Lab #2: Multiplication and Division in RISC-V Assembly

Introduction

This project is intended to get you familiar with some of the basic operations and instructions involved in writing an assembly program in RISC-V. You will be implementing shift-and-add multiplication and shift-and-subtract division of ***unsigned*** numbers without using the explicit multiplication and division instructions. For more information regarding multiplication and division algorithms please review lecture viewgraphs (slides 9, 16, 17 in week1 seq&multipliers_2023_updated.pdf slidedeck). The algorithm flowcharts are also provided below for convenience.

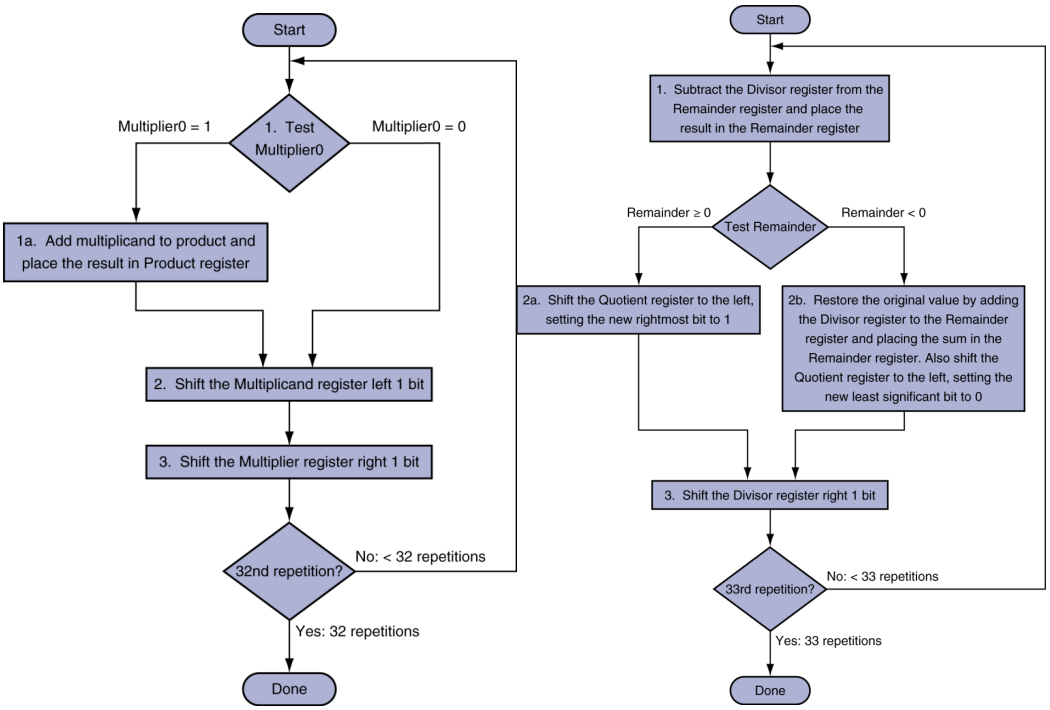

Figure 1. Flowcharts for (left) shift-and-add multiplication and (right) shift-and-subtract division algorithms.

Overview

The goal of this project is to write a function that performs integer multiplication and a function that performs integer division in RISC-V using base integer set of instructions (RV32I), e.g. any instruction from reference sheet page (Figure 2). Avoid using pseudoinstruction (that are not real isntructions and translated to one or several real insructions upon assembly) except for "j" jump instruction that was discussed in class and can be used instead of "jalr" which will be discussed later. Just for a reference, see a list of pseudo-instruction (that should be avoided) in "Assembler Pseudo-instructions" sections at https://michaeljclark.github.io/asm.html .

Note that you cannot just use add or subtract operations repeatedly.

We will use RARS RISC-V emulator https://github.com/TheThirdOne/rars . RARS is distributed as an executable jar. You will need at least Java 8 to run it. For convenience the latest RARS executable file is included in the lab package.

Also provided are a skeletons of code for each function. You are to write the body of each function.

Multiplication
For the multiplication function, start with skeleton code mult.s. You are provided the two operands in a0 and a1 which can be up to 8-bits in length. Place the result of the multiplication in a2. The contents of a0 and a1 must be preserved.

a2 = a0 x a1

Division
For the division function, start with skeleton code div.s. You are provided the divisor in a0 and the dividend in a1. Each operand can be up to 8-bits in length. The quotient of the division should be placed in a2 and the remainder in a3. The contents of a0 and a1 must be preserved.

a2 + reminder a3 = a1/a0

Hints
The code for each function can be greatly simplified through looping. For testing, you can edit op1 and op2 located at the top of the files. op1 is loaded into a0 and op2 is loaded into a1. Make sure that your code covers all relevant corner cases, e.g., division by zero, etc.

Lab Report

Your lab report should be a single PDF file, which has all the following items in the following order and clearly labeled:

1. **Please indicate how many hours you spent on this lab.** This will be helpful for calibrating the workload for the next time the course is taught.
2. Your "mult.s" file *
3. Your "div.s" file. *
4. If you have any feedback on how we might make the lab even better for next quarter, that's always welcome. Please submit it in writing at the end of your lab

* For your code files, please copy and paste your code clearly in a monospace font (ex., Courier New). Provide in the comment the equivalent C code for each added line of the code in the skeleton.

Autograder

You will need to submit your "mult.s" and "div.s" files to the Gradescope assignment, "test lab2". When you make a submission, an autograder will run several tests to ensure your design is

correct. You must pass all the tests to get a full score on the lab. You can resubmit as many times as you like.

What to Turn In

You will need to make two separate Gradescope submissions: “lab2 Code” and “lab2 Report”.

- For “lab2 Code”, please submit your “mult.s” and “div.s” files.
- For “lab2 Report”, please submit your lab report PDF file.

Only one submission per group is needed; be sure to add all your group members.

RISC-V Instruction Set Summary

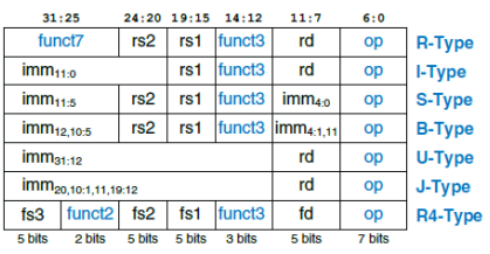

Figure B.1 RISC-V 32-bit instruction formats

- imm: signed immediate in $imm_{11:0}$
- uimm: 5-bit unsigned immediate in $imm_{4:0}$
- upimm: 20 upper bits of a 32-bit immediate, in $imm_{31:12}$
- Address: memory address: rs1 + SignExt($imm_{11:0}$)
- [Address]: data at memory location Address
- BTA: branch target address: PC + SignExt({$imm_{12:1}$, 1'b0})
- JTA: jump target address: PC + SignExt({$imm_{20:1}$, 1'b0})
- label: text indicating instruction address
- SignExt: value sign-extended to 32 bits
- ZeroExt: value zero-extended to 32 bits
- csr: control and status register

Table B.1 RV32I: RISC-V integer instructions

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	slli rd, rs1, uimm	shift left logical immediate	rd = rs1 << uimm
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	srli rd, rs1, uimm	shift right logical immediate	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	srai rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	ori rd, rs1, imm	or immediate	rd = rs1 | SignExt(imm)
0010011 (19)	111	–	I	andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC
0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	[Address]$_{15:0}$ = rs2$_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	[Address]$_{31:0}$ = rs2
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2
0110011 (51)	000	0100000	R	sub rd, rs1, rs2	sub	rd = rs1 – rs2
0110011 (51)	001	0000000	R	sll rd, rs1, rs2	shift left logical	rd = rs1 << rs2$_{4:0}$
0110011 (51)	010	0000000	R	slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	srl rd, rs1, rs2	shift right logical	rd = rs1 >> rs2$_{4:0}$
0110011 (51)	101	0100000	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> rs2$_{4:0}$
0110011 (51)	110	0000000	R	or rd, rs1, rs2	or	rd = rs1 | rs2
0110011 (51)	111	0000000	R	and rd, rs1, rs2	and	rd = rs1 & rs2
0110111 (55)	–	–	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}
1100011 (99)	000	–	B	beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA
1100111 (103)	000	–	I	jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
1101111 (111)	–	–	J	jal rd, label	jump and link	PC = JTA, rd = PC + 4

*Encoded in instr$_{31:25}$, the upper seven bits of the immediate field

Figure 2. Set of instructions that can be used in this lab.