"The cheapest, fastest and most reliable components of a computer system are those that aren't there." – Gordon Bell

Learning Objectives

1. Understanding memory-mapped I/O in hardware
2. Understanding interrupt handling in hardware
3. Understanding the functioning of coprocessor 0

Work that needs to be handed in on Github

Note: *Please Make sure to check that your code compiles and runs on EWS before submitting it! Lots of points are lost each week on simple mistakes.*

First deadline

1. `timer.v`: The Verilog implementation of the timer circuit.

Second deadline

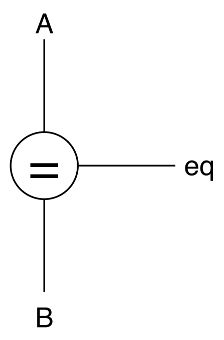
1. `cp0.v`: The Verilog implementation of coprocessor 0.
2. `machine.v`: The single-cycle machine combined with the timer and coprocessor 0 circuits.

Guidelines

- **Please read the entirety of the handout carefully and make sure you understand the circuits before implementing them.** It'll make your life a lot easier and also help you debug and test your circuits.
- **Look over the provided Verilog files.** You don't need to understand the syntax used (since it's deliberately goofy), but you should be aware of what modules we provide, since you'll need to use some of them in your implementation. In particular, we've provided flip flops and registers with *synchronous* reset inputs, which will be used in our implementation.
- **This lab is more limited in scope than Lab 5.** In previous labs, we've built up the full, single-cycle datapath. To decrease complexity for this lab (and future labs), we've limited the functionality of the CPU. Please refer to Figure 3 and 4 on the last page of the handout for the changes. Please take some time to understand the differences between Lab 5's diagram and this one (Lab 8's).

Equality circuit

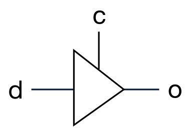
For our timer circuit, we'll need to compare addresses against known values. We'll use the following symbol to indicate this, where A and B can be any number of bits wide and eq is 1 if A and B are equal. In Verilog, you could simply write `A == B`.


A
=
eq
B

Buses and tri-state buffers

As discussed in lecture, all hardware devices use the memory bus to communicate, which means all devices need to output to the data bus. So far, we've been using muxes to select between multiple sources for the same output line, but a more scalable (and realistic) solution is the *tri-state buffer*.

A tri-state buffer has a data input d, a control input c, and an output o. If c is 1, the input is passed to the output, *i.e.*, $o = d$. If c is 0, however, the output is *high impedance*, which is essentially equivalent to it being disconnected. You can think of a tri-state buffer as a switch, where $c = 0$ means the switch is off and $c = 1$ means the switch is on. In Verilog, a high impedance value is represented as `z`, or as a yellow line in gtkwave.



c	d	o
0	X	z
1	0	0
1	1	1

Figure 1. A tri-state buffer and its truth table

A high impedance wire can be connected to other wires without affecting them. Tri-state buffers therefore allow multiple outputs to be connected to the same bus directly. Only one tri-state should be enabled at a time, and the output of the enabled tri-state becomes the output of the entire bus. For example, here's how to implement a 2-to-1 mux using tri-state buffers (and larger muxes are in fact usually made from tri-state buffers):

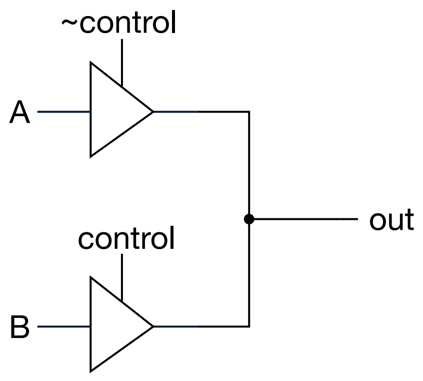


Figure 2. A 2-to-1 mux built from tri-state buffers

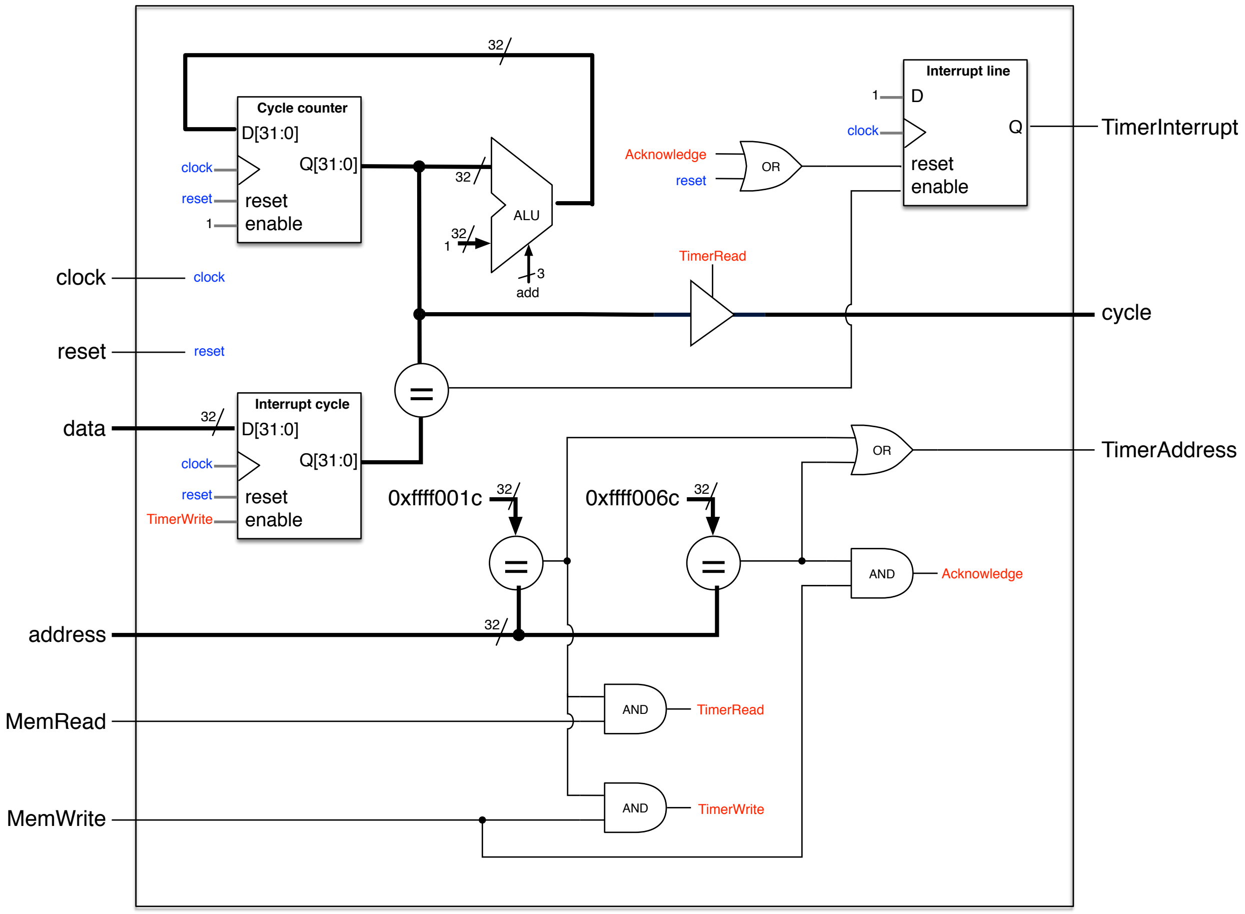
We've provided a parametrized tri-state buffer in `modules.v`. We've also modified our data memory such that when its read input is 0, its data output is high impedance (using a tri-state buffer). You'll use this to connect the data memory and the timer device to the same data bus.

The timer circuit [30 points]

The *timer* is a hardware device which allows the processor to read the number of clock cycles that have elapsed, as well as request an interrupt at a certain cycle.

As with any hardware device, the processor communicates with the timer using memory-mapped I/O. It can load from address `0xffff001c` to get the current number of cycles, and it can store a value to this address to request an interrupt at that cycle. It can also store any value to address `0xffff006c` to acknowledge the timer interrupt.

Here's what our timer circuit will look like:



It takes as input the memory address and data, and whether memory is being read or written, as well as the clock and reset signals. The `cycle` output is the current cycle if the timer was being read, otherwise high impedance. The `TimerAddress` output is 1 when the address is a timer memory-mapped I/O address (so that data memory reads and writes can be disabled). The `TimerInterrupt` output is 1 when a timer interrupt is being raised.

The cycle counter register keeps track of the number of cycles. The interrupt cycle register stores which cycle to interrupt the processor on, and when the current cycle matches the interrupt cycle, a timer interrupt is raised by enabling the interrupt line flip flop. The interrupt line stays asserted until the interrupt is acknowledged or the machine is reset.

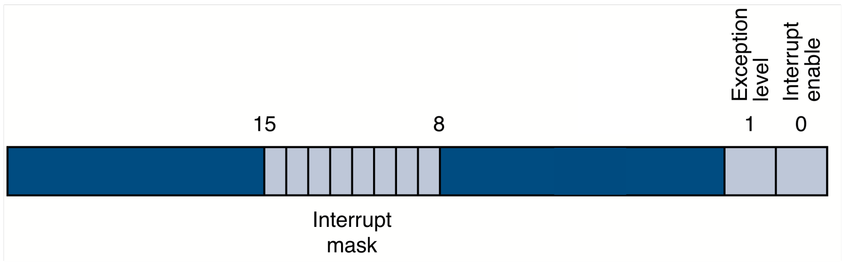
Make sure you understand how this circuit works, and then write the Verilog for it in `timer.v`. We've provided a basic test bench in `timer_tb.v`, which you should augment with your own test cases. You can use `make timer` to compile and run the test bench.

Coprocessor 0 [35 points]

Coprocessor 0 handles interrupts and exceptions. In this lab, we only have to handle the timer interrupt, and we don't deal with exceptions. Our coprocessor will therefore be quite simple, but it'll still capture all the important ideas underlying a real coprocessor 0 implementation.

Status "register"

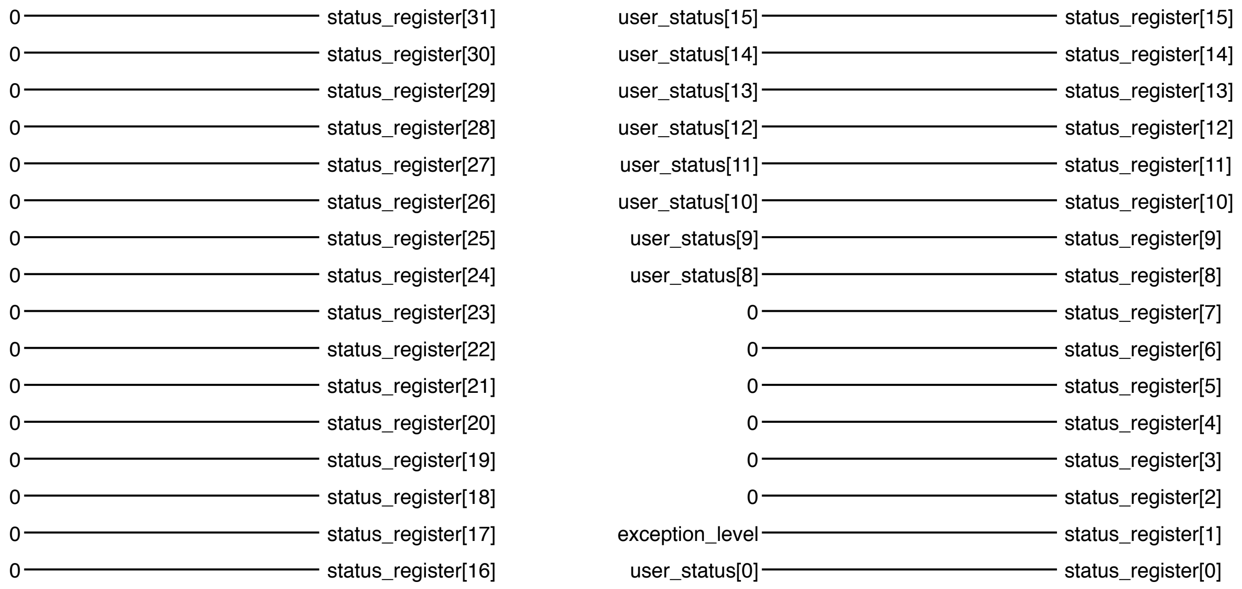
The status "register" (coprocessor 0 register 12) controls which interrupts are enabled and whether any are enabled at all. We're concerned with the following bits of the status register:



Each bit in the interrupt mask corresponds to a disabling (if it's 0) or enabling (if it's 1) a particular type of interrupt; for us, bit 15 corresponds to the timer interrupt. The interrupt enable bit controls whether any interrupts are taken. These bits can be set by user code using the `mtc0` instruction.

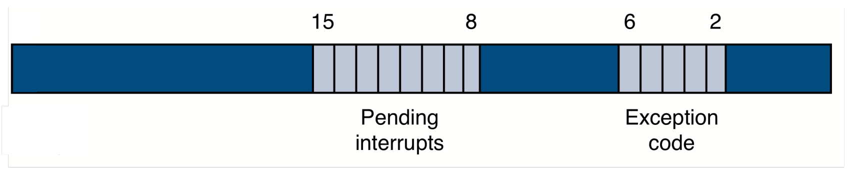
The exception level bit is 1 whenever an interrupt or exception is currently being handled. This along with all the other status register bits (which we'll just force to 0 in our implementation) **cannot** be set by user code; an `mtc0` to the status register should not modify these bits.

To enforce this, we'll actually have one register to hold the bits which can be set by the user, and another flip flop to hold the exception level bit. We'll combine the two into a single **bus**:

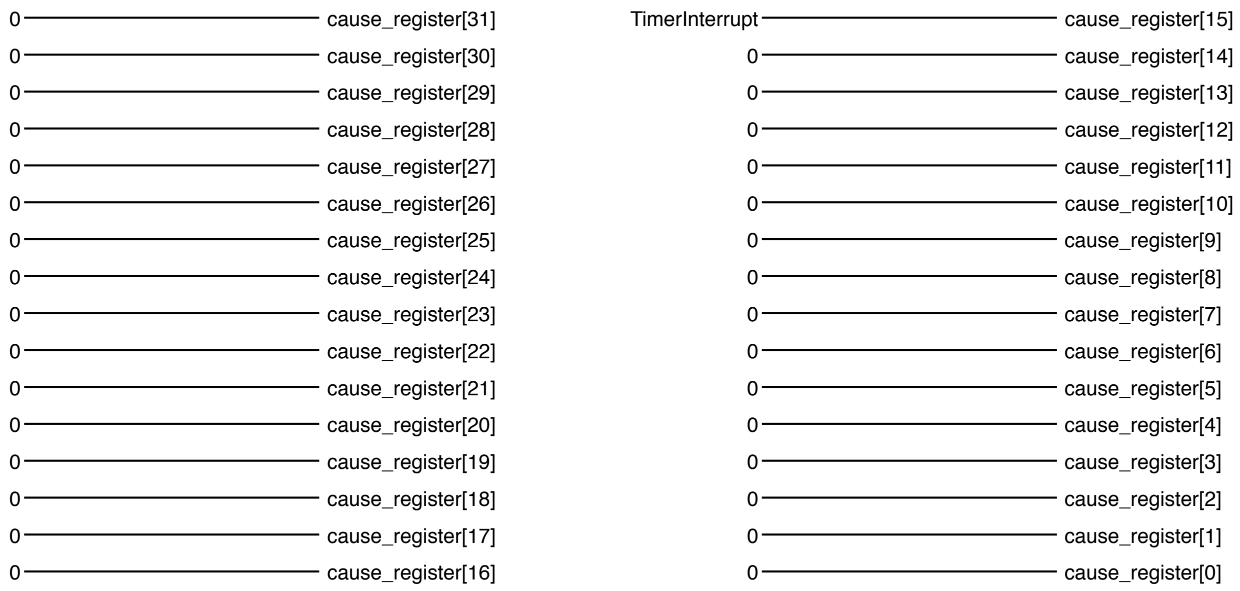


Cause "register"

The cause register (coprocessor 0 register 13) tells us which interrupt or exception occurred:



Like the status "register", the cause "register" isn't implemented as an actual register. Instead, each of the pending interrupt bits is connected to the corresponding interrupt line; for us, bit 15 will be connected to the timer interrupt line. The exception code would be encoded from the incoming exception lines, but since we don't have any exceptions, we'll just hardcode the exception code bits to 0, forming the following **bus**:



EPC register

The exception program counter register (coprocessor 0 register 14) stores the address the interrupt handler should return to. Whenever an interrupt is taken, this register is updated with what the next PC would have been normally, so that execution can resume there after the interrupt is serviced and the handler issues an `eret` instruction.

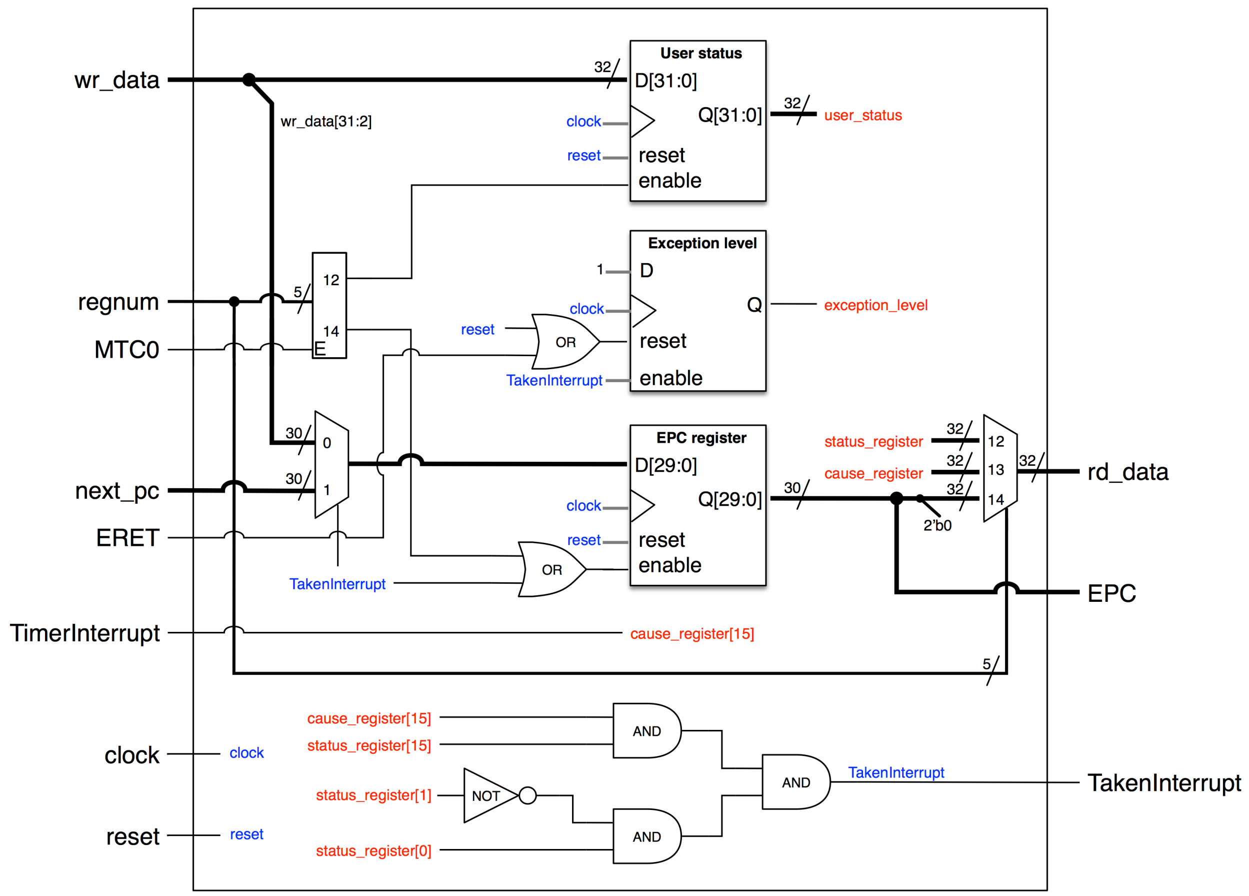
Reading and writing coprocessor 0 registers

The `mtc0` instruction is used to write coprocessor 0 registers. As usual, we'll use a decoder to enable the register being written. Since the actual coprocessor has 32 registers, we'll use a 5-to-32 decoder, although only bits 12 and 14 of the decoder output will get used (since we can only write to the status and EPC registers).

The `mfc0` instruction is used to read coprocessor 0 registers. We'll use a 32-to-1 mux to select which register value to output. Inputs 12, 13 and 14 will be connected to the status, cause and EPC registers, respectively; the rest will be connected to 0s (since we don't implement any other registers).

Complete coprocessor 0 circuit

We can put all our components together to get a complete coprocessor 0 circuit:



It takes as input what coprocessor register number is being read or written, what data is being written to it, whther the current instruction is an `mtc0` or an `eret`, what the regular next PC would have been, whether a timer interrupt is being raised (from our timer circuit), and the clock and reset signals. It outputs whether an interrupt should be taken, the value of the coprocesor register being read, and the value of the EPC register.

The EPC register can be written both by user code (using the `mtc0` instruction) and by the coprocessor itself (when an interrupt is taken), hence enabling it under both circumstances and using a mux to select which value to write.

We take an interrupt when the timer interrupt is enabled and we have a pending timer interrupt, the interrupt enable bit is set, and we're not already handling an interrupt. In an actual coprocessor, we would have to check for all other interrupts being enabled and pending too.

The `eret` instruction is used to return from an exception. The exception level bit should be set to 0 upon the execution of this instruction, hence resetting the exception level flip flop when an `eret` instruction is seen.

Make sure you understand how this circuit works, and then write the Verilog for it in `cp0.v`. We've provided a basic test bench in `cp0_tb.v`, which you should augment with your own test cases. You can use `make cp0` to compile and run the test bench.

Putting it all together [35 points]

Now that we have our timer and coprocessor 0 circuits, we can combine them with the given single-cycle datapath (figure included below for reference) to support I/O and interrupts in our machine. The required additions are described in textual and via a circuit diagram.

Our provided decoder (in `modules.v`) already supports the `mfc0`, `mtc0` and `eret` instructions, and outputs control signals for each of them.

You'll connect the timer circuit's address and data inputs to the same sources as the data memory's address and write data inputs, respectively, the MemRead and MemWrite inputs to the corresponding control signals, and the clock and reset inputs to the corresponding signals. The TimerAddress output being 1 should force the data memory's read and write inputs to be 0 (through ANDing with the NotIO signal), and the cycle output should be connected directly to the same bus as the data memory's read data output (because of the internal tri-state buffers in both).

Coprocessor 0's TimerInterrupt input should be connected to the TimerInterrupt output from the timer circuit. The MTC0 and ERET inputs should be connected to the corresponding decoder outputs, and clock and reset connected to the corresponding signals. The regnum input should be connected to `rd` and the c0's wr_data intput to `R[rt]`, since those are the registers used by the `mfc0` and `mtc0` registers. The processor's next_pc input should be connected to the corresponding bus (to be precise, what that bus represents in the unmodified datapath). You'll need to write the c0's rd_data output to the register file on an `mfc0` instruction; the provided decoder already takes care of the control signals, so you just need to adjust the register file's write data input accordingly.

Finally, you'll need to adjust control flow to account for interrupts and interrupt returns. When the TakenInterrupt output from coprocessor 0 is 1, the PC should be set to `0x80000180` (the MIPS interrupt handler address), and on an `eret` instruction, the PC should be set to the EPC output from coprocessor 0. **Be careful; this machine uses a 30-bit PC, which means you *cannot* just use `0x80000180` without modification.** Hint: How did we deal with a 30-bit PC in the datpath exam?

Make all these changes to the single cycle datapath in `machine.v`, and then compile and run your circuit using `make machine`. You can modify `test.s` to change the assembly code being executed.

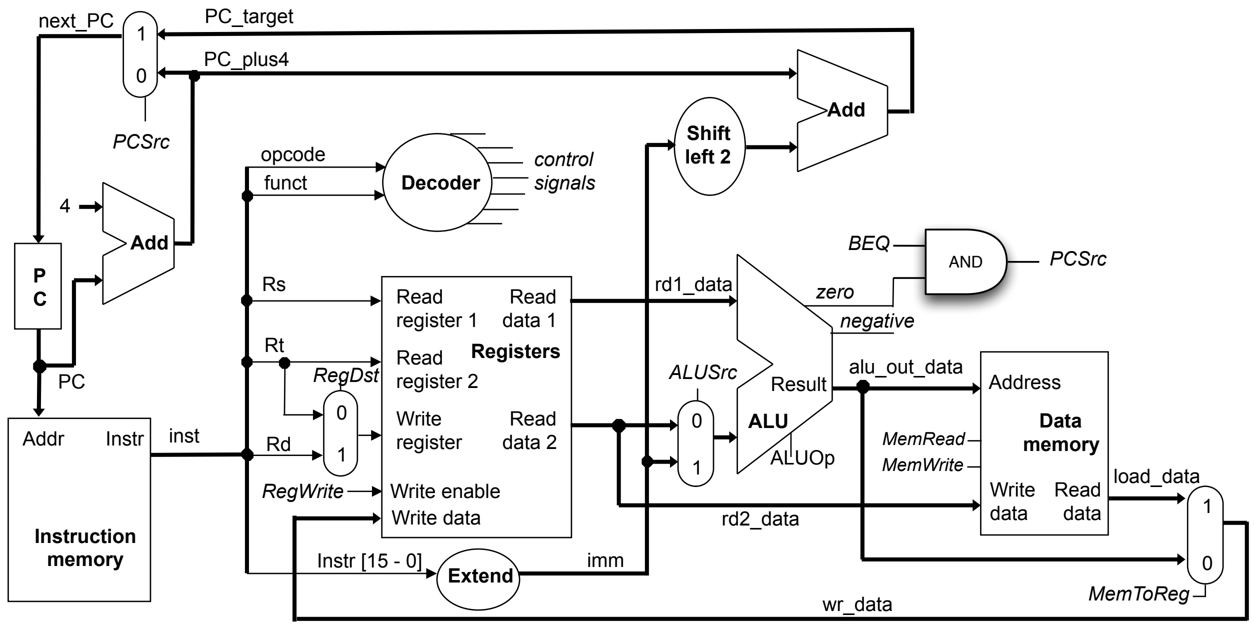


Figure 3. Single-cycle datapath as given.

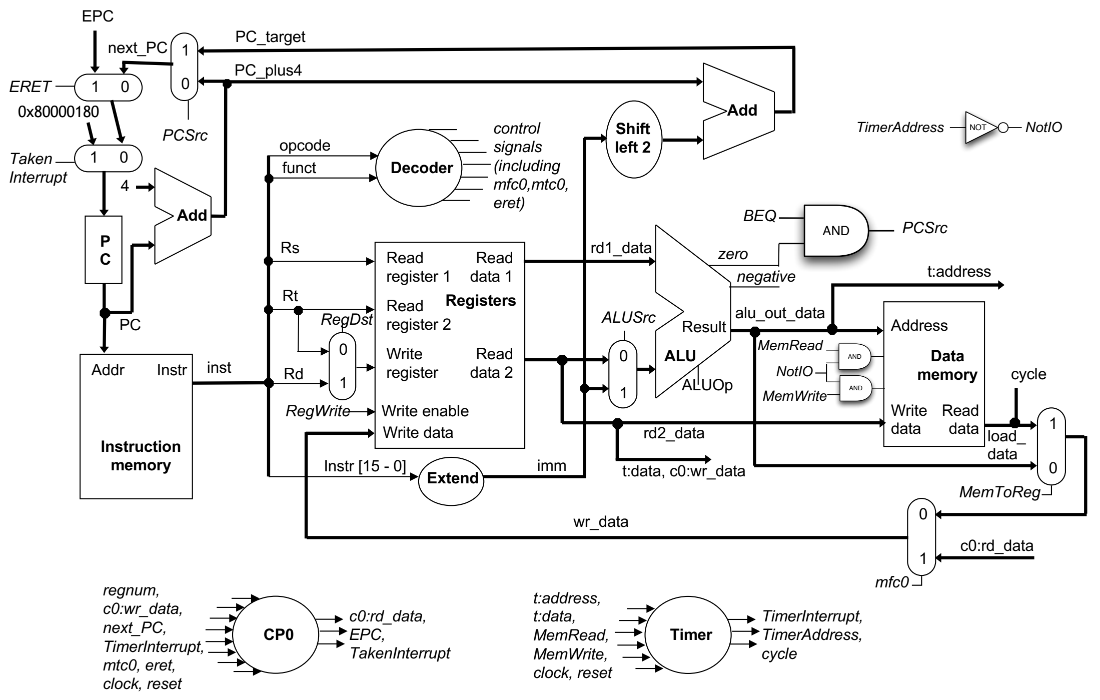


Figure 4. Single-cycle datapath extended so that it can be connected to the timer and co-processor 0 modules.