MULTIPROCESSING FOR THE IMPOVERISHED

Part 1: a 6809 Uniprocessor

by Brad Rodriguez

This article first appeared in The Computer Journal #61 (May/June 1993).

INTRODUCTION

"We need hardware articles," says TCJ's Editor and Publisher. So I'm taking a short break from my discussion of Forth kernels, to let you in on the Big Picture -- my Grand Project toward which all these articles are leading: a multiprocessor 6809 system. This month I'll describe the 6809 core; in a future issue I'll describe the multiprocessor bus and arbitration logic.

HISTORY AND DESIGN DECISIONS

Why on earth would anyone build a multiprocessor 6809 system?

Several years ago, when I was a "captive" employee, I developed a Forth kernel for a multiprocessor 68000 system. (I believe this was the first published application of Forth to tightly-coupled multiprocessors [ROD89].) The system employed a VME-bus crate, four 68000 processor boards, and a common RAM board . . . and cost about $10,000. Alas, I left that employer shortly afterward.

I've always wanted to continue that research into multiprocessing Forth, but VME bus products are out of my price range these days. Finally I decided to build my own multiprocessor system, using cheap technology -- preferably with components I could buy from Jameco, JDR, or B.G.Micro.

Two essentials for a multiprocessor CPU are a shared-bus request mechanism, and an indivisible memory operator (for semaphores). The 8086/8088 provides these through its bus LOCK mechanism, but only in the "maximum" mode, which requires a pile of support logic. The 68000's has a TAS (test-and-set) instruction, but 16-bit-wide memory doubles the number of memory devices, and who can find 64-pin DIP sockets? (Forget the 68008 -- only distributors carry them.) I briefly considered the Z8002 -- a new CPU would be fun to learn, and it has versatile multiprocessor support -- but it too uses 16-bit memory, and Z8000's are even harder to find than 68008s. (A local distributor quoted $25 each. Times four? Ouch.)

It finally dawned on me that what I wanted was a dirt cheap, commonly available, 8-bit microprocessor, like a Z80 or 6809. And while designing tentative bus arbitration circuits for the 16-bit micros, I realized that I really didn't need special multiprocessor support from the CPU! I now claim that I can parallel any CPU that:

1. can "stretch" its memory cycle to accommodate slow memory, i.e., has a READY or WAIT input;
2. has some direct memory modify instruction, such as ROR M or INC M.

(Strictly speaking, this second condition can be relaxed . . . but at a substantial cost in performance and in extra support logic.)

I was tempted to design around the Z80, since it's just about the cheapest suitable CPU -- but I'm up to my ears in Z80 systems. The 6809 costs only $1 more, is far superior for Forth work . . . and I had some old 6809 hardware designs lying around, and a Forth assembler and Forth kernel for it.

(I will, however, throw in some Z80 design notes. If someone wants to design and prototype a Z80 multiprocessor system on this model, I'm sure TCJ would be happy to print it!)

THE 6809 UNIPROCESSOR

The journey of a thousand miles begins with one step. - Lao-Tsze

. . . and the journey to a multiprocessor begins with a single CPU. New CPU boards are hard enough to debug without introducing peculiar memory logic. So, first build a uniprocessor and make it work; then add the multiprocessing extensions.

Figure 1 shows the 6809 "core": CPU and memory. The signal labels follow the OrCad convention, using a backslash instead of an over-bar to indicate logical inversion (e.g., RD\ is "read-bar," an active low read strobe).

The power-on reset circuit and the clock oscillator are "external to CPU board." This is for two reasons:

a) in the multiprocessor system, all CPUs can share a common reset and clock, so these don't need to be replicated on each CPU card.

b) the 4 MHz 6809 can share the 3.6864 MHz clock required by the 2681 DUART's baud rate generator, saving one crystal.

If you're just building a single 6809 processor, you can use the 6809's internal oscillator by tying a crystal (4 MHz or less) to the 6809 pins X1 and EX2. A second crystal (3.6864 MHz) can be tied to pins X1 and X2 of the 2681.

Note that the oscillator and reset circuits both use sections of U8, although they require a 74HC04 and 74HCT14, respectively. The oscillator follows the T.I. application note for HCMOS inverters [TEX84], except that I lowered R3 to 10K to get it to oscillate at 3.6864 MHz. (If the circuit oscillates at too low a frequency, R3 is too high.) My original reset circuit used the 74HC04, but every variation I tried gave noisy reset pulses. Finally I was forced to do it "properly," and I plugged in a 74HCT14 Schmitt trigger inverter. Fortunately, the oscillator still seemed to work with the 74HCT14, although by all rights it shouldn't have. Probably you should play safe and use two different parts.

The full multiprocessor CPU design puts 4 gate loads on the oscillator. For eight CPUs this is 32 loads, which even in CMOS is too much for comfort. So, the oscillator input to each board is buffered by U7B. U7C (on figure 2) provides the complimentary clock for the 2681. Note that U7 must be an HCMOS part, to output a sufficient "high" level for the 2681's clock input. You can use LSTTL if you add 470 ohm pullups to +5 to the 2681's clock inputs X1 and X2.

You might be tempted to use the 6809E for this project. Don't! The 6809E, an external clock version of the 6809, requires a quadrature clock input, and does not include the MRDY logic. Use a plain 6809, or the higher-speed versions 68A09 or 68B09.

The 6809 is easy to interface to memory. Every CPU clock cycle is a memory cycle, and the E clock output is also the active-high data strobe. R/W\ is high for read cycles and low for write cycles. When the processor is doing internal operations, address FFFF is output on the address bus and R/W\ is high, causing memory reads to location FFFF. (This must be a valid memory location, since it is the low byte of the 6809 reset vector.)

I find separate RD\ and WR\ strobes more useful for most memory and I/O devices. These can be generated from R/W\ and E with an inverter and two NAND gates, but I prefer to use half of a 74HCT139 (U5B). It's easier to wirewrap, and I had a spare '139 section on the board anyway.

Figure 3 shows the timing diagram for the CPU [MOT83]. The CPU runs at 1/4 of the oscillator frequency, and outputs clock signals E and Q in phase quadrature (Q leads E by 90 degrees). With a 3.6864 MHz clock, the data strobe E is high for 540 nsec, plenty for even slow memories. Note that the address is valid at the rising edge of Q. You can generate a wider data strobe by using E+Q (the logical OR of E and Q); this may be desirable if you use a faster 6809.

Part of the design problem is deciding how much RAM and ROM to include. I've divided the 64K address range as follows:

0000-1FFF on-board RAM #1, 8K
2000-3FFF on-board RAM #2, 8K
4000-5FFF external (multiprocessor) bus, 8K
6000-7FFF on-board I/O
8000-FFFF on-board PROM, 32K

I contemplated using an 8K PROM, which is enough for a Forth kernel. But I have extra 27256's in my parts box, and I thought that, as I develop the multiprocessor extensions to Forth, I'd want to put them in PROM. Since the 6809 holds its reset and interrupt vectors in high memory (addresses FFF0 to FFFF hex), the 27256 PROM, U2, is mapped into the high 32K of the address space. Whenever address line A15 is high, U6A asserts CSROM\ low.

The remaining 32K of address space is divided into four 8K regions by U5A. Whenever address line A15 is low, U5A will assert one of four strobes, depending on A14 and A13. I have a surplus of 6264 8K RAM chips left over from a previous project, so I designed the board to hold two of these (U3 and U4).

In any multiprocessor system it is important to have "private" memory for each processor, and "public" memory for all processors to share. An 8K region is mapped to this "public" memory. When -- and only when -- the CPU accesses any address

in the range 4000 to 5FFF, it requests the use of the external (shared) memory bus. (More on this in the next article.) I've arranged the address map so the on-board RAM and the external RAM form a contiguous 24K region.

For simplicity of decoding, I've allotted an 8K region to the 2681 DUART. This is extravagant, since the 2681 requires only 16 memory locations.

Figure 2 [page 2 of the PDF file] shows how to use a supplementary address decoder to divide this 8K space into four 2K regions; this allows three I/O devices in addition to the 2681. Note that some of the decoding done by U6A is duplicated in the 74HCT138, U10. This illustrates an important design principle. You might generate eight chip selects instead of four, by routing CS6\ to an active-low gate input of U10, and then decoding A10 through A12. But this would cascade the propagation delays of two decoders! After many CPU projects, I have concluded that address decoding should be done in parallel, preferably with no more than one stage of decoder delay to any address strobe.

U5, U6, and U10 can be 74LS parts if desired. Note also the temporary connections of the 6809's FIRQ\ and MRDY\ to VCC. These inputs will be used later, for the full multiprocessor CPU.

Figure 2 shows the serial I/O, built around a Signetics SCN2681 DUART (Dual UART). This versatile chip has two UARTs with programmable baud rate generators, a counter/timer, a 7-bit input port, and an 8-bit output port. (This and the Z8530 are my two favorite serial chips.)

The 2681 is unusual in that it requires an active-high RESET signal, provided by U6D. Incidentally, I've used two sections of a quad NAND and two sections of a quad NOR to provide four inverters because the full multiprocessor design needs NAND and NOR gates. You see the "leftover" gates here as inverters.

The interrupt-request output of the 2681, INTR\, is tied to the 6809's IRQ\ input. Since this is an open-drain output (the MOS equivalent of open-collector), it needs a 10K pullup to +5.

The "A" port of the 2681 is an RS-232 port, using a 1488 and a 1489 for drivers and receivers. Since I have extra 9-pin D connectors, plus an abundance of adapters, cables, null modems, and whatnot for the IBM PC/AT, I've made this serial port look exactly like a PC/AT serial port. This means I have to use a null modem or other reversing cable to connect to my PC.

The 1489 has one receiver too few, so the RI (Ring Indicator) signal is left unimplemented. The 1488 has one extra driver, so I use it to drive an LED. I always find a way to put an LED on a new CPU board; it is invaluable for debugging.

Port "B" of the 2681 is a bidirectional RS-485 port. I'm experimenting with two-wire RS-485 local area networks, and the 2681 supports the 9-bit (address mark) format used by many modern microcontrollers. One DUART output line is used as a direction control.

The high three bits of the output port are reserved for a page select on the multiprocessor bus (more on this in the next article). One output is unused. Note that the four unused inputs are tied to ground; this is most important for a MOS device like the 2681! I learned the hard way: floating MOS inputs pick up noise and cause rapid transitions, which in turn puts noise on the power bus and sends the device's power dissipation through the roof.

Another warning from the 2681 School of Hard Knocks: do not access the 2681 registers which are "Reserved!" (Read Registers 02, 0A, and 0C, and Write Register 0C.) Even reading a reserved register will throw the 2681 into a self-test mode that ignores the RESET input. Once you've done this, only powering off and back on will restore normal operation.

The power connections and the bypass capacitors on each chip aren't shown. I use a three-voltage (+5, +12, and -12) supply so that I can use the 1488s and 1489s. I could have used MAX232's and a single +5 supply, but when you multiply the cost by eight CPUs, the three-voltage supply is the better bargain.

TEST SOFTWARE

Listing 1 is the test program I have used to exercise this board. I include it here mainly to illustrate the initialization of the 2681; refer to the Signetics manual [SIG86] for more details. This code is written for PseudoCorp's freeware PseudoSam Level I assembler, A689, which I obtained from the Real-time Control Forth Board (RCFB) BBS in Denver, Colorado, phone (303) 278-0364. PseudoCorp also advertises their professional assemblers in TCJ.

A full Forth kernel for this board, and a metacompiler that runs under F83, will be appearing soon on the Forth Interest Group Roundtable (FORTH) on GEnie.

SOME IMPOVERISHED OBSERVATIONS

I call this system the ScroungeMaster I, because I'm particularly proud of how cheaply I've obtained the components. I found 44-pin .156" edge connectors at a Toronto surplus shop. Vectorboards to match, and wirewrap IC sockets, came from the Trenton Computer Festival. The 8K RAMs and 27256s were leftovers from old projects, and almost all of the glue logic came from closeout sales. The 6809s were $2.50 apiece from B.G.Micro. Ironically, the most expensive part was the 2681 DUART, for which I had to pay Jameco the full retail price of $6.95 each! (Maybe I should have redesigned to use up all those Z8530s I have lying around . . .)

SIDEBAR: MEMORY DECODING FOR PROGRAMMERS

You say your most intimate contact with a CPU has been assembly language programming? Or you've been a Z80 man all your life, and don't know what the E signal is? Read on...

Most computers have separate CPU and memory chips, so some kind of control signals are needed to transfer data between them. If you've programmed in assembly, you're familiar with the **ADDRESS**. The CPU always sends this to the memory, to tell it which memory location to access. On Z80s and 6809s there is one physical line -- a pin on the chip, and a wire on the board -- for each of the 16 address bits. These are called A0 through A15. Most CPUs call the least-significant bit A0, and the most-significant A15. When an address bit is '1', +5 volts is output on the corresponding pin; when it is '0', 0 volts is output. These are "TTL levels," and "active high".

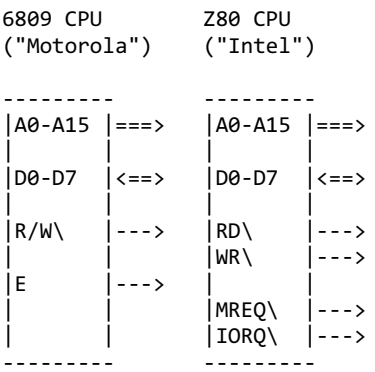
(Some other processors "multiplex" the address, but don't worry about this now -- it would just confuse things. Also, the "high" voltage can be anything from +2.4 to +5 volts, and the "low" voltage anything from 0.0 to +0.8 volts, but we usually say "+5" and "0" for brevity.)

You're also familiar with the **DATA**. Unlike the address, this can go two ways. If you are reading the memory -- as in a Load instruction -- the data lines are output by the memory chip, and input by the CPU. If you are writing the memory -- as in a Store instruction -- the data lines are output by the CPU, and input by the memory. That's why these pins on the chips are called "bidirectional." Like before, the eight data lines are numbered D0 to D7, with D0 usually the least significant bit; and like before, +5 volts is put on the line for a binary '1' and 0 volts for a binary '0'.

From this you can see that two important pieces of information are needed to transfer data, namely:

(a) which way is the data going, to or from the CPU?, and
(b) when should the transfer take place?

This information is always provided (output) by the CPU. There are two common schemes for this, which I call the "Motorola" and "Intel" approaches. They are illustrated here:



The Motorola scheme uses separate control signals for (a) and (b). The signal **R/W** is +5 volts for a CPU read, i.e., data transfer from memory to the CPU (Load). It is 0 volts for a CPU write, i.e., data transfer from CPU to memory (Store). The name is mnemonic, and should be read as "Read / Write-Bar". The backslash is an OrCad convention -- this is as close as the ASCII character set can come to an over-bar, the usual symbol for logical inversion. This means that when the signal is high, it's a read, and when it's low, it's a write.

The signal **E** is the "data strobe." When E goes active (high), it means that the address has been output on the A0-A15 lines, and the data transfer can take place. If it's a CPU write, it also means that the data has been output on D0-D7. If it's a CPU read, it means that the memory can now put data on the data lines. When E goes inactive (low), the data transfer is over. The CPU stops

outputting address and (on a write cycle) data. Note that the memory has no control over the timing: when on a read cycle, the memory had better output its data before E goes inactive!

The Intel (and Zilog) approach also uses two control signals, but they carry the information in a combined form. One signal, **RD** ("Read-Bar"), simultaneously informs the memory that it is a CPU read cycle, and that it's time to transfer data. The cycle begins when RD\ goes low, and ends when RD\ goes high. This is called an "active-low" signal, and is why it's labelled Read-Bar. The second signal is **WR** ("Write-Bar"), which simultaneously says that it's a CPU write cycle, and that it's time to transfer data. This too is an active-low signal.

Two other signals appear on some Intel and Zilog chips, such as the Z80. This is because these chips distinguish between memory devices and I/O devices. When **MREQ** ("Memory Request") is low, it means that the read or write cycle is to a memory. When **IORQ** is low, it means that the read or write cycle is to an I/O device (such as a UART or disk controller). Obviously, these two signals are never both active (low) at the same time. Electrically, memories and I/O devices look the same to the CPU, so Motorola treats them both as though they were memory.

A final word about decoding: if there are several memory (or I/O) chips, only one can transfer data at a time. This because they all share the same data lines (the data "bus"). So, you need some way to select which chip is to participate in any data transfer. This is usually done by recognizing, or "decoding", certain combinations of the high address bits. For example, this article's 6809 board uses extra logic devices to decode the following combinations of the three highest address bits:

```
A15 A14 A13
 0   0   0 selects the first RAM chip
 0   0   1 selects the second RAM chip
 0   1   0 selects nothing (yet)
 0   1   1 selects the 2681 UART chip
 1   X   X selects the PROM chip
```

"X" indicates that either a '0' (low) or a '1' (high) will be accepted. These are called "don't-cares".

To facilitate this decoding, most memory and I/O chips have a "Chip Select" (CS\) or "Chip Enable" (CE\) input. Usually these are active low: to access any given chip, it's Chip Select must be held low. When the chip select input is held high, that chip completely ignores the address, data, and control signals.

SIDEBAR: WHAT ARE THOSE OTHER FUNNY SIGNALS?

So you understand address, data, R/W\, and E...what are all those other signals on the 6809 CPU?

X1 and **EX2** are where you connect a crystal, to provide the frequency reference for the 6809's clock oscillator. The 6809 can also accept a clock signal from an external source: put the clock signal on EX2, and tie X1 to ground. The "E" in "EX2" reminds you that it is the pin to use for External clock.

E is (you may recall) the active-high data strobe output by the CPU. It is also the internal CPU clock, which runs continuously at one-fourth of the crystal (or external oscillator) frequency. The design of the 6809 is such that every clock cycle is a memory reference cycle, so this one signal can do double duty as clock output and data strobe.

Q is the "quadrature" clock output. This is another internal clock signal of the 6809. Q is the same frequency as E, but 90 degrees out of phase. (Q leads E by 90 degrees.) This is sometimes useful for odd timing logic, such as that required by dynamic RAM. Q can also be used to "stretch" the memory cycle of the 6809. Often it can be simply ignored.

MRDY is the "Memory Ready" input. Elsewhere I said that the timing of the memory cycle was under control of the CPU. This signal gives memory and I/O devices a way to inform the CPU that they need a longer cycle. When MRDY is held low ("Not Ready"), the CPU will keep outputting the address and (if appropriate) data, and will hold the E signal (data strobe) high. When the memory device has had enough time, it pulls MRDY back high ("Ready"), allowing the CPU to complete the memory cycle. On the Z80, the equivalent signal is called WAIT\.

The **RESET** input, when pulled low, stops and completely resets the CPU. Any information in the CPU is lost; but the memory usually remains valid. When RESET\ is pulled high, the CPU starts executing machine code at an address which is stored in memory locations FFFE:FFFF. Obviously this should be PROM!

IRQ is the Interrupt Request input. When this is pulled low, the CPU starts executing machine code at an address which is stored in memory locations FFF8:FFF9. Unlike RESET\, IRQ\ saves the state of the program which had been executing. When the interrupt code is finished, it can resume the previous program exactly where it left off. Software can "turn off" this input.

NMI is the Non-Maskable Interrupt input, which works the same as IRQ\ except that a) it fetches a code address from FFFC:FFFD, and b) there is no way for the program to disable this input.

FIRQ is the Fast Interrupt Request input, which works the same as IRQ\ except that a) it fetches a code address from FFF6:FFF7, and b) it doesn't save all the CPU state. The "unsaved" part of the CPU state can be saved by the software, if desired.

A note for the technical purists: IRQ\ and FIRQ\ are "level-sensitive," which means they cause an interrupt as long as the input is low. NMI\ is "edge-sensitive," so it only causes an interrupt when it makes a transition from high to low.

The **HALT** input, when pulled low, stops the CPU. But unlike RESET\, no data is lost. When HALT\ is pulled back high, execution of the program will continue where it left off. Z80 fans should note that the Z80 has a HALT output, which performs a different function.

Note: The schematic shows RESET\, IRQ\, NMI\, FIRQ\, and HALT\ as active-low, not with an over-bar, but with a little inversion circle at the input to the chip.

The **DMA\/BREQ** input stands for "Direct Memory Access/Bus Request." This provides a way for some other device to read and write the CPU's memory chips. When DMA\/BREQ\ is pulled low, the CPU suspends its program and stops outputting R/W\, address, and data. Then some other device -- usually a "DMA controller" chip -- can use the address and data busses and the R/W\ line to transfer data to and from memory, independently of the CPU. (The DMA controller is expected to use E -- which the CPU continues to output -- as its data strobe.) When large blocks of data need to be transferred, Direct Memory Access is usually faster than software. Many disk controllers use this technique. On the Z80, this signal is called BUSREQ\.

BA and **BS** stand for "Bus Available" and "Bus Status," and are output by the CPU to indicate one of the following four conditions:

```
BA BS
 0  0 normal (running)
 0  1 interrupt or reset acknowledge cycle
 1  0 synchronize (wait for interrupt)
 1  1 halted, or Bus Request (DMA) granted
```

The most common use of BA and BS is to provide the Interrupt Acknowledge and DMA Grant signals required by some I/O chips.

The Z80 uses an entirely different scheme to indicate processor status, so there's no easy comparison.

REFERENCES

[MOT83] Motorola, Inc., 8-Bit Microprocessor and Peripheral Data, Motorola data book (1983).

[ROD89] Rodriguez, B. J., "Multiprocessor Forth Kernel," Forth Dimensions XI:3 (Sep/Oct 1989).

[SIG86] Signetics Corp., Microprocessor Data Manual 1986. More recent Signetics data books should also carry the 2681.

[TEX84] Texas Instruments Inc., High-Speed CMOS Logic Data Book (1984).

AUTHOR'S BIOGRAPHY

After twelve years of designing and programming embedded systems, Brad Rodriguez decided he didn't know everything, and went back to school. He is now working full time toward a Ph.D. in Computer Engineering, focusing on real-time applications of artificial intelligence. He still does a little work "on the side" as T-Recursive Technology, and can be contacted as b.rodriguez2@genie.geis.com on the Internet, a.k.a. B.RODRIGUEZ2 on GEnie. The telecommunicationally disadvantaged can write to him at Box 77, McMaster University, 1280 Main St. 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.