4.2 Focused pseudocoding of some FIFOs

The first examples will be of *focused pseudocoding*. This is very close to real occam programming, and is usually used to design low-level hardware or something very close to it. The idea is to imply a fully detailed state machine as a specification. Later implementations may not greatly resemble the pseudocode, since they can use combinatorial as well as sequential hardware design, but they are required to perform exactly as the pseudocode implies.

The example here is a common component of basic hardware: a First-In First-Out (FIFO) buffer. This is the same as a queue. In it, *up to a certain capacity,* packets can be stored and later output in the order of their reception. Contrast it with a stack, or Last-In First-Out (LIFO) buffer.

The size of a packet is usually fixed, but can be any value. Even one-bit packets can be queued in a “shift register.” A shift register is usually synchronous, meaning every member of the queue moves one slot upon a clock signal. That, however, is too inflexible for most queue uses with packets of one byte or bigger. Usually an asynchronous design is desired, where the packet input and the packet output are under independent external timing controls.

4.2.1 Simple FIFO

The simplest asynchronous FIFO design is rawFIFO, a PAR of n identical members, specified in Table 4.1.

When considering the design of rawFIFO, remember the definition of a PAR in Chapter 3. The execution of members of the PAR can be ordered **in any way** that is consistent with points of synchronization (that is here communication on a channel). This includes interleaving, sequence, or true parallel.

In focused hardware pseudocoding, internal channels as well as programming and internal (local) variables are a convenience serving to define the device’s behavior from the point of view of the outside world. Also usually convenient is a presumption that internal activity is “fast” compared with interactions with the outside world. These two considerations lead to the definition of a STABLE STATE as *a state that cannot change as long as the part of it that is shared with the outside world does not change.*

Now rawFIFO (with n=5) can be visualized as Figure 4.1, with the external channels `in` and `out` connecting to `buffer.d[4]` and `buffer.d[0]` respectively. (Here remember period in occam, like underscore in C, is part of names; and I am using `buffer.d[k]` to refer to that instance of `PROC buffer` that is called

```
-- This is a raw FIFO implemented in occam-like Crawl Space
-- pseudocode. It is the raw or theoretical FIFO of n parallel
-- buffers, each storing one PACKET and passing it on as soon as
-- possible. An actual implementation of a FIFO using this approach
-- is usually not practical, since it passes each packet a large
-- number of times internally, and a circular indexing system as in
-- "FIFO.occ" is preferred. However, this is the defining standard
-- against which any FIFO implementation (software or hardware) is
-- judged.
-- Global resources are:
-- CHAN OF PACKET in : for the input to the FIFO
-- CHAN OF PACKET out : for the output from the FIFO
-- The size of the FIFO is defined below. (The structure assumes
-- n >= 2.)
VAL INT n IS 32 :
VAL INT nm1 IS n - 1 :
VAL INT nm2 IS n - 2 :
-- this buffer code is repeated n times in parallel
PROC buffer(CHAN OF PACKET enterch, exitch, PACKET pkt)
  WHILE TRUE
    SEQ
      enterch ? pkt
      exitch ! pkt
:
-- this is the main program and its local resources
[n]PACKET pktall : -- for clarity; these could be local to the buffers
[nm1]CHAN OF PACKET ch : -- ch[n-1]=in and ch[-1]=out, conceptually
PAR
  buffer(ch[0], out, pktall[0])
  PAR i FROM 1 FOR nm2
    buffer(ch[i], ch[i-1], pktall[i])
  buffer(in, ch[nm2], pktall[nm1])
--END PAR
}}}
```

Table 4.1: Toplevel view of rawFIFO.occ in origami (irrelevant comments removed)

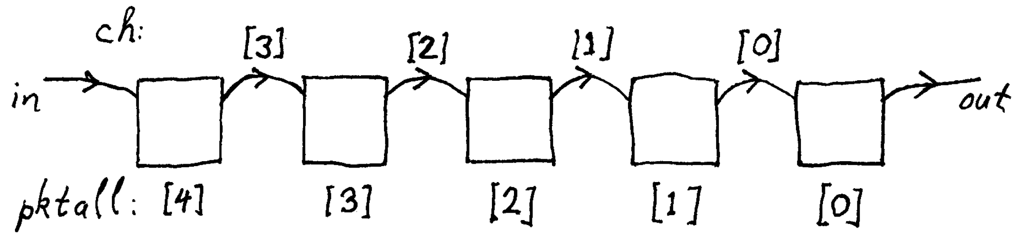

Figure 4.1: rawFIFO (n = 5)

with `pktall[k]`.) All the links operate only in one direction, which means that information is transmitted "backwards" (toward `in`) only by blocking.

Since this is a FIFO that is being designed, internal FIFOs in the channels would be superfluous. An output transmission (!) by `buffer.d[k]` is therefore simultaneous with an input reception (?) by `buffer.d[k-1]` to its right. Branching, looping and communicating are clearly not stable states, since they proceed to completion. Therefore the only stable state candidates for each member of the PAR are just before the input and just before the output, which can be points of blocking. They will be denoted ? and ! respectively, resulting in an n-letter word made of these characters.

It is immediately obvious that !? anywhere in this word is not stable, since a communication can and will begin immediately. Therefore all the ! must be to the right of all the ?. It follows that the only stable states are denoted by the count of ! (waits on output):

```
????? 0
????! 1
???!! 2
??!!! 3
?!!!! 4
!!!!! 5
```

Closer examination shows that each member of the PAR begins at an input, so that stable state 0 is the initial state. An internal packet communication means a !? changes to a ?! in the word, but a packet coming in on `in` causes the leftmost ? to change to !, and a packet leaving on `out` causes the rightmost ! to change to ?. Therefore the first state change from stable state 0 has to be a packet input on `in`.

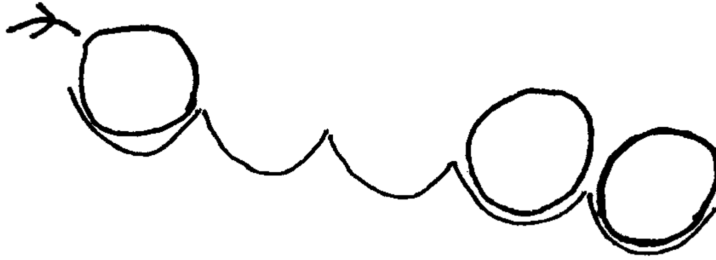

Figure 4.2: Unstable after `in`

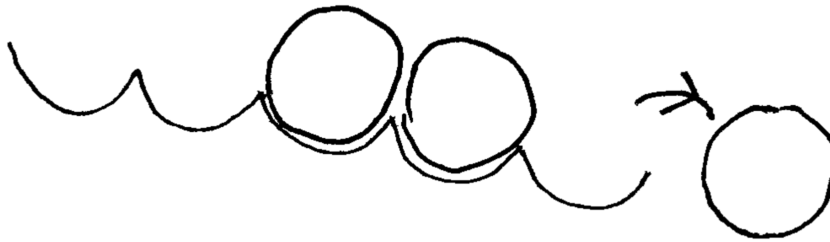

Figure 4.3: Unstable after `out`

The key insight is that information flows to the right. Therefore even though after !? changes to ?! the packet still remains in the left member (it's a *copy* not a *move*), it can be treated as a move, and the left copy of the packet can be ignored. It will be overwritten as soon as new information enters from `in`. A member's buffer is **full** if it has received data that it has not re-transmitted, and **empty** if either it has never received data, or if the data it currently contains has been re-transmitted. This means that (whether stable or unstable) ! always means full, and ? always means empty.

Figure 4.2 is an unstable state with two packets ready to output, and a new packet just input. It corresponds to `!??!!`. Figure 4.3 is an unstable state that came from a stable state of three packets queued, but the rightmost packet was just output. It corresponds to `??!!?`. It is easy to see how these will "fix" themselves (if they are given time) and reach the stable states `??!!!` and `???!!` respectively.

Inspection shows that, if the code is strictly followed, Figure 4.4 cannot happen. The transmission from [2] to [1] cannot be at the same time as that from [1] to [0], because of the sequence in [1]. If `PROC buffer` were made into a

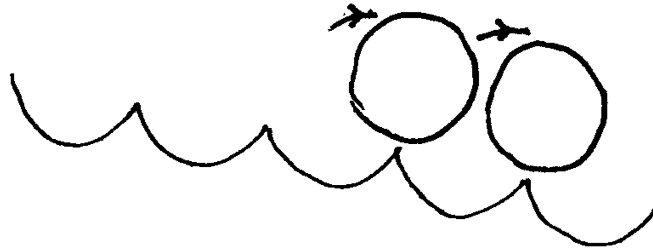

Figure 4.4: Simultaneous move ?

loop on PAR, that would not solve this problem, since the shared buffer would be illegal. And yet it seems that the simultaneous communications would be the most time-efficient way to do the move.

Not quite. The communication into [1] must lag a little behind that out of [1], otherwise the value transmitted out will be uncertain. Careful hardware can make this lag much less than a packet, thus improving on the efficiency of the strict occam `rawFIFO`. Given that the pseudocode is intended to define the function of the FIFO (its behavior to the outside world), this is perfectly OK.

4.2.2 Circular indexed FIFO

In any case, there is an inefficiency with `rawFIFO`, in that every packet must be transmitted n-1 times internally. One can imagine the FIFO as a circular buffer, with rotating input and output access, where input can fire whenever the buffer is not full and output can fire whenever it is not empty. "Rotating input and output access" can be modeled by two indices. But the necessary race presents a difficulty. It would be most simply represented by an ALT on both input and output. And, like the occam language, I have settled on input-only ALTs.

Why? The reason is that for a pure CSP communication, both sides have to act independently, not under control of an all-knowing supervisor. That is critical to the goal of simplifying design by dividing it into multiple independently maintained entities. Therefore, to avoid an infinite regress, that means one side of each communication has to commit unconditionally to communicating. In the occam tradition, it is the output side.

This is actually expressed in Transputer design. At one point I said that Transputer occam communication is *almost* unbuffered. This is because when two Transputers are connected by a wired link, the outputting side always

```
-- This is a FIFO implemented in occam-like Crawl Space pseudocode.
-- On a fast system it is equivalent to n parallel buffers, each
-- storing one PACKET and passing it on as soon as possible.
-- Global resources are:
-- CHAN OF PACKET in : for the input to the FIFO
-- CHAN OF PACKET out : for the output from the FIFO
-- The size of the FIFO is defined below.
VAL INT n IS 32 :
...  PROC shelf(CHAN OF PACKET out, fromstore, CHAN OF BOOL need, PACKET pkt)
...  PROC store(CHAN OF PACKET in, toshelf, CHAN OF BOOL need, []PACKET pkt)
...  main program with local resources
}}}
```

Table 4.2: Toplevel view of FIFO.occ in origami (irrelevant comments removed)

starts the communication by sending one byte across the link. That byte is committed, though if the receiver never sends an ACK, then the communication is deemed not to be complete. There can be an ALT on the receiving side, and the link may not be the winner. Every point-to-point communication between independent hardware entities must deal with this problem in a similar way.

It would be possible to reverse this, and have the receiver send a REQ before the transmitter sends a byte, but you cannot have it both ways on the same communication. It would even be possible to have a FIFO that did ACK-type communication on its input end and REQ-type communication on its output end, and then the input and output ALT FIFO would be possible. But this has strange and nonstandard implications, and we can save the standard communication order with a special member of the PAR. Table 4.2 shows the solution.

In `FIFO`, the storage is divided into a one-packet `shelf` and an n-1-packet `store`. Here, `shelf` corresponds to `pktall[0]` in `rawFIFO`, the buffer just before the external output channel. The circular buffer `store` corresponds to all the other storage in `rawFIFO`. And a new, upstream internal channel is added, through which `shelf` can send an event (a timing-only communication, here represented by a `BOOL`) to `store`. The analogy is to a retail shelf and a backroom store.

Table 4.3 shows the programming of `shelf`. It is like `buffer` in `rawFIFO` except that, before inputting a new packet, it sends its ready signal to the upstream member. This allows the upstream member to handle the circular buffer with an input-only ALT.

Table 4.4 shows the coding for a circular buffer. Indexing takes the place of the internal communications of `rawFIFO`. Here a feature of the ALT implemen-

```
{{{  PROC shelf(CHAN OF PACKET out, fromstore, CHAN OF BOOL need, PACKET pkt)
PROC shelf(CHAN OF PACKET out, fromstore, CHAN OF BOOL need, PACKET pkt)
  WHILE TRUE
    SEQ
      -- When waiting on following instruction, shelf is EMPTY.
      need ! TRUE -- Because of the nature of the code of PROC store,
      fromstore ? pkt -- this input communication NEVER blocks.
      -- When waiting on following instruction, shelf is FULL.
      out ! pkt
:
}}}
```

Table 4.3: First fold view in origami

tation in [2] is of great importance. If `shelf` reaches the communication on channel `need` first, it must clearly block waiting for `store` to execute an ALT in which `need` wins. But if `store` gets there first, then `shelf` still has to block, because control has to go back to `store` to disable (adjudicate) the ALT and declare `need` the winner. Therefore, in every case, the first thing to happen after the event is that `store` will run its code to the point of outputting on `toshelf`. It must there block, because `shelf` hasn't had a chance to run yet, and `shelf` will then run and do the input on `fromstore` **without having to wait.** And even though `store` has to block momentarily while `shelf` is doing this, `shelf` will quickly run to block at output either on `out` or on `need`, and `store` will run back to the ALT.

Therefore the only possibilities for stable states in `shelf` are at the outputs, not the input. And the only possibility for a stable state in `store` is at the ALT. The state where `shelf` is at the output on `need` and `store` has `fl > 0` is not stable because the ALT will immediately fire and a packet will be transmitted to `shelf`. Therefore `shelf` can be waiting at either `need` or `out` if `fl = 0` but must be waiting at `out` if `fl > 0`. These correspond to queue length 0, 1, and `fl`+1, respectively.

Both `shelf` and `store` above are PROCs, exactly like void functions in C, and they must be called, their formals filled in, in a calling program (like `main`). Table 4.5 shows this. Each packet is communicated three times: input on `in`, internally on `passer`, and output on `out`. The upstream communication on `need` is done with a minimum of data transmission, and since the value of its `BOOL` is never checked, can be implemented by a timing-only event.

```
{{{  PROC store(CHAN OF PACKET in, toshelf, CHAN OF BOOL need, []PACKET pkt)
PROC store(CHAN OF PACKET in, toshelf, CHAN OF BOOL need, []PACKET pkt)
  VAL INT nm1 IS n - 1:
  VAL INT nm2 IS n - 2:
  INT fl, indin, indout :
  BOOL ready :
  SEQ
    indin := 0 -- input index
    indout := 0 -- output index
    fl := 0 -- number of packets (starts empty)
    WHILE TRUE
      PRI ALT
        -- If there is stuff in store, stocking the shelf takes priority
        (fl > 0) & need ? ready
          SEQ
            toshelf ! pkt[indout]
            IF
              indout < nm2
                indout := indout + 1
              TRUE
                indout := 0
            fl := fl - 1
        -- If the store is not full, new packets are input.
        -- Starvation of this line is not possible because output will
        -- soon lead to fl = 0, which deactivates the top alternative.
        (fl < nm1) & in ? pkt[indin]
          SEQ
            IF
              indin < nm2
                indin := indin + 1
              TRUE
                indin := 0
            fl := fl + 1
:
}}}
```

Table 4.4: Second fold view in origami

```
{{{  main program with local resources
[n]PACKET pktall :
CHAN OF PACKET passer :
CHAN OF BOOL need :
PAR
  store(in, passer, need, [pktall FROM 1 FOR n-1])
  shelf(out, passer, need, pktall[0])
--END PAR
}}}
```

Table 4.5: Third fold view in origami

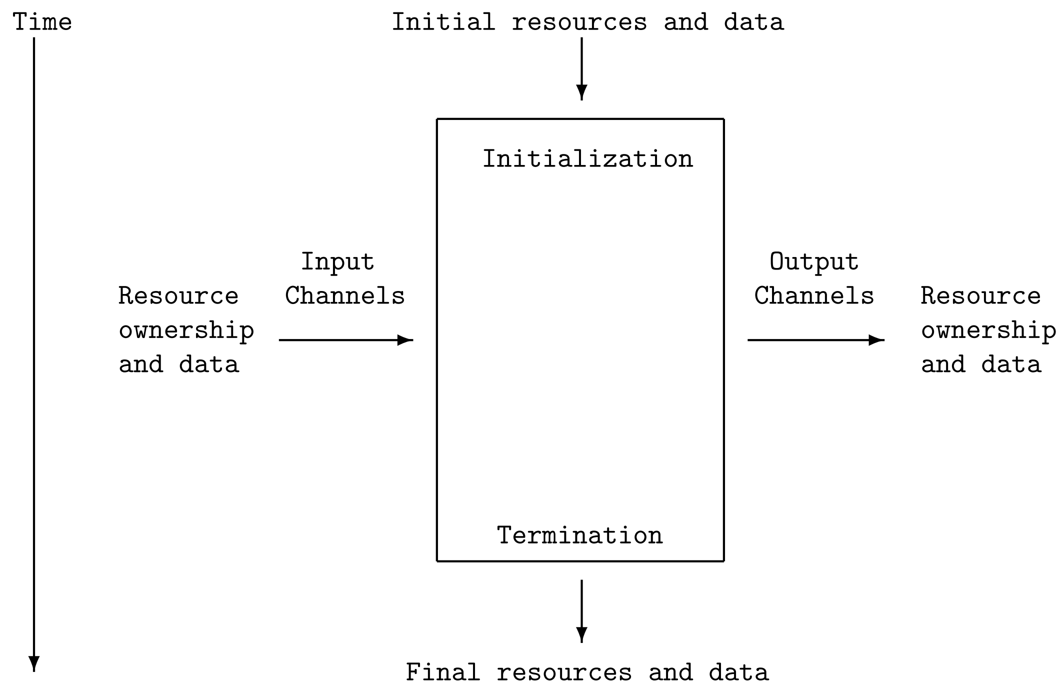

Figure 4.5: Box data and resource flow over time (strictly enforced)

4.2.3 Startup, shutdown, and nesting

Both programs described above are somewhat fragmentary, because they omit details of startup and shutdown. See Figure 4.5, which was taken from [1].

The structure both of `rawFIFO.occ` and `FIFO.occ` consists of some initialization, followed by outer PARs with members ending in a `WHILE TRUE` loop. This is like the "setup" and "loop" approach found in many embedded IDEs like Arduino and its variants.

It begs the question of shutdown completely, since a `WHILE TRUE` loop means "go on for all eternity." It really means "go on until an uncontrolled failure like a power-off." So strictly speaking, it is an invalid code design. What it really means is "the output of this code will cease to matter before hardware failure causes its behavior to become undefined." In many simple embedded uses this is OK, as long as everyone is aware of it. In the focused hardware design case, there will actually have to be some work done beyond this pseudocode to make sure the FIFO quits gracefully.

The code, as we have noted above, implies that there are data buffers but no valid packet data at startup. However, there are other things to note. In both programs, `n` is defined but the external channels are not. This would normally mean they are passed in by a PROC definition embracing all the code, or by nested PARs. The reason the outer code has to be PARs is that this code has to be the last to be executed in its sequence, if there is a sequence. Otherwise the `WHILE TRUE`, which never terminates, will make the following code inaccessible and definitely incorrect. What this means is that the FIFO is outermost with respect to Deep nesting, which is just what you would expect, since it is hardware or emulates hardware.

Given this PAR structure embracing our FIFO code, with `in` and `out` declared in some outdented high place, another interesting possibility follows. The kind of thing shown in Table 3.1 of Chapter 3 is possible, with the clients using the FIFO. Since it outlives these clients, there is nothing in the `WHILE TRUE` loop to stop its being nonempty between clients. This is a consequence of a lack of proper shutdown that is an actual problem using buffered IO such as serial. If a running program is deaf to the keyboard buffer, for example, you can actually type in another command during its run, and that will flash on the screen and be executed right after the old program ends.

When doing data-flow (higher level) pseudocode, it will pay to consider shutdown. In occam, a Boolean "notdone" variable may be turned off by anything capable of interrupting the program, which not only loops on `WHILE notdone`, but all significant actions are enclosed by an IF test checking this variable. Then you actually have to design to the time pad offered by your Uninterruptible Power Supply (UPS) and provide a graceful shutdown at each point.