MULTIPROCESSING FOR THE IMPOVERISHED

Part 3: Mid-Course Corrections

by Brad Rodriguez

This article first appeared in The Computer Journal #66 (Mar/Apr 1994).

"[I]nnovation really happens by blundering through to success on the back of one's mistakes."
- Gifford Pinchot III, *Intrapreneuring*

BRAD'S FIRST MISTAKE

TCJ reader Andrew Houghton was quick to spot a potentially disastrous goof in my 6809 Request/Grant logic. The 6809 is a *dynamic* processor, which means its internal registers are like dynamic RAM -- they need to be constantly refreshed. This refresh occurs whenever the internal clock is running. But, unlike the Z80, the 6809 *stops* its internal clock when the MRDY (a.k.a. WAIT\) line is held low. The result: hold MRDY low for more than 16 usec, and the CPU registers become garbage! (This interesting fact is contained in a teeny tiny footnote in the 6809 data sheet.)

You can see that if eight CPUs are contending randomly for the bus, the odds of one being "shut out" for 16 usec or more are high. The solution has two parts: a) limit the amount of time any one CPU can grab the bus, and b) force the processors to use the bus in strict rotation. For example, if there are eight CPUs, and each holds the bus for no longer than 2 usec, no CPU will ever have to wait more than 16 usec for its turn at the bus.

Part (b) is easy -- it affects only the design of the bus arbiter. Part (a) is more tricky. Recall from the last article that the Read-Modify-Write logic works by stretching every bus request for two extra E cycles. Thus the bus is "held" until the Write portion of the RMW begins. A side-effect is that the bus is held for two E cycles *after* the write...a total of 5 E cycles for the complete RMW operation. If the next instruction also uses shared memory -- or worse, is executing from shared memory -- this stretched bus request may be continued by the next instruction, and the bus may be held indefinitely.

What we *really* want is a circuit that stretches the bus request only until the end of the Write. Since we don't know which memory references are RMW cycles, we'll settle for a circuit that stretches a bus request for *only 3 E cycles*, even if a second access (the Write) occurs. In other words, we don't want to "retrigger" the delay if another access occurs during the stretch period. I couldn't think of such a circuit offhand, so I dusted off my old college text and went through the full ordeal of formal state machine design (see sidebar). Suffice it to say that It Can Be Done, and the new delay circuit requires only replacing the D flip-flops with JK flip-flops, and adding one inverter (see schematic).

On a 1 MHz 6809, 3 E cycles require 3 usec, so up to five CPUs can share the bus in rotation without violating the 16 usec limit for any CPU. Using a 1.5 MHz 68A09 or a 2 MHz 68B09 allows eight CPUs.

BRAD'S SECOND MISTAKE

My second mistake was Thinking Small. I had decided that the simplest possible CPU board would be best for TCJ. But everyone who sends me mail seems to want the improved ScroungeMaster II! So, at the risk of abusing the patience of TCJ's readers (and editor), I'm going to complete this series of articles with the "SM II" design. If you've already bought parts for the 6809 uniprocessor, you'll be happy to know that only the 2681 DUART, 75176 transceiver, and a 74LS139 are discarded. I think you'll find the improvements worth the cost.

To keep our Esteemed Editor happy, I'll publish and describe the six pages of schematics in three installments. In this article, I will cover the (slightly revised) CPU, some new memory mapping logic, and address decoding. Following that will be memory, parallel I/O, an interface to IBM PC bus peripheral cards, and the bus arbiter. Third will be the serial I/O and some other frills.

THE REVISED CPU AND RMW LOGIC

Figure 1 [page 1 of the PDF file] shows the revised CPU circuit. U1 is the 6809, unchanged from the uniprocessor design.

OFFBD\ comes from the address decoding logic, and is pulled low during an access to the external bus. When this happens, U7A and U14 generate a "stretch" signal for two clock cycles (see sidebar). When either OFFBD\ is low or STRETCH\ is low, the REQ ("request") signal is output high by U6A. This signal is sent to the bus arbiter, to indicate that this CPU desires to obtain (or hold) the bus. With JP3 in the "3CLK" position (as shown), the bus will be grabbed for three E cycles, and Read-Modify-Write accesses will be indivisible. If JP3 is moved to the "1CLK" position, the stretch circuit is disabled, and the bus will only be grabbed for one E cycle at a time. (For experimentation with other mutual exclusion schemes.)

If this CPU does not "own" the bus, GRANT\ will be high. The combination of GRANT\ high and REQ high will output a low from U6B, clearing all the flip-flops of U15. This will cause all the Q\ outputs to be high, one of which is jumpered (via JP6) to U27A. This forces U27A to output a low, pulling MRDY low and halting the CPU. This function can be disabled (for experimentation) by jumpering JP6 to the bottom (ground) position.

GRANT\ is pulled low when the CPU acquires the bus. This releases the CLR input of U15, causing its outputs Q1\ through Q4\ to go low in one, two, three, and four clock cycles, respectively. JP6 selects how many clock cycles will be added before the CPU completes the memory access.

When both GRANT\ and OFFBD\ are low (and thus OFFBD is high), U7C and U6C pull the DRIVENBL\ signal low. This enables the three-state drivers on the external bus. Note that neither GRANT\ nor OFFBD alone is sufficient: if GRANT\ enabled the drivers, and the bus is still granted to this CPU when we read the on-board EPROM (for example), both the bus transceiver and the EPROM would attempt to drive the CPU's data lines simultaneously, causing a "conflict." If OFFBD\ enabled the drivers, this CPU could attempt an off-board access and activate its bus drivers while the bus was granted to another CPU, causing a conflict on the external bus.

The new external bus accommodates IBM PC peripheral cards, which may, for their own purposes, pull IORDY low to stretch the memory access. Obviously, this signal should be observed only by the CPU which is on the bus, so U27B gates IORDY with DRIVENBL\: *both* must be low to assert XWAIT high ("external wait"). XWAIT is logically ORed with the "internal wait" signal by U27A: when *either* wait is requested, the CPU's MRDY input is pulled low.

IBM PC peripheral cards may also assert an active-high interrupt. One of the five interrupt signals on the PC bus is selected by a jumper (not shown here) as the signal XIRQ. XIRQ is inverted to active-low by U27C. It may then be jumper-routed to either the CPU's IRQ\ input, or IRQ4\, an auxiliary interrupt input of an I/O chip. U27 is *not* open-collector, so *if the external interrupt is connected to IRQ\, no other interrupt source should also be connected to IRQ\.* JP4 can be removed entirely if external interrupts aren't used.

JP5 can connect the REQ signal to a programmed output pin, to allow experimentation with software request/grant schemes rather than hardware. For standalone use (not plugged into a bus), U6, U14, U15, and U27 may be removed. (MRDY is then pulled up to +5 by R9.) Terminal block J1 is a power connector for standalone use.

MEMORY MAPPING

The ScroungeMaster II expands the 6809 address space to 1 MB, and allows both memory and I/O access to IBM PC peripherals. Figure 2 [page 2 of the PDF file] shows the address generation and decoding logic. The 6809's 64K memory space is divided into nine regions:

```
address

0000-0FFF "page 0", 4K )
1000-1FFF "page 1", 4K )
2000-2FFF "page 2", 4K ) each of these 4K "pages"
3000-3FFF "page 3", 4K ) can be "mapped"
4000-4FFF "page 4", 4K ) to any 4K region in a
5000-5FFF "page 5", 4K ) 1 MB address space
6000-6FFF "page 6", 4K )
7000-7FFF "page 7", 4K )

8000-FFFF fixed EPROM region, 32K
```

The addresses from 8000-FFFF are "unmapped" -- the raw physical address lines from the 6809 CPU are connected to the EPROM, so that whenever the program reads 8000-FFFF, it *always* gets the EPROM. The addresses from 0000-7FFF are "mapped" -- they are passed through a circuit which converts the 16-bit address from the CPU (actually 15-bit, since we know A15 is zero) to a 20-bit address.

This is done by U2 and U3, a pair of 74S189 16 word x 4-bit high speed RAMs. Assume for the moment that the TASK line is pulled high (jumper JP2 removed). When the CPU outputs an address, bitsA12-A14 select one of eight locations in the fast RAM. Eight bits are output from that location (four from U2, and four from U3). These bits are the high eight bits of the "mapped address," MA12-MA19. Every chip except the EPROM uses these as if they were the "real" address bits output by the CPU. (Note that the low twelve address bits A0-A11 are unchanged, and are used by all chips.)

Suppose that these eight locations contain hex 12, 34, 56, 78, 9A, BC, DE, and F0, respectively. Then,

```
when the program reads 0000-0FFF, it actually gets 12000-12FFF.
when the program reads 1000-1FFF, it actually gets 34000-34FFF.
when the program reads 2000-2FFF, it actually gets56000-56FFF.
when the program reads 3000-3FFF, it actually gets78000-78FFF.
when the program reads 4000-4FFF, it actually gets9A000-9AFFF.
when the program reads 5000-5FFF, it actually gets BC000-BCFFF.
when the program reads 6000-6FFF, it actually getsDE000-DEFFF.
when the program reads 7000-7FFF, it actually gets F0000-F0FFF.
```

The 6809 can only access 32K of "mapped" memory at any one time (in eight 4K chunks), but by rewriting the mapping RAM (U2 and U3), all of a 1 MB address space can eventually be reached. (Note that U2 and U3 completely ignore A15, so they remain active even during EPROM accesses.)

The mapping RAM is written by a Write to an address in the EPROM space (8000-FFFF, A15 high). U7D and U8B detect the combination of A15 high, R/W\ low (write), and E high (data strobe), and produce the WRMAP\ signal. *The location which is written depends upon A12-A14.* Thus, to write the first mapping RAM location, write a byte to any address in 8000-8FFF.

```
to program the map for 0000-0FFF, write location 8xxx.
to program the map for 1000-1FFF, write location9xxx.
to program the map for 2000-2FFF, write locationAxxx.
to program the map for 3000-3FFF, write locationBxxx.
to program the map for 4000-4FFF, write locationCxxx.
to program the map for 5000-5FFF, write locationDxxx.
to program the map for 6000-6FFF, write locationExxx.
to program the map for 7000-7FFF, write locationFxxx.
```

Since the 74S189 has logically inverted outputs, you actually have to write the *complement* of the desired data to the mapping RAM. So, for the example above, you would write hex ED, CB, A9, 87, 65, 43, 21, and 0F, respectively.

Finally, the TASK input is connected to a programmable output bit on one of the I/O chips. When a '1' is output, the *last* eight registers in the 74S189s will be used for mapping. When a '0' is output, the *first* eight registers are used. This means that two independent maps can be stored in the mapping RAM, and switched by changing one bit. This could be used to support two tasks, or perhaps "main task" and "interrupt" memory maps. Note that you have to select map 0 before writing map 0, and likewise for map 1.

ADDRESS DECODING

U8C generates the read signal for the EPROM when A15 is high, R/W\ is high (read), and E is high (data strobe). All other chip selects and data strobes are generated from the mapped address. The 1 MB mapped address space is divided as follows:

```
00000-DFBFF: external bus, memory references (895 KB)
DFC00-DFFFF: external bus, I/O references (1 KB)
E0000-FFBFF: on-board RAM (127 KB)
FFC00-FFFFF: on-board I/O (1 KB)
```

The 8086 used in the IBM PC distinguishes between I/O and memory references. Since the 6809 has no such provision, some segment of its memory address space must be assigned to simulate the I/O signals. I have chosen to generate the I/O Read and Write signals when any location in the last 1K of the external address space is referenced (the IBM PC I/O space is 1K long). Memory Read and Write signals are generated for the other 895K.

This complex address map requires a two-level decoding scheme. Normally I prefer to minimize the number of levels of decoding logic, since this is in a critical timing path. (I particularly despise cascades of 74LS138s.) In this case I'm willing to incur the timing penalty of an extra level of NAND gates.

U4 generates a signal, IOZONE\, which is low when either the external I/O space *or* the on-board I/O space is accessed. This is easier to see if you look at the binary addresses for these two regions:

```
external I/O: 1101 1111 11xx xxxx xxxx
on-board I/O: 1111 1111 11xx xxxx xxxx
```

If MA17 is ignored, the logical AND of the remaining high address bits will correctly identify both regions. U8A generates a signal, ONBOARD\, which is low when either the on-board RAM or on-board I/O is accessed -- that is, whenever the top three mapped address bits are '1'.

The four possible combinations of IOZONE\ and ONBOARD\ identify the four memory regions: LCLIO (local I/O), RAM (local RAM), MEM (PC bus memory), and IO (PC bus I/O). These are combined in U5 with the R/W\ selection signal and the data strobe E, to produce four Read strobes and four Write strobes. You may wish to verify that all of the combinations are correctly decoded. Note that, by the nature of the 74LS138, only one strobe can be asserted at any time. Since the mapped address is always generated, even during EPROM accesses, these strobes must be blocked when A15 is high (EPROM space). This is done by feeding A15 into one of U5's active-low enable inputs.

The bus request logic needs a simpler signal. OFFBD\ is asserted (low) whenever an address in the external bus space is detected. We can't just use the logical inverse of ONBOARD\, since ONBOARD\ is unpredictable when accessing the EPROM. OFFBD\ is asserted, by U7E and U6D, when ONBOARD\ is high *and* A15 is low.

When IOZONE\ is low *and* MA17 is high, the address is in the on-board I/O space. When this occurs -- and, again, when A15 is low -- decoder U24 is enabled. This generates eight on-board I/O chip select signals, depending on the values of A7-A9. In effect, this divides the 1 KB on-board I/O space into eight 128-byte regions. Each of these regions will typically be occupied by one I/O chip. (You might think that the inclusion of MA17 is redundant, since LCLIOWR\ and LCLIORD\ are only generated when MA17 is high. But some I/O chips, such as the 6522, use R/W\ and E instead of the RD and WR signals, so the chip selects *must* be qualified by MA17 too.)

It's my habit to put all of the bypass capacitors (C1-C30) in one place on the schematic. This was the page where I had some extra room.

PC BOARDS

I'm pleased to announce that -- thanks to the interest of many TCJ readers -- I am proceeding with the layout and production of a PC board for the ScroungeMaster II. Estimated cost is US$20 each; the final cost won't be known until the layout is complete. If you would like boards from the first (and maybe only) production run, please contact me at b.rodriguez2@genie.geis.com on Internet, B.RODRIGUEZ2 on GEnie, or at Box 77, McMaster University, 1280 Main Street West, Hamilton, Ontario, L8S 1C0, Canada.

Note for web publication: the email and postal addresses given above are no longer valid. Contact the author at this email address.

SIDEBAR: STATE MACHINES

A digital circuit that contains flip-flops (or some other kind of storage) is usually called a *sequential circuit*. At any particular time, there will be some pattern of 1s and 0s in its flip-flops (or other storage); this is called the *state* of the circuit. A *state machine* is a sequential circuit that can change from one state to another, depending upon its inputs. (State machines can also be simulated in software, but that's not important here.)

State machines are designed by first identifying all the states which are required, and all of the possible transitions between the states. The decision of which pattern of 1s and 0s corresponds to which state is frequently left for last. This is because the logic can often be simplified, if a "non-obvious" pattern of 1s and 0s is used.

For example, here are the states of the new RMW stretch circuit, in tabular form. There are only three states, numbered 1-3, and one input (OFFBD\). The rightmost columns indicate the "next state", that is, the state to which the circuit will change on the next clock pulse, for any given input.

```
                  Next state if  Next state if
#  state          OFFBD\ low     OFFBD\ high

1 Idle             2              1
2 1st delay cycle  3              3
3 2nd delay cycle  1              1
```

The circuit will remain in the Idle state until OFFBD\ is asserted (low). Then the circuit will move through the two "delaying" states, *regardless of the level of the OFFBD\ input*. This is exactly the behavior we want -- the Write cycle does *not* retrigger the delay.

At least two bits (flip-flops) will be needed to represent three states. In this case, JK flip-flops require the least extra logic. The operation of a JK flip-flop when the clock pulse occurs is as follows:

```
inputs  outputs
J   K   Q   Q\

0   0   hold previous outputs
0   1   0   1
1   0   1   0
1   1   toggle previous outputs
```

In this case, I was able to have one of the flip-flop outputs directly represent the "stretch" signal (i.e., active during states 2 and 3), and also to represent Idle as 00, so that RESET\ could set the circuit to the Idle state. I encourage you to take the schematic and the JK table given above, and to verify that the circuit does step through all the states and produce the correct "stretch" output for any input combination. Start at state 00 (Idle), and note that the 74LS73 transition occurs at the falling edge of E. Observe also that the circuit cannot remain "stuck" in the unused fourth state, if it ever accidentally gets there.

Space prohibits a full description of the design techniques I used. This circuit is small enough that the simplest manual techniques were adequate: Karnaugh maps, flip-flop transition lists, and an exhaustive trial of state assignments [HIL74]. A much more powerful tool is the Quine-McCluskey method, which can be done manually or by computer. Newer and better methods have doubtless been devised; consult any good textbook on digital logic design.

Curmudgeonly observations: Alas, state machine minimization is going the way of long division: before long, no one will be taught how it's done. They'll just punch "divide" on their calculators, and click "minimize" on their PLD design programs. I call it "Engineering without Understanding." Harrumph.

REFERENCES

[HIL74] Hill, Frederick J. and Peterson, Gerald R., *Introduction to Switching Theory and Logical Design*, Second Edition, John Wiley & Sons, New York (1974), ISBN 0-471-39882-9. A classic, but probably superseded by many newer textbooks.