Completing Your RISC-V CPU

Table of contents

Introduction

Now that our test program is executing properly, let's go back and complete the logic for the remaining instructions.

Learning Objectives

This chapter serves to:

- Reenforce the concepts from prior chapters;
- Complete your understanding of the base RISC-V ISA.

Test Program

We will provide you with a new test program that tests each instruction in the RV32-I instruction set.

In place of the test program used in the previous chapter (everything delimited by **//----------** comments), instantiate this macro (properly indented): **m4_test_prog()** and compile/simulate.

Since this new program comes from an included macro, you can no longer see it or edit it in the source code, nor will it be visible in NAV-TLV, but it should be visible now in VIZ.

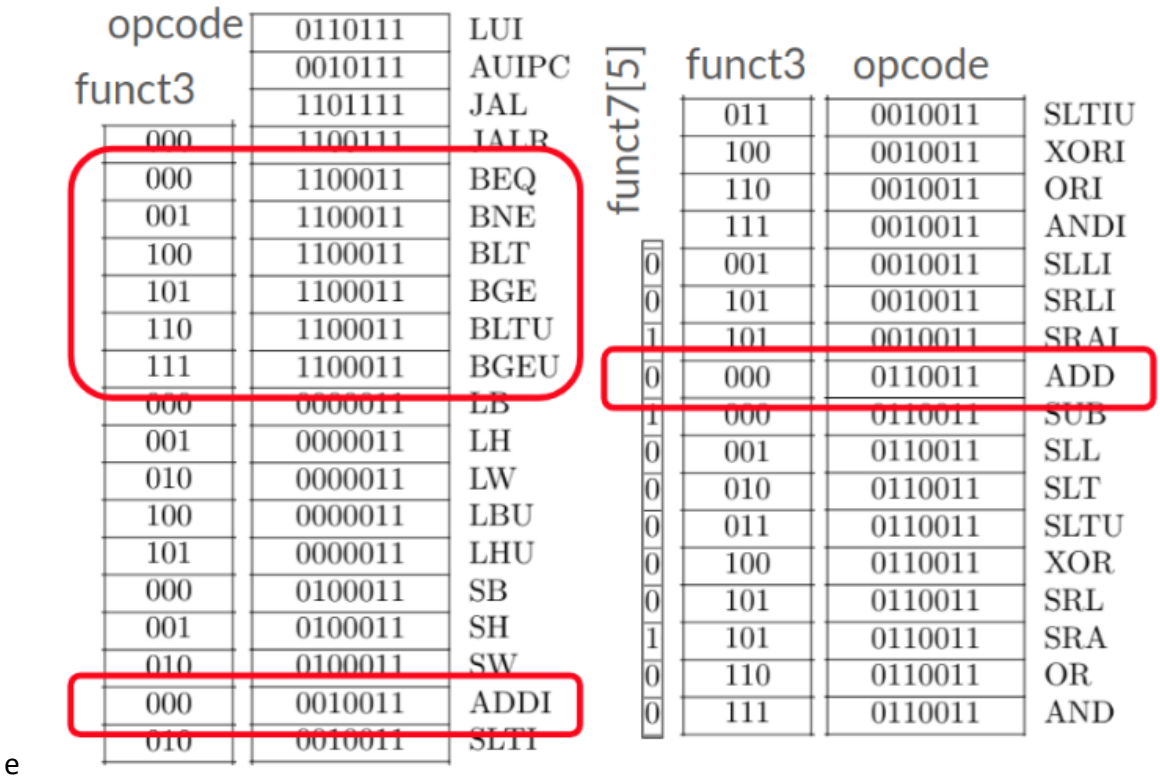
For the remaining exercises, it will be easier to debug in hexadecimal. The **m4_test_prog()** macro configures VIZ to now display register values in hexadecimal. If you are not yet comfortable with hexadecimal, this will be good practice. Remember, each hexadecimal digit represents four binary digits.

This test program executes each instruction once, each producing a result in a unique register starting from x5 and increasing from there. For each, it XORs the result with a value that will produce a 1 if it was correct. If all instructions are working, registers x5-x27 will contain 1 when the test completes (and x28-x30 are written with 1 as well). You can use VIZ to determine which instructions produced incorrect values and debug the issues. Of course, you haven't implemented most instructions yet, so most registers will not currently be written with 1s.

We'll continue to use the same **m4+tb()** test bench. It will report "Passed" once the program terminates properly, but this test bench does not check that the register values are 1. You must check this in VIZ.

Decode Logic

Previously, you implemented decode logic for the instructions circled in red.



funct3	opcode	
	0110111	LUI
	0010111	AUIPC
	1101111	JAL
000	1100111	JALR
000	1100011	BEQ
001	1100011	BNE
100	1100011	BLT
101	1100011	BGE
110	1100011	BLTU
111	1100011	BGEU
000	0000011	LB
001	0000011	LH
010	0000011	LW
100	0000011	LBU
101	0000011	LHU
000	0100011	SB
001	0100011	SH
010	0100011	SW
000	0010011	ADDI
010	0010011	SLTI

funct7[5]	funct3	opcode	
	011	0010011	SLTIU
	100	0010011	XORI
	110	0010011	ORI
	111	0010011	ANDI
0	001	0010011	SLLI
0	101	0010011	SRLI
1	101	0010011	SRAI
0	000	0110011	ADD
1	000	0110011	SUB
0	001	0110011	SLL
0	010	0110011	SLT
0	011	0110011	SLTU
0	100	0110011	XOR
0	101	0110011	SRL
1	101	0110011	SRA
0	110	0110011	OR
0	111	0110011	AND

e

Picture 1. Instruction decode table

Check the box for each completed step, to ensure none is skipped:

– With the exception of load and store instructions (LB, LH, LW, LBU, LHU, SB, SH, SW), complete the decode logic for the remaining non-circled instructions above (**$is_<instr> = …**). Remember, you can use "x" for don't-care bits;

– Our implementation will treat all loads and all stores the same, so assign **$is_load** based on opcode only. **$is_s_instr** already identifies stores, so we do not need any additional decode logic for stores;

– Compile, and note that VIZ instruction decode now shows instruction mnemonics. Note that the LOG will be full of warnings for all of these unused signals, but we'll clean these up next.

As your design gets larger, it is possible, though unlikely, that the DIAGRAM may fail to generate properly. This may be an inconvenience, but is not necessarily an issue with your design.

Arithmetic Logic Unit

Now we will add support in the ALU for the remaining instructions. We do this by extending the assignment statement for **$result**. Since there will be an expression for almost every instruction, there is a lot of code to write here. We'll provide the expressions, but we'll ask you to do the typing yourself so

you have a chance to reflect on each instruction. If you'd like more information about these instructions, the RISC-V green card [1] is a useful reference, or you can reference the RISC-V Unprivileged ISA Specification [2].

The existing expressions for ADD and ADDI are pretty simple. Most of the other instructions have simple expressions as well, but a few are more complex. A few have common subexpressions, so let's first create assignments for these subexpressions.

```
// SLTU and SLTI (set if less than, unsigned) results:
$sltu_rslt[31:0]  = {31'b0, $src1_value < $src2_value};
$sltiu_rslt[31:0] = {31'b0, $src1_value < $imm};

// SRA and SRAI (shift right, arithmetic) results:
//   sign-extended src1
$sext_src1[63:0] = { {32{$src1_value[31]}}, $src1_value };
//   64-bit sign-extended results, to be truncated
$sra_rslt[63:0] = $sext_src1 >> $src2_value[4:0];
$srai_rslt[63:0] = $sext_src1 >> $imm[4:0];
```

Picture 2. Subexpressions needed by the ALU

Complete and check off each step:

– Enter the assignment statements above before the existing assignment of $result. Think about these expressions as you type them;

– Compile/simulate and debug any errors in LOG;

Now, to implement the complete ALU. We provide you with the expressions, some of which use the subexpressions you just implemented.

[1] https://inst.eecs.berkeley.edu/~cs61c/fa17/img/riscvcard.pdf

[2] https://riscv.org/technical/specifications/

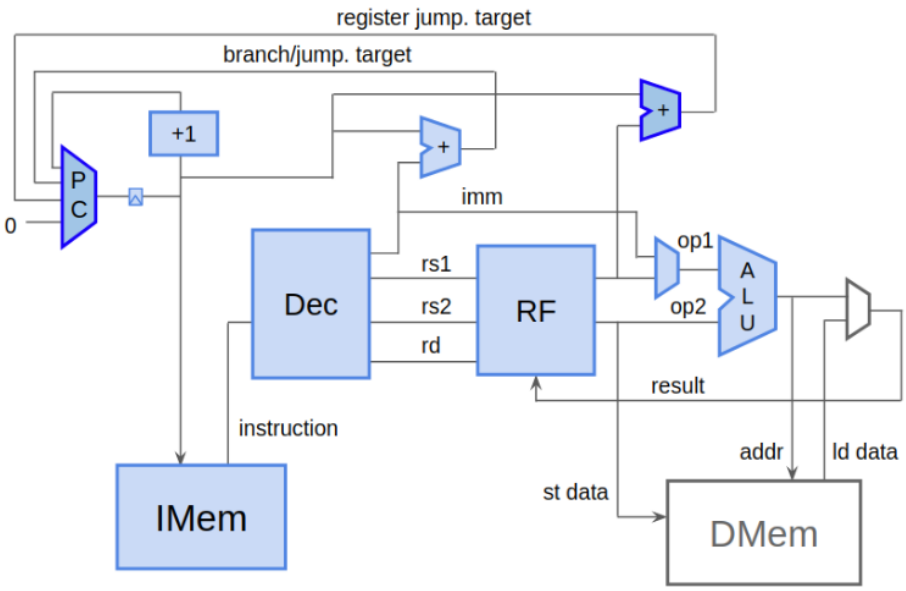
```
ANDI:   $src1_value & $imm
ORI:    $src1_value | $imm
XORI:   $src1_value ^ $imm
ADDI:   $src1_value + $imm
SLLI:   $src1_value << $imm[5:0]
SRLI:   $src1_value >> $imm[5:0]
AND:    $src1_value & $src2_value
OR:     $src1_value | $src2_value
XOR:    $src1_value ^ $src2_value
ADD:    $src1_value + $src2_value
SUB:    $src1_value - $src2_value
SLL:    $src1_value << $src2_value[4:0]
SRL:    $src1_value >> $src2_value[4:0]
SLTU:   $sltu_rslt
SLTIU:  $sltiu_rslt
LUI:    {$imm[31:12], 12'b0}
AUIPC:  $pc + $imm
JAL:    $pc + 32'd4
JALR:   $pc + 32'd4
SLT:    ( ($src1_value[31] == $src2_value[31]) ?
              $sltu_rslt :
              {31'b0, $src1_value[31]} )
SLTI:   ( ($src1_value[31] == $imm[31]) ?
              $sltiu_rslt :
              {31'b0, $src1_value[31]} )
SRA:    $sra_rslt[31:0]
SRAI:   $srai_rslt[31:0]
```

Picture 3. Result value TL-Verilog expressions for the ALU for each instruction

– Extend the expression for **$result** to complete the ALU to support the remaining instructions;

– If any of these new instructions are not resulting in register values of 1 in VIZ, debug them. To be specific, at the end of the simulation, register values should be 1 except x0-4, x27, and x31. Save your work outside of Makerchip.

Jump Logic



Picture 4. Implementing jump logic

The ISA, in addition to conditional branches, also supports jump instructions (which some other ISAs refer to as "unconditional branches"). RISC-V has two forms of jump instructions:

– JAL – Jump and link. Jumps to PC + IMM (like branches, so this target is **$br_tgt_pc**, already assigned);

JALR – Jump and link register. Jumps to SRC1 + IMM;

"And link" refers to the fact that these instructions capture their original PC + 4 in a destination register, as you already coded in the ALU. (The link register is particularly useful for jumps that are used to implement function calls, which must return to the link address after function execution).

Check the box for each completed step, to ensure none is skipped:

– Compute $jalr_tgt_pc[31:0] (SRC1 + IMM).

– Update the PC logic to select the correct **$next_pc** for JAL (**$br_tgt_pc**) and JALR (**$jalr_tgt_pc**). In the test program, JAL and JALR instructions should jump to the next subsequent instruction (as if not jumping at all), with the exception of the final JAL, which should jump to itself. Assuming x30 is also properly set to 1, this final JAL will result in the test reporting "Passed" in LOG and VIZ (though loads and stores are not working yet). Verify this behavior in VIZ.

Load, Store, and Data Memory

Addressing Memory

So far, all of our instructions are operating on register values. What good is a CPU if it has no memory? Let's add some. But first, let's prepare the load and store instructions that will read from and write to this memory.

Both load and store instructions require an address from which to read, or to which to write. As with the IMem, this is a byte-address. Loads and stores can read/write single bytes, half-words (2 bytes), or words (4 bytes/32 bits).

Picture 5. Word, Half, Byte

We will, however, avoid this nuance and implement all load/store instructions to operate on words, assuming that the lowest two address bits are zero. In other words, we are assuming work loads/stores with *naturally-aligned* addresses.

The address for loads/stores is computed based on the value from a source register and an offset value (often zero) provided as the immediate:

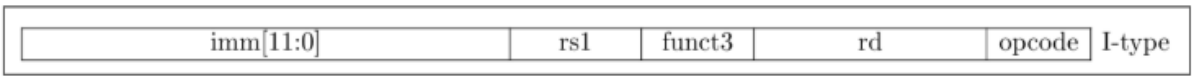
```
addr = rs1 + imm
```

Loads

A load instruction (LW,LH,LB,LHU,LBU) takes the form:

LOAD rd, imm(rs1).

It uses the I-type instruction format:



Picture 6. I-type instruction format

It writes its destination register with a value read from the specified address of memory, which we can denote as:

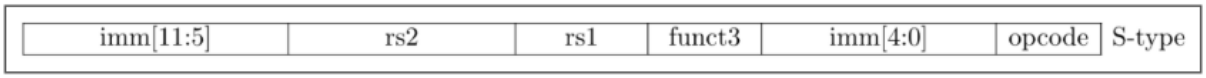
rd <= DMem[addr] (where, **addr = rs1 + imm**)

Stores

A store instruction (SW,SH,SB) takes the form:

STORE rs2, imm(rs1).

It has its own S-type instruction format:



Picture 6. S-type instruction format

It writes the specified address of memory with a value from the rs2 source register:

DMem[addr] <= rs2 (where, **addr = rs1 + imm**)

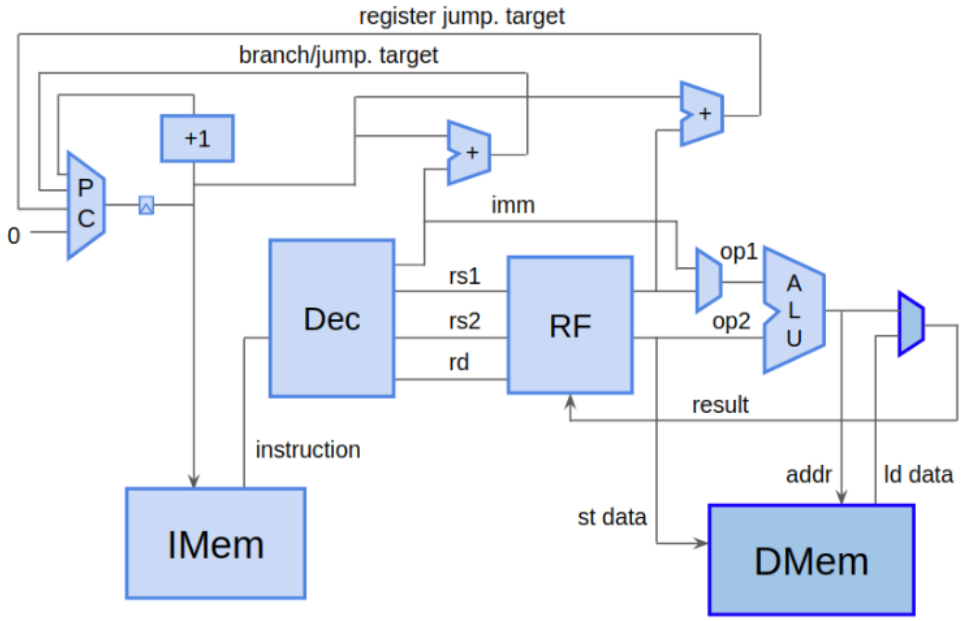
Address Logic

The address computation, **rs1 + imm**, is the same computation performed by ADDI. Since load/store instructions do not otherwise require the ALU, we will utilize the ALU for this computation.

Check the box after completing the step:

For loads/stores (**$is_load**/**$is_s_instr**), compute **$result** as the address (**rs1 + imm**), as with the ADDI instruction. (This change will not be visible in VIZ, yet.)

Data Memory



Picture 7. Implementing data memory

To keep our simulations zippy, we'll instantiate a very small data memory--the same size as our register file.

Unlike our register file, which is capable of reading two values each cycle and, on the same cycle, writing a value, our memory needs only to read one value or write one value each cycle to process a load or a store instruction. Similar to our register file, our DMem is word-granular. Recall that we are supporting only work loads/stores with naturally-aligned addresses (so the lower two bits zero are assumed to be zero).

Based on the discussion above:

– write is enabled for stores ($is_s_instr);

– read is enabled for loads ($is_load);

– the ALU result (**$result**) provides the read/write address; this is a byte address, while our memory is indexed by 32-bit words;
– rs2 (**$src2_value**) provides the write data;
– the only output of the DMem is the load data (which we'll call **$ld_data**).

Complete and check off each step:

– Similar to what we did for the register file, there is a commented macro instantiation for **m4+dmem(32, 32, $reset, $addr[4:0], $wr_en, $wr_data[31:0], $rd_en, $rd_data)**. Uncomment it;

– Provide proper macro arguments to connect the correct input and output signals. Be sure to extract the appropriate bits of the byte address to drive the DMem's word address. Since the memory has a single read port, fewer arguments are needed for the DMem than for the RF;

– Compile/simulate, and debug compilation errors;

The load data (**$ld_data**) coming from DMem must be written to the register file. A new multiplexer is needed to select **$ld_data** for load instructions, as depicted in the figure.

– Add this new multiplexer to write **$ld_data**, rather than **$result**, to the register file, when **$is_load** asserts;

– Debug compilation errors. Your LOG should be clean at this point (no errors or warnings);

The test program, toward the end, does a store and a load of the hexadecimal value **32'h15**.

– Examine the store (SW) and load (LW) instructions in VIZ. Confirm that the value **'h15** is stored to memory location 2 and loaded into register x27;

– Confirm that x5-x30 are all 1 at the end of a passing simulation.

That’s it! You’ve got a working RISC-V core! Be sure to save all of your hard work.

Next Steps

If you are wondering what your next steps in the RISC-V journey may be, here are a few:

– Continue exploring Makerchip to dig deeper into TL-Verilog and its ecosystem;
– Revisit the course repository, which may be updated with recent opportunities;
– Explore more about RISC-V at riscv.org.
– Explore other course offerings from The Linux Foundation, and stay tuned for more RISC-V courses.