

Le parallélisme au niveau instructions (*ILP : Instruction Level Parallelism*) est un niveau de parallélisme qui a ses limites.

Une autre approche consiste à utiliser des architectures capables d'exécuter plusieurs flots d'instructions indépendants simultanément sur plusieurs flots de données, en fonction de la disponibilité des ressources internes. Citons par exemple les architectures superscalaires de type SMT (Simultaneous Multithreading). Ces architectures exploitent le parallélisme de l'application au niveau des tâches (*TLP : Thread Level Parallelism*). Le Pentium4 HyperThreading est un exemple commercial de ce type de processeur [1].

Pourtant, la recherche de performances supplémentaires sur la base d'un processeur unique ne peut plus être poursuivie pour des raisons thermiques et de consommation d'énergie. Cela signifie la fin de l'augmentation des fréquences d'horloge.

L'alternative consiste à multiplier le nombre de processeurs sur une même puce en utilisant une fréquence d'horloge plus basse. La Figure 16 montre les architectures de processeurs (a) monoprocesseur scalaire in-order (b) monoprocesseur superscalaire out-of-order (c) monoprocesseur SMT (Simultaneous Multithreading) (d) CMP (Chip MultiProcessor/multicore).

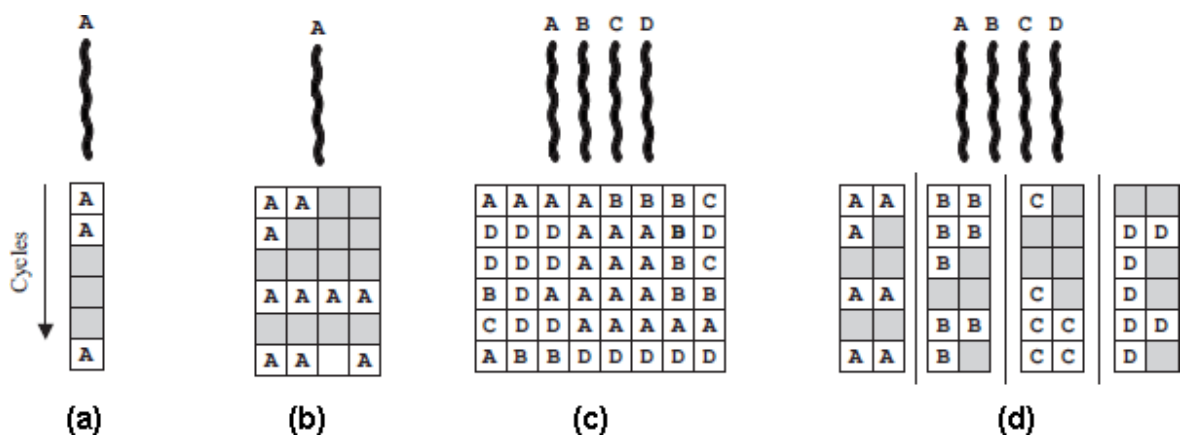


Figure 16 a) scalaire b) superscalaire c) SMT Simultaneous Multithreading d) CMP Chip MultiProcessor

IBM POWER6 [2], SUN UltraSPARC T2 [3], et Intel Ivy Bridge [4] sont des exemples commerciaux parmi plusieurs qui existent sur le marché.

Par contre, en multipliant le nombre de processeurs, des nouvelles questions et de nouveaux problèmes se posent :

- Type des cœurs de processeur (scalaire, superscalaire)
- Hiérarchie mémoire
- Synchronisation et communication entre les cœurs de processeur
- Cohérence de cache

Le but de ce TD/TP est d'explorer les architectures multiprocesseurs CMP en exécutant une application parallèle. En particulier, les performances par rapport au nombre de cœurs et le type de chaque cœur seront étudiées. Pour ce TD/TP, on va utiliser le simulateur gem5 (Annexes 2 et 3) pour générer un simulateur de type SMP (Symmetrical MultiProcessing) afin de simuler des architectures multiprocesseurs de type CMP.

Description de l'application multiplication de matrice parallèle

Pour ce TD/TP, on va utiliser l'application de multiplication de 2 matrices A et B, qui est parallélisée pour être exécutée par m threads. A et B sont 2 matrices de taille $n * n$, et le résultat est stocké dans une matrice C.

La Figure 18 montre une description de cet algorithme pour 2 threads et une matrice de taille $4 * 4$

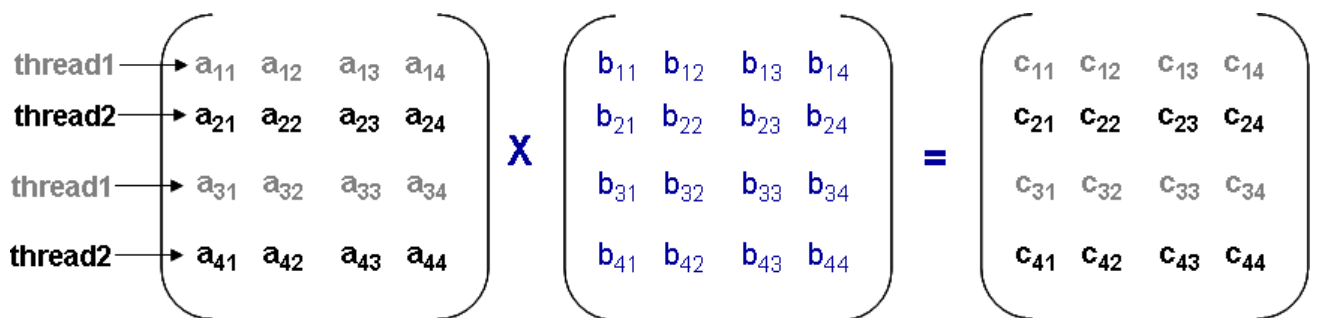


Figure 17 Multiplication de matrice 4x4 avec 2 threads

Le thread principal créé ('fork') $m-1$ threads secondaires indépendants. Ce pool de m threads exécute alors le calcul matriciel comme visible sur la Figure 19.

Puis, le thread principal attend la fin de l'exécution de tous les threads secondaires et considère que la multiplication des matrices a réussi.

Cette génération de threads secondaires est gérée dans la boucle **for** principale du programme **test_omp.cpp** grâce à une librairie de programmation parallèle très utilisée dans la communauté scientifique, OpenMP [6].

Dans le programme **test_omp.cpp** (compilé vers l'exécutable **test_omp**), le nombre de threads **m** ainsi que la taille des matrices **n** sont paramétrables.

Les paramètres d'entrée de l'application sont donc :

./test_omp <nthreads> <size>

<nthreads> : nombre de threads en parallèles (de 1 à $m < n$)

<size> : nombre de lignes et colonnes de la matrice carrée (**Note** : pour conserver des temps de simulations raisonnables, garder $n < 256$)

Pour récupérer le binaire **test_omp** :

```
$cp -v /home/c/nom-mdc/ES201/tools/TP5/test_omp /your/target/directory/
```

Attention : La librairie de programmation pthreads fournie pour votre TD/TP pour le mode « SE » de gem5 (**voir Annexes 2 et 3**) est toujours en cours de développement et impose pour le moment de conserver l'équivalence suivante : **n threads = n cœurs**.

Attention 2 : Se référer au fichier : /home/c/nom-mdc/ES201/tools/TP5/README expliquant la procédure (ainsi qu'aux annexes).

```
export GEM5=/home/c/nom-mdc/ES201/tools/TP5/gem5-stable
```

L'exécution du programme **test_omp** se fera donc de la manière suivante (avec ncores = nthreads) :

```
$GEM5/build/ARM/gem5.fast $GEM5/configs/example/se.py -n <ncores> -c test_omp -o "<nthreads> <size>"
```

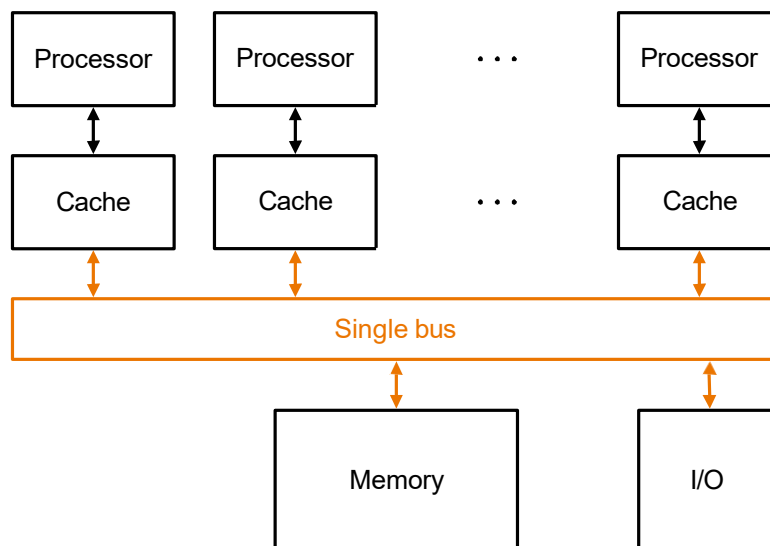


Figure 18 Architecture multicœurs à base de bus

Note : Pensez à vous référer aux Annexes 2 et 3 ainsi qu'à l'aide de l'exécutable pour accéder à toutes les options d'exécution possibles (type de cpu, configuration des caches, etc.) :

`$GEM5/build/ARM/gem5.opt $GEM5/configs/example/se.py --help`

Questions

1. Analyse théorique de cohérence de cache

Q1 : En considérant que chaque thread s'exécute sur un processeur dans une architecture de type multicoeurs à base de bus et 1 niveau de cache (comme décrit Figure 21), décrivez le comportement de la hiérarchie mémoire et de la cohérence des caches pour l'algorithme de multiplication de matrices. On supposera que le thread principal se trouve sur le processeur d'indice 1.

2. Paramètres de l'architecture multicoeurs

Le simulateur gem5 permet de faire varier de très nombreux paramètres des éléments architecturaux (types de processeur et paramètres associés à un processeur donné, mémoires caches L1, L2, cohérence de cache, etc.). Tous ces paramètres ne sont pas « accessibles » depuis les options de configuration système **se.py**, mais il est pourtant possible de les faire varier.

Q2 : Examinez le fichier de déclaration d'un élément de type « processeur superscalaire out-of-order », et présentez sous forme de tableau cinq paramètres configurables de ce type de processeur avec leur valeur par défaut. Choisissez de préférence des paramètres étudiés lors des séances TD/TP précédentes. Le fichier à consulter est le suivant :

`$GEM5/src/cpu/o3/O3CPU.py`

Q3 : Examinez le fichier d'options de la plateforme **se.py**, puis déterminez et présentez sous forme de tableau les valeurs par défaut des paramètres suivants :

- Cache de données de niveau 1 : associativité, taille du cache, taille de la ligne
- Cache d'instructions de niveau 1 : associativité, taille du cache, taille de la ligne
- Cache unifié de niveau 2 : associativité, taille du cache, taille de la ligne

Le fichier d'options à consulter est le suivant :

`$GEM5/configs/common/Options.py`

3. Architecture multicoeurs avec des processeurs superscalaires in-order (Cortex A7)

On se propose dans cette partie d'étudier une architecture multiprocesseur de type CMP à base de cœurs équivalents au Cortex A7 étudié dans le TD/TP5. Dans notre simulateur gem5, le modèle de CPU associé à ce type de processeur est le modèle **arm_detailed** (`--cpu-type=arm_detailed`).

En fixant la taille de la matrice à **m**, et en faisant varier le nombre de threads parallèles de l'application (nombre de threads = 1, 2, 4, 8, 16, ..., **m**), répondre aux questions suivantes :

- Q4 :** Déterminez quel est le processeur exécutant toujours le plus grand nombre de cycles. Expliquez pourquoi. Expliquez également pourquoi l'analyse du nombre de cycles sur ce processeur revient à analyser le nombre total de cycles d'exécution de l'application.
- Q5 :** Pour chaque configuration, quel est le nombre de cycles d'exécution de l'application ? Vous pourrez présenter vos résultats sous forme de graphe 2 axes
- Q6 :** Dédurre le speedup par rapport à la configuration à 1 thread.
- Q7 :** En utilisant le nombre total d'instructions simulées, déterminez quelle est la valeur maximale de l'IPC pour chaque configuration ?
- Q8 :** Discussion et interprétation (*max. 10 lignes*).

4. Architecture multicoeurs avec des processeurs superscalaires out-of-order (Cortex A15)

On se propose dans cette partie d'étudier une architecture multiprocesseur de type CMP à base de cœurs équivalents au Cortex A15 étudié dans le TD/TP5. Dans notre simulateur gem5, on dispose d'un modèle de processeur superscalaire out-of-order : le modèle **o3** (`--cpu-type=detailed`)

En fixant la taille de la matrice à **m**, et en faisant varier le nombre de threads parallèles de l'application (nombre de threads = 1, 2, 4, 8, 16, ..., **m**), et la largeur du processeur superscalaire (nombre de voies = 2, 4, 8), répondre aux questions suivantes :

- Q9 :** Pour chaque configuration, quel est le nombre de cycles d'exécution de l'application ? Vous pourrez présenter vos résultats sous forme de graphe 3 axes.
- Q10 :** Dédurre le speedup par rapport à la configuration à 1 thread.
- Q11 :** En utilisant le nombre total d'instructions simulées, déterminez quelle est la valeur maximale de l'IPC pour chaque configuration ?
- Q12 :** Discussion et interprétation (*max. 10 lignes*).

5. Configuration CMP la plus efficace

Q13 : Proposez une configuration ou une gamme de configuration de l'architecture CMP (nombre de threads de l'application **test_omp**, nombre et type de cœurs) qui vous semble la plus appropriée si la contrainte recherchée par le concepteur du système est l'efficacité surfacique ? Discussion et interprétation (*max. 10 lignes*).

N.B. : Vous vous appuyerez sur les résultats des deux TD/TP (4 et 5).

6. Facultatif :

En utilisant des tailles de matrices relativement grandes (taille des lignes et des colonnes en octets supérieure à la taille du cache de données de niveau 1) ou des tailles de caches de données relativement petites, un grand nombre de threads peut parfois faire apparaître un comportement « supra-linéaire » (comportement pour lequel le speedup est supérieur à n , où n est le nombre de threads).

Q14 : Au regard de l'évolution théorique du speedup et son évolution constatée lors des questions précédentes, proposez une tentative d'explication (*max. 10 lignes*).

Remettre un rapport de TP par **quadrinôme** en version électronique **au format PDF** (**TP5-nom1-nom2-nom3-nom4.pdf**) incluant l'ensemble des réponses aux questions précédentes pour le **23/02/2026** aux adresses emails hammami@ensta.fr et celle de votre chargé de TD. avec en sujet de message **ECE_4ES01_TA/TP5**.

La note du rapport comptera pour 15% de la note globale.

N.B. : La lisibilité et le format font partie de la note de votre rapport.

```
// test_omp.cpp
// Good old matrix multiply using openmp

#include <assert.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int64_t* A;
int64_t* B;
int64_t* C;

int main(int argc, const char** argv) {

    if (argc != 3) {
        printf("Usage: ./test_omp <nthreads> <size>\n");
        exit(1);
    }

    int nthreads = atoi(argv[1]);

    if (nthreads < 1) {
        printf("nthreads must be 1 or more\n");
        exit(1);
    }

    int size = atoi(argv[2]);

    if (size < 1) {
        printf("size must be 1 or more\n");
        exit(1);
    }

    printf("Setting OMP threads to %d\n", nthreads);
    omp_set_num_threads(nthreads);

    A = (int64_t*) calloc(size*size, sizeof(int64_t));
    B = (int64_t*) calloc(size*size, sizeof(int64_t));
    C = (int64_t*) calloc(size*size, sizeof(int64_t));

    printf("Starting with row/col size=%d\n", size);

    for (int x = 0; x < size; x++) {
        for (int y = 0; y < size; y++) {
            A[x*size + y] = x*y;
        }
    }
    printf("A initialized\n");

    for (int x = 0; x < size; x++) {
        for (int y = 0; y < size; y++) {
            B[x*size + y] = x*y - y;
        }
    }
    printf("B initialized\n");
    printf("Computing A*B with %d threads\n", nthreads);

    #pragma omp parallel for
    for (int x = 0; x < size; x++) {
        for (int y = 0; y < size; y++) {
            int64_t tot;
            for (int m = 0; m < size; m++) {
                tot += A[x*size + m]*B[m*size + y];
            }
            C[x*size + y] = tot;
        }
    }

    printf("Done\n");
    return 0;
}
```


TD/TP6 : Microprocesseurs et Energie



Exercice 1 : Lois Physiques Energie – Puissance

Q1 : 2 milliards de téléphones portables dans le monde avec des utilisations en mode idle ?

Q2 : 2 milliards de téléphones portables dans le monde avec des utilisations en mode actif ?

Q3 : Consommation équivalente en éoliennes ?



Q4 : consommation équivalente en champ solaires ?



Exercice 2 : Mode de gestion de l'énergie des microprocesseurs (ARM) et FSM

Q1. Citer 3 méthodes matérielles pour réduire la consommation énergétique dans les multi-cœurs

Q2. Pour chaque méthode, quelles sont les implications énergétiques ? (Statique, Dynamique, ..., Timing, Area.) Faire un tableau de comparaison

Q3. Dans la documentation cortex A9, quelles sont les techniques mises en œuvre et comment le sont-elles ?

Exercice 3 : Microélectronique

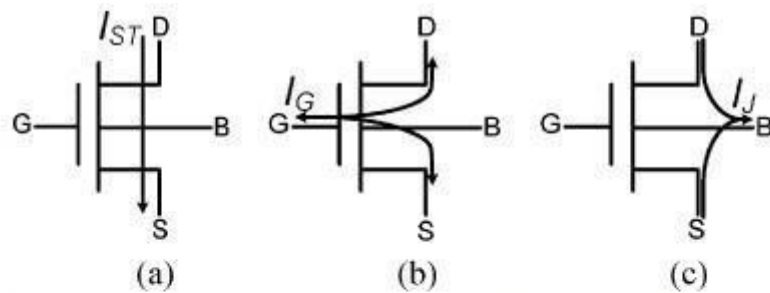


Fig. 2. NMOS transistor current contributions in subthreshold. (a) Subthreshold current. (b) Gate current. (c) Junction current.

TABLE I
NMOS/PMOS TRANSISTOR PARAMETERS (65-NM, STD- V_{TH}) [17]

	n	I_0 [A]	V_{TH0} [V]	λ_{DS}	λ_{BS}
NMOS	1.39	6.65E-5	0.598	9.0E-2	9.9E-2
PMOS	1.27	5.95E-6	0.532	8.0E-2	1.1E-1

Q1 : Technologie actuelle / ordre de grandeur du nombre de transistors dans un processeur récent

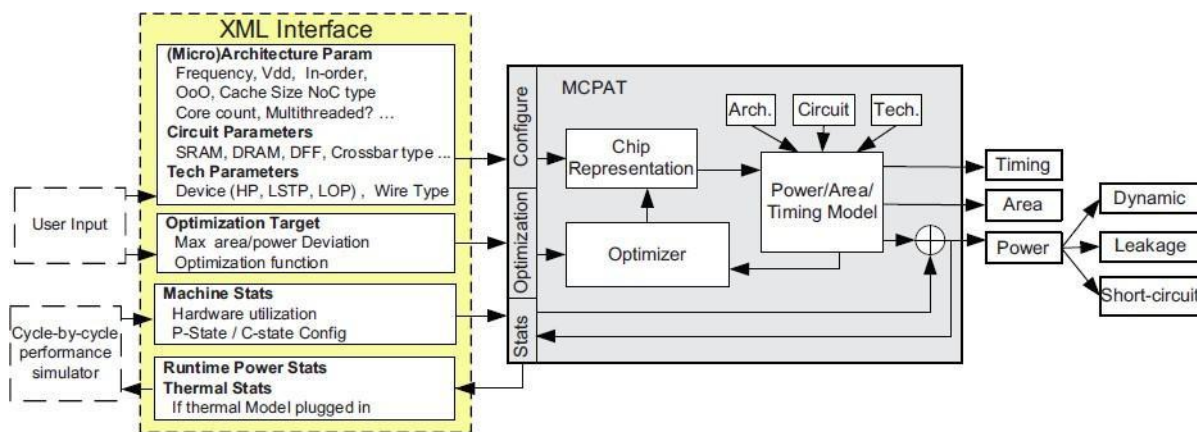
Q2 : Qu'est-ce que la conduction en régime de faible inversion et pourquoi est-ce utilisé ? Principaux "tuning knobs" accessibles ?

Q3 (opt.) : Quels sont les effets les plus pénalisants pour le transistor MOS ?

Sources et références :

Exercice 4 : Evaluation de performances – McPAT

On recherche à optimiser le processeur choisi, afin de réduire la consommation énergétique globale du système. Dans ce cas d'étude, nous nous intéressons à l'intérêt d'avoir une unité flottante dans le processeur. Le cas d'étude s'appuie sur les applications SSCA2 et PageRank (du benchmark miBench) sur un processeur de type ARM cortex A9. Pour avoir les informations de performance, le simulateur Gem5 est utilisé. Celui-ci est « connecté » à l'outil McPat permettant d'avoir les informations de surface et consommation énergétique du processeur.



Block diagram of the McPAT framework.

Commandes utiles :

```
export HOME_TOOLS=/home/c/nom-mdc/ES201/tools/TP6/tools
cp -r /home/c/nom-mdc/ES201/tools/TP6/bin/ /your/path/
cd /your/path/bin/<app>
$HOME_TOOLS/run_all.sh -b <app> (-f pour floating point)
<app> : [SSCA2 | pagerank]
```

Q1 Simuler PageRank et SSCA2 avec et sans l'extension Floating point. Lancer le simulateur et l'émulateur énergétique. Quel est l'accélération à utiliser l'unité flottante ? Ya-t-il un gain énergétique ? Regarder dans m5out/.xml et m5out/*.xml_simple*

SSCA2	PageRank
Basic	Basic
J in sec	J sec
core_area	core_area
core_leakage	core_leakage
core_runtime_dynamic	core_runtime_dynamic
core_energy	core_energy

fpu J in sec core_area core_leakage core_runtime_dynamic core_energy	fpu J sec core_area core_leakage core_runtime_dynamic core_energy
--	---

(Résultat d'énergie et de surface comprenant le cœur et le cache L1)

	SSCA2	PageRank
Speedup		
Energy gain		

Q2 Quelle est votre analyse pour chaque application ?

Q3 Si les 2 applications doivent tourner sur la même plateforme quelles méthodes peut-on employer pour réduire l'énergie ? Proposer des idées dans le cadre d'un mono-cœur et d'un multi-cœur. Les solutions de l'exercice 1 sont-elles utilisables ?

Solution TD/TP1 : Evaluation de performances - programmation assembleur RISC-V

Exercice 1 : Evaluation de performances analytique - Loi d'Amdahl

Nous souhaitons améliorer les performances d'un microprocesseur en réduisant d'un facteur 5 le temps d'exécution des instructions flottantes. Nous souhaiterions évaluer l'impact d'une telle transformation sachant que le temps d'exécution d'un benchmark avant l'amélioration prend 10 secondes et que 50% du temps est dépensé dans l'exécution de calcul flottant.

Q1 : Quel serait le speed up ?

$$S_T = \frac{1}{(1-P) + (P/S_P)}$$

A1 : Suivant l'équation d'Amdahl, le speedup total est égal à :

On obtient pour **P = 0.5** et **S_P = 5**

$$\Rightarrow S_T = [(1-0.5) + 0.5/5]^{-1}$$

$$\Rightarrow S_T = 1.66$$

Exercice 2 : Evaluation de performances analytique - Loi d'Amdahl

Nous souhaiterions améliorer les performances d'un microprocesseur et il existe 2 moyens :

2. Réduire le temps d'exécution des instructions de multiplication d'un facteur 4.
3. Réduire le temps d'exécution des instructions d'accès mémoire d'un facteur 2.

L'exécution du programme benchmark prend 100 secondes et utilise :

- 20 % des instructions pour la multiplication,
- 50% des instructions pour les accès mémoire,
- 30% des instructions pour les autres types d'opérations.

Q1 : Quel est le speedup si l'on améliore uniquement la multiplication ?

A1 : Le speedup si l'on améliore uniquement la multiplication pour **P = 0.2** et **S_P = 4** est égal à :

$$\Rightarrow S_T = [(1-0.2) + 0.2/4]^{-1}$$

=> **$S_T = 1.17$**

Q2 : *Quel est le speedup si l'on améliore uniquement les accès mémoires ?*

A2 : Le speedup si l'on améliore uniquement les accès mémoire pour **P = 0.5** et **S_p = 2** est égal à :

$$\Rightarrow S_T = [(1-0.5) + 0.5/2]^{-1}$$

$$\Rightarrow S_T = \mathbf{1.33}$$

Q3 : *Quel est le speedup si l'on améliore les deux ?*

A3 : La généralisation de la loi d'Amdahl pour prendre en compte plusieurs améliorations est

$$S_T = \frac{1}{(1 - \sum_{i=0}^n P_i) + (\sum_{i=0}^n P_i / S_{p_i})}$$

donnée par :

Le speedup si l'on améliore la multiplication et les accès mémoire pour $P = 0.5$ et $S_p = 2$ est égal à :

$$\Rightarrow S_T = [(1-0.2-0.5) + (0.2/4 + 0.5/2)]^{-1}$$

$$\Rightarrow S_T = 1.66$$

Exercice 3 : Evaluation de performances – simulation

Nous souhaitons mesurer les performances d'un programme en utilisant un **simulateur de processeur RISC-V** (gem5 en mode *SE – System Call Emulation*). Les informations concernant l'installation de Gem5 se trouvent en Annexe 3.1.

Pour cela, nous allons écrire un programme C compilé en binaire RISC-V puis simuler son exécution.

3. Ecrire un programme en C effectuant la somme de deux vecteurs de taille $N = 50$ que vous initialiserez de manière aléatoire.

4. Compilez ce programme avec

```
riscv64-linux-gnu-gcc -O0 -static -o vadd.riscv vadd.c
```

Lancer le simulateur avec un CPU in-order (MinorCPU) :

```
cd ~/gem5
build/RISCV/gem5.opt ~/TP/RISCV_se.py -b ~/TP/TP1/vadd.riscv
```

Les résultats se trouvent dans 'm5out/stats.txt' et peuvent être extraits par :

```
grep -E "simInsts|numCycles|ipc|cpi" m5out/stats.txt
```

Complétez Tableau 1 avec les valeurs suivantes :

Q1 : *Quel est le nombre total d'instructions exécutées par votre programme sur ce processeur ? **simInsts***

A1 : **simInsts = 7180**

Q2 : *Quel est le nombre total de cycles processeur nécessaires pour l'exécution du programme ? **numCycles***

A2 : **numCycles = 15601**

Q3 : *Quel est le CPI de votre programme sur ce processeur ? **CPI***

A3 : **CPI = Cycle Per Instruction = # clock cycles / # of instructions**

\Rightarrow **core.cpi = sim_cycles / sim_total_insn = 15601 / 7180 = 2.173**

Q4 : Quel est l'IPC de votre programme sur ce processeur ? IPC

A4 : $IPC = \text{Instruction Per Cycle} = 1 / CPI$

=> $\text{core.ipc} = 1 / \text{sim_CPI} = 0.46$

Tableau 1 Evaluation de performance

Paramètres	Valeur
nombre total d'instructions exécutes	7180
nombre total de cycles processeur	15601
CPI	2.173
IPC	0.46

N.B : Les résultats dépendent de la manière avec laquelle les vecteurs ont été programmés et initialisés.

Q5 : Pour un processeur ayant une fréquence d'horloge de 3 GHz, quelle est la durée d'exécution de votre programme ?

A5 : Une fréquence f de 3 GHz correspond à une période $T = 1/f = 1/3\text{GHz} = \mathbf{0.333\text{ ns}}$

=> La durée d'exécution du programme est égale à : # clock cycles * $T = 11259 * 0.333\text{ ns} = \mathbf{5200.33\text{ ns}}$

Exercice 4 : Commentaire d'un programme en assembleur RISC-V

A l'aide du Tableau 2 du jeu d'instructions assembleur RISC-V, commentez le programme suivant et indiquez ce que fait la fonction $F(X, Y, Z)$.

On suppose que la variable X se trouve dans le registre $s1$, la variable Y dans $s2$ et la variable Z dans $s3$. Le résultat de la fonction $F(X, Y, Z)$ se trouve dans le registre $s7$.

add s7,s1,s1	=>	$s7 = s1 + s1$	=>	$s7 = 2X$
add s7,s7,s7	=>	$s7 = s7 + s7$	=>	$s7 = 4X$
add s7,s7,s3	=>	$s7 = s7 + s3$	=>	$s7 = 4X + Z$
add s7,s7,s3	=>	$s7 = s7 + s3$	=>	$s7 = 4X + 2Z$
add s7,s7,s2	=>	$s7 = s7 + s2$	=>	$s7 = 4X + Y + 2Z$

$F(X,Y,Z) = \mathbf{4X + Y + 2Z}$

Exercice 5 : Programmation en assembleur

A l'aide du Tableau 2 du jeu d'instructions RISC-V, programmez en assembleur MIPS la fonction $F(X,Y,Z) = X + 2Y - 3Z$

On suppose que la variable X se trouve dans le registre $s1$, la variable Y dans $s2$, la variable Z dans $s3$ et que le résultat de la fonction $F(X,Y,Z)$ se trouve dans le registre $s7$.

add s7, s3, s3	=>	$s7 = s3 + s3$	=>	$s7 = 2Z$
add s7, s7, s3	=>	$s7 = s7 + s3$	=>	$s7 = 3Z$
sub s7, s2, s7	=>	$s7 = s2 - s7$	=>	$s7 = Y - 3Z$
add s7, s2, s7	=>	$s7 = s2 + s7$	=>	$s7 = 2Y - 3Z$
add s7, s1, s7	=>	$s7 = s1 + s7$	=>	$s7 = X + 2Y - 3Z$

=> $F(X,Y,Z) = X + 2Y - 3Z$

Note pour exercice 4 et 5 :

L'utilisation des instructions de multiplication est bien sûr aussi possible, mais ces instructions ont en général un temps d'exécution plus long (entre 2 et 20 cycles). En effet, le temps de propagation à travers un multiplieur 32 bits (généralement multi-cycles) est supérieur au temps de propagation d'un additionneur 32 bits (1 cycle).

L'instruction 'sl' (shift left logical) peut aussi être utilisée au lieu de l'instruction 'add'.

Exercice 6 : Encodage en assembleur

A l'aide du Tableau 3 et du Tableau 4 des codages des instructions RISC-V, codez les instructions suivantes :

add s5, s6, s7

add => R format (voir Tableau 4)

(Note : R format = opération 2 registres sources vers 1 registre destination)

Field size	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
R-type	funct7	rs2	rs1	funct3	rd	opcode

op: indique que l'instruction est R format

rs1: première opérande source de l'instruction

rs2: seconde opérande source de l'instruction

rd: registre destination

funct: code de la fonction arithmétique pour le R-type

1)

funct7 = (000 0000) _{bin}

rs2 = s7 = x23 = (1 0111) _{bin}

rs1 = s6 = x22 = (1 0110) _{bin}

funct3 = (000) _{bin}

rd = s5 = x21 = (1 0101) _{bin}

opcode = (011 0011) _{bin}

codage instruction :

Funct7	x23	x22	Funct3	x21	opcode
000000	10111	10110	000	10101	0110011

and s6,s2,s5

and => R format (voir Tableau 4)

funct7 = (000 0000)_{bin}

rs2 = s5 = x21 = (1 0101)_{bin}

rs1 = s2 = x18 = (1 0010)_{bin}

funct3 = (111)_{bin}

rd = s6 = x22 = (1 0110)_{bin}

opcode = (011 0011)_{bin}

codage instruction :

Funct7	x21	x18	Funct3	x22	opcode
000000	10101	10010	111	10110	0110011

lw => I format (voir Tableau 4)

(Note : I format = opération 1 registre source et 1 immédiat vers 1 registre destination)

Field size	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
I-format	imm[11:0]		rs1	funct3	rd	opcode

imm[11:0] = (50)_{dec} = (0000 0011 0010)_{bin}

rs1 = s4 = x20 = (1 0100)_{bin}

funct3 = (010)_{bin}

rd = s2 = (1 0010)_{bin}

opcode = (00 0011)_{bin}

codage instruction :

50	X20	Funct3	rd	opcode
000000110010	10100	010	10010	0000011

Exercice 7 : Commentaire du programme en assembleur

Commentez le programme suivant et indiquez ce qu'il fait.

Soit A un vecteur de dimension N = 10. L'adresse se trouve dans le registre s5.

A l'initialisation : s5 => adresse de base du vecteur A

(s5+0) => contenu de la case mémoire pointée par s5

=> A[0] = (s5+0)

A[1] = (s5+4) => 4 cases mémoire pour pointer sur le mot suivant du vecteur

A[2] = (s5+8) => 8 cases mémoire pour pointer sur le mot suivant du vecteur

...

(Note : le pointeur vers A étant incrémenté de 4 en 4, on peut en déduire que A contient des

éléments de taille 32 bits...)

```

andi s3,s3,0      =>   s3 = s3 & 0      => s3 = 0
andi s1,s1,0      =>   s1 = s1 & 0      => s1 = 0
addi s1,s1,1      =>   s1 = s1 + 1      => s1 = 0 + 1    => s1 = 1
andi s11,s11,0    =>   s11 = s11 & 0    => s11 = 0
andi s2,s2,0      =>   s2 = s2 & 0      => s2 = 0
addi s2,s2,10     =>   s2 = s2 + 10     => s2 = 0 + 10  => s2 = 10
Loop : lw s8,0(s5) =>   Loop: label/étiquette permettant de réaliser un branchement.
                        A l'initialisation, 0(s5) = A[0]
                        => Chargement du premier élément du tableau dans le registre s8

```

s8 est registre de destination de l'opération de chargement mémoire au niveau du processeur.

Première boucle, s8 = A[0]

Deuxième boucle, s8 = A[1]

Dernière boucle, s8 = A[9]

```
add s11,s11,s8      =>   s11 = s11 + s8
```

Le registre s11 contient le résultat intermédiaire permettant de calculer la somme de tous les éléments du tableau.

A l'initialisation, s11 = 0

Première boucle, s11 = 0 + A[0]

Deuxième boucle, s11 = A[0] + A[1]

Dernière boucle, s11 = A[0] + A[1] + ... + A[9]

```
addi s5,s5,4        =>   s5 = s5 + 4
```

Le registre s5 est incrémenté de la taille d'un mot et contient l'adresse de l'élément suivant

```
sub s2,s2,s1        =>   s2 = s2 - 1
```

Le registre s2 est l'index de la boucle permettant de balayer les divers éléments du tableau A.

A l'initialisation, s2 = 10

Première boucle, s2 = 9

Deuxième boucle, s2 = 8

Dernière boucle, s2 = 0

Le registre s1 contient la constante 1 qui permet de décrémenter l'index de boucle

```
bne s2,s3, Loop      =>   branch on not equal : instruction de branchement
```

Tant que s2 ≠ s3, aller à l'étiquette « Loop » et exécuter les instructions entre « Loop » et bne.

Dès que s2 = s3, exécuter l'instruction suivante du programme (et donc la fin du programme, puisque la condition d'arrêt de la boucle est s2 = s3 = 0)

D'où s11 = somme (de i=0 à 9) de A[i]

Code équivalent en C:

```

i=10;           //équivalent à s2
sum=0;          //équivalent à s11
for (i; i > 0; i--)
    sum += A[i]

```

Exercice 8 : Programmation en assembleur MIPS (somme d'un tableau)

Ecrivez un programme en assembleur qui calcule la somme des éléments pairs dans un tableau A de dimension N = 10. L'adresse de A se trouve dans le registre \$s5

Similaire au programme précédent

$$s11 = A[0] + A[2] + A[4] + A[6] + A[8]$$

Il suffit de remplacer l'instruction **addi s1, s1, 1** par **addi s1, s1, 2** => $s1 = s1 + 2$

L'indice de boucle est alors décrémenté de 2 à chaque itération.

Il faut de plus remplacer l'instruction **addi s5, s5, 4** par **addi s5, s5, 8** afin de pointer 2 éléments/mots plus loin dans le vecteur A.

Solution TD/TP2 : Programmation assembleur RISC-V - profiling d'applications et ISS (instruction set simulator)

N.B#1: En **RISC-V**, la valeur du **PC (Program Counter)** n'est **pas directement accessible** par le programmeur. Elle est **implicitement modifiée** par le flot d'exécution et par les instructions de **saut et de branchement** (jal, jalr, beq, bne, etc.).

➔ À la différence de MIPS, RISC-V fournit cependant un accès **indirect** au PC via auipc, qui permet d'obtenir **PC + immédiat** dans un registre.

N.B#2: En **RISC-V**, **aucun registre général n'est "réservé" matériellement** comme en MIPS.

Cependant, la **convention d'appel (ABI)** impose des usages précis :

- x0 : constante zéro (lecture seule)
- x1 (ra) : adresse de retour
- x2 (sp) : pointeur de pile
- x3 (gp), x4 (tp) : registres système/runtime

➔ **Le programmeur peut techniquement les utiliser**, mais **c'est une mauvaise pratique** : cela casse les conventions, les appels de fonctions et l'OS.

Exercice 1 : Switch statement

Veillez écrire en assembleur l'équivalent du code C suivant :

```
switch(k) {
    case 0 : f = i + j ;break ; /* k = 0 */
    case 1 : f = g + h ;break ; /* k = 1 */
    case 2 : f = g - h ;break ; /* k = 2 */
    case 3 : f = i - j ;break ; /* k = 3 */
}
```

On supposera que les variables f, g, h, i, j et k se trouvent dans les registres s0-s5 respectivement.

***Q1:** Proposez 2 solutions dont une version où le temps d'exécution est indépendant du nombre de cas.*

Solution 1 : temps d'exécution dépendant du nombre de cas : if-then-else imbriqués

```
beq    s5, x0, CASE0      # if (k==0), goto CASE0
addi   t0, x0, 1           # t2 = 1
beq    s5, t0, CASE1      # if (k==1), goto CASE1
addi   t0, x0, 2           # t2 = 2
beq    s5, t0, CASE2      # if (k==2), goto CASE2
addi   t0, x0, 3           # t2 = 3
beq    s5, t0, CASE3      # if (k==3), goto CASE3
j      EXIT              # if (k<0 || k > 3), goto EXIT => wrong case condition
```

```
CASE0:
add    s0, s3, s4          # k=0 so f = i + j
j      EXIT               #end of this case so go to EXIT
```

```
CASE1:
add    s0, s1, s2          # k=1 so f = g + h
j      EXIT               #end of this case so go to EXIT
```

```
CASE2:
sub    s0, s1, s2          # k=2 so f = g - j
j      EXIT               #end of this case so go to EXIT
```

```
CASE3:
sub    s0, s3, s4          # k=3 so f = i - j
```

```
EXIT:                      #end of switch statement
```

Solution 2 : temps d'exécution indépendant du nombre de cas : *approche par indexation de table*

```

blt      s5, x0, EXIT      // test if k < 0 go to Exit
addi     t3, x0, 4          // t3 = 4
bge      s5, t3, EXIT      // test if k >= 4 go to Exit
slli     t1, s5, 2          // t1 = 4*k
add      t1, t1, x3         // global pointer + 4*k
lw jalr  t0, 0(t1)         // temp reg t0 =
                          JumpTable[k]

```

JumpTable :

```

.word CASE0
.word CASE1
.word CASE2
.word CASE3

```

CASE0:

```

add      s0, s3, s4         // k = 0 so f gets i + j
j        EXIT              // end of this case so go to
                          Exit

```

CASE1:

```

add      s0, s1, s2         // k = 1 so f gets g + h
j        EXIT              // end of this case so go to
                          Exit

```

CASE2:

```

sub      s0, s1, s2         // k = 2 so f gets g - h
j        EXIT              // end of this case so go to
                          Exit

```

CASE3:

```

sub      s0, s3, s4         // k = 3 so f gets i - j
j        EXIT

```

Discussion :

La solution à cet exercice est basée sur l'hypothèse qu'une table JumpTable de 4 éléments contient de manière consécutive les adresses des étiquettes de branchement (label) CASE0, CASE1, CASE2 et CASE3 associées aux 4 possibilités du **switch**. Le premier intérêt de cette approche utilisée par les compilateurs comparés à l'approche simple de tests successifs (transformation d'une structure switch en constructions if-then-else imbriquées) est le déterminisme du temps d'exécution. En effet, l'approche if-then-else imbriqués fait que le temps d'exécution de cette structure dépend de la valeur **k**, avec le temps le plus court pour **k = 0** et le temps le plus long pour **k = 3**. Par contre, l'approche par indexation de table n'est pas possible lorsque les valeurs pour **k** ne sont pas contiguës et il faut alors recourir aux if-then-else imbriqués.

Une dernière optimisation possible lorsque le **switch** est implémenté sous la forme de if-then-else imbriqués est de les imbriquer par probabilité d'occurrence. Cela suppose que le programmeur a une

connaissance suffisamment forte à priori de son application pour estimer ces probabilités. Ces trois manières d'implémenter le **switch** suivant la situation sont celles habituellement recommandées pour l'optimisation logicielle (cf. [1] pour exemple).

[1] *Software optimization guide for AMD Athlon 64 and AMD Opteron Processors*, Publication 25112 Rev. 3.05 November 2004. www.amd.com

Exercice 2 : Boucles

Veuillez écrire en assembleur l'équivalent du code C suivant :

```
sum = 0 ;
for(i= 0 ; i < 1000 ; i++) {
    for(j= 500 ; j > 0 ;j--) {
        for(k = 0 ; k < 300; k = k + 10) {
            sum = sum + (i + j + k);
        }
    }
}
```

On supposera que les variables sum, i, j et k se trouvent dans les registres \$s1-\$s4 respectivement.

```
addi    s1, x0, 0           // sum = 0
addi    s2, x0, 0           // i = 0
addi    t0, x0, 1000        // i max = 1000
addi    t1, x0, 500         // j max = 500
addi    t2, x0, 300         // k max = 300

Loop_I:
addi    s3, t1, 0           // j = 500
Loop_J:
addi    s4, x0, 0           // k = 0
Loop_K:
add      s1, s1, s2          // sum = sum + (i );
add      s1, s1, s3          // sum = sum + (i + j );
add      s1, s1, s4          // sum = sum + (i + j + k);
addi     s4, s4, 10          // k = k + 10
blt      s4, t2, Loop_K     // while (k < 300)
addi     s3, s3, -1          // j--
blt      s3, x0, Loop_J     // while (j > 0)
addi     s2, s2, 1           // i++
blt      s2, t0, Loop_I     // while (i < 1000)
Exit:
```

Une optimisation possible est de remarquer qu'au sein de la boucle d'indice **k** la partie **i+j** de l'expression **sum + (i + j + k)** reste inchangée et qu'il est donc possible d'économiser de nombreuses instructions d'additions inutiles en calculant une seule fois **i+j** en dehors de la boucle d'indice **k**.

Exercice 3 : Appels de fonctions

Veuillez écrire en assembleur l'équivalent du code C suivant :

```
int leaf(int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

On supposera que les variables g, h, i et j sont transmises par les registres \$a0, \$a1, \$a2, \$a3 lors de l'appel de fonction, et que f correspond à \$s0. Lors de l'exécution d'un appel de fonction, il est nécessaire de sauvegarder dans une pile en mémoire le contenu des registres utilisés. Pour rappel, le registre \$sp donne l'adresse de la pile.



Figure 19 Evolution de la pile lors d'un appel de fonction

L'exécution d'un appel de fonction est constituée de 3 étapes :

1. Avant l'appel (caller) : sauvegarde les registres caller-saved qu'il veut conserver, puis effectue l'appel (jal).
2. Dans la fonction (callee) : prologue : ajuste sp, sauvegarde les registres callee-saved qu'elle va modifier (et éventuellement ra), puis exécute le corps.
3. Retour (callee puis caller) : épilogue : restaure les registres sauvegardés, réajuste sp, retourne (jr ra). Le caller reprend ensuite.

La pile est implémentée de sorte à : (1) progresser (empiler) dans le sens des adresses décroissantes dans l'espace d'adressage mémoire (2) le pointeur de pile pointe sur le sommet de la pile qui correspond à l'adresse la plus basse actuellement occupée par la pile.

La Erreur ! Source du renvoi introuvable. montre la situation de la pile avant empilement (a) après

empilement (b) et enfin après dépilement (situation initiale) en (c).

On rappelle que le pointeur de pile **sp** est placé dans un registre du banc de registres mais que rien ne le différencie d'un point de vue physique des autres registres du banc de registres.

leaf_example:

```
addi    sp, sp, -12           // adjust stack to make room for 3 items
sw      s0, 0(sp)             // save register s0 for further use
sw      s1, 4(sp)             // save register s1 for further use
sw      s2, 8(sp)             // save register s2 for further use

add     s2, a1, a2            // register s2 contains g + h
add     s1, a3, a4            // register s1 contains i + j
sub     s0, s2, s1            // f = s2 - s1 which is (g+h) - (i+j)
addi    a0, s0, 0             // returns f (a0 = s0 + 0)

lw      s0, 0(sp)             // restore register s0 for caller
lw      s1, 4(sp)             // restore register s1 for caller
lw      s2, 8(sp)             // restore register s2 for caller
addi    sp, sp, 12            // readjust the stack pointer
jr      ra                    // jump back to calling routine (ra = return address)
```

Q1 : Pourquoi les appels de fonctions affectent les performances des programmes ?

A1 : Les appels de fonctions dégradent les performances car ils introduisent un surcoût fixe indépendant du calcul effectué : sauvegarde/restauration des registres (prologue/épilogue), modification du pointeur de pile, et instructions de saut (jal, jr) qui perturbent le flot de contrôle.

Ces sauts peuvent provoquer des pénalités de pipeline (flush, mauvaise prédiction de branchement) et empêcher certaines optimisations du compilateur (inlining, ordonnancement global). En conséquence, pour être rentable, le coût du corps de la fonction doit être significativement supérieur au coût de l'appel.

Q2 : Pourquoi la récursivité en programmation affecte les performances ?

A2 : La récursivité fait des appels récursifs à la même fonction (même interprétation qu'A1), d'où le compromis facilité de programmation vs performance. De plus, chaque récursion implique autant de sauvegardes sur la pile, ce qui peut être source d'une importante consommation mémoire.

Exercice 4 : Appels de fonctions

Veillez écrire en assembleur l'équivalent du code C suivant :

```
void swap(int v[], int k)
```



```

{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}

swap :
addi    sp, sp, -4           // adjust stack to make room for 1 item
sw      s0, 0(sp)           // save register s0 for use afterwards

slli    t0, a1, 2           // reg t0 = k * 4
add     t0, a0, t0          // reg t0 = &v[k]

lw      s0, 0(t0)           // temp = v[k]
lw      t1, 4(t0)          // t1 = v[k+1]

sw      t1, 0(t0)           // v[k] = v[k+1]
sw      s0, 4(t0)           // v[k+1] = temp

lw      s0, 0(sp)           // restore register s0 for caller
addi    sp, sp, 4           // readjust the stack pointer
jr      ra                 // return to calling routine

```

Exercice 5 : Profiling d'applications (% d'instructions par catégorie) par simulation sur ISS (instruction set simulator)

Nous allons maintenant nous intéresser au profiling d'une application tirée de l'inférence de réseaux de neurones convolutif (CNN). Ceux-ci utilisent de nombreuses opérations de convolution afin d'appliquer des filtres sur les données et extraire des informations importantes pour décrire ces données. Plusieurs déclinaisons sont proposées pour l'opération de convolution :

1. La convolution d'une matrice d'entiers 320x320 représentant une image avec un filtre 4x4.
2. La convolution d'une matrice de flottants 320x320 représentant une image normalisée avec un filtre 4x4.
3. La convolution d'une matrice d'entiers 320x320 représentant une image avec un filtre 4x4 avec un déroulage de boucle permettant une première optimisation des instructions.

Q1 : Générez le pourcentage de chaque classe d'instructions utilisées par ces programmes, puis remplissez le tableau suivant

Tableau 13 Pourcentage de chaque classe d'instructions

Classe d'instructions	P1 (%)	P2 (%)	P3 (%)
Instructions de lecture (Load)	21.0	23.5	34.3
Instructions d'écriture (Store)	0.7	0.7	2.9
Instructions de branchement	13.8	15.4	2.9
Instructions de calcul entier (Int computation)	78.3	63.2	17.2
Instructions de calcul flottant (Fp computation)	0	12.5	45.7
Nombre total d'instructions exécutées	15 276 562	13 668 738	3 521 291

(N.B : Pour la multiplication de polynômes les résultats dépendent de la manière dont ont été programmés et initialisés la multiplication.)

Le produit de convolution a été compilé avec m5ops pour ne mesurer que la partie utile du code. Il a de plus été compilé avec l'option d'optimisation du compilateur -O2.

Q2 : Quelle catégorie d'instructions nécessiterait une amélioration de performances ?

A2 :

Le profiling d'applications a pour objectif de détecter quelles sont les instructions (et par conséquent les ressources) les plus utilisées lors de l'exécution d'un programme donné sur un microprocesseur donné.

Pour P1, les instructions de calcul entier prennent près de 80 % des instructions exécutées, dont 14% pour la gestion des branchements. Dans le cas de la convolution de matrices d'entiers, les opérations entre les entiers sont confondues avec les opérations de boucles. En fait, le processeur passe bien plus de temps à gérer les boucles que les accumulations de produits. Ceci est bien mis en évidence avec la version de flottants P2.

=> En l'état, un processeur superscalaire avec plusieurs ALUs pour paralléliser l'exécution des instructions de calcul entier et doté d'un bon prédicteur de branchement pourrait profiter à l'application. Cependant, une optimisation du code au niveau du compilateur et/ou une réécriture du code C afin de réduire cette proportion de calcul entier serait bienvenue dans un premier temps.

La réécriture des boucles (P3) pour simplifier le travail des branchements et des calculs des itérateurs a un impact très important sur le nombre total d'instructions exécutées. Cependant, on observe une augmentation des instructions de chargement des données. On compense ainsi une partie des calculs par un flux de mémoire. Le programme n'est plus cpu-bound mais trouve un équilibre qui réduit le temps de traitement du programme.

Solution TD/TP3 : Processeur pipeline – compilateur

Exercice 1 : Pipeline

Identifier toutes les dépendances de données dans le code suivant. Quelles dépendances sont des aléas de données qui seront résolus par *forwarding* ?

add \$2,\$5,\$4

add \$4,\$2,\$5

sw \$5, 100(\$2)

add \$3,\$2,\$4

Vous devez vous appuyer dans votre analyse sur la Figure 2 décrivant le pipeline du processeur.

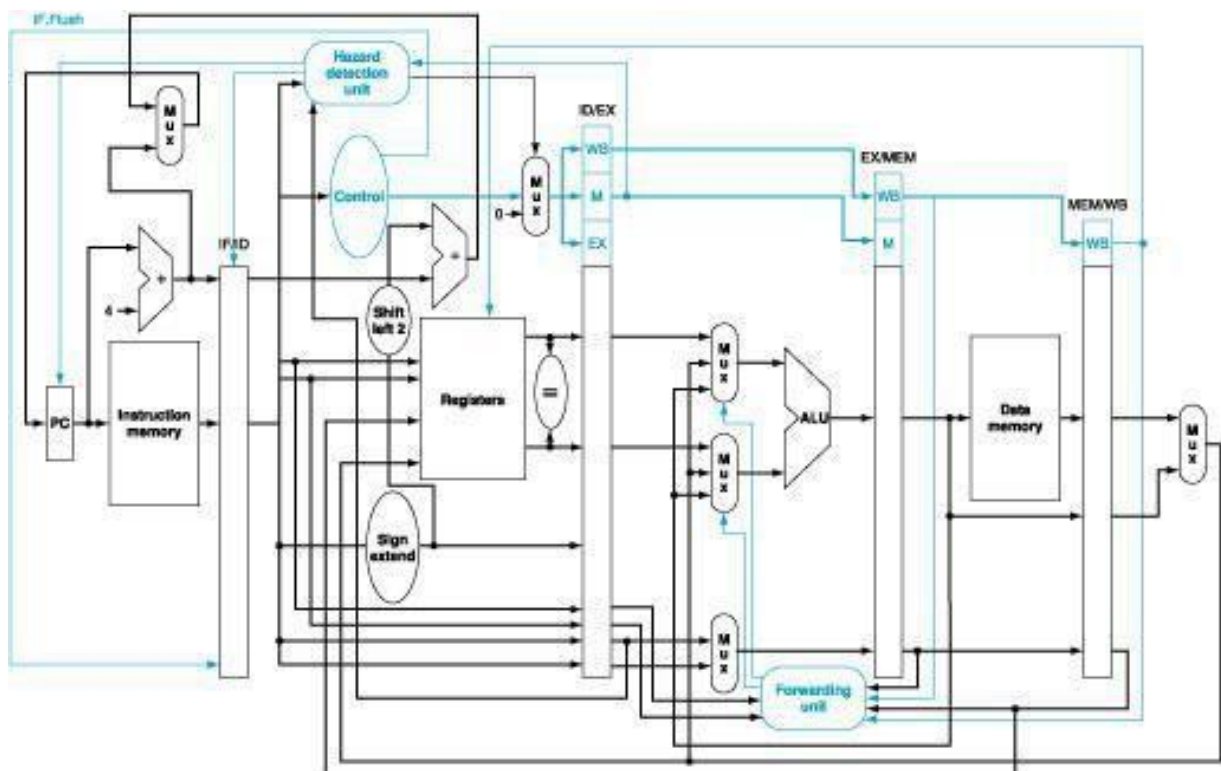


Figure 20 Processeur RISC pipeline 5 étage

Les dépendances de données existent dans les cas suivants :

RAW (Read After Write) : lecture après écriture, c'est-à-dire une instruction lit/utilise une donnée écrite/produite par l'instruction précédente. La manière de détecter cette situation est de rechercher si pour une instruction donnée le registre destination est utilisé en tant que source (opérande) lors des instructions suivantes.

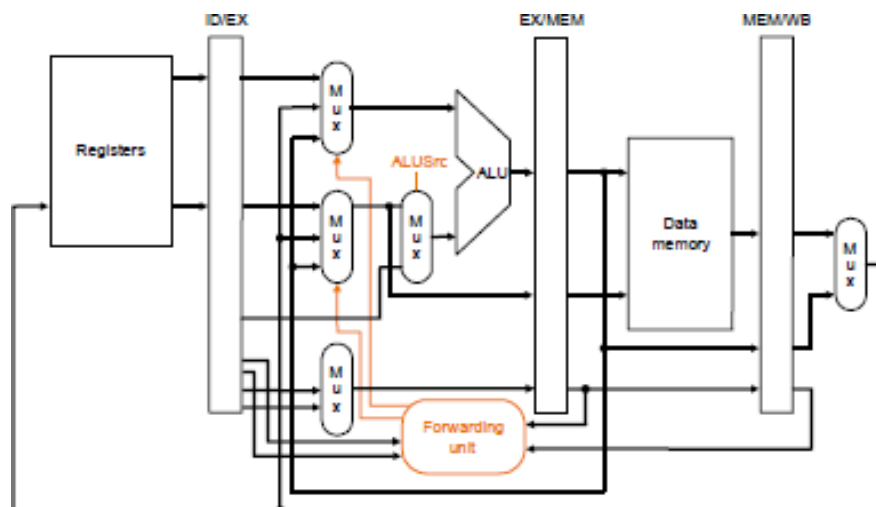
inst1 -> inst2 est une dépendance sur le registre \$2

inst1 -> inst3 est une dépendance sur le registre \$2

inst1 -> inst4 est une dépendance sur le registre \$2

inst2 -> inst4 est une dépendance sur le registre \$4

Le **forwarding** s'applique de la manière suivante :



et donc :

inst1 -> inst2 sera résolue par l'unité de forward lorsque S1 sera dans l'étage MEM et S2 sera dans EX

inst1 -> inst3 sera résolue par l'unité de forward lorsque S1 sera dans l'étage WB et S3 sera dans EX

inst1 -> inst4 sera résolue normalement les données étant présentes dans le banc de registres

inst2 -> inst4 sera résolue par l'unité de forward lorsque S2 sera dans l'étage WB et S4 sera dans l'étage EX

Exercice 2 : Pipeline - Data Forwarding

Nous souhaitons modifier le pipeline du processeur pour lui ajouter 2 nouvelles unités fonctionnelles :

1. Une unité de multiplication
2. Une unité racine carrée

Modifiez le chemin de données du processeur et l'unité de data forwarding pour prendre en compte ces deux nouvelles unités fonctionnelles.

Vous pouvez vous appuyer dans votre analyse sur la Figure 3 décrivant le data forwarding au sein du pipeline du processeur.

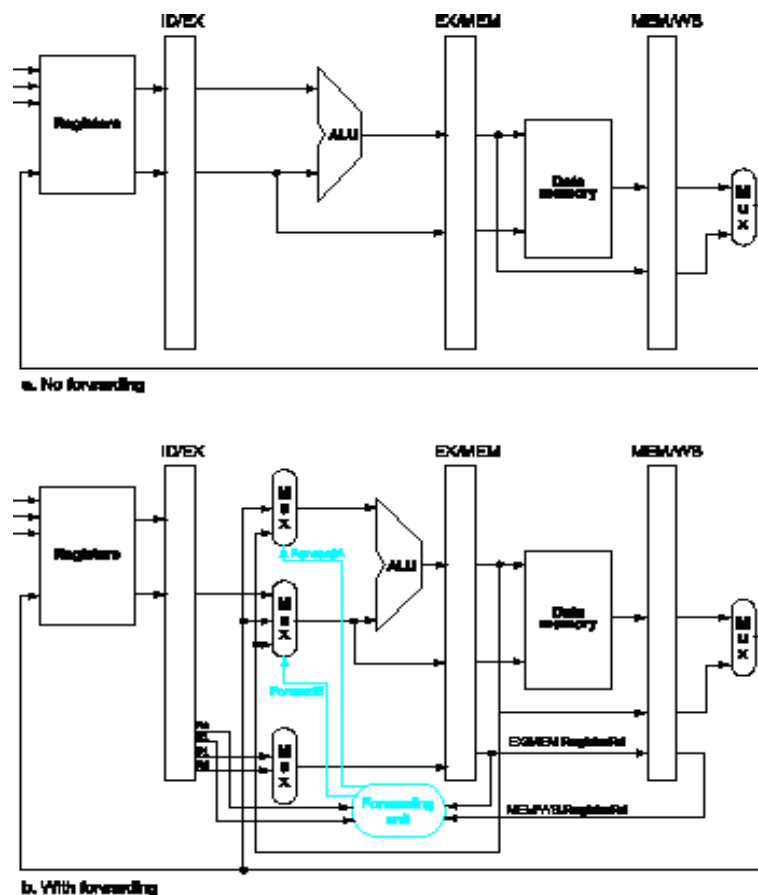
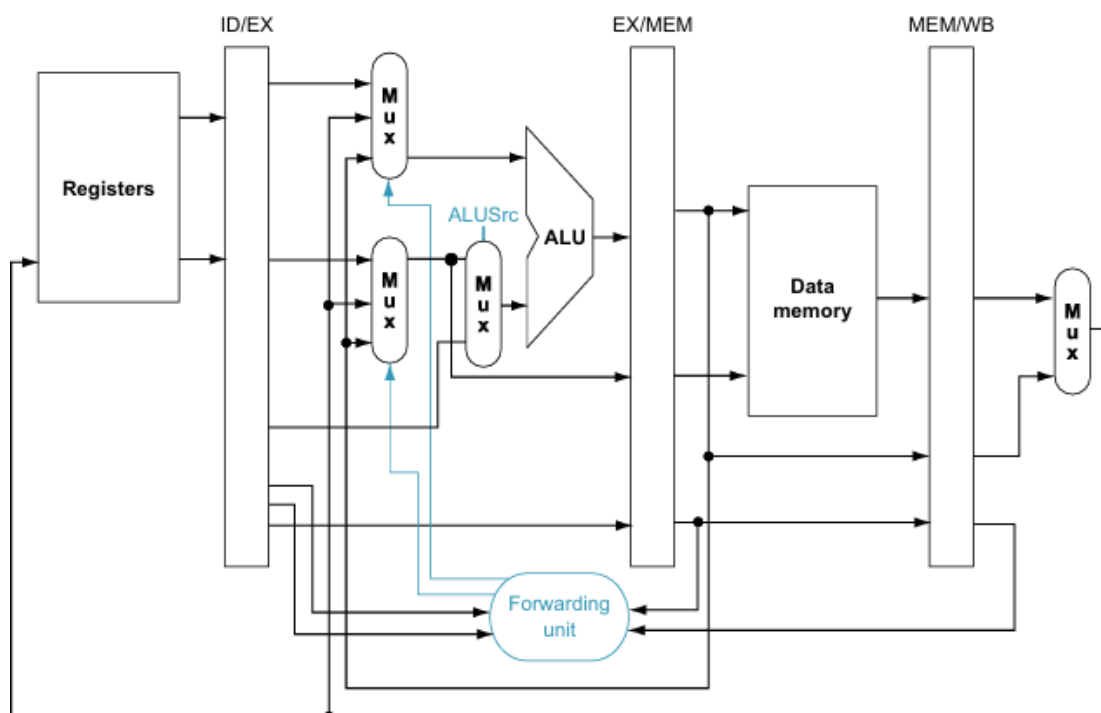


Figure 21 Data forwarding a) No forwarding b) With forwarding

Pour traiter l'unité racine carrée, on définit une nouvelle instruction personnalisée `sqrt rd, rs`, dont la sémantique est la suivante : le processeur calcule la racine carrée de la valeur contenue dans le registre source `rs` et place le résultat dans le registre destination `rd`.

Les instructions de multiplication sont conservées et suivent le modèle RISC-V standard : elles possèdent un registre destination explicite `rd`. La détection des dépendances de données s'effectue donc par comparaison des identifiants des registres source (`rs1`, `rs2`) et du registre destination (`rd`), comme pour les instructions de l'ALU. Les instructions de multiplication peuvent ainsi bénéficier du mécanisme de data forwarding et y contribuer, sans nécessiter de traitement particulier.

Par conséquent, l'ajout des unités fonctionnelles `MUL` et `SQRT` ne requiert pas de gestion de dépendances cachées ni de registres implicites. Le mécanisme de forwarding existant est simplement étendu afin de prendre en compte les résultats produits par ces nouvelles unités fonctionnelles.

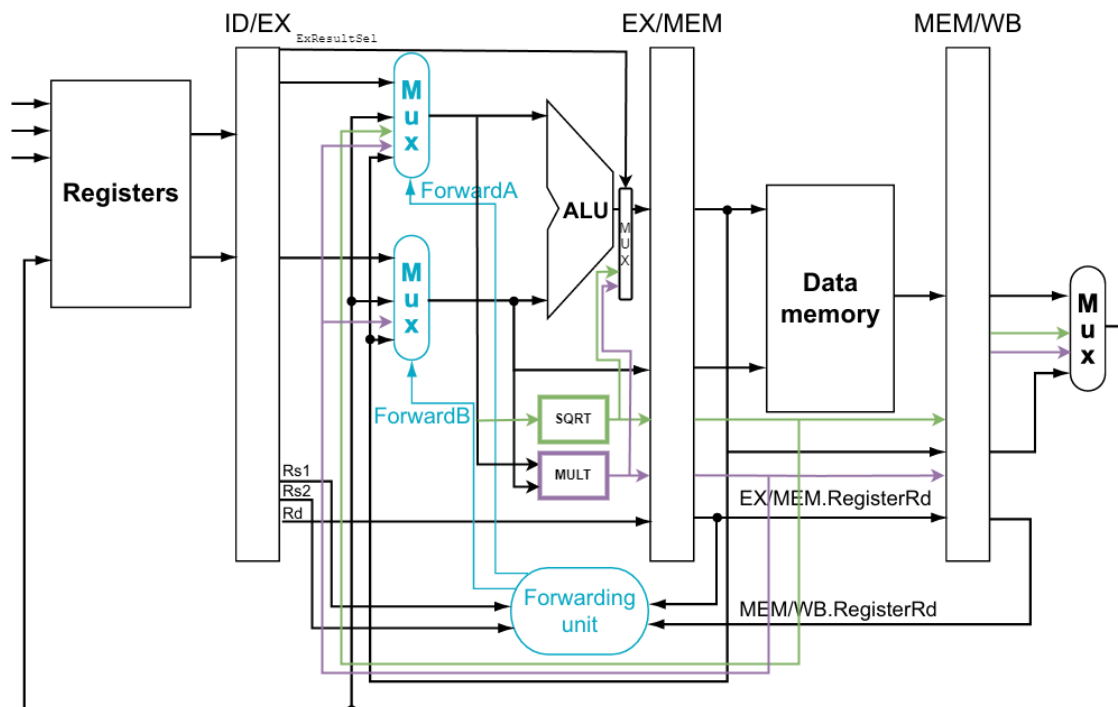


Les modifications principales à appliquer concernent :

- Ajout de deux nouvelles unités fonctionnelles dans l'étage EX : une unité de multiplication `MUL` et une unité de racine carrée `SQRT`. Les instructions de multiplication suivent le format et le comportement standard RISC-V et produisent un résultat dans un registre destination explicite `rd`. L'unité `SQRT` n'étant pas définie dans l'ISA de base, une instruction personnalisée `sqrt rd, rs` est introduite. Cette instruction calcule la racine carrée de la valeur contenue dans le registre source `rs` et place le résultat dans le registre destination `rd`. Elle est traitée comme une instruction arithmétique et utilise un format de type I, dont la partie immédiate n'est pas exploitée.
- La connexion des entrées et sorties de ces unités dans l'étage EX, ainsi que le routage de leurs résultats à travers les registres de pipeline EX/MEM et MEM/WB, jusqu'à l'écriture finale dans

le banc de registres. Un multiplexeur de sélection du résultat est ajouté ou étendu afin de choisir, selon l'instruction exécutée, entre le résultat de l'ALU, de l'unité MUL ou de l'unité SQRT avant l'étape de write-back.

- L'ajout des chemins de données de forwarding permettant d'acheminer les résultats produits par les différentes unités fonctionnelles (ALU, MUL, SQRT) depuis les étages EX/MEM et MEM/WB vers les entrées de l'étage EX, afin de résoudre les dépendances de données sans introduire de stalls inutiles.
- Ajout ou extension des multiplexeurs de sélection à l'entrée des unités fonctionnelles de l'étage EX, afin de permettre à l'unité de data forwarding de sélectionner correctement, pour chaque opérande source, soit la valeur issue du banc de registres, soit une valeur forwardée depuis un étage ultérieur du pipeline.
- L'unité de forwarding est augmentée pour prendre en compte les résultats des instructions de multiplication et de racine carrée, en comparant les registres destination rd des instructions présentes dans les étages EX/MEM et MEM/WB avec les registres source rs1 et rs2 de l'instruction en cours d'exécution.
- Étant donné que, en RISC-V, les instructions de multiplication et l'instruction personnalisée sqrt écrivent leur résultat dans un registre destination explicite, la détection des dépendances repose uniquement sur la comparaison des identifiants de registres, sans nécessiter l'analyse spécifique des codes opérations ni la gestion de registres implicites.



Les codes à vérifier sont les suivants :

Le premier code teste le forwarding entre les instructions de l'ALU et le multiplieur.

add s1, s3, s8

Figure 22 Architecture finale du pipeline


```
add    s2, s7, s9
mul    s10, s1, s2
add    s5, s5, s1
```

Et le code suivant :

```
sqrt   s1, s2
add    s4, s1, s3
mul    s5, s2, s3
add    s6, s6, s5
```

teste les trois unités. Le schéma donné ici donne un aperçu sur l'architecture générale.

Exercice 3 : Impact du nombre d'unités fonctionnelles

Dans gem5, le processeur DerivO3CPU permet de configurer finement le nombre et le type d'unités fonctionnelles à l'aide d'un pool d'unités fonctionnelles (FU pool).

--ialu	M	nombre total d'ALUs (unité arithmétique et logique) entière
--imult	M	nombre total de multiplieurs/diviseurs entiers
--fpalu	M	nombre total d'ALUs flottantes (données de type float)
--fpmult	M	nombre total de multiplieurs/diviseurs flottants (données de type float)
--mempport	M	nombre de ports mémoire, c'est-à-dire le nombre maximal d'accès mémoire (load/store) pouvant être émis par cycle

Ce nombre total M peut varier de 0 à M où M dans les processeurs actuels ne dépasse pas 8, de manière générale.

Exemple de commande correspondant à une configuration avec 4 ALUs, 1 multiplieur, 4 ALUs flottantes, 1 multiplieur flottant :

```
$GEM5 -d "out_dir" "se.py" --cmd="program.riscv" --cpu-type=O3 --caches \
--ialu=4 --imult=1 --fpalu=4 --fpmult=1 --mempport=2
```

Exemple de commande correspondant à une configuration avec 8 ALUs, 4 multiplieurs, 1 ALU flottante, 1 multiplieur flottant :

```
$GEM5 -d "out_dir" "se.py" --cmd="program.riscv" --cpu-type=O3 --caches \
--ialu=8 --imult=4 --fpalu=1 --fpmult=1 --mempport=2
```

Du point de vue applicatif, l'étude sera menée sur l'algorithme PageRank, développé par Google comme métrique mesurant l'importance d'une page web [1]. Il s'avère que cette métrique peut également être utilisée dans d'autres types de réseaux (e.g. sociaux) afin de caractériser l'importance relative d'un sommet.

A l'étape initiale, chaque sommet se voit attribuer un rang égal. Chaque sommet recalcule son rang en fonction du rang de ses voisins et de ses degrés entrants/sortants. On considère que l'algorithme a convergé lorsque la variation de rang, d'une itération à l'autre, est inférieur à un seuil (voir [1] pour le détail).

Q1 : Analysez le code de l'algorithme PageRank en faisant varier la taille du graph d'entrée (min, med, max) et en faisant varier le nombre d'unités fonctionnelles M de 1 à 8 (1, 2, 4, 8). (Voir annexe « Applications »)

Q2 : Tracez 2 figures indiquant le nombre de cycles et le CPI associés en fonction de la taille du problème traité ($N=\{\text{min, med, max}\}$) et des unités fonctionnelles. Quelle est votre analyse ?

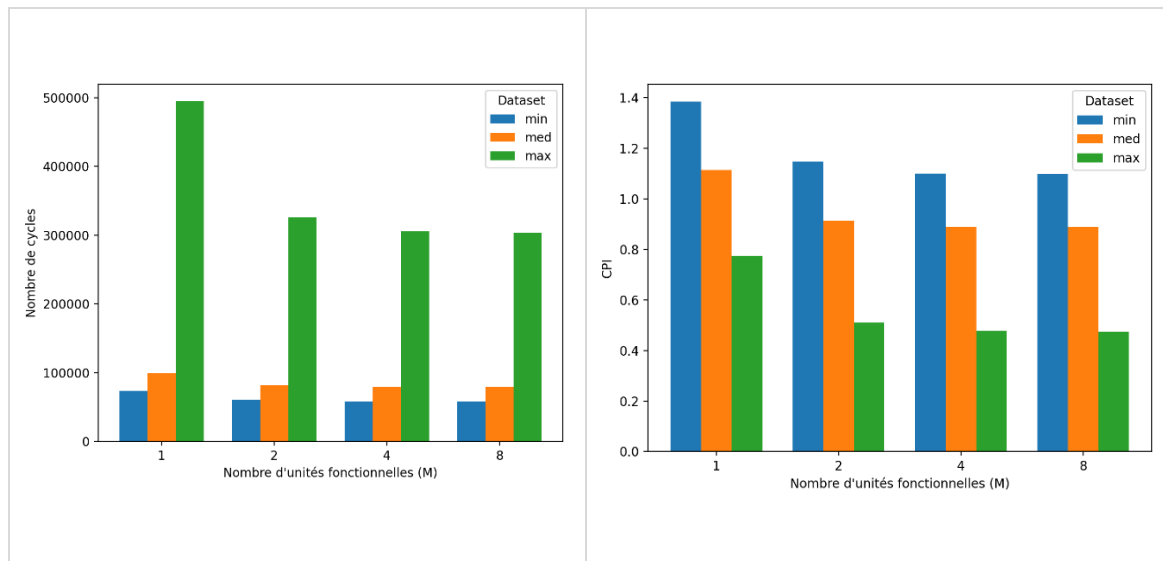


Figure 23 a) Nombre de cycles vs N vs Unités fonctionnelles

b) CPI vs N vs Unités fonctionnelles

A1 et A2 :

On fait varier la taille du graphe d'entrée du programme pagerank en utilisant les programmes suivants : pagerank_<version>.riscv, ou <version> prend les valeurs suivantes :

- **min (plus petit)**
- **med**
- **max (plus grand)**

Pour le nombre d'unités fonctionnelles, on fait varier le nombre d'ALUs et de multiplieurs entiers et flottants en même temps.

**\$GEM5 -d "\$OUT" "\$CFG" **

**--cmd="\$BIN" **

**--cpu-type=O3 --caches **

--ialu=\$M --imult=\$M --fpalu=\$M --fpmult=\$M --memport=2

où M est le nombre d'unités fonctionnelles.

Nombre de cycles

PR/M	1	2	4	8
Min	73k	61k	58k	58k
Med	99k	81k	79k	79k
Max	495k	326k	305k	303k

Interprétation : Quand on augmente le nombre d'UF, les performances augmentent jusqu'à atteindre un seuil maximal pour nombre d'UF = 4. Le processeur superscalaire n'arrive pas à exploiter efficacement plus de 4 UF parce qu'il manque de parallélisme d'instructions. Ce comportement s'explique par le fait que le processeur DerivO3CPU n'est plus limité par les ressources de calcul, mais par le parallélisme d'instructions exploitable et le débit mémoire. En particulier, le nombre de ports mémoire (memport = 2) limite le nombre d'accès mémoire pouvant être émis par cycle, ce qui constitue un goulot d'étranglement majeur pour l'algorithme PageRank, fortement dépendant des accès mémoire. Même si l'on augmentait la taille de ces buffers, le gain en performances ne serait pas suffisant pour justifier l'augmentation de la surface du processeur.

Un concepteur d'architecture de processeur doit toujours trouver le seuil optimal qui peut donner la meilleure efficacité surfacique (performance en MOPS / surface en mm²).

N.B : les résultats peuvent varier par rapport au programme que vous exécutez.

Exercice 4 : Exécution dans l'ordre (in-order) VS exécution dans le désordre (out-of-order)

Dans un processeur superscalaire à ré-ordonnancement dynamique des instructions, l'exécution des instructions peut s'effectuer dans le désordre dans l'objectif d'éviter les contraintes artificielles de la séquentialité.

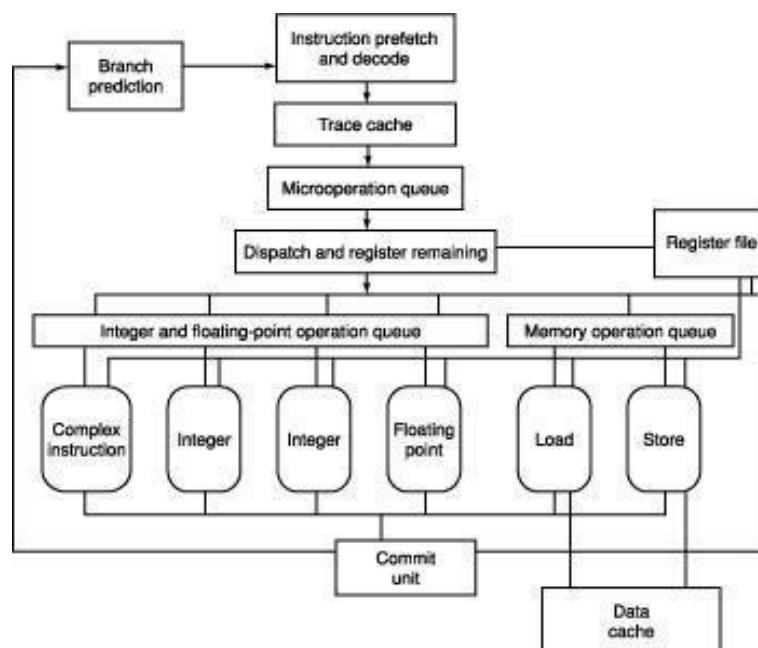


Figure 24 Pipeline d'exécution dans le désordre (out-of-order)

L'exécution d'un programme tel que généré sans optimisations par un compilateur représente l'ordre sémantique normal du programme. Pourtant l'ordre d'exécution des instructions peut être modifié tout en préservant la sémantique. Cette exécution est appelée **out-of-order exécution**.

Le simulateur **sim-outorder** permet d'exécuter les instructions dans l'ordre ou le désordre (*out-of-order exécution*). Les deux commandes suivantes permettent les 2 possibilités :

- Exécution dans l'ordre : `sim-outorder -issue:inorder true program.ss`
- Exécution dans le désordre : `sim-outorder -issue:inorder false program.ss`

Par défaut le simulateur, ayant un modèle de processeur superscalaire avec ré-ordonnancement dynamique, exécute suivant un mode *out-of-order exécution*.

Q1 : Évaluez l'impact de l'exécution (CPI, nombre de cycles) dans l'ordre par rapport à l'exécution dans le désordre pour l'algorithme PageRank

A1 :

On utilise les paramètres par défaut du simulateur, en faisant varier paramètre **-issue:inorder** uniquement, sur l'application `pagerank_med`.

In-order VS out-of-order

	In-order	Out-of-order
CPI	3.74	0.89
Cycles	334455	79878

Le tableau montre bien une différence de performances **importante** entre un processeur **superscalaire out-of-order** et un processeur **superscalaire in-order**.

Cela s'explique par le fait que le processeur **superscalaire in-order** n'exploite pas dynamiquement le parallélisme qui existe entre les instructions du code (ILP : Instruction Level Parallelism). Dans le cas d'un **superscalaire in-order**, ce parallélisme n'est analysé que lors de la compilation, contrairement au **superscalaire out-of-order**, qui ainsi exploite mieux les unités fonctionnelles de l'architecture.

Plus précisément, lorsqu'un aléa a lieu dans le pipeline (aléa structurel, aléa de données), le processeur **superscalaire in-order** est bloqué ('stall' en anglais) jusqu'à ce que la cause de l'aléa ne soit résolue, tandis que le processeur **superscalaire out-of-order** peut exécuter d'autres instructions (indépendantes de celles qui causent l'aléa). Cela augmente la performance globale du processeur.

N.B : les résultats peuvent varier par rapport au programme que vous exécutez.

Solution TD/TP4 : Microprocesseur superscalaire / mémoires caches – Analyse de configurations d’architectures de microprocesseurs

Exercice 1 : Prédiction de branchement

La prédiction de branchement permet aux processeurs d’exécuter un flot d’instructions sans pénalité de blocage dû à l’attente de l’évaluation de la condition de branchement. Tous les microprocesseurs haute-performance incluent une unité de prédiction de branchement. La Figure 6 montre la surface prise par l’unité de prédiction de branchement dans un processeur multi-cœur Intel.

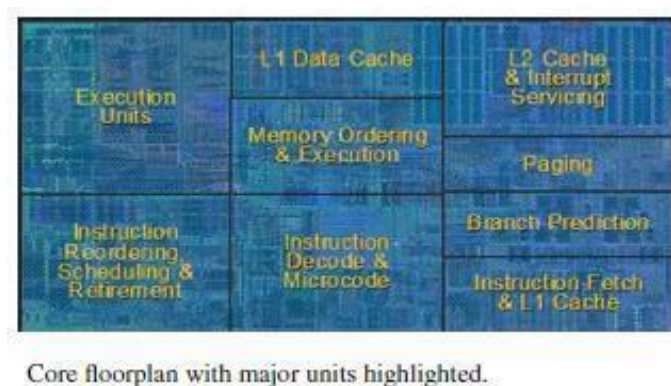


Figure 25 Floorplan d’un cœur de processeur Intel

Nous souhaitons dans cet exercice évaluer l’impact de différents mécanismes de prédiction de branchement sur le CPI et le nombre de cycles pour 2 applications :

1. La fonction de hachage SHA-1 ([7] [9]). Voir l’annexe « Applications ».
2. L’algorithme de calcul de la centralité d’intermédiarité (Betweenness Centrality Score / BCS) dans la suite SSCA2 [TP2.1], se référer à l’annexe « Applications ».

Les mécanismes de prédiction que nous souhaitons comparer sont les suivants :

- **nottaken** : ce mécanisme de prédiction de branchement statique considère toujours que le branchement n’est pas pris
- **taken** : ce mécanisme de prédiction de branchement statique considère toujours que le branchement est pris
- **perfect** : ce prédicteur « virtuel » représente la prédiction parfaite
- **bimod** : le prédicteur de branchement bimodal utilise un BTB (Branch Target Buffer) avec compteurs à 2 bits
- **2lev** : prédicteur de branchement adaptatif à 2-niveaux

Vous pouvez spécifier en utilisant le simulateur SimpleScalar le mécanisme de prédiction de branchement de votre choix en utilisant la commande suivante :

sim-outorder -bpred <type> <prog>.ss [options]

La documentation de SimpleScalar précise les détails pour les différents mécanismes de prédiction de branchement. La Figure 7 et la Figure 8 décrivent 2 types de prédicteurs de branchement, et la Figure 9 décrit les différents paramètres en entrée pour le prédicteur de branchement de SimpleScalar.

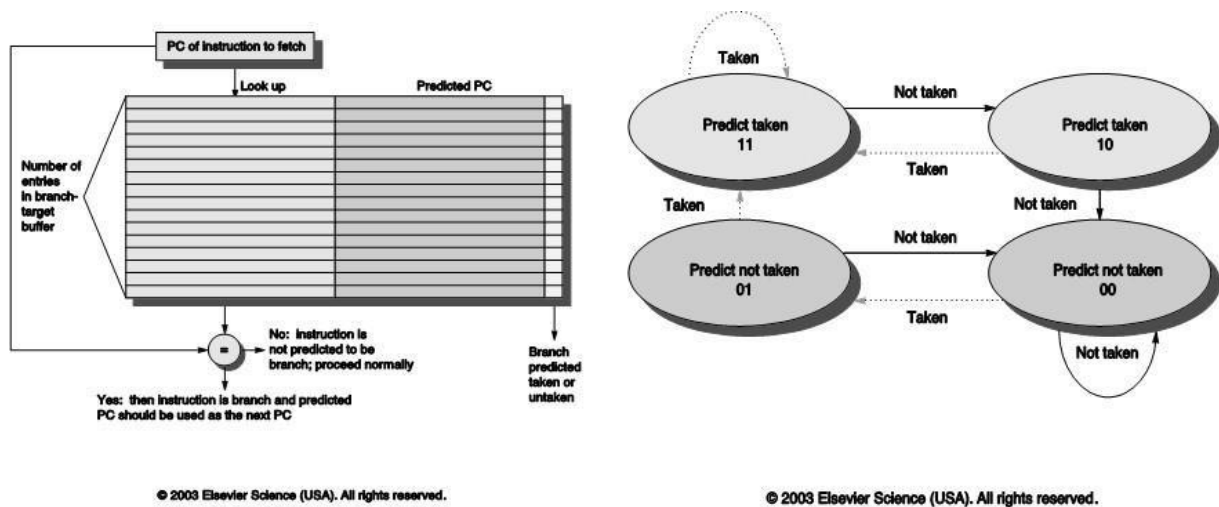


Figure 27 Le prédicteur de branchement bimodal

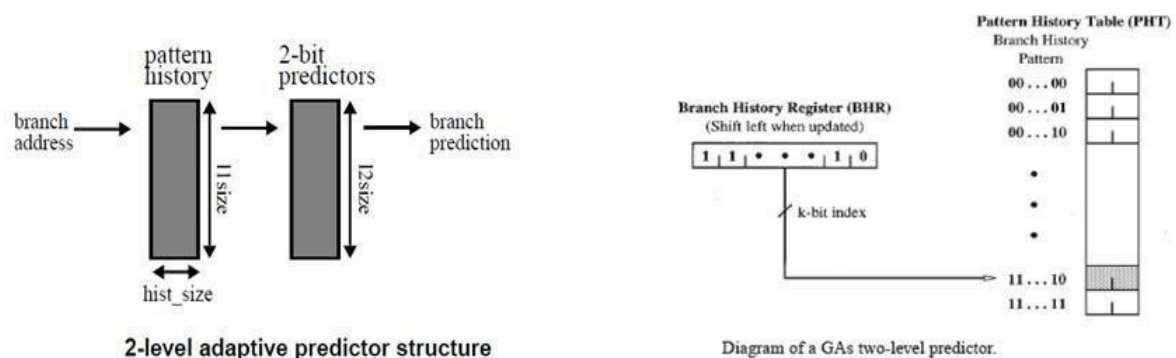


Figure 26 Le prédicteur de branchement adaptatif à 2-niveaux

Tableau 14 Paramètres du prédicteur de branchement

predictor	l1_size	hist_size	l2_size	xor
GAg	1	W	2^W	0
GAp	1	W	$>2^W$	0
PAg	N	W	2^W	0
PAP	N	W	2^{N+W}	0
gshare	1	W	2^W	1

Branch predictor parameters

On utilise les paramètres par défaut du simulateur, en faisant varier le paramètre **-bpred:<type>** uniquement.

Prédicteur de branchement pour SHA-1

	Not-taken	Taken	Perfect	Bimod	2lev
CPI	0.6815	0.6815	0.3521	0.3681	0.3683
Cycles	10068798	10068408	5202419	5438930	5441322

Le tableau montre que les prédicteurs de branchement **dynamiques (bimod et 2lev)** sont beaucoup plus performants que les prédicteurs de branchement **statiques (not-taken et taken)**, et sont très proches du seuil maximal d'un prédicteur de branchement **idéal** (perfect), qui bien évidemment n'existe pas en pratique.

On constate aussi que le prédicteur de branchement **bimod** est vraisemblablement plus efficace que **2lev**, puisque le surcout surfacique du prédicteur 2lev est plus important que celui du bimod (plus de mémoire pour stocker les tableaux), pour une augmentation de performances très relative.

N.B : les résultats peuvent varier par rapport au programme que vous exécutez.

Exercice 2 : Impact de la fenêtre d'instructions RUU

Le modèle d'exécution superscalaire de SimpleScalar utilise un mécanisme de ré-ordonnancement basé sur une partie centrale : la RUU (Register Update Unit).

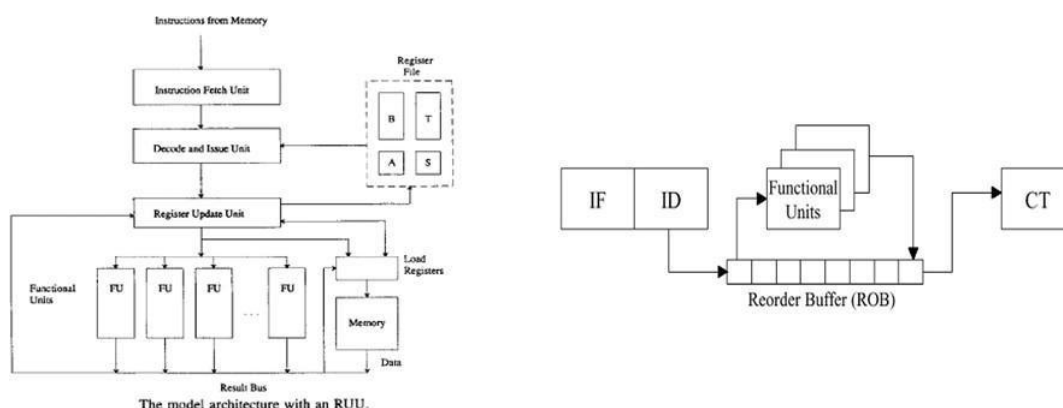


Figure 28 Register Update Unit (RUU)

L'augmentation de la taille de la RUU permet à priori de considérer davantage d'instructions potentielles pour l'exécution. Nous souhaitons analyser l'impact de cette taille sur le CPI.

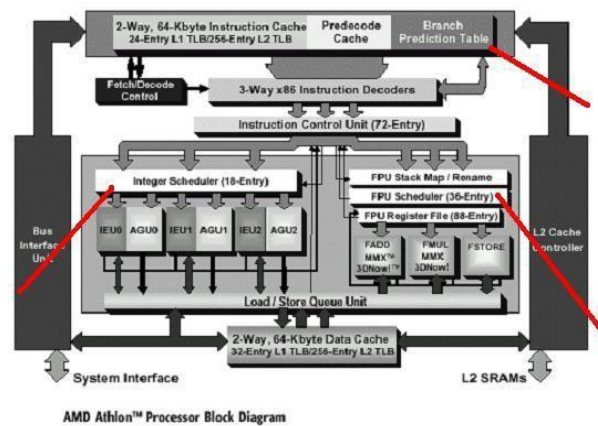


Figure 29 AMD Athlon Processor Block Diagram

Q1: Faites varier la taille de la RUU de 16 à 128 et évaluez l'impact sur le CPI et le nombre de cycles sur une application de votre choix.

A1 :

On utilise les paramètres par défaut du simulateur, en faisant varier le paramètre `-ruu:size <int>` uniquement.

Register Update Unit (RUU) pour SHA-1

	RUU=16	RUU=32	RUU=64	RUU=128
CPI	0.3681	0.3470	0.3426	0.3423
Cycles	5438930	5126270	5061994	5057066

Le tableau montre que de meilleures performances en faisant augmenter la taille de la RUU, mais que le coût de cette amélioration ne justifie pas non plus le gain obtenu au-delà de RUU=32.

Pour cette application, il n'y a aucun intérêt à augmenter davantage la taille de la RUU (au-delà de 32), puisque le processeur superscalaire n'arrive pas à trouver un nombre suffisant d'instructions indépendantes à exécuter dans le désordre.

N.B : les résultats peuvent varier par rapport au programme que vous exécutez.

Exercice 3 : Mémoires caches - Evaluation des performances de différentes configurations de mémoires caches (instructions et données) pour 4 algorithmes de multiplication de matrices.

De nombreuses applications informatiques dans différents domaines font appel à des opérations sur des matrices, et en particulier la multiplication de matrices. Cette opération classique peut être répétée de nombreuses fois au cours de l'exécution de ces applications et il est donc primordial que ses performances soient optimisées. La multiplication de matrices sur des matrices de dimensions importantes est une opération dont la performance en temps d'exécution dépend beaucoup des performances de la hiérarchie mémoire, et notamment l'organisation des caches du microprocesseur sur lequel elle s'exécute. Le taux de défauts de caches (**miss rate**), qui est le rapport entre le nombre d'accès à un cache qui se traduisent par un défaut de cache (**cache miss**) sur le nombre total d'accès au cache, est le paramètre principal pour évaluer cette performance. Nous considérons ici des microprocesseurs ayant des caches d'instructions et de données séparés. De ce fait, le miss rate du cache d'instructions et du cache de données méritent une analyse.

Un simulateur de cache, comme **sim-cache** (**Annexe 1**), permet d'effectuer une évaluation du miss rate pour une configuration de caches particulière et pour un programme en entrée. Lorsque l'on souhaite évaluer plusieurs configurations de caches il est alors nécessaire d'effectuer autant de simulations. Ce processus se répète pour le nombre de programmes que l'on considère en entrée.

Travail demandé

Nous considérons les 2 organisations de caches décrites dans le Tableau 7. On supposera que l'algorithme de remplacement est LRU pour toutes les configurations.

Tableau 15 Configurations de caches pour 2 processeurs

Configuration	Instruction cache	Data cache	L2 cache	Block size (bytes)
C1	4KB direct-mapped	4KB direct-mapped	32KB direct-mapped	32
C2	4KB direct-mapped	4KB 2-way set-asso	32KB 4-way set-asso	32

Q1 : Pour chacune des 2 configurations de mémoires cache, complétez le Tableau 8. Pour cela, vous devez déterminer les paramètres d'entrée du simulateur de cache « *sim_cache* » de SimpleScalar.

Tableau 16 Paramètres de *sim-outorder* pour chaque configuration

Configuration	IL1	DL1	UL2
C1	il1:128:32:1:l	dl1:128:32:1:l	ul2:1024:32:1:l
C2	il1:128:32:1:l	dl1:64:32:2:l	ul2:256:32:4:l

*Q2 : Complétez les Tableau 9, Tableau 10 et Tableau 11. Pour cela vous devez simuler à l'aide de **sim-cache** les différentes configurations et collecter les informations suivantes :*

- Le taux de défauts dans le cache d'instructions il1 : **il1.miss_rate**
- Le taux de défauts dans le cache de données dl1 : **dl1.miss_rate**
- Le taux de défauts dans le cache unifié (L2) ul2 : **ul2.miss_rate**

A2 :

Dans cet exercice, on modélise une hiérarchie mémoire de 2 niveaux, constituée d'un cache L1 d'instructions et de données séparés, et d'un cache L2 partagé par les 2 caches L1, et qui est relié à une mémoire externe de taille supposée infinie.

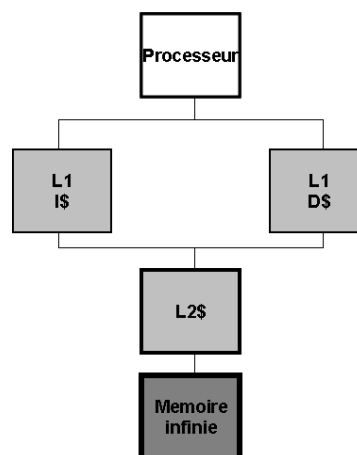


Figure 30 Hiérarchie mémoire vue du processeur

Le code pour les 4 algorithmes de matrices se trouve dans :

/usr/ensta/pack/simplescalar-2.0/simplescalar-4.0/mase/matrix/<algo>.ss

Où <algo> = **normale, pointeur, tempo, unrol**

Vous pouvez consulter également le fichier source <algo>.c correspondant pour comparer les 4 algorithmes de multiplication de matrices.

La commande pour exécuter un programme avec la configuration de cache C1 est :

sim-cache -cache:il1 il1:128:32:1:l -cache:dl1 dl1:128:32:1:l -cache:il2 dl2 -cache:dl2 ul2:1024:32:1:l /usr/ensta/pack/simplescalar-2.0/simplescalar-4.0/mase/matrix/<algo>.ss

La commande pour exécuter un programme avec la configuration de cache C2 est :

sim-cache -cache:il1 il1:128:32:1:l -cache:dl1 dl1:64:32:2:l -cache:il2 dl2 -cache:dl2 ul2:256:32:4:l /usr/ensta/pack/simplescalar-2.0/simplescalar-4.0/mase/matrix/<algo>.ss

Les résultats de performances des caches pour les 4 algorithmes sont :

Tableau 17 Instruction Cache (il1) Miss Rate

Programmes	<i>Configurations de caches</i>	
	C1	C2
P1 (normale)	0.0%	0.0%
P2 (pointeur)	0.01%	0.01%
P3 (tempo)	0.0%	0.0%
P4 (unrol)	0.0%	0.0%

Tableau 18 Data Cache (dl1) Miss Rate

Programmes	<i>Configurations de caches</i>	
	C1	C2
P1 (normale)	3.67%	3.38%
P2 (pointeur)	4.13%	4.00%
P3 (tempo)	4.41%	4.27%
P4 (unrol)	5.12%	4.96%

Tableau 19 Unified Cache (ul2) Miss Rate

Programmes	<i>Configurations de caches</i>	
	C1	C2
P1 (normale)	41.88%	40.42%
P2 (pointeur)	44.14%	40.17%
P3 (tempo)	44.13%	40.17%
P4 (unrol)	44.18%	40.17%

Q3 : Les 4 algorithmes de multiplication de matrices présentent-ils une bonne localité de références pour le code ? Pourquoi ?

A3 :

Interprétation :

Au début de la simulation, le code et les données des programmes sont dans la mémoire externe. Les premiers accès à la mémoire (cold miss) engendrent des caches miss au niveau L2 et L1.

L1 I\$: le miss rate du cache d'instructions L1 est négligeable, ce qui signifie que la taille du code pour les 4 algorithmes peut tenir dans un cache de 4KB direct-mapped sans aucun impact sur les performances.

L1 D\$: le miss rate du cache de données L1 pour la configuration C1 (direct-mapped) est plus élevé que celui de C2 (2-way set associative). Ceci s'explique par le fait qu'en augmentant l'associativité du cache, le miss rate du cache de données diminue => moins d'accès au cache L2 => moins de blocages du processeur dus à la latence de la mémoire. L'algorithme de multiplication 'normale' présente la meilleure localité de référence, puisqu'il a le moins de miss rate.

L2\$: de la même manière, l'augmentation de l'associativité du cache en C2 diminue le miss rate => moins d'accès à la mémoire externe qui est pénalisante (de l'ordre de 100 cycles).

Le miss rate du cache L2 est élevé (> 40%). Cela montre que le cache L2 n'est pas bien dimensionné et qu'un cache de plus grande taille paraît judicieux pour ces 4 algorithmes de multiplication.

Solution TD/TP6 : Microprocesseurs et Energie



Exercice 1 : Lois Physiques Energie – Puissance

Q1 : 2 milliards de téléphones portables dans le monde avec des utilisations en mode idle

A1. Prenons pour l'exemple le smartphone cité en référence, dont la consommation « idle » est de l'ordre de 70mW

$$2 \cdot 10^9 \times 70mW \approx 140MW \text{ (17+ Cray Titan, 140+ Bugatti Veyron)}$$

$$140MW \times 365 \times 24 = 1.2TWh \text{ par an (Si tous les téléphones étaient en idle pendant l'année)}$$

Q2 : 2 milliards de téléphones portables dans le monde avec des utilisations en mode actif

A2. Prenons le même exemple : la consommation « active » est en moyenne de l'ordre de 2W.

$$2 \cdot 10^9 \times 2W = 4GW \text{ (30+ Genève / 8000 Eoliennes / 2+ tranches de réacteur IV)}$$

$$4GW \times 365 \times 24 = 35TWh \text{ par an (Si tous les téléphones étaient actifs pendant l'année).}$$

Q3 : consommation équivalente en éoliennes

A3. En se basant sur un modèle « gros grain » d'utilisation 50% idle / 50% actif, on obtiendrait en moyenne :

$$\frac{35}{2} + \frac{4}{2}$$

$$19.5$$

$$2 \quad 2$$

$= 19.5TWh$



1 éolienne (40m / 43km/h) = 500kW Pnominal

En France : 15.9 TWh (2012), il faudrait environ 1.22 fois la production éolienne équivalente en France pour produire l'énergie nécessaire à l'ensemble des smartphones.

Notes : 15.9TWh, soit 1.81GW Pnominal (au lieu des 8 GW), rendement « faible »

Q4 : consommation équivalente en champ solaires



A4. En France : 4.9 TWh (2013) sont produit par champ solaire, il faudrait donc environ 4 fois la production solaire française pour produire l'énergie nécessaire à l'ensemble des smartphones.

Notes : 4.9TWh produit en 2013 équivaut à une installation « parfaite » de 560 MWc (au lieu des 4.7 GWc installés).

Notes : Le plus grand champ solaire du monde : 1 GWc théorique, en Chine pour 2020 (en 2014, 540MWc). Il est à noter que les champs solaires ne sont qu'une manière d'exploiter l'énergie solaire (solaire thermique/thermodynamique)

Remarques générales & ordres de grandeurs :

Puissance :

- Cortex A9 @1GHz	=	1W
- Intel Core i5-6440HQ @ 2.60GHz	=	45W
- Intel Xeon X7460 @ 2.66Ghz	=	130W
- Radiateur sèche serviette	=	500W-2kW
- Saab 9-3 Aero (MY06)	=	191 kW (260cv)
- Koenigsegg One :1	=	1MW (1360cv)
- Cray Titan	=	8.2 MW
- Ville de Genève	=	150 MW
- LHC (CERN)	=	220-300 MW
- Tranche EPR Gen. IV	=	1-1.5 GW
- Data centers	=	30 GW
- Puissance peak EDF	=	136 GW

- Laser BELLA (Thales) = 1 PW

On se méfierait tout de même, et d'une manière générale, des seuls chiffres de puissance (notamment pour l'éolien/le solaire). On mettrait plutôt en perspective ces valeurs en termes de Wattheures produit par an ou bien de W/m^2 pour le solaire qui sont des métriques intéressantes car elles permettent d'apprécier le rendement de production (ou surfacique). Par exemple pour l'éolien français, la puissance nominale des installations comparée à la puissance effectivement délivrée nous donne un rendement de l'ordre de 20%, relativement faible ($\mu = P_{prod}/P_{nom}$).

De même, le modèle « gros grain » 50/50 utilisée pour le calcul de l'énergie utilisée par les smartphones sur un an part du principe que le téléphone n'est actif que 50% du temps... or ce modèle n'est pas réaliste. La complexité logicielle de ces systèmes ne facilite pas l'identification d'un modèle de consommation précis (recherche d'un réseau Wi-Fi /GSM en permanence, Mise à jour de l'OS/des applications en tâche de fond, etc...). De plus, la variabilité des usages d'un smartphone rendrait un tel modèle non-nécessairement applicable à tous les utilisateurs finaux. Il faudrait donc arriver à composer un modèle tenant compte des différences architecturales, logicielles et d'utilisation des smartphones.

Tous ces calculs nous amènent à la conclusion que le moindre pourcent d'économie d'énergie qu'il est possible de gagner sur des systèmes tels que les smartphones (qui embarquent de plus en plus les mêmes processeurs/capteurs) sont bons à prendre à l'échelle du milliard de « devices » → $1\% * 19.5 \text{ TWh} = 195 \text{ GWh}$ (22MW), soit l'équivalent de la consommation électrique sur un an d'une petite ville de 30 000 habitants...

Consommation moyenne / habitants : (FR-2011) : 7MWh (800W) / (INT-2012) : 3MWh (350W)

Consommation mondiale d'électricité en 2012 : environ 20 PWh

Production mondiale annuelle d'électricité : quelques dizaines de PWh... (Production totale d'énergie : 184000 PWh)

Data centers : 30 GW / 263 TWh ...

Exercice 2 : Mode de gestion de l'énergie des microprocesseurs (ARM) et FSM

Q1. Citer 3 méthodes matérielles pour réduire la consommation énergétique dans les multi-cœurs

1/Clock gating: Couper l'horloge dans certaines parties des cœurs.

2/Power gating: Extinction de certaines parties des cœurs.

3/Dynamic Voltage Frequency Scaling (DVFS): Ajustement de la fréquence/tension dynamiquement.

Q2. Pour chaque méthode, quelles sont les implications énergétiques ? (Statique, dynamique, surface, timing) Faire un tableau de comparaison

	Energie Statique	Energie Dynamique	Surface	Timing
Clock Gating	Pas de changements	Pas d'énergie dynamique consommée	Augmentation légère de la surface	Overhead : temps d'établissement de la clock
Power Gating	Pas d'énergie statique consommée	Pas d'énergie dynamique consommée	20 % à 30 % de plus en cas de registres + surface des interconnexions	Temps de sauvegarde avant extinction et temps de backup au réveil sur les données volatiles
DVFS	Réduction adaptable	Réduction adaptable	Module de gestion voltage/fréquence à ajouter (faible surface)	Temps d'établissement de la clock en cas de changement.

Q3. Dans la documentation cortex A9, quelles sont les techniques mises en œuvre et comment le sont-elles ?

Voir (2.4) dans la Technical Reference Manual du Cortex A9 :

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388i/index.html>

Par exemple 2.4.3: Power Gating « The Cortex-A9 uniprocessor contains optional placeholders between the Cortex-A9 logic and RAM arrays, or between the Cortex-A9 logic and the NEON SIMD logic, when NEON is present, so that these parts can be implemented in different voltage domains. »

Exercice 3 : Microélectronique

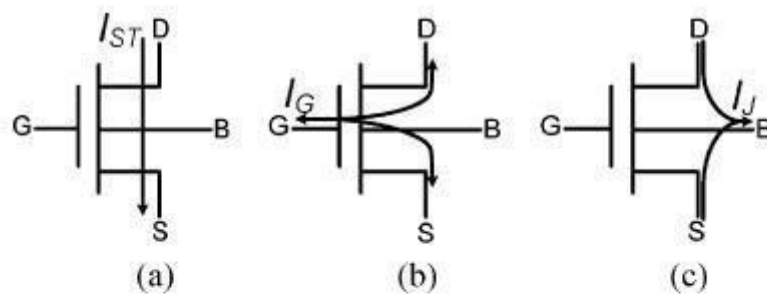


Fig. 2. NMOS transistor current contributions in subthreshold. (a) Subthreshold current. (b) Gate current. (c) Junction current.

TABLE I
NMOS/PMOS TRANSISTOR PARAMETERS (65-NM, STD- V_{TH}) [17]

	n	I_0 [A]	V_{TH0} [V]	λ_{DS}	λ_{RS}
NMOS	1.39	6.65E-5	0.598	9.0E-2	9.9E-2
PMOS	1.27	5.95E-6	0.532	8.0E-2	1.1E-1

Q1 : Technologie actuelle / combien de transistors / consommation / surface dans un processeur récent ?

A1. Quelques milliards de transistors (2014):

Intel Core-M family : 14nm (1,300,000,000 t) / 30x16.5 mm² / 4.5 Watts (TDP) / 1.2-2.9 GHz.

Intel i7 family : 22nm (1,400,000,000 t) / 52.5x45 mm² / 130 Watts (TDP) / 3.6-4.0 GHz.

Q2 : Qu'est-ce que la conduction en régime de faible inversion et pourquoi est-ce utilisé ? Principaux "tuning knobs" accessibles ?

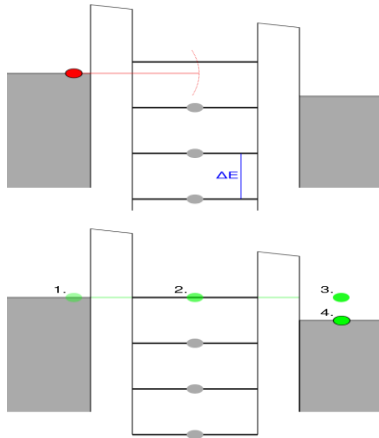
« Subthreshold conduction » on « fait conduire » un transistor sous V_{th} . Le canal n'est pas formé au sens strict du terme comme en forte inversion, mais un courant de diffusion, fonction de V_{ds} traverse le transistor. Ce régime de faible inversion est intéressant car il expose une transconductance très élevée (mais importante influence du process de fabrication et des défauts).

Tuning Knobs :

Le facteur de forme du transistor : W/L . Si l'on augmente W , V_{th} peut augmenter (RNCE) ce qui rend le courant peut sensible aux variations de W pour les transistors « fins » (on augmentera plutôt L : RSCE, détérioration de V_{th}).

De fait, V_{th} est aussi un paramètre important. Mais toutes ces caractéristiques sont déterminées par le designer du circuit... (Autres facteurs : polarisation du substrat (V_{bb}),

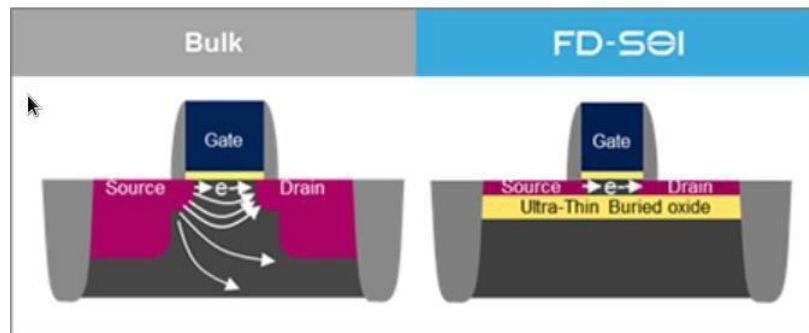
Q3 (opt.) : Quels sont les effets les plus pénalisants pour le transistor MOS ?



- μ (pondération par champ vertical/horizontal de la mobilité des porteurs)
- d (pondération du courant par non-uniformité du substrat)
- dC/C Augmentation de C_{ox} par déplétion de la grille
- v_L saturation en vitesse des porteurs
- DIBL diminution de v_{th} par effet de drain
- Perçage passage de courant dans le substrat hors contrôle de la grille
- R résistance série du transistor
- I_{sub} courant de substrat
- NPN Effet du transistor NPN parasite

Remarques

Lien Design/Puissance : $P = \frac{1}{2} V^2 C f$ / Static : $I_{leak} \propto \frac{W}{L}$



FDSOI (rajout d'un film pour isoler le canal du bulk: DIBL réduit, effet parasite du substrat réduit + meilleur contrôle du transistor) vs. ThinFET (faire ressortir la grille en surface pour une meilleure isolation des parasites du substrat/drain)

SET – Single Electron Transistors

Transistor ambipolaire, contrôle du courant à l'électron près : électronique TRES low-power. Dimensions nanométriques (« Coulomb Island » de quelques nm), donc très fort potentiel d'intégration. Courant de commande très faible, sensible aux perturbations électromagnétiques. Température de fonctionnement très basse...

Exercice 4 : Evaluation de performances – MPACT

On recherche à optimiser le processeur choisi, afin de réduire la consommation énergétique globale du système. Dans ce cas d'étude, nous nous intéressons à l'intérêt d'avoir une unité flottante dans le processeur. Le cas d'étude s'appuie sur les applications ftt et susan (du benchmark miBench) sur un processeur de type ARM cortex A9. Pour avoir les informations de performance, le simulateur Gem5 est utilisé. Celui-ci est « connecté » à l'outil McPat permettant d'avoir les informations de surface et consommation énergétique du processeur.

Commandes utiles :

```
export HOME_TOOLS=/home/c/nom-mdc/ES201/tools/TP6/tools/
```

```
cp -r =/home/c/nom-mdc/ES201/tools/TP6/bin/ /your/path/ cd /your/path/bin/<app>
```

```
$HOME_TOOLS/run_all.sh -b <app> (-f pour floating point)
```

Q1 Simuler PageRank et SSCA2 avec et sans l'extension Floating point. Lancer le simulateur et l'émulateur énergétique. Quel est l'accélération à utiliser l'unité flottante ? Y a-t-il un gain énergétique ? Regarder dans m5out/.xml et m5out/*.xml_simple*

SSCA2	PageRank
Basic 0.000267324201618375 J in 0.00209697125 sec core_area 0.759698 mm ² core_leakage 0.012725105 W core_runtime_dynamic 0.039784 W core_energy 0.000110110083548231 J	Basic 0.000199285153215875 J in 0.00157287875 sec core_area 0.759698 mm ² core_leakage 0.012725105 W core_runtime_dynamic 0.0395964 W core_energy 8.22953833825187e-05 J
fpu 0.00024410315597925 J in 0.0018347675 sec core_area 1.43199 mm ² core_leakage 0.01859642 W core_runtime_dynamic 0.0393282 W core_energy 0.00010627821022585 J	fpu 0.000140310721439 J in 0.00107046625 sec core_area 1.43199 mm ² core_leakage 0.01859642 W core_runtime_dynamic 0.0378871 W core_energy 6.04637018412e-05 J

(résultat d'énergie et de surface comprenant le cœur et le cache L1)

	SSCA2	Pagerank
Speedup	1.1430	1.4693
Energy gain	1.0951	1.4250

Q2 Quelle est votre analyse pour chaque application ?

Dans le cas de PageRank, l'ajout d'une FPU fait sens dans la mesure où les performances en termes d'exécutions sont largement améliorées (47% en temps d'exécution et 43% en efficacité énergétique). Dans le cas de Susan, les performances sont améliorées bien plus légèrement.

Etant donné l'important gain en termes de surface utilisé lors de l'ajout d'une FPU, ainsi que le leakage occasionné par l'ajout d'une FPU, la décision de l'ajout ou non d'une FPU au système doit se faire en tenant compte d'autres critères : variétés d'applications utilisées, caractérisation du profil d'utilisation (si le système reste très souvent au repos, le leakage important avec une FPU risque de se révéler pénalisant par exemple) contraintes temporelles et/ou spatiales.

Q3 Si les 2 applications doivent tourner sur la même plateforme quelles méthodes peut-on employer pour réduire l'énergie ? Proposer des idées dans le cadre d'un mono-cœur et d'un multi-cœur. Les solutions de l'exercice 1 sont-elles utilisables ?

On peut utiliser le power gating autour de la FPU dans un mono cœur afin de couper l'extension quand on ne s'en sert pas (le clock gating ne sert pas, car les registres flottant continuent à dissiper de l'énergie statique, l'impact sera moindre). Le DVFS ne résout pas la question de l'utilisation ou non des extensions.

Dans un multi-cœur, il existe un type de plateforme dite « asymétrique » où les extensions ne sont mises que sur certains coeurs. (cf. Article d'intel: <http://www.neotextus.net/wp-content/uploads/2013/03/hpca10.pdf>, thèse d'Alexandre AMINOT).

Hors-Programme – Pour aller plus loin...

Compilateur – analyse de dépendances

Déterminer les dépendances dans les boucles de code en se basant sur l'hypothèse que les indices de tableaux ont des formes affines ($a * i + b$) où a et b sont des constantes et i est l'index de boucle. Déterminer s'il existe une dépendance entre 2 références au même tableau dans une boucle est équivalent à déterminer si 2 fonctions affines peuvent avoir la même valeur pour des indices différents dans l'intervalle de valeur de l'indice de boucle.

Un test simple et suffisant pour détecter l'absence de dépendances est le PGCD. Le test est basé sur l'observation que si une dépendance à travers une boucle (loop-carried dependence) existe alors le $\text{PGCD}(c, a)$ doit diviser $(d - b)$. Le test du PGCD est suffisant pour garantir qu'aucune dépendance n'existe.

Déterminer pour la boucle suivante s'il existe une dépendance entre les variables du tableau.

```
for(i = 1 ; i <= 1000 ; i = i + 1) {
    x[2*i + 3] = x[2*i] + 5.0;
}
```

$a*i + b = 2*i + 3 \Rightarrow a = 2, b = 3$

$c*i + d = 2*i \Rightarrow c = 2, d = 0$

$\text{pgcd}(c, a) = \text{pgcd}(2, 2) = 2$ ne divise pas $(d - b) = -3$. **Donc il n'existe pas de dépendance entre $2*i + 3$ et $2*i$.**

Impact du ré-ordonnement de code - Loop unrolling

Une technique utilisée par les compilateurs pour améliorer les performances de programmes s'exécutant sur des processeurs avec une structure pipeline consiste à déplier (**unroll**) des boucles de code (**loop unrolling**). Le principe général est simple et consiste donc à dupliquer le corps de la boucle.

Considérons l'exemple suivant :

```
for(i = 1000 ; i > 0 ; i = i - 1)
    x[i] = x[i] + s;
```

Dans cet exemple, un dépliage de niveau 5 donnerait :

```
for(i = 1000 ; i > 0 ; i = i - 5)
{
    x[i] = x[i] + s;
    x[i-1] = x[i-1] + s;
    x[i-2] = x[i-2] + s;
    x[i-3] = x[i-3] + s;
    x[i-4] = x[i-4] + s;
}
```

Q1: *Codez en assembleur la boucle initiale et estimez le nombre de cycles nécessaires à son exécution pour le pipeline du processeur.*

A1 :

Supposons que \$s3 contienne l'adresse de x[1000]

```
addi    $s2, $zero, 1000    // initialisation de l'indice de boucle
start :
lw      $s1, 0($s3)         // lecture donnée en mémoire
addi    $s1, $s1, s         // ajout de la constante s
sw      $s1, 0($s3)         // écriture et mise à jour en mémoire
subi    $s3, $s3, 4         // passage à l'élément suivant dans le tableau
subi    $s2, $s2, 1         // mise à jour de l'indice de boucle
bne     $s2, $zero, start   // passage à l'itération suivante
```

Estimation du nombre de cycles :

Soit N le nombre d'instructions à exécuter dans un pipeline de processeur à M étages.

=> le nombre de cycles nécessaires à l'exécution sans aléas est : **$M + N - 1$**

Le nombre d'instructions de ce programme est :

$N = 1 + [\text{nombre d'itérations}] * [\text{nombre d'instructions dans la boucle}]$

=> $N = 1 + [1000] * [6] = 6001$ instructions

Le nombre de cycles nécessaires à l'exécution est :

$[\text{Nombre de cycles sans aléas}] + [\text{nombre de cycles dus aux aléas}]$

=> $[M + N - 1] + [\text{nombre de cycles dus aux instructions de branchements} + \text{nombre de cycles dus aux aléas de dépendances}]$

=> $[5 + 6001 - 1] + [999 * 2] + [1000 * 1]$

En effet, chaque branchement sauf le dernier génèrera une pénalité de 2 cycles (bne \$s2, \$zero, start), puisque le branchement est exécuté dans l'étage EX et donc les 2 instructions qui le suivent sont fetchées (i.e. ces instructions avancent au début du pipeline, dans IF puis ID). Ces 2 instructions seront donc annulées à cause du branchement.

Enfin, il existe une dépendance entre lw \$s1, 0(\$s3) et addi \$s1, \$s1, s qui est non résolue par **forwarding**, obligeant le pipeline à une pénalité de 1 cycle.

Le nombre total de cycles est donc :

$[5 + 6001 - 1] + [999 * 2] + [1000 * 1] = 9003$ cycles.

Q2 : Codez en assembleur la boucle dépliée et estimez le nombre de cycles nécessaires à son exécution pour le pipeline du processeur.

A2 :

Supposons que \$s3 contienne l'adresse de x[1000]

```
addi    $s2, $zero, 1000           // initialisation de l'indice de boucle.
```

start :

```
//----- // iteration i
```

```
lw      $s1,0($s3)
```

```
addi    $s1,$s1,s
```

```
sw      $s1,0($s3)
```

```
subi    $s3,$s3,4
```

```
//----- // iteration i+1
```

```
lw      $s1,0($s3)
```

```
addi    $s1,$s1,s
```

```
sw      $s1,0($s3)
```

```
subi    $s3,$s3,4
```

```
//----- // iteration i+2
```

```
lw      $s1,0($s3)
```

```
addi    $s1,$s1,s
```

```
sw      $s1,0($s3)
```

```
subi    $s3,$s3,4
```

```
//----- // iteration i+3
```

```
lw      $s1,0($s3)
```

```
addi    $s1,$s1,s
```

```
sw      $s1,0($s3)
```

```
subi    $s3,$s3,4
```

```
//----- // iteration i+4
```

```
lw      $s1,0($s3)
```

```
addi    $s1,$s1,s
```

```
sw      $s1,0($s3)
```

```
subi    $s3,$s3,4
```

```
//-----
```

```
subi    $s2,$s2,5           // mise à jour de l'indice de boucle
```

```
bne     $s2,$zero, start
```

Le nombre total d'instructions du corps de la boucle est 22.

Le nombre total d'instructions exécutées est donc : **$N = 1 + [22 * 200] = 4401$**

Le nombre d'instructions de branchement exécutées est de 200 dont 199 générant une pénalité de branchement de 2 cycles => **$[199 * 2]$**

Enfin il existe 5 aléas de dépendance par itération de boucle dus à la dépendance entre lw \$s1,0(\$s3) et addi \$s1,\$s1,s => le nombre de pénalités est alors :

nombre d'itérations * nombre d'aléas de dépendance par itération = $[200 * 5]$

Le nombre de cycles total nécessaires à l'exécution est donc:

$$[5 + 4401 - 1] + [199 * 2] + [200 * 5] = 4405 + 398 + 1000 = 5803 \text{ cycles}$$

=> Le speedup est de $9003/5803 = 1.55$, soit 1.55 plus rapide avec le loop unrolling

Q3: Reprenez les 2 estimations précédentes en considérant que le processeur est muni d'un prédicteur de branchement statique

A3 :

Le prédicteur de branchement statique va considérer que les branchements backwards sont pris et donc recherchera les bonnes instructions pour le fetch. Cela élimine les pénalités dues aux branchements dans les 2 équations précédentes :

$$[5 + 6001 - 1] + [999 * 2] + [1000 * 1] \text{ devient } [5 + 6001 - 1] + [1000 * 1] \Rightarrow 7005 \text{ cycles.}$$

$$[5 + 4401 - 1] + [199 * 2] + [200 * 5] \text{ devient } [5 + 4401 - 1] + [200 * 5] \Rightarrow 5405 \text{ cycles.}$$

Compilation – Optimisation automatique

Les compilateurs offrent des options de compilation permettant d'optimiser l'exécution des programmes. Le compilateur C/C++ d'Intel (*icc*: compilateur C - *icpc*: compilateur C++) offre certains niveaux d'optimisation dont les plus habituels sont -O*i* (avec *i*=0,1,2,3 - 3 maximum d'optimisation). L'ensemble des options peut être consulté avec *man icc*, ou bien en lisant le manuel `/opt/intel_cc/doc/c_ug_lnx.pdf`

Pour ce TP, nous allons utiliser la version 8.0 du compilateur *icc* (la version la plus récente est la version 13.0). Pour initialiser l'environnement, tapez la commande : *Use_intel_cc*

Options de compilation

Dans la première partie, nous souhaitons évaluer l'impact des différents niveaux d'optimisations sur le processeur de votre PC: **-O0, -O1, -O2, -O3**.

Linux* and Mac OS* X	Windows*	Description
-O3	/O3	Enables aggressive optimization for code speed. Recommended for code with loops that perform substantial calculations or process large data sets.
-O2 (or -O)	/O2	Affects code speed. This is the default option; the compiler uses this optimization level if you do not specify anything.
-O1	/O1	Affects code size and locality. Disables specific optimizations.
-fast	/fast	Enables a collection of common, recommended optimizations for run-time performance. Can introduce architecture dependency.
-O0	/Od	Disables optimization. Use this for rapid compilation while debugging an application.

Figure 31 Options de compilation pour icc

Ecrivez un programme qui fait la **multiplication de 2 matrices d'entiers, et de taille 1000** (array_mult.c), que vous initialiserez de manière aléatoire. Le résultat est sauvegardé dans une autre matrice de la même taille. Cette opération sera répétée **10** fois afin de calculer une moyenne plus ou moins précise du temps d'exécution d'une itération.

Pour mesurer précisément le temps pris pour faire ces multiplications, on utilise la fonction **clock()** (dans la librairie <time.h>) avant et après l'exécution du cœur de boucle. La différence entre ces 2 valeurs nous donnera le temps total en microsecondes.

***Q1 :** Compilez le programme avec chacune des options d'optimisations et complétez le tableau suivant :*

A1 :

Tableau 7 Résultats pour les différentes options d'optimisations

Option	Taille du programme (Byte)	Temps (secondes)
-O0	34980	24.78
-O1	35870	18.92
-O2	35870	18.92
-O3	35870	18.91

Observation 1: Taille du programme

Pour une compilation sans optimisation -O0, le compilateur a généré un programme qui est plus petit qu'avec les optimisations -O1, -O2, -O3. Ces derniers sont de taille identique. Cependant, le compilateur utilise des techniques d'optimisation qui, bien que pénalisantes en terme de nombre d'instructions (taille du programme), permettent en général d'augmenter les performances.

Observation 2 : Performances

Pour ce programme, le compilateur a fait les mêmes optimisations pour les options -O1, -O2, -O3. Ceci se remarque en analysant les temps d'exécution ainsi que la taille du programme (invariants). Le temps d'exécution pour les options -O1, -O2, -O3 est 18.92 secondes, soit 5.86 secondes d'amélioration par rapport à la version sans optimisation -O0. Notons que ces informations sont dépendantes du programme et ne peuvent pas être généralisées, puisque dans d'autres programmes, l'option -O3 peut donner des performances bien meilleures que les autres.

Loop unrolling

Dans la seconde partie, nous souhaitons évaluer l'impact de l'option « loop unrolling ».

Dans la version 8.0 du compilateur Intel, le « loop unrolling » est fait automatiquement par le compilateur avec les optimisations **O1, O2, O3**.

Le programmeur peut contrôler l'activation de cette optimisation en utilisant des « pragma » sur les boucles à partir de son code C.

La syntaxe de la directive 'unroll' est :

#pragma unroll	<--- optimisateur décide le nombre d'unrolling
#pragma unroll(n)	<--- unroll n times, $0 < n < 255$
#pragma nounroll	<--- disable loop unrolling

Cette directive doit être insérée avant la boucle la plus interne.

*Q2 : Insérez la directive **#pragma nounroll** avant la boucle la plus interne de votre programme, et recompilez avec l'option -O3. Que constatez-vous ?*

A2 :

Le temps d'exécution avec **#pragma nounroll** et compilation -O3 est : **18.104 secondes < 18.92**

secondes

Quand on force le compilateur à ne pas faire de « loop unrolling » pour la boucle interne, le temps d'exécution diminue et devient donc meilleur qu'avec l'option -O3, qui par défaut utilise cette optimisation. Ce résultat, qui peut paraître surprenant, sera expliqué dans Q3.

Q3 : *Insérez la directive **#pragma unroll(n)** avant la boucle la plus interne de votre programme en faisant varier le nombre **n**, et recompiler avec l'option « -O3 ». Complétez le tableau suivant. Que constatez-vous ?*

A3 :

Tableau 20 Résultats du **#pragma unroll(n)**

n	Taille du programme (Byte)	Temps (secondes)
2	35870	19.02
4	35870	18.35
8	35870	18.08
100	39966	17.838
255	53310	17.846

On constate que le compilateur n'a pas choisi automatiquement le nombre **n** optimal.

En fait, la décision du compilateur est basée sur des heuristiques qui ne peuvent pas être optimales, et qui nécessitent donc l'intervention du programmeur pour ajuster sa décision.

Pour faire un choix optimal, le programmeur doit comprendre la limite des optimisations du compilateur, l'architecture du processeur, ainsi que l'algorithme du programme.

Q4 : *Facultatif : En gardant la directive **#pragma nounroll**, effectuez un « loop unrolling » **n** fois de votre boucle manuellement afin de regagner le temps perdu. Vous utiliserez la valeur optimale de **n** obtenue en **Q3**.*

N.B : Tous ces résultats et ces analyses dépendent forcément de la machine utilisée, de la version du compilateur, et du programme que vous avez codé.

Machine utilisée :

AMD Athlon 64-bit Dual core, fréquence = 1 GHz, L1 cache = 128 KB, L2 cache = 512 KB

Compilateur :

Intel C compiler (icc) version 8.0

Code array_mult.c :

```
#include <stdio.h>

#include <stdlib.h>
#include <time.h>

#define N 1000
int main ()
{
    double start, end;

    int loop;
    int i,j,k, count;
    int **a, **b, **c;

    a = (int **) malloc(N*sizeof(int*));
    b = malloc(N*sizeof(int*));
    c = malloc(N*sizeof(int*));

    for (i=0; i < N; i++)
    {
        a[i] = (int *) malloc(N*sizeof(int));
        b[i] = (int *) malloc(N*sizeof(int));
        c[i] = (int *) malloc(N*sizeof(int));
    }

    loop = 10 ;

    for (i=0; i < N; i++)
    {
        for (j=0; j < N; j++)
        {
            a[i][j] = i % 10;
            b[i][j] = i % 20;
            c[i][j] = 0;
        }
    }

    start = clock();

    count = 0;
```



```
while (count < loop)
{
    for (i=0; i < N; i+=1)
    {
        //#pragma nounroll
        //#pragma unroll
        //#pragma unroll(2)
        //#pragma unroll(4)
        //#pragma unroll(8)
        //#pragma unroll(100)
        //#pragma unroll(255)
        for (j=0; j < N; j++)
        {
            c[i][j] = 0;

            for (k=0; k < N; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
    }
    count++;
}
end = clock() - start;
printf("End of program unroll at elapsed time = %10E ...average=%f\n", end, end/loop);
return 0;
}
```



**INSTITUT
POLYTECHNIQUE
DE PARIS**

Annexe 1 : Cache

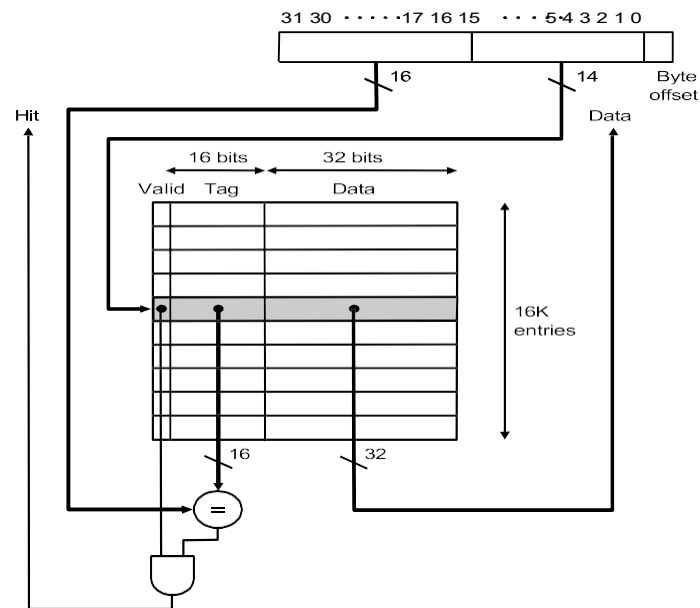


Figure 32 Organisation de cache : exemple DECstation 3100 direct-mapped cache (64 KB, 16 Kwords, 1 word/block)

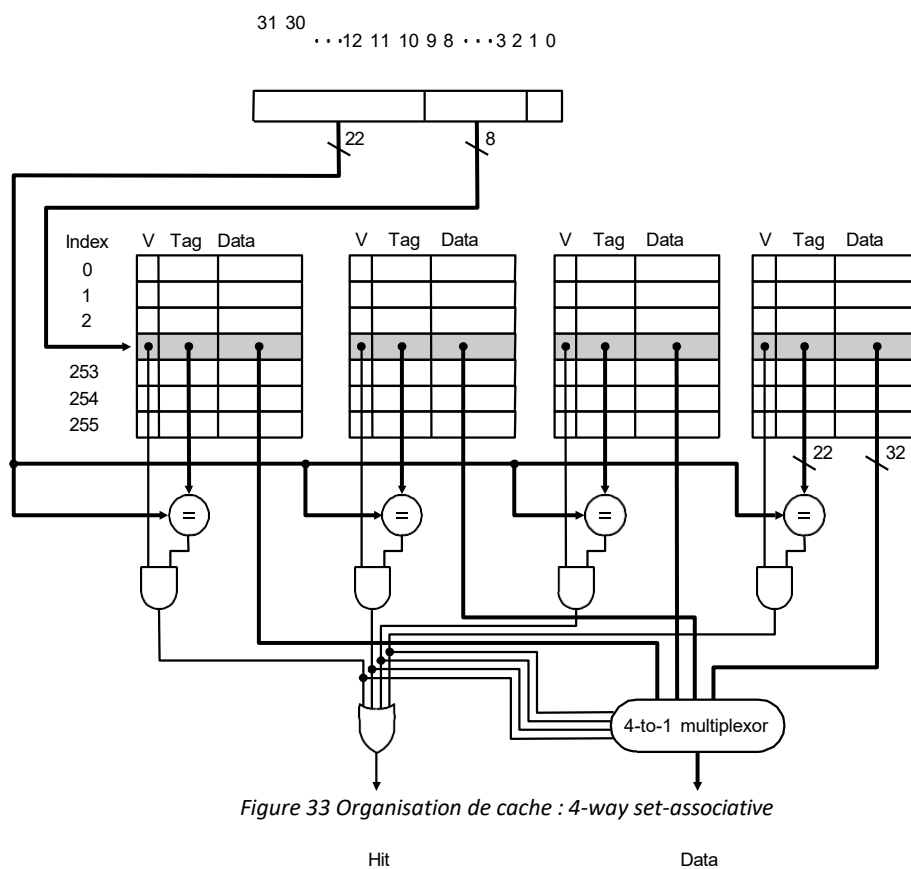


Figure 33 Organisation de cache : 4-way set-associative

Annexe 2 : Applications (A modifier)

Les codes sources de PageRank et de BCS peuvent être récupérés en utilisant la commande suivante, ou *<target directory>* doit être remplacé par le répertoire local que vous ciblez :

```
cp -rfv /home/c/nom-mdc/ES201/tools/graph/* <target directory>
```

avec **nom-mdc** le nom de votre maître de conférences

PageRank

Une implémentation C de l'algorithme de PageRank est proposée. Le source tree est le suivant :

*/home/c/**nom-mdc**/ES201/tools/graph/PageRank/:*

- main.c examples.h (source code)
- Makefile
- pagerank (binary file, compiled with gcc)
- pagerank_* (binary file for SimpleScalar:
 - o pagerank_min (smallest graph example)
 - o pagerank_med (medium graph example)
 - o pagerank_max (biggest graph example)

Nettoyage du répertoire : **make clean**

Compilation de tous les binaires : **make all**

Betweenness Centrality Score – SSCA #2

Le benchmark *Synthetic Scalable Compact Applications #2* contient des applications de type traitement de graphes. En l'occurrence le benchmark est composé de plusieurs étapes (*kernels*) dont notamment :

- Génération du graphe (2^{SCALE} sommets / $8 \cdot 2^{\text{SCALE}}$ arcs)
- Prétraitements (sélection de sous-ensembles et extraction)
- Analyse du graphe (Calcul du score de centralité d'intermédiarité pour un sous set de sommet).

La génération du binaire se fait via "make". **USAGE** : `sim-* SSCA2.ss <SCALE>` (valeurs recommandées : entre 4 et 8).

Source tree:

*/home/c/**nom-mdc**/ES201/tools/graph/SSCA2v2-C/*

- *.ch] (source code)
- *.doc README (article/document détaillant la suite SSCA #2)
- Makefile
- SSCA2.ss (binary file for SimpleScalar)

La multiplication de polynômes

Voici un petit rappel sur la multiplication de polynôme. Multiplier deux polynômes de degré N ($N+1$ coefficients) revient à faire le calcul de la convolution linéaire de leur séquence de coefficient. Soit $A = \sum_{i=0}^N a_i \cdot X^i$ et $B = \sum_{i=0}^N b_i \cdot X^i$ deux polynômes.

Le polynôme produit de A par B :

$$C = A * B = \sum_{i=0}^{2N} c_i \cdot X^i$$

Les coefficients de C vérifient pour tout $i \in \llbracket 0; 2N \rrbracket$:

$$c_i = \sum_{k=0} a_k \cdot b_{i-k}.$$

Secure Hash Algorithm (SHA-1)

Les fonctions de hachage (hash functions) calculent une empreinte servant à identifier rapidement, bien qu'incomplètement, une donnée initiale. En cryptographie ces fonctions sont employé entre autres pour la signature électronique, et rend également possibles des mécanismes d'authentification par mot de passe sans stockage de ce dernier.

Le code C utilisé dans ce TP provient de l'ensemble de benchmarks MiBench. Le Makefile original a été modifié pour compiler l'application avec le compilateur de SimpleScalar.

Récupérez le code source avec les commandes suivantes :

```
cp /home/c/nom-mdc/ES201/tools/sha_modified.tar.gz <your_directory>
```

```
tar xzf <your_directory>/sha_modified.tar.gz
```

Compiler le binaire pour SimpleScalar:

```
cd <your_directory>/sha
```

```
make
```

Le binaire pour SimpleScalar est désormais à votre disposition: `<your_directory>/sha/sha.ss` Le fichier d'entrée qui nous intéresse est: `<your_directory>/sha/input_small.asc`

Voici un exemple de lancement d'une simulation SimpleScalar :

```
sim-outorder <options> sha.ss input_small.asc
```

Bloc cipher BlowFish

Les chiffrements par bloc (bloc cipher) sont les éléments centraux des protocoles de chiffrement symétrique [TP4/11]. BlowFish est l'un des premiers chiffrements par bloc à être entré dans le domaine public (non breveté).

Le code C utilisé dans ce TP provient de l'ensemble de benchmarks MiBench. Le Makefile original ainsi que les fichiers d'inputs ont été modifié pour s'adapter à notre TP.

Récupérez le code source avec les commandes suivantes :

```
cp /home/c/nom-mdc/ES201/tools/blowfish_modified.tar.gz <your_directory>
```

```
tar xzf <your_directory>/blowfish_modified.tar.gz
```

Compiler le binaire pour SimpleScalar:

```
cd <your_directory>/blowfish
```

```
make
```

Le binaire pour SimpleScalar est désormais à votre disposition: <your_directory>/blowfish/bf.ss

Les fichiers d'entrée qui nous intéressent sont:

<your_directory>/blowfish/input_[small/medium/large].asc

Voici un exemple de lancement d'une simulation SimpleScalar :

```
sim-outorder <options> bf.ss e input_small.asc output_small.enc 1234567890abcdeffedcba0987654321
```

Avec:

e : option indiquant à bf.ss qu'on est en mode chiffrement (encrypt)

input_small.asc : le fichier d'entrée (les données à chiffrer)

output_small.dec : le fichier de sortie (les données chiffrées)

1234567890abcdeffedcba0987654321 : la clé de chiffrement

Autres applications 2020 :

- **MLPerf** : Fair and useful benchmarks for measuring training and inference performance of ML hardware, software, and services. <https://mlperf.org/>
- **EEMBC** : <https://www.eembc.org/>

Annexe 3 : gem5 – généralités

Le simulateur gem5 est une plateforme d'exploration architecturale issue de la fusion de deux projets de recherche (GEMs et M5). Cette plateforme est de plus en plus utilisée, aussi bien dans le milieu industriel que dans le domaine de la recherche. Citons à titre d'exemple les contributions actives sur le projet de AMD, ARM, HP, MIPS, Princeton, MIT, et les universités du Michigan, du Texas et du Wisconsin.

Cet outil étant de plus en plus admis par la communauté, nous nous proposons de l'utiliser pour explorer quelques paramètres d'une architecture multiprocesseur. L'utilisateur de la plateforme gem5 a la possibilité de construire et de simuler des systèmes complets, en connectant de nombreux éléments paramétrables : modèles de processeurs (scalaires, superscalaires in-order, superscalaires out-of-order), jeux d'instructions (ARM, Alpha, MIPS, PowerPC, SPARC, x86, RISC-V), systèmes mémoire, protocoles de cohérence de cache, bus système, gestionnaire d'appels systèmes, etc.

Pour mieux comprendre l'infrastructure de ce simulateur, veuillez consulter la documentation depuis : <https://www.gem5.org/documentation/>

Toutes les étapes pour construire et utiliser un environnement de simulation complet sont détaillées ci-dessous.

1. Résolution des dépendances, récupération des sources et compilation de gem5 + cross-compileur RISC-V :

La liste des dépendances nécessaires pour l'obtention et la compilation des fichiers sources de gem5 peut être trouvée à l'adresse suivante :

https://www.gem5.org/documentation/general_docs/building

A titre d'exemple, la commande suivante convient sur une distribution Ubuntu récente :

```
> sudo apt update && sudo apt install -y build-essential git m4 scons python3 python3-dev zlib1g-dev  
libprotobuf-dev protobuf-compiler libprotobuf-dev libgoogle-perftools-dev swig gcc-riscv64-linux-gnu  
g++-riscv64-linux-gnu libc6-dev-riscv64-cross
```

Sous réserve de pouvoir résoudre les dépendances ci-dessus, il est bien évidemment possible d'utiliser gem5 dans votre distribution préférée ou dans un environnement différent.

Pour ceux qui travaillent sur Windows, il est préférable d'utiliser WSL/Ubuntu avec la version 22.04.

Ensuite, les commandes suivantes permettent de se déplacer dans le répertoire d'installation locale de gem5 et d'extraire le dépôt stable le plus récent de la plateforme :

```
> cd /path/to/local-installation  
  
> git clone https://gem5.googlesource.com/public/gem5  
  
> cd gem5
```

```
>      scons build/RISCV/gem5.opt -j$(nproc)
```

Notons qu'il est possible de compiler différents types de simulateurs (*gem5.debug*, *gem5.opt*, *gem5.fast*, *gem5.prof*, *gem5.perf*) suivant les besoins de l'utilisateur. Pour plus d'informations, se référer à la documentation.

Notons également que dans le cas présent, la plateforme compilée est un simulateur dont les processeurs utilisent le jeu d'instruction RISC-V. Il est bien sûr possible de construire une plateforme équivalente pour un autre type de jeu d'instructions, pour peu qu'il soit supporté (voir ci-dessus).

2.(optionnel) Installation de la librairie m5threads :

La librairie *m5threads*, développée par des contributeurs de *gem5*, est nécessaire pour implémenter des applications multithreads (création, destruction, gestion de threads). Il faut donc d'abord récupérer les sources de la librairie sur le dépôt suivant :

```
>      cd /path/to/local-installation
>      git clone https://github.com/gem5/m5threads
>      cd m5threads/tests
```

Ensuite, on modifie (si nécessaire) le **Makefile** situé à la racine du dépôt afin de compiler la librairie avec la chaîne d'outils installée dans l'étape 1, puis :

```
>      make
```

3. Compilation et exécution d'un programme sur le simulateur gem5 :

Il existe principalement deux manières d'utiliser le simulateur *gem5*, dont les différentes spécificités et les différents cas d'utilisations sont trouvables sur la documentation en ligne :

- FS mode (Full System mode)
- SE mode (Syscall Emulation mode)

Dans le cadre des TD/TP, nous utiliserons le mode SE. Par défaut, le répertoire *gem5* contient déjà des exécutables d'exemple afin de lancer une première simulation, comme ceci :

```
>      build/RISCV/gem5.opt configs/example/se.py -c tests/test-progs/hello/bin/riscv/linux/hello
```

gem5.opt : exécutable du simulateur

se.py : fichier de configuration de la plateforme

hello : binaire à simuler

Pour compiler notre propre programme à exécuter dans gem5, on utilise la chaîne d'outils installée dans l'étape 1. Notons qu'il est nécessaire d'embarquer, dans l'exécutable qui sera généré, tous les symboles (fonctions, données) contenues habituellement dans des bibliothèques dynamiques. Pour ce faire, on compilera le programme en statique (ci-dessous et pour exemple *prog.c*) :

```
> riscv64-linux-gnu-gcc -O2 -static -o prog.riscv prog.c
```

Cas particulier : Si votre programme est multithread, il faut bien évidemment linker le fichier objet (*libpthread.so*) de gestion de threads, créé dans l'étape 2. Lors de la compilation de l'application, la commande de création de liens devient donc :

```
> /path/to/toolchain/bin/riscv64-linux-gnu-gcc -o matrix_mult.exe matrix_mult.o pthreads.o -static
```

Note : À tout moment, vous pouvez accéder à la documentation de l'exécutable du simulateur et à la documentation de la configuration utilisée :

```
> build/RISCV/gem5.opt --help
```

```
> build/RISCV/gem5.opt config/examples/se.py --help
```

4. Utilisation de m5opts.h :

GEM5 fournit, via le fichier *m5ops.h*, un ensemble d'instructions spéciales permettant au programme simulé d'interagir directement avec le simulateur. Ces instructions, appelées *m5ops*, permettent notamment de réinitialiser les statistiques (*m5_reset_stats*), de les sauvegarder (*m5_dump_stats*), ou encore de contrôler l'exécution de la simulation. Elles sont particulièrement utiles pour définir une *region of interest* (ROI) et mesurer précisément les performances d'une portion donnée du code, en excluant les phases d'initialisation ou de terminaison. L'utilisation de *m5ops.h* permet ainsi d'obtenir des statistiques fiables et reproductibles, centrées uniquement sur le noyau de calcul étudié.

Son utilisation nécessite la compilation de la bibliothèque *libm5.a* fournie avec GEM5, [m5ops](#):

```
> cd ~/gem5/util/m5
> scons build/riscv/out/m5
```

Si vous obtenez le message d'erreur :

```
> riscv64-unknown-linux-gnu-g++: not found
```

Il faudra créer les liens symboliques :

```
> sudo ln -s /usr/bin/riscv64-linux-gnu-gcc /usr/bin/riscv64-unknown-
linux-gnu-gcc
> sudo ln -s /usr/bin/riscv64-linux-gnu-g++ /usr/bin/riscv64-unknown-
linux-gnu-g++
> sudo ln -s /usr/bin/riscv64-linux-gnu-ar /usr/bin/riscv64-unknown-linux-
gnu-ar
> sudo ln -s /usr/bin/riscv64-linux-gnu-ld /usr/bin/riscv64-unknown-linux-
gnu-ld
```

Puis vérifier la présence de la librairie :

```
> ls /path_to_gem5/util/m5/build/riscv/out/  
➔ libm5.a m5
```

Nouvelle compilation avec libm5.a :

```
> riscv64-linux-gnu-gcc -O2 -static -I/path_to_gem5/include conv_m5.c  
/path_to_gem5/util/m5/build/riscv/out/libm5.a -o prog.riscv
```

5. Statistiques d'exécution de gem5 :

Pour visualiser les statistiques d'exécution d'un programme, ouvrir avec votre éditeur de texte favori le fichier généré suivant :

```
> $WORK/m5out/stats.txt
```

Note annexe : Vous pouvez exécuter le simulateur *gem5.opt* en faisant varier une très importante quantité de paramètres système. Pour cela, il convient de bien se familiariser avec la structure et la syntaxe du simulateur, afin de pouvoir modifier correctement la structure du système, qui pour rappel est décrite dans le fichier :

```
> $WORK/config/examples/se.py
```

Annexe 4 : gem5 – mémo TP5-6

Log in sur machine salle 2013 :

```
> ssh <votre_login>@salle2013.ensta.fr
```

Compilation de l'application de multiplication de matrice :

Récupérer et compiler les sources de m5threads :

```
> hg clone http://repo.gem5.org/m5threads
> cd m5threads/tests
> make
```

Commande pour lancer une simulation :

<GEM5 exécutable> <architecture système> <options architecture> <exécutable appli>

où :

<GEM5 executable> : **/opt/gem5/build/RISCV/gem5.[opt|fast]**

<architecture système> : **/opt/gem5/configs/example/se.py**

<options architecture> : **--cpu-type = atomic|arm-detailed|detailed**

Atomic => Processeur scalaire

arm-detailed => Cortex A7

detailed => Cortex A15

--caches *//à utiliser avec 'arm-detailed' et 'detailed'*

-n NCPU *//NCPU = nombre de CPUs*

-w WIDTH *//WIDTH = issue width pour 'detailed' model*

< exécutable appli> : **-c <nom de l'exécutable>**

-o "<args>"

<nom de l'exécutable> = **\$HOME/m5threads/tests/test_omp**

<args> = **"N M"**

N: nombre de threads

M: taille de la matrice carrée

N.B#1 : le nombre de CPU **NCPU** doit être plus grand ou égal au nombre de thread logiciel **N**

N.B#2 : pour savoir les paramètres de l'architecture modélisée, il suffit de consulter l'aide

/opt/gem5/build/RISCV/gem5.fast /opt/gem5/configs/example/se.py --help

Statistiques :

Les statistiques de la simulation se trouvent dans **m5out/stats.txt**

Exemples de statistiques intéressantes pour le TD/TP6 :

- > **sim-insts** : total executed instructions (on all cores)
- > **system.cpu<i>.committedInstns** : executed instructions on core <i>
- > **system.cpu<i>.numCycles** : cycles on core <i>
- > **system.cpu<i>.cpi**
- > **system.cpu<i>.ipc**
- > **Les statistiques de caches**

Commandes utiles :

TP5 (fichier /home/c/nom-mdc/ES201/tools/TP5/README):

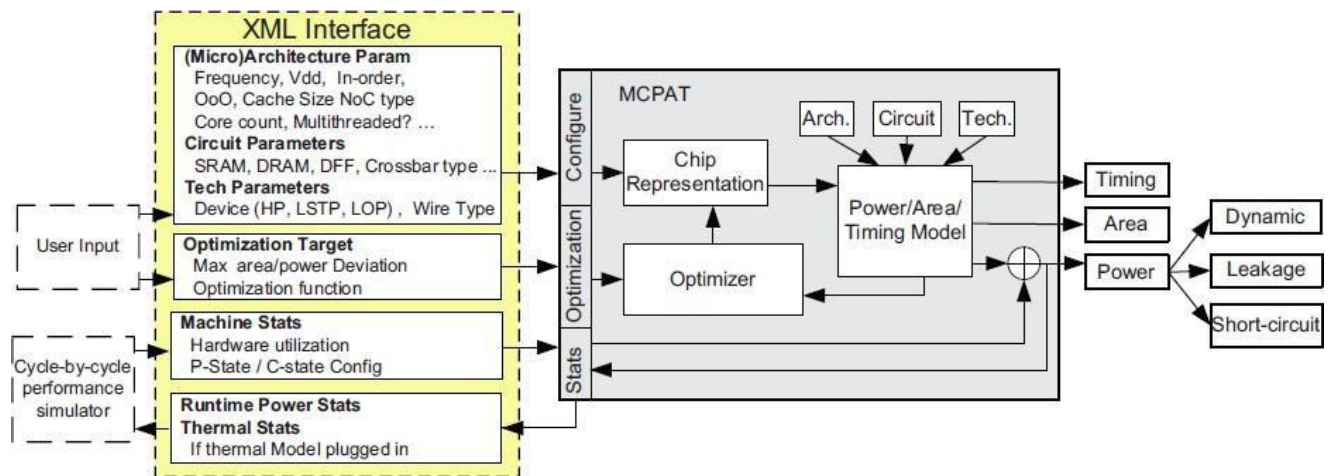
\$cp -v /home/c/nom-mdc/ES201/tools/TP5/test_omp <target directory>

export GEM5=/home/c/nom-mdc/ES201/tools/TP5/gem5-stable

\$GEM5/build/ARM/gem5.fast \$GEM5/configs/example/se.py -c <bin> -o "<options>"

Annexe 5 : McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures

<https://www.hpl.hp.com/research/mcpat/>



Block diagram of the McPAT framework.

Références

www.intel.com, www.arm.com, www.amd.com, www.ieee.org, www.acm.org, www.cadence.com,
www.synopsys.com, www.qualcomm.com, www.mentor.com

Energy Efficient Microprocessor Design Burd, T. D., Brodersen, R. W. (2002)

Computer Organization and Design RISC-V Edition: The Hardware Software Interface (ISSN) (English Edition) 1 de D.A. Patterson , J. L. Hennessy

Parallel Computer Organization and Design by Michel Dubois , Murali Annavaram, et al.

Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design) J.L. Hennessy and D.A. Patterson

Computer Organization and Architecture (10th Edition) 10th Edition by William Stallings

TP2

Online pipeline simulator: <http://www.ecs.umass.edu/ece/koren/architecture/windlx/main.html>

[1] **D. A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, T. Meuse**, HPCS Scalable Synthetic Compact Applications #2 Graph Analysis
http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.1.pdf

[2] **Léo Ducas**, Une cryptographie nouvelle: le réseau euclidien, GLMF-178, GNU/Linux Magazine, <http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-178/Une-cryptographie-nouvelle-le-reseau-euclidien>

[3] **J. Fan and F. Vercauteren**, Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

TP3

[1] **L. Page, S. Brin, R. Motwani, and T. Winograd**, The PageRank Citation Ranking: Bringing Order to the Web. Technical Report. Stanford InfoLab <http://ilpubs.stanford.edu:8090/422/>

TP4

[1] EDN 2010 Microprocessor Directory

[2] Intel : www.intel.com

[3] AMD : www.amd.com

- [4] Arm : www.arm.com
- [5] MIPS : www.mips.com
- [6] ARM big.LITTLE : www.arm.com/files/downloads/big.LITTLE_Final.pdf
- [7] MiBench : www.eecs.umich.edu/mibench
- [8] CACTI : <http://www.hpl.hp.com/research/cacti/>
- [9] B. Schneier, Applied Cryptography, *Protocols Algorithms and Source Code in C*, 1996, John Wiley & Sons
- [10] B. Schneier,
https://www.schneier.com/academic/archives/1994/09/description_of_a_new.html
- [11] M. Bellare and P. Rogaway, Introduction to modern cryptography, *Lectures Notes*,
<https://cseweb.ucsd.edu/~mihir/cse207/classnotes.html>

TP5

- [1] Intel Pentium 4 HT : <http://www.intel.com/technology/platform-technology/hyper-threading/index.htm>
- [2] IBM POWER5 : <http://www-03.ibm.com/systems/power/>
- [3] Sun UltraSPARC T2 : www.sun.com/processors/UltraSPARC-T2/
- [4] Intel Ivy Bridge : <http://www.intel.com/technology/architecture-silicon/next-gen/>
- [5] gem5 : <http://www.m5sim.org/>
- [6] OpenMP : <http://www.openmp.org/>

TP6

- [1] <http://www.connaissancedesenergies.org/fiche-pedagogique/chiffres-cles-production-d-energie>
- [2] http://www.nytimes.com/2012/09/23/technology/data-centers-waste-vast-amounts-of-energy-belying-industry-image.html?_r=0
- [3] **Carroll, Aaron and Heiser, Gernot**, “The Systems Hacker’s Guide to the Galaxy, Energy Usage in a Modern Smartphone”, available online (<http://www.nicta.com.au/pub?doc=7044>)
- [4] **Shye, Alex et al.**, “Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures” available online (<http://dl.acm.org/citation.cfm?doid=1669112.1669135>)

École Nationale Supérieure
de **Techniques Avancées**

828, boulevard des Maréchaux - 91762 Palaiseau Cedex - France
www.ensta-paris.fr

ENST2



**INSTITUT
POLYTECHNIQUE
DE PARIS**