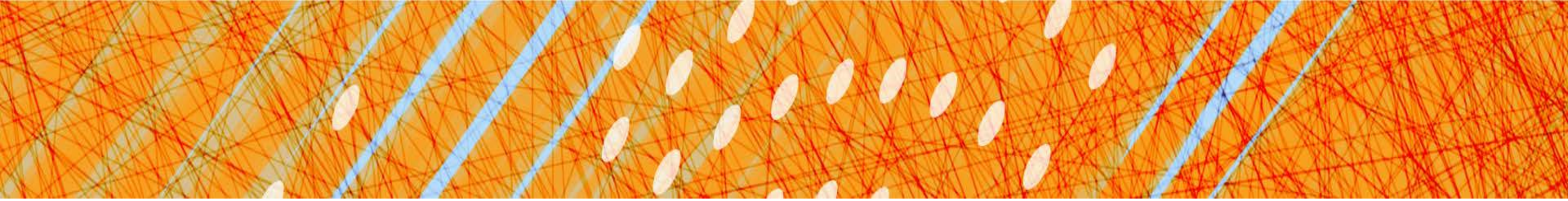


Assemblers, Linkers, and Loaders

Hakim Weatherspoon
CS 3410
Computer Science
Cornell University

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

[Weatherspoon, Bala, Bracy, and Sirer]

Big Picture: Where are we going?

C

compiler

RISC-V assembly

assembler

machine code

CPU

Circuits

Gates

Transistors

Silicon

```
int x = 10;
x = 2 * x + 15;
```

```
addi x5, x0, 10
muli x5, x5, 2
addi x5, x5, 15
```

x0 = 0
x5 = x0 + 10
x5 = x5<<1 #x5 = x5 * 2
x5 = x15 + 15

10 | r0 | r5 | op = addi

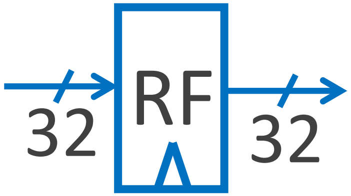
`000000001010 00000 000 00101 0010011`

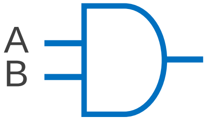
`00000000001000101000001010000000`

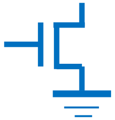
`000000001111 00101 000 00101 0010011`

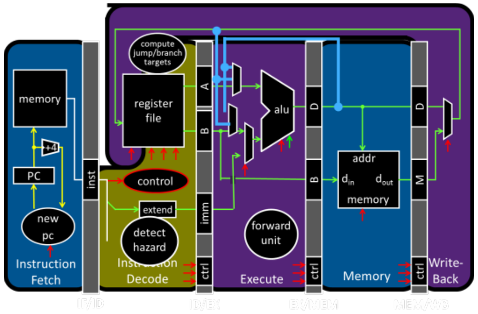
15 | r5 | r5 | op = addi

op = r-type | x5 | shamt=1 | x5 | func=sll









Big Picture: Where are we going?

C

compiler

RISC-V assembly

assembler

machine code

CPU

Circuits

Gates

Transistors

Silicon

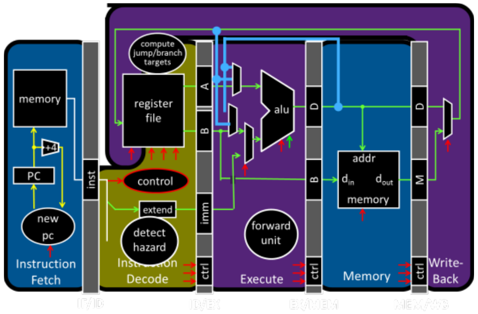
```
int x = 10;
x = 2 * x + 15;
```

High Level Languages

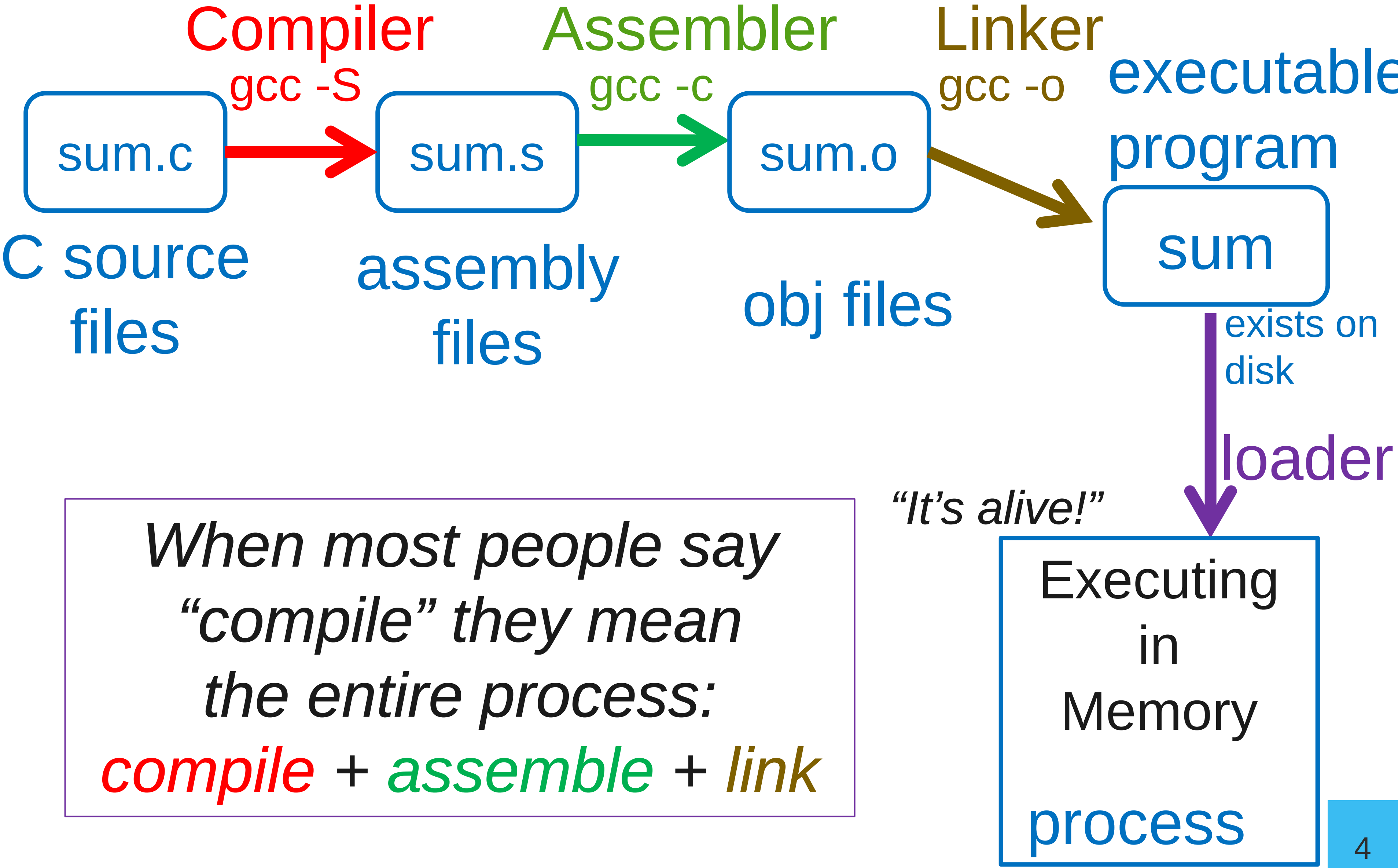
```
addi x5, x0, 10
muli x5, x5, 2
addi x5, x5, 15
```

```
00000000101000000000001010010011
00000000001000101000001010000000
00000000111100101000001010010011
```

Instruction Set Architecture (ISA)



From Writing to Running



When most people say "compile" they mean the entire process:
compile + *assemble* + *link*

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)
        sw       $a1,-40($fp)
        la       $a5,n
        lw       $a5,0($a5)
        sw       $a5,-28($fp)
        sw       $0,-24($fp)
        li       $a5,1
        sw       $a5,-20($fp)
```

```
$L2:    lw      $a4,-20($fp)
        lw      $a5,-28($fp)
        blt     $a5,$a4,$L3

        lw      $a4,-24($fp)
        lw      $a5,-20($fp)
        addu    $a5,$a4,$a5
        sw      $a5,-24($fp)
        lw      $a5,-20($fp)
        addi    $a5,$a5,1
        sw      $a5,-20($fp)
        j       $L2
$L3:    la      $4,$str0
        lw      $a1,-28($fp)
        lw      $a2,-24($fp)
        jal     printf
        li      $a0,0
        mv      $sp,$fp
        lw      $ra,44($sp)
        lw      $fp,40($sp)
        addiu   $sp,$sp,48
        jr      $ra
```

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
```

→

```
00000000101000000000001010010011
0000000000100010100000101
00000000111100101000001010010011
```

RISC-V Assembly Instructions

Arithmetic/Logical

- ADD, SUB, AND, OR, XOR, SLT, SLTU
- ADDI, ANDI, ORI, XORI, LUI, SLL, SRL, SLTI, SLTIU
- MUL, DIV

Memory Access

- LW, LH, LB, LHU, LBU,
- SW, SH, SB

Control flow

- BEQ, BNE, BLE, BLT, BGE
- JAL, JALR

Special

- LR, SC, SCALL, SBREAK

Pseudo-Instructions

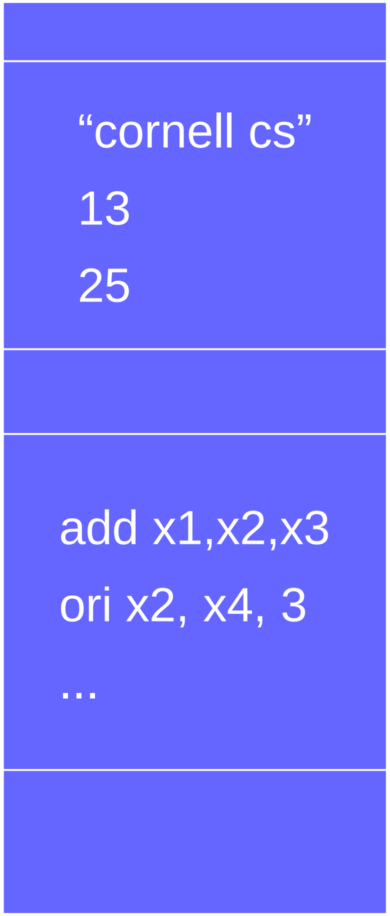
Assembly shorthand, technically not machine instructions, but easily converted into 1+ instructions that are

Pseudo-Insns	Actual Insns	Functionality
NOP	SLL x0, x0, 0	# do nothing
MOVE reg, reg	ADD r2, r0, r1	# copy between regs
LI reg, 0x45678	LUI reg, 0x4 ORI reg, reg, 0x5678	#load immediate
LA reg, label		# load address (32 bits)
B		# unconditional branch
BLT reg, reg, label	SLT r1, rA, rB BNE r1, r0, label	# branch less than

+ a few more…

Program Layout

- Programs consist of segments used for different purposes
  - Text: holds instructions
  - Data: holds statically allocated program data such as variables, strings, etc.



“cornell cs”

13

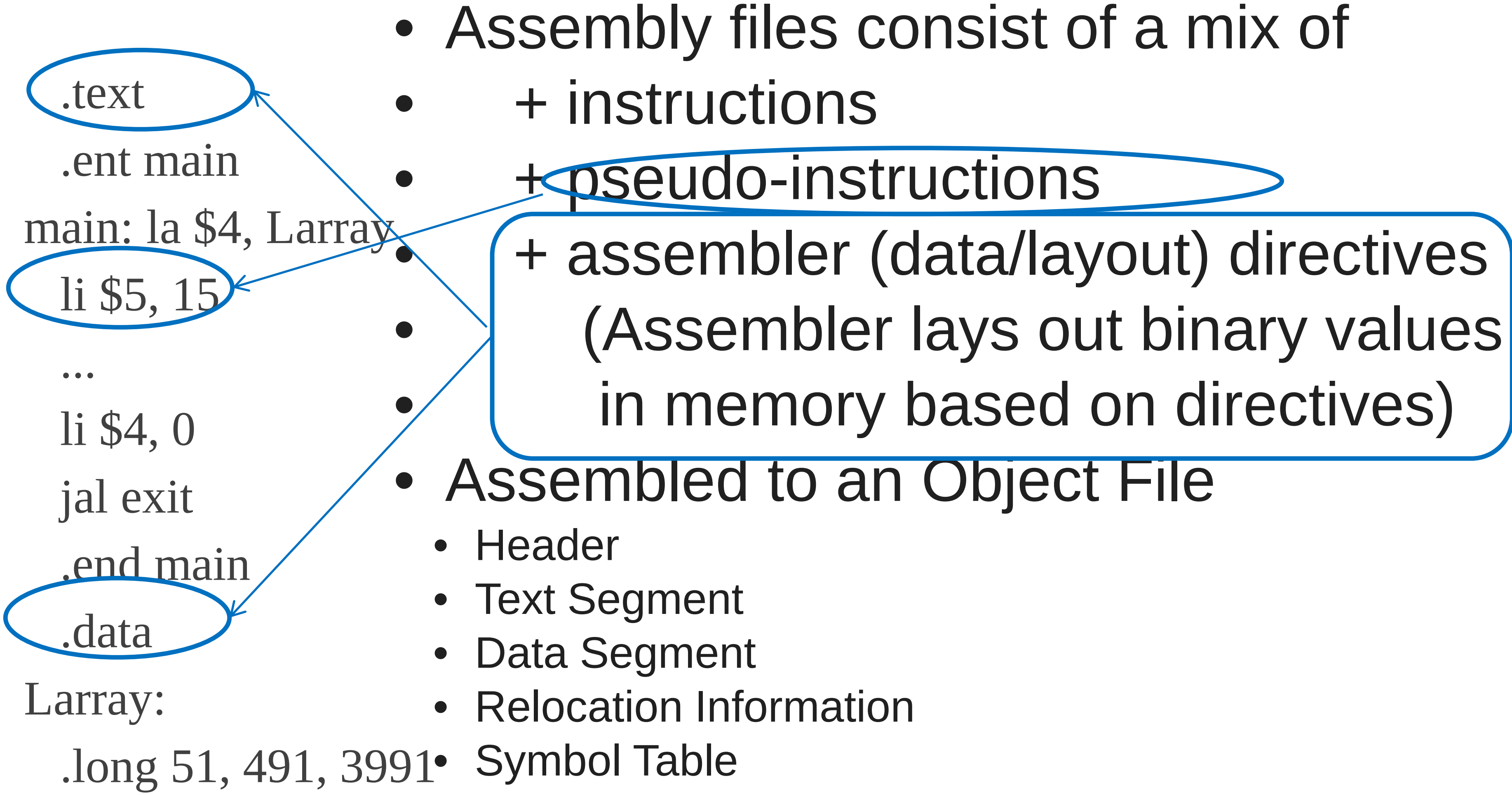
25

add x1,x2,x3

ori x2, x4, 3

...

Assembling Programs

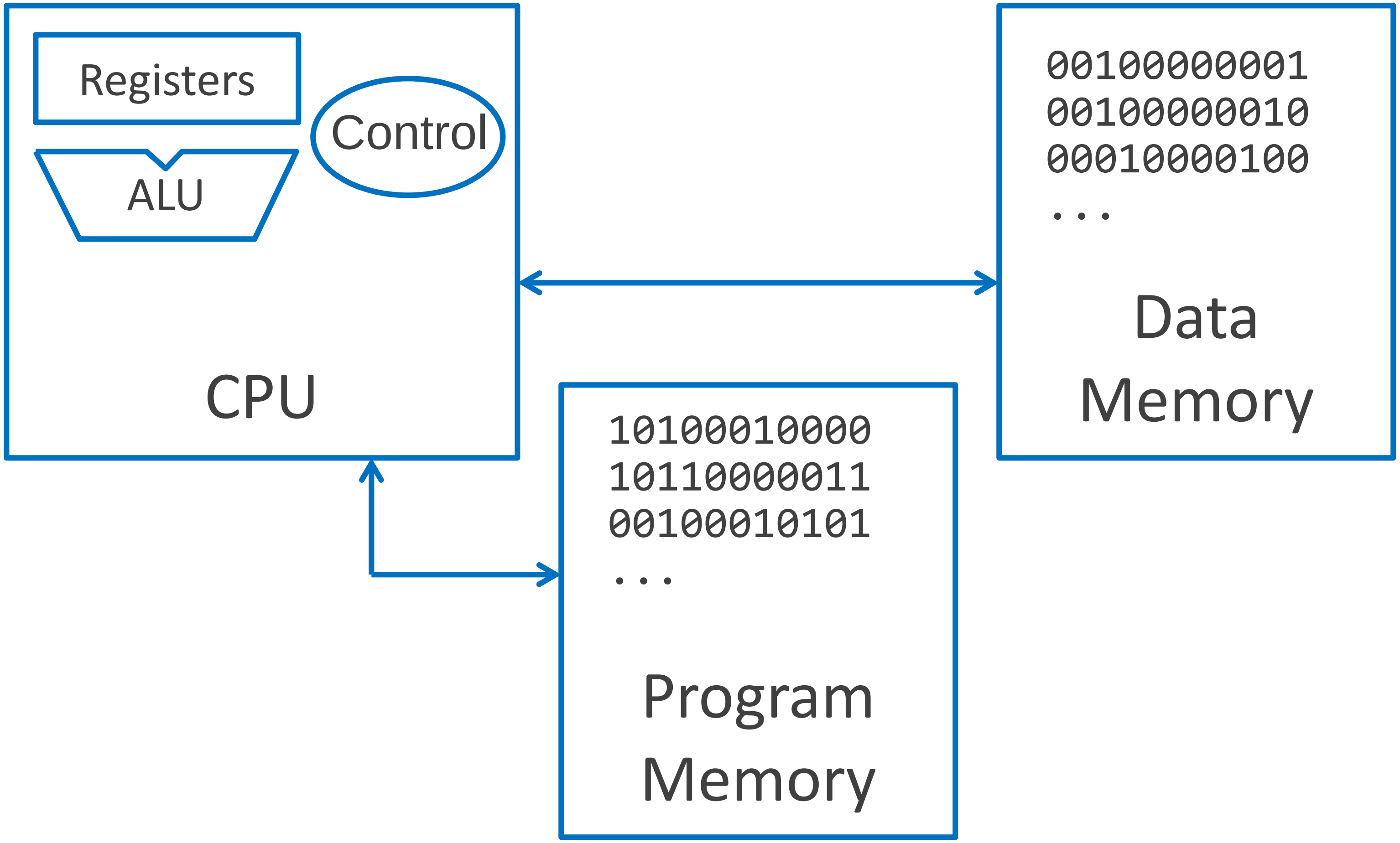


```
.text
.ent main
main: la $4, Larray
li $5, 15
...
li $4, 0
jal exit
.end main
.data
Larray:
.long 51, 491, 3991
```

- 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

Assembling Programs

- Assembly using a (modified) Harvard architecture
  - Need segments since data and program stored together in memory



Takeaway

- Assembly is a low-level task
  - Need to assemble assembly language into machine code binary. Requires

    - Assembly language instructions

    - *pseudo-instructions*

    - And Specify layout and data using *assembler directives*

  - Today, we use a modified Harvard Architecture (Von Neumann architecture) that mixes data and instructions in memory

    … but kept in separate *segments*

    … and has separate caches

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 symbols

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

Handling forward references

Example:

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

Looking for L

Found L

The assembler will change this to

```
bne x1, x2, +1
sll x0, x0, 0
addi x2, x3, 0x2
```

Final machine code

```
0X14220001 # bne      actually:  000101...
0x00000000 # sll                 000000...
0x24620002 # addiu               001001...
```

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

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

```
> mipsel-linux-objdump --disassemble math.o

Disassembly of section .text:

00000000 <get_n>:
   0:   27bdfff8  addiu sp,sp,-8
   4:   afbe0000  sw    s8,0(sp)
   8:   03a0f021  move  s8,sp
   c:   3c020000  lui   v0,0x0
  10:   8c420008  lw    v0,8(v0)
  14:   03c0e821  move  sp,s8
  18:   8fbe0000  lw    s8,0(sp)
  1c:   27bd0008  addiu sp,sp,8
  20:   03e00008  jr    ra
  24:   00000000  nop
```

elsewhere in another file: `int usrid = 41;`

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

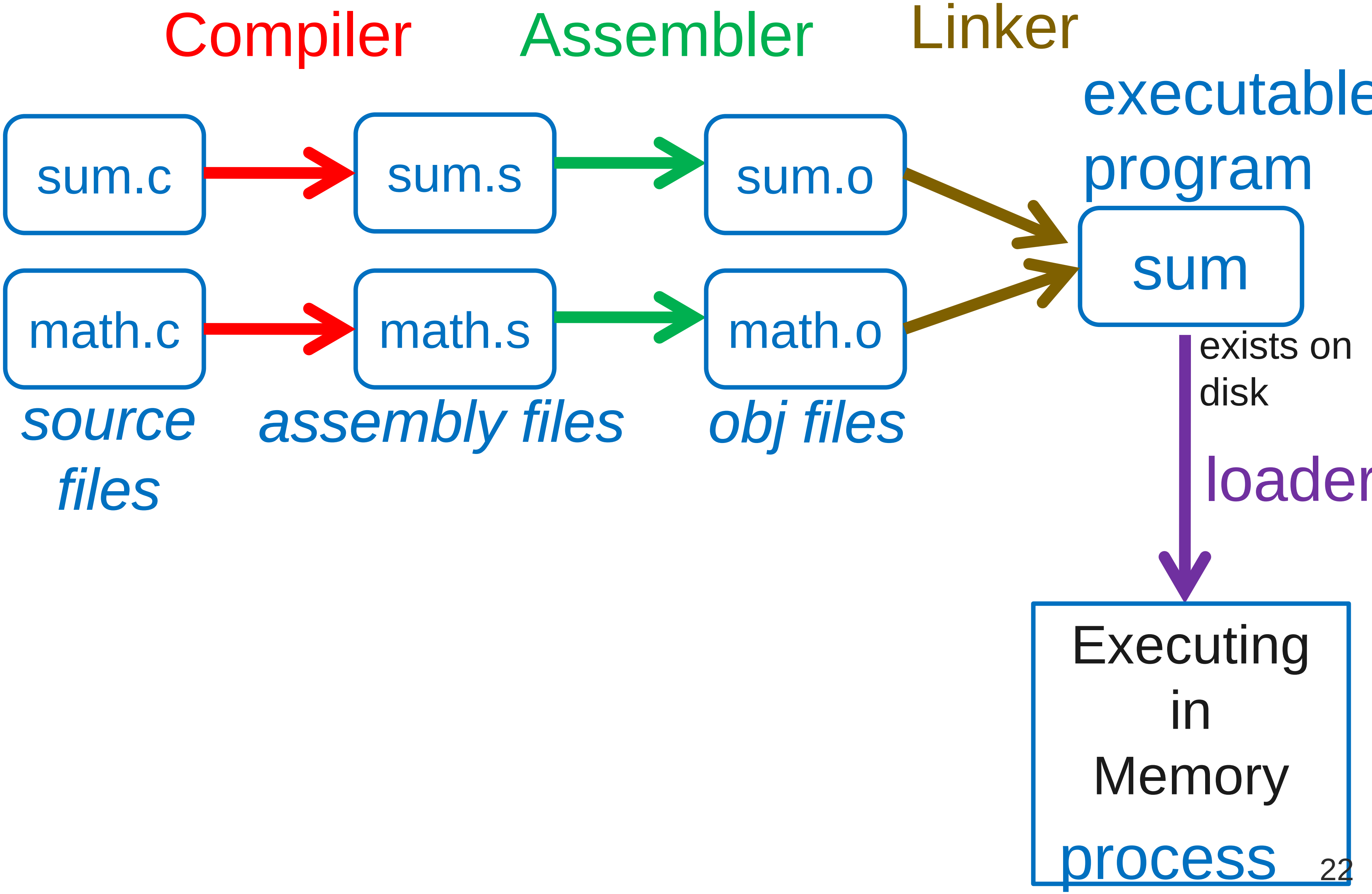
Objdump symbols

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

```
> mipsel-linux-objdump --syms math.o
```

```
SYMBOL TABLE:     segment        size
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  F  .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  F  .text        00000028 get_n
00000028 g  F  .text        00000038 square
00000088 g  F  .text        0000004c pick_prime
00000000       *UND*        00000000 usrid
00000000       *UND*        00000000 printf
```

Separate Compilation & Assembly



Linkers

Linker combines object files into an executable file

- Resolve as-yet-unresolved symbols
- Each has illusion of own address space
  - → Relocate each object's text and data segments
- Record top-level entry point in executable file

End result: a program on disk, ready to execute

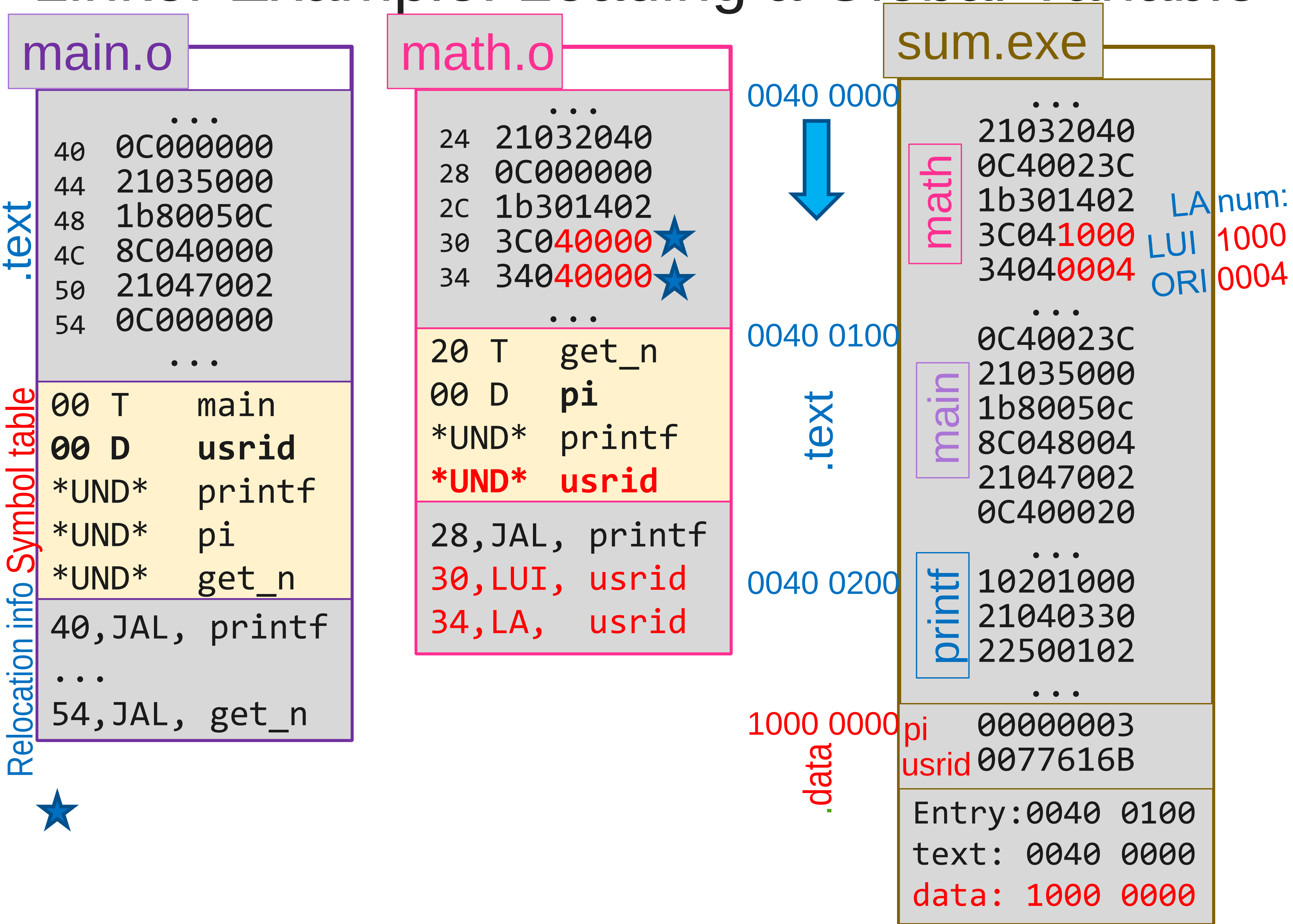
E.g.	./sum	Linux
	./sum.exe	Windows
	simulate sum	Class RISC-V simulator

Static Libraries

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

Q: Every program contains the entire library?!?

Linker Example: Loading a Global Variable



main.o

.text

```
   ...
40 0C000000
44 21035000
48 1b80050C
4C 8C040000
50 21047002
54 0C000000
   ...
```

Symbol table

```
00 T     main
00 D     usrid
*UND*    printf
*UND*    pi
*UND*    get_n
```

Relocation info

```
40,JAL, printf
...
54,JAL, get_n
```

★

math.o

```
   ...
24 21032040
28 0C000000
2C 1b301402
30 3C040000 ★
34 34040000 ★
   ...
```

```
20 T    get_n
00 D    pi
*UND*   printf
*UND*   usrid
```

```
28,JAL, printf
30,LUI, usrid
34,LA,  usrid
```

sum.exe

0040 0000

.text

```
...
21032040
0C40023C
1b301402
3C041000
34040004
...
```

LA num:
LUI 1000
ORI 0004

0040 0100

```
0C40023C
21035000
1b80050c
8C048004
21047002
0C400020
...
```

0040 0200

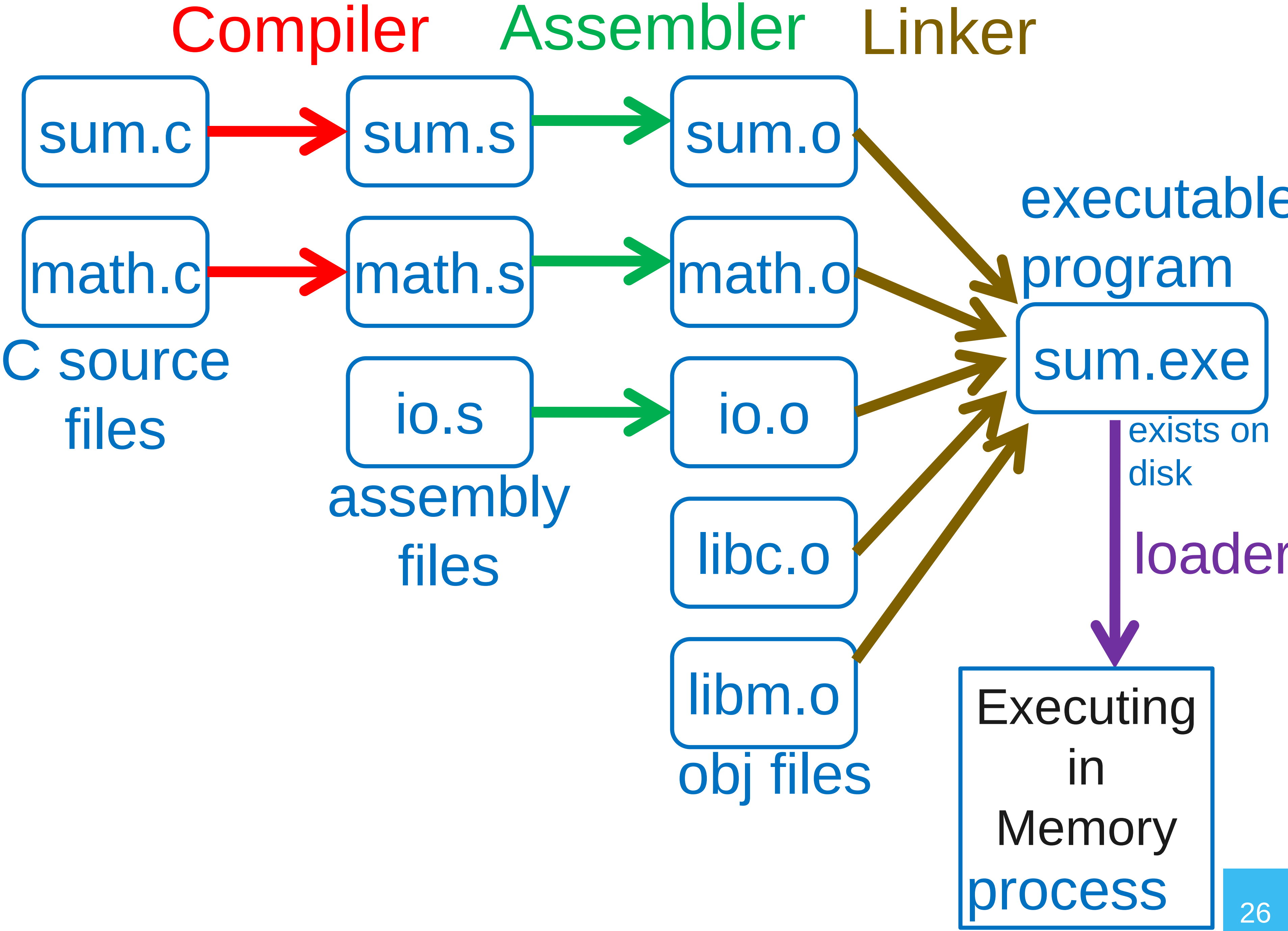
```
10201000
21040330
22500102
...
```

1000 0000

.data

```
pi     00000003
usrid  0077616B
```

```
Entry:0040 0100
text: 0040 0000
data: 1000 0000
```



Compiler

Assembler

Linker

sum.c

math.c

C source files

sum.s

math.s

io.s

assembly files

sum.o

math.o

io.o

libc.o

libm.o

obj files

executable program

sum.exe

exists on disk

loader

Executing in Memory

process

Loaders

Loader reads executable from disk into memory

- Initializes registers, stack, arguments to first function
- Jumps to entry-point

Part of the Operating System (OS)

Shared Libraries

Q: Every program contains parts of same library?!

Static and Dynamic Linking

Static linking

- Big executable files (all/most of needed libraries inside
- Don’t benefit from updates to library
- No load-time linking

Dynamic linking

- Small executable files (just point to shared library)
- Library update benefits all programs that use it
- Load-time cost to do final linking
  - But dll code is probably already in memory
  - And can do the linking incrementally, on-demand

Takeaway

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)