Universität Bremen

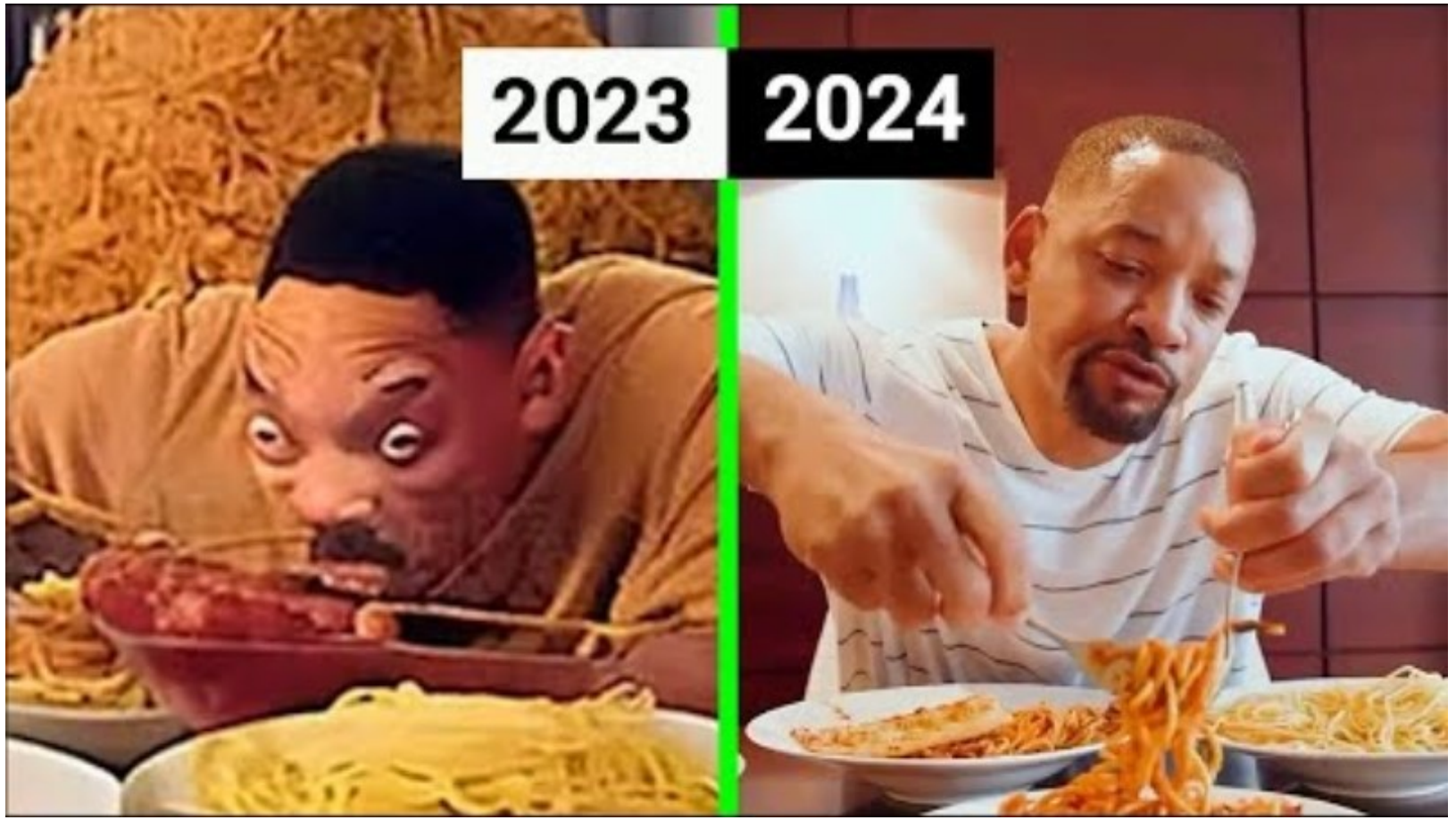
# From Software to Hardware

Prof. Dr. Rolf Drechsler
Dr. Muhammad Hassan
M.Sc. Jan Zielasko
M.Sc. Milan Funck

Universität Bremen

# Today's Agenda

- Translation vs Interpretation
- Compiler
- Assembler
  - Symbol table
  - Relocation Table
  - Object File Format
- Linker
  - Address Types
- Loader
- Hello World Example


2023
2024

# Nvidia Jetson Nano

# Example C Program – hello.c

## Compile hello.c → hello.s

gcc -S helloworld.c

```
#include <stdio.h>
int main()
{
    printf("Hello, %s\n“, "world");
    return 0;
}
```

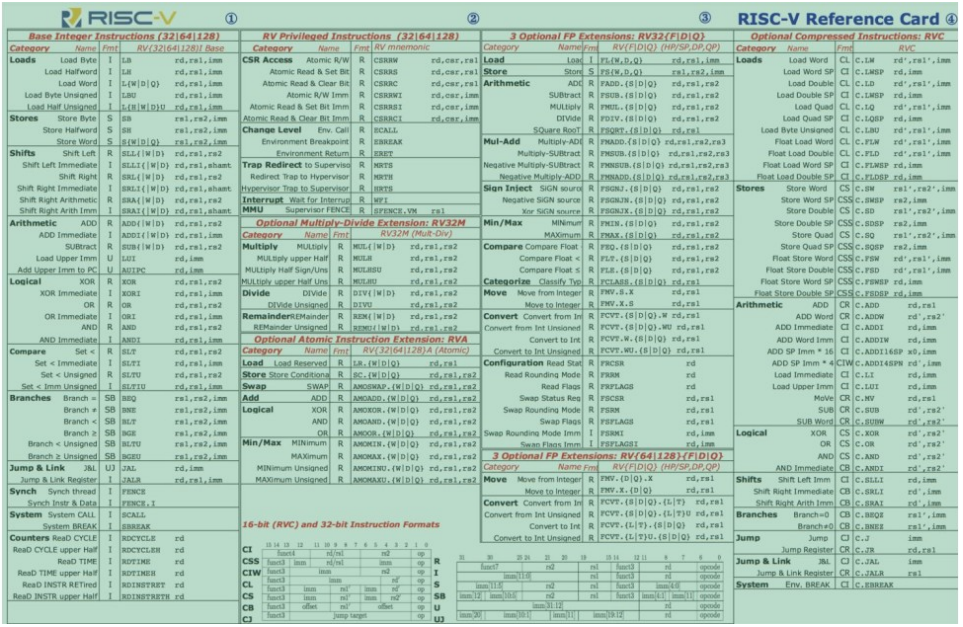
| | |
|---|---|
| **.text**<br>  **.align** 2<br>  **.globl main** | Directive: Enter **text** section<br>Directive: Align code to 2^2 bytes<br>Directive: Declare global symbol **main** |
| **main:**<br>  addi sp,sp,-4<br>  sw ra,0(sp)<br>  la a0, str1<br>  la a1, str2<br>  call printf<br>  lw ra,0(sp)<br>  addi sp,sp,4<br>  li a0,0<br>  Ret | Label for start of **main**<br>Allocate stack frame<br>save return address<br>Pseudoinstructions<br>load addresses of str1, str2<br>Pseudoinstruction: call function printf<br>Restore return address,<br>deallocate stack frame<br>Return with value 0<br>---- |
| **.section** .rodata<br>  **.balign** 4 | Directive: Enter read-only data section<br>Directive: Align data section to 4 bytes |
| str1:<br>  **.string** "Hello, %s!\n"<br>str2:<br>  **.string** "world" | Label for str1<br>Directive: null-terminated string<br>Label for str2<br>Directive: null-terminated string |

# Pseudoinstructions

- Convenient variations of instructions understood by the assembler, but not by the machine

| Pseudoinstructions | Read Instructions | |
|---|---|---|
| mv rd, rs1 | addi rd, rs1, 0 | |
| not rd, rs | xori ??, ??, ?? | |
| li rd, imm | # <= 12-bit signed imm | # >12-bit immediate |
| | addi ??, ??, ?? | lui ??, ??, ?? |
| | | addi ??, ??, ?? |
| j Label | jal ??, ?? | |
| jr rs1 | jalr ??, ?? | |
| la rd, label | | |
| call label | | |

Check out the green card for the full list of pseudoinstructions!




https://cdn2.hubspot.net/hubfs/3020607/An%20Introduction%20to%20the%20RISC-V%20Architecture.pdf

# Pseudoinstructions

- Convenient variations of instructions understood by the assembler, but not by the machine

| Pseudoinstructions | Read Instructions | |
|---|---|---|
| mv rd, rs1 | addi rd, rs1, 0 | |
| not rd, rs | xori rd, rs, -1 | |
| li rd, imm | # <= 12-bit signed imm | # >12-bit immediate |
| | addi rd, x0, imm | lui rd, imm [31:12] |
| | | addi rd, rd, imm [11:0] |
| j Label | jal x0, Label | |
| jr rs1 | jalr x0, rs1 | |
| la rd, label | #absolute address | # PC-relative address |
| | lui rd, addr[31:12] | auipc rd, addr [31:12] |
| | addi rd, rd, addr [11:0] | addi rd, rd, addr [11:0] |
| call label | auipc ra, addr [31:12] | |
| | jalr ra, addr [11:0] | |

Compiler often use call for jal far_away, e.g., in a different file

- Not actual hardware instructions
- Pseudoinstructions can vary between different assemblers and architectures
- might be implemented differently on different systems

# Levels of Representation/Interpretation

How do we translate programs from a high-level language into machine code?

**High-level language program (e.g., C)**

Compiler

**Assembly language program (e.g., RISC-V)**

Assembler

**Machine language program (e.g., RISC-V)**

**HW Architecture (e.g., block diagrams)**

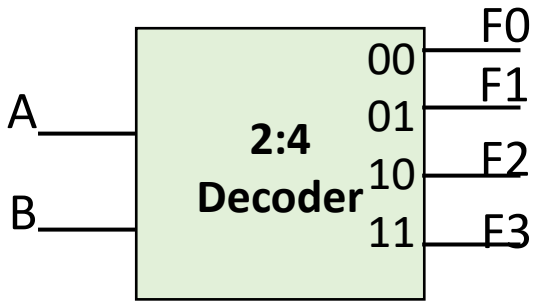
Architecture implementation

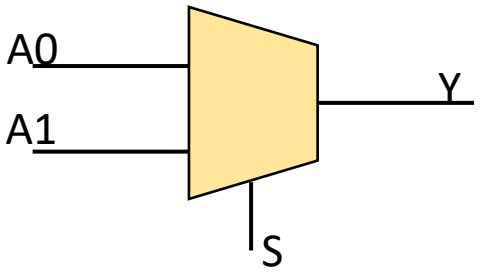
**Logic Circuit (e.g., schematic)**

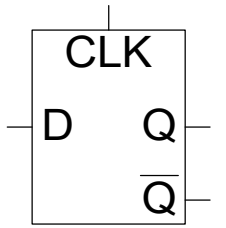
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

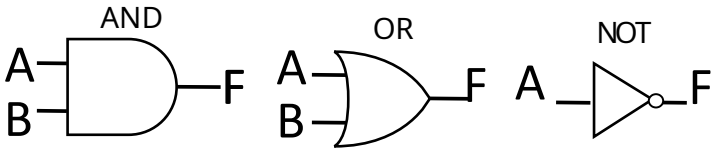
```
lw x3, 0(x10)
lw x4, 4(x10)
sw x4, 0(x10)
sw x3, 4(x10)
```

```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```



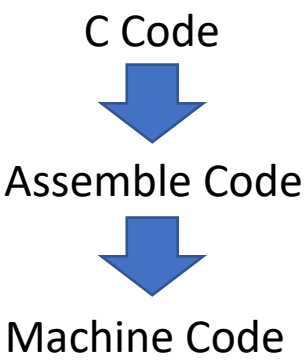



# How Do We Run a C Program?

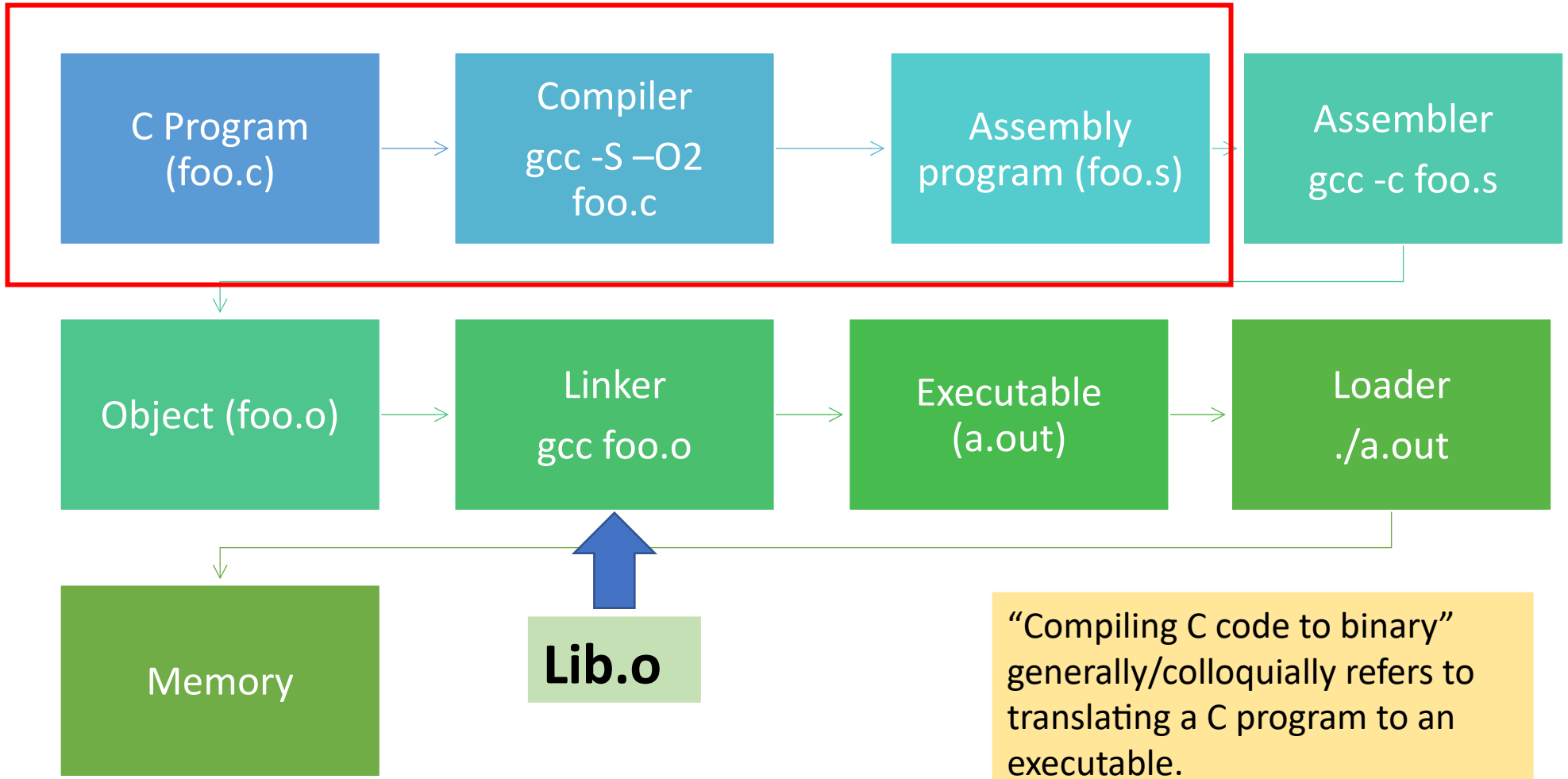
- Translator
  - Converts a program from a source language to an equivalent program in another language
  - Translating/compiling to lower-level languages almost always means higher efficiency, higher performance
- Interpreter
  - Directly executes a program in source language
  - Note: C programs/RISC-V can also be interpreted!
  - Example: Venus RISC-V simulator* useful for learning/debugging

```
#include <stdio.h>
int main()
{
    printf("Hello, %s\n“, "world");
    return 0;
}
```




* https://github.com/kvakil/venus

# Steps in Compiling and Running a C Program




"Compiling C code to binary" generally/colloquially refers to translating a C program to an executable.

Universität Bremen

# Compiler

- Input – high-level language code, e.g., foo.c
- Output – assembly language code, e.g., foo.s for RISC-V
- gcc -S helloworld.c
- Output may contain **pseudoinstructions**
  - mv, li, call, j etc

# Example C Program – hello.c

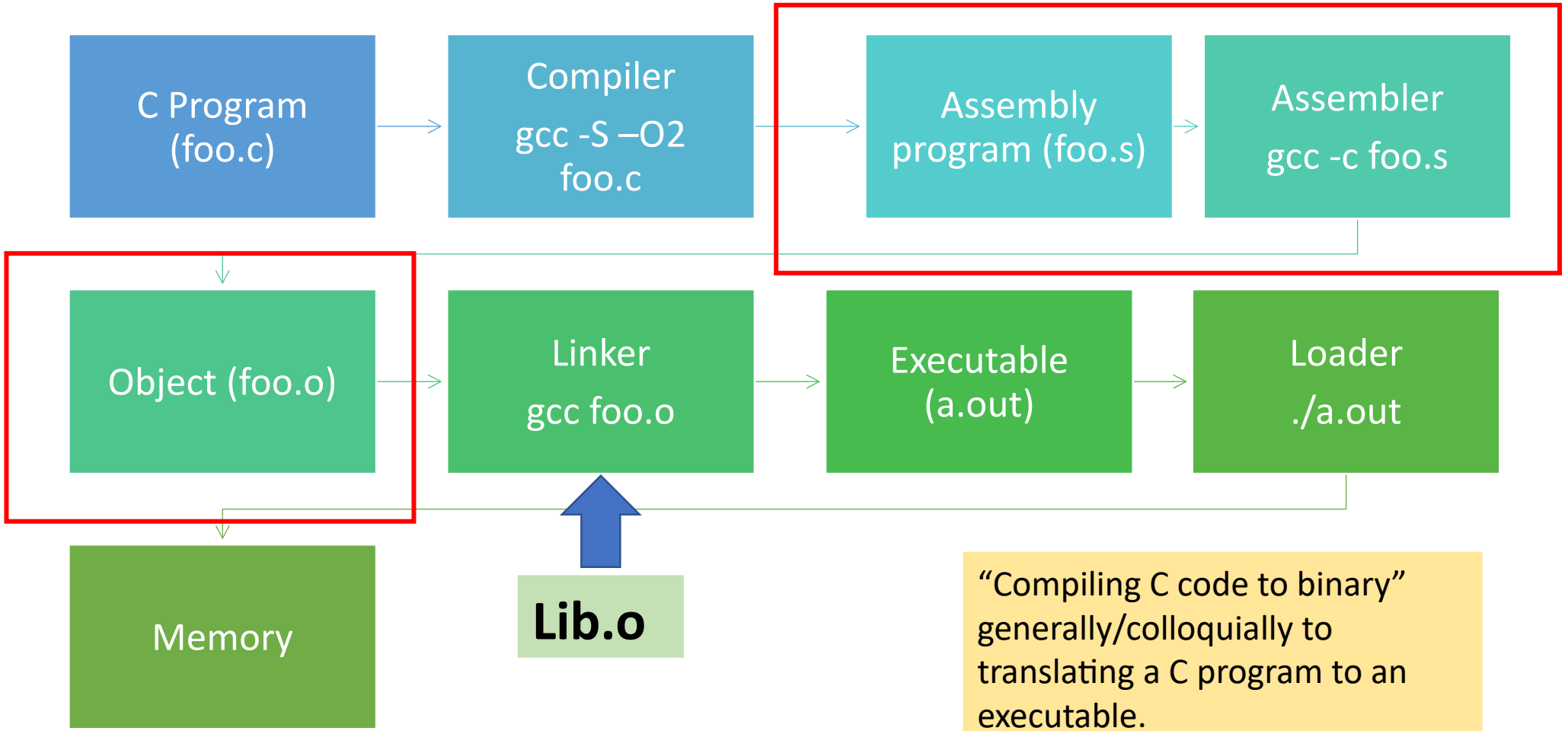
## Compile hello.c → hello.s

gcc -S helloworld.c

```
#include <stdio.h>
int main()
{
    printf("Hello, %s\n“, "world");
    return 0;
}
```

| | |
|---|---|
| **.text**<br>  **.align** 2<br>  **.globl main** | Directive: Enter **text** section<br>Directive: Align code to 2^2 bytes<br>Directive: Declare global symbol **main** |
| **main:**<br>  addi sp,sp,-4<br>  sw ra,0(sp)<br>  la a0, str1<br>  la a1, str2<br>  call printf<br>  lw ra,0(sp)<br>  addi sp,sp,4<br>  li a0,0<br>  Ret | Label for start of **main**<br>Allocate stack frame<br>save return address<br>Pseudoinstructions<br>load addresses of str1, str2<br>Pseudoinstruction: call function printf<br>Restore return address,<br>deallocate stack frame<br>Return with value 0<br>---- |
| **.section** .rodata<br>  **.balign** 4 | Directive: Enter read-only data section<br>Directive: Align data section to 4 bytes |
| str1:<br>  **.string** "Hello, %s!\n"<br>str2:<br>  **.string** "world" | Label for str1<br>Directive: null-terminated string<br>Label for str2<br>Directive: null-terminated string |

# Steps in Compiling and Running a C Program




"Compiling C code to binary" generally/colloquially to translating a C program to an executable.

# Assembler

- Input – assembly language code, e.g., foo.s for RISC-V
  - may contain **pseudoinstructions**
- Output – Machine language module, object file (e.g., foo.o for RISC-V)
  - Object Code (machine language)
  - Information for linking and debugging
    - Symbol Table, Relocation Information, Data Segment, etc
- gcc -c helloworld.S -o helloworld.o
- Reads and uses **directives**
- Replaces pseudoinstructions with real assembly
  - Then produce machine language code

# Directives

- Directives give directions to the assembler
  - Often generated by the compiler (previous stage)
  - Directives do not produce machine instructions! Rather, they inform how to build different parts of the **object file**

| | |
|---|---|
| .text<br>  .align 2<br>  .globl main | Directive: Enter text section<br>Directive: Align code to 2^2 bytes<br>Directive: Declare global symbol **main** |
| **main:**<br>  addi sp,sp,-4<br>  sw ra,0(sp)<br>  la a0, str1<br>  la a1, str2<br>  call printf<br>  lw ra,0(sp)<br>  addi sp,sp,4<br>  li a0,0<br>  Ret | Label for start of **main**<br>Allocate stack frame<br>save return address<br>Pseudoinstructions<br>load addresses of str1, str2<br>Pseudoinstruction: call function printf<br>Restore return address,<br>deallocate stack frame<br>Return with value 0<br>---- |
| .section .rodata<br>  .balign 4 | Directive: Enter read-only data section<br>Directive: Align data section to 4 bytes |
| str1:<br>  .string "Hello, %s!\n"<br>str2:<br>  .string "world" | Label for str1<br>Directive: null-terminated string<br>Label for str2<br>Directive: null-terminated string |

| Directive | Description |
|---|---|
| .text | Subsequent items put in user Text segment (machine code) |
| .data | Subsequent items put in user Data segment (source file data in binary) |
| .global 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 |

Universität Bremen

# Object File Format

- Object File Header
  - Provides metadata about object file
  - Size and position of other parts of the object file
  - Helps the linker or loader interpret and parse the file correctly
- Text Segment
  - Machine code (executable instructions that were translated from source code)
- Data Segment
  - Binary representation of static data in the source file
- Symbol Table
  - List of file's labels, static data that can be referenced by other programs
- Relocation Information
  - Lines of code to fix later (by linker)
- Debugging information

# Producing Machine Code

- Simple case
  - Arithmetic, Logical, Shifts, etc.
  - All necessary info is within the instruction already!

```
add x18 x18,x10
addi x19,x19,-1
```

- PC-relative branches and jumps
  - e.g., beq/bne/etc. and jal
  - Position-Independent Code (PIC):
    - code that can execute correctly regardless of where in memory it is loaded
    - Once pseudoinstructions are replaced with real ones, all **PC-relative addressing** can be computed
  - Determine the offset to encode by counting the number of half-word instructions between current instruction and target instruction

j label

jal x0, label

# Computing PC-relative addresses

- Can we compute all offsets in a single pass?
  - No!
- Forward reference problem
  - Branches, PC-relative jumps can refer to labels that are “forward” in the program.
  - Precise positive offsets are unknown in the first pass!
- Instead, take two passes over the program
  - Pass 1: Remember positions of labels (store in symbol table).
  - Pass 2: Use label positions to generate machine code.

**3 words forward**
**(byte-offset +12)**

```
addi t2, zero, 9          # t2 = 9;
Loop: slt t1, zero, t2    # t1 = (0 < t2);
beq t1, zero, Exit        # t2 <= 0; goto Exit
addi t2, t2, -1           # t2--;
j Loop                    # go back to Loop
Exit:
```

**3 words back**
**(byte-offset -12)**

# What About Other References?

- Reference to other files
  - e.g., calling strlen from the C string library
- References to static data
  - e.g., la (load address) gets broken up until lui and addi
  - These require knowing the full 32-bit address of the data

- These can’t be determined yet, so the assembler writes them down in two tables
  - Relocation information
  - Symbol Table

Universität Bremen

# Symbol Table

- List of items in this file
- Instruction labels
  - Used to compute machine code for PC-relative addressing in branches, function calling, etc.
  - .global directive: labels can be referenced by other files.
- Data
  - Anything in the .data section
  - Global variables may be accessed/used by other files.

Universität Bremen

# Relocation Information

- List of items whose address this file needs
- Any external label jumped to
  - External label (including lib files): jal ext_label
- Any piece of data in static section
  - e.g., la instruction (for lw/sw base register)

# Example C Program – hello.c

Compile hello.c → hello.s

```
#include <stdio.h>
int main()
{
    printf("Hello, %s\n“, "world");
    return 0;
}
```

| | |
|---|---|
| **.text**<br>  **.align** 2<br>  **.globl main** | Directive: Enter **text** section<br>Directive: Align code to 2^2 bytes<br>Directive: Declare global symbol **main** |
| **main:**<br>  addi sp,sp,-4<br>  sw ra,0(sp)<br>  la a0, str1<br>  la a1, str2<br>  call printf<br>  lw ra,0(sp)<br>  addi sp,sp,4<br>  li a0,0<br>  Ret | Label for start of **main**<br>Allocate stack frame<br>save return address<br>Pseudoinstructions<br>load addresses of str1, str2<br>Pseudoinstruction: call function printf<br>Restore return address,<br>deallocate stack frame<br>Return with value 0<br>---- |
| **.section** .rodata<br>  **.balign** 4 | Directive: Enter read-only data section<br>Directive: Align data section to 4 bytes |
| str1:<br>  **.string** "Hello, %s!\n"<br>str2:<br>  **.string** "world" | Label for str1<br>Directive: null-terminated string<br>Label for str2<br>Directive: null-terminated string |

# Assemble – hello.s → hello.o

**Text segment**

```
00000000 <main> :
00: FF010113   addi  sp,sp, -16
04: 00112623   sw    ra, 12 (sp)
08: 00000537   lui   a0, 0x0
0C: 00050513   addi  a0, a0, 0
10: 000005B7   lui   a1, 0x0
14: 00058593   addi  a1, a1, 0
18: 00000097   auipc ra, 0x0
1C: 000080E7   jalr  ra, 0
20: 00C12083   lw    ra, 12 (sp)
24: 01010113   addi  sp, sp, 16
28: 00000513   addi  a0, a0, 0
2C: 00008067   jalr  rac
```

la –> lui, addi
call -> auipc, jalr
pseudoinstructions replaced

Address placeholders

| | |
|---|---|
| .text<br>.align 2<br>.globl main | Directive: Enter text section<br>Directive: Align code to 2^2 bytes<br>Directive: Declare global symbol main |
| main:<br>addi sp,sp,-4<br>sw ra,0(sp)<br>la a0, str1<br>la a1, str2<br>call printf<br>lw ra,0(sp)<br>addi sp,sp,4<br>li a0,0<br>Ret | Label for start of main<br>Allocate stack frame<br>save return address<br>Pseudoinstructions<br>load addresses of str1, str2<br>Pseudoinstruction: call function printf<br>Restore return address,<br>deallocate stack frame<br>Return with value 0<br>---- |
| .section .rodata<br>.balign 4 | Directive: Enter read-only data section<br>Directive: Align data section to 4 bytes |
| str1:<br>.string "Hello, %s!\n"<br>str2:<br>.string "world" | Label for str1<br>Directive: null-terminated string<br>Label for str2<br>Directive: null-terminated string |

**Object file (Assembler output)**

```
...          ff010113  00112623 00000537
00050513 000005b7 00058593 00000097
000080e7 00c12083  01010113 00000513
00008067 ...
```

**Symbol Table**

| Label | Address (in module) | type |
|---|---|---|
| main: | 0x00000000 | global text |
| str1: | 0x00000000 | local data |
| str2: | 0x0000000c | local data |

**Relocation Information**

| Address | type | Dependency |
|---|---|---|
| 0x00000008 | lui | %hi(str1) |
| 0x0000000c | addi | %low(str1) |
| 0x00000010 | lui | %hi(str2) |

# Assemble – hello.s → hello.o

**Text segment**

```
00000000 <main> :
00: FF010113   addi  sp,sp, -16
04: 00112623   sw    ra, 12 (sp)
08: 00000537   lui   a0, 0x0
0C: 00050513   addi  a0, a0, 0
10: 000005B7   lui   a1, 0x0
14: 00058593   addi  a1, a1, 0
18: 00000097   auipc ra, 0x0
1C: 000080E7   jalr  ra, 0
20: 00C12083   lw    ra, 12 (sp)
24: 01010113   addi  sp, sp, 16
28: 00000513   addi  a0, a0, 0
2C: 00008067   jalr  rac
```

For the instruction la a0, str1:

lui a0, %hi(str1) loads the high 20 bits of str1's address into a0.

addi a0, a0, %lo(str1) adds the lower 12 bits of the address to a0.

**Object file (Assembler output)**

```
...          ff010113  00112623  00000537
00050513 000005b7 00058593 00000097
000080e7 00c12083  01010113 00000513
00008067 ...
```

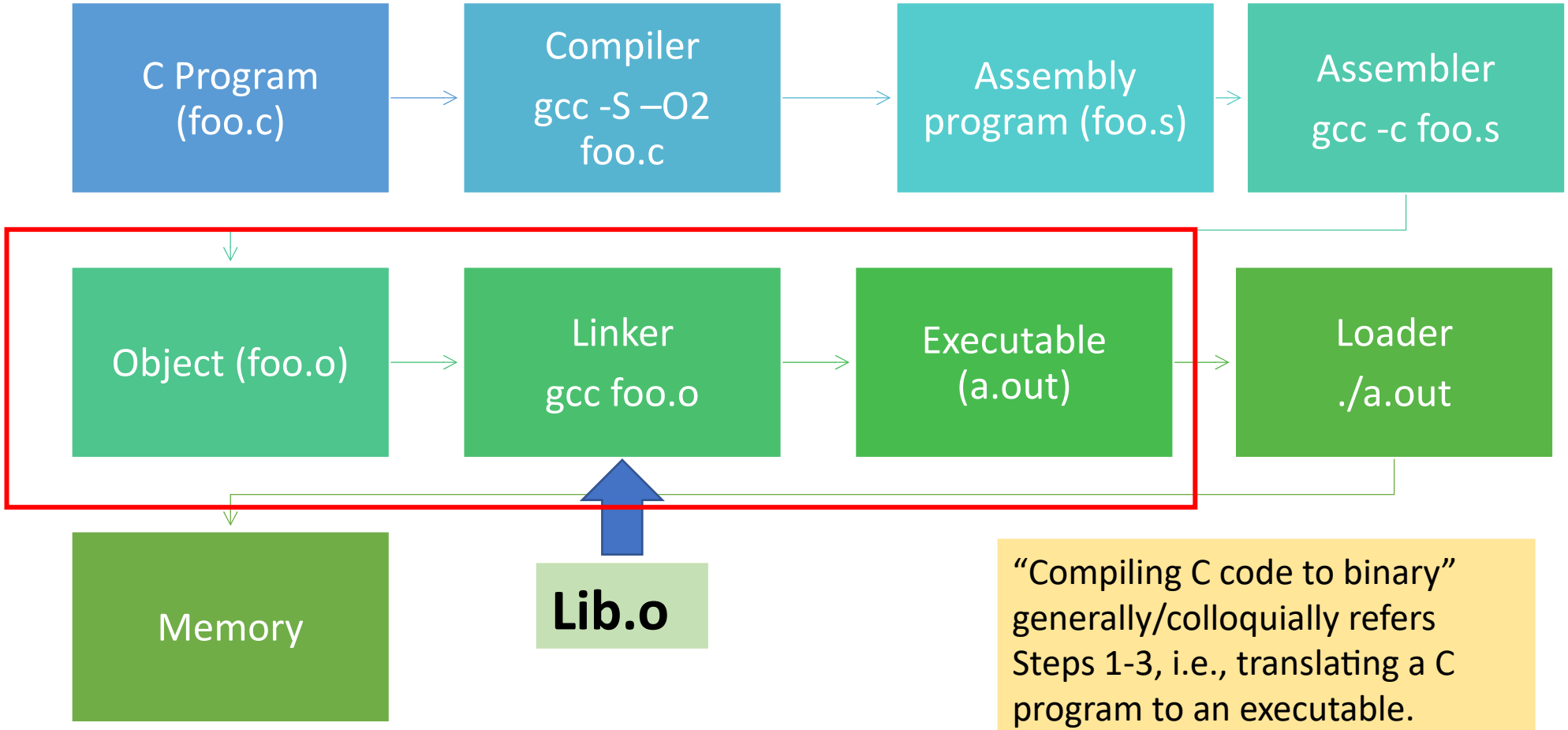
**Symbol Table**

| Label | Address (in module) | type |
|---|---|---|
| main: | 0x00000000 | global text |
| str1: | 0x00000000 | local data |
| str2: | 0x0000000c | local data |

**Relocation Information**

| Address | type | Dependency |
|---|---|---|
| 0x00000008 | lui | %hi(str1) |
| 0x0000000c | addi | %low(str1) |
| 0x00000010 | lui | %hi(str2) |

# Steps in Compiling and Running a C Program

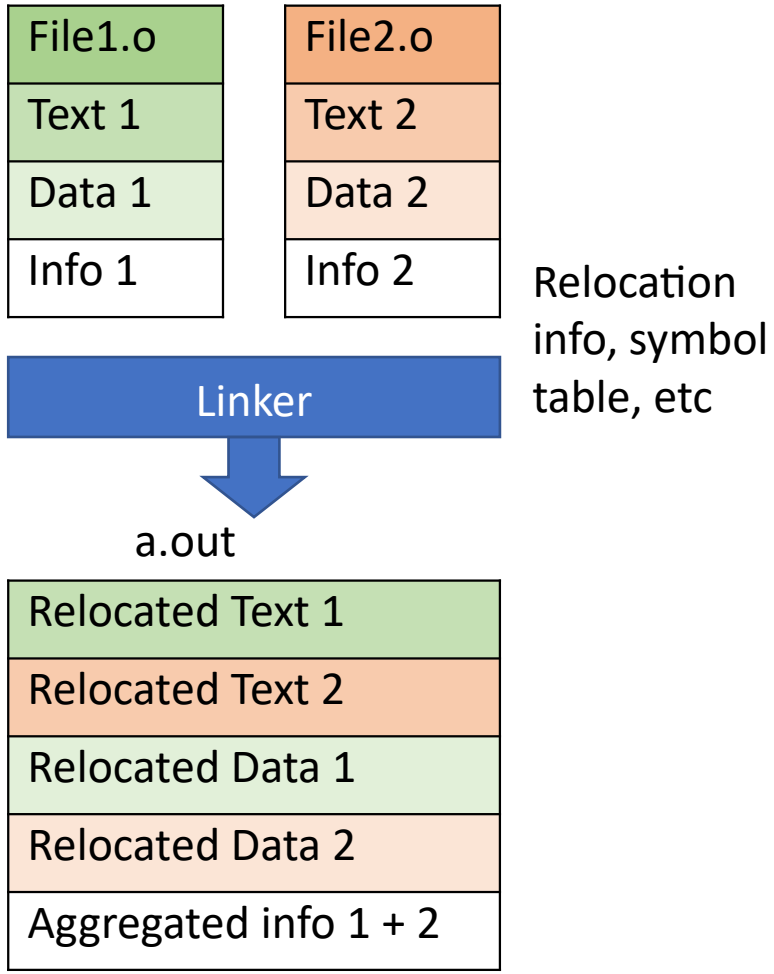



"Compiling C code to binary" generally/colloquially refers Steps 1-3, i.e., translating a C program to an executable.

# Linker

- Input – object files (foo.o, lib.o for RISC-V)
  - Text/Data segments, Info Tables per file
- Output – Executable machine code (e.g., a.out for RISC-V)
- gcc helloworld.o -o helloworld
- The linker enables separate compilation of files
  - Changes to one file does not require recompilation of the entire program. (e.g., Linux source >20M lines of code)
  - Old name: “Link Editor” from editing the “links” in jump-and-link instructions.

# Linker Patches Together Multiple Objects




1. Put together text segments from each .o file
2. Put together data segments from each .o file, then concatenate this at the end of Step 1's segment
3. Resolve reference
   - Go through Relocation Table and handle each entry, i.e., fill in all absolute addresses

# Which Addresses Need Relocating?

- PC-relative addressing
  - beq, bne, jal, auipc/addi, etc
  - Never relocate
    - Position Independent Code (PIC)
- External Function Reference
  - usually jal or auipc/jalr
  - Always relocate
    - Addresses unknown at the time of assembling
    - final location will only be determined during the linking phase
- Static Data Reference
  - lw, sw, lui/addi
  - Always relocate
    - Data segment has relocated

# Which Instructions Need Relocation Editing?

- J-Format (only when jumping externally)


**J-Type**

- Loads and stores using gp to access .data variables
  - Global pointer (gp), is a pointer to the data/static segment.


**I-Type**

**S-Type**

- Lui, addi, auipc/jalr (if jumping externally)
- Conditional branches (B-type) don't need editing
  - PC-relative addressing preserved even if text is relocated

# Resolving References

- For RV32, linker assumes first text segment starts at address **0x10000**
- Linker Knows
  - Length of each text/data segment
  - Ordering of text and segments
- Linker Calculates
  - Absolute address of each label to be jumped to and of each piece of data referenced
- Linker Resolves References
  - Search for reference (data or label) in all ”user” symbol tables.
  - If not found, search library files (e.g., for printf).
  - Once absolute address determined, fill in machine code appropriately.
- Executable
  - Executable containing text and data (+ header/debugging info)

# Static vs. Dynamic Linking

- So far, we have described the traditional way – statically linked libs
  - The library is now part of the executable – if the library updates, we won't get the fix unless we recompile the user program.
  - The executable includes the entire library, even if not all of it is used by the user program.
  - Pro: The Executable is self-contained!
- What is the alternative?
  - Dynamically linked libraries (DLL)
    - Common on Windows and UNIX

# Dynamically-Linked Libraries

- Space vs. Time
  - Storing a program requires less disk space; less time to send a program.
  - Executing two programs requires less memory (if they share a library)
  - Runtime overhead: extra time to dynamically link!
- Reasonable handling of upgrades
  - Replacing libXYZ.so upgrades every program using library XYZ.
  - Having the program executable isn't enough anymore!
- Prevailing approach to DLL uses machine code as the lowest common denominator
  - "Linking at the machine code level," i.e., linker does not know what compiler/what language compiled from.
  - Not the only way to do it

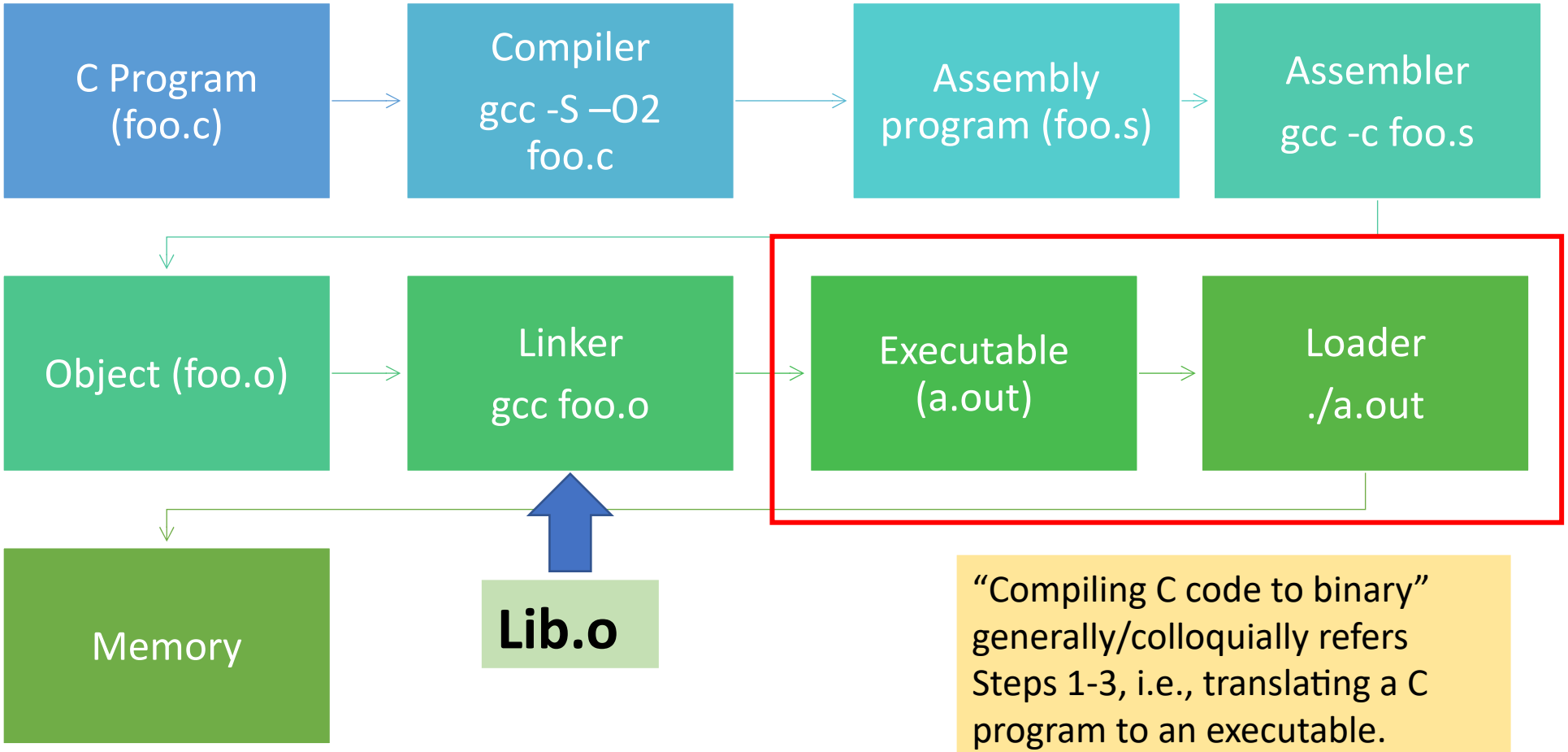
Overall dynamic linking adds complexity to compiler, linker and OS. However, it provides many benefits that often outweigh its complexities.

Universität Bremen

# Benefits of Dynamic Linking

- Shared libraries
  - Multiple programs can share the same library (resource sharing)
  - Beneficial for commonly used libs
- Smaller executable size
- Simplified updates
- Modular design

# Steps in Compiling and Running a C Program




"Compiling C code to binary" generally/colloquially refers Steps 1-3, i.e., translating a C program to an executable.

# Loader

- Input – Executable code (a.out for RISC-V)
- Output – program is run
- Executable files are stored on disk
- When an executable is run, loader loads it into memory and start it running
- Load program into a new created address space
  - Read executable's file header for sizes of text, data segments.
  - Create new address space for program large enough to hold text and data segments, along with a stack segment.
    - Copy instructions, data from executable file into new address space.
    - Copy arguments passed to the program onto the stack
- Initialize machine registers
  - Most registers cleared; stack pointer (sp) assigned address of first free stack location
- Jump to start-up routine, which does the following
  - Copy program arguments from stack to registers, set PC
  - If main routine returns, terminate program with exit system call.

In reality: Loader is the operating system (OS)!
Loading is one of the OS tasks.

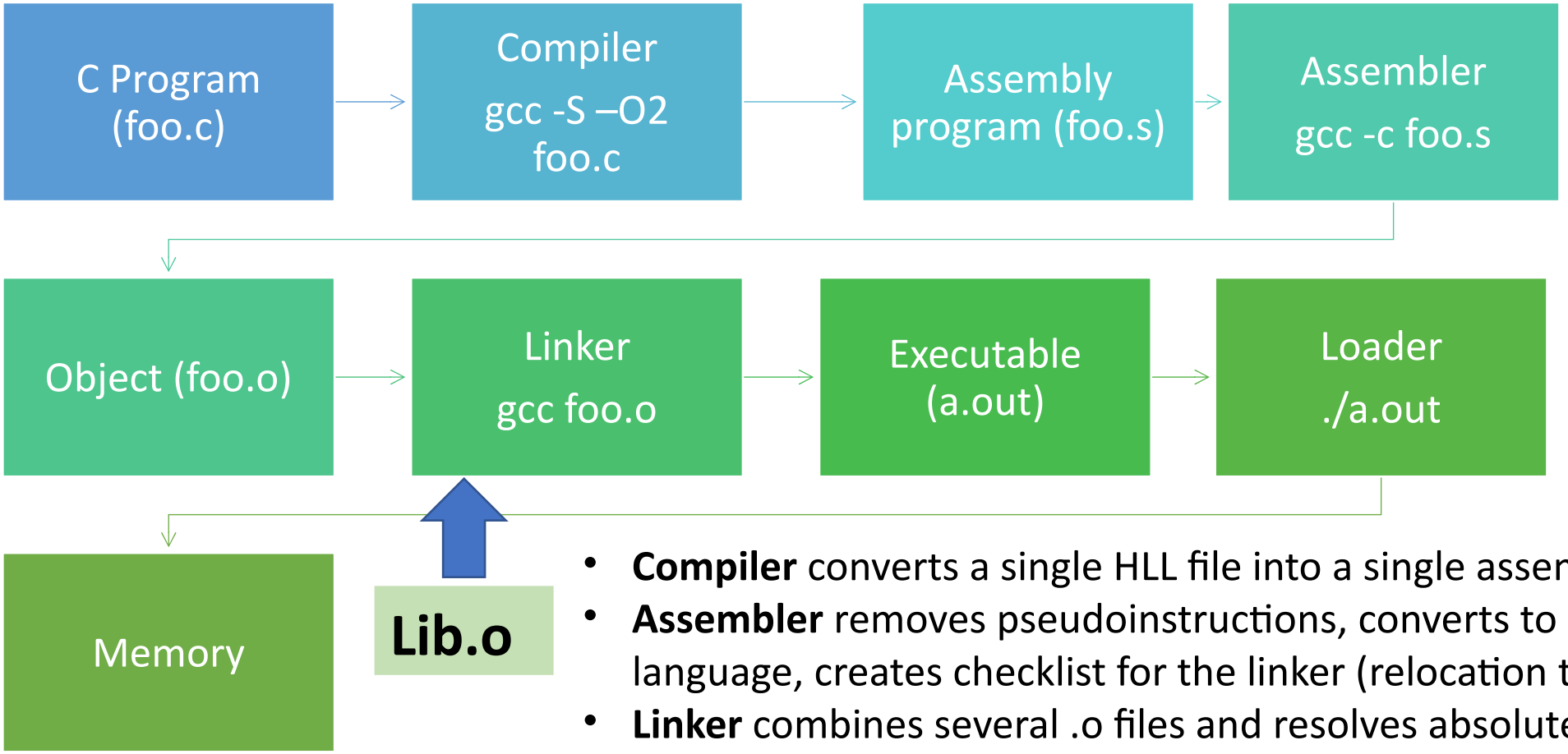
# Putting it all together – Example C Program – hello.c

Compile hello.c → hello.s

```
#include <stdio.h>
int main()
{
    printf("Hello, %s\n“, "world");
    return 0;
}
```

| | |
|---|---|
| **.text**<br>  **.align** 2<br>  **.globl main** | Directive: Enter **text** section<br>Directive: Align code to 2^2 bytes<br>Directive: Declare global symbol **main** |
| **main:**<br>  addi sp,sp,-4<br>  sw ra,0(sp)<br>  la a0, str1<br>  la a1, str2<br>  call printf<br>  lw ra,0(sp)<br>  addi sp,sp,4<br>  li a0,0<br>  Ret | Label for start of **main**<br>Allocate stack frame<br>save return address<br>Pseudoinstructions<br>load addresses of str1, str2<br>Pseudoinstruction: call function printf<br>Restore return address,<br>deallocate stack frame<br>Return with value 0<br>---- |
| **.section** .rodata<br>  **.balign** 4 | Directive: Enter read-only data section<br>Directive: Align data section to 4 bytes |
| str1:<br>  **.string** "Hello, %s!\n"<br>str2:<br>  **.string** "world" | Label for str1<br>Directive: null-terminated string<br>Label for str2<br>Directive: null-terminated string |

# Steps in Compiling and Running a C Program




- **Compiler** converts a single HLL file into a single assembly file
- **Assembler** removes pseudoinstructions, converts to machine language, creates checklist for the linker (relocation table)
- **Linker** combines several .o files and resolves absolute addresses
- **Loader** loads executable into memory and begins execution

Universität Bremen

# From Software to Hardware

Prof. Dr. Rolf Drechsler
Dr. Muhammad Hassan
M.Sc. Jan Zielasko
M.Sc. Milan Funck

# Peer Instruction

- At what point in the execution process are all the machine code bits determined for the following assembly language?

`add x6, x7, x8`

A. After compilation
B. After assembly
C. After linking
D. After loading

`jal x1, fprintf`

A. After compilation
B. After assembly
C. After linking
D. After loading

# Link – hello.o → a.out

**Text segment**

```
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 <str1>
101C0: 000215B7   lui   a1, 0x21
101C4: a1c58593   addi  a1, a1, -1508      # 20a1c <str2>
101C8: 288000ef   jal   ra, 10450          # printf
101CC: 00C12083   lw    ra, 12 (sp)
101D0: 01010113   addi  sp, sp, 16
101D4: 00000513   addi  a0, a0, 0
101D8: 00008067   jalr  ra
```

How did the linker compute these immediates?

# Assemble – hello.s → hello.o

**Text segment**

```
00000000 <main> :
00: FF010113   addi  sp,sp, -16
04: 00112623   sw    ra, 12 (sp)
08: 00000537   lui   a0, 0x0
0C: 00050513   addi  a0, a0, 0
10: 000005B7   lui   a1, 0x0
14: 00058593   addi  a1, a1, 0
18: 00000097   auipc ra, 0x0
1C: 000080E7   jalr  ra, 0
20: 00C12083   lw    ra, 12 (sp)
24: 01010113   addi  sp, sp, 16
28: 00000513   addi  a0, a0, 0
2C: 00008067   jalr  rac
```

la, call pseudoinstructions replaced

Address placeholders

**Object file (Assembler output)**

```
…          ff010113  00112623 00000537
00050513 000005b7 00058593 00000097
000080e7 00c12083  01010113 00000513
00008067 …
```

**Symbol Table**

| Label | Address (in module) | type |
|---|---|---|
| main: | 0x00000000 | global text |
| str1: | 0x00000000 | local data |
| str2: | 0x0000000c | local data |

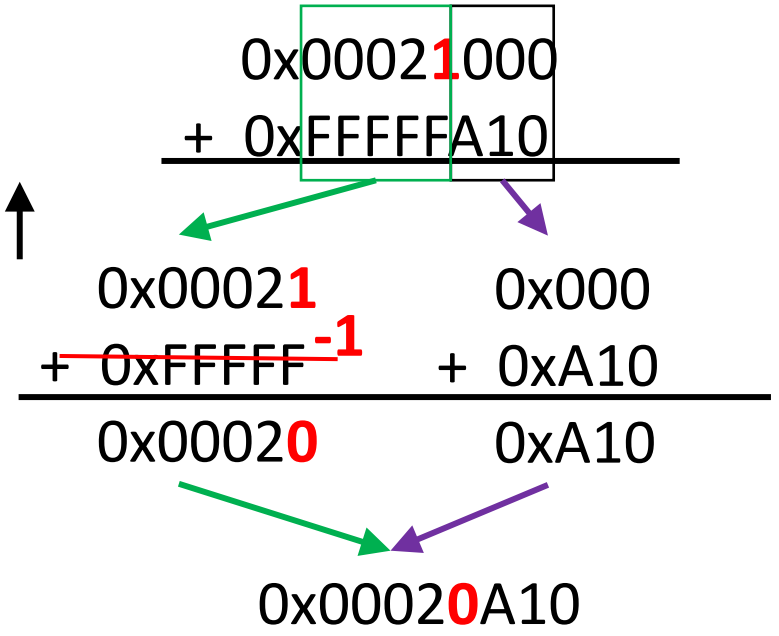
**Relocation Information**

| Address | type | Dependency |
|---|---|---|
| 0x00000008 | lui | %hi(str1) |
| 0x0000000c | addi | %low(str1) |
| 0x00000010 | lui | %hi(str2) |

# lui/addi Address Calculation, Redux

Address of str1

```
101b8: 00021537   lui    a0, 0x21
101bC: a1050513   addi a0, a0, -1520    # 20a10 <str1>
```




- Recall that addi extends sign
- To create the 32-bit immediate 0x20A10 (address of str1):
  - The lower 12-bit addi immediate has **negative sign bit.**
  - ***Add 1*** to the upper 20-bit lui immediate.

- Lower 12-bit immediate **-1520**

  Two's complement of **0xFFFFFA10:**

  $0x000005EF + 1 = 0x5F0 = 1520_{ten}$

  So $0xFFFFFA10 = -1520_{ten}$

# Commands

- Compiler to assembly
  - gcc -S -o program.s program.c
- Assembly to object code
  - gcc -c -o program.o program.s
- Link the object code to create the executable
  - gcc -o program program.o
- Run the program
  - ./program