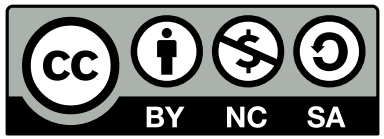
0011 - Assembleur RISC-V

Instructions, directives, entrées-sorties, branchements, appels de routines et bibliothèques

Jean Privat

Université du Québec à Montréal

INF2171 - Organisation des ordinateurs et assembleur
v241



Rappel

- On sait faire des additions (`add`, `sub`, etc.)
- Mais qu'en est-il de faire de vrais programmes?

Plan

Éléments d'assembleur RISC-V

Éléments d'assembleur

Ressources

- Syntaxe RISC-V officielle
- Syntaxe GNU assembler

Fiche de référence (aide mémoire)

- Liste les instructions, directives et détails techniques
- On parle aussi de « *green card* »
  Étymologie: IBM/360 Reference data card, ~1970
- Aide mémoire du cours
- Open RISC-V Reference Card
- Disponible dans l'aide de RARS (menu ou bouton `Help`)

Vocabulaire

- **Assembleur** (*assembly*): **langage** de programmation
  - Synonymes: langage d'assemblage, *assembler language*
  - Spécification (syntaxe et sémantique)
- **Code assembleur**: **programme** écrit en langage d'assemblage
  - En INF2171 ce sera notre **code source**
  - (Habituellement, le **langage source** est de plus haut niveau)
- **Code machine**: programme écrit en langage machine
  - Synonymes discutables: code binaire, code exécutable
  - Spécifique à une architecture (ISA)
- **Assembleur** (*assembler*): programme d'assemblage
  - Transforme du code assembleur en code machine
  - On utilise juste « **outil** » pour être général
    Inclure les simulateurs, les désassembleurs, etc.

Assembleur: instructions et directives

Instructions

- Correspond à une instruction **machine**
  (ou plusieurs dans le cas de certaines pseudoinstructions)
- Les instructions machine sont représentez en binaire
  (4 octets en RISC-V)
- Exemple `addi x1, x0, 10` $\rightarrow$ `00 A0 00 93`

Directives

- Destinée à l'**assembleur**
- Commencent par un point
- Syntaxe et sémantiques particulières
- Génération ou non de binaire
- Exemple `.ascii "Hello"` $\rightarrow$ `48 65 6C 6C 6F`

Attention: spécificité des outils

L'assembleur n'est pas normalisé

- Il y a autant d'assembleurs que d'outils ou d'architectures
- Pour une même architecture: plusieurs outils
  - Il peut y avoir des différences importantes
  - Surtout au niveau des directives (destinées à l'outil)

RARS (*RISC-V Assembler and Runtime Simulator*)

- Outil utilisé dans le cours
- Assembleur relativement simple (mais limité)
- Relativement compatible avec GNU assembleur
  - GNU assembleur est beaucoup plus riche et complexe
  - Par défaut ce qu'on présente est compatible
- RTFM dans le doute

Code source assembleur

Formatage en colonne

- Étiquette
- Instruction ou directive
- Opérandes
- Commentaire

Chaque colonne est optionnelle

Objectif

- Lisibilité pour l'humain
- Prédominance des commentaires

Autrement, si c'est juste pour pas comprendre ce que l'on lit, autant coder directement en langage machine

Valeurs

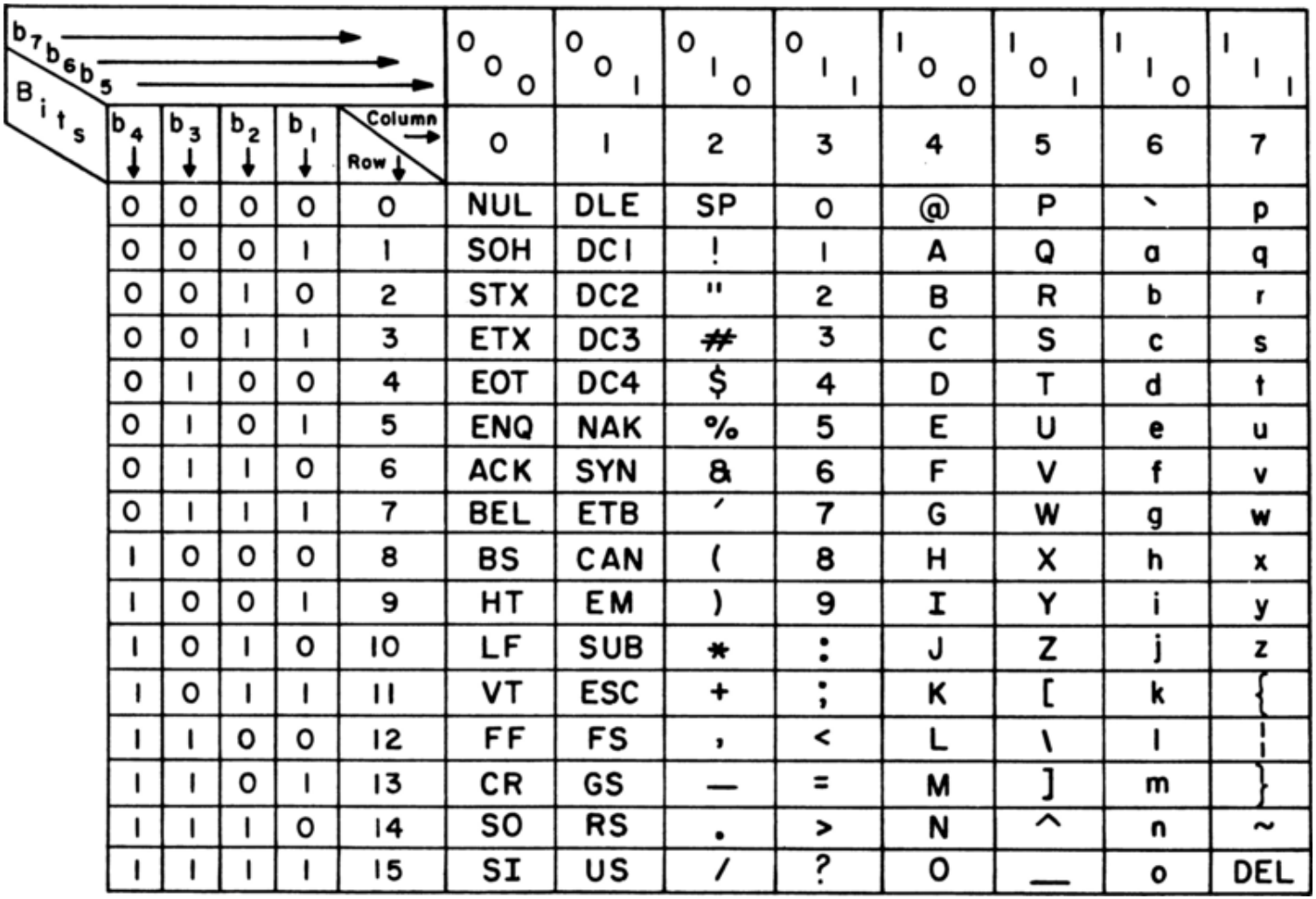
Constantes numériques entière littérales

- Décimaux : `42`
- Hexadécimaux : `0x2A`
- Caractères : `'*'` (code ASCII)

Note: les 3 exemples donne la même valeur numérique

Code ASCII, 1968

- Chaque caractère représenté par un code de 7 bits
- ASCII = *American Standard Code for Information Interchange*
- `man ascii`



$b_7 b_6 b_5$ →	000	001	010	011	100	101	110	111
Bits b_4 ↓	b_3 ↓	b_2 ↓	b_1 ↓	Column → / Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	¦
1	1	0	1	13	CR	GS	—	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

- Les 32 premier codes sont des caractères de contrôles
  Peu utilisés pour la plupart
- Les majuscules et minuscules diffèrent d'un seul bit
- Qu'est-ce qu'il manque?

Code ISO/IEC 8859-1 (dit latin-1), 1985

Famille ISO/IEC 8859

- Représenté par un code de 8 bits
- Inclut (presque) et étend l'ASCII
- 16 codes incompatibles pour diverse langues

ISO 8859-1

- Inclut des lettres accentuées des langues d'Europe de l'ouest
  Français, allemand, espagnol, islandais...
- Et des symboles divers
- `man latin1`

Latin-1

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
A_		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B_	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C_	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D_	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E_	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F_	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

- 80 à 9F sont non définis (et 00 à 1F)
- A0 est un espace insécable (*non breaking space*)
- AD est un trait d'union conditionnel (*soft hyphen*)
- Qu'est-ce qu'il manque ?

Unicode, 1991

The Unicode® Standard V15.0 – Core, 2022 (1060 pages)

Liste de caractères

- Numérotée. point de code unicode
  - U+0049 *Latin Capital Letter I* (1.1, June 1993) « I »
  - U+2665 *Black Heart Suit* (1.1, June 1993) « ♥ »
- 149 186 caractères dans la version 15
- Les 256 premiers caractères de l'Unicode sont ceux de latin-1

Caractéristiques de chaque caractère

- Nom, noms alternatifs, commentaires, etc.
- Type de caractère: lettre? majuscule? chiffre? ponctuation?
- ...

Règles et algorithmes: interopérabilité et affichage

- Écriture bidirectionnelle, compositions de caractères, conversions, etc.

Codages unicodes

Une demi-douzaine de codages possibles mais deux principaux

UTF-8, 1993

- Taille variable: 1, 2 3 ou 4 octets
- Compatible avec l'ASCII 7 bit
- Majoritaire (98% des pages web)

« I❤ça » → 49 e2 99 a5 c3 a7 c3 a0

UTF-16, 1996

- Taille variable: un ou deux paquets de deux octets
- Compliqué (raisons historiques) + problème de boutisme
- Principalement utilisé par Java et Windows

« I❤ça » → 49 00 65 26 e7 00 e9 00

Registres

- Noms et rôles et syntaxes spécifiques à l'architecture

RISC-V

- 32 registres généraux (`x0` à `x31`)
- Rôles et utilisations **conventionnées**: noms d'ABI
  - ABI = *Application Binary Interface*: on y reviendra
  - `a0` à `a7` (`x10` à `x17`): arguments
  - `t0` à `t6` (`x5` à `x7` et `x28` à `x31`): temporaires
  - `s0` à `s11` (`x8`, `x9` et `x18` à `x27`): sauvegardés
  - `zero` (`x0`): registre particulier, toujours à zéro
  - `ra` (`x1`): adresse de retour (*return address*)
  - `sp` (`x2`): pointeur de pile (*stack pointer*)
  - `gp` (`x3`): pointeur global (*global pointer*)
  - `tp` (`x4`): pointeur de thread (*thread pointer*)
  - `pc`: compteur ordinal, non accessible directement (*program counter*)

Autres registres

Registres flottants

- Pour opérations mathématiques flottantes
- RISC-V: `f0` à `f31`. Extension `F` (et supérieures)

Registres d'état et de contrôle

- Registres spécifiques: configuration, gestion des privilèges, des interruptions, surveillance, sondes, compteurs, etc.
- RISC-V: CSR (*control and status register*)
  - `cycle`, `time`, `instret`, etc. de base dans RV32I
  - `fcsr` (et `fflags`, `frm`) extension F
  - `utvec` (et 7 autres) extension U
  - Autres CSR: extensions + mode privilégié (volume II)

Registres vectoriels

- Opérations SIMD (*Single instruction, multiple data*)
- RISC-V: `v0` à `v31` de l'extension `V`

Symboles (identifiants)

- Alphanumérique
- Portée globale dans le fichier
- Associé à une valeur numérique **constante**

Important: Ce ne sont pas des variables

Définition de symboles

- Directive `.eqv symbole, valeur`
- Définition avant utilisation

Exemple

```
.eqv quarantedeux, 42
.eqv etoile, '*'

li s0, etoile
addi s1, s0, quarantedeux
```

Note: ici `etoile` et `quarantedeux` ont la même valeur

Étiquettes

- L'**étiquette** définie un **symbole** associé à une **adresse**
- Syntaxe « `etiquette:` » au début d'une ligne (ou seul sur une ligne)
- On y reviendra

Exemple

```
affichage: li a7, 1
# ...
hello: .ascii "Hello"
```

Clean Code

- Indentez **proprement**
- Nommez les symboles avec **sagesse**
- Utilisez les noms d'ABI des registres
  - Avec la bonne **sémantique**
- Commentez le **rôle** des registres
  - À leurs initialisations
- **Commentez** les groupes d'instructions
  - Commenter $\neq$ **traduire** ni **paraphraser** l'assembleur
- Utilisez la **bonne** représentation des constantes
  - Décimal, hexadécimal, caractère, symboles, etc.
- **Nommez** les constantes quand c'est pertinent (`.eqv`)
- **Évitez** le code trop long et compliqué
  - Rend le code **dur à lire** et cache sans doute des **bogues**
- **Évitez** les micro-optimisations
  - Rend le code **dur à lire** et cache sans doute des **bogues**
- Etc.

Rappel: représentation des données

Représentation des données

- Une donnée n'est qu'un paquet de bits
- La sémantique d'un paquet de bits n'est pas dans dans le paquet de bits
- Un paquet de bits peut être déclaré et stockée de façons diverses

Utilisation des données

- Une instruction s'attend à un certain type de donné
  mais n'importe quel tas de bits lui conviendra en général
- En fonction des instructions, un même paquet de bits peut avoir plusieurs sens différents

Entrées-sorties (du débutant)

Entrées-sorties

S'ouvrir au monde

- Modifier et calculer des registres dans un processeur
  - Pas très utile en soi
- On peut pouvoir communiquer avec le monde extérieur
  - Humains: utilisateurs (et développeurs qui testent)
  - Machines: capteurs, moteurs, disques, réseau, etc.

Problème

L'environnement d'exécution 🌎 détermine les entrées-sorties disponibles et leur modalités d'utilisation

- Beaucoup de variabilité
- Nombreux détails
- Nécessite des concepts qu'on a pas encore vus
- RARS est assez limité :(

Entrées-sorties: 2 approches

Accès direct

- Le périphérique s'expose a certaines adresses mémoire
  - Rappel: les périphériques sont connectés au bus
- Disponible de base dans certains simulateurs et systèmes embarqués
- Chaque périphérique a ses propres règles
- On y reviendra...

Appels système / appels d'environnement

- Délègue l'entrée-sortie au *niveau du dessus*
  Système d'exploitation, simulateur, etc.
- Chaque environnement a ses propres règles
- RISC-V: instruction `ecall` (*environment call*)

Convention d'appel système

Linux/RISC-V et RARS

- `a7` contient le numéro de l'appel système (RTFM)
- `a0` à `a6` contiennent les arguments éventuels de l'appel système
- Instruction `ecall` (sans opérande)
- `a0` (et parfois `a1`) contient alors la valeur de retour éventuelle
- Les autres registres sont inchangés
- Les détails de la magie plus tard, et en INF3173

Bonnes pratiques

- Déclarez des symboles pour les appels systèmes (`.eqv`)
- Initialisez `a7` en premier pour mieux voir les arguments

Appels système RARS

Quelques appels systèmes RARS

- PrintInt: a7: 1
  - Entrée: a0: nombre à afficher
  - Bogue RARS1.6: tronque à 32 bits en RV64
- PrintChar: a7: 11
  - Entrée: a0: caractère à afficher
- ReadInt: a7: 5
  - Sortie: a0: nombre lu
- ReadChar: a7: 12
  - Sortie: a0: caractère lu
  - Bogue RARS1.6: brisé en mode ligne de commande
- Exit: a7: 10

Détails RARS

- Onglet *Syscalls* dans l'aide pour la liste et la documentation
- Option *Settings>Popup dialog for input syscalls*

Exemple appels système RARS

```
# Appels système utilisés
.eqv PrintInt, 1
.eqv Exit, 10
# ...

li a7, PrintInt
li a0, 42
ecall
li a7, Exit
ecall
```

Exercice appels système RARS

- Écrire un programme `somme.s` qui
  - Lit deux nombres
  - Affiche la somme
  - Puis termine

Exercice appels système RARS

- Écrire un programme `somme.s` qui
  - Lit deux nombres
  - Affiche la somme
  - Puis termine
- Écrire un programme `bonjour.s` qui
  - Lit deux caractères (vos initiales)
  - Affiche « Bonjour » suivi des initiales
  - Puis termine

Appels système Linux

- Numérotation Linux officielle unistd.h ($\approx$ 400)
- Note: numérotation change en fonction des architectures
- Certain sont supportés par RARS (compatibilité)
- Pages du manuel pour les détails (commande `man` section 2)
- On y reviendra...

Exit (v2)

- Termine le programme avec un code de retour (cf INF1070)
- a7: 93
- a0: le code de retour

Les autres

- PrintInt, ReadInt, PrintChar, ReadChar, etc.
- $\rightarrow$ Pas d'équivalent Linux direct

Point d'arrêt (*breakpoint*)

- `ebreak` (*environment break*)
- Appel à l'environnement pour suspendre le programme

RARS

- Pause l'exécution
- Pratique pour déboguer

Linux

- Utilisé par `gdb` (GNU debuger) pour déboguer les programmes
- Autrement crashe le programme
- Détails en INF600C

Structures de contrôles

Branchements direct et inconditionnel

`j label` (pseudoinstruction, *jump*)

- Indique l'adresse de l'instruction suivante
- `label` doit être à +/- 1Mio de l'instruction
- L'instruction machine contient le **décalage**
  - Plutôt que l'adresse absolue de la cible
  - `pc = pc + décalage`
  - Décalage signé sur 21 bits

Note: la vraie instruction est `jal`... plus tard

Exemple, que fait ce programme

```
        li   a7, 1 # PrintInt
        j    ici
labas:  li   a0, 1
        ecall
        j    fin
ici:    li   a0, 2
        ecall
        j    labas
fin:    li   a7, 10 # Exit
        ecall
```

Branchement conditionnel (type B)

- `beq rs1, rs2, label` (*branch equal*)
- `bne rs1, rs2, label` (*branch not equal*)
- `blt rs1, rs2, label` (*branch less than*)
- `bge rs1, rs2, label` (*branch greater or equal*)
- `bltu rs1, rs2, label` (*branch less than unsigned*)
- `bgeu rs1, rs2, label` (*branch greater or equal unsigned*)
- `bgt`, `ble`, `bgtu`, `bleu` pseudoinstructions (permute `rs1` et `rs2`)

Si la condition est vérifiée,
Alors branche à `label`,
Sinon ne branche pas et continue à l'instruction suivante

- `label` doit être à +/- 4ko de l'instruction
  codage sur 13 bits signé, le bit de poids faible fixé à 0

Branchement conditionnel

Pseudoinstructions sur zéro

- `beqz`, `bnez`, `blez`, `bgez`, `bltz`, `bgtz`
- Comparent un registre avec 0 (`x0`)

Pas de version immédiate

- Initialiser un registre avec la valeur
  - Instruction `li` juste avant
  - Utiliser `t0` à `t6`
- Comparer avec zéro
  - Quitte à faire une soustraction avant

Exemple

```
# Lit un nombre plus grand que un
li a7, 5 # ReadInt
ecall
li t0, 1
blt a0, t0, erreur
# on continue si >=0
# ...

erreur: # instructions si <0
```

Exercices

Écrire un programme `max.s`

- Lire deux nombres et affiche le plus grand des deux

Écrire un programme `compte.s`

- Compter de 1 à 100 (inclus)

Écrire un programme `lettre.s`

- Lire des caractères (terminée par `.`)
- Afficher le nombre de lettres `'o'`
- Exemple: « `Bonjour le monde.` » $\rightarrow$ 3

Jump and link (la vérité sur j)

- `jal rd, label` (type J, *jump and link*)
- Sauvegarde l'adresse de retour dans `rd` puis branche à `label`
- **adresse de retour**: l'adresse de l'instruction suivante
- `rd`: registre sauvegardant l'adresse retour
- `label`: doit être à +/- 1Mo de l'instruction
  - L'instruction machine accepte un **décalage** signé de 21 bits

Pseudoinstructions

- `jal label` → `jal ra, label`
  - utilise `ra` (*return address*) par défaut
- `j label` → `jal x0, label` (pas de sauvegarde)

Branchement indirect et inconditionnel

- `jalr rd, rs1, offset` (type I, *jump and link register*)
  - Branche à l'adresse `rs1+offset`
  - `rd`: registre sauvegardant l'adresse de retour
  - `offset`: décalage par rapport à `rd` (12 bits signé)

Pseudoinstructions

- `jr rs1` → `jalr x0, rs1, 0`
  - Ni décalage, ni sauvegarde
- `jalr rs1` → `jals ra, rs1, 0`
  - Sauvegarde dans `ra`; pas de décalage
- `call label`
  - Combine avec `auipc` pour brancher loin, sauvegarde dans `ra`
- `tail label`
  - Combine avec `auipc` pour brancher loin, pas de sauvegarde
- `ret` → `jalr x0, ra, 0`
  - Branche sur `ra` (sans décalage ni sauvegarde)

Bibliothèques et appel de routines

Routines

- Synonymes: sous-programmes, procédures, fonctions, *subroutine*
- Idée: encapsuler du code pour le réutiliser
  - Dans le même programme
  - Voire dans des programmes différents
- Traitement spécifique et circonscrit
- Relativement indépendant du reste du programme
- Vieux concept: 1947

Bénéfices

- Permet d'**organiser** le code
  - En le découpant en entités plus simples
  - Diviser pour reigner
- Permet de **factoriser** du code
  - Réutilisation
- Permet de **sous-traiter**
  - Déléguer le développement entre personnes et organisations

Convention d'appel

- **Invoquant**: morceau de code qui fait l'appel (*caller*)
- **Invoqué**: morceau de code qui reçoit l'appel (*callee*)

Convention d'appel

- Les conventions utilisées entre invoquants et invoqués
  - Comment représenter et partager l'information?
  - Qui a la responsabilité de quoi?
- ABI: Interface binaire-programme (*application binary interface*)
  - Contient la convention d'appel
  - Et d'autre informations (on y reviendra)

ABI RISC-V ELF (de base)

- *Processor-specific application binary interface document for RISC-V (v1.0)*

Convention d'appel RISC-V ELF

- `a0` jusqu'à `a7` contiennent les arguments (**a***rgument register*)
- `a0` (et parfois `a1`) contient les résultats
- Les registres `s0` à `s11` restent inchangés (**s***aved register*)
  - Ainsi que `ra`, `sp`, `gp` et `tp`
- Les autres registres (`a` et `t`) peuvent être modifiés sans préavis par la routine
  - N'y mettez rien que vous souhaitez garder
  - Donc, rangez vos données dans les registres `s` (*saved*)
  - Ou rangez vos données en mémoire (une autre fois)

Bibliothèque de routines (*library*)

Bibliothèque (en général)

- Collection de routines
- Prêtes à l'emploi par d'autres programmes
- Note: évitez de dire « librairie »

Mises en œuvre et détails spécifiques

- Aux langages de programmation
- Aux outils
- Aux environnements d'exécution

Bibliothèques en RARS

Interface graphique

- Option *Settings*>*Assemble all files in directory*
- Option *Settings*>*Assemble all files currently open*

En ligne de commande

- Mettez plusieurs fichiers en argument
- `java -jar rars.java program.s lib1.s lib2.s`

Bibliothèques GNU/Linux

Plusieurs fichiers en argument de gcc

`gcc` s'occupe des détails pour vous

- `gcc program.s lib1.s lib2.s -static -nostdlib -o program`
- ou `riscv64-linux-gnu-gcc` en compilation croisée

Compilation séparée (détails)

- Génération de fichiers `.o` intermédiaires
- Utilisation de bibliothèques binaires statiques
- Édition de liens pour lier le tout
- Utilisation de bibliothèques binaires dynamique
- Le tout une autre fois...

Bibliothèque `libs.s`

- Utilisée dans le cours
- Portable RARS et GNU/Linux
- Développement maison

Quelques routines disponibles

- `printInt`, `printChar`, `readInt`, `readChar`
- Même comportement (à peu près) aux appels systèmes RARS
- Permet de contourner des limitations de RARS
- Permet la compatibilité avec Linux

Exemple

```
li a0, 42
call printInt

li a0, 0
call exit
```

- On utilise `call` pour les appels de routines de bibliothèques
- Car on ne sait pas d'avance où est la routine
- Ni si elle est proche de `pc`

Exercice

Écrire un programme `max2.s`

- Lire deux nombres et afficher le plus grand des deux
- Contrainte: utiliser `libs.s`
- Tester avec RARS
- Tester avec GNU/Linux

Conclusion

Programmes assembleur

Instructions, directives

- Instructions
  - Nouvelles instruction: `ecall`, `ebreak`, `beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu`, `jal`, `jalr`, `sltiu`, `jal`, `jalr`
  - Nouvelles pseudoinstructions: `j`, `bgt`, `ble`, `bgtu`, `bleu`, `beqz`, `bnez`, `blez`, `bgez`, `bltz`, `bgtz`, `j`, `jr`, `ret`, `call`, `tail`
- Directives: seulement `.eqv` (pour l'instant)

Ordre et discipline

- Commentaires, symboles et valeurs
- Le langage d'assemblage est difficile
  - Le programmeur a beaucoup de responsabilité
  - Pas de petites roues
- Stratégies
  - Codez proprement et simplement
  - Une étape à la fois

La prochaine fois

Mémoire et adressage

- Lecture et écriture
- Boutisme et alignement
- Chaînes de caractères
- Et plein de directives \o/