RISC-V-Subset CPU

Table of contents

RISC-V-Subset CPU *1*

Introduction *2*

Learning Objectives 2

CPU Labs Setup *3*

Showcasing Your Work 3

Opening the Starting-Point Code 3

CPU Microarchitecture and Implementation Plan *4*

PC Logic *6*

Instruction Memory *7*

Decode Logic *9*

Instruction Type 9

Instruction Fields 10

Instruction 12

Register File Read *14*

Arithmetic Logic Unit *16*

Register File Write *17*

Branch Logic *17*

Introduction

In this chapter, you will build a subset of your RISC-V CPU core capable of executing a test program that adds numbers from 1 to 9. Subsequently, you will complete the functionality of your core.

Learning Objectives

By the end of this chapter, you should be able to:

– Explain the role of the fundamental components of a basic CPU microarchitecture;

– Be experienced in expressing digital logic using TL-Verilog;

– Develop an appreciation for the debug process within Makerchip, including:

  – the interpretation of messages in the logs;

  – use of visual debug to understand the overall behavior of your logic;

  – use of the waveform viewer to understand detailed behavior;

  – tracing faulty behavior from symptom to cause.

– Instantiate pre-existing Verilog and TL-Verilog components.

CPU Labs Setup

Showcasing Your Work

Now that you are preparing to create something more substantial, it’s worth considering a few options for how you will develop, save, and showcase your design (all of which are optional).

Consider capturing your work in GitHub (or another Git hosting platform). If you are or will be job hunting, your GitHub profile often speaks more strongly to a potential employer than your resume. GitHub is also a great place to snapshot your code as you develop it to be sure you don’t lose your work. You can create a fresh repository for your work, or fork the course repository. You can edit files directly via GitHub’s web interface and paste your code from Makerchip, all within your browser, or you can clone your repository on your local system and paste your code into a text editor.

We also have a convenient option for working with local files on your own desktop, whether in a git repository or not. You can launch Makerchip from your desktop to work with a local TL-Verilog source file. Makerchip runs in your browser, but autosaves back to your desktop.

Only if you’d like to try this workflow, first get the starting point code locally by cloning the course GitHub repository.

git clone https://github.com/stevehoover/LF-Building-a-RISC-V-CPU-Core.git

(and enter your credentials)

Then install the Makerchip app:

pip3 install makerchip-app

Opening the Starting-Point Code

Now, open the starting-point code template. As you complete each step, check the box to ensure no step is skipped.

– Only if you are using the Makerchip app to edit code on your local computer, first copy the code from **LF-Building-a-RISC-V-CPU-Core/risc-v_shell.tlv** to wherever you would like to edit it, then: **makerchip <path>/risc-v_shell.tlv**

– Otherwise, simply click this link to open the starting code in your browser, and, once the code loads, "*Save as New Project*".

– Simulation should run. The LOG should report "*Simulation FAILED!!!*" (and will until this chapter is successfully completed). VIZ should show the test program and signals that have not yet been implemented. (Mouse-wheel-down or use the "-" button to bring these fully into view).

Whether you are using the Makerchip app or not, after each lab, quickly check that your work has been properly autosaved to the cloud or to your local system by checking the status bar within Makerchip.

Generally, if you get off track and need to get back to a stable version of your code, **Ctrl-Z** will do the trick, but it might also be wise to save snapshots (or git commits) on occasion.

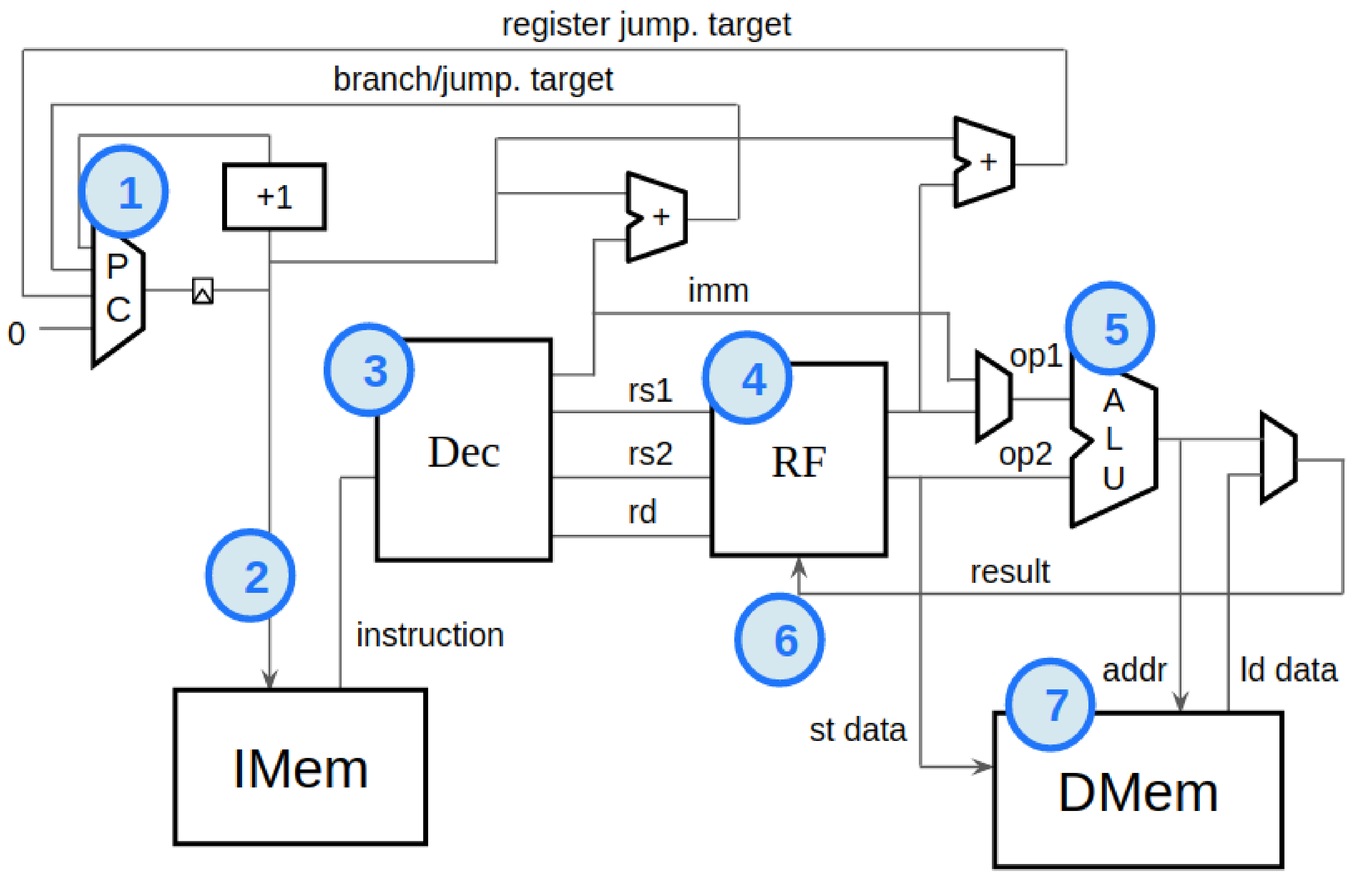
CPU Microarchitecture and Implementation Plan

CPUs come in many flavors, from small microcontrollers, optimized for small area and low power, to desktop and server processors, optimized for performance. Within several hours, you will construct a CPU core that could be appropriate as a microcontroller. In contrast, a desktop or server CPU chip might be built by a team of hundreds of seasoned engineers over a period of several years.

Our CPU will fully execute one instruction with each new clock cycle. Doing all of this work within a single clock cycle is only possible if the clock is running relatively slowly, which is our assumption.

We will start by implementing enough of the CPU to execute our test program. As you add each new piece of functionality, you will see in the VIZ pane the behavior you implemented, with more and more of the test program executing correctly until it is successfully summing numbers from one to nine. Then we will go back to implement support for the bulk of the RV32I instruction set.

Let’s look at the components of our CPU, following the flow of an instruction through the logic. This is also roughly the order in which we will implement the logic.



Picture 1. RISC-V CPU Block Diagram

– **PC Logic**

This logic is responsible for the program counter (PC). The PC identifies the instruction our CPU will execute next. Most instructions execute sequentially, meaning the default behavior of the PC is to increment to the following instruction each clock cycle. Branch and jump instructions, however, are non-sequential. They specify a target instruction to execute next, and the PC logic must update the PC accordingly.

– **Fetch**

The instruction memory (IMem) holds the instructions to execute. To read the IMem, or "fetch", we simply pull out the instruction pointed to by the PC.

– **Decode Logic**

Now that we have an instruction to execute, we must interpret, or decode, it. We must break it into fields based on its type. These fields would tell us which registers to read, which operation to perform, etc.

– **Register File Read**

The register file is a small local storage of values the program is actively working with. We decoded the instruction to determine which registers we need to operate on. Now, we need to read those registers from the register file.

– **Arithmetic Logic Unit (ALU)**

Now that we have the register values, it’s time to operate on them. This is the job of the ALU. It will add, subtract, multiply, shift, etc, based on the operation specified in the instruction.

– **Register File Write**

Now the result value from the ALU can be written back to the destination register specified in the instruction.

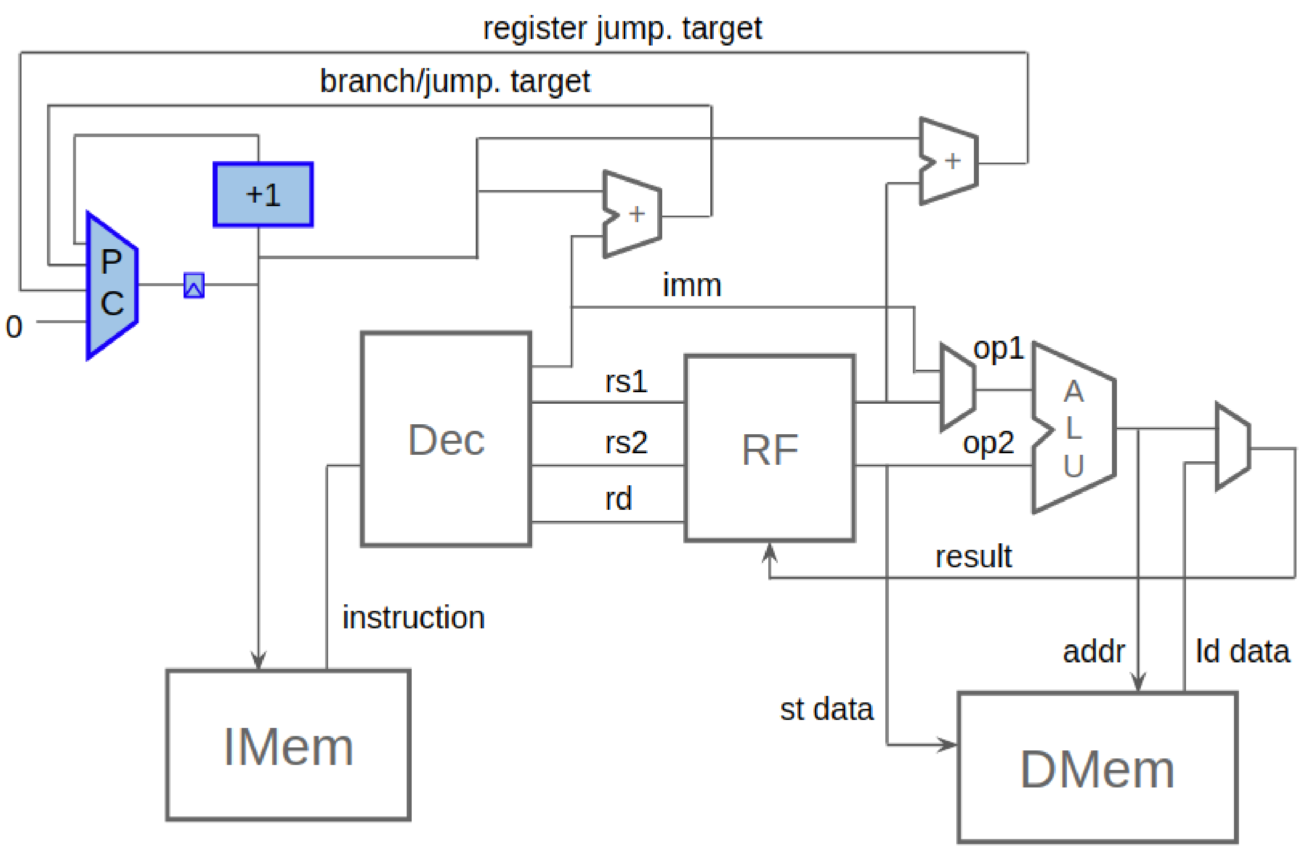
– **DMem**

Our test program executes entirely out of the register file and does not require a data memory (DMem). But no CPU is complete without one. The DMem is written to by store instructions and read from by load instructions.

In this course, we are focused on the CPU core only. We are ignoring all of the logic that would be necessary to interface with the surrounding system, such as input/output (I/O) controllers, interrupt logic, system timers, etc.

Notably, we are making simplifying assumptions about memory. A general-purpose CPU would typically have a large memory holding both instructions and data. At any reasonable clock speed, it would take many clock cycles to access memory. *Caches* would be used to hold recently-accessed memory data close to the CPU core. We are ignoring all of these sources of complexity. We are choosing to implement separate, and very small, instruction and data memories. It is typical to implement separate, single-cycle instruction and data caches, and our IMem and DMem are not unlike such caches.

PC Logic



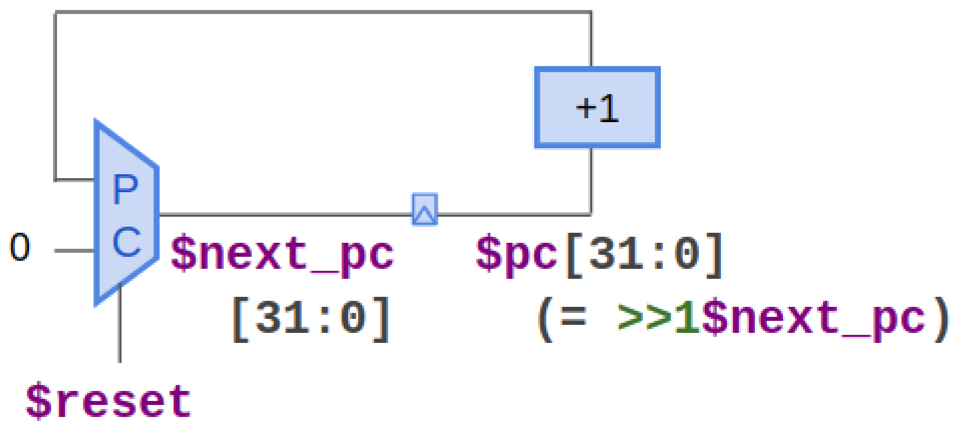
Picture 2. Implementing PC Logic

Initially, we will implement only sequential fetching, so the PC update will be, for now, simply a counter. Note that:

– The PC is a byte address, meaning it references the first byte of an instruction in the IMem. Instructions are 4 bytes long, so, although the PC increment is depicted as "+1" (instruction), the actual increment must be by 4 (bytes). The lowest two PC bits must always be zero in normal operation.

– Instruction fetching should start from address zero, so the first **$pc** value with **$reset** deasserted should be zero, as is implemented in the logic diagram below.

– Unlike our earlier counter circuit, for readability, we use unique names for **$pc** and **$next_pc**, by assigning **$pc** to the previous **$next_pc**.



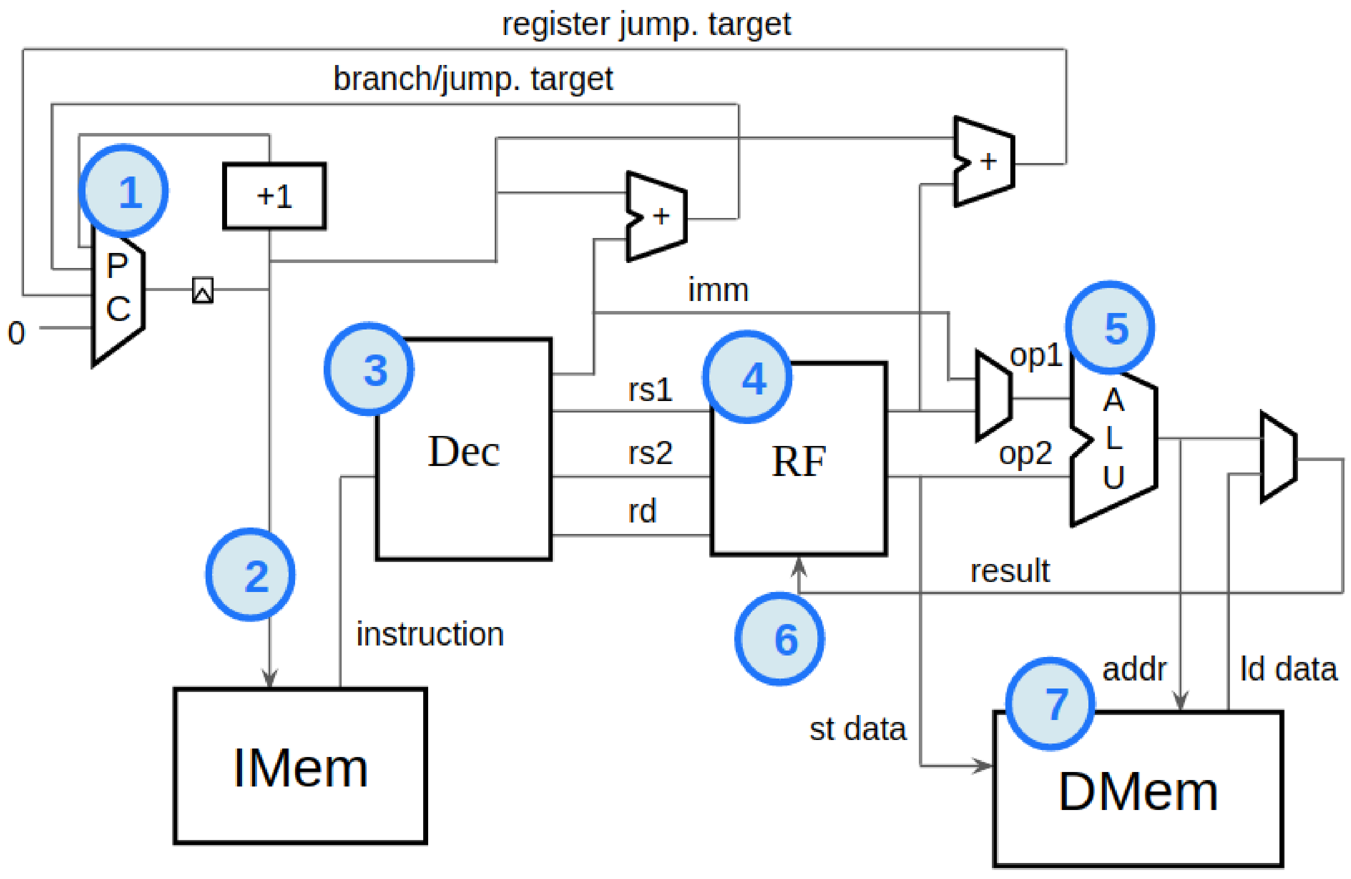
Picture 3. Initial PC Logic

Once the following step is completed, check the box.

– Implement the circuit above (replacing the "YOUR CODE HERE" comment), and confirm in VIZ and WAVEFORM that the PC is now starting at zero and incrementing as it should.

Remember, you can find reference solutions for all RISC-V labs in the course GitHub repository.

Instruction Memory



Picture 4. Implementing instruction memory

We will implement our IMem by instantiating a Verilog macro. This macro accepts a byte address as input, and produces the 32-bit read data as output. The macro can be instantiated, for example, as:

`READONLY_MEM($addr, $$read_data[31:0])

Verilog macro instantiation is preceded by a back-tick (not to be confused with a single quote).

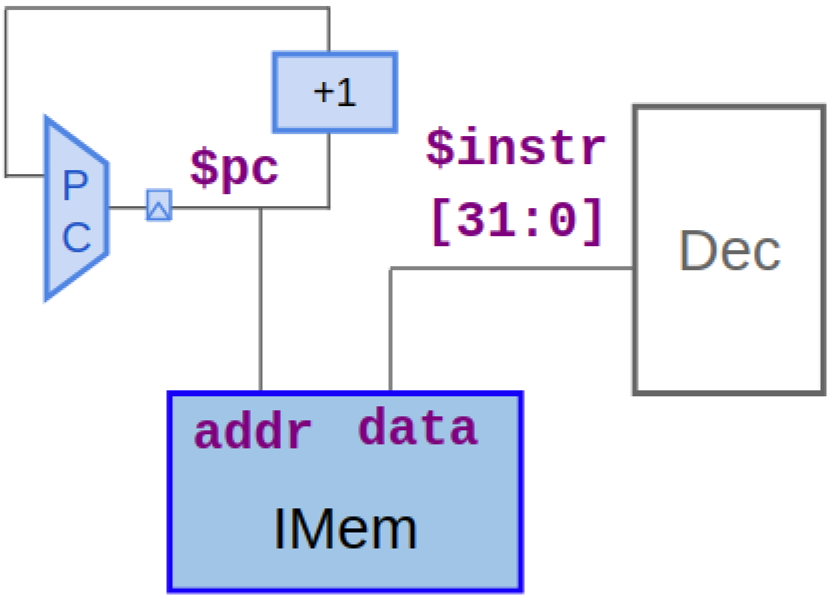
In expressions like this that do not syntactically differentiate assigned signals from consumed signals, it is necessary to identify assigned signals using a "$$" prefix. And, as always, an assigned signal declares its bit range. Thus, **$$read_data[31:0]** is used above.

This macro is simplified in several ways versus what you would typically see for an array macro:

– There is no way to write to our array. The program specified in the template is magically populated into this array for you.

– Typically, an array would have a read enable input as well. This read enable would indicate, each cycle, whether to perform a read. Our array will always read, and we are not concerned with the power savings a read enable could offer.

– Typically, a memory structure like our IMem would be implemented using a physical structure called static random access memory, or SRAM. The address would be provided in one clock cycle, and the data would be read out in the next cycle. Our entire CPU, however, will execute within a single clock cycle. Our array provides its output data on the same clock cycle as the input address. Our macro would result in an implementation using flip-flops that would be far less optimal than SRAM.



Picture 5. Instruction memory hookup

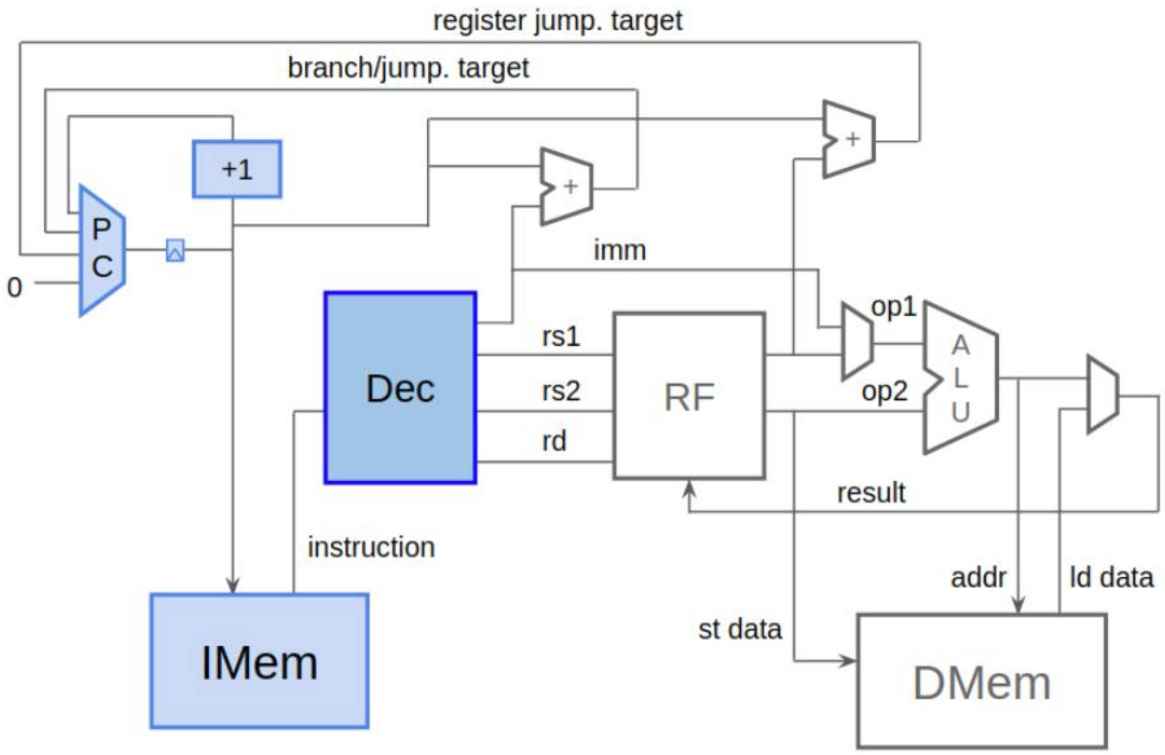
Implement instruction fetch by instantiating IMem with proper connectivity. Check the box for each completed step, to ensure none is skipped.

– Instantiate the **`READONLY_MEM** macro after your PC logic, providing **$pc** as the address and **$$instr[31:0]** as the output. Be sure to align this with other statements always using three spaces of indentation.

– Be sure that: the LOG indicates the dangling **$instr** output, the DIAGRAM looks right, and VIZ shows instructions being read from the IMem. If anything looks wrong, debug using WAVEFORM, and verify that **$instr** no longer appears in the VIZ "To Be Implemented" signals.

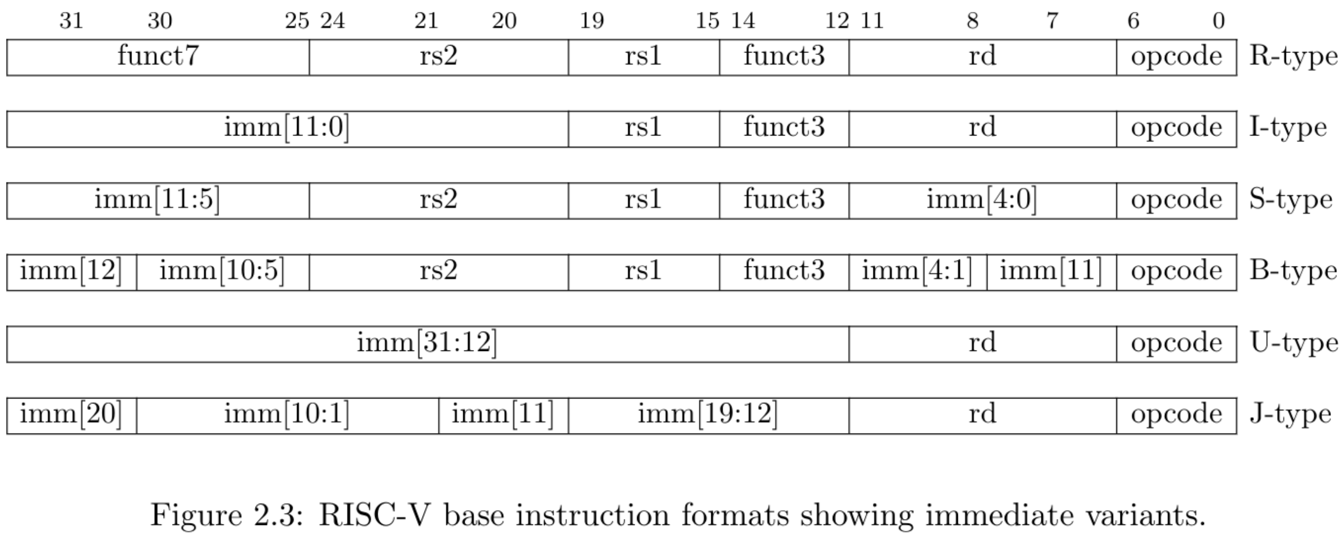
Decode Logic

Instruction Type



Picture 6. Implementing decode logic

Now that we have an instruction, let’s figure out what it is. Remember, RISC-V defines various instruction types that define the layout of the fields of the instruction, according to this table from the RISC-V specifications:



31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

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

Picture 7. Base instruction formats showing instruction fields for each instruction type

Before we can interpret the instruction, we must know its type. This is determined by its opcode, in **$instr[6:0]**. In fact, **$instr[1:0]** must be **2'b11** for valid RV32I instructions. We will assume all instructions to be valid, so we can simply ignore these two bits. The ISA defines the instruction type to be determined as follows.

instr[4:2] instr[6:5]	000	001	010	011	100	101	110	111
00	I	I	-	-	I	U	I	-
01	S	S	-	R	R	U	R	-
10	R4	R4	R4	R4	R	-	-	-
11	B	I	-	J	I (unused)	-	-	-

Picture 8. Instruction types from opcode[6:2] (instr[6:2])

You'll assign a boolean signal for each instruction type that indicates whether the instruction is of that type. For example, we could decode U-type as:

$is_u_instr = $instr[6:2] == 5'b00101 ||

$instr[6:2] == 5'b01101;

Complete and check off this step:

– Observe how the binary values in this expression correspond to the two U-type boxes in the table. SystemVerilog gives us an operator that makes this comparison a little simpler:

$is_u_instr = $instr[6:2] ==? 5'b0x101;

The **==?** operator above allows some bits to be excluded from the comparison by specifying them as "x" (referred to as *don't-care*).

Complete and check off the following steps:

– Add this assignment statement to your code and write the remaining 5 statements for I, R, S, B, and J instruction types. (Gray cells can be ignored as these are not used in RV32I.)
– Compile and simulate. Review LOGs. The VIZ "Instr. Decode" box should indicate the instruction type now. Debug as needed using the WAVEFORM.

Instruction Fields

Now, based on the instruction type, we can extract the instruction fields. Most fields always come from the same bits regardless of the instruction type but only have meaning for certain instruction types. The **imm** field, an "immediate" value embedded in the instruction itself, is the exception. It is constructed from different bits depending on the instruction type.

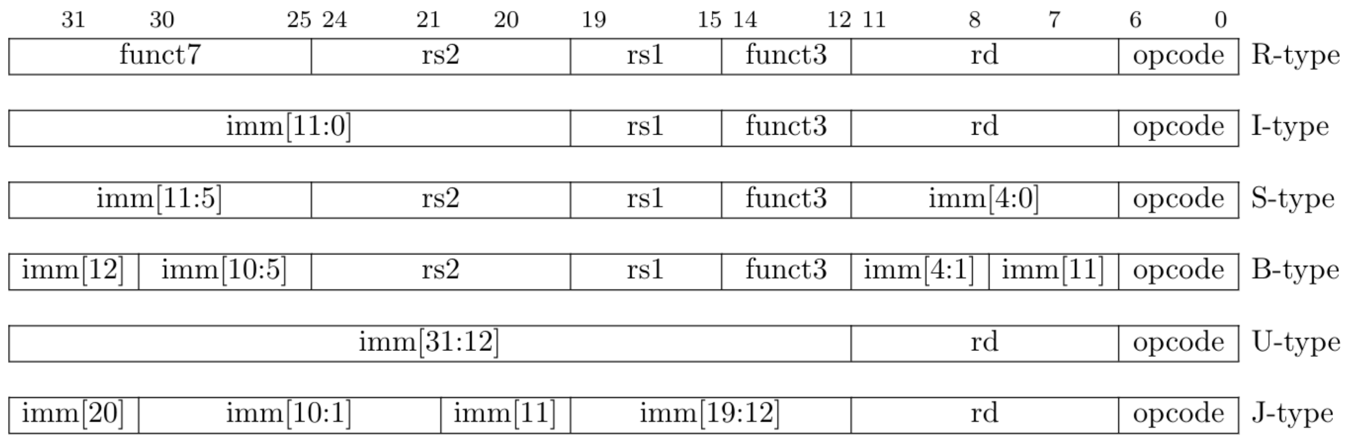

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

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

Picture 9. Base instruction formats showing immediate variants (repeated for reference)

Let's start with the simpler, non-immediate fields: **$funct3**, **$rs1**, **$rs2**, **$rd**, **$opcode**. We will not use **$funct7**, so you can skip this field. Check the box for each completed step, to ensure none is skipped.

– Extract these fields, for example: **$rs2[4:0] = $instr[24:20];**

– Compile/simulate, check LOG (with warnings for these new dangling signals), and debug. As you add these signals, they should be removed from the "To Be Implemented" list in VIZ;

– Determine when these fields are valid (excluding **$opcode**, which is always valid). For example:

$rs2_valid = $is_r_instr || $is_s_instr || $is_b_instr;

Provide **$imm_valid** as well, asserting for all types but R, even though we haven't determined **$imm** yet

– Compile/simulate, check LOG (now rather lengthy) and debug. Your "To Be Implemented" signals list should be getting shorter;

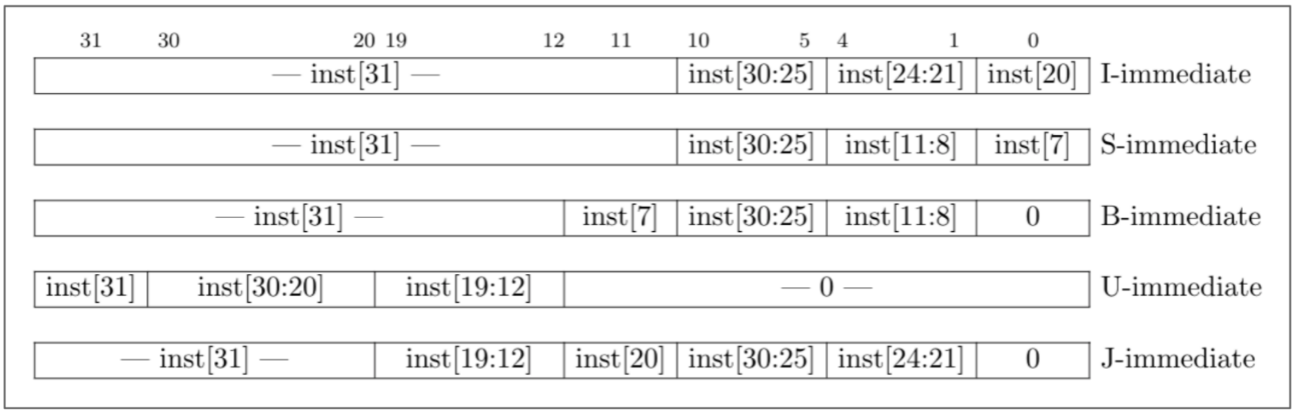
– All this clutter in our LOG is getting annoying. Suppress these warnings using the following:

`BOGUS_USE($rd $rd_valid $rs1 $rs1_valid ...)

This produces no logic, but looks like a signal consumption, so warnings are suppressed. Note that there are no commas between signals. Also note that if you extend this expression to a second line, this second line must be indented with spaces relative to the first. This line could be removed after signals are used, though some of these signals will only be used for VIZ and will remain unconsumed by your logic.

– Compile/simulate and confirm that the LOG is now clean. You should now see register indices in VIZ (in the blue portion of instruction decode). Confirm that they seem correct, and debug if necessary.

The immediate value is a bit more complicated. It is composed of bits from different fields depending on the type.


31 30 20 19 12 11 10 5 4 1 0
— inst[31] — | inst[30:25] | inst[24:21] | inst[20] | I-immediate
— inst[31] — | inst[30:25] | inst[11:8] | inst[7] | S-immediate
— inst[31] — | inst[7] | inst[30:25] | inst[11:8] | 0 | B-immediate
inst[31] | inst[30:20] | inst[19:12] | — 0 — | U-immediate
— inst[31] — | inst[19:12] | inst[20] | inst[30:25] | inst[24:21] | 0 | J-immediate

Picture 10. Base instruction formats

The immediate value for I-type instructions, for example is formed from 21 copies of instruction bit 31, followed by **inst[30:20]** (which is broken into three fields above for consistency with other formats).

The immediate field can be formed, based on this table, using a logic expression like the following. It uses a combination of bit extraction (e.g. **$instr[30:20]**), bit replication (e.g. {21{...}}), and bit concatenation (e.g. {..., …}):

```
  $imm[31:0] = $is_i_instr ? {  {21{$instr[31]}},  $instr[30:20]  } :
     $is_s_instr ? {...} :
     ...
             32'b0;  // Default
```

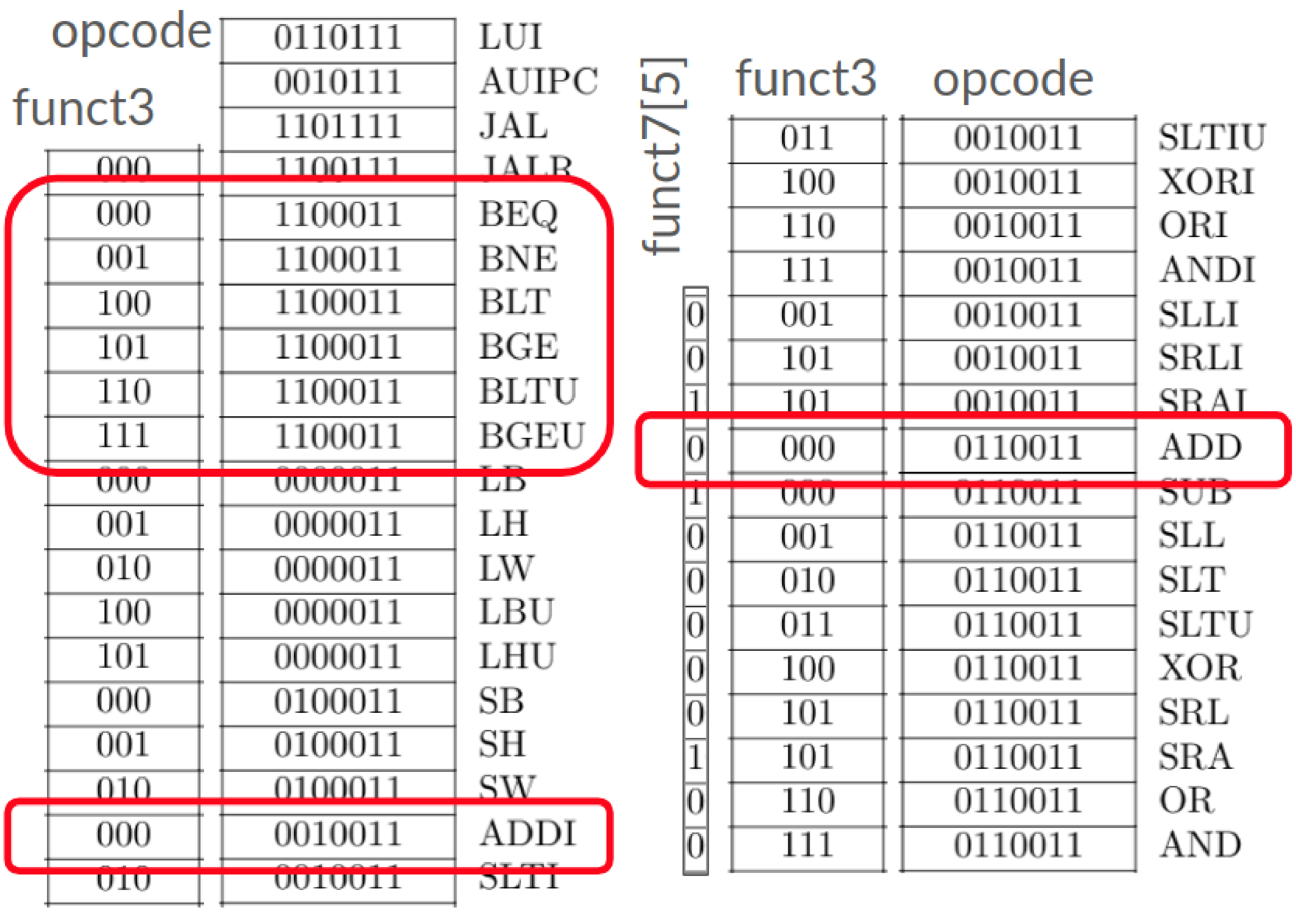
Complete and check off the following steps:

– Complete the above logic expression for **$imm**;

– Verify in WAVEFORM and VIZ that the value of **$imm** corresponds to the instructions in the test program. This will test your understanding of binary, decimal and hexadecimal. For example, the ADDI, x12, x10, 1010 instruction shows the immediate value in binary, VIZ should represent the value in decimal as i[10], and WAVEFORM should show a hexadecimal "a".

Instruction

Now we need to determine the specific instruction. This is determined from the **opcode**, **instr[30]**, and **funct3** fields as follows. Note that **instr[30]** is **$funct7[5**] for R-type, or **$imm[10]** for I-type and is labeled "**funct7[5]**" in the table below.



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

Picture 11. Instruction decode table, with needed instructions circled

Complete and check off each step:

– For convenience, concatenate the relevant fields into a single bit vector signal, as: **$dec_bits[10:0] = {$instr[30],$funct3,$opcode};**

– For each of the instructions circled in red (we'll come back and do the rest later), determine if **$dec_bits** identifies this instruction. For example:

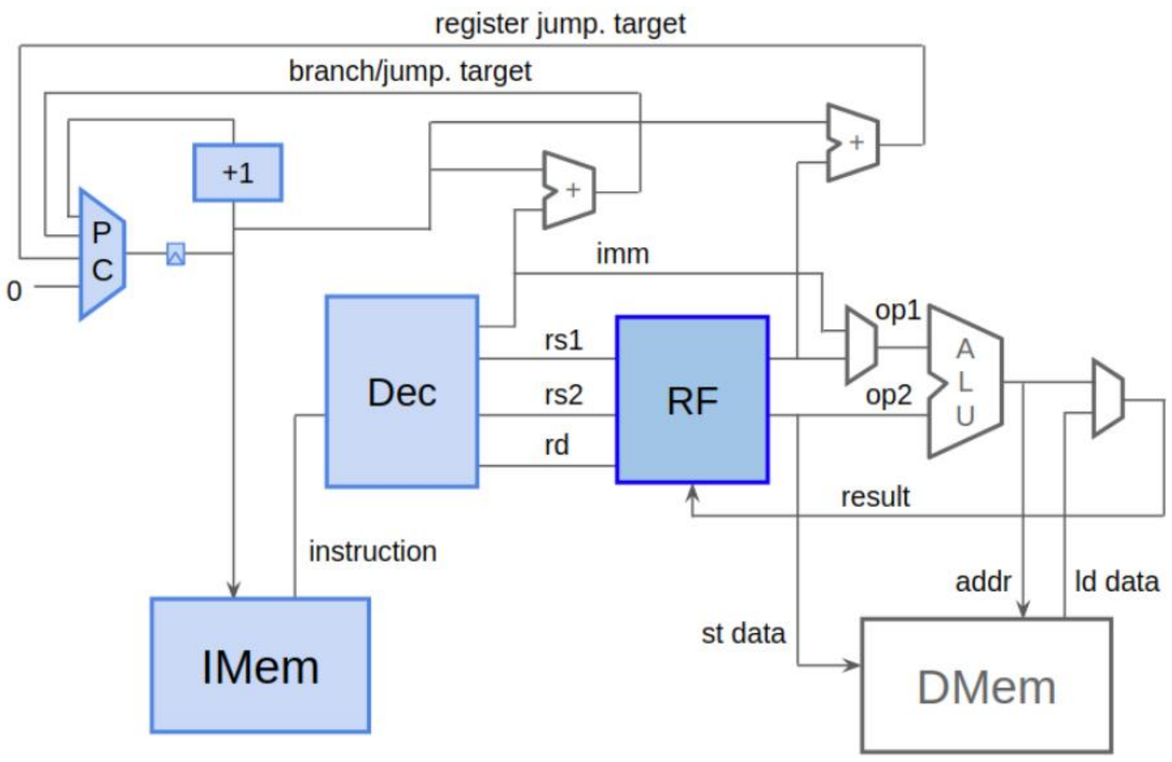
$is_beq = $dec_bits ==? 11'bx_000_1100011;

Note that underscores here are optional to help delimit fields. Also note, we're using "x" as a don't-care for the **instr[30]** bit, which is not used by BEQ (or any other instruction in the left column);

– Compile and simulate. Again, there will be many warnings for these unused signals. It is worthwhile to again use **'BOGUS_USE** to keep the LOG clean;

– Confirm that VIZ is now displaying the correct instruction *mnemonics* in the blue section of "Instr. Decode". Debug as needed.

Register File Read



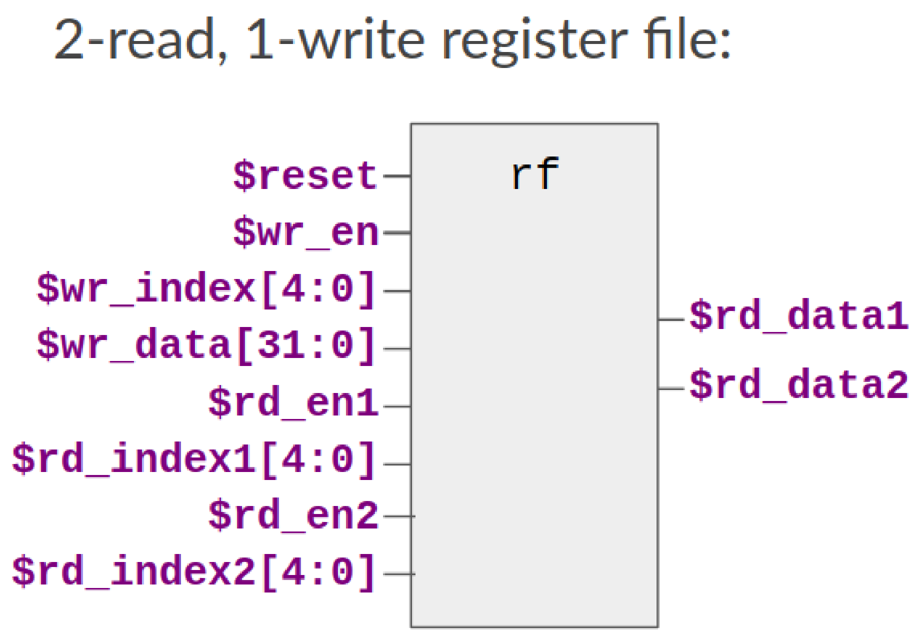
Picture 12. Implementing register file read

Like our mini IMem, the register file is a pretty typical array structure, so we can find a library component for it. This time, rather than using a Verilog module or macro as we did for IMem, we will use a TL-Verilog array definition, expanded by the M4 macro preprocessor.

Near the bottom of your code, and commented out, you'll find the following example instantiation of a register file (RF) macro:

//m4+rf(32, 32, $reset, $wr_en, $wr_index[4:0], $wr_data[31:0], $rd1_en, $rd1_index[4:0], $rd1_data, $rd2_en, $rd2_index[4:0], $rd2_data)

This would instantiate a 32-entry, 32-bit-wide register file connected to the given input and output signals, as depicted below. Each of the two read ports requires an index to read from and an enable signal that must assert when a read is required, and it produces read data as output (on the same cycle).



Picture 13. The provided register file instantiation (before you modify it)

For example, to read register 5 (x5) and register 8 (x8), **$rd_en1** and **$rd_en2** would both be asserted, and **$rd_index1** and **$rd_index2** would be driven with 5 and 8.

A few things to note:

– For this macro, output signal arguments are signal names. Inputs are expressions;

– Note that here we are using "rd" as an abbreviation for read, which is easily confused with the destination register to be written by an instruction which is also referred to in RISC-V as "rd";

You will modify the macro arguments related to register read to properly read the instructions' source register values. Check the box for each completed step, to ensure none is skipped.

– Uncomment the rf instantiation. (Noting that M4 macros are an experimental construct, this macro instantiation must remain on a (very long) single line. Also, keep it where it is, near the bottom of the file so its expansion does not clutter NAV-TLV.);

– Though you have not connected the macro properly yet, compile and simulate this, and feel free to explore the results with dangling signals. You'll see in NAV-TLV the expansion of this macro. Don't worry about the code; it uses syntax we haven't introduced, but you will see warnings/errors highlighted over the dangling input/output signals. As you connect this in the following steps, these warnings will be eliminated. You'll see messages in the LOG as well, and there will be disconnects in the DIAGRAM. You will see the register file in VIZ as well, though its behavior will be random until its inputs are properly connected.

Your instruction decode logic provides the signals needed for register file read. It determines, based on the instruction type, whether source registers are needed. It extracts the rs1 and rs2 fields which provide the indices for these registers if valid. Don't forget to check the box for each completed step, to ensure none is skipped.

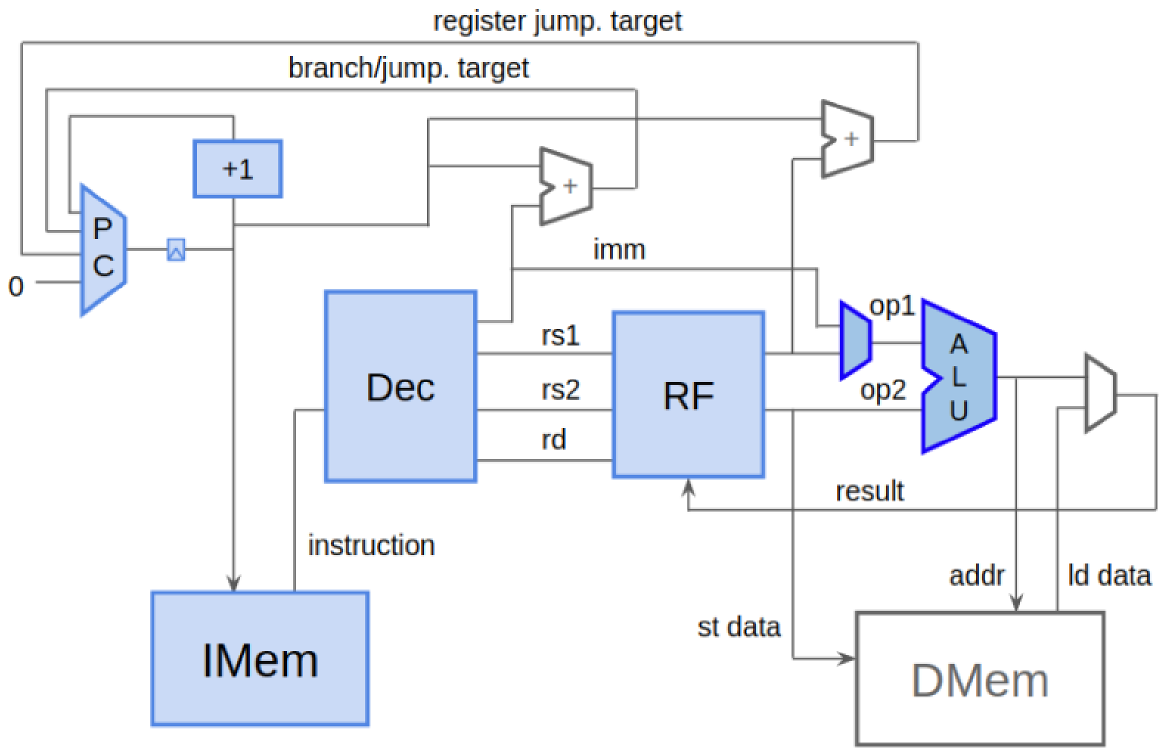
– Modify the appropriate RF macro arguments to connect the decode output signals to the register file read input signals to read the correct registers when they are needed;

– Connect the output read data to new signals named **$src1_value** and **$src2_value** by replacing the appropriate macro arguments with these new signal names. (Bit ranges are not needed, as they are explicit within the macro definition.);

– Compile/simulate. Observe that the register value in each entry of the register file is equal to the entry index. A more typical choice would have been to initialize all values to zero. We have provided non-zero initial values to simplify the next step;

– Confirm proper operation in VIZ by observing the source register being read from the RF. Save your work outside of Makerchip.

Arithmetic Logic Unit



Picture 14. Implementing the arithmetic logic unit

Now, you have source values to operate on, so let’s create the ALU. The ALU is much like our initial calculator circuit. It computes, for each possible instruction, the result that it would produce. It then selects, based on the actual instruction, which of those results is the correct one.

At this point, we are only going to implement support for the instructions in our test program. Since branch instructions do not produce a result value, we only need to support ADDI (which adds the immediate value to source register 1) and ADD (which adds the two source register values). Check the box one you completed this step.

Note an error in the diagram. The immediate value is used in place of **op2 (src2)**, not **op1 (src1)**.

– Use a structure like the following to assign the ALU **$result** in a single assignment expression for ADDI and ADD instructions:

```
$result[31:0] =
  $is_addi ? $src1_value + $imm :
  ...
        32'b0;
```

– Compile/simulate. You should now see computed results in the VIZ "Instr. Decode" box (though incorrect values write back to the register file).

Register File Write

$result needs to be written back to the destination register (rd) in the register file (if the instruction has a destination register). Complete and check off each step:

– Connect the register file's write inputs to perform this write-back for instructions that have a valid destination register.

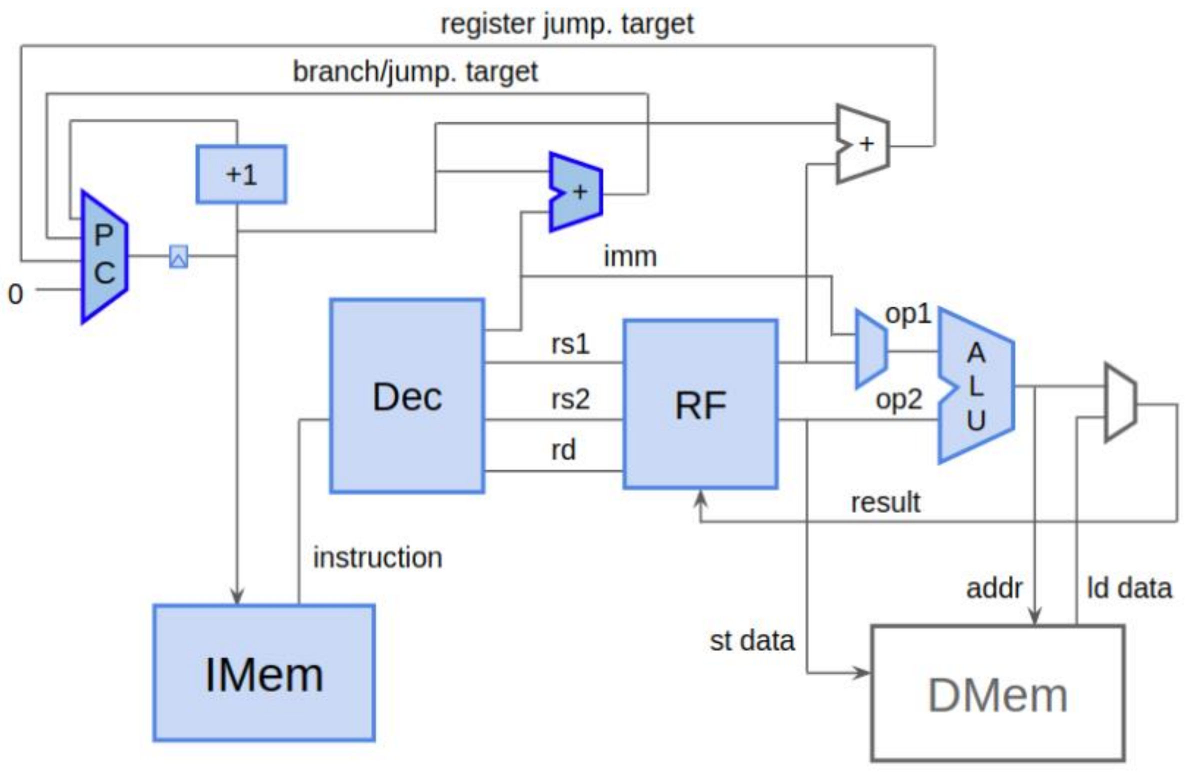
– Compile/simulate. Check LOG. And confirm using VIZ that the destination register is being written to the register file.

In RISC-V, **x0** (at register file index 0) is "always-zero". One way to implement this behavior is to avoid writing **x0**.

– Currently, our test program doesn't write **x0**, so we have no way to test this change. Add an instruction after the branch that writes a non-zero value to **x0**, and watch it write in VIZ.

– Modify your logic to deassert the register file write enable input if the destination register is 0. Compile/simulate, debug, and confirm in VIZ that the jump instruction no longer writes. Delete the added test instruction if you like, it won't matter.

Branch Logic

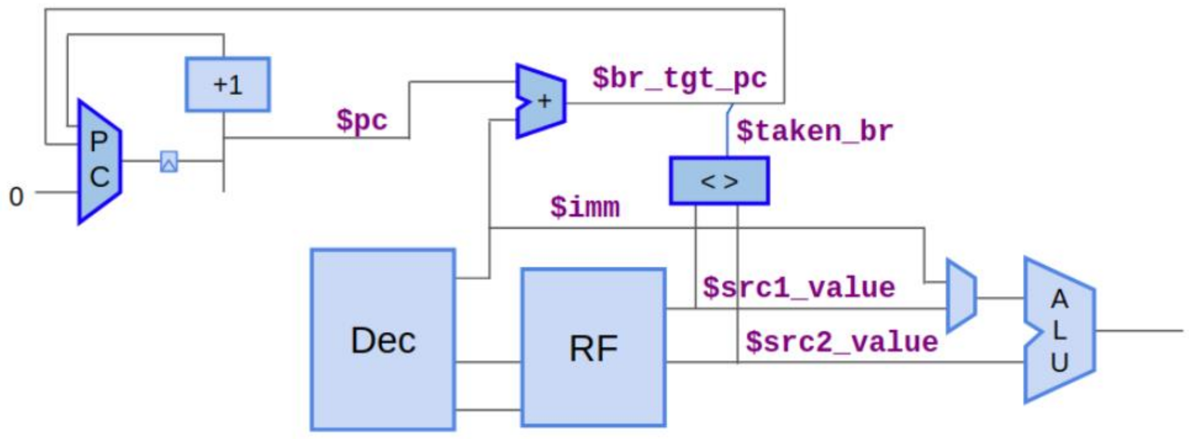


Picture 15. Implementing branch logic

The last piece of the puzzle to get your test program executing properly is to implement the branch instructions. Our test program uses BLT to repeat the loop body if the next incrementing value to accumulate is less than ten. And it uses BGE to loop indefinitely at the end of the test program. We'll go ahead and implement all the conditional branch instructions now.

A conditional branch instruction will branch to a target PC if its condition is true. Conditions are a comparison of the two source register values. Implementing conditional branch instructions will require:

– Determining whether the instruction is a branch that is taken (**$taken_br**);
– Computing the branch target (**$br_tgt_pc**);
– Updating the PC (**$pc**) accordingly.



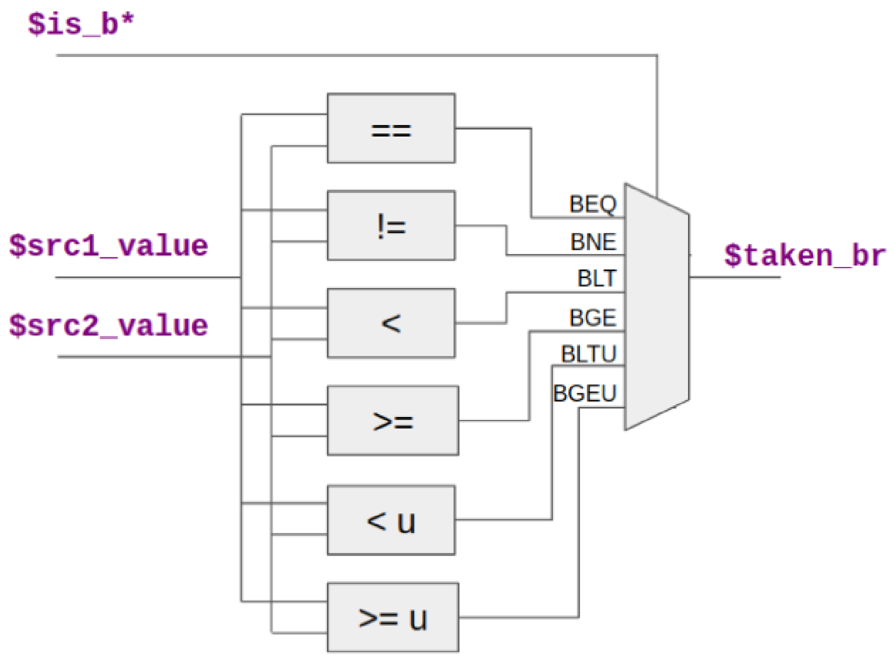
Picture 16. Branch logic in greater detail

Let's start with the branch condition (**$taken_br**). Each conditional branch instruction has a different condition expression based on the two source register values (**$src1_value** and **$src2_value**, represented as x1 and x2 below).

Instruction	Meaning	Condition Expression
BEQ	Branch if equal	`x1 == x2`
BNE	Branch if not equal	`x1 != x2`
BLT	Branch if less than	`(x1 < x2) ^ (x1[31] != x2[31])`
BGE	Branch if greater than or equal	`(x1 >= x2) ^ (x1[31] != x2[31])`
BLTU	Branch if less than, unsigned	`x1 < x2`
BGEU	Branch if greater than or equal, unsigned	`x1 >= x2`

Picture 17. Condition expressions for each conditional branch instruction

Similar to the structure of the ALU, you'll determine whether a branch is to be taken by selecting the appropriate comparison result.



Picture 18. Branch taken logic diagram

Check the box for each completed step:

– Code this as a single expression, as with the ALU. As the default case of the ternary operator, assign to zero for non-branch instructions. For each branch instruction, determine the value based on the Conditional Expression for that instruction listed in the table above.

We also need to know the target PC of the branch instruction. The target PC is given in the immediate field as a relative byte offset from the current PC. So, the target PC is the PC of the branch plus its immediate value.

– Code an expression for **$br_tgt_pc[31:0] = ...;**

– If the instruction is a taken branch, its next PC should be the branch target PC. Update the existing **$next_pc** expression to reflect this;

– Compile/simulate and debug. Once all is correct, your program will be looping. It should stop looping once it has produced the sum of values 1..9 (45). The final ADDI subtracts 44 from this and should therefore produce a value of 1 in x30. Then the final BGE should loop indefinitely to itself.

Now that our test program seems to be working, let's add some automated checking. We can tell the Makerchip platform that our test passed or failed by assigning the provided Verilog output signals passed and failed. In the **\TLV** context, Verilog signals are referenced with a preceeding "*****".

We'll give you a little check that the program's PC repeats, and that x30 contains a value of 1.

– Enable this check by replacing the line ***passed = 1'b0** with **m4+tb().**

– Feel free to find the resulting macro expansion defining ***passed** in NAV-TLV. Check LOG for "Simulation PASSED!!!" message. CONGRATULATIONS!!! Save your work outside of Makerchip.