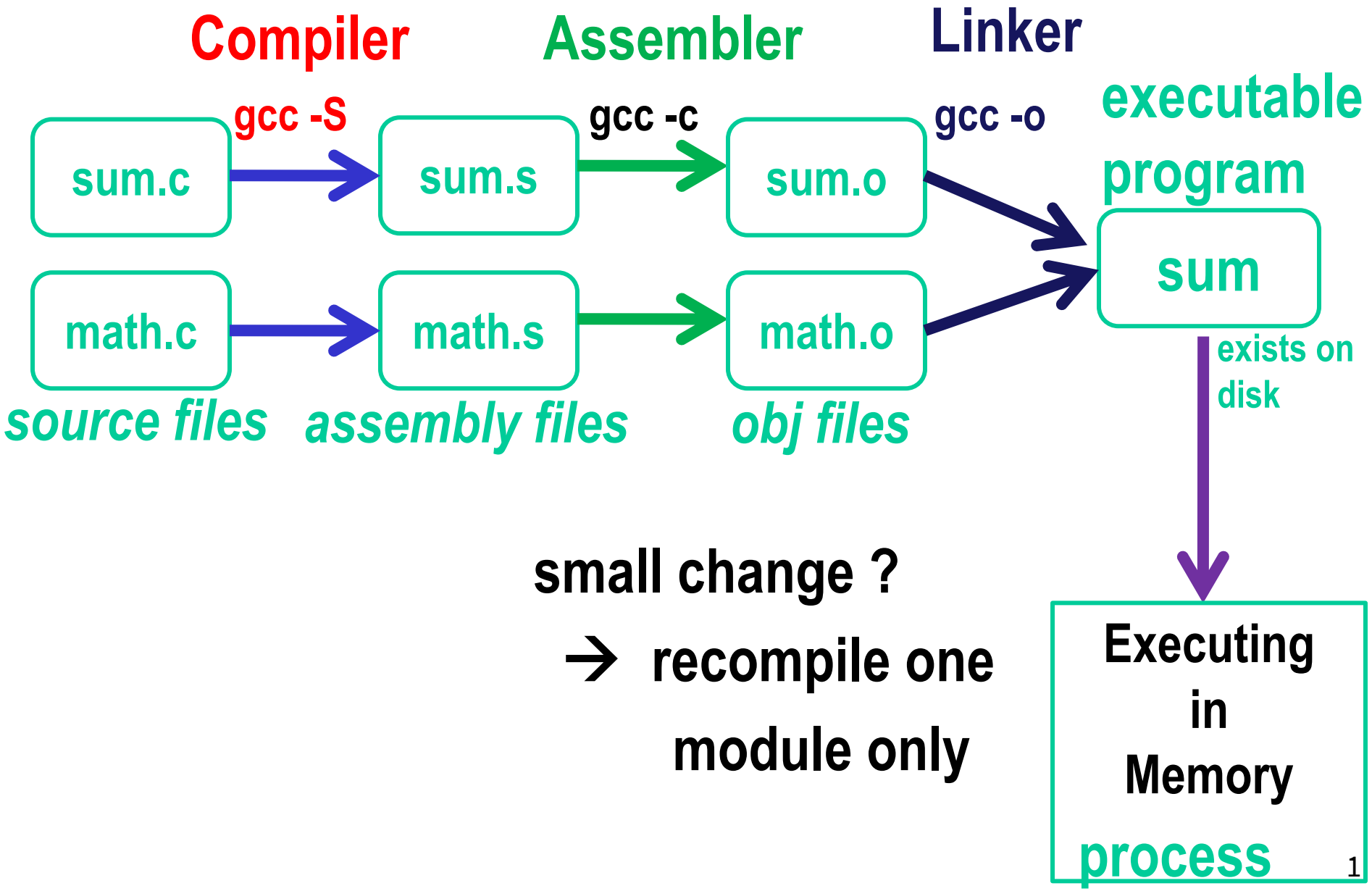
Compilation & Assembly



Example: sum.c

```
#include <stdio.h>
int n = 100;
int main (int argc, char* argv[ ]) {
    int i;
    int m = n;
    int sum = 0;

    for (i = 1; i <= m; i++) {
        sum += i;
    }
    printf ("Sum 1 to %d is %d\n", n, sum);
}
```

Compiler

Input: Code File (.c)

- Source code
- #includes, function declarations & definitions, global variables, *etc.*

Output: Assembly File (RISC-V)

- RISC-V assembly instructions

  (.s file)

```
for (i = 1; i <= m; i++) {
    sum += i;
}
```

```
li   x2,1
lw   x3,fp,28
slt  x2,x3,x2
```

sum.s (abridged)

```
        .globl  n
        .data
        .type    n, @object
n:      .word   100
        .rdata
$str0:  .string  "Sum 1 to %d is %d\n"
        .text
        .globl  main
        .type   main, @function
main:   addiu    $sp,$sp,-48
        sw       $ra,44($sp)
        sw       $fp,40($sp)
        move     $fp,$sp
        sw       $a0,-36($fp)   $a0
        sw       $a1,-40($fp)   $a1
        la       $a5,n
        lw       $a5,0($a5)     n=100
        sw       $a5,-28($fp)   m=n=100
        sw       $0,-24($fp)    sum=0
        li       $a5,1
        sw       $a5,-20($fp)   i=1
        lw       $a4,-20($fp)   i=1
        lw       $a5,-28($fp)   m=100
        blt      $a5,$a4,       if(m < i)
                                100 < 1
        lw       $a4,-24($fp)   0(sum)
        lw       $a5,-20($fp)   1(i)
        addu     $a5,$a4,$a5    1=(0+1)
        sw       $a5,-24($fp)   sum=1
        lw       $a5,-20($fp)   a5=i=1
        addi     $a5,$a5,1      i=2=(1+1)
        sw       $a5,-20($fp)   i=2
        j
        la   $a0 $4,$str0       str
        lw   $a1 $a1,-28($fp)   m=100
        lw   $a2 $a2,-24($fp)   sum
        jal      printf
        li       $a0,0          main returns 0
        mv       $sp,$fp
        lw       $ra,44($sp)
        lw       $fp,40($sp)
        addiu    $sp,$sp,48
        jr       $ra
```

prologue

call printf

epilogue

Assembler

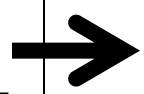
Input: Assembly File (.s)

- assembly instructions, pseudo-instructions
- program data (strings, variables), layout directives

Output: Object File in binary machine code RISC-V instructions in executable form

(.o file in Unix, .obj in Windows)

```
addi r5, r0, 10
muli r5, r5, 2
addi r5, r5, 15
```



```
00000000101000000000010100010011
00000000000100010100000101000000
00000000111100101000001010010011
```

Assembling Programs

```
    .text
    .ent main
main: la $4, Larray
    li $5, 15
    ...
    li $4, 0
    jal exit
    .end main
    .data
Larray:
    .long 51, 491, 3991
```

- Assembly files consist of a mix of
- + instructions
- + pseudo-instructions
- + assembler (data/layout) directives
  (Assembler lays out binary values in memory based on directives)
- Assembled to an Object File
  - Header
  - Text Segment
  - Data Segment
  - Relocation Information
  - Symbol Table
  - Debugging Information

Object file

Object File

Header

- Size and position of pieces of file

Text Segment

- instructions

Data Segment

- static data (local/global vars, strings, constants)

Debugging Information

- line number → code address map, *etc.*

Symbol Table

- External (exported) references
- Unresolved (imported) references

math.c

Symbols and References

```
int pi = 3;
int e = 2;
static int randomval = 7;

extern int usrid;
extern int printf(char *str, …);

int square(int x) { … }
static int is_prime(int x) { … }
int pick_prime() { … }
int get_n() {
        return usrid;
}
```

(extern == defined in another file)

Global labels: Externally visible "exported" symbols

- Can be referenced from other object files
- Exported functions, global variables
- Examples: pi, e, userid, printf, pick_prime, pick_random

Local labels: Internally visible only

- Only used within this object file
- static functions, static variables, loop labels, …
- Examples: randomval, is_prime

Two Passes Overview

- Pass 1:
  - Expands pseudo instructions encountered
  - Remember position of labels
  - Take out comments, empty lines, etc
  - Error checking
- Pass 2:
  - Use label positions to generate relative addresses (for branches and jumps)
  - Outputs the object file, a collection of instructions in binary code

Handling forward references

Example:

```
        bne x1, x2, L
        sll x0, x0, 0
L:  addi x2, x3, 0x2
```

The assembler will change this to

```
bne x1, x2, +8
sll x0, x0, 0
addi x2, x3, 0x2
```

Final machine code

```
0X00208413 # bne
0x00001033 # sll
0x00018113 # addi
```

actually:

```
0000 0000 0010...
0000 0000 0000...
0000 0000 0000...
```

Object File Formats

Unix

- a.out
- COFF: Common Object File Format
- ELF: Executable and Linking Format

Windows

- PE: Portable Executable

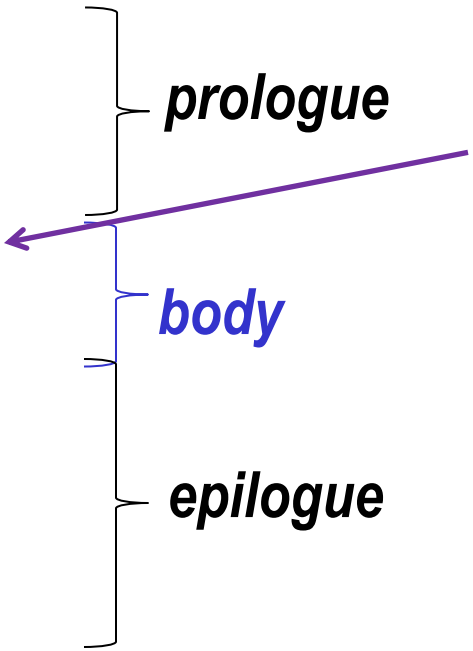
All support both executable and object files

Objdump disassembly

```
> riscv32-unknown-elf--objdump --disassemble math.o
```

Disassembly of section .text:

```
00000000 <get_n>:
   0:   27bdfff8  addi sp,sp,-8
   4:   afbe0000  sw   fp,0(sp)
   8:   03a0f021  mv   fp,sp
   c:   3c020000  lui  a0,
  10:   8c420008  lw   a0,8(a0)
  14:   03c0e821  mv    sp,fp
  18:   8fbe0000  lw   fp,0(sp)
  1c:   27bd0008  addi sp,sp,8
  20:   03e00008  jr   ra
```



elsewhere in another file: int = 41;

```
int get_n() {
  return      ;
}
```

Objdump symbols

[O]bject
[l]ocal
[g]lobal

```
> riscv-unknown-elf--objdump --syms math.o
```

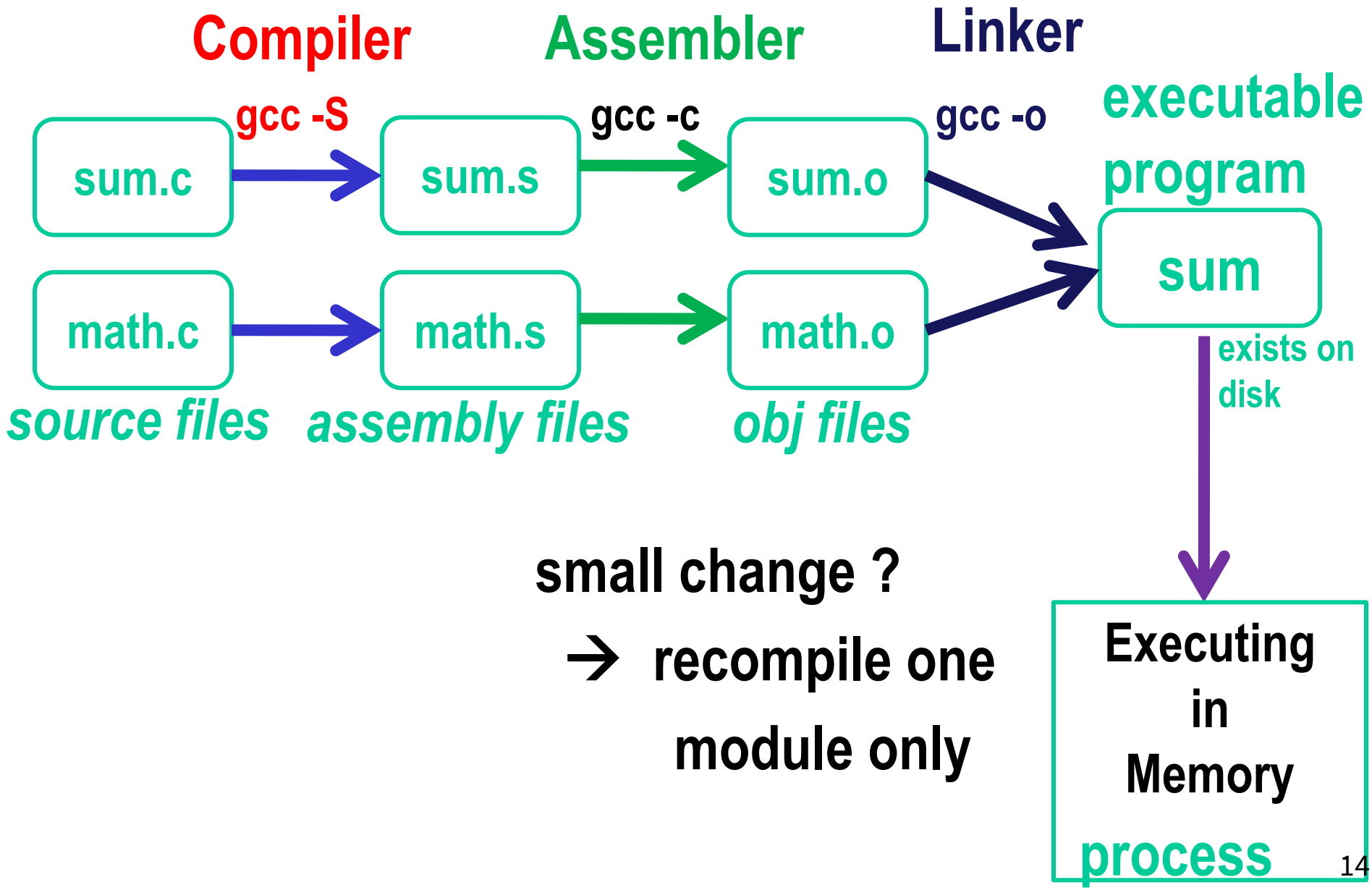
segment *size*

```
SYMBOL TABLE:
00000000 l  df *ABS*          00000000 math.c
00000000 l  d  .text          00000000 .text
00000000 l  d  .data          00000000 .data
00000000 l  d  .bss           00000000 .bss
00000008 l  O  .data          00000004 randomval
00000060 l     .text          00000028 is_prime
00000000 l  d  .rodata        00000000 .rodata
00000000 l  d  .comment       00000000 .comment
00000000 g  O  .data          00000004 pi
00000004 g  O  .data          00000004 e
00000000 g     .text          00000028 get_n
00000028 g     .text          00000038 square
00000088 g     .text          0000004c pick_prime
00000000       *UND*          00000000 usrid
00000000       *UND*          00000000 printf
```

static local fn
@ addr 0x60
size = 0x28 bytes

external references (undefined)

Separate Compilation & Assembly

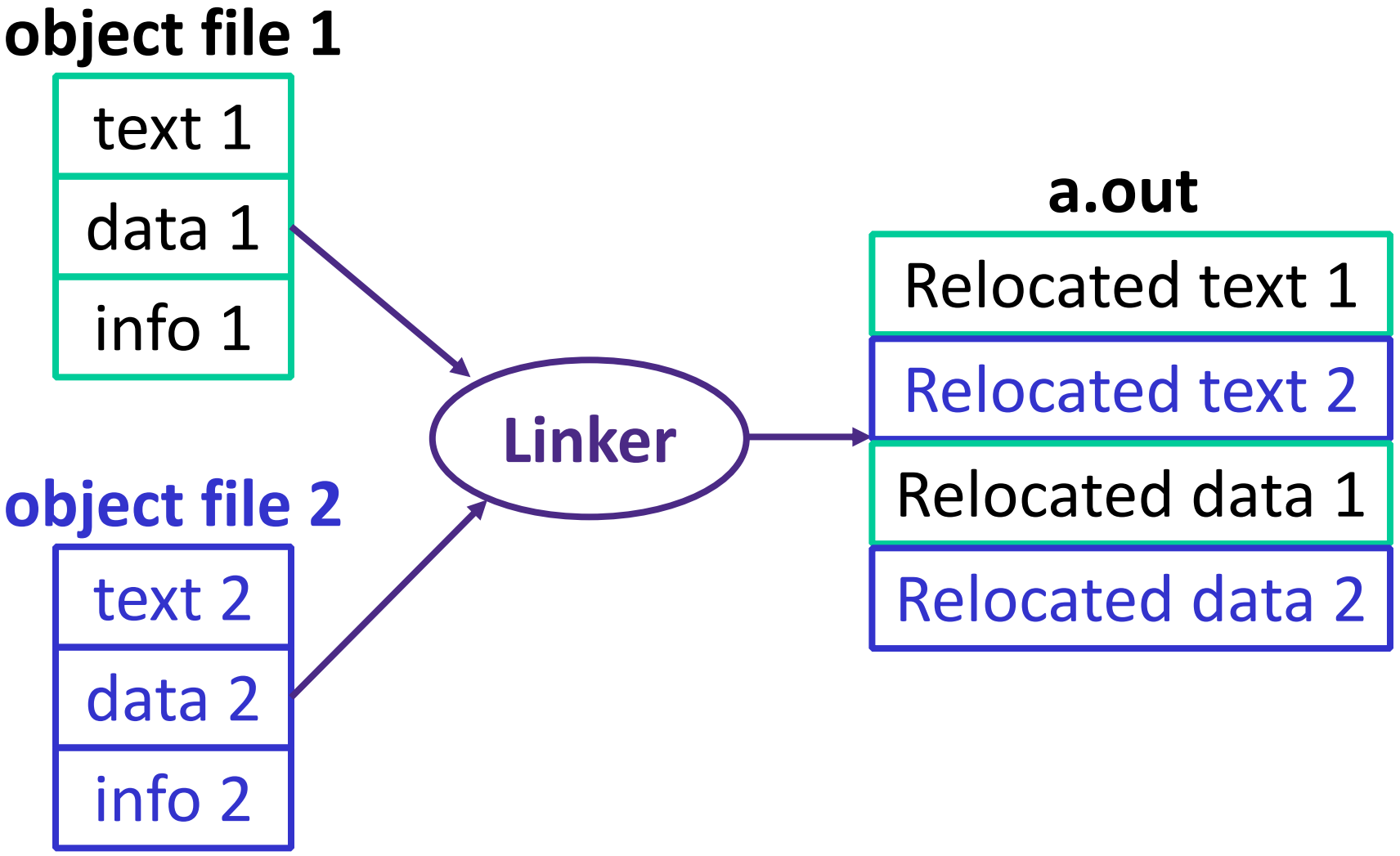


Linker (1/3)

- **Input:** Object Code files, information tables (e.g. `foo.o`, `lib.o` for RISC-V)
- **Output:** Executable Code (e.g. `a.out` for RISC-V)

- Combines several object (`.o`) files into a single executable ("linking")
- Enables separate compilation of files
  - Changes to one file do not require recompilation of whole program
  - Old name "Link Editor" from editing the "links" in jump and link instructions

Linker (2/3)


object file 1
text 1
data 1
info 1
object file 2
text 2
data 2
info 2
Linker
a.out
Relocated text 1
Relocated text 2
Relocated data 1
Relocated data 2

Linker (3/3)

1) Take text segment from each .o file and put them together
2) Take data segment from each .o file, put them together, and concatenate this onto end of text segments
3) Resolve References
   - Go through Relocation Table; handle each entry
   - i.e. fill in all absolute addresses

Resolving References (1/2)

- Linker assumes the first word of the first text segment is at 0x10000 for RV32.
  - More later when we study "virtual memory"
- Linker knows:
  - Length of each text and data segment
  - Ordering of text and data segments
- Linker calculates:
  - Absolute address of each label to be jumped to (internal or external) and each piece of data being referenced

Resolving References (2/2)

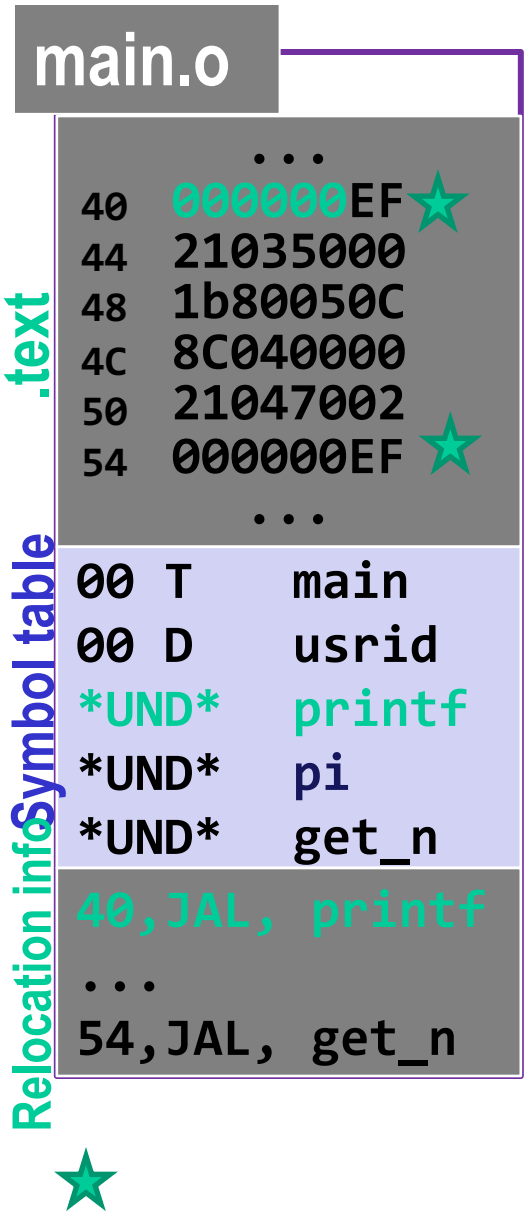
- To resolve references:
  1) Search for reference (data or label) in all "user" symbol tables
  2) If not found, search library files (e.g. `printf`)
  3) Once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

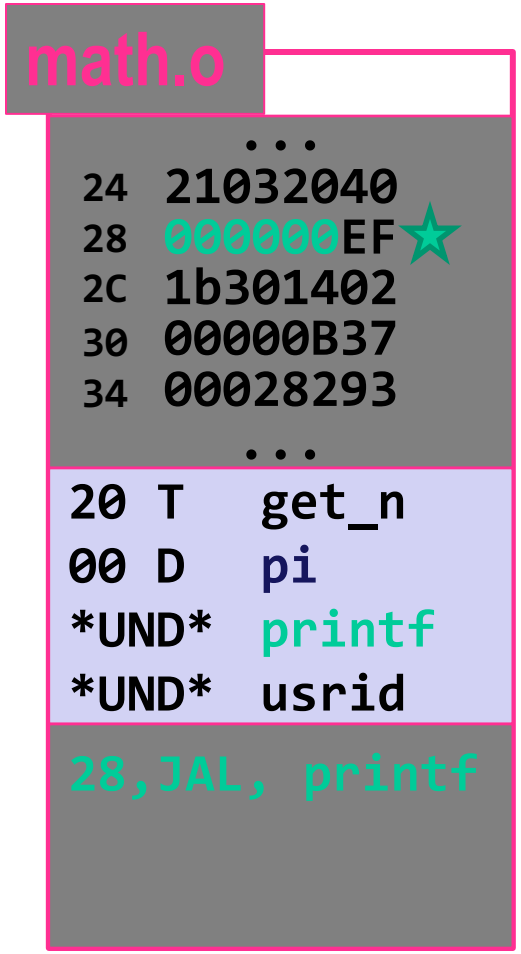
Three Types of Addresses

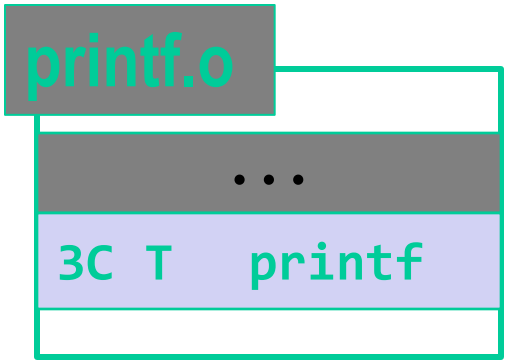
- PC-Relative Addressing (beq, bne, jal)
  - never relocate

External Function Reference (usually jal)
  - always relocate

Static Data Reference (often auipc and addi)
  - always relocate
  - RISC-V often uses auipc rather than lui so that a big block of stuff can be further relocated as long as it is fixed relative to the pc



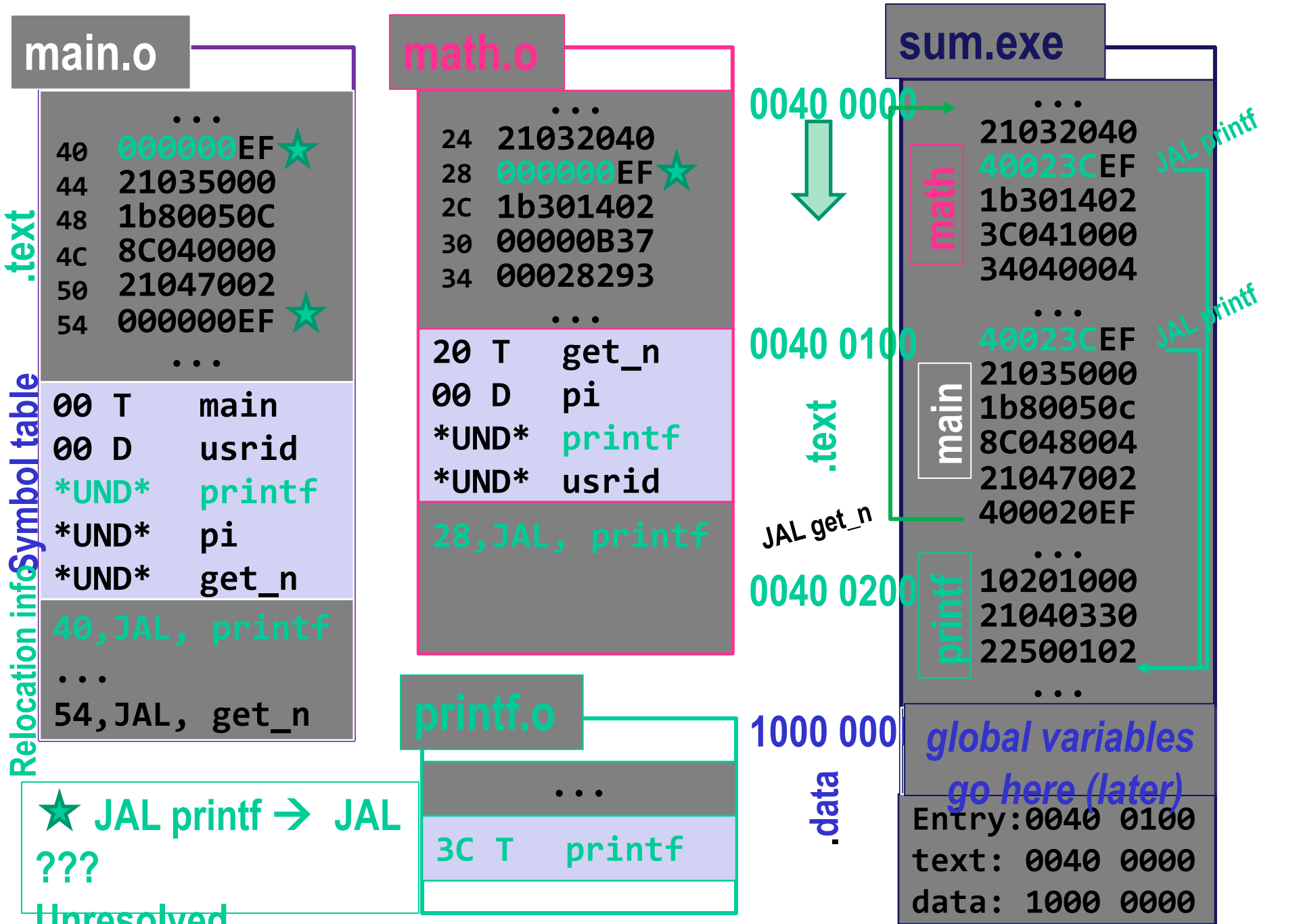




Which symbols are undefined according to **both** main.o and math.o's symbol table?

A) printf
B) pi
C) get_n
D) usr
E) printf & pi

JAL printf → JAL ???
Unresolved references to printf and get_n


main.o
...
40 000000EF
44 21035000
48 1b80050C
4C 8C040000
50 21047002
54 000000EF
...
.text
00 T main
00 D usrid
UND printf
UND pi
UND get_n
Symbol table
40,JAL, printf
...
54,JAL, get_n
Relocation info
math.o
...
24 21032040
28 000000EF
2C 1b301402
30 00000B37
34 00028293
...
20 T get_n
00 D pi
UND printf
UND usrid
28,JAL, printf
printf.o
...
3C T printf
JAL printf → JAL ???
Unresolved
sum.exe
0040 0000
0040 0100
0040 0200
1000 000
.text
.data
math
main
printf
JAL get_n
JAL printf
JAL printf
...
21032040
40023CEF
1b301402
3C041000
34040004
...
40023CEF
21035000
1b80050c
8C048004
21047002
400020EF
...
10201000
21040330
22500102
...
global variables go here (later)
Entry:0040 0100
text: 0040 0000
data: 1000 0000

Static Libraries

Static Library: Collection of object files
(think: like a zip archive)

Q: Every program contains the entire library?!?

A: No, Linker picks only object files needed to resolve undefined references at link time

e.g. libc.a contains many objects:

- printf.o, fprintf.o, vprintf.o, sprintf.o, snprintf.o, …
- read.o, write.o, open.o, close.o, mkdir.o, readdir.o, …
- rand.o, exit.o, sleep.o, time.o, ….

Question

Where does the assembler place the following symbols in the object file that it creates?

A. Text Segment
B. Data Segment
C. Exported reference in symbol table
D. Imported reference in symbol table
E. None of the above

```
#include <stdio.h>
#include heaplib.h

#define HEAP SIZE 16
static int ARR SIZE = 4;

int main() {
   char heap[HEAP SIZE];
   hl_init(heap, HEAP SIZE * sizeof(char));
   char* ptr = (char *) hl alloc(heap, ARR SIZE * sizeof(char));
   ptr[0] = 'h';
   ptr[1] = 'i';
   ptr[2] = '\0';
   printf(%s\n, ptr); return 0;
}
```

Q1: HEAP_SIZE
Q2: ARR_SIZE
Q3: hl_init

Summary

Compiler produces assembly files

(contain RISC-V assembly, pseudo-instructions, directives, etc.)

Assembler produces object files

(contain RISC-V machine code, missing symbols, some layout information, etc.)

Linker joins object files into one executable file

(contains RISC-V machine code, no missing symbols, some layout information)

Loader puts program into memory, jumps to 1st insn, and starts executing a *process* (machine code)

Peer Question

At what point in process are all the machine code bits determined for the following assembly instructions:

1) `add x6, x7, x8`

2) `jal x1, fprintf`

A: 1) & 2) After compilation

B: 1) After compilation, 2) After assembly

C: 1) After assembly, 2) After linking

D: 1) After assembly, 2) After loading

Peer Question

At what point in process are all the machine code bits determined for the following assembly instructions:

1) `add x6, x7, x8`

2) `jal x1, fprintf`

A: 1) & 2) After compilation

B: 1) After compilation, 2) After assembly

C: 1) After assembly, 2) After linking

D: 1) After assembly, 2) After loading