```
34 _cclkMask long |<CLK
35 _cmosiMask long |<MOSI
```

Line 13 Set the loop index and set `Z` to zero.

Lines 14–16 Set `_cdt` to the current counter; add 1/8 ms (125us); and wait until then. `waitcnt` takes two arguments: the counter will wait until it is equal to the first argument. It will then add the second argument to the first so that you can call `waitcnt` again.

Line 19 Lower CS. `xor outa, _ccsMask` will toggle the CS.

Line 23–24 Get the bits, Most Significant Bit first and place it on the MOSI line. The `rol _clogVal, #1 wc` will rotate left and put the highest bit in `C`; `muxc outa, _cmosiMask` will set MOSI to `C`.

Lines 26–29 Raise the clock line, wait 1/8th of a ms, lower the clock, wait again, and loop.

Line 33 Lower the CS.

10.5.3. Logging deadlock

> **Warning!**
>
> If you call `WRITE_SPI`, you must have another cog which raises the REQ line otherwise you will block forever.

At any time, you can call `WRITE_SPI` in the PASM code. It will block until the Spin code raises the `REQ` line. At that point the PASM code will transmit the log value and continue.

10.6. Locks

> On single-track lines, the railroads needed a foolproof way to prevent two trains from traveling on the same section of track at the same time. They settled on a simple but elegant solution: the semaphore. A brass cylinder or token is cast and engraved with the name of the stations at each end of the section of single-track. A train could enter that section *if and only if* the engineer had physical possession of the token. When he arrived at the station at the far end, he would give the token to the station master, who could pass it on to a train traveling in the other direction.
>
> You still see a version of this when road crews are working on potholes and signals at each end control the traffic.

Figure 10.4.: Waiting for the signal, Santa Fe RR Train, Melrose, NM. Photo by Jack Delano, 1943. From the Library of Congress Farm Security Administration archives. http://www.loc.gov/pictures/item/fsa1992000785/PP/

10.6.1. Introduction to locks

The propeller has semaphores for exactly the same reason: to control access to *critical shared resources*. In order to prevent a collision (for example one cog is modifying an array at the same time that another is reading from it), the propeller has eight semaphores.

`locknew/lockret` Create or destroy a semaphore.

`lockset` Set the state of the semaphore to 1 and return the previous state.

`lockclr` Set the state of the semaphore to 0 and return its previous state.

Here is a sketch of the process, with time proceeding to the right ("time" is a bit of misnomer here—because locking is a HUB operation, even though both cogs may request the lock at the same time, the HUB will service those requests in order). At time 1, cog 0 acquires the lock. Because `lockset` is a HUB operation, only one cog a time can request the lock; though it looks like both cogs are competing for the lock, in fact, only of them can obtain it. In this case, cog 0 received the lock, and when cog 1 requested it (one clock cycle later), it was informed that somebody else had the lock (because `lockset` returns 1). Remember, the propeller operates in round robin fashion for HUB operations, providing exclusive access to the HUB for one cog, then the next, and so on.

At time 2, cog 1 again requests the semaphore, and the propeller again returns the *previous* state of the lock, which is 1, which indicates that somebody else has it. This continues until time 5, when cog 0 releases the semaphore, and it is set to 0. At time 6, cog 1 again requests the lock, and this time the *previous* value is 0, so cog 1 knows that it has the semaphore[3]. It holds it until time 9.

cog0	set				clr				
0	1	1	1	1	0	1	1	1	0
cog1	set	set	set	set	set	set			clr
0	1	2	3	4	5	6	7	8	9

The semaphore is created and then a `lockset/lockclr` pair of instructions brackets the critical section of code. So for example, in our code, cog 0 could acquire samples between times 1 and 5, but we must prevent cog 1 from attempting to compress them during that time. Between times 6 and 9 the compression can safely proceed.

> The way we have written the code there is in fact no chance of a collision between sample acquisition and compression. The main cog *blocks* while the compression cog is working, so it is impossible for it to modify `sampsBuf`. However, that isn't very good design: it is wasteful for `main` to sit there twiddling her thumbs while `steim` is off doing his job. She could be (and usually is) performing other tasks. It is under that type of system design that the semaphore is most useful. After all, if one has

[3] I lie. In reality, cog 0 would have released the cog at time 5, and during the next clock cycle, cog 1 would have requested and received the lock, but for illustration purposes, I fibbed.