CS61C Fall 2025

Precheck: Inst. Trans., CALL

Discussion 6

1 Instruction Translation Pre-Check

1.1 True or False: In RISC-V, the opcode field of an instruction determines its type (R-Type, S-Type, etc.).

1.2 Convert the following RISC-V registers into their binary representation:

`s0`:

`sp`:

`x9`:

`t4`:

1.3 True or False: In RISC-V, the instruction `li x5 0x44331416` will always be encoded in 32 bits when translated into binary.

1.4 True or False: We can use a branch instruction to move the PC by one byte.

2 Instruction Translation

Recall that every instruction in RISC-V can be represented as a 32-bit binary value, which encodes the type of instruction, as well as any registers/immediates included in the instruction. To convert a RISC-V instruction to binary, and vice-versa, you can use the steps below. The 61C reference sheet will be very useful for conversions!

RISC-V ⟹ Binary

(a) Identify the instruction type (R, I, I*, S, B, U, or J)
(b) Find the corresponding instruction format
(c) Convert the registers and immediate value, if applicable, into binary
(d) Arrange the binary bits according to the instruction format, including the opcode bits (and possibly funct3/funct7 bits)

Binary ⟹ RISC-V

(a) Identify the instruction using the opcode (and possibly funct3/funct7) bits
(b) Divide the binary representation into sections based on the instruction format
(c) Translate the registers + immediate value
(d) Put the final instruction together based on instruction type/format

Below is an example of a series of RISC-V instructions with their corresponding binary translations.

example.S	example.bin

`main:`
`  addi  sp,sp,-4`
`  sw  ra,0(sp)`
`  addi  s0,sp,4`
`  mv  a0,a5`
`  call  printf`
`  ...`
	`...`
	`11111111110000010000000100010011`
	`00000000000100010010000000100011`
	`00000000010000010000010000010011`
	`00000000000000000000010100010011`
	`00000000010001000000000011101111`
	`...`

3 CALL Pre-Check

3.1 The compiler may output pseudoinstructions.

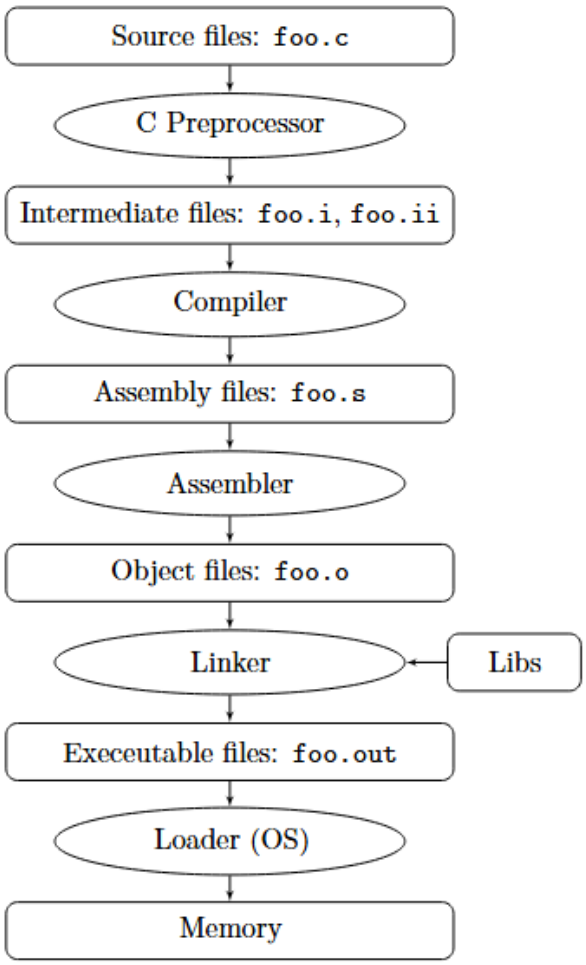
3.2 The main job of the assembler is to perform optimizations on the assembly code.

3.3 The object files produced by the assembler are only moved, not edited, by the linker.

3.4 The destination of all jump instructions is completely determined after linking.

4 CALL

The following is a diagram of the CALL stack detailing how C programs are built and executed by machines:



To translate a C program to an executable:

1. (C Preprocessor): translates all defined macros before passing to the compiler.
2. **Compiler**: Translates high-level language code (e.g. `foo.c`) to assembly
   - **Output**: Assembly language code (RISC-V) which may contain pseudoinstructions!
3. **Assembler**: Replaces pseudoinstructions and creates an *object file* with machine language, symbol table, relocation table, and debugging information.
   - **Output**: Object file (`foo.o`)
4. **Linker**: Combines multiple object files / libraries to create an executable.
   - **Output**: Executable machine code (e.g. `foo.out`).
5. **Loader**: Creates the environment to run machine code and begins execution.

4.1 How many passes through the code does the Assembler have to make? Why?

4.2 Which step in CALL resolves relative addressing? Absolute addressing?

4.3 Describe the six main parts of the object files outputted by the Assembler (Header, Text, Data, Relocation Table, Symbol Table, Debugging Information).