UNIVERSITE M'Hamed BOUGARA – BOUMERDES
FACULTE DES SCIENCES
DEPARTEMENT D'INFORMATIQUE

Module : Architecture des Ordinateurs **Responsable :** Mokrani Hocine
Filière : Licence Informatique **Semestre :** 3

Prise en main : Assembleur MIPS 32 bits & QtSPIM

I. Assembleur MIPS

Cette section décrit le langage assembleur du processeur MIPS R3000. On présente ici successivement l'organisation de la mémoire centrale, les instructions et les macro-instructions, les directives acceptées par l'assembleur, les quelques appels système disponibles dans le simulateur QtSpim, ainsi que quelques remarques sur les conventions imposées pour les appels de fonctions et la gestion de la pile.

Cette section résume les informations de l'article : ftp://asim.lip6.fr/pub/mips/mips.asm.pdf.

I.1. Organisation de la mémoire

Dans l'architecture MIPS R3000, l'espace adressable est divisé en deux segments : le segment utilisateur, et le segment noyau. La figure 1 montre l'organisation de la mémoire d'une architecture MIPS avec les différentes adresses.

Un programme utilisateur utilise généralement trois sous-segments (appelés sections) dans le segment utilisateur :

- **Section `text` :** contient le code exécutable en mode utilisateur. Elle est implantée conventionnellement à l'adresse `0x00400000`. Sa taille est fixe et calculée lors de l'assemblage. La principale tâche de l'assembleur consiste à générer le code binaire correspondant au programme source décrit en langage d'assemblage, qui sera chargé dans cette section ;
- **Section `data` :** contient les données globales manipulées par le programme utilisateur. Elle est implantée conventionnellement à l'adresse `0x10000000`. Sa taille est fixe et calculée lors de l'assemblage. Les valeurs contenues dans cette section peuvent être initialisées grâce a des directives contenues dans le programme source en langage d'assemblage;
- **Section `stack` :** contient la pile d'exécution du programme. Sa taille varie au cours de l'exécution. Elle est implantée conventionnellement à l'adresse `0x7FFFEFFF`. Contrairement aux sections `data` et `text`, la pile s'étend vers les adresses décroissantes.

Trois autres sections sont définies dans le segment noyau :

- **Section `ktext` :** contient le code exécutable en mode noyau. Elle est implantée conventionnellement à l'adresse `0x80000000`. Sa taille est fixe et calculée lors de l'assemblage ;
- **Section `kdata` :** contient les données globales manipulées par le système d'exploitation en mode noyau. Elle est implantée conventionnellement à l'adresse `0xC0000000`. Sa taille est fixe et calculée lors de l'assemblage ;
- Section `kstack` : contient la pile d'exécution du programme. Sa taille varie au cours de l'exécution. Elle est implantée conventionnellement à l'adresse `0xFFFFEFFF`. Contrairement aux sections `kdata` et `ktext`, la pile s'étend vers les adresses décroissantes.

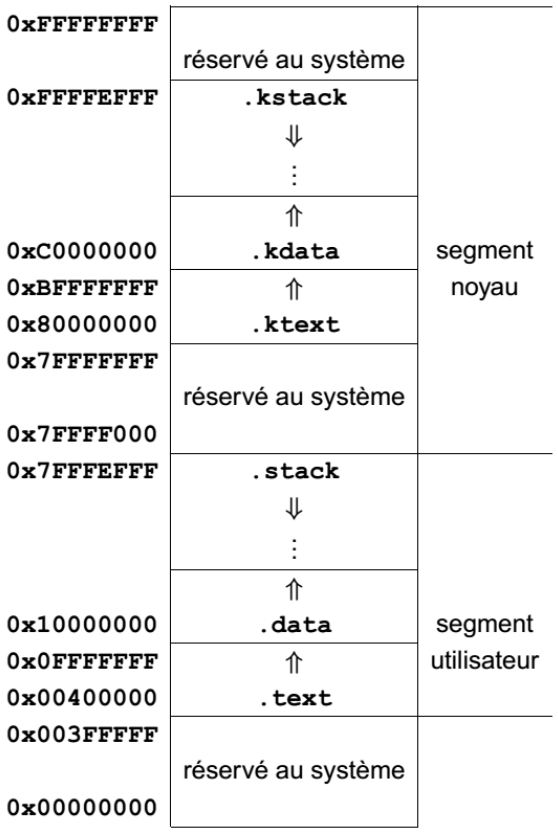

Figure 1 : Organisation de la mémoire de l'architecture MIPS R3000

I.2. Règles syntaxiques

- **Noms de fichiers :**
  Les noms des fichiers contenant un programme source en langage d'assemblage doivent être suffixés par « .s ». Exemple : « Tprogramme.s ».

- **Commentaires :**
  Ils commencent par un # ou un ; et s'achèvent à la fin de la ligne courante.

- **Entiers :**
  Une valeur entière décimale est notée 250, une valeur entière octale est notée 0372 (préfixée par un zéro), et une valeur entière héxadécimale est notée 0xFA (préfixée par zéro suivi de x). En hexadécimal, les lettres de A à F peuvent être écrites en majuscule ou en minuscule.

- **Chaînes de caractères :**
  Elles sont simplement entre guillemets, et peuvent contenir les caractères d'échappement du langage C.

- **Labels :**
  Ce sont des mnémoniques correspondant à des adresses. Ces adresses peuvent être soit des adresses de variables, soit des adresses de saut. Ce sont des chaînes de caractères qui commencent par une lettre, majuscule ou minuscule, un « $ », un « _ », ou un «.». Ensuite, un nombre quelconque de ces mêmes caractères auxquels on ajoute les chiffres. Pour la déclaration, le label doit être suffixé par « : ».

- **Valeurs immédiats :**
  Ce sont les opérandes contenus dans l'instruction. Ce sont des constantes. Ils sont soit des entiers, soit des labels. Ces constantes doivent respecter une taille maximum qui est fonction de l'instruction qui l'utilise : 16 ou 26 bits.

- **Registres :**
  Le processeur M IPS possède 32 registres accessibles au programmeur. Chaque registre est connu par son numéro, qui varie entre 0 et 31, et est préfixé par un $. Le tableau 1 montre les différents registres avec leur description.

Nom	Numéro	Description
$zero	0	constante 0
$at	1	réservé à l'assembleur
$v0,$v1	2-3	résultats d'évaluation
$a0,..,$a3	4-7	arguments de procédure
$t0,..,$t7	8-15	valeurs temporaires
$s0,..,$s7	16-23	sauvegardes
$t8,$t9	24-25	temporaires
$k0,$k1	26-27	réservé pour les interruptions
$gp	28	pointeur global
$sp	29	pointeur de pile
$fp	30	pointeur de bloc
$ra	31	adresse de retour

- **Arguments :**
  Si une instruction nécessite plusieurs arguments, comme par exemple l'addition entre deux registres, ces arguments sont séparés par des virgules, dans lequel est mis le résultat de l'opération, puis ensuite le premier registre source. Exemple : add $3, $2, $1

I.3. Instructions MIPS

a) Instructions de transferts

Syntaxe Assembleur		Opérations	Commentaires	Effet	Format
Opérations de transferts ALU (*move from/to*)					
mfhi	Rd	Copie du champ Hi	*cf.* mult/div	Rd ← Hi	R
mflo	Rd	Copie du champ Lo	*cf.* mult/div	Rd ← Lo	R
mthi	Rs	Chargement du champ Hi	*cf.* mult/div	Hi ← Rs	R
mtlo	Rs	Chargement du champ Lo	*cf.* mult/div	Lo ← Rs	R
lui	Rt	Chargement immédiat	Chargement d'une constante	Rt ← I << 16	R
Opérations de transferts registre-RAM (*load/store*)					
lw	Rt, I(Rs)	Chargement mot		Rt ← RAM[Rs+I]	I
sw	Rt, I(Rs)	Déchargement mot		RAM[Rs+I] ← Rt	I
lh	Rt, I(Rs)	Chargement demi-mot	Extension de signe	Rt ← RAM[Rs+I]	I
lhu	Rt, I(Rs)	Chargement demi-mot non-signé	Sans extension de signe	Rt ← RAM[Rs+I]	I
sh	Rt, I(Rs)	Déchargement demi-mot		RAM[Rs+I] ← Rt	I
lb	Rt, I(Rs)	Chargement octet	Extension de signe	Rt ← RAM[Rs+I]	I
lbu	Rt, I(Rs)	Chargement octet non-signé	Sans extension de signe	Rt ← RAM[Rs+I]	I
sb	Rt, I(Rs)	Déchargement octet		RAM[Rs+I] ← Rt	I

b) Instructions de branchement de code

Syntaxe Assembleur		Opérations	Commentaires	Effet	Format
Saut conditionnel (*branch if equal/not equal/greater/less... [than zero] [and link]*)					
beq	Rs, Rt, Etq	Branchement si égalité		PC ← PC + 4 + 4 × I si Rs=Rt PC ← PC + 4 sinon	I
bne	Rs, Rt, Etq	Branchement si différent		PC ← PC + 4 + 4 × I si Rs≠Rt PC ← PC + 4 sinon	I
bgez	Rs, Etq	Branchement si sup. ou égal à 0		PC ← PC + 4 + 4 × I si Rs⩾ 0 PC ← PC + 4 sinon	I
bgtz	Rs, Etq	Branchement si sup. à 0		PC ← PC + 4 + 4 × I si Rs > 0 PC ← PC + 4 sinon	I
blez	Rs, Etq	Branchement si inf. ou égal à 0		PC ← PC + 4 + 4 × I si Rs ⩽ 0 PC ← PC + 4 sinon	I
bltz	Rs, Etq	Branchement si inf. à 0		PC ← PC + 4 + 4 × I si Rs < 0 PC ← PC + 4 sinon	I
bgezal	Rs, Etq	Branchement si sup. ou égal et liaison		PC ← PC + 4 + 4 × I si Rs ⩾ 0 PC ← PC + 4 sinon R31 ← PC + 4 dans tous les cas	I
bltzal	Rs, Etq	Branchement si inf. ou égal et liaison		PC ← PC + 4 + 4 × I si Rs < 0 PC ← PC + 4 sinon R31 ← PC + 4 dans tous les cas	I
Saut inconditionnel (*jump [and link]*)					
j	Etq	Saut étiquette		PC ← PC + 4[31...28].4×Etq	J
jal	Etq	Saut et liaison		R31 ← PC + 4 PC ← PC + 4[31...28].4×Etq	J
jr	Rs	Saut sur registre		PC ← Rs	R
jalr	Rs	Saut et lien sur registre		R31 ← PC + 4 PC ← Rs	R
jalr	Rd, Rs	Saut et lien sur registre		Rd ← PC + 4 PC ← Rs	R

c) Instructions logiques

Syntaxe Assembleur		Opérations	Commentaires	Effet	Format
Opérations logiques de base (bits-à-bits)					
or	Rd, Rs, Rt	OU logique		Rd ← Rs | Rt	R
and	Rd, Rs, Rt	ET logique		Rd ← Rs & Rt	R
xor	Rd, Rs, Rt	OU exclusif logique		Rd ← Rs ^ Rt	R
nor	Rd, Rs, Rt	NON OU logique		Rd ← ∼(Rs | Rt)	R
ori	Rt, Rs, I	OU logique avec op. imm.	immédiat non-signé	Rt ← Rs | I	I
andi	Rt, Rs, I	ET logique avec op. imm.	immédiat non-signé	Rt ← Rs & I	I
xori	Rt, Rs, I	OU exclusif avec op. imm.	immédiat non-signé	Rt ← Rs ^ I	I
Opérations de décalage de bits (*shift left/right logical)*					
sllv	Rd, Rt, Rs	Décalage à gauche avec registre	Rs : 5 bits de poids faibles	Rd ← Rt << Rs	R
srlv	Rd, Rt, Rs	Décalage à droite avec registre	Rs : 5 bits de poids faibles	Rd ← Rt >> Rs	R
srav	Rd, Rt, Rs	Décalage arithmétique* à droite avec registre	Rs : 5 bits de poids faibles	Rd ← Rt >>* Rs	R
sll	Rd, Rt, Dec	Décalage à gauche		Rd ← Rt << Dec	R
srl	Rd, Rt, Dec	Décalage à droite		Rd ← Rt >> Dec	R
sra	Rd, Rt, Dec	Décalage à droite arithmétique	Extension de signe	Rd ← Rt >>* Dec	R
Opérations de test de conditions (*set if less than*)					
slt	Rd, Rs, Rt	positionné si inférieur à		Rd ← 1 si (Rs < Rt) Rd ← 0 sinon	R
sltu	Rd, Rs, Rt	positionné si inférieur à (op. non-signé)		Rd ← 1 si (Rs < Rt) Rd ← 0 sinon	R
slti	Rt, Rs, I	positionné si inférieur à (op. immédiat)	extension de signe pour l'immédiat	Rt ← 1 si (Rs < I) Rt ← 0 sinon	I
sltiu	Rt, Rs, I	positionné si inférieur à (op. immédiat, non-signé)		Rt ← 1 si (Rs < I) Rt ← 0 sinon	I

d) Instructions arithmétiques

Syntaxe Assembleur		Opérations	Commentaires	Effet	Format
Opérations d'addition et de soustraction					
add	Rd, Rs, Rt	addition	Détection de dépassement de capacité	Rd ← Rs + Rt	R
sub	Rd, Rs, Rt	soustraction	Détection de dépassement de capacité	Rd ← Rs − Rt	R
addu	Rd, Rs, Rt	addition (non-signée)	Pas de détection de dépassement de capacité	Rd ← Rs + Rt	R
subu	Rd, Rs, Rt	soustraction (non-signée)	Pas de détection de dépassement de capacité	Rd ← Rs − Rt	R
addi	Rt, Rs, I	addition (op. imm.)	Détection de dépassement de capacité	Rt ← Rs + I	I
addiu	Rt, Rs, I	addition (op. imm., non signée)	Pas de détection de dépassement de capacité	Rt ← Rs + I	I
Opérations de multiplication et de division					
mult	Rs, Rt	multiplication		Hi ← $pref_{32}(Rs \times Rt)$ Lo ← $suff_{32}(Rs \times Rt)$	R
multu	Rs, Rt	multiplication opérandes non signés		Hi ← $pref_{32}(Rs \times Rt)$ Lo ← $suff_{32}(Rs \times Rt)$	R
div	Rs, Rt	division		Hi ← Rs mod Rt Lo ← Rs/Rt	R
divu	Rs, Rt	division opérandes non signés		Hi ← Rs mod Rt Lo ← Rs/Rt	R

I.4. Directives supportées par l'assembleur MIPS

Les directives ne sont pas des instructions exécutables par la machine, mais permettent de donner des ordres à l'assembleur. Toutes les pseudo-instruction commencent par le caractère « . », ce qui permet de les différencier clairement des instructions.

a) Déclaration des sections

Six directives permettent de spécifier quelle section de la mémoire est concernée par les instructions, macro-instructions ou directives qui les suivent. Sur ces six directives, deux sont dynamiquement gérées à l'exécution : ce sont celles qui concernent la pile utilisateur, `stack`, et la pile système, `kstack`. Ceci signifie que l'assembleur gère quatre compteurs d'adresse indépendants correspondants aux quatre sections `text, data, ktext` et `kdata`.

Déclaration	Action	Description
`.text`	Passage dans la section text	Toutes les instructions et directives qui suivent concernent la section text dans le segment utilisateur.
`.data`	Passage dans la section data	Toutes les instructions et directives qui suivent concernent la section data dans le segment utilisateur.
`.stack`	Passage dans la section stack	Toutes les instructions et directives qui suivent concernent la section stack dans le segment utilisateur.
`.ktext`	Passage dans la section ktext	Toutes les instructions et directives qui suivent concernent la section ktext dans le segment noyau.
`.kdata`	Passage dans la section kdata	Toutes les instructions et directives qui suivent concernent la section kdata dans le segment noyau.
`.kstack`	Passage dans la section kstack	Toutes les instructions et directives qui suivent concernent la section stack dans le segment noyau.

Table 2 : Déclaration des sections

b) Déclaration et initialisation de variable

Les directives suivantes permettent de d'initialiser certaines zones dans les sections text ou data de la mémoire.

Déclaration	Exemples	Description
`.ascii chaîne, [chaîne,]`	`message:` `.ascii "Bonjour, Maître!\n\0"`	Cet opérateur place à partir de l'adresse du compteur d'adresse correspondant à la section active la suite de caractères entre guillemets. S'il y a plusieurs chaînes, elles sont placées à la suite. Cette chaîne peut contenir des séquences d'échappement du langage C, et doit être terminée par un zéro binaire si elle est utilisé avec un appel système.
`.asciiz chaîne, [chaîne,]`	`message:` `.asciiz "Bonjour, Maître!\n"`	Cet opérateur est strictement identique au précédent, la seule différence étant qu'il ajoute un zéro binaire à la fin de chaque chaîne.
`.byte expression, [expression,]`	`table:` `.byte 1, 2, 4, 8, 16, 32, 64, 32, 16, 8, 4, 2, 1`	La valeur de chacunes des expressions est tronquée à 8 bits, et les valeurs ainsi obtenues sont placées à des adresses successives de la section active.

`.align expression`	`.align 2`
	`.byte 12`
	`.align`
	`.byte 24`
		Cet opérateur aligne le compteur d'adresse sur une adresse telle que les n bits de poids faible soient à zéro. Cette opération est effectuée implicitement pour aligner correctement les instructions, demi-mots et mots.

.half expression, [expression,]	coordonnées: .half 0 , 1024	La valeur de chacunes des expressions est tronquée à 16 bits, et les valeurs ainsi obtenues sont placées dans des adresses successives de la section active.
.word expression, [expression,]	entiers: .word -1, -1000, -100000, 1	La valeur de chaque expression est placée dans des adresses successives de la section active.
.space expression	nuls: .space 1024 #initialise 1 kilo de mémoire à zéro	Un espace de taille expression octets est réservé à partir de l'adresse courante de la section active.

Table3 : Déclaration et initialisation des variables

I.5. Appels systèmes

Pour exécuter certaines fonctions système, typiquement les entrées/sorties (lire ou écrire un nombre, ou un caractère), il faut utiliser des appels système.

Par convention, le numéro de l'appel système est contenu dans le registre $2, et son unique argument dans le registre $4. Cinq appels système sont actuellement supportés dans l'environnement de simulation QtSpim.

Fonction	Description	Exemples

écrire un entier	Il faut mettre l'entier à écrire dans le registre $4 et exécuter l'appel système numéro 1.	`li $4, 1234567 ; met 1234567 dans l'argument`
		`ori $2, $0, 1 ; code de 'print_integer'`
		`syscall ; affiche 1234567`

lire un entier	La valeur de retour d'une fonction (système ou autre) est positionnée dans le registre $2. Ainsi, lire un entier consiste à exécuter l'appel système numéro 5 et récupérer le résultat dans le registre$2.	`ori $2, $0, 5 ; code de 'read_integer'` `syscall ; $2 contient ce qui a été lu`

écrire une chaîne	Une chaîne de caractères étant identifiée par un pointeur, il faut passer ce pointeur à l'appel système numéro 4 pour l'afficher.	`.data`
		`str: .asciiz "Chaîne à afficher\n"`
		`.text`
		`la $4, str ; charge le pointeur dans $4`
		`ori $2, $0, 4 ; code de 'print_string'`
		`syscall ; affiche la chaine pointée`

lire une chaîne	Pour lire une chaîne de caractères, il faut un pointeur et une longueur maximum. Il faut passer ce pointeur, dans $4, et cette longueur, dans $5, et exécuter l'appel système numéro 8. Le résultat sera mis dans l'espace pointé.	`.data`
		`read_str:`
		`.space 256`
		`.text`
		`la $4, read_str ; charge le pointeur dans $4`
		`ori $5, $0, 255 ; charge longueur max dans $5`
		`ori $2, $0, 8 ; code de 'read_string'`
		`syscall ; lit la chaîne dans l'espace`
		`; pointé par $4`

Quitter	L'appel système numéro 10 effectue l'exit du programme au sens du langage C.	`ori $2, $0, 10 ; indique l'appel à exit` `syscall ; quitte pour de bon!`

Table 4 : Appels système

I.6. Appels de fonctions

L'exécution de fonctions nécessite une pile en mémoire. Cette pile correspond à la section stack. L'utilisation de cette pile fait l'objet de conventions qui doivent être respectées par la fonction appelée et par la fonction appelante.

- La pile s'étend vers les adresses décroissantes ;
- Le pointeur de pile pointe toujours sur la dernière case occupée dans la pile. Ceci signifie que toutes les cases d'adresse inférieure au pointeur de pile sont libres ;

- Le R3000 ne possède pas d'instructions spécifiques à la gestion de la pile. On utilise les instructions lw et sw pour y accéder.
- Les appels de fonction utilisent un pointeur particulier, appelé pointeur de pile. Ce pointeur est stocké conventionnellement dans le registre $29. On le désigne aussi par la notation $sp. La valeur de retour d'une fonction est conventionnellement présente dans le registre $2.
- Par ailleurs, l'architecture du processeur MIPS R3000 impose l'utilisation du registre $31 pour stocker l'adresse de retour lors d'un appel de fonction (instructions de type jal).

À chaque appel de fonction est associée une zone dans la pile constituant le « contexte d'exécution » de la fonction. Dans le cas des fonctions récursives, une même fonction peut être appelée plusieurs fois et possèdera donc plusieurs contextes d'exécution dans la pile. Lors de l'entrée dans une fonction, les registres $5 à $26 sont disponibles pour tout calcul dans cette fonction.

II. Prise en main QtSpim

QtSpim est un logiciel de simulation de l'exécution de programmes assembleur MIPS 32 bits, disponible sur l'adresse suivante : http://spimsimulator.sourceforge.net/. Cette section est une traduction française d'une partie de menu d'aide du logiciel QtSpim.

II.1. Fenêtre principale

Lorsque QtSpim démarre, une fenêtre s'ouvre qui ressemble à celui ci-dessous :

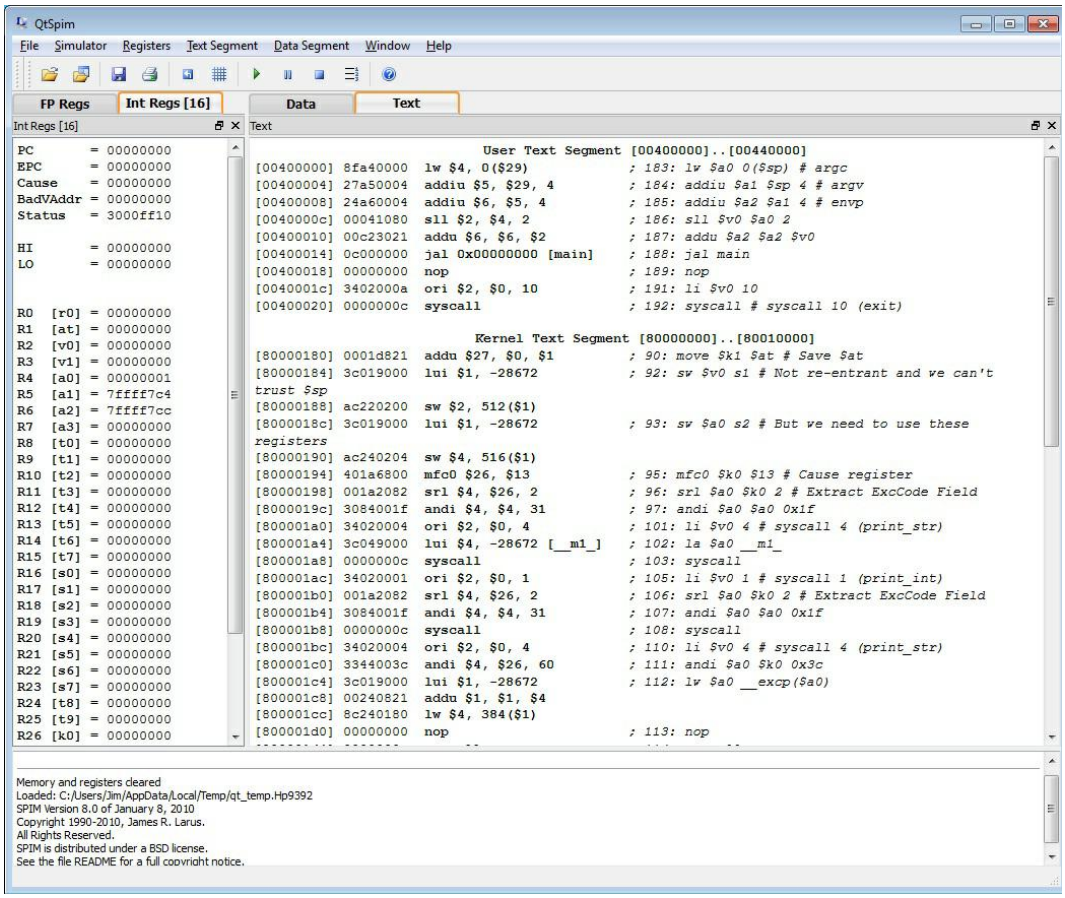

Figure 2 : Fenêtre principale QtSpim.

La fenêtre principale de QtSpim comporte trois parties:

1. Le volet étroit sur la gauche peut afficher les registres des entiers ou les registres à virgule flottante. Sélectionnez l'ensemble de registres en cliquant sur l'onglet en haut de la fenêtre (FP Regs, Int Regs [16]).
2. Le volet le plus large sur la droite peut afficher le segment de texte, qui contient des instructions assembleurs, et les segments de données qui contient les données d'entrée du programme assembleur. Choisissez entre le texte et les données en cliquant sur l'onglet en haut de la fenêtre.
3. Le petit volet en bas est là où QtSpim écrit ses messages.

QtSpim ouvre aussi une autre fenêtre appelée console qui affiche la sortie de votre programme.

II.2. Chargement d'un programme

Votre programme doit être stocké dans un fichier. Les Fichiers Assembleur ont généralement l'extension **« .s »**, par exemple **« Exercice1.s »**. Pour charger un fichier, allez dans le menu **« Fichier »** et sélectionnez **« Load File »**. L'écran va changer car le fichier est chargé, pour montrer les instructions et les données dans votre programme.

Une autre commande très utile sur les fichiers est « **Réinitialize and Load File ».** Il efface d'abord tous les changements apportés par un programme, y compris la suppression de la totalité de ses instructions, puis recharge le dernier fichier que l'utilisateur veut charger. Cette commande est utile lors du débogage d'un programme. Vous pouvez modifier votre programme et tester rapidement dans un nouvel ordinateur sans fermer et de redémarrer QtSpim.

II.3. Exécution d'un programme

Pour démarrer un programme en cours d'exécution après que vous l'avez chargé, allez dans le menu **« Simulator »** et cliquez sur **« Run/Continue »**. Votre programme sera exécuté jusqu'à ce qu'il se termine ou jusqu'à ce qu'une erreur se produise. De toute façon, vous verrez les changements que votre programme réalise sur les registres MIPS et de la mémoire, et la sortie de votre programme apparaîtra dans la fenêtre de la console.

Si votre programme ne fonctionne pas correctement, il ya plusieurs choses que vous pouvez faire. Le plus simple est l'exécution de votre programme instruction par instruction, ce qui vous permet de voir les changements apportés par chaque instruction. Cette commande est également dans le menu Simulator et est nommé « **Single Step** ».

Cependant, parfois, vous devez exécuter votre programme pendant un certain temps avant qu'une erreur se produise, et l'exécution de programme pas à pas serait trop lent. QtSpim vous permet de définir un point d'arrêt **«BreakPoint»** à une instruction spécifique, se qui arrête l'exécution du programme avant que l'instruction est exécutée. Donc, si vous pensez que votre problème est dans une fonction spécifique dans votre programme, définissez un point d'arrêt à la première instruction dans la fonction, et QtSpim arrêtera chaque fois que la fonction est invoquée.

Vous définissez un point d'arrêt par un clic droit sur l'instruction où vous voulez arrêter, et en sélectionnant Set **« Breakpoint »**. Lorsque vous avez terminé avec le point d'arrêt, vous pouvez le supprimer en sélectionnant Effacer le point d'arrêt à la place.

Si vous souhaitez arrêter votre programme pendant son exécution, allez dans le menu de Simulator et cliquez sur Pause. Cette commande arrête votre programme, laissez vous regardez autour, et de poursuivre l'exécution si vous voulez. Si vous ne voulez pas continuer à exécuter, cliquez sur Arrêter la place.

Lorsque QtSpim s'arrête, soit à cause d'une erreur dans votre programme, un point d'arrêt, après avoir cliqué sur Pause, ou après le pas à pas, vous pouvez poursuivre le programme en cours d'exécution en cliquant sur

« Run/Continue » (ou vous pouvez continuer le pas à pas en cliquant « SingleStep »). Si vous cliquez sur Arrêter, au lieu de Pause, puis en cliquant sur « Run/Continue » , le programme va se redémarrer depuis le début, au lieu de continuer d'où il est arrêté.

II.4. Option d'affichage

Les trois autres menus – « Registres », « Text segment », et « Data segment » - contrôlent les affichages de QtSpim. Par exemple, le menu Register contrôle la façon dont QtSpim affiche le contenu des registres, soit en binaire, base 8 (octal), base 10 (décimal), ou la base 16 (hexadécimal). Il est souvent très pratique pour faire basculer entre ces représentations pour comprendre vos données.

Ces menus vous permettent également de désactiver l'affichage des différentes parties du simulateur, ce qui peut aider à réduire l'encombrement sur l'écran et laissez vous vous concentrez sur les parties du programme ou des données qui comptent vraiment.

II.5. Modification des registres et de la mémoire

Vous pouvez modifier le contenu d'un registre ou d'un emplacement de mémoire par un clic droit dessus et en sélectionnant « **Change Register Contents »** ou «**Change Memory Contents »**, respectivement.

III. Documentations utiles

[1] Processeur MIPS R3000, langage d'assemblage (Version 1.4), 2001, UNIVERSITÉ PIERRE ET MARIE CURIE. ftp://asim.lip6.fr/pub/mips/mips.asm.pdf

[2] Processeur MIPS R3000, Architecture externe R3000 (Version 2.2), 1999, UNIVERSITÉ PIERRE ET MARIE CURIE. ftp://asim.lip6.fr/pub/mips/mips.externe.pdf

[3] Pirouz Bazargan, François Dromard, Alain Greiner, Processeur MIPS R3000, Architecture externe R3000 (Version 2.5), 2002, UNIVERSITÉ PIERRE ET MARIE CURIE. ftp://asim.lip6.fr/pub/mips/mips.interne.pdf