Introduction

In this lab, you will enhance your existing 3-stage RISC-V processor with branching and jump instructions. These will require some changes to parts of your pipeline, but will greatly improve the range of programs your CPU will be able to accept.

Proposed Approach

This lab builds on the work you have done in the previous lab, and will require you to extend several parts in order to support branch and jump type instructions. This section will outline several major modifications you will want to explore in completing this lab.

You will want to **extend your instruction decoder module** (or whatever other system you were using for decoding instructions) to support the two new instruction formats (B-type and J-type). For both types these should result in one or more sign-extended immediate values being produced (for the jump target offset).

You will also want to **extend your control unit** to support the new instructions. This will probably require you to add new cases to deal with B-type and J-type instructions. You will also want to add an extra I-type case to deal with the `jalr` instruction.

Your **fetching logic will need to change** as well, in order to support jumps and branches. We recommend the modifications suggested in the lecture 14 slides, which will change `PC_FETCH` from incrementing each cycle, to a more complex design.

You will need to handle the following cases in your updated fetch logic:

- `PC_FETCH + 1` :: Normal next instruction.
- `branch_addr_EX` :: Address to jump to if the branch is taken. (An offset from the current PC value.)
- `jal_addr_EX` :: Jump target of a `jal` instruction. (An offset from the current PC value.)
- `jalr_addr_EX` :: Jump to address specified in the `jalr` instruction. (`rs1` + offset)

We also recommend creating a `stall_FETCH` and `stall_EX` set of control signals (likely tied to the control unit) as a way of propagating stalls across your pipeline. You can use the stall signals to short out the regfile write- enable signal to ensure the instruction already executing is a no-op.

We recommend making these modifications incrementally – just one group of instructions at a time. This can make it easier to spot errors in your implementation, since you will be testing the updated fetching and branch control logic as you go. A good place to start would be the unconditional jump instruction `jal`.

Note that to support using the HEX displays, you should re-use your `hexdecocoder` module from the previous lab.

Tips & Hints

- It is expected that `make test` and `make gui` will result in compile errors when you first download the project skeleton. This is because the build system cannot find your register file and program memories. You can copy over your RTL code from Lab 2, and that should make these errors disappear immediately.
- Remember that for the simulator to load your program ROM when you run `make gui`, you need to name it `program.rom` and place it in the top-level directory of the project.
- If you don't want to open the RARs gui to compile your `program.rom` file, you can use the command `./tests/assemble.sh ./support/rars.jar bin2dec.asm program.rom`
- You should look at the test benches provided for the register file and ALU. They may be good starting points for you to write your own self-checking testbenches for other modules.
- Just because the ModelSim simulation completed without error, does not mean your test bench passed, remember to check your output carefully. It is perfectly possible to successfully simulate a design which does not work. To phrase this differently, the fact that a broken design can be simulated not working without error does not imply the design itself it without error.

Tools & Methods

The project skeleton includes a variety of tools that can help you to build your CPU. You should consult the users manual, disseminated as a separate handout, for more information. In particular, the "Logic Probes" section may be of particular use.

This assignment also includes an example test harness to help with single-stepping through your CPU design in ModelSim. It doesn't test any behaviors of the CPU, it's just there to automate driving the clock and reset lines. Instructions for using it are included in the file.

Beyond what we have set up for you, you are encouraged to build and develop your own tools and scripts. You are free to build whatever you think will help you, in whatever programming language(s) you like. The VM image provided for the course includes a variety of compilers and interpreters for various languages. Additionally, you may install further compilers if you wish, within reason[1].

Some examples of things you may wish to build:

[1] If you aren't sure of what qualifies as "within reason", contact us

- Further workflow automation beyond what is in the provided `Makefile` (e.g. custom testing scripts as described in *Appendix E*).
- Scripts to generate SystemVerilog macro defines for constant values (such as instruction op-codes).
- Scripts to generate test cases, or to evaluate the results of running tests.

If you choose to build any custom tools, you **must** follow the following rules:

- Clearly document any tools/scripts in a `README.txt` file in the top-level directory of your project.
- Clearly document any special build steps needed beyond the normal `Makefile` to build and run either the GUI or to run your test suite in the `README.txt` file.

If you don't have a `README.txt`, we will run `make gui`, `make test`, and `make testbench` in a clean VM image, and assume that whatever happens is what you intended to occur.

Design Requirements

- (i) Your CPU must implement a three-stage pipeline
  - At a given clock cycle, your CPU should be fetching one instruction, stalling or executing another, and possibly (if not stalled) writing the result of a third instruction to the register file.
- (ii) Your CPU must implement all of the R-Type instructions listed in *Appendix D*.
- (iii) Your CPU must implement all of the I-Type instructions listed in *Appendix D*.
- (iv) Your CPU must implement all of the U-Type instructions listed in *Appendix D*.
- (v) Your CPU must implement all of the J-Type instructions listed in *Appendix D*.
- (vi) Your CPU must implement all of the B-Type instructions listed in *Appendix D*.
- (vii) You must write a complete suite of test cases that validate your CPU works
  - Your test suite may be some combination of the end-to-end tests described in *Appendix F*, test benches, and/or your own test scripts or programs.
  - To check if your test suite works, try choosing an instruction at random, and modify your control unit to produce incorrect values for that instruction, then run your test suite. Did your test suite throw an error? If not, it is incomplete. You should complete this process at least a couple of times for different instructions.
- (viii) The instruction `csrrw x0 io2 rs1` should cause the value of register `rs1` to be displayed on the hex displays.

- (ix) The instruction `csrrw rd io0 x0` should modify the value of register `rd` to be set to the current switch value.
- (x) You must write a RISC-V assembler program, which you should save as `sqrt.asm`. This program should read a value from the switches (`io0`), and should the display the square root of that value as a decimal number on the HEX displays.
  - For example, a switch value of `1101` should cause `13` to be displayed via the HEXes.

The behavior of your CPU will be considered correct for a given instruction if any of the following conditions are met, with the exception of the CSRRW instruction which must work as described in *Apendix D*:

- Your CPU runs the instruction as described in *Appendix D* of this lab sheet.
- Your CPU runs the instruction identically to RARs.
- Your CPU runs the instruction identically to the linked RISC-V RV32I specification.

If the above three conditions are in conflict, prefer the lab sheet, but we will accept any. If you know there is a conflict, this is probably an error on our part, and you should let us know.

Included Materials

The ALU, register file, and self-checking testbenches for both are included with the project, along with the end-to-end testing suite described in *Apendix F*. These are provided for your convenience. If you do not wish to use them, you may delete them and use your own code for any or all of these components.

If you modify one of these components or write your own, it is your responsibility to ensure that it performs correctly and does not introduce errors into your design. If you modify one the given files, you should leave a comment indicating that you did so, and describe what changes you made.

As with the past lab, you **must** retain the given `simtop.sv` and use it as your top-level module for your simulated code, or else it will be impossible to grade your design and you will lose points. Beyond this, the structure of your design is fully up to your discretion.

Rubric

- (A) 20 points – `sqrt.asm` program
- (B) 35 points – jump instructions (jal, jalr)
- (C) 35 points – branch instructions (beq, bne, blt, bge, bltu, bgeu)

- (E) 10 points – README document describing how to test the CPU design

Maximum score: 100 points.

Note that the following may cause you to lose points:

- Academic honesty violation, such as turning in another student's code.
- Code which does not compile.
- Failure to follow requirements *(viii)* and *(ix)*, which will make it very difficult to grade your project.
- Failure to meet design requirements from the last lab; these include i - iv, and viii - ix.
- Attempts to subvert automated testing.

What to Turn In

When you are satisfied that your design works correctly, generate your submission file with `make pack`. Your submission file must be named `CSCE611_<Semester>_jb_<Your USC Username>.7z`. Your "USC username" is whatever you log into your university computer accounts with. For example, a student with the email address `jsmith@email.sc.edu` might turn in a file named `CSCE611_Fall2020_jb_jsmith.7z`. Please use the provided Make target, **do not pack your submission manually**.

How You Will Be Graded

To grade your `sqrt.asm` program, we will load it into your CPU and plug in a number of random inputs in the range $0 \ldots 2^{18} - 1$, asserting that the HEX displays produce the correct outputs. We will use the same random inputs for the entire class. We will use your CPU so that if your program works around a bug or nonstandard behavior in your CPU, you will only lose points for the CPU itself, rather than both the CPU and the program.

Your CPU will be graded via the end-to-end testing harness described in *Appendix F*, but with our own test cases. Because we don't know how you will choose to design your CPU, we will not test individual components, only the behavior of the complete system. We will ensure you have implemented a 3-stage pipeline by checking the number of clock cycles it takes for certain test programs to execute.

Your test suite will be graded by editing parts of your CPU in ways that should break them (e.g. changing the instruction decoder or control unit), then verifying that your test suite catches these bugs. Because each student's CPU will be different, we don't have a standard set of tests to apply for this aspect of grading. Some fair-game things we expect your test suite to catch would include:

- Changing the ALU operation of an instruction in the control unit.
- Swapping fields in the instruction decoder.

- Changing sign extension behavior for immediate values.

Note that in principle, your test suite could consist of only end-to-end tests, but this will make it very difficult to develop your CPU incrementally as independent modules, since you will not be able to test it until it is almost complete.

Appendix A - Background

For the remainder of this course, we will focus on developing a RISC-V processor. RISC-V is an open source instruction set. Conceptually, it is very similar to MIPS, which you may have worked with in previous courses such as CSCE212. RISC-V was introduced in 2012 as a standardized instruction set architecture (ISA) which academics and private companies can use as the basis for their own designs. RISC-V is seeing increasing adoption in academia, as well as in private sector.

Previous iterations of this course were built around the MIPS instruction set, due to its simplicity and widespread use in embedded devices. However the popularity of MIPS has waned even as that of RISC-V has waxed. As a result, we believe RISC-V is more likely to be relevant to any future work you may do in this field than MIPS. However, you will find that the fundamentals of CPU construction are very similar irrespective of the ISA chosen.

For the purposes of this and future labs, we will give you the portions of the specification that you require. RISC-V is broken into multiple separate, but related standards. RV32I is a 32-bit integer-only instruction set – we will be implementing many but not all of its instructions, as well as part of the RV32M multiplication extension. However, if you wish to view the full RISC-V specification, you may do so via this link.

Appendix B - RV32I Instruction Encoding

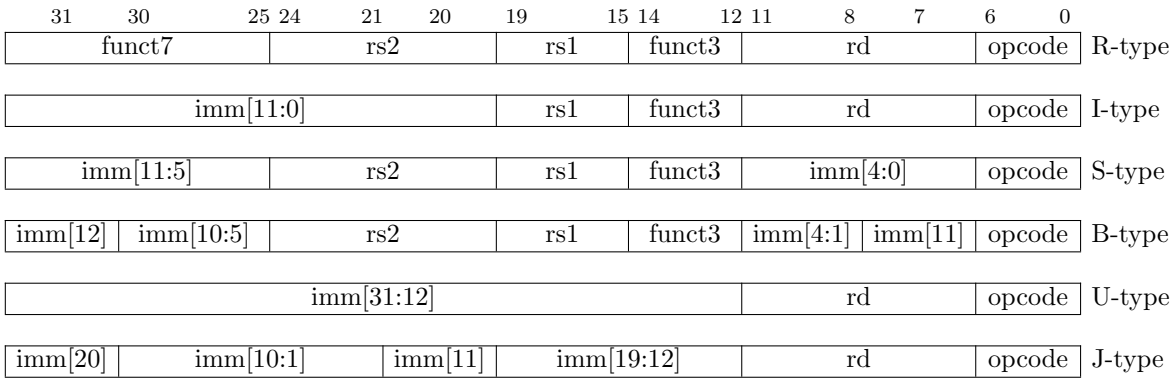

Figure 2.3: RISC-V base instruction formats showing immediate variants.

Figure 1: RISC-V instruction encoding, sourced from page 12 of the specification.

NOTE: S-types will not be used during this course.

Appendix C - Register File Specification

- (i) The register file must have two 32 bit read ports and associated 5 bit read addresses ($\lceil \log_2(32) \rceil = 5$).
- (ii) The register file must have one 32 bit write port and associated 5 bit write address.
- (iii) On any rising clock edge, if the write enable signal is asserted, the value of the write data signal must be written to the memory address specified by the write address.
- (iv) Both read data ports should always have the value stored at the memory address defined by the corresponding read address, except as noted below.
  - (iv.a) If the write enable signal is asserted, then a read via either port of the address defined by writedata should instead yield the value of write data. In other words, both read ports must implement write-bypassing.
  - (iv.b) Any read to the address zero will yield a value of zero via the relevant read data port.
- (v) The register file should have the following inputs/outputs
  - clock

- reset
- read data 1
- read address 1
- read data 2
- read address 2
- write address
- write data
- write enable

Appendix D - RV32I R/I/U/B/J Instruction Reference

Note: In some cases, instructions are grouped with types other than their actual encoding for logical clarity. Such cases are specifically noted.

Note: All immediate values should be sign-extended. The sign bit of all immediate values is in the most-significant bit (bit 31) of the instruction.

R-Type

These instructions operate on two registers, storing the result in a third register.

31-25	24-20	19-15	14-12	11-7	6-0
`funct7`	`rs2`	`rs1`	`funct3`	`rd`	`opcode`

add

`add rd, rs1, rs2`: R[rd] = R[rs1] + R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
`0000000`	`rs2`	`rs1`	`000`	`rd`	`0110011`

sub

`sub rd, rs1, rs2`: R[rd] = R[rs1] - R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0100000	rs2	rs1	000	rd	0110011

and

`and rd, rs1, rs2`: R[rd] = R[rs1] & R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0000000	rs2	rs1	111	rd	0110011

or

`or rd, rs1, rs2`: R[rd] = R[rs1] | R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0000000	rs2	rs1	110	rd	0110011

xor

`xor rd, rs1, rs2`: R[rd] = R[rs1] ^ R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0000000	rs2	rs1	100	rd	0110011

sll

`sll rd, rs1, rs2`: R[rd] = R[rs1] « R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0000000	rs2	rs1	001	rd	0110011

sra

`sra rd, rs1, rs2`: R[rd] = R[rs1] »s R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0100000	rs2	rs1	101	rd	0110011

srl

`srl rd, rs1, rs2`: R[rd] = R[rs1] »u R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0000000	rs2	rs1	101	rd	0110011

slt

`slt rd, rs1, rs2`: R[rd] = R[rs1] <s R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0000000	rs2	rs1	010	rd	0110011

sltu

`sltu rd, rs1, rs2`: R[rd] = R[rs1] <u R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0000000	rs2	rs1	011	rd	0110011

mul

`mul rd, rs1, rs2`: R[rd] = R[rs1] * R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
0000001	rs2	rs1	000	rd	0110011

mulh

mulh rd, rs1, rs2: (R[rd] = R[rs1] s*s R[rs2]) » 32

31-25	24-20	19-15	14-12	11-7	6-0
0000001	rs2	rs1	001	rd	0110011

mulhu

mulhu rd, rs1, rs2: (R[rd] = R[rs1] u*u R[rs2]) » 32

31-25	24-20	19-15	14-12	11-7	6-0
0000001	rs2	rs1	011	rd	0110011

I-Type

These instructions are similar but include an immediate value, or a literal constant as the second operand.

31-20	19-15	14-12	11-7	6-0
imm[11:0]	rs1	funct3	rd	opcode

addi

addi rd, rs1, immediate: R[rd] = R[rs1] + R[rs2]

31-20	19-15	14-12	11-7	6-0
imm[11:0]	rs1	000	rd	0010011

andi

andi rd, rs1, immediate: R[rd] = R[rs1] & R[rs2]

31-20	19-15	14-12	11-7	6-0
imm[11:0]	rs1	111	rd	0010011

ori

ori rd, rs1, immediate: R[rd] = R[rs1] | R[rs2]

31-20	19-15	14-12	11-7	6-0
imm[11:0]	rs1	110	rd	0010011

xori

xori rd, rs1, immediate: R[rd] = R[rs1] ^ R[rs2]

31-20	19-15	14-12	11-7	6-0
imm[11:0]	rs1	100	rd	0010011

slli

slli rd, rs1, immediate: R[rd] = R[rs1] « shamt

31-25	24-20	19-15	14-12	11-7	6-0
0000000	shamt[4:0]	rs1	001	rd	0010011

srai

`srai rd, rs1, immediate`: R[rd] = R[rs1] »s shamt

31-25	24-20	19-15	14-12	11-7	6-0
0100000	shamt[4:0]	rs1	101	rd	0010011

srli

`srli rd, rs1, immediate`: R[rd] = R[rs1] »u shamt

31-25	24-20	19-15	14-12	11-7	6-0
0000000	shamt[4:0]	rs1	101	rd	0010011

jalr

Note: this instruction deals with control-flow, but is encoded as an I-type.

`jalr rd, offset`: t=PC+4; PC += (R[rs1] + sext(offset))&~1; R[rd] = t

31-20	19-15	14-12	11-7	6-0
imm[11:0]	rs1	000	rd	1100111

U-Type

These instructions are similar to I-type, but include a longer immediate value, and don't accept any source registers.

lui

`lui rd, immediate`: R[rd] = imm

Note that the lower 12 bits of the immediate value are treated as zero.

31-12	11-7	6-0
imm[31:12]	rd	0110111

B-Type

These instructions prevent the cpu from executing the next instruction in the program and instead cause it to begin executing a sequence of instructions in another memory location.

b

`b offset`: PC += sext(offset)

31	30-25	24-20	19-15	14-12	11-8	7	6-0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

beq

`beq rs1, rs2, offset`: if (R[rs1]==R[rs2]) PC += sext(offset)

31	30-25	24-20	19-15	14-12	11-8	7	6-0
imm[12]	imm[10:5]	rs2	rs1	000	imm[4:1]	imm[11]	1100011

bge

`bge rs1, rs2, offset`: if (R[rs1]>=R[rs2]) PC += sext(offset)

31	30-25	24-20	19-15	14-12	11-8	7	6-0
imm[12]	imm[10:5]	rs2	rs1	101	imm[4:1]	imm[11]	1100011

bgeu

`bgeu rs1, rs2, offset`: if (R[rs1]>=uR[rs2]) PC += sext(offset)

31	30-25	24-20	19-15	14-12	11-8	7	6-0
imm[12]	imm[10:5]	rs2	rs1	101	imm[4:1]	imm[11]	1100011

blt

`blt rs1, rs2, offset`: if (R[rs1]<R[rs2]) PC += sext(offset)

31	30-25	24-20	19-15	14-12	11-8	7	6-0
imm[12]	imm[10:5]	rs2	rs1	100	imm[4:1]	imm[11]	1100011

bltu

`bltu rs1, rs2, offset`: if (R[rs1]<uR[rs2]) PC += sext(offset)

31	30-25	24-20	19-15	14-12	11-8	7	6-0
imm[12]	imm[10:5]	rs2	rs1	110	imm[4:1]	imm[11]	1100011

J-Type

Note: This instruction is the only J-type we will implement this semster.

jal

`jal rd, offset`: R[rd] = PC+4; PC += sext(offset)

31	30-21	20	19-12	11-7	6-0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	1101111

S-Type

These instructions load and store data to the attached data memory. **Note**: we will not implement these instructions this semester.

`lw`

Note: this instruction is actually encoded as an I-type.

`lw rd, offset(rs1)`: R[rd] = M[R[rs1] + sext(offset)]

31-20	19-15	14-12	11-7	6-0
`imm[11:0]`	`rs1`	`010`	`rd`	`0000011`

`sw`

`sw rs2, offset(rs1)`: M[R[rs1] + sext(offset)] = R[rs2]

31-25	24-20	19-15	14-12	11-7	6-0
`imm[11:5]`	`rs2`	`rs1`	`010`	`imm[4:0]`	`0100011`

Control and Status Register (CSR) Instructions

These instructions read and write to I/O registers. In this course, the IO registers will allow the CPU to interact with the switches, keys, and hex displays of the virtual DE2.

We will define a total of four I/Os, two inputs, and two outputs.

- `io0`, CSR 0xf00, input
- `io1`, CSR 0xf01, input
- `io2`, CSR 0xf02, output
- `io3`, CSR 0xf03, output

`io0` should be connected to the switches, and `io2` should be connected to the HEX displays. You may use `io1` and `io3` for whatever purpose you like, such as for debugging, connecting the LEDs or KEYs, etc.

Note: `io0`, `io1`, `io2`, and `io3` are not standard RISC-V CSRs. A specially modified version of RARS is included with your project skeleton that supports these CSRs. Official versions of MARS will not assemble programs which use these CSRs.

We only need to implement CSRRW for our purposes. Although it ostensibly both writes to and reads from a CSR, the CSRs that we implement are either read-only or write-only, so this instruction can be correctly used to read from or write to any of them. Reading from a write-only I/O such as `io2`, or from an unknown CSR address, should result in a value of `0`. Writing to a read-only I/O such as `io0`, or to an unknown CSR address, should have no effect.

`csrrw`

`csrrw rd, csr, rs1`: t = CSrs[csr]; CSrs[csr] = r[rs1]; r[rd] = t;

31-20	19-15	14-12	11-7	6-0
`imm[11:0]`	`rs1`	`001`	`rd`	`1110011`

Appendix E - Using Multiple Separate Test Benches

The `make testbench` command has been improved since the last lab to allow multiple to-level testbenches to be defined. Any SystemVerilog (`.sv`) files placed in `testbench/` which end with the string `testtop.sv` will be assumed to be their own individual top-level testbenches, and run separately. For example, you will notice that the provided `alu_testtop.sv` and `regfile_testtop.sv` files both run separately.

Appendix F - Writing End-To-End Tests

Because creating the necessary harness for end-to-end testing your CPU is difficult, we have included with this project in the `tests/` folder an end-to-end testing system. You need to have some form of end-to-end testing, but you don't have to use ours (though we recommend you do). This section documents how to use it.

When you run `make test`, each file starting with `test_` and ending with `.py` in the `tests/` folder is executed. They all rely on the `test_util.py` module, which automates assembling programs and running them through a special `test_frontend` command that links into the simulation of your CPU. You need not worry too much about how this works.

Each test file needs to start with `import test_util`, and should then contain one or more calls to `test_util.run_test`. Each test consists of an assembler program, and a dictionary of expected register values. Let's look at a simple example for `addi`:

```
# tests/test_addi.py
import test_util

test_util.run_test("""
addi x1 x0 1
addi x1 x1 2
""", {1: 3})
```

This tests that after the assembler program is run, the register `1` should contain the value `3`. No assertion is made about the state of any other registers after the test is run.

Note that each test file is technically a Python program. However, you don't need not know Python, or write any Python beyond simply copy-pasting the above code example and replacing the assembler program and expected register values.

Appendix G - Full Instruction Table

For your convenience, the entire table of RV32I and M instructions is given below. Note that includes some instructions we will not use in this course.

mnemonic	spec	funct7	funct3	opcode	encoding
LUI	RV32I			0110111	U
AUIPC	RV32I			0010111	U
JAL	RV32I			1101111	J
JALR	RV32I		000	1100111	I
BEQ	RV32I		000	1100011	B
BNE	RV32I		001	1100011	B
BLT	RV32I		100	1100011	B
BGE	RV32I		101	1100011	B
BLTU	RV32I		110	1100011	B
BGEU	RV32I		111	1100011	B
LB	RV32I		000	0000011	I

mnemonic	spec	funct7	funct3	opcode	encoding
LH	RV32I		001	0000011	I
LW	RV32I		010	0000011	I
LBU	RV32I		100	0000011	I
LHU	RV32I		101	0000011	I
SB	RV32I		000	0100011	S
SH	RV32I		001	0100011	S
SW	RV32I		010	0100011	S
ADDI	RV32I		000	0010011	I
SLTI	RV32I		010	0010011	I
SLTIU	RV32I		011	0010011	I
XORI	RV32I		100	0010011	I
ORI	RV32I		110	0010011	I
ANDI	RV32I		111	0010011	I
SLLI	RV32I	0000000	001	0010011	R
SRLI	RV32I	0000000	101	0010011	R
SRAI	RV32I	0100000	101	0010011	R
ADD	RV32I	0000000	000	0110011	R
SUB	RV32I	0100000	000	0110011	R
SLL	RV32I	0000000	001	0110011	R
SLT	RV32I	0000000	010	0110011	R
SLTU	RV32I	0000000	011	0110011	R
XOR	RV32I	0000000	100	0110011	R
SRL	RV32I	0000000	101	0110011	R
SRA	RV32I	0100000	101	0110011	R
OR	RV32I	0000000	110	0110011	R
AND	RV32I	0000000	111	0110011	R
FENCE	RV32I		000	0001111	I

mnemonic	spec	funct7	funct3	opcode	encoding
FENCE.I	RV32I		001	0001111	I
ECALL	RV32I		000	1110011	I
EBREAK	RV32I		000	1110011	I
CSRRW	RV32I		001	1110011	I
CSRRS	RV32I		010	1110011	I
CSRRC	RV32I		011	1110011	I
CSRRWI	RV32I		101	1110011	I
CSRRSI	RV32I		110	1110011	I
CSRRCI	RV32I		111	1110011	I
MUL	RV32M	0000001	000	0110011	R
MULH	RV32M	0000001	001	0110011	R
MULHSU	RV32M	0000001	010	0110011	R
MULHU	RV32M	0000001	011	0110011	R
DIV	RV32M	0000001	100	0110011	R
DIVU	RV32M	0000001	101	0110011	R
REM	RV32M	0000001	110	0110011	R
REMU	RV32M	0000001	111	0110011	R

Appendix H – ALU Operation Table

This table is also given in the lecture slides, but is reproduced here for convenience.

Note that the ALU B input is used as the shift amount (shamt) for shift instructions, since it would otherwise be unused.

4-bit opcode	function
0000	A and B
0001	A or B
0010	A xor B

4-bit opcode	function
0011	A + B
0100	A - B
0101	A * B (low, signed)
0110	A * B (high, signed)
0111	A * B (high, unsigned)
1000	A « shamt
1001	A » shamt
1010	A »> shamt
1100	A < B (signed)
1101	A < B (unsigned)
1110	A < B (unsigned)
1111	A < B (unsigned)

