LU2IN014 : Machine et Représentation

Cours 3 : Programmes Assembleur Mips

Quentin Meunier

Stockage des informations

Unités de stockage et localisation

- À tout instant, les données et les instructions du programme en cours d'exécution sont stockées en mémoire
- Une partie de ces informations est à un instant donné dans le processeur : l'instruction en cours d'exécution + éventuellement une (petite) partie des données manipulées par le programme
- Dans le processeur, toutes les données sont temporaires et stockées en registre
- Transfert des informations entre la mémoire et le processeur via le bus

Les registres d'un processeur

Les registres d'un processeur

- L'architecture du processeur définit le nombre, la taille et le nom des registres du processeur
- Certains peuvent être manipulés explicitement par le programmeur via des instructions

Manipulation des registres

- Affectation explicite d'une valeur à un registre par le biais d'instructions
  - Exemple en Mips : \$5 $\leftarrow$ 2, \$6 $\leftarrow$ \$7 + \$8
- Affectation implicite de certains registres lors de l'exécution d'instructions particulières (exemples dans la suite)

Les registres du Mips

Les registres du Mips

- Les registres du Mips font 32 bits
- Les **registres généraux** : 32 registres de $0 à $31 ; ce sont des registres de travail, accessibles directement par le logiciel (code assembleur), mais chacun de ces registres a une utilisation prédéfinie
- **PC** (Programme Counter) contient l'adresse de l'instruction en cours d'exécution (ou la suivante) ; modifié après l'exécution de chaque instruction
- **IR** (Instruction Register) contient l'instruction en cours de traitement
- **HI**/**LO** (High/Low) sont les registres contenant le résultat d'opérations de multiplication ou de division
- Autres registres non utilisés dans ce cours : EPC, BAR, SR, ...

Les registres généraux du Mips

Sémantique et utilisation (1)

- $0 (zero) : contient toujours la valeur 0 ; une écriture dans ce registre ne modifie pas son contenu
- $1 (at) : registre réservé à l'assembleur (programme qui génère le binaire)
- $2 (v0) : résultat d'un appel de fonction + numéro d'un appel système
- $3 (v1) : résultat d'un appel de fonction
- $4 à $7 (a0 à a3) : utilisés pour le passage d'arguments lors des appels de fonction ou les appels système (non persistants)
- $8 à $15 (t0 à t7), $24 (t8) et $25 (t9) : registres non persistants (utilisation libre)
- $16 à $23 (s0 à s7) : registres persistants (utilisation libre)

Les registres généraux du Mips

Sémantique et utilisation (2)

- $26 et $27 (k0, k1) : registres réservés au système d'exploitation
- $28 (gp) : contient l'adresse de base des variables globales ("global pointer")
- $29 (sp) : contient l'adresse du sommet de la pile ("stack pointer", ou pointeur de pile)
- $30 (fp) : contient l'adresse de base de la fenêtre courante ("frame pointeur")
- $31 (ra) : adresse de retour dans un appel de fonction

Remarques

- En Mips, un registre peut être noté avec un `R`, `r` ou `$` (`R7 = r7 = $7`)
- Les registres en rouge ne seront pas utilisés dans le cadre de ce cours

Instructions, données et programme

Programme

- Un programme définit un traitement à appliquer à des données
- Il comporte deux parties :
  - Les données
  - Le traitement qui est une suite d'opérations sur les données

Représentation en machine

- Les données sont représentées en binaire avec la représentation associée à leur nature (entiers relatifs, caractères, ...) et stockées en mémoire
- Le traitement à réaliser est traduit en instructions compréhensibles par le processeur cible : ces instructions sont "en langage machine"
- Elles sont codées en binaire et stockées en mémoire

Jeu d'instructions

Vue externe d'un processeur

- La vue externe d'un processeur peut être définie par l'ensemble des instructions qu'il est capable de traiter : c'est son jeu d'instructions

Jeu d'instructions : composition

- Le jeu d'instructions d'un processeur (ISA) est défini par :
  - L'ensemble des instructions qu'il peut effectuer
  - Le codage de ces instructions en binaire
- Le jeu d'instructions et l'architecture interne du processeur sont définis conjointement

Qu'est ce qu'une instruction ?

Définition d'une instruction

- C'est une commande donnée au processeur qui définit :
  - Le traitement à effectuer maintenant
  - Quelle sera la prochaine instruction à exécuter
- Les instructions sont codées en binaire et stockées en mémoire les unes à la suite des autres
- En Mips, toutes les instructions font **1 mot** (32 bits)

Qu'est ce qu'une instruction ?

Traitement à effectuer

- L'opération mise en jeu : addition, opération logique, opération mémoire, ...
- Les opérandes sur lesquelles elle porte : le ou les opérandes sources, l'opérande destination s'il y en a un

Prochaine instruction à exécuter

- Séquentiel par défaut (implicitement) : la prochaine instruction à exécuter est celle implantée en mémoire à la suite de l'instruction courante
- Explicite dans des instructions spécifiques (instructions dites de saut ou de branchement)

Exécution d'une instruction

Exécution d'une instruction

- L'exécution d'une instruction nécessite plusieurs étapes :
  - Lire l'instruction en mémoire : transfert mémoire
  - Décoder l'instruction : détermination de quelle opération, puis de quelles opérandes
  - Exécuter l'instruction : réalisation du traitement correspondant à l'instruction (qui peut elle-même nécessiter plusieurs étapes, exemple : lecture)
  - Déterminer l'adresse de la prochaine instruction (implicite ou explicite)

Opérandes dans une instruction

Opérandes

- Les opérandes mis en jeu dans les instructions sont soit des **immédiats**, soit des **registres**
- Les immédiats sont des valeurs **codées dans l'instruction** (ex : -1, 0xFFFF, 3, ...)
- Les registres de travail $0, $1, ..., $31 sont spécifiables dans l'instruction
- Les autres registres manipulables (PC, LO, HI) sont implicites avec l'opération
- PC est manipulé via les instructions de rupture de séquence uniquement
- Toutes les valeurs manipulées par le processeur MIPS sont sur 32 bits (taille des registres)

Syntaxe assembleur des instructions

Instruction assembleur de la forme : CodeOperation Operandes

- **CodeOperation** est un mnémonique : `addu`, `andi`, `lw`, `sw`, `j`, `beq`
- **Operandes** est une suite vide ou non d'opérandes :
  - `OpReg, OpReg, OpReg`
  - `OpReg, OpReg, OpImm`
  - `OpReg, OpReg`
  - `OpReg, OpImm(OpReg)`
  - `OpReg, OpReg, OpLabel`
  - `OpReg`
  - `OpLabel`

Syntaxe assembleur des instructions

Exemples d'instructions avec différents formats d'opérandes

- OpReg, OpReg, OpReg : `addu $8, $9, $10`
- OpReg, OpReg, OpImm : `ori $8, $8, 0xABCF`
- OpReg, OpReg : `mult $16, $17`
- OpReg, OpImm(OpReg) : `lw $16, 8($20)`
- OpReg, OpReg, OpLabel : `beq $10, $11, loop`
- OpReg : `jr $31`
- OpLabel : `j ma_fonction`

Les 4 différentes classes d'instructions

Instructions arithmétiques et logiques

- Ces instructions utilisent l'ALU pour réaliser un calcul sur des données

Instructions de transfert mémoire (cours 4)

- Ces instructions lisent ou écrivent des données en mémoire

Instructions de rupture de séquence

- Elle permettent de spécifier quelle sera la prochaine instruction à exécuter

Instruction d'appel système

- Elle demande un service au système

Instructions arithmétiques et logiques

Principe

- Ces instructions utilisent l'ALU pour réaliser un calcul sur des données
  - Le résultat est toujours stocké dans un registre
  - Les opérandes sources sont des registres et/ou des constantes entières codées sur 16 bits (et étendues sur 32 bits à l'exécution)

Exemple d'instructions arithmétiques

- Addition
  - Entre 2 registres : `addu $8, $9, $10`
  - Entre un registre et un immédiat : `addiu $8, $9, 1`
- Multiplication `mult` :
  - Entre 2 registres : `mult $16, $17`

Instructions arithmétiques et logiques

Opérations logiques

- OU logique :
  - Entre 2 registres : `or $12, $14, $13`
  - Entre un registre et un immédiat : `ori $12, $13, 0x00F0`
- ET logique :
  - Entre 2 registres : `and $12, $14, $13`
  - Entre un registre et un immédiat : `andi $12, $13, 0x00F0`
- OU exclusif entre 2 registres :
  - Entre 2 registres : `xor $11, $11, $11`

Affectation de registre

- Mettre une valeur sur les 16 bits de poids fort : `lui $16, 0xABCD`
- Mettre le contenu de HI dans $4 : `mfhi $4`
- Mettre le contenu de LO dans $4 : `mflo $4`

Instructions arithmétiques et logiques

Opérations de décalage

- À gauche
  - Avec deux operandes registres : `sllv $2, $4, $3`
  - Avec un operande registre et un immédiat : `sll $2, $4, 16`
- À droite 'signé' dit arithmétique
  - Avec deux operandes registres : `srav`
  - Avec un operande registre et un immédiat : `sra`
- À droite 'non signé' dit logique
  - Avec deux operandes registres : `srlv`
  - Avec un operande registre et un immédiat : `srl`

Rupture de séquence

Effet

- Ces instructions indiquent quelle est l'adresse de la prochaine instruction
- Elles modifient la valeur du registre PC

Deux types de saut

- Les **sauts inconditionnels** qui ont toujours lieu, appelés `jump` (ou **sauts**)
  - `j etiquette_inst`, `jal ma_fonction`, `jr $31`
- Les **sauts conditionnels** (ou **branchements**), qui sont réalisés si et seulement une condition est vérifiée
- Cette condition est spécifiée dans le code de l'instruction :
  - Égalité ou différence de deux registres : `beq $0, $4, etiquette_inst`, `bne $0, $4, etiquette_inst`,
  - Comparaison d'un registre à 0 : `bltz $4, etiquette_inst`

Instruction d'appel système

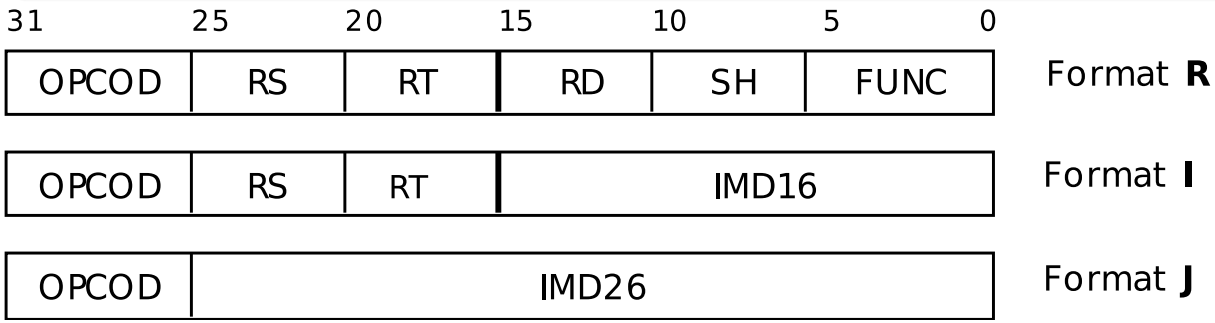
Instruction d'appel système

- Elle correspond à une demande de service fourni par le système :
  - Affichage sur l'écran d'un caractère, d'un entier, d'une chaine de caractères
  - Lecture depuis le clavier d'un entier, d'une chaine de caractères
- C'est l'instruction `syscall` qui réalise cette demande de service.
- Le système offre un certain nombre de services, chacun a un numéro qu'il faut placer dans le registre $2 avant d'exécuter l'instruction `syscall`.

Codage des instructions MIPS

Formats de codage MIPS

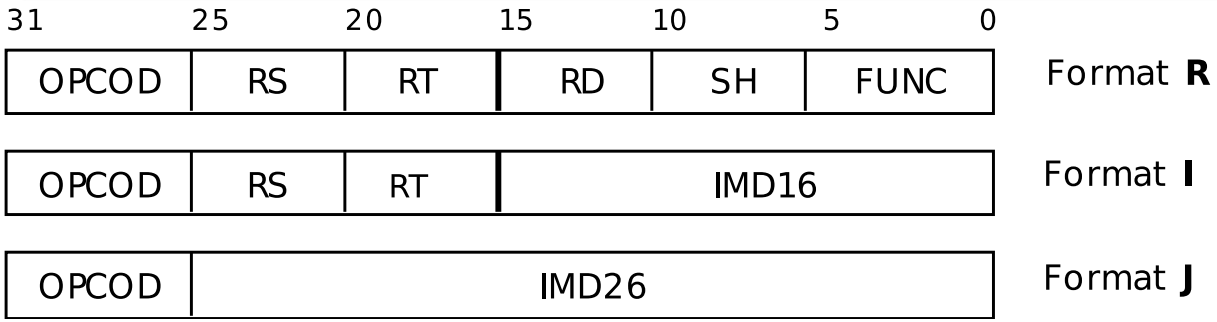
- Toutes les instructions sont codées sur 32 bits
- 3 formats de codage appelés R, I, J
  - Le format R pour les instructions arithmétiques et logiques avec 2 registres sources et pour toutes les instructions de décalage
  - Le format I pour toutes les instructions arithmétiques et logiques avec 1 opérande source immédiat (sauf décalage), les accès mémoire et les sauts conditionnels
  - Le format J pour les instructions de saut inconditionnel avec adressage direct



Codage des instructions

Format de codage et champ du code opération

- Chaque format comporte plusieurs champs
- Un champ contient une information (code opération, code ou valeur opérande, ...)
- Tous les formats ont un champ OPCOD de 6 bits (bits 31 à 26)
  - L'opcode (ou code opération) code l'opération correspondant à l'instruction
  - Sa valeur est donnée par une table qui fait la correspondance entre mnémonique et un codage sur 6 bits
  - Ce champ permet au processeur de savoir comment décoder les 26 bits restants et l'opération à effectuer



Code opération

Exemples

- Le mnémonique `addi` a pour code opération 0b001000
- Le code 0b100011 correspond au mnémonique `lw`

DECODAGE OPCOD

INS 31 : 29 \ INS 28 : 26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

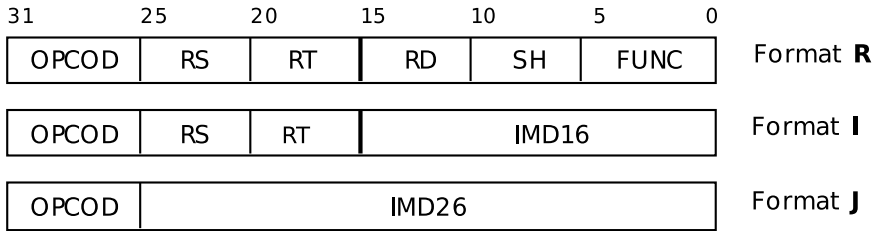
Décodage d'une instruction

Exemple

- Soit le mot 0x2062ABCD = 0b0010 0000 0110 0010 1010 1011 1100 1101
- OPCOD =
- Format =
- Champs restants =

DECODAGE OPCOD

INS 31:29 \ INS 28:26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								



Code opération versus code spécial

Opcode SPECIAL

- L'opcode SPECIAL correspond au format R et à une famille d'opérations précisée dans le champ FUNC (bits 5 à 0)
- L'opération est alors donnée par une seconde table, celle qui donne le codage du champ FUNC
- Par exemple si INS[31 :25] = 0b000000 et INS[5 :0] = 0b100101, l'opération est un `or`

DECODAGE OPCOD

INS 31 : 29 \ INS 28 : 26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

OPCOD = SPECIAL

INS 5 : 3 \ INS 2 : 0	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

Décodage d'une instruction

Exemple

- Soit 0x00641020 = 0b0000 0000 0110 0100 0001 0000 0010 0000
- OPCOD =
- Format =
- Champs restants =

DECODAGE OPCOD

INS 31 : 29 \ INS 28 : 26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

OPCOD = SPECIAL

INS 5 : 3 \ INS 2 : 0	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

Codage d'une instruction

Exemple

- Instruction addiu $15, $21, -5
- OPCOD =
- Format (voir sur le mémento) =
- Champs restants =
- Codage binaire =
- Codage en hexadécimal =

DECODAGE OPCOD

INS 31 : 29 / INS 28 : 26

	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

OPCOD = SPECIAL

INS 5 : 3 / INS 2 : 0

	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

Codage d'une instruction

Exemple

- Instruction or $4, $2, $1
- OPCOD =
- Format (voir sur le mémento) =
- Champs restants =
- Codage binaire =
- Codage en hexadécimal =

DECODAGE OPCOD

INS 31 : 29 \ INS 28 : 26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

OPCOD = SPECIAL

INS 5 : 3 \ INS 2 : 0	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

Uniformité de la mémoire

Mémoire

- Elle est modélisée comme un tableau de mots de 4 octets adressable en octets : l'adresse 0x101 se situe 1 octet plus loin que l'adresse 0x100
- Elle stocke sous forme de suite d'octets tout ce qui est nécessaire à l'exécution d'un programme : les données et les instructions

Uniformité $\Rightarrow$ universalité

- Cette uniformité confère à l'ordinateur son universalité : les traitements exécutés par le processeur sont précablés/finis mais l'agencement des traitements possibles est infini

- Les données et les instructions sont stockées sur le même support : rien ne différencie *a priori* un mot contenant une donnée d'un mot contenant une instruction si ce n'est l'interprétation que le processeur en fait

Uniformité de la mémoire : exemple

Un programme peut être utilisé pour obtenir de nouveaux programmes

- Rédaction de programme source
  - Donnée = programme texte (codage ASCII)
  - Programme = l'éditeur de texte
- Compilation d'un programme source en un binaire exécutable :
  - Donnée d'entrée = programme source (texte codage ASCII)
  - Programme = compilateur
  - Donnée de sortie = programme binaire exécutable
- Exécution d'un programme binaire : pour exécuter un programme binaire, il faut d'abord le charger en mémoire. C'est le travail du *loader*. Une fois le programme chargé, le loader donne la main au programme binaire
  - Programme = loader, donnée = programme binaire exécutable
  - Programme = programme binaire, données = celles du programme binaire

Structuration de la mémoire

Besoin de structuration

- Un programme n'est pas le seul objet présent en mémoire : il y a les données et programmes de l'OS, éventuellement d'autres utilisateurs de la machine

Protections nécessaires

- Des informations du système vis-à-vis des programmes utilisateurs
- Des différents types d'informations au sein d'un même programme
- Des programmes vis-à-vis d'autres programmes

Protection du système vis-à-vis des programmes utilisateurs

Séparation des informations système/utilisateurs

- La mémoire est découpée en deux zones distinctes :
  - Zone réservée au système (OS) : accessible uniquement en mode superviseur/super utilisateur/kernel
  - Zone utilisateur : accès non restreint

En MIPS

- Ces deux zones sont distinguées par leurs adresses :
  - Espace utilisateur (ou *user*) : 0x00000000 $\rightarrow$ 0x7FFFFFFF
  - Espace système (ou noyau ou *kernel*) : 0x80000000 $\rightarrow$ 0xFFFFFFFF

Protection des informations au sein d'un programme

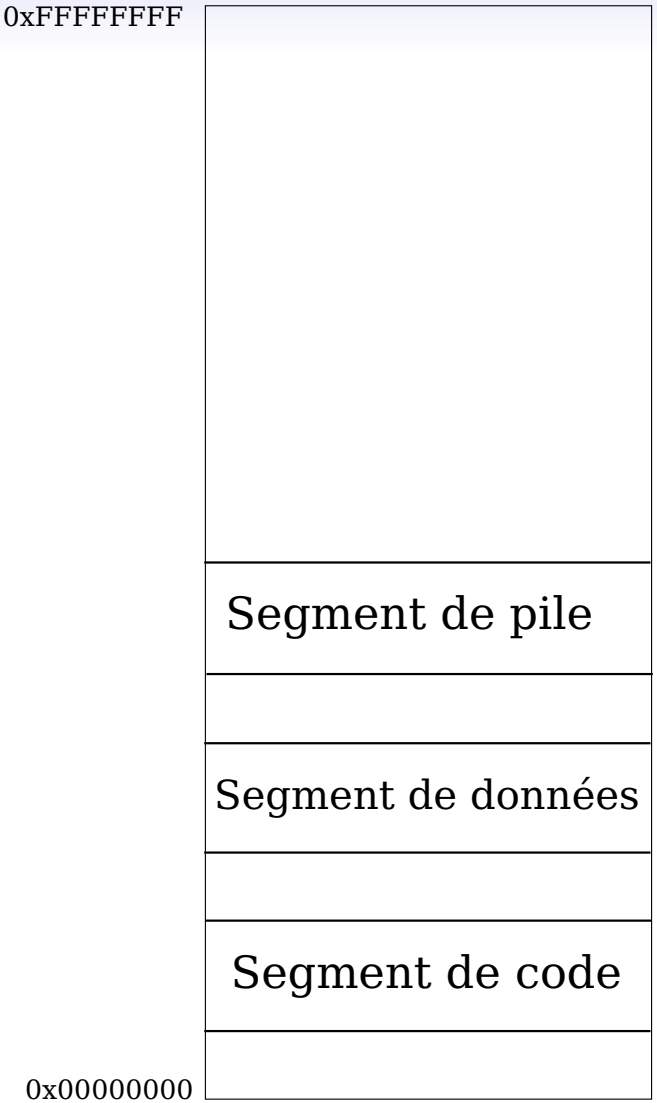
Segments

- Les différents types d'informations d'un programme sont regroupés en zones distinctes appelées segments
- Un segment est un espace d'adressage contigu muni d'une taille maximale dans lequel sont stockées les informations de même nature

Un programme possède au moins 3 segments

- Segment de code : contient les instructions codées en binaire selon l'ISA du processeur
- Segment de données : contient les données/variables globales, codées en binaire selon leur type
- Segment de pile : mémoire allouée à la demande au cours de l'exécution pour les contextes d'exécution des fonctions : paramètres d'appel, variables locales, etc.

Les 3 segments mémoire d'un programme



Un programme possède au moins 3 segments utilisateur

- Segment de code (instructions)
- Segment de données (données globales)
- Segment de pile (mémoire gérée par le programme pour les contextes d'exécution des fonctions)

Programme : les différents niveaux

Niveaux d'abstraction pour l'écriture d'un programme

- Il existe différents niveaux de programmation qui correspondent aux différents niveaux d'abstraction des traitements à exécuter
  - Programme de haut niveau
  - Programme en langage d'assemblage/assembleur
  - Programme binaire

Composition des niveaux de programmes

Haut niveau

- Instructions écrites dans un langage source indépendant de l'ISA cible
- Definition de structures de données
- Définition de variables nommées utilisables dans les instructions
- Structuration des traitements, gestion d'erreurs

Assembleur

- Suite d'instructions assembleur
- Présence d'étiquettes pour désigner les adresses (données ou instructions)

Binaire

- Suite d'instructions en langage machine
- Traduction avec les instructions assembleur quasiment 1 pour 1

Traduction de programmes

Programme haut niveau → programme assembleur

- La traduction d'un langage de haut niveau vers un langage d'assemblage est réalisée par un compilateur. Les instructions de haut niveau sont traduites en une suite d'instructions assembleur. Les registres sont utilisés pour réaliser les traitements.

Programme assembleur → programme binaire exécutable

- La traduction assembleur vers binaire ou assemblage réalise la traduction binaire des instructions (programme qui fait ça = assembleur). Elle nécessite 2 phases :
  - Traduction en binaire et assignation des adresses d'implantation
  - Résolution des adresses : les étiquettes dans les instructions sont converties en adresses absolues ou relatives (exemple : le champ des sauts conditionnels/inconditionnels)

Exemple ASM

Assembleur (ASM)	**Binaire**
Instructions assembleur et présence d'étiquettes	Instructions converties en langage machine
`beq $5, $0, fin`	`0x10A00003`
`etiq: andi $4, $4,0x0FF0`	`0x30840FF0`
`addiu $4, $4, -1`	`0x2084FFFF`
`j etiq`	`0x08100001`
`fin: add $8, $4, $6`	`0x00864020`

Etiquettes

- Une étiquette est de la forme "`nom_etiq:`"
- Une étiquette désigne l'adresse de ce qui suit : donnée (variable) ou instruction
- Son nom peut ensuite être utilisé dans certaines instructions

Programmation assembleur

Pourquoi savoir écrire du code assembleur ?

- Si on doit intervenir dans le développement d'un compilateur
- Certaines fonctions très utilisées que l'on veut très optimisées
- Certaines parties d'un OS doivent être écrites en assembleur

Qu'est-ce qu'on va faire dans ce cours ?

- Écriture de programmes directement en assembleur
- Allocation et initialisation des données
- Description des traitements à réaliser avec les instructions du jeu d'instructions du processeur cible (ici : Mips)
- Conventions liées au processeur cible

Structure d'un programme assembleur Mips

- En MIPS, tout programme assembleur est constitué d'au moins 2 sections : la section de données et la section de code

Section de code : directive `.text`

- La directive `.text` désigne la section de code
- Elle spécifie que ce qui suit sera implanté dans le segment de code
- Les instructions doivent se trouver dans cette section/après cette directive dans un programme assembleur

Section de données : directive `.data`

- La directive `.data` désigne la section de données
- Elle spécifie que ce qui suit correspond aux données globales : celles-ci seront implantées dans le segment de données
- Les données globales doivent être allouées, et initialisées si besoin, dans cette section du programme

Déclaration des variables globales

Variables globales

- La syntaxe générale pour déclarer une variable globale est :
  `nom: directive [init]`
- Toutes les variables globales sont déclarées dans la section .data
- Les directives que l'on utilisera sont `.word`, `.byte`, `.half`, `.space`, `.ascii`, `.asciiz`

Déclaration des variables globales

Utilisation des directives

- `n: .word 5` : réserve un mot et l'initialise à la valeur 5
- `n: .byte 1` : réserve un octet et l'initialise à la valeur 1
- `n: .half 12` : réserve un demi-mot et l'initialise à la valeur 12
- `buf: .space 20` : réserve 20 octets non initialisés
- `s: .asciiz "abc"` : alloue et initialise 4 octets pour la chaine de caractère "abc", y compris le caractère de fin de chaine
- `s: .ascii "abc"` : alloue et initialise 3 octets pour la chaine de caractère "abc", sans le caractère de fin de chaine (pas très utile pour nous)

Assemblage et chargement d'un programme en mémoire

- Une fois un programme assembleur écrit, il faut l'assembler afin de créer un programme binaire exécutable
- Pour pouvoir exécuter un programme, il faut d'abord le charger en mémoire $\Rightarrow$ Fait par un programme du système appelé *loader*
- La section de code est rangée dans le segment de code et la section de données est rangée dans le segment de données
- La première instruction de la section `.text` est implantée à l'adresse 0x00400000 et les suivantes le sont consécutivement en mémoire dans le segment de code, selon leur ordre d'apparition dans le fichier assembleur
- La première donnée de la section `.data` est implantée à l'adresse 0x10010000 et les données déclarées ensuite sont allouées en séquence dans le segment de données

Structure d'un programme assembleur

STRUCTURE D'UN FICHIER ASSEMBLEUR

mon_fichier.s

.data

Declaration/allocation des donnees globales

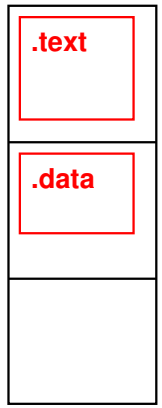
Mis dans le segment de donnees

directives pour les declarations
.word
.asciiz **(voir le memento)**
.space

.text

suite d'instructions assembleur decrivant le traitement a realiser

mon_fichier.s **assemblage** → **binaire executable** **execution** →



Segment de code : les instructions sont implantees en sequence. La premiere instruction du programme est la premiere de la section .text , son adresse est 0x0400 0000

Segment de donnees : implantation en sequence des donnees ; les adresses sont assignees aux donnees dans l'ordre de leur declaration. La premiere donnee est implantee a 0x1001000

Lors du lancement d'un programme, le PC contient l'adresse de la premiere inst (c'est le point d'entree).

Exécution et terminaison d'un programme assembleur

Déroulement

- La première adresse de la section de code correspond au point d'entrée du programme
- Lors du lancement d'un programme, l'adresse du point d'entrée du programme est mise par le *loader* dans le registre PC : la première instruction est alors lue en mémoire, puis exécutée
- L'exécution continue ensuite séquentiellement, avec parfois des sauts quand il y a une instruction de saut ou de branchement
- Le programme s'arrête avec un appel système demandant la terminaison du programme

Appels système

Quoi et comment ?

- Un appel système est une demande de service fourni par le système
- En MIPS, chaque service a un numéro
- Pour utiliser un service, il faut mettre le numéro correspondant dans le registre $2 avant d'exécuter l'instruction `syscall`

Quelques appels système et leur numéro

- Terminaison d'un programme : numéro 10
- Affichage d'un entier : numéro 1 ; il faut aussi mettre l'entier à afficher dans le registre $4 avant l'appel
- Affichage d'une chaine de caractères : numéro 4 ; il faut aussi mettre dans le registre $4 l'adresse du premier caractère de la chaîne

Exemples de programmes faisant des appels système

3 exemples

- Programme qui affiche la valeur 2 et termine.
- Programme qui affiche "Hello world !" et termine.
- Programme qui lit un entier au clavier et affiche la valeur de l'entier + 1

Premier exemple

Afficher 2 et terminer

- Afficher 2 : service d'affichage d'un entier
- Afficher un entier : appel système numéro 1 + entier à afficher dans $4
- Terminer : service de terminaison de programme ⇒ appel système numéro 10

Second exemple

Afficher "Hello World !" et terminer

- Allouer et initialiser la chaine de caractères dans la section de données, avec la bonne directive
- Affichage d'une chaine de caractères : appel système numéro 4
- Chaine à afficher : mettre son adresse dans le registre $4 avant d'effectuer l'appel système
- Terminaison du programme : appel système numéro 10

Troisième exemple

Lire un entier au clavier et afficher sa valeur + 1

- Lire un entier au clavier : appel système numéro 5
- Entier lu dans le registre $2
- Affichage d'un entier : appel système numéro 1
- Terminaison du programme : appel système numéro 10