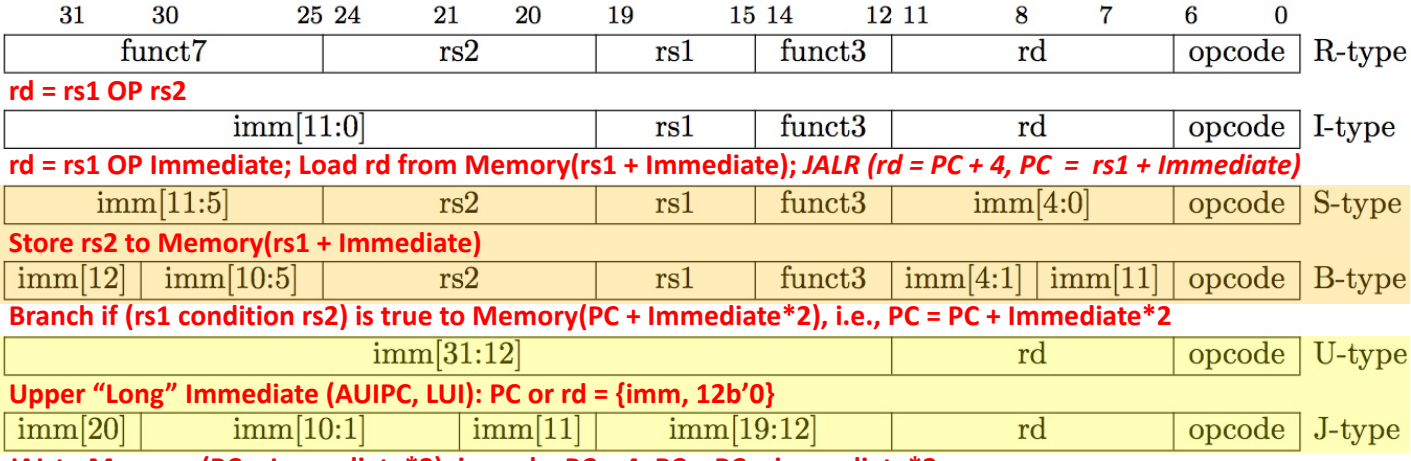
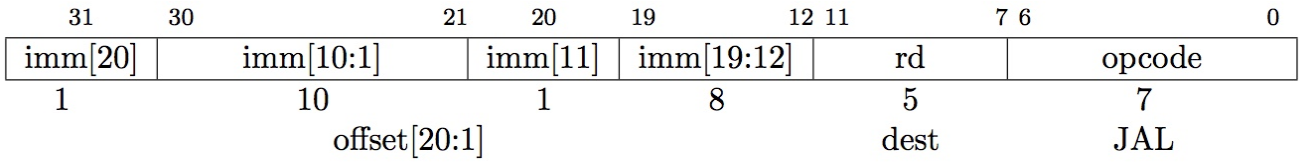
Quelques precisions sur les instructions et la compilation

- Format des Instructions et focus sur J/JAL
- Multiplication et Division
- Interpretation vs. Translation
- De l'assembleur à l'exécution

Format des instructions: RISC-V



J-Format pour les instruction Jump (JAL)



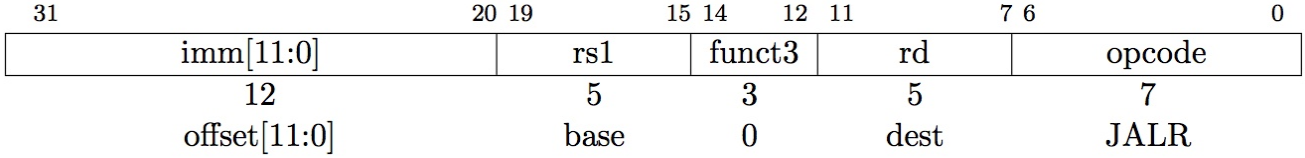
- JAL sauvegarde PC+4 dans le register rd (adresse de retour)
  - L'instruction assembleur "**j**" jump est une pseudo-instruction, utilise JAL mais place rd=x0 pour ignorer l'adresse de retour.
- Met à jour la valeur du PC = PC + offset

Utilisations de JAL

```
# j pseudo-instruction
j Label = jal x0, Label # l'adresse de retour est ignorée

# Appel d'une fonction dans un espace de 2^18 instructions autour du PC
jal ra, FuncName
```

Instruction JALR (I-Format)



- `JALR rd, rs, immediate`
  - Sauvegarde `PC+4` dans `rd` (adresse de retour)
  - Fixe le `PC` = `rs` + `immediate` (12 bit)

Utilisation de JALR

```
# ret et jr pseudo-instructions
ret = jr ra = jalr x0, ra, 0

# Appel de fonction à n'importe quelle adresse de 32 bits
lui x1, <hi20bits>
jalr ra, x1, <lo12bits>

# Saut "PC-relative" avec un offset de 32 bits
auipc x1, <hi20bits>
jalr x0, x1, <lo12bits>
```

Pseudo Instructions

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = –F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = –R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

Instructions valides en assembleur mais non en langage machine
(i.e., plusieurs instructions en langage machine sont nécessaires pour les exécuter)

Quelques precisions sur les instructions et la compilation

- Format des Instructions et focus sur J/JAL
- Multiplication et Division
- Interpretation vs. Translation
- De l'assembleur à l'exécution

Multiplication d'entier (1/3)

- Un exemple (unsigned):

```
opérande1        1000    8
opérande2      x 1001    9
                 1000
                0000
               0000
          +  1000
           01001000     72
```

- m bits x n bits = produit de m + n bits

Multiplication d'entier (2/3)

- Dans l'architecture RISC-V, on multiplie le contenu des registres:
  - Valeur de 32bits value x valeur de 32bits = valeur de 64 bits
- Syntaxe de la Multiplication (signed):
  - `MUL` réalise une multiplication 32-bit×32-bit et sauvegarde les 32 bits de poids faible du résultat dans le registre de destination
  - `MULH` réalise une multiplication 32-bit×32-bit et sauvegarde les 32 bits de poids fort du résultat dans le registre de destination

Multiplication d'entier (3/3)

- Exemple:
  - En C: `a = b * c;  # a doit être declaré comme un long long`
  - En assembleur RISC-V:
    - B dans t2, c dans t3 et a dans t0 et t1

```
                    # upper half of
                    # product into $s0
                    # lower half of
                    # product into $s1
```


Multiplication d'entier (3/3)

- Exemple:
  - En C: `a = b * c;  # a doit être declaré comme un long long`
  - En assembleur RISC-V:
    - B dans t2, c dans t3 et a dans t0 et t1

```
mulh t0,t2,t3       # upper half of
                    # product into $s0
                    # lower half of
                    # product into $s1
```

Multiplication d'entier (3/3)

- Exemple:
  - En C: `a = b * c;  # a doit être declaré comme un long long`
  - En assembleur RISC-V:
    - B dans t2, c dans t3 et a dans t0 et t1

```
mulh t0,t2,t3      # upper half of
                   # product into $s0
 mul  t1,t2,t3     # lower half of
                   # product into $s1
```


Division d'entier

- Syntaxe de la Division (signed):
  - DIV realise une division d'un entire de 32 bits par un entire de 32 bits, REM donne le reste de la division.

```
DIV rdq, rs1, rs2
REM rdr, rs1, rs2
```

- Implementation en C de la division division (/) et du modulo (%)
- C: `a = c / d;` `b = c % d;`
- Assembleur RISC-V: a↔t0; b↔t1; c↔t2; d↔t3

```
div t0,t2,t3   # a=c/d
rem      t1,t2,t3   # b=c%d
```

Question

Parmi les propositions ci dessous, quelle(s) séquence(s) charge(nt) l'adresse de LOOP dans T1?

1)
```
la t2, LOOP
lw t1, 0(t2)
```

2)
```
jal LOOP
LOOP: add t1, ra, x0
```

3)
```
la t1, LOOP
```


Question

Parmi les propositions ci dessous, quelle(s) sequence(s) charge(nt) l'adresse de LOOP dans T1?

1)
```
la t2, LOOP
lw t1, 0(t2)
```

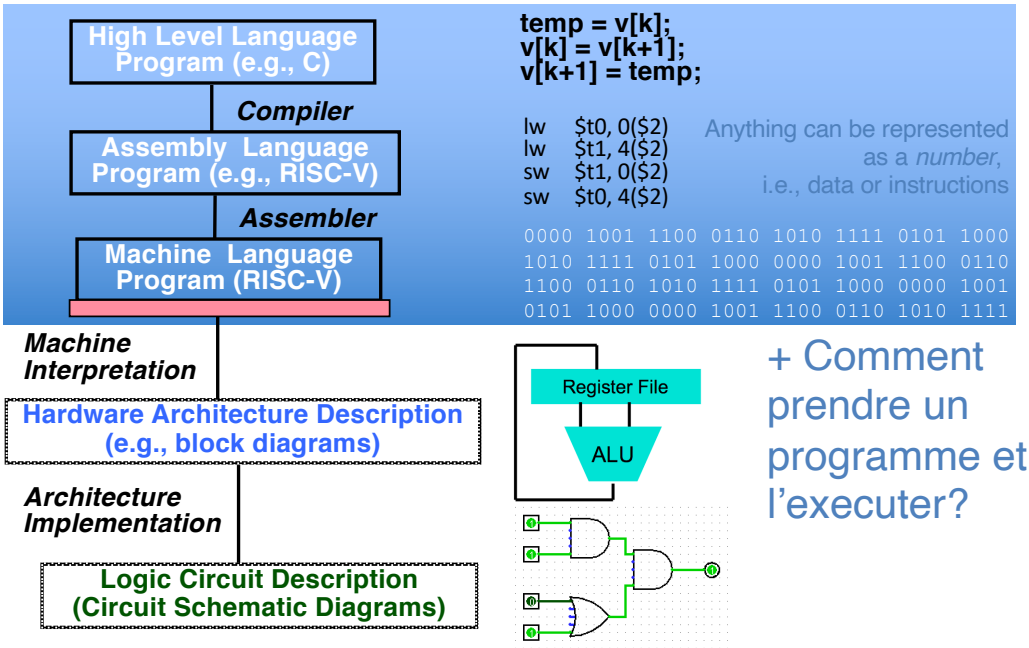
2)
```
jal LOOP
LOOP: add t1, ra, x0
```

3)
```
la t1, LOOP
```

Quelques precisions sur les instructions et la compilation

- Format des Instructions et focus sur J/JAL
- Multiplication et Division
- Interpretation vs. Translation
- De l'assembleur à l'exécution

Levels of Representation/Interpretation

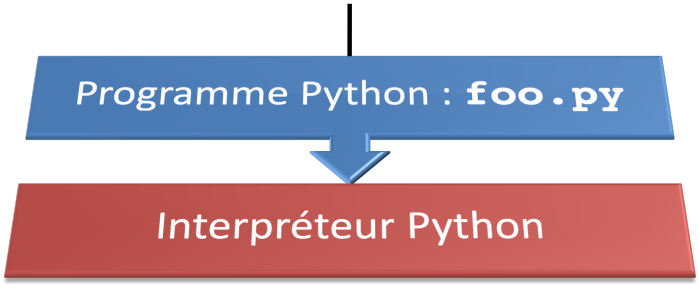


Interprétation vs Compilation

- Un Interpreteur est un programme qui exécute un autre programme:
  - L'interprétation du code source est un processus « pas à pas » : l'interpréteur va exécuter les lignes du code une par une, en décidant à chaque étape ce qu'il va faire ensuite.
- Dans un langage interprété, le même code source pourra marcher directement sur tout ordinateur. Avec un langage compilé, il faudra (en général) tout recompiler à chaque fois ce qui pose parfois des soucis.
- Dans un langage compilé, le programme est directement exécuté sur l'ordinateur, donc il sera en général plus rapide que le même programme dans un langage interprété.

Interpretation

- Par exemple, considérons un programme en python **`foo.py`**



- L'interpéteur Python est simplement un programme qui lit le python et exécute les fonctionnalités de ce programme.

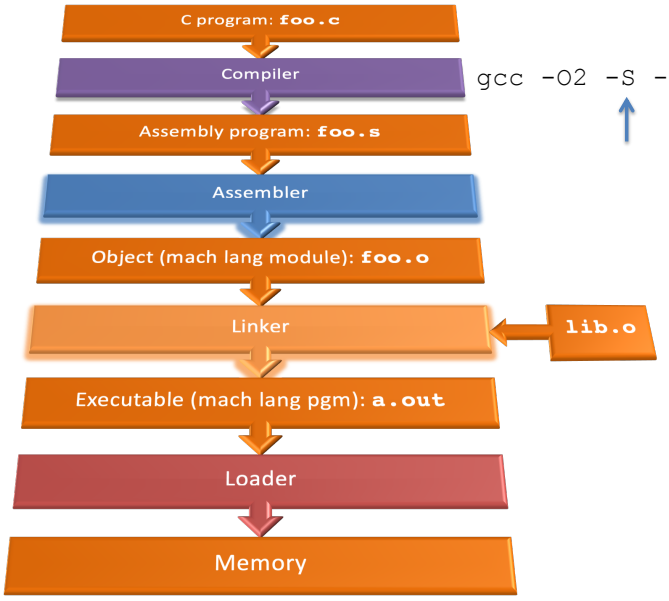
Interpretation vs. compilation? (1/2)

- C'est généralement plus simple d'ecrire un interpréteur
- L'interpréteur est plus proche du langage haut niveau et peut donc donner plus d'informations de debuggig
- L'interpréteur est plus lent (10x?), le code plus petit(2x?)
- L'interpréteur rend le code à executer independent du jeu d'instruction

Interpretation vs. compilation? (2/2)

- Le code compilé est plus efficace:
  - Important pour beaucoup d'applications
- La compilation aide à garder secret le code source du programme à l'utilisateur

Les étapes pour compiler un code C

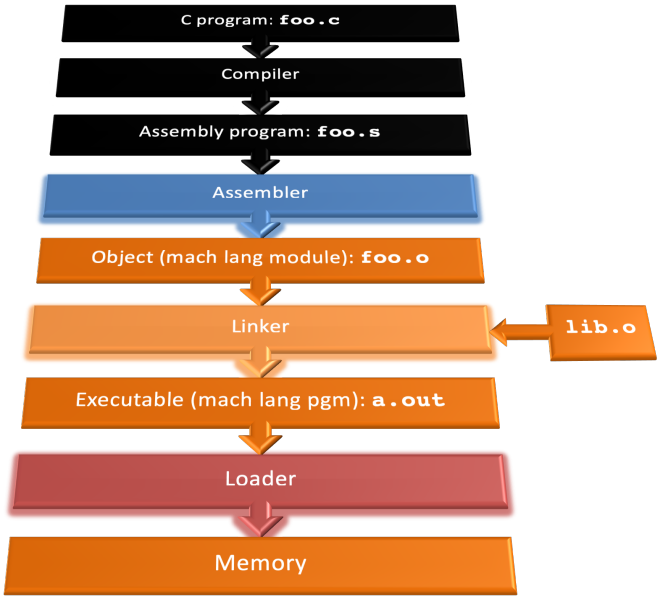


Credits UC Berkley's CS61C.

Compilateur

- Entrée: code langage de haut niveau(e.g., **`foo.c`**)
- Sortie: code assembleur
  (e.g., **`foo.s`** pour RISC-V)
- Note: La sortie peut contenir des pseudo instructions
- Pseudo-instructions: instructions que l'assembleur comprend mais pas présente dans l'architecture
  - Par exemple (move t2 to t1):
  - **`mv t1,t2`** ⇒ **`addi t1,t2,0`**

Du programme à l'exécution



Assembleur

- Entrée: langage assembleur
  (e.g., **`foo.s`** pour RISC-V)
- Sortie: Code objet pur assembleur seulement)
  (e.g., **`foo.o`** pour RISC-V)
- Lits et utilises des Directives
- Remplace les Pseudo-instructions
- Produit le langage Machine
- Crée le fichier objet

Directives assembleur

- Donne des directives à l'assembleur mais ne produit pas d'instructions machines

`.text:`	Subsequent items put in user text segment (machine code)
`.data:`	Subsequent items put in user data segment (binary rep of data in source file)
`.globl sym:`	declares `sym` global and can be referenced from other files
`.string str:`	Store the string **`str`** in memory and null-terminate it
`.word w1…wn:`	Store the *n* 32-bit quantities in successive memory words

Remplacement des Pseudo-instructions

-

Pseudo:	Real:
`mv t0, t1`	`addi t0,t1,0`
`neg t0, t1`	`sub t0, zero, t1`
`li t0, imm`	`addi t0, zero, imm`
`not t0, t1`	`xori t0, t1, -1`
`beqz t0, loop`	`beq t0, zero, loop`

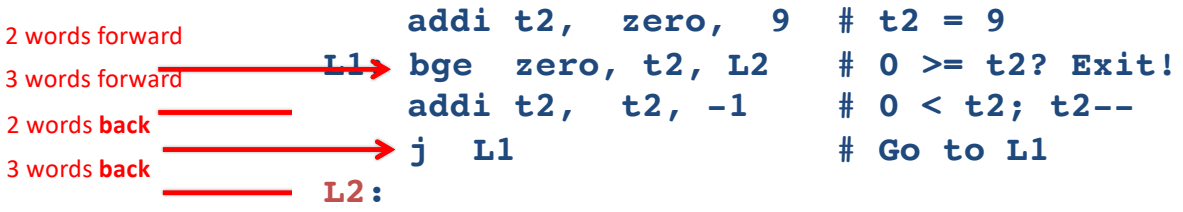
`la t0, str`	`lui t0, str[31:12]`
	`addi t0, t0, str[11:0] ou`
	`auipc t0, str[31:12]`
	`addi t0, t0, str[11:0]`

Génération du langage machine (1/3)

- Cas simple
  - Opérations arithmétiques et logiques
  - Toutes les infos sont connues dans l'instruction
- Branchements et sauts?
  - PC-Relative (e.g., **`beq/bne`** and **`jal`**)
  - Lorsque les pseudo instructions sont remplacées par les instructions réelles, on sait de combien d'instructions on doit "sauter"

Génération du langage machine(2/3)

- Le problème des "Forward Reference"
  - Une instruction de branchement peut faire reference à des labels qui sont plus loin dans le programme



  - C'est résolu en faisant deux passes sur le programme
    - La première passe mémorise les positons des labels
    - La seconde utilise la position des labels pour générer le code

Génération du langage machine(3/3)

- Comment gérer les sauts ou branchements avec des déplacements par rapport au PC (PC-relative jumps (**`jal`**) and branches (**`beq`**, **`bne`**))?
  - **`j offset`** *pseudo instruction* qui correspond à **`JAL zero, offset`**
  - On compte simplement le nombre d'instructions entre la cible et le saut pour déterminer l'offset (à donner en demi mots), on parle de code indépendant de la position : *position-independent code (PIC)*
- Que se passe-t-il si on fait référence à des données statiques?
  - **`la`** correspond aux instructions **`lui`** et **`addi`**
  - Cela nécessite de connaitre l'adresse complète sur 32 bits
- Cela ne peut donc pas être connu à cet instant, on crée donc une table

Table des Symboles

- La **table des symboles** est une **table** qui indique à un instant donné à quoi correspondent les identificateurs
- Que sont les identificateurs?
  - Labels: appel de fonctions
  - Data: les éléments stockés dans la section **`.data`**

Relocation Table

- List of "items" whose address this file needs
  What are they?
  - Any absolute label jumped to: **`jal, jalr`**
    - Internal
    - External (including lib files)
    - Such as the **`la`** instruction
      E.g., for **`jalr`** base register
  - Any piece of data in static section
    - Such as the **`la`** instruction
      E.g., for **`lw/sw`** base register

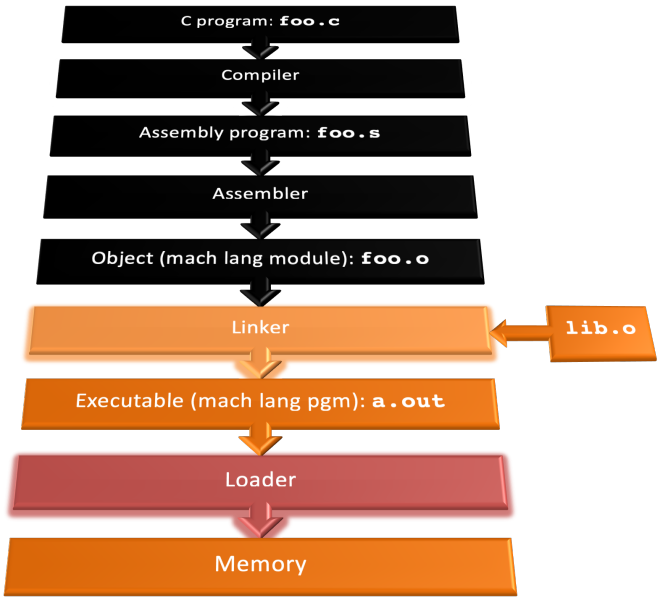
Object File Format

- object file header: size and position of the other pieces of the object file
- text segment: the machine code
- data segment: binary representation of the static data in the source file
- relocation information: identifies lines of code that need to be fixed up later
- symbol table: list of this file's labels and static data that can be referenced
- debugging information
- A standard format is ELF (except MS)
  `http://www.skyfree.org/linux/references/ELF_Format.pdf`

Quelques precisions sur les instructions et la compilation

- Format des Instructions et focus sur J/JAL
- Multiplication et Division
- Interpretation vs. Translation
- De l'assembleur à l'exécution

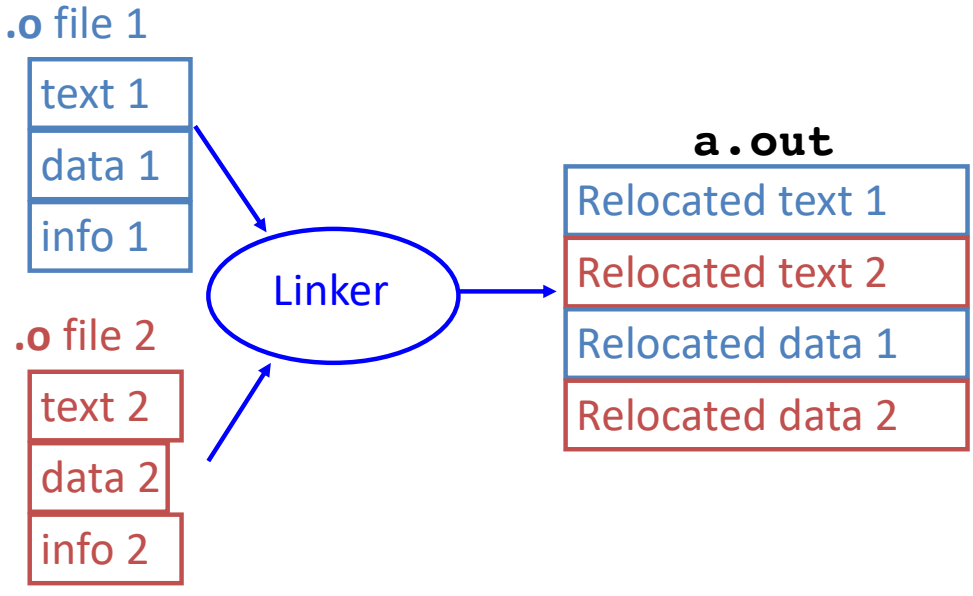
Du programme à l'exécution



Linker (1/3)

- Entrées: fichiers code objet, tables d'informations(e.g., `foo.o,libc.o` for RISC-V)
- sortie: code Exécutable (e.g., `a.out` for RISC-V)
- Combine plusieurs fichiers objet(`.o`) pour former un seul exécutable("linking")
- Cela permet la compilation séparée
  - Le changement d'un fichier ne nécessite pas de tout recompiler
    - Sources de Linux > 20 M lignes of code!

Linker (2/3)



Linker (3/3)

1. Extraction de chaque segment .text des fichiers `.o` file pour les réunir dans un seul
2. Extraction de chaque segment .data segment des fichiers `.o` pour les réunir dans un seul et les concaténer au segment.text créé précédemment
3. Résolution des références
   – Accès à chaque table pour résoudre les références.

4 Types d'adresse

- PC-Relative Addressing (**`beq`**, **`bne`**, **`jal; auipc/addi`**)
  – Pas besoin de resolution(PIC: position independent code)
- Absolute Function Address (**`auipc/jalr`**)
  – Besoin de résolution
- External Function Reference (**`auipc/jalr`**)
  – Besoin de résolution
- Static Data Reference (often **`lui/addi`**)
  – Besoin de résolution

Adressage absolu RISC-V

- Quelles instructions nécessitent de résoudre les adresses?
  - J-format: jump/jump and link

xxxxx	rd	jal

  - I-,S- Format: Loads and stores to variables in static area, relative to global pointer

xxx	gp		rd	lw

xx	rs1	gp		x	sw

  - What about conditional branches?

xx	rs1	rs2		x	beq bne

  - PC-relative addressing preserved even if code moves

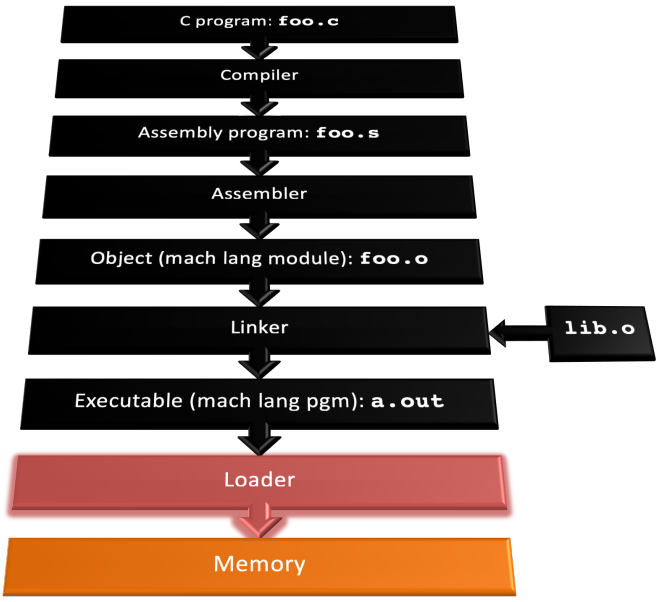
Résolution des références (1/2)

- Le Linker considére que le 1er mot du 1er segment text est à **0x10000**.
- Le Linker connait:
  - La longueur de chaque segment text et data
  - L'ordre des segments
- Le linker calcule :
  - Les addresses absolues de chaque label et des données qui sont référencées

Résolution des références(2/2)

- Pour résoudre les références:
  - Recherche des références dans les tables des symboles
  - Si non présent recherche dans les fichiers de librairies(e.g., pour **`printf`**)
  - Lorsque l'adresse absolue est déterminée, le code machine est complété
- Le linker génère le fichier exécutable

Du programme à l'execution



Le Loader

- Entrée: Code exécutable (e.g., **`a.out`** for RISC-V)
- Sortie: Le programme s'exécute
- Les fichiers exécutables sont stockés dans le disque
- Lorsqu'un exécutable est lancée, le job du loader est de le charger en mémoire et de le lancer.
- Dans les systèmes, le loader est le système d'exploitation (OS)
  - C'est une des taches de l'OS

Le loader … Qu'est ce qu'il fait?

- Lit l'entête de l'exécutable pour déterminer la taille des segments text et data.
- Crée un nouvel espace d'adressage suffisamment grand pour le programme pour stocker les segments de pile, data et text
- Copie les instructions et les données depuis le fichier exécutable dans l'espace d'adressage alloué.
- Copie les arguments passés au programme dans la pile
- Initialise les registres
- Saute à la routine de démarrage qui copie les arguments de la pile vers les registres et place la valeur du PC

Compiled `Hello.c`: `Hello.s`

```
.text                                  # Directive: enter text section
  .align 2                             # Directive: align code to 2^2 bytes
  .globl main                          # Directive: declare global symbol main
main:                                  # label for start of main
  addi sp,sp,-16                       # allocate stack frame
  sw   ra,12(sp)                       # save return address
  lui  a0,%hi(string1)                 # compute address of
  addi a0,a0,%lo(string1)              #   string1
  lui  a1,%hi(string2)                 # compute address of
  addi a1,a1,%lo(string2)              #   string2
  call printf                          # call function printf
  lw   ra,12(sp)                       # restore return address
  addi sp,sp,16                        # deallocate stack frame
  li   a0,0                            # load return value 0
  ret                                  # return
  .section .rodata                     # Directive: enter read-only data section
  .balign 4                            # Directive: align data section to 4 bytes
string1:                               # label for first string
  .string "Hello, %s!\n"               # Directive: null-terminated string
string2:                               # label for second string
  .string "world"                      # Directive: null-terminated string
```


Assembled `Hello.s`: Linkable `Hello.o`

```
00000000 <main>:
0:  ff010113 addi sp,sp,-16
4:  00112623 sw ra,12(sp)
8:  00000537 lui a0,0x0        # addr placeholder
c:  00050513 addi a0,a0,0      # addr placeholder
10: 000005b7 lui a1,0x0        # addr placeholder
14: 00058593 addi a1,a1,0      # addr placeholder
18: 00000097 auipc ra,0x0      # addr placeholder
1c: 000080e7 jalr ra           # addr placeholder
20: 00c12083 lw ra,12(sp)
24: 01010113 addi sp,sp,16
28: 00000513 addi a0,a0,0
2c: 00008067 jalr ra
```

Linked `Hello.o`: `a.out`

```
000101b0 <main>:
  101b0: ff010113 addi sp,sp,-16
  101b4: 00112623 sw   ra,12(sp)
  101b8: 00021537 lui  a0,0x21
  101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
  101c0: 000215b7 lui  a1,0x21
  101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
  101c8: 288000ef jal  ra,10450    # <printf>
  101cc: 00c12083 lw   ra,12(sp)
  101d0: 01010113 addi sp,sp,16
  101d4: 00000513 addi a0,0,0
  101d8: 00008067 jalr ra
```


Quelques precisions sur les instructions et la compilation

- Format des Instructions et focus sur J/JAL
- Multiplication et Division
- Interpretation vs. Translation
- De l'assembleur à l'exécution

En conclusion, …

- Le compilateur convertit un fichier en langage de haut niveau en un fichier assembleur
- L'assemblage supprime les pseudo instructions pour les convertir tout ce qui est possible en langage machine et crée les tables pour le linker
  - 2 passes sont nécessaires pour résoudre les adresses
- Le Linker assemble plusieurs fichiers .o et résout les adresses.
  - Cela permet la compilation séparée et résout les derniers problèmes d'adressage
- Le Loader charge l'exécutable en mémoire et débute l'exécution