CS 61C — CALL, Boolean Algebra

Spring 2023 — Discussion 5

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 The compiler may output pseudoinstructions.

True. It is the job of the assembler to replace these pseudoinstructions.

1.2 The main job of the assembler is to generate optimized machine code.

False. That's the job of the compiler. The assembler is primarily responsible for replacing pseudoinstructions and resolving offsets.

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

False. The linker needs to relocate all absolute address references.

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

False. Jumps relative to registers (i.e. from jalr instructions) are only known at run-time. Otherwise, you would not be able to call a function from different call sites.

2 Translation

2.1 In this question, we will be translating between RISC-V code and binary/hexadecimal values.

Translate the following Risc-V instructions into binary and hexadecimal notations

```
addi  s1  x0  -24  =     0b______________________________    =  0x____________
sh  s1  4(t1)      =     0b______________________________    =  0x____________
```

For this question, use the reference sheet to get information about the instructions and convert them to binary representation. One thing that helps is splitting the parsing into parts. For question 1:

```
addi  s1  x0  x4:
rd= s1 = 0b01001
rs1 = x0 = 0b00000
immediate = -24 = 0b1111 1110 1000
opcode = 001 0011
funct3 = 000
Bringing it together - 0b1111 1110 1000 0000 0000 0100 1001 0011 = 0xFE800493
```

For question 2, with a similar method we get the answer: 0b0000 0000 1001 0011 0001 0010 0010 0011 = 0x00931223

2.2 In this question, we will be translating between RISC-V code and binary/hexadecimal values.

Translate the following hexadecimal values into the relevant RISC-V instruction.

```
0x234554B7 =  _________________________________
0xFE050CE3 = _________________________________
```

For the reverse conversion, we want to first determine the instruction type. In order to do that, we first look at the opcode (and then func3/func7 if necessary). Let's start with the first one:

```
0x234554B7 = 0b0010 0011 0100 0101 0101 0100 1011 0111, the opcode is always the last 7 bits so opcode = 011 0111, which corresponds to the operation lui!
Looking at lui, we can see that the first 20 bits correspond to the immediate, and the next 5 ones are the register ones. So:
0b0010 0011 0100 0101 0101 = 0x23455 So, the immediate input was indeed 0x23455.
Looking at the next 5 bits, they must be the rd register values. So, we have
rd = 0b01001
That is equal to 9, which is the register x9 = s1. Thus, overall we have
lui s1 0x23455
```

For question 2, with a similar approach: beq a0, x0, -8

3 RISC-V Addressing

We have several *addressing modes* to access memory (immediate not listed):

1. Base displacement addressing adds an immediate to a register value to create a data memory address (used for lw, lb, sw, sb).
2. PC-relative addressing uses the PC and adds the immediate value of the instruction (multiplied by 2) to create an instruction address (used by branch and jump instructions).
3. Register Addressing uses the value in a register as an instruction address. For instance, `jalr`, `jr`, and `ret`, where `jr` and `ret` are just pseudoinstructions that get converted to `jalr`.

3.1 What is the range of 32-bit instructions that can be reached from the current PC using a branch instruction? Recall that RISC-V supports 16b instructions via an extension.

The B-format instruction encoding has space for a 12 bit immediate field. Since RISC-V supports 16b instructions, the byte offset needed to branch to any instruction will always be divisible by 2. Thus, we can assume an implicit 0 at bit 0, allowing a 13 bit byte offset (that's why the reference sheet describes the immediate as `imm[12:1]`!). Since the byte offset is signed, the branch immediate can move the PC in the range of $[-2^{12}, 2^{12}-1]$ bytes. As we are looking for the range of 32-bit instructions, we look for only offsets divisible by 4, leading to a total of $[-2^{10}, 2^{10}-1]$ 32-bit instructions to branch to.

3.2 What is the maximum range of 32-bit instructions that can be reached from the current PC using a jump instruction?

The immediate field of the jal instruction is 20 bits, while that of the jalr instruction is only 12 bits, so jal can reach a wider range of instructions. Similar to above, this 20-bit immediate is multiplied by 2 and added to the PC to get the final address. Since the immediate is signed, we have a range of $[-2^{20}, 2^{20}-2]$ bytes, or $[-2^{19}, 2^{19}-1]$ 2-byte instructions. As we actually want the number of 4-byte instructions, we can reference those within $[-2^{18}, 2^{18}-1]$ instructions of the current PC.

3.3 Given the following RISC-V code (and instruction addresses), fill in the blank fields for the following instructions (you'll need your RISC-V reference sheet!). Each field refers to a different block of the instruction encoding.

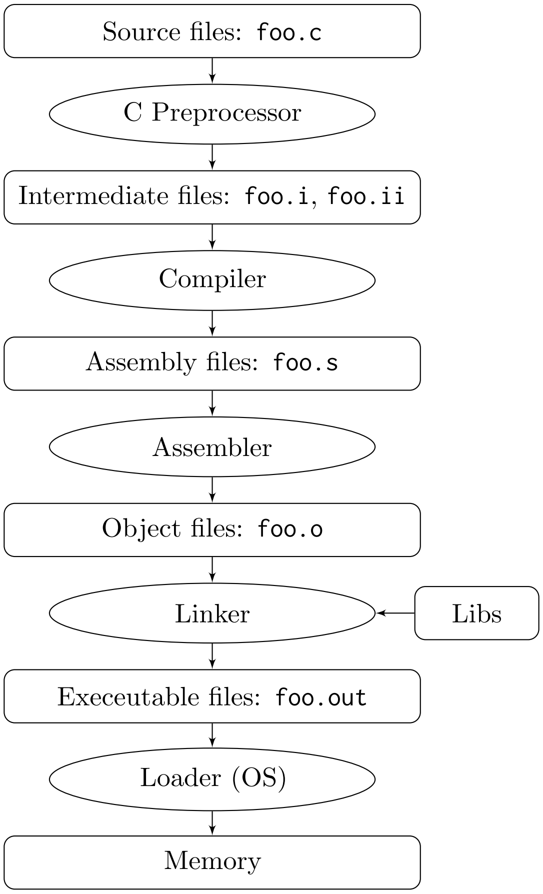
```
0x002cff00: loop: add t1, t2, t0          |________|________|________|________|________|__0x33__|
0x002cff04:       jal ra, foo             |__________________________|_________________|__0x6F__|
0x002cff08:       bne t1, zero, loop      |________|________|________|________|________|__0x63__|
...
0x002cff2c: foo:  jr ra                 ra = ________________________________
```

```
0x002cff00: loop: add t1, t2, t0       |   0   |   5   |   7    |  0  |  6  |  0x33  | → 0x00538333
0x002cff04:       jal ra, foo          |  0  |  0x14  |  0  |   0   |   1   |  0x6F  | → 0x028000ef
0x002cff08:       bne t1, zero, loop   | 1 |  0x3F  | 0 | 6 | 1 |  0xC  | 1 |  0x63  | → 0xfe031ce3
```

```
...
0x002cff2c: foo:  jr ra                  ra = ________0x002cff08________
```

4 CALL

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



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

Two: The first finds all the label addresses, and the second resolves forward references while using these label addresses.

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

Assembler, Linker

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

- Header: Sizes and positions of the other parts
- Text: The machine code
- Data: Binary representation of any data in the source file

- Relocation Table: Identifies lines of code that need to be "handled" by the Linker (jumps to external labels (e.g. lib files), references to static data)
- Symbol Table: List of file labels and data that can be referenced across files
- Debugging Information: Additional information for debuggers

5 Boolean Logic

In digital electronics, it is often important to get certain outputs based on your inputs, as laid out by a truth table. Truth tables map directly to Boolean expressions, and Boolean expressions map directly to logic gates. However, in order to minimize the number of logic gates needed to implement a circuit, it is often useful to simplify long Boolean expressions.

We can simplify expressions using the nine key laws of Boolean algebra:

Name	AND Form	OR form
Commutative	AB = BA	A + B = B + A
Associative	AB(C) = A(BC)	A + (B + C) = (A + B) + C
Identity	1A = A	0 + A = A
Null	0A = 0	1 + A = 1
Absorption	A(A + B) = A	A + AB = A
Distributive	(A + B)(A + C) = A + BC	A(B + C) = AB + AC
Idempotent	A(A) = A	A + A = A
Inverse	$\mathrm{A}(\overline{\mathrm{A}}) = 0$	$\mathrm{A} + \overline{\mathrm{A}} = 1$
De Morgan's	$\overline{\mathrm{AB}} = \overline{\mathrm{A}} + \overline{\mathrm{B}}$	$\overline{\mathrm{A} + \mathrm{B}} = \overline{\mathrm{A}}(\overline{\mathrm{B}})$

5.1 Simplify the following Boolean expressions:

(a) $(A + B)(A + \overline{B})C$

$$\begin{aligned} (A + B)(A + \bar{B})C &= (A + B\bar{B})C \\ &= AC \end{aligned}$$

(b) $\bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + AB\bar{C} + A\bar{B}\bar{C} + ABC + A\bar{B}C$

$$\begin{aligned} \bar{A}\bar{C}(\bar{B} + B) + A\bar{C}(B + \bar{B}) + AC(B + \bar{B}) &= \bar{A}\bar{C} + A\bar{C} + AC \\ &= \bar{A}\bar{C} + A\bar{C} + A\bar{C} + AC \\ &= (\bar{A} + A)\bar{C} + A(\bar{C} + C) \\ &= \bar{C} + A \end{aligned}$$

Alternatively, using the identity from 2.1:

$$\begin{aligned}
\bar{A}\bar{C}(\bar{B}+B)+A\bar{C}(B+\bar{B})+AC(B+\bar{B}) &= \bar{A}\bar{C}+A\bar{C}+AC \\
&= \bar{C}(\bar{A}+A)+AC \\
&= \bar{C}+AC \\
&= \bar{C}+A
\end{aligned}$$

(c) $\overline{A(\bar{B}\bar{C}+BC)}$

$$\begin{aligned}
\overline{A(\bar{B}\bar{C}+BC)} &= \bar{A}+\overline{\bar{B}\bar{C}+BC} \\
&= \bar{A}+\overline{\bar{B}\bar{C}}\,\overline{BC} \\
&= \bar{A}+(B+C)(\bar{B}+\bar{C}) \\
&= \bar{A}+B\bar{C}+\bar{B}C
\end{aligned}$$

(d) $\overline{A}(A+B)+(B+AA)(A+\overline{B})$

$$\begin{aligned}
\overline{A}(A+B)+(B+AA)(A+\overline{B}) &= (\overline{A}A+\overline{A}B)+(B+AA)(A+\overline{B}) \\
&= \overline{A}B+(B+A)(A+\overline{B}) \\
&= \overline{A}B+(BA+AA+B\overline{B}+A\overline{B}) \\
&= \overline{A}B+(BA+A+A\overline{B}) \\
&= \overline{A}B+A \\
&= A+B
\end{aligned}$$

Alternatively,

$$\begin{aligned}
\overline{A}(A+B)+(B+AA)(A+\overline{B}) &= \overline{A}(A+B)+(A+B)(A+\overline{B}) \\
&= (A+B)(\overline{A}+A+\overline{B}) \\
&= A+B
\end{aligned}$$

$\boxed{5.2}$ Use multiple iterations of De Morgan's laws to prove the identity $\bar{A} + AB = \bar{A} + B$.

$$\begin{aligned} \bar{A} + AB &= \overline{A\overline{AB}} \\ &= \overline{A(\bar{A} + \bar{B})} \\ &= \overline{A\bar{A} + A\bar{B}} \\ &= \overline{A\bar{B}} \\ &= \bar{A} + B \end{aligned}$$

$\boxed{5.3}$ Prove that De Morgan's law can be generalized for the complement of any number of terms.

We will prove $\overline{A + B + C + ...} = \overline{A}(\overline{B})(\overline{C})....$ Treat $B + C + ...$ as a single term, then apply De Morgan's on both terms, resulting in $\overline{A}(\overline{B + C + ...})$. Repeating this for all terms results in $\overline{A}(\overline{B})(\overline{C})....$

A similar approach works for the other identity, as both AND and OR are associative.