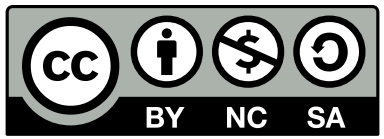
Chapitre 5 - Compilation

INF7641 Compilation

Jean Privat

Université du Québec à Montréal

INF7641 Compilation
v251



Plan

RISC-V

RISC-V

Jeu d'instruction (*instruction set architecture*, ISA)

- *Reduced instruction set computer*
- Instructions élémentaires peu nombreuses
- Beaucoup de registres
- Peu de cas particuliers
- Spécification RISC-V

Architecture ouverte

- Pas de redevances
- License CC-BY-4.0

À la mode

- Framework Laptop's RISC-V board for open source diehards is available

Caractéristiques RISC-V

Processeur « à la carte »

- 32, 64 ou 128 bits
- Normal (I) ou *embeded* (E)
- Plus de 40 extensions

RV64I (nu)

- 32 registres généraux (entiers) 64 bits
  mais un, x0, moins général que les autres...
- 1 compteur ordinal (*program counter*, pc)
- Opérations arithmétiques, logiques et de contrôle classiques
  Une quarantaine...
- Chaque instruction sur 32 bits

Exemple : extension M

- Multiplication et division
- 8 opérations arithmétiques supplémentaires

Langage machine

- Codage binaire exécuté microélectroniquement par le processeur
- Séquences d'instructions en mémoire et compteur ordinal

Instructions

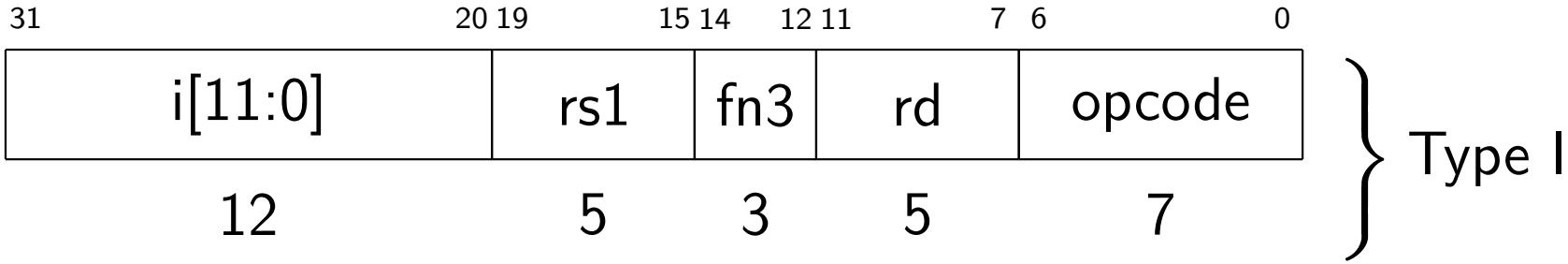
- Opération = quoi faire?
- Opérandes = quels registres, valeurs immédiates, emplacements mémoires?

Catégories d'instructions

- Instruction arithmétiques et logiques (calculs)
- Instructions de transfert (déplacement d'information)
- Instructions de contrôle (déroulement de l'exécution)

Langage machine RISC-V

Type I: instructions avec une petite valeur immédiate



- opcode $\rightarrow$ code d'opération
- rd $\rightarrow$ registre destination (5 bits: $2^5 = 32$ registres)
- fn3 $\rightarrow$ sous-fonction (3 parce que 3 bits ici)
- rs1 $\rightarrow$ registre source
- i[11:0] $\rightarrow$ les bits 0 à 11 de la valeur immédiate (généralement avec extension de signe)

Exemple addi (instruction de type I)

- `addi s0, s1, 42` « range dans `s0` la somme de `s1` et de 42 »

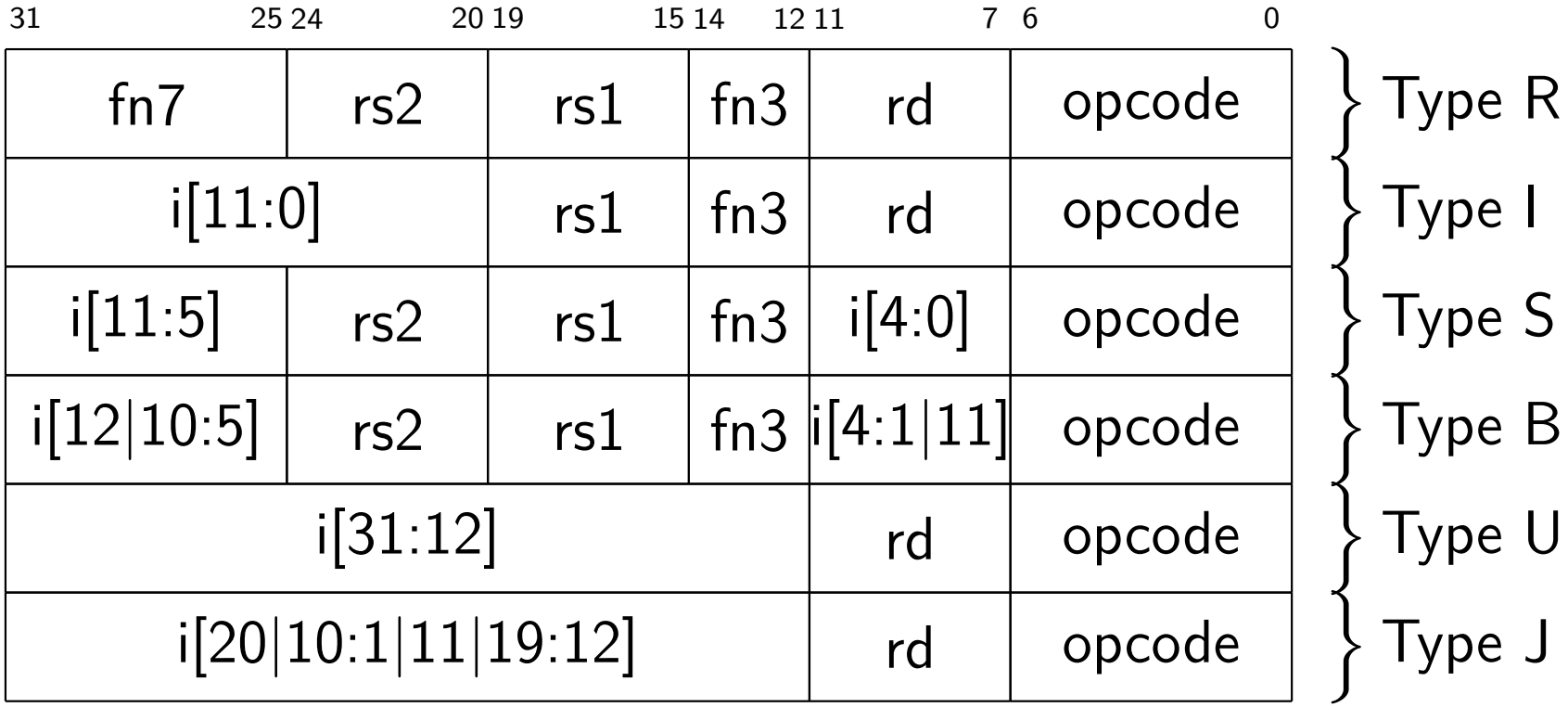
31 20	19 15	14 12	11 7	6 0
i[11:0]	rs1	fn3	rd	opcode
42	s1=x9	addi	s0=x8	OP-IMM
0 0 0 0 0 0 1 0 1 0 1 0	0 1 0 0 1	0 0 0	0 1 0 0 0	0 0 1 0 0 1 1
0 2 A	4	8	4	1 3

- Soit `02A48413` en hexadécimal

Notes

- Les frontière de champs ne sont pas aux octets
- Les immédiats ont 12 bits seulement dans le type I

Les différents types d'instructions RISC-V



- Le décodage est pénible à la main
  Surtout les valeurs immédiates
- Mais facile pour une micro-architecture

Exercice de décodage

Instruction `0x00813903`?

- Utilisez l'annexe ou fouillez dans les spécifications
- Transformer en binaire
- Chercher l'opcode (7 bits de poids faible)
- Découper les champs
- Décoder les champs un à un
- Fabriquer l'instruction
- Trouver la pseudoinstruction la plus simple équivalente (si besoin)

Instruction en mémoire

Petit-boutisme

- Les instructions RISC-V sont petit boutistes
- L'octet de poids faible est stocké en mémoire en premier
- Avantage: La micro-architecture connaît l'opcode en premier
- Attention: L'ordre des octets quand on décode à la main
  - 32 bits d'un coup (dans un registre interne)
  - 32 bits en mémoire (4 octets un par un petit-boutiste)

Alignement

- Les instruction RISC-V doivent être alignés sur 32 bits
  (ou 16 bits avec l'extension C)
- L'adresse de l'instruction doit être un multiple de 4
  (ou de 2 avec l'extension C)

Note: cela explique pourquoi i[0] n'apparaît pas dans les type B et J

Fichier objet (on y reviendra)

- Du code machine exécutable
- Des données binaires brutes
- Des méta-informations
  - Pour faire l'édition de liens
  - Pour charger les programmes en mémoire
  - De débogage

Langage d'assemblage (assembleur)

Langage de programmation fortement en correspondance avec le langage machine

- Syntaxe ASCII
- Mnémoniques pour les instructions `addi`, `beq`...
- Et pseudo-instructions `li`, `call`...
- Noms pour les registres `sp`, `a0`...
- Valeurs littérales `-42`, `'*'`...
- Étiquettes `main`, `.L25`...
  (pour pas calculer les adresses ou les décalages à la main)
- Directives
  - Pour déclarer des données en mémoires `.word`, `.space`...
  - Pour contrôler le processus d'assemblage `.globl`, `.text`...

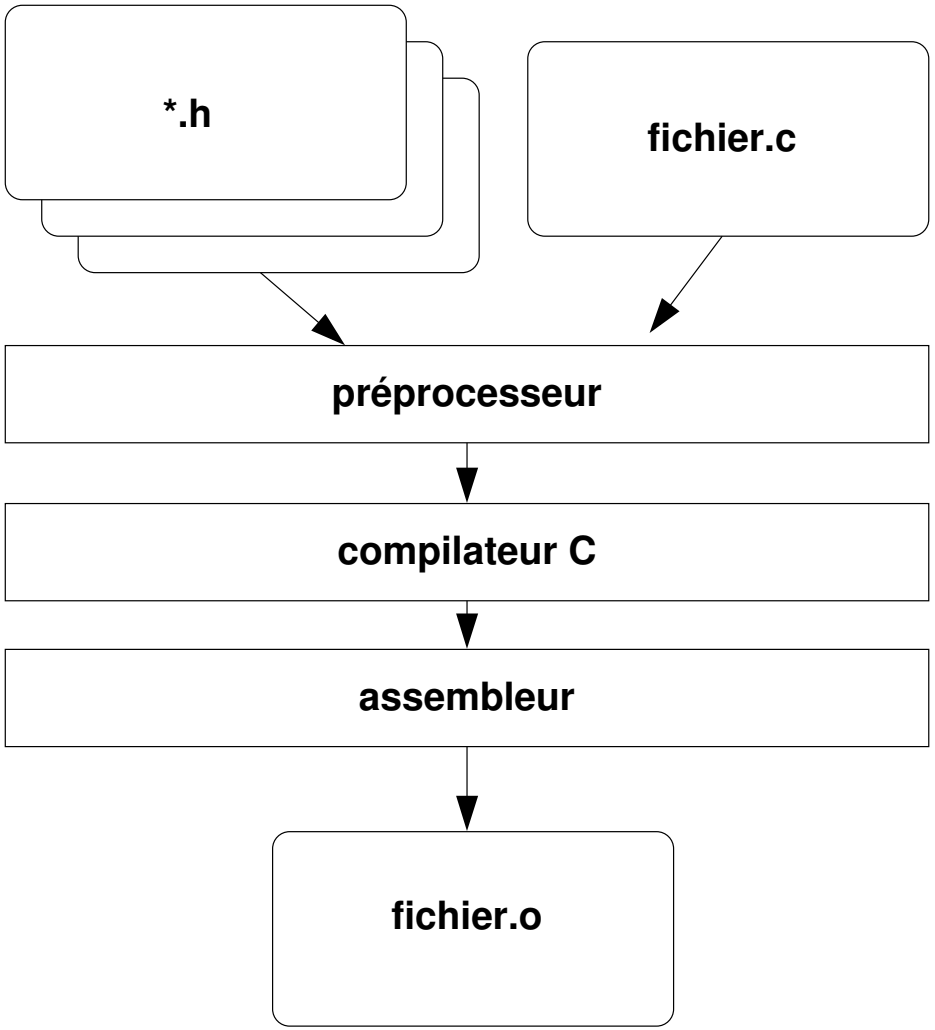
Assembleur RISC-V

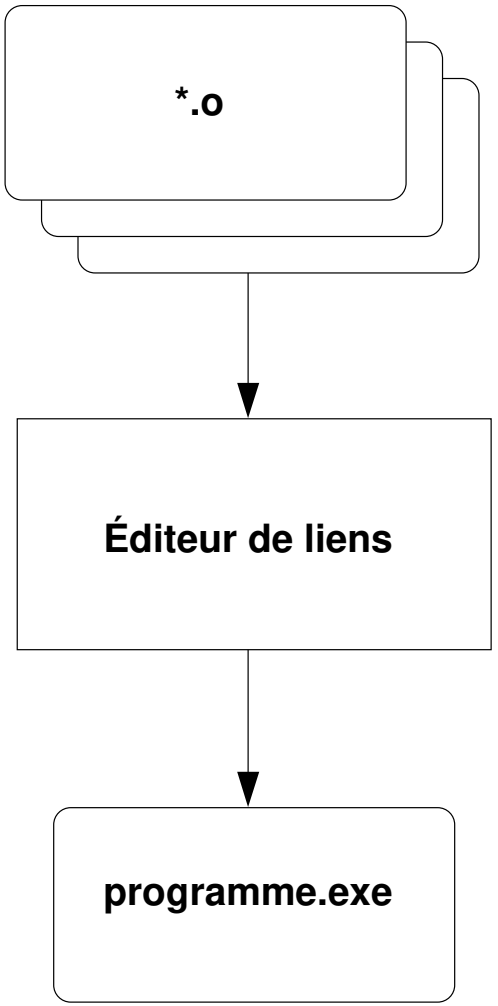
- Travail en cours
- Pour le reste, c'est `GNU assembler`.

```
    li   a0, 15
    li   a2, 0
    blez a0, .affiche
    li   a1, 0
    li   a2, 1
    addi a0, a0, -1
.boucle:
    blez     a0, .affiche
    add  a3, a2, a1
    mv   a1, a2
    mv   a2, a3
    addi a0, a0, -1
    j    .boucle
.affiche:
    li   a7, 1       # PrintInt(a0)
    mv   a0, a2
    ecall
    li   a7, 10      # Exit
    ecall
```

MiniCC : Compilateur pour MiniC

Schéma de compilation classique en C (rappel)





MiniC vs. RISC-V

- `NInt`
- `NStm_Assign`
- `NExp_Add`
- `NExp_Sub`
- `NExp_Mul`
- `NExp_Lt`
- `NStm_If` et `NStm_While`
- `NStm_Call` et `NExp_Call`
- `NStm_Return`
- `Variable`

- `li`
- `mv`
- `add`
- `sub`
- `mul`
- `slt`
- `j`,`breqz`
- `call`
- `ret`
- registre

Mise en œuvre

- Générer l'assembleur en une passe
- Registres : un curseur, avant=utilisé après ou égal=disponible
  - De `s0` à `s11` (ICE au delà...)
  - Associer les variables à des registres disponibles
  - Associer les expression intermédiaires à des registres disponibles
  - Restaurer le curseur au fur et à mesure
- Structures de contrôles (if/while)
  - Utiliser étiquettes, `j` et `breqz`
- Sous-programmes ?
  - Plus tard...

Runtime

Point d'entrée ?

- Convention C: l'étiquette `main` est le point d'entrée.
- Exporter l'étiquette avec `.globl`

Primitives ?

- `printint`, `printbool`, `println` ?

Runtime

Point d'entrée ?

- Convention C: l'étiquette `main` est le point d'entrée.
- Exporter l'étiquette avec `.globl`

Primitives ?

- `printint`, `printbool`, `println` ?
- Fournir une implémentation binaire clé en main
- `minic.c` !

Runtime

Point d'entrée ?

- Convention C: l'étiquette `main` est le point d'entrée.
- Exporter l'étiquette avec `.globl`

Primitives ?

- `printint`, `printbool`, `println` ?
- Fournir une implémentation binaire clé en main
- `minic.c` !
- Mais comment les appeler ?

Application Binary Interface (ABI)

ABI = Règles de savoir-vivre en société

- Pour l'interopérabilité de programmes en code machine
- Code objet, bibliothèques statiques, bibliothèques dynamiques
- Déterminé par convention
- Triplet architecture / fournisseur / système d'exploitation
  - `gcc -dumpmachine`

Portée des ABI

- Utilisation et vie des registres
- Taille, alignement et organisation des types de donnés
- Convention d'appels
  - Passage d'arguments et de valeurs de retour
  - Nettoyage éventuel de la pile
- Relocation (édition de liens)
- Chargement dynamique
- Information de débogage
- Etc.

Quelques ABI

- x86 : nombreuses conventions d'appel. fiche Wikipédia
  - Dépendent des outils, systèmes et langages
- System V Application Binary Interface AMD64
  - Pour Unixes sur x86_64, 2024, 154 pages
- RISC-V ABIs Specification
  - Pour Unixes, 2022, 55 pages

ABI RISC-V : TL;DR

Registres

- `a0` à `a7` pour args et retour (et pile si pas assez)
- `t0` à `t6` pour registres temporaires (peuvent être écrasés par certains appels ou pseudo-instructions)
- `s0` à `s11` pour registre sauvés (doivent survivre entre les `call`)
- `sp` pointeur de pile
  - Chaque fonction est responsable et nettoie sa pile
  - Alignée à 16 octets
- `ra` pour l'adresse de retour
  - À sauvegarder si besoin
- `gp` et `tp` pointeurs global et de thread

Appel de fonction

Appel de fonction

En C, chaque fonction est son propre « petit royaume »

- Étiquette d'entrée unique
- Accès au données (argument, retour, et autre) selon l'ABI
- Utilisation des registres selon l'ABI

Pile

En C, chaque fonction exécutée est son propre « petit royaume »

Avec son petit bout de pile : stack frame.

- Allocation et désallocation de pile
  Décrémenter et incrémenter `sp`
- Sauvegarder ce qu'impose l'ABI (`s`, `ra` si modifié)
- Rendre la pile dans l'état où on l'a trouvé
- Stocker plus de variables locales (de de plus grosses)

MiniCC : Runtime (le retour)

- Passer les arguments de `printint` et `printbool` dans a0

MiniCC : Runtime (le retour)

- Passer les arguments de `printint` et `printbool` dans `a0`
- Et sauver `ra` dans la pile

MiniCC : Runtime (le retour)

- Passer les arguments de `printint` et `printbool` dans a0
- Et sauver `ra` dans la pile
- Donc allouer/désallouer 16 octets dans la pile

Exécuter (simple)

Natif

- `gcc programme.s minic.c -o programme`
- `./programme`

Croisé & émulateur qemu

- `riscv64-linux-gnu-gcc programme.s minic.c -o programme -static`
- `qemu-riscv64 programme`

Assembleur et simulateur RAR

Exécuter (simple)

Natif

- `gcc programme.s minic.c -o programme`
- `./programme`

Croisé & émulateur qemu

- `riscv64-linux-gnu-gcc programme.s minic.c -o programme -static`
- `qemu-riscv64 programme`

Assembleur et simulateur RAR

- On a besoin d'un runtime qui implémente les `print*` → `minic_rars.s`
- `java -jar rars.jar programme.s minicc-rv64.s`

MiniCC: Gestion des sous-programmes

- On travaille toujours en une passe
- On sait pas d'avance qui est est modifié par la fonction
- Donc, on prend pas de chance, on sauve tout
- On gère pas non plus les variables en mémoire (pour l'instant)

Mise en œuvre (par fonction)

Prologue

- Étiquette d'entrée unique: le nom de la fonction
- Sauvegarder les 12 registres `s0` à `s11`
- Sauvegarder `ra`
- Combien d'octets allouer dans la pile ?

Mise en œuvre (par fonction)

Prologue

- Étiquette d'entrée unique: le nom de la fonction
- Sauvegarder les 12 registres `s0` à `s11`
- Sauvegarder `ra`
- Combien d'octets allouer dans la pile ?
- `addi sp, sp, -112`: 12*8 (`s`) + 8 (`ra`) + 8 (alignement)
- Copier les paramètres dans les registres `s` associés.

Épilogue

- Unique aussi, pour simplifier
- Restaurer les registres `s0` à `s11`, `ra`, `sp`
- Note: compiler `NReturn` par assigner `a0` suivi d'un `j` vers l'épilogue
- Mettre `a0` à 0 avant l'épilogue, en cas de sortie sans `return` explicite

Édition de liens et chargement

Utilisation de bibliothèques (rappel)

- Un programme principal
- Des bibliothèques fournissant des routines
- Combiner le tout pour avoir un programme exécutable

```
java -jar rars.jar program.s lib1.s lib2.s
gcc program.s lib1.s lib2.s -static -o program
```

Executable and Linkable Format (ELF) 1993

- Format de fichier standard et multi-plateforme des systèmes Unix
- Pour fichiers objet, exécutables, bibliothèques partagées (et modules noyaux, et *core dumps*)

Entête générale

- `objdump -f` / `readelf -h`
- architecture, noyau, ABI, point d'entrée

Entête du programme (ou des segments)

- `objdump -p` / `readelf -l`
- Pour charger le fichier en mémoire
- Décalage, taille, droit (`rwx`), alignement des morceaux

Entête des sections

- `objdump -h` / `readelf -S`
- Pour faire l'édition de liens

Compilation séparée

Assembler indépendamment chaque fichier du programme

- `gcc -c` (compile) et assemble, produit un fichier **objet** (`.o`)
- En fait `gcc` invoque simplement `as` (utiliser `-v` pour voir)

```
gcc -c program.s
gcc -c lib1.s
gcc -c lib2.s
```

Édition de liens

Combiner les `.o` pour faire un exécutable

- `ld` mais options compliqués
- `gcc` sans option comme `-c` produit un exécutable
  - Appelé `a.out` pour des raisons historiques
  - $\rightarrow$ Utilisez `-o` pour renseigner un meilleur nom
- En fait `gcc` invoque indirectement `ld`
  - `-v` montre `collect2` un utilitaire interne qui appelle `ld`
  - Beaucoup d'option et configuration complexe

```
gcc program.o lib1.o lib2.o -static -o program
```

- `-static` désactive l'utilisation de bibliothèques dynamiques
- `-nostdlib` désactive l'utilisation des bibliothèques par défaut

Export de symboles

Directive `.globl`

- Exporte un symbole défini dans le fichier assembleur
- Pour qu'il soit utilisable (public)

Importation

- Pas besoin de déclarer les importations en `GNU as` et `RARS`
- Un symbole utilisé **non défini** dans le fichier est considéré externe

Bibliothèques binaires

- Prête à l'emploi et empaquetés

Bibliothèques statiques

- Intégrés lors de l'éditions de liens dans l'exécutable final
- Fichiers `.a` sous Linux: `find /lib/ -name '*.a'`
  - C'est une simple archive de `.o`, cf. commande `ar`

Bibliothèques dynamiques

- Non intégrée dans l'exécutable final
- Chargés durant l'exécution du programme
  - Lors du démarrage du programme
  - Ou paresseusement lors de son exécution
  - Ou programmativement (plugins par exemple)
- Fichiers `.so` sous Linux: `find /lib/ -name '*.so'`
- Fichiers `.dll` sous Windows

Bibliothèques standard GNU

`libc`, `crt0` et autre

- Contient les routines standard du langage C
- Et les routines de démarrage d'un exécutable en C
  - `crt0` = C Runtime 0
  - Fournit le point d'entrée `_start`
  - `_start` appelle (indirectement) `main`
  - `main` est implémenté par le programme principal
- Initialise et configure de nombreuses affaires

Adresses des étiquettes

```
la s0, etiquette
sw s1, etiquette
```

Question: comment déterminer l'adresse de l'étiquette

Adresses des étiquettes

```
la s0, etiquette
sw s1, etiquette
```

Question: comment déterminer l'adresse de l'étiquette

- La position finale des étiquettes n'est pas connu à l'assemblage, mais
  - à l'édition de liens
  - au chargement du programme
  - au chargement dynamique de bibliothèques pendant l'exécution du programme
- Le code machine généré à l'assemblage doit alors être prévu pour

Édition de liens

Assemblage

- On génère du code machine
- Les symboles inconnus sont laissés non-résolus
- Mais on note le symbole non résolu-quelque part
  $\rightarrow$ La **table des symboles**

ELF: Table des symboles

- `objdump -t` / `readelf -s` / `nm`
- Pour l'édition de liens (et le débogage)
- Nom, position, section associée, liaison (local, global, faible...), type (fonction, objet...)

Réadressage (*relocation*)

Quand on connaît les vraies valeurs des symboles:

- On « *bricole* » le code machine déjà généré
- On insère les bonne valeurs directement dans le binaire
- C'est bas-niveau, on bricole les instructions machine ou les données binaires
- ELF: `objdump -r` / `readelf -r`

Conventions

Les règles et techniques de **réadressage**

- Ne sont pas déterminées par l'**architecture**
- Mais par les conventions des outils et environnements (*ABI*)
- RISC-V ABIs Specification voir section 8.4. Relocations

Code indépendant de la position (PIC)

- Utilisé par les bibliothèques dynamiques
- Et certains exécutables (*position-independent executable*, PIE)

Pourquoi ?

- Pouvoir charger une bibliothèque ou un exécutable n'importe où en mémoire
- Mais garder le même code machine
- $\rightarrow$ Partager la mémoire entre différents processus

Une solution : *PC-relative*

- Ne pas utiliser d'adresses absolues dans le code machine
- Mais utiliser des adresses relatives à l'instruction courante (`pc`)
- Exemple: `auipc` en RISC-V

Chargement dynamique

- Fait en espace utilisateur par une bibliothèque `ld.so`
- `man ld.so`
- `LD_DEBUG=all` pour que `ld.so` nous parle
- `LD_PRELOAD` pour injecter nos propres symboles

ELF

- Table des symboles dynamiques
  - `objdump -T` / `readelf --dyn-syms` / `nm -D`
- Table de réadressage dynamique
  - `objdump -R` / `readelf -r`

Global Offset Table (GOT)

- Mécanisme pour permettre PIC et partage de mémoire
  - Rappel : on veut pas d'adresse absolue dans le segment code partagé (`r-x`)
- Table à un position connue (*PC-relative*)
  - Dans des données privés du processus (`rw-`)
  - Contient les adresses globales des symboles utilisés
- *load* et *store* globaux coûtent un *load* de plus
  - Charger l'adresse réelle depuis la bonne case de la GOT
  - Puis utiliser l'adresse réelle
- Le chargeur dynamique calcule les adresses et remplit la GOT
- Une GOT par exécutable et par bibliothèque partagée

Procedure Linkage Table (PLT)

- Mécanisme utilisé par le chargement dynamique de bibliothèques partagées
- À chaque fonction externe utilisée, un trampoline local est associé
  - Tous ces trampolines sont rangés dans la PLT
  - la PLT est dans le segment texte (`r-x`) partagé
- Chaque appel de fonction externe utilise l'adresse (*PC-relative*) du trampoline
- Chaque trampoline de la PLT contient du code similaire

```
la t0, GOT+8 # Le décalage dépend du numéro de l'entrée
jr t0
```

- Un appel global coûte un load et un saut indirect

Chargement paresseux

Charger les fonctions externes au besoin

De façon transparente

- Pour chaque fonction externe:
  - la GOT contient initialement l'adresse du chargeur
- Le chargeur (fourni par `ld.so`)
  - Charge la bibliothèque en mémoire, si besoin
  - Cherche l'adresse globale du symbole demandé
  - Met-à-jour l'entrée dans la GOT

Activable/désactivable

- Avec `-znow` ou `-zlazy` dans `ld`
- Donc `-Wl,-znow` dans `gcc`
- Le défaut est `-znow` de nos jours