How the JSYS Instruction is Simulated on the KI10

Jack Freeman
March 13, 1974

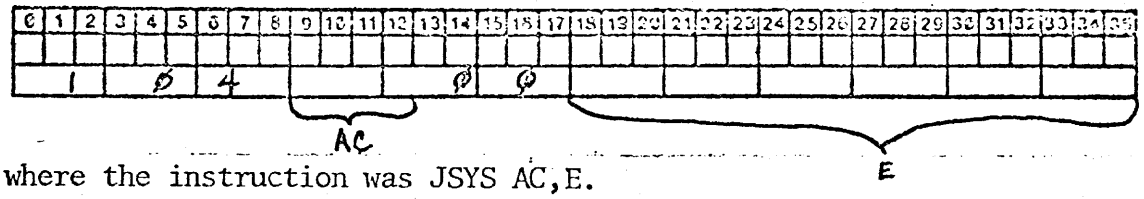
How JSYS works in the KI system.

1. Hardware

JSYS is opcode 104. The processor "executes" it as follows.

(a) Compute the effective address, E. (Note that this will be the JSYS number). Store in location 424 of the UPT* a word that looks like:

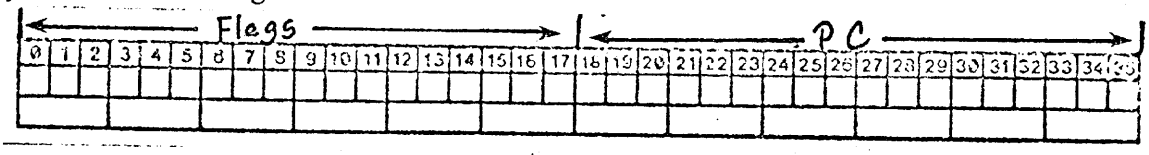
UPT + 424
(KIMUUO)



where the instruction was JSYS AC,E.

(b) Store the flags and PC in loc 425 of the UPT:

UPT + 425
(KIMUPC)



Note that PC is the location of the instruction next after the JSYS. Bit 5 (User mode) will be set in the case of a normal JSYS issued by a user program.

(c) Load the flags and PC from one of UPT + 430 through UPT + 437, depending on one thing and another.

	Flags	PC
UPT + 430:	0	KIMUOM
431:	0	KITRPM
432:	0	KIMUOS
433:	0	KITRPS
434:	USER IN-OUT	KIMUOU
435:	USER IN-OUT	KITRPU
436:	USER IN-OUT	KIMUOU
UPT + 437:	USER IN-OUT	KITRPU

UPT + 434 will be used for a normal JSYS issued by a user program.

* This is the User Process Table the KI10 hardware knows about, not the TENEX User Page Table.

2. Software (Normal, user-issued case)

Once the JSYS has been executed, control will go to KIMUOU (p. 9 of KISRV). The processor will be in kernel mode. The USER IN-OUT flag will be set. The fact that we're in kernel mode means that normal references to the accumulators will access register block ∅.

KIMUOU is one entry point into a "program" that handles all 4 "MUO" cases one way or another. The "trap" cases are handled by a separate program.

The code at KIMUOU saves ACs 1 & 2 in KIMAC1 and KIMAC2, two Monitor global storage cells (KISRV, p 2).

It loads KIMUUO (the word with opcode and effective address) AC2, then loads the opcode from there into AC1 and tests to see if it's JSYS. If not it jumps to the 10/50 MUUO handler. If so, it proceeds.

The word where the user PC and flags were stored (KIMUPC) is loaded into AC7. The effective address of the JSYS (in AC2 right) is tested for being $<1000_8$. If it is, this is a regular old JSYS. We jump to some code which is also used if regular old JSYSes are executed by the Monitor.

This is at KIMUO4 (KISRV, p 7).

We increment the global counter KIPJCT. This apparently just counts the total number of JSYSes executed. (Per-process would make more sense.)

The saved user PC and flags (from KIMUPC) are stored into FPC in the PSB.

It is time for an aside about what we're up to.

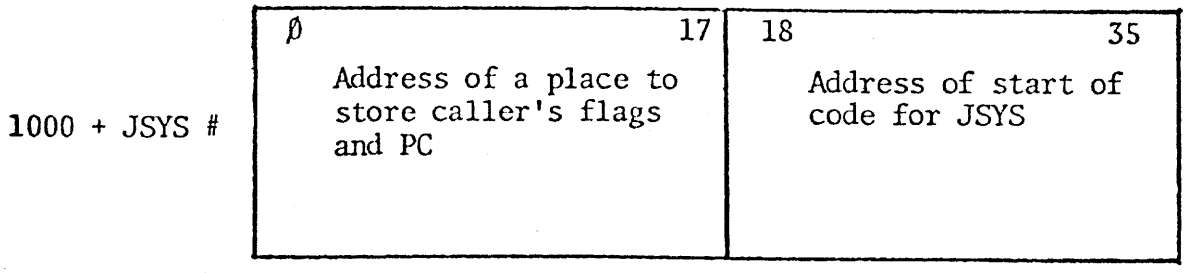
The way we got started was with some user program executing a simple little instruction:

```
4761:   JSYS        69
```

What's supposed to happen is that we end up at the code for JSYS 69. The address of the beginning of this code is at location 1000_8 + 69 in the Monitor's address space. One would think that about all we need to do is just jump to that location. And this is effectively what will usually happen, but there are various obscuring factors.

There are two sources of confusion. The first has to do with some logic to allow the interrupting of the flow of program control to run the Scheduler module before going on to process the JSYS. The second confusion factor just has to do with special kludgery for speeding entry into "slow" JSYSes.

To understand the screwing around at KIMUO4 one needs to know how the JSYS instruction, as it was implemented by BBN, worked. There is this transfer vector at location 1000 in the Monitor address space. An entry looks like this.



In BBN's hardware implementation, the caller's PC word (KIMUPC, for us) was automatically stored in the word addressed by the left half, and control

was transferred to the code addressed by the right half. This is what we are simulating.

The left halves of all the entries in the transfer vector point to the same location, namely FPC. So, we have taken care of this part of the simulation by our storing of KIMUPC into FPC. We have yet to take care of the transfer of control.

The first step is to compute the address of the start of the JSYS code. We want to form a flags-PC word with the current processor flags in the left half and the JSYS code address in the right half. We proceed as follows

```
JSP  1,. + 1; gets current flags into AC1 left.  Note that
              USER I/O will be set if the JSYS was executed in
              User mode, reset if it was executed in Monitor mode.

HRR  1, 1ØØØ(2); this gets the JSYS code address into AC1 right.
                 (recall that AC2 contains KIMUUO, which has the
                 JSYS # in its right half).
```

Now this leaves us with pretty good stuff in AC1. In fact a

```
JRSTF  Ø(1)
```

would get us to the JSYS code with USER I/O set correctly. But here's where the confusion factors come in. Ignoring the "slow" JSYS factor for a moment, we now describe the strangeness introduced by our desire to give the scheduler a chance to run.

The idea is that at some time in the recent past, someone may have determined that it was time to run the Scheduler but circumstances may have been such that

it wasn't possible to run it at that time. The "someone" will have just set a flag called KIP7F to indicate the imminent need to run the Scheduler. At various places where the Monitor is safely interruptable, this flag gets tested and the Scheduler gets run if it's set. One of these safe places is at JSYS entry, which is where we are right now.

The way we give the Scheduler its chance is not totally straight-forward. Instead of testing KIP7F directly and doing something reasonable like "calling" the Scheduler, we proceed as follows:

- We put our flags and PC word that we have constructed in AC1 into a global variable called KIP7P. (If the Scheduler gets entered, it will stuff KIP7P into its return link so control will go to our JSYS after the Scheduler is finished.)
- We restore AC1 and AC2 from KIMAC1 and KIMAC2 where we saved them back at initial entry (KIMUOU).
- We jump to KITRET (KISRV, p 19). This is common code executed from a bunch of places. Here is where the check is made to see whether to run the Scheduler.

  If KIP7F is non-zero, we are going to run the Scheduler. We increment KIP7Q (this flag will let the Scheduler know that it got entered this way and will cause it to exit "through" KIP7P). Finally, we "call" the Scheduler by by generating an interrupt on channel 7.

  If KIP7F is Ø, we jump directly to the JSYS code with a JRSTF @KIP7P. Note that in either case we will end up at the 1st instruction of the JSYS code and we'll have USER I/O set or reset according to whether we came from the user or the monitor originally.

The special kludgery for "slow" JSYSes needs to be explained. We have ignored the code for it in the above description. Many JSYSes (the so-called "slow" ones) begin by themselves executing a JSYS MENTR. This MENTR routine does various state-saving operations so that the real

JSYS the user is trying to call will be "interruptable." Typically, JSYSes which may take a long time to execute are handled this way - hence the name "slow" JSYS.

Apparently the guy that wrote this code we're describing couldn't stand the idea that, for slow JSYSes, the very first thing that would happen after he got through simulating the original JSYS would be another one. So he decided to kill two birds with one stone, and send control directly to MENTR. This gets done as follows:

At the point where AC1 has the address of the 1st instruction of the JSYS in its right half (i.e., right after the HRR 1,1000(2)), we fetch the first instruction into AC2 and test to see if it's JSYS MENTR. If not, we proceed as has already been described. But if it is, we

- Add 1 to AC1, so that it now addresses the 2nd instruction of the target JSYS.
- Store AC1 into XMENTR, which is MENTR's return link.
- Replace AC1 right with the address of the 1st instruction of MENTR itself. (This is the instruction labelled UUOH1).

And then proceed as already described. That is, store AC1 into KIP7P, restore AC1 and AC2 from where they were saved, and jump to KITRET to get transferred, eventually, to UUOH1 in MENTR. The idea is, of course, that with XMENTR set up like it is, when MENTR returns, control will go to the 2nd instruction in the target JSYS.