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.

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

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

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

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

```
1  addi  s1  x0  -24  =     0b______________________________   =  0x___________
2  sh  s1  4(t1)      =     0b______________________________   =  0x___________
```

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.

```
1  0x234554B7 =  _________________________________
2  0xFE050CE3 = _________________________________
```

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.

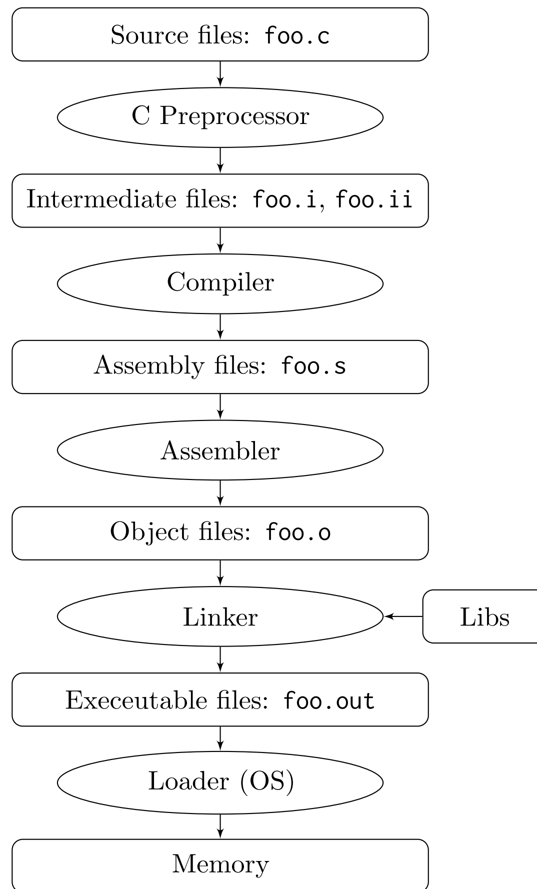
3.2 What is the maximum range of 32-bit instructions that can be reached from the current PC using a jump instruction?

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.

```
1  0x002cff00: loop: add t1, t2, t0        |________|________|________|________|________|__0x33__|
2  0x002cff04:       jal ra, foo           |__________________________|_________________|__0x6F__|
3  0x002cff08:       bne t1, zero, loop    |________|________|________|________|________|__0x63__|
4  ...
5  0x002cff2c: foo:  jr ra                ra = ______________________________
```

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?

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).

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	$\mathrm{AB} = \mathrm{BA}$	$\mathrm{A} + \mathrm{B} = \mathrm{B} + \mathrm{A}$
Associative	$\mathrm{AB(C)} = \mathrm{A(BC)}$	$\mathrm{A} + (\mathrm{B} + \mathrm{C}) = (\mathrm{A} + \mathrm{B}) + \mathrm{C}$
Identity	$1\mathrm{A} = \mathrm{A}$	$0 + \mathrm{A} = \mathrm{A}$
Null	$0\mathrm{A} = 0$	$1 + \mathrm{A} = 1$
Absorption	$\mathrm{A(A + B)} = \mathrm{A}$	$\mathrm{A} + \mathrm{AB} = \mathrm{A}$
Distributive	$\mathrm{(A + B)(A + C)} = \mathrm{A} + \mathrm{BC}$	$\mathrm{A(B + C)} = \mathrm{AB} + \mathrm{AC}$
Idempotent	$\mathrm{A(A)} = \mathrm{A}$	$\mathrm{A} + \mathrm{A} = \mathrm{A}$
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) $(A+B)(A+\overline{B})C$

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

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

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

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

5.3 Prove that De Morgan's law can be generalized for the complement of any number of terms.

CS 61C Reference Card

Version 1.6.0

	Instruction	Name	Description	Type	Opcode	Funct3	Funct7
Arithmetic	**add rd rs1 rs2**	ADD	**rd = rs1 + rs2**	R	011 0011	000	000 0000
	sub rd rs1 rs2	SUBtract	**rd = rs1 - rs2**	R	011 0011	000	010 0000
	and rd rs1 rs2	bitwise AND	**rd = rs1 & rs2**	R	011 0011	111	000 0000
	or rd rs1 rs2	bitwise OR	**rd = rs1 | rs2**	R	011 0011	110	000 0000
	xor rd rs1 rs2	bitwise XOR	**rd = rs1 ^ rs2**	R	011 0011	100	000 0000
	sll rd rs1 rs2	Shift Left Logical	**rd = rs1 << rs2**	R	011 0011	001	000 0000
	srl rd rs1 rs2	Shift Right Logical	**rd = rs1 >> rs2** (Zero-extend)	R	011 0011	101	000 0000
	sra rd rs1 rs2	Shift Right Arithmetic	**rd = rs1 >> rs2** (Sign-extend)	R	011 0011	101	010 0000
	slt rd rs1 rs2	Set Less Than (signed)	**rd = (rs1 < rs2) ? 1 : 0**	R	011 0011	010	000 0000
	sltu rd rs1 rs2	Set Less Than (Unsigned)		R	011 0011	011	000 0000
	addi rd rs1 imm	ADD Immediate	**rd = rs1 + imm**	I	001 0011	000	
	andi rd rs1 imm	bitwise AND Immediate	**rd = rs1 & imm**	I	001 0011	111	
	ori rd rs1 imm	bitwise OR Immediate	**rd = rs1 | imm**	I	001 0011	110	
	xori rd rs1 imm	bitwise XOR Immediate	**rd = rs1 ^ imm**	I	001 0011	100	
	slli rd rs1 imm	Shift Left Logical Immediate	**rd = rs1 << imm**	I*	001 0011	001	000 0000
	srli rd rs1 imm	Shift Right Logical Immediate	**rd = rs1 >> imm** (Zero-extend)	I*	001 0011	101	000 0000
	srai rd rs1 imm	Shift Right Arithmetic Immediate	**rd = rs1 >> imm** (Sign-extend)	I*	001 0011	101	010 0000
	slti rd rs1 imm	Set Less Than Immediate (signed)	**rd = (rs1 < imm) ? 1 : 0**	I	001 0011	010	
	sltiu rd rs1 imm	Set Less Than Immediate (Unsigned)		I	001 0011	011	
Memory	**lb rd imm(rs1)**	Load Byte	**rd =** 1 byte of memory at address **rs1 + imm**, sign-extended	I	000 0011	000	
	lbu rd imm(rs1)	Load Byte (Unsigned)	**rd =** 1 byte of memory at address **rs1 + imm**, zero-extended	I	000 0011	100	
	lh rd imm(rs1)	Load Half-word	**rd =** 2 bytes of memory starting at address **rs1 + imm**, sign-extended	I	000 0011	001	
	lhu rd imm(rs1)	Load Half-word (Unsigned)	**rd =** 2 bytes of memory starting at address **rs1 + imm**, zero-extended	I	000 0011	101	
	lw rd imm(rs1)	Load Word	**rd =** 4 bytes of memory starting at address **rs1 + imm**	I	000 0011	010	
	sb rs2 imm(rs1)	Store Byte	Stores least-significant byte of **rs2** at the address **rs1 + imm** in memory	S	010 0011	000	
	sh rs2 imm(rs1)	Store Half-word	Stores the 2 least-significant bytes of **rs2** starting at the address **rs1 + imm** in memory	S	010 0011	001	
	sw rs2 imm(rs1)	Store Word	Stores **rs2** starting at the address **rs1 + imm** in memory	S	010 0011	010	

	Instruction	Name	Description	Type	Opcode	Funct3
Control	`beq rs1 rs2 label`	Branch if EQual	`if (rs1 == rs2)` `PC = PC + offset`	B	110 0011	000
	`bge rs1 rs2 label`	Branch if Greater or Equal (signed)	`if (rs1 >= rs2)` `PC = PC + offset`	B	110 0011	101
	`bgeu rs1 rs2 label`	Branch if Greater or Equal (Unsigned)		B	110 0011	111
	`blt rs1 rs2 label`	Branch if Less Than (signed)	`if (rs1 < rs2)` `PC = PC + offset`	B	110 0011	100
	`bltu rs1 rs2 label`	Branch if Less Than (Unsigned)		B	110 0011	110
	`bne rs1 rs2 label`	Branch if Not Equal	`if (rs1 != rs2)` `PC = PC + offset`	B	110 0011	001
	`jal rd label`	Jump And Link	`rd = PC + 4` `PC = PC + offset`	J	110 1111	
	`jalr rd rs1 imm`	Jump And Link Register	`rd = PC + 4` `PC = rs1 + imm`	I	110 0111	000
Other	`auipc rd imm`	Add Upper Immediate to PC	`rd = PC + (imm << 12)`	U	001 0111	
	`lui rd imm`	Load Upper Immediate	`rd = imm << 12`	U	011 0111	
	`ebreak`	Environment BREAK	Asks the debugger to do something (`imm = 0`)	I	111 0011	000
	`ecall`	Environment CALL	Asks the OS to do something (`imm = 1`)	I	111 0011	000
Ext	`mul rd rs1 rs2`	MULtiply (part of mul ISA extension)	`rd = rs1 * rs2`	(omitted)		

#	Name	Description	#	Name	Desc
`x0`	`zero`	Constant 0	`x16`	`a6`	*Args*
`x1`	`ra`	*Return Address*	`x17`	`a7`	
`x2`	`sp`	Stack Pointer	`x18`	`s2`	Saved Registers
`x3`	`gp`	Global Pointer	`x19`	`s3`	
`x4`	`tp`	Thread Pointer	`x20`	`s4`	
`x5`	`t0`	*Temporary Registers*	`x21`	`s5`	
`x6`	`t1`		`x22`	`s6`	
`x7`	`t2`		`x23`	`s7`	
`x8`	`s0`	Saved Registers	`x24`	`s8`	
`x9`	`s1`		`x25`	`s9`	
`x10`	`a0`	*Function Arguments or Return Values*	`x26`	`s10`	
`x11`	`a1`		`x27`	`s11`	
`x12`	`a2`	*Function Arguments*	`x28`	`t3`	*Temporaries*
`x13`	`a3`		`x29`	`t4`	
`x14`	`a4`		`x30`	`t5`	
`x15`	`a5`		`x31`	`t6`	

Caller saved registers

Callee saved registers (except `x0`, `gp`, `tp`)

Pseudoinstruction	Name	Description	Translation
`beqz rs1 label`	Branch if EQuals Zero	`if (rs1 == 0)` `PC = PC + offset`	`beq rs1 x0 label`
`bnez rs1 label`	Branch if Not Equals Zero	`if (rs1 != 0)` `PC = PC + offset`	`bne rs1 x0 label`
`j label`	Jump	`PC = PC + offset`	`jal x0 label`
`jr rs1`	Jump Register	`PC = rs1`	`jalr x0 rs1 0`
`la rd label`	Load absolute Address	`rd = &label`	`auipc`, `addi`
`li rd imm`	Load Immediate	`rd = imm`	`lui` (if needed), `addi`
`mv rd rs1`	MoVe	`rd = rs1`	`addi rd rs1 0`
`neg rd rs1`	NEGate	`rd = -rs1`	`sub rd x0 rs1`
`nop`	No OPeration	do nothing	`addi x0 x0 0`
`not rd rs1`	bitwise NOT	`rd = ~rs1`	`xori rd rs1 -1`
`ret`	RETurn	`PC = ra`	`jalr x0 x1 0`

	31 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
I*	funct7	imm[4:0]	rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
J	imm[20|10:1|11|19:12]				rd	opcode

Immediates are sign-extended to 32 bits, except in I* type instructions and `sltiu`.