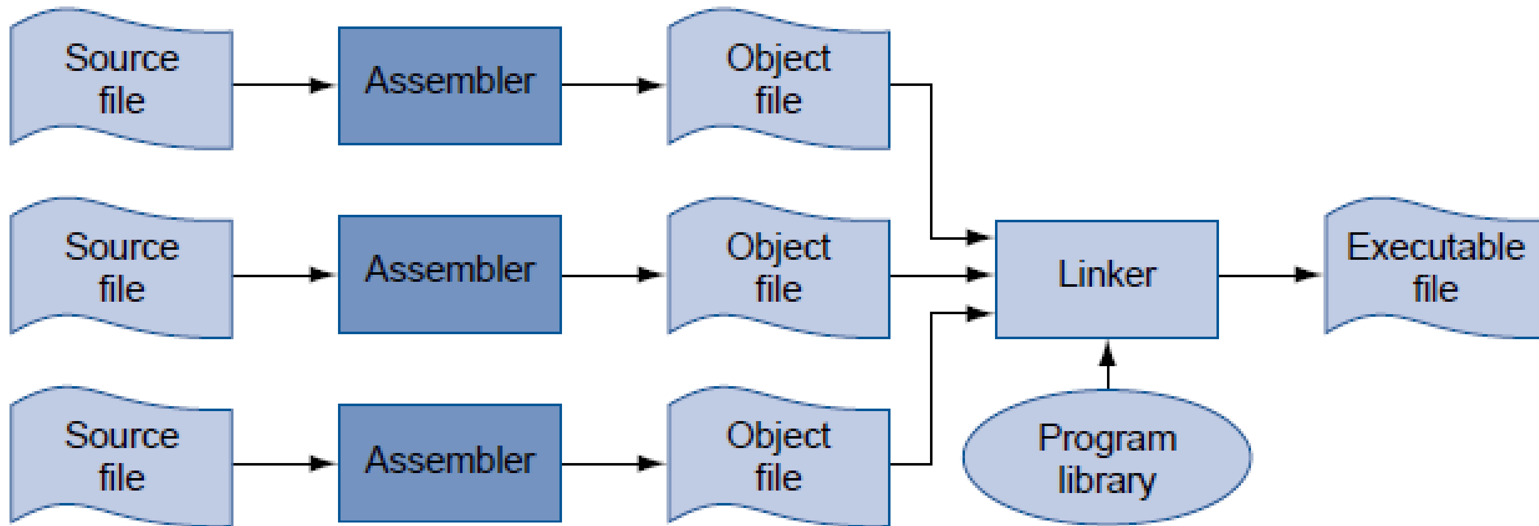
Processo di Trasformazione

Riprendiamo e dettagliamo il processo di trasformazione…



Gli assemblatori (o assembler)

- **Il processo di traduzione è costituito da due "passate":**
  1. traduzione in indirizzi delle etichette
  2. traduzione delle istruzioni assembly (codice operativo, identificatori di registri, etichette) in istruzioni macchina

- **Prima passata → Tabella dei Simboli (ST – Symbol Table)**
  - *Etichette (label)* definite in un modulo:
    - esterne (o globali): visibili anche all'esterno
    - locali: visibili solo all'interno

- **Seconda passata: file oggetto**
  - non ancora eseguibile: etichette definite in altri moduli

1^ Passata

1. Legge ciascuna linea del codice e la scompone in parti elementari dette ***lexemi*** (o ***lessemi***: parole singole, numeri e caratteri di interpunzione)
   - Es: ble $t0 , 100 , loop contiene sei lexemi
2. Se una linea comincia con una label, registra nella **ST** il nome della label e l'indirizzo che l'istruzione corrente occupa;
3. Poi calcola quante word di memoria l'istruzione corrente occupa;
   - facile per istruzioni a lunghezza fissa come in MIPS
4. Effettua un calcolo simile per le direttive dati;
5. Quando raggiunge la fine del file assembly, la **ST** registra la locazione di ogni label definita *nel modulo*, e le label non definite

2^ Passata

Usa la **ST** per produrre effettivamente il codice oggetto.

Esamina ancora ciascuna linea del file:

1. Se la linea contiene un'istruzione, combina il suo *opcode* e gli operandi (registri o locazioni di memoria) in un'istruzione legale binaria
2. Istruzioni e dati che si riferiscono a un simbolo definito in un altro modulo non possono essere assemblati completamente (rimangono *non risolti*)
   - NB: l'assembler non si lamenta dei riferimenti non risolti: è probabile che le label corrispondenti siano definite in un altro modulo.

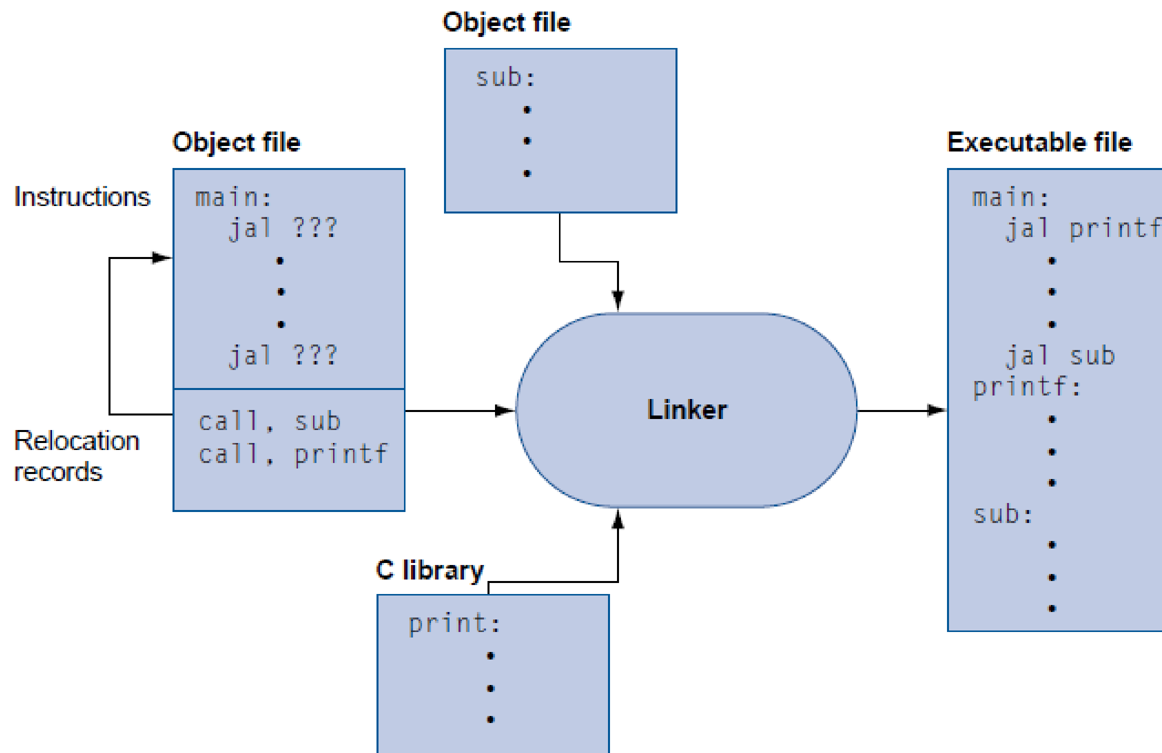
Il formato del file oggetto

Ha 6 parti:

1. **Object file header** descrive dimensione e posizione degli altri pezzi del file oggetto stesso;
2. **Text segment** contiene il codice macchina per le routine definite nel file sorgente (in generale non eseguibili, per via di riferimenti non risolti)
3. **Data segment** contiene la rappresentazione binaria dei dati nel file sorgente
4. **Informazione di rilocazione** identifica istruzioni e dati che dipendono da *indirizzi assoluti*.
   - Indirizzi assoluti (istruzioni `j` e `jal`): da rilocare sempre
   - Indirizzi relativi al PC (istruzioni `beq` e `bne`): da NON rilocare
5. **Symbol Table** associa un indirizzo alle etichette definite nel file; inoltre elenca i riferimenti non risolti.
6. **Debugging information**: descrizioni concisa di come sono stati compilati i moduli.

Linker

- **Compilazione separata => diversi moduli**
- **Il linker unisce i moduli:**
  - risolve i riferimenti tra moduli, confrontando le loro **ST**
  - cerca nelle librerie le procedure usate dal programma
  - riloca i file oggetto



Loader

- **Loader:** programma di sistema che carica un programma oggetto nella memoria principale per poter essere eseguito.
- **Il loader (in Unix):**
  1. lettura dell'intestazione del file eseguibile per determinare la dimensione dei segmenti testo e dati;
  2. creazione di uno spazio di indirizzamento (locazioni contigue in memoria) abbastanza grande da contenere i segmenti testo e dati, ed un segmento per lo stack;
  3. copia delle istruzioni e dei dati dal file eseguibile in memoria all'interno del nuovo spazio di indirizzamento;
  4. copia nello stack degli eventuali parametri passati al programma principale;
  5. inizializzazione dei registri dell'elaboratore (es. stack pointer,…);
  6. salto ad una procedura di inizializzazione (start-up routine) che copia gli argomenti del programma dallo stack ai registri e che chiama la procedura principale del programma (main). Quando main termina, la start-up routine conclude il programma (`exit`).

Architetture alternative

- **Alternativa di progetto: istruzioni più potenti (CISC)**
  - Scopi: ridurre il numero di istruzioni eseguite, assembly più facile
  - Pericoli: ciclo di clock più lungo, cicli/istruzione più alto

- **“RISC vs. CISC”**
  - dal 1982, la maggior parte dei processori sono stati RISC
  - VAX (CISC): minimize code size, make assembly language easy

    instructions from 1 to 54 bytes long!

- **Panoramica su 80x86 (CISC)**

80x86

- **1978: The Intel 8086 is announced (16 bit architecture)**
- **1980: The 8087 floating point coprocessor is added**
- **1982: The 80286 increases address space to 24 bits, +instructions**
- **1985: The 80386 extends to 32 bits, new addressing modes**
- **1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)**
- **1997: MMX is added**
- **… (*see your textbook for a more detailed description*)...**
- **…**
  - "This history illustrates the impact of the "golden handcuffs" of compatibility"

    "adding new features as someone might add clothing to a packed bag"

    "an architecture that is difficult to explain and impossible to love"

A dominant architecture: 80x86

- **Complexity:**
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
  - e.g., “base or scaled index with 8 or 32 bit displacement”
- **Saving grace:**
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow
- ***“what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective”***

Riassumendo

- **Instruction complexity is only one variable**
  - lower instruction count vs. higher CPI / lower clock rate
- **Design Principles:**
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast