In class: Complete the register file

Download the starter ZIP, unzip it, and open `register-file.dig` in Digital. The ZIP includes the tests. You do not need to clone a repository for this activity.

1. Plan before wiring

Our register file has 32 register numbers, x0–x31, each representing a 32-bit value. It has two independent read ports and one write port. **x0 always reads as zero.** Writes to x1–x31 occur on the rising clock edge when `RegWrite = 1`. Reads are combinational: after the circuit settles, each output reflects the selected register without waiting for a clock edge.

Fill in:

- A register number needs ☐ bits; each data value needs ☐ bits.
- Choosing one write destination uses a ☐.
- Choosing one of 32 values for a read port uses a ☐.
- Two simultaneous reads require ☐ read-selection circuits.
- For register i (i = 1–31), its write enable is `RegWrite AND (WriteReg ==` ☐ `)`.

2. Fill six circuit blanks

All components, widths, and wires are supplied. **Stop simulation before editing.** Right-click each tunnel named `BLANK_A` through `BLANK_F` and replace its **Net Name** with your answer. Matching tunnel names connect signals without a long wire. Preserve capitalization and spaces; do not rename the circuit's input/output pins.

Choose from these existing signal names (use each once):

`Clock`, `Enable 1`, `ReadReg1`, `ReadReg2`, `WriteData`, `WriteReg`

Blank	Where to look	What belongs here?
A	Below the decoder, beside the supplied write-control logic	The requested destination register number: ☐
B	Upper-left input of `Reg 1`	The new 32-bit value to store: ☐
C	Middle-left input of `Reg 1`	The signal whose rising edge triggers storage: ☐
D	Lower-left input of `Reg 1`	Permission for this particular register to update: ☐
E	Below the left read multiplexer	The register number for read port 1: ☐
F	Below the right read multiplexer	The register number for read port 2: ☐

Zoom in on each area. Registers 2–31 are already wired. The decoder outputs are named `Enable 1` through `Enable 31`.

Supplied write-control logic: the five-bit AND gate below the decoder passes `WriteReg` when `RegWrite = 1` and forces the decoder address to 0 when `RegWrite = 0`. Decoder output 0 is unused, so selecting it enables no writable register. You do not need to edit the splitter or AND gate.

The `Read 0` tunnel is driven by a 32-bit ground (constant zero), rather than a storage register. Explain why a write to x0 cannot change either read port's value when it selects x0:

3. Predict, then test

Assume x1 initially holds 7 and x2 initially holds 9. Rows occur **in order**. All numbers are decimal. Give outputs after each action and after the circuit settles.

Action	ReadReg1	ReadReg2	ReadData1	ReadData2
Set `WriteReg=1`, `WriteData=42`, `RegWrite=1`; hold clock low	1	2		
Raise the clock from 0 to 1	1	2		
Set `RegWrite=0`, `WriteData=99`; lower then raise the clock	1	2		
Set `WriteReg=0`, `WriteData=123`, `RegWrite=1`; lower then raise the clock	0	1		
Hold clock unchanged; change only the read selectors	2	1		

Save your circuit and run the test cases from the green **Register File Test** block using Digital's test command. The starter is intentionally incomplete and may report unconnected inputs until all six blanks are filled. `tests.txt` is a readable copy of the embedded tests; you do not need to import it. Keep expected outputs unchanged.

To try the table manually, first write 7 into x1 and 9 into x2 with `RegWrite=1`, pulsing the clock low then high for each write. Return the clock to low before the first row.

4. Discuss

1. If both read selectors equal 1, can both outputs read x1 at once?

2. Why is `RegWrite` alone insufficient as the enable input for every register?

3. In a pipelined CPU, can one instruction read registers while another writes a register? What timing question arises if they use the same register?

Keep your completed `register-file.dig`: you will reuse it as a custom component in your RISC-V CPU. Follow any collection instructions given in class.

Reuse in the RISC-V CPU (reference for later)

This is the full RV32 register file, with 32 register numbers and 32-bit data. Save the completed file as **`register-file.dig`**, keep it alongside `cpu.dig`, and insert it with **Components → Custom**. Preserve these port names and widths:

Register-file port	Width	CPU connection
`ReadReg1` (input)	5	`rs1`: instruction bits 19–15
`ReadReg2` (input)	5	`rs2`: instruction bits 24–20
`WriteReg` (input)	5	`rd`: instruction bits 11–7
`WriteData` (input)	32	Writeback result; ALU result for the arithmetic-only CPU
`RegWrite` (input)	1	Register-write control signal
`Clock` (input)	1	CPU clock (`clk` in the Spring CPU starter)
`ReadData1` (output)	32	ALU operand A
`ReadData2` (output)	32	Register input of the ALU operand-B mux

`Clock` is an ordinary input pin so the CPU supplies the clock. For standalone manual tests, click it low then high. Do not add an independent clock inside this component. For immediate instructions, the CPU's operand-B mux selects the immediate instead of `ReadData2`. If loads are added later, the CPU's writeback mux selects memory data for `WriteData`.

For a pipelined CPU, drive the write ports with the **writeback instruction's** destination, result, and write-enable; drive the read selectors with the instruction in the register-read stage. Do not use the decode-stage `rd` to write back an older instruction. Forwarding and stalls are separate CPU logic. This register file reads combinationally and writes on the rising edge; it does not include a bypass path or reset input.