HOW TO DESIGN YOUR OWN MICRO PROCESSOR

How to Design Your Own Microprocessor

By David C. Wyland
Microprocessor Design Manager

A computer is a memory controller. Its function is to move and combine data in the memory unit it controls. A computer can be used as a universal digital interface if the equipment being interfaced is made to look like computer memory. In the current controversies over computers, minicomputers, and microcomputers, these simple definitions tend to be overlooked.

All computers are basically alike. The only difference among them is the size and speed of the memory controlled, the efficiency with which the data can be moved or combined relative to a given application, and peculiar restrictions on interfacing I/O devices. Even today's terminology of microcomputer, minicomputer, etc., provides no clear separation among what are commonly thought to be different classes of machines. The only consistent definition appears to be based on physical size. For instance:

A microcomputer	=	100 cm^3
A minicomputer	=	0.1 m^3
A midicomputer	=	1 m^3
A computer (IBM)	=	10 m^3
A super computer	=	100 m^3

This list can be verified by the fact that some minicomputers can out perform some of the larger conventional full scale computers. Likewise, microcomputer performance is approaching that of the simpler minicomputers.

The significant characteristics of a computer/memory controller are:

A. Size of the memory controlled in words
B. Width of the memory word in bits
C. Memory speed
D. Efficiency of moving and combining data within the memory for the intended range of applications
E. Any peculiar restrictions on I/O device attachment, such as limitations on the number of I/O devices which may be physically attached, special I/O device instructions for addressing the I/O device registers rather than having them appear as memory locations, etc.

Most current computer/memory controllers have the general structure shown in Figure 1. A computer consists of a memory, input and output devices (I/O devices), and a Central Processing Unit (CPU). The CPU portion provides the memory control function and therefore defines the structure of the system. For this reason, the terms "CPU" and "computer" are often used interchangeably, although a CPU is a part within a computer.

The Central Processing Unit (CPU) of Figure 1 consists of a Program Counter (PC), an Instruction Register (IR), instruction execution logic, a Memory Address Register (MAR), a General Purpose Register (GPR) file, and an Arithmetic and Logic Unit (ALU).

The CPU communicates with memory and I/O devices over a memory bus. This bus has various names in different computers; such as, I/O Bus, Data Bus, etc., as well as a host of

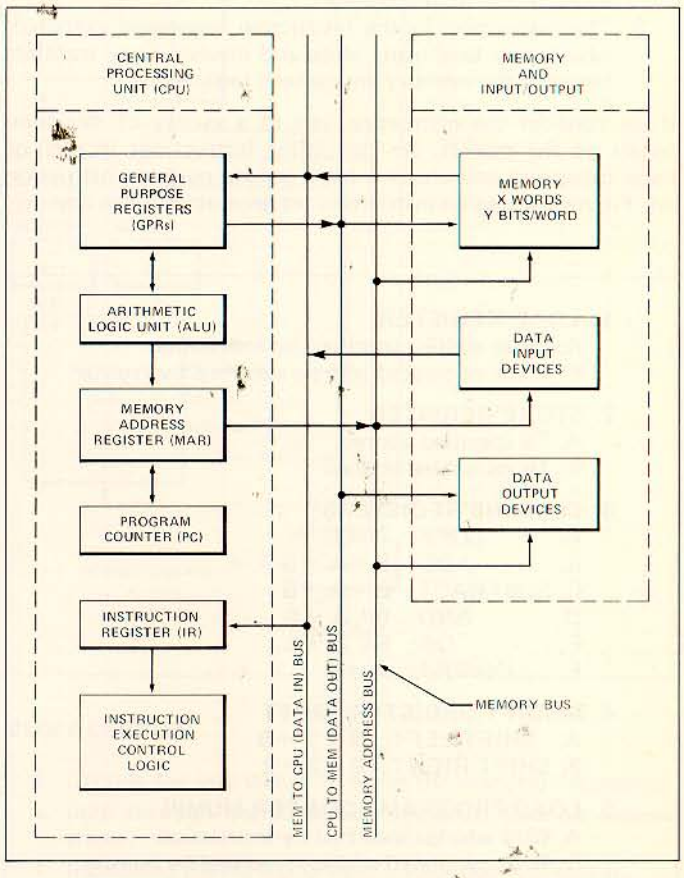


Figure 1
GENERAL COMPUTER STRUCTURE (with MAR)

unique trade names. The memory bus consists of a memory address bus, a memory data to CPU bus, and a CPU data to memory bus. These three buses are typically combined into one time shared bus. The memory address bus is driven by the Memory Address Register (MAR), which defines the memory address for data transfer. Data is transferred between the memory and the CPU registers over the memory data bus(es).

The General Purpose Register (GPR) file is typically 2 to 16 registers in size. The GPRs hold temporary memory data and addresses, and are used to move and combine memory data. Movement is performed by transferring data from memory to a GPR, and then from the GPR to the new memory location. Combination is performed by transferring data from memory to the GPRs performing the combination among the GPRs using the ALU, and returning the result from the GPRs to memory. GPRs may also be used to calculate memory addresses for move and combine operations.

Computers move and combine memory data according to a list of instruction words stored in the same memory. Each computer has a Program Counter (PC), an Instruction Register (IR), and instruction execution logic. The PC defines the location of the next instruction to be executed. The IR holds the instruction word for the current instruction being

executed. The instruction execution logic causes each instruction to be retrieved (fetched) from memory, decoded, and executed. The instruction execution logic does this in the following manner:

1. The contents of this Program Counter are sent to the Memory Address Register to define the location of the next instruction to be executed.
2. The contents of the memory at the address are loaded into the instruction register.
3. The instruction in the instruction register is executed, which may take many steps and involve many transfers between the memory and general register file.

If we consider the instruction sets of a variety of minicomputers on the market, we can define instructions in each of these categories and create a basic general purpose instruction set. Figure 2 shows an instruction set generated in this manner.

1. LOAD REGISTER
- A. From address specified by instruction
- B. From calculated address specified by register

2. STORE REGISTER
- A. To specified address
- B. To calculated address

3. COMBINE REGISTERS
- A. COPY: $A \rightarrow B$
- B. ADD: $B + A \rightarrow B$
- C. SUBTRACT: $B - A \rightarrow B$
- D. AND: $B \wedge A \rightarrow B$
- E. OR: $B \vee A \rightarrow B$
- F. INVERT: $\overline{A} \rightarrow B$

4. MODIFY REGISTER: SHIFT
- A. SHIFT LEFT: $B \times 2 \rightarrow B$
- B. SHIFT RIGHT: $B \div 2 \rightarrow B$

5. LOAD PROGRAM COUNTER (JUMP)
- A. With address specified by instruction
- B. With calculated address specified by register

6. LOAD PROGRAM COUNTER AND SAVE OLD VALUE (JUMP TO SUBROUTINE)

7. TEST (RESULT OF PREVIOUS COMBINE OPERATION AND LOAD PC IF:
- A. Result was zero
- B. Result was negative
- C. A carry was generated

8. ILLEGAL INSTRUCTION

Figure 2
BASIC COMPUTER INSTRUCTION SET WITH ILLEGAL OP–18 instructions

Several comments about the instruction set of Figure 2 are required:

1. In the case of the shift instruction, the bit shifted out is saved and zeroes or a specified bit is shifted into the location vacated.
2. The Load Program Counter and Save Old (Jump to Subroutine) instruction provides the ability to set the PC to a new value, execute a list of instructions, and then return to the original list and continue. This is a useful and powerful technique which allows programs to be sectioned into modules, called subroutines, which are then executed in sequence from the main program by the above procedure.
3. Note that no instructions have been included for hardware input and output. Individual registers for hardware input and output are assumed to have memory addresses so that input and output operations appear no different from other transfers between the general purpose registers and memory. This technique is currently in wide use in the Digital Equipment Corp. PDP-11 minicomputer.
4. Note that an illegal instruction has been defined. This covers the case of hardware or software error which results in the accidental interpretation of data as an instruction. Since most data consists of small positive or negative numbers, operation codes of all zeroes and all ones are usually reserved as illegal instructions.

One of the best ways to illustrate the above principles is to design a computer/memory controller. One of the first questions which must be asked, is what kind of a memory do we propose to control? If we examine activity in the minicomputer field, popular memory sizes range from a minimum of 4096 words up to 32,768 words. The most popular width is sixteen bits. Speed of the memory is essentially defined by current technology and is in the range of 0.1 to 2.0 microseconds.

Sixteen bit word size allows direct specification of memory addresses for memories of up to 65,536 words. This covers the popular ranges described above. This means that the GPRs can hold the results of both address and data calculations. Note that most of the currently popular microprocessors are 8 bit types. Since 12 to 15 bits are required to specify from 4096 to 32,768 words, respectively, these machines require several program steps or special instructions to do address calculations on 16 bit address words.

Let us design a 16 bit computer using four of the Monolithic Memories' 6701 microcontroller chips. A block diagram of one of these chips is shown in Figure 3. These chips consist of a 16 register file which may be read simultaneously by two address multiplexers, A and B. Data contained in the selected registers passes through the A and B latches, respectively. A and B data are selected by applying four bit binary codes to the A and B address inputs respectively. The B address inputs are also used to select a register to be loaded with new data. The A and B latches hold the A and B output data during a B load operation to provide edge triggered, master/slave type operation. The remainder of the chip consists of an ALU similar to the 74181 and various multiplexers to provide data routing and shifting of results before storage. An auxiliary Q register is provided, with its own shift multiplexers. This register can be used for temporary storage of results and for double precision shift operations in conjunction with the B shift multiplexers.

The 6701s will provide us with 16 GPRs and an ALU. If we assign one of the 6701 registers as the Program Counter, this still leaves us with 15 GPRs. The only items which must be added to the 6701s to complete the CPU block diagram of Figure 1 are an Instruction Register, a Memory Address Register, and instruction execution control logic.

In order to design the instruction execution control logic we must define the instruction format and the execution sequence for each instruction. The instruction format defines how the instruction will be decoded, and the execution sequence defines the steps required in the sequencing logic to execute the instruction.

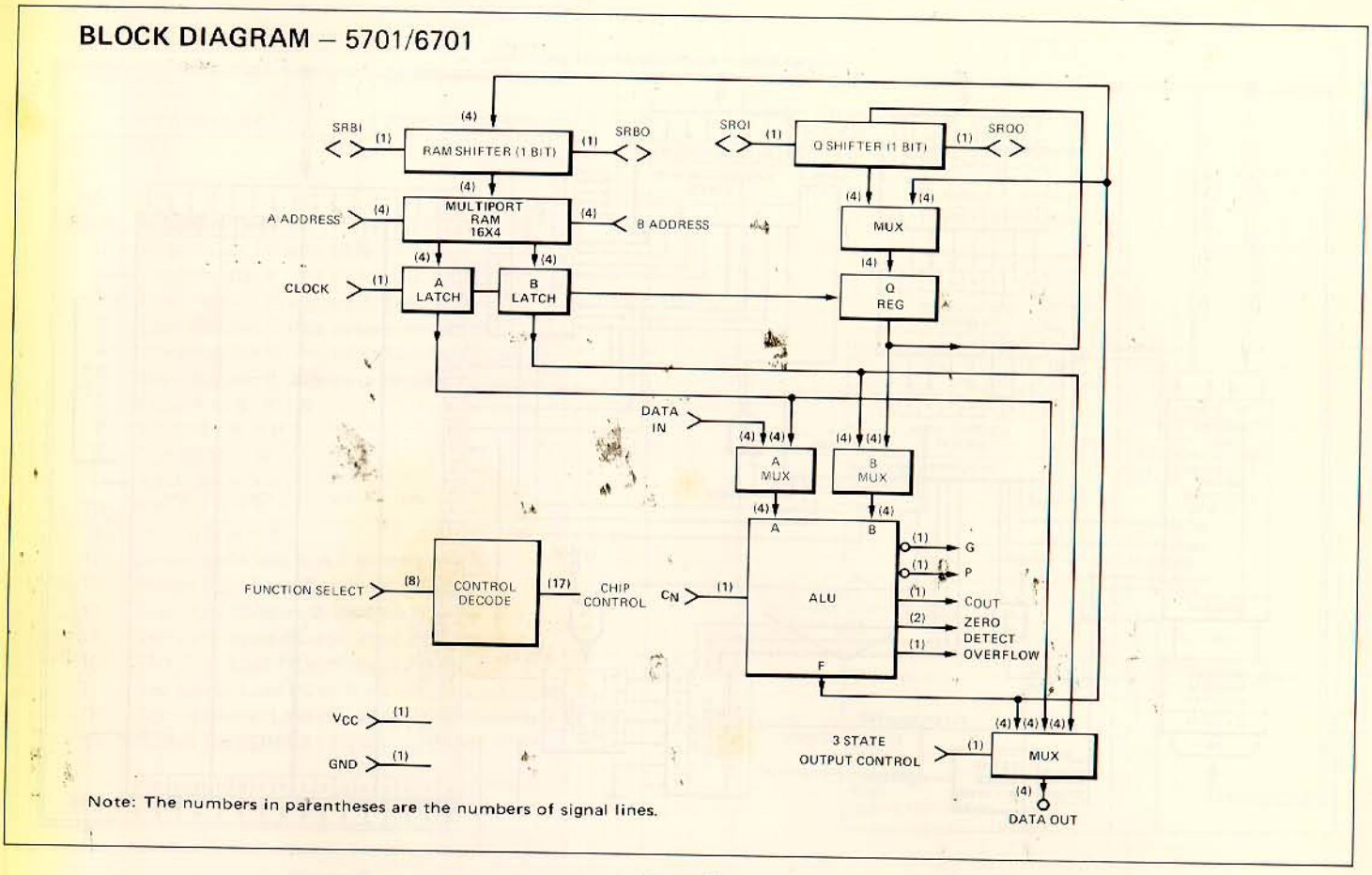


Figure 3
6701 MICROCONTROLLER BLOCK DIAGRAM

In order to define an instruction format for each instruction we must know the instruction word size, the number of GPRs, and the number of instructions in the instruction set. We have defined the memory word size as 16 bits, and the number of GPRs as 15. If we assume the instruction set of Figure 2 we have a total of 18 instructions to implement. This means that our operation code field must be at least five bits in size to uniquely specify each of the 18 instructions. The modifier fields required for each instruction are determined by listing the additional information required to execute that instruction. For instance, in a Load Register instruction we must specify one of the 15 GPRs to be loaded, and a 16 bit address for the memory data. After listing all required modifier data, the resulting instruction formats are then examined for similarity. Similar formats are then reduced to a minimum number of common formats, preferably one, for ease of remembering and ease of implementing the instruction execution control logic.

For the machine we have defined, all instructions can be specified by a single instruction format as shown in Figure 4. This format consists of an 8 bit operation code field and two 4 bit fields for defining up to two GPRs. A second modifier word following the instruction word is added for definition of memory addresses and immediate data for GPR Load and Store, Load PC, and Test instructions.

The instruction execution control logic generates a sequence of machine operations which do the following:

1. Get the next instruction from memory and load it into the instruction register. Memory location is defined by the contents of PC.
2. Decode the instruction to select the execution sequence.
3. Step through the execution sequence (one or several steps).
4. Increment the PC to the next instruction and repeat.

Steps 1 and 4 are often combined and the PC is incremented after the IR is loaded. In the case of Test instructions, one of two sequences is selected in Step 3: a load PC sequence if the test condition is satisfied or a dummy, no operation sequence if the test is not satisfied.

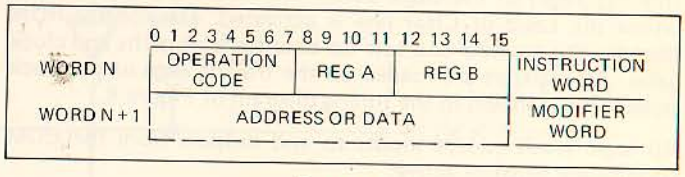


Figure 4
SIMPLE INSTRUCTION FORMAT

A ROM and counter combination will be used as the basis for the instruction execution control logic for our machine. To select and generate a timing sequence, the counter is set to the sequence start value and it is incremented for each sequence step. The ROM decodes each counter value to activate the appropriate ROM output lines for each step of the sequence. This type of sequence control is called Microprogram Control, since the sequence of operations is controlled by the contents of a memory, the control ROM.

Figure 5 shows details of the microprogram control for this machine. This includes an 8 bit ROM counter, a 256 x 24

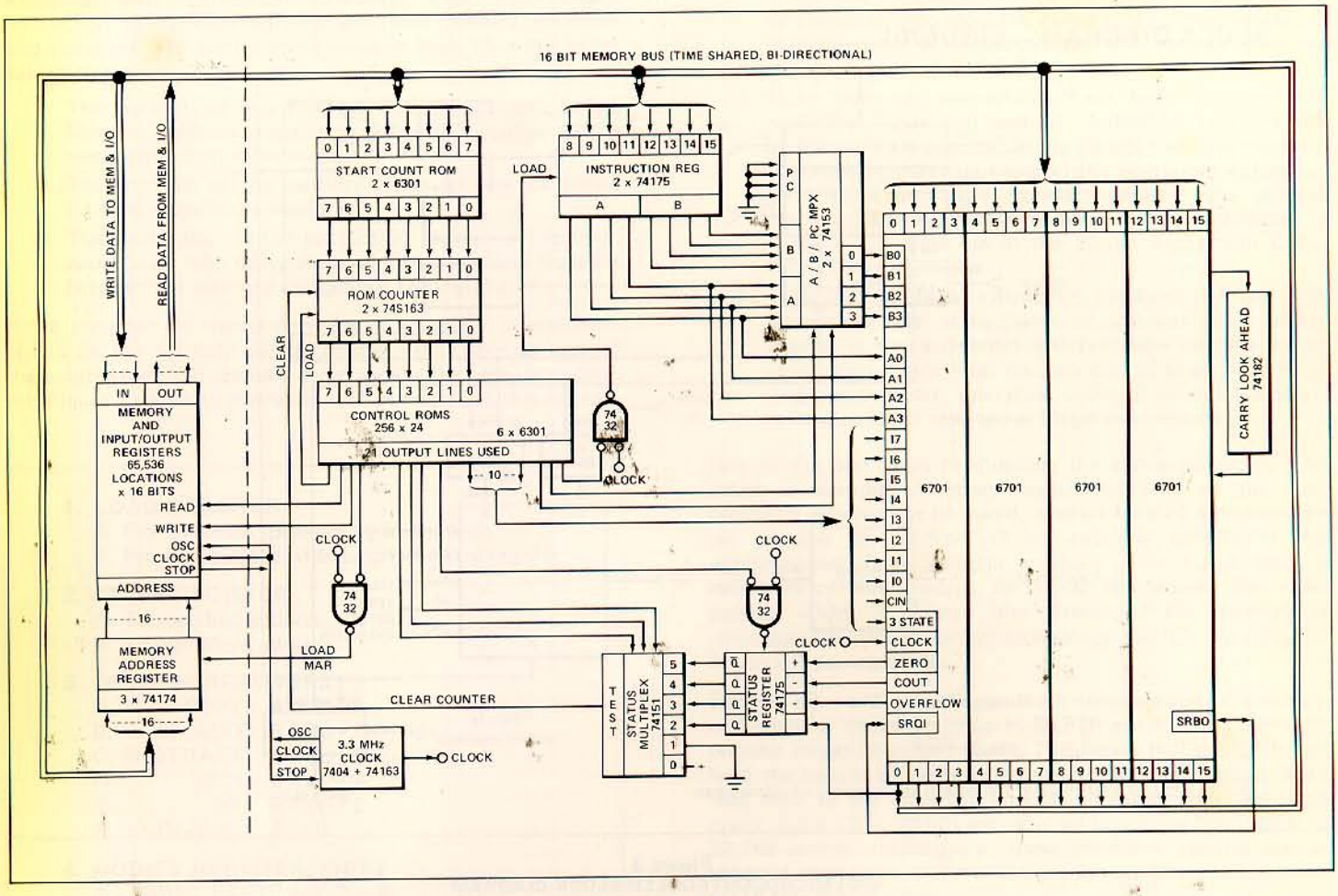


Figure 5
MICROPROGRAM CONTROL

control ROM, the Instruction Register, a 256 x 8 Start Count ROM, a multiplexer to select the A or B fields of the IR or the PC (register 15, all lows) for the 6701 B address inputs, a 4 bit status register, a counter control multiplexer, and a clock oscillator.

The 8 bit ROM counter is a 74S163 type. This counter increments, loads, or clears on the negative-to-positive edge (trailing edge) of the clock pulse. The counter will increment unless the Load or Clear line is activated. The control ROM decodes the counter and sets up data transfer paths and clock gates. All registers are loaded on the trailing edge of the clock pulse. This is shown in the timing diagram of Figure 8.

All logic levels except inputs to and outputs from the ROM counter are negative active:

logic 1 = +0.0 volts, logic 0 = +5.0 volts

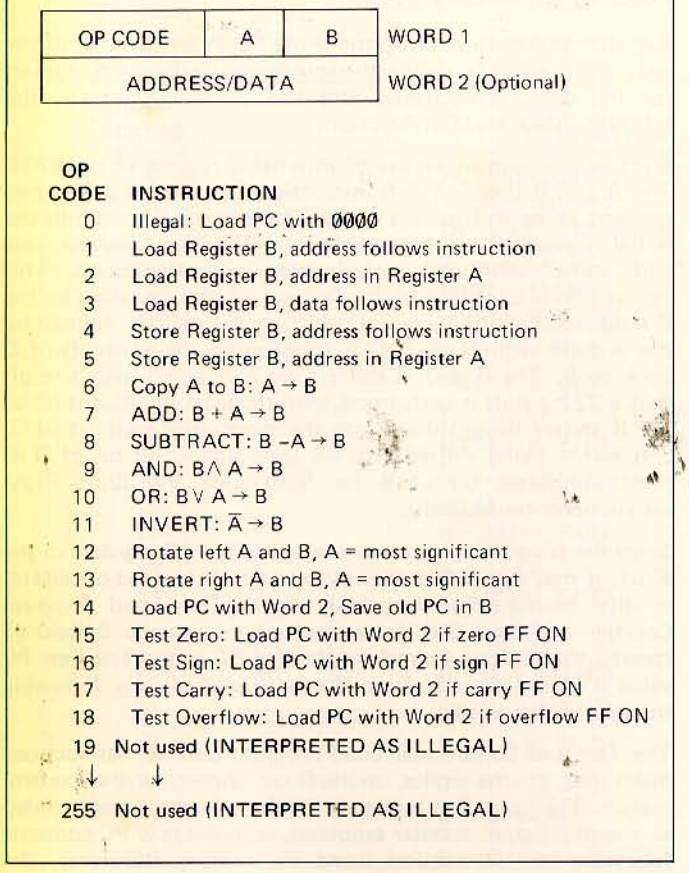
This provides three features:

1. Noise immunity for inactive high lines in TTL is much greater than inactive low.
2. When not in use or disconnected, the data bus lines will float to an all zeroes state and the control lines will float to the inactive state.
3. Open collector drivers can be used on the data bus, if pull-up resistors are provided.

Instruction fetch and execute in this machine proceeds according to the flow chart of Figure 7. The first two steps are common to all instructions and include the fetch and decode of the instruction. The instruction is decoded to select one of 256 possible execution sequences. This is done by decoding the first eight bits of the instruction to generate an 8 bit starting address for the execution sequence ROM. The execution sequence proceeds beginning at this address. After the last execution step, the ROM counter is cleared to zero and fetch of the next instruction begins. To illustrate this procedure we will fetch and execute a Register-to-Register Add instruction. This instruction requires three steps for instruction fetch and execute: Ø, 1, and 15. The ROM counter begins with a count of 0, State Ø:

State Ø—The contents of the PC, register 15 in the 6701s, are sent to the Memory Address Register (MAR). The contents of PC are also incremented at the end of the state. This is done by the Control ROM decoding the value of zero in the ROM counter and setting up the following:

a. The 6701 B input multiplexer is forced to all lows out, selecting register 15.
b. The 6701 is set to perform a B to output, B + 1 to B function. This will simultaneously gate out the old PC value and increment PC at the end of the cycle.
c. The 6701 tri-state drivers are gated onto the bus and the MAR load clock is enabled. Note that we have combined Steps 1 and 4 of our execution control logic sequence mentioned above.
d. The ROM counter will step from Ø to 1 at the end of the state.



OP CODE	INSTRUCTION
0	Illegal: Load PC with 0000
1	Load Register B, address follows instruction
2	Load Register B, address in Register A
3	Load Register B, data follows instruction
4	Store Register B, address follows instruction
5	Store Register B, address in Register A
6	Copy A to B: A → B
7	ADD: B + A → B
8	SUBTRACT: B –A → B
9	AND: B∧ A → B
10	OR: B∨ A → B
11	INVERT: $\overline{A}$ → B
12	Rotate left A and B, A = most significant
13	Rotate right A and B, A = most significant
14	Load PC with Word 2, Save old PC in B
15	Test Zero: Load PC with Word 2 if zero FF ON
16	Test Sign: Load PC with Word 2 if sign FF ON
17	Test Carry: Load PC with Word 2 if carry FF ON
18	Test Overflow: Load PC with Word 2 if overflow FF ON
19	Not used (INTERPRETED AS ILLEGAL)
↓	↓
255	Not used (INTERPRETED AS ILLEGAL)

Figure 6
SIMPLE MACHINE INSTRUCTION SET

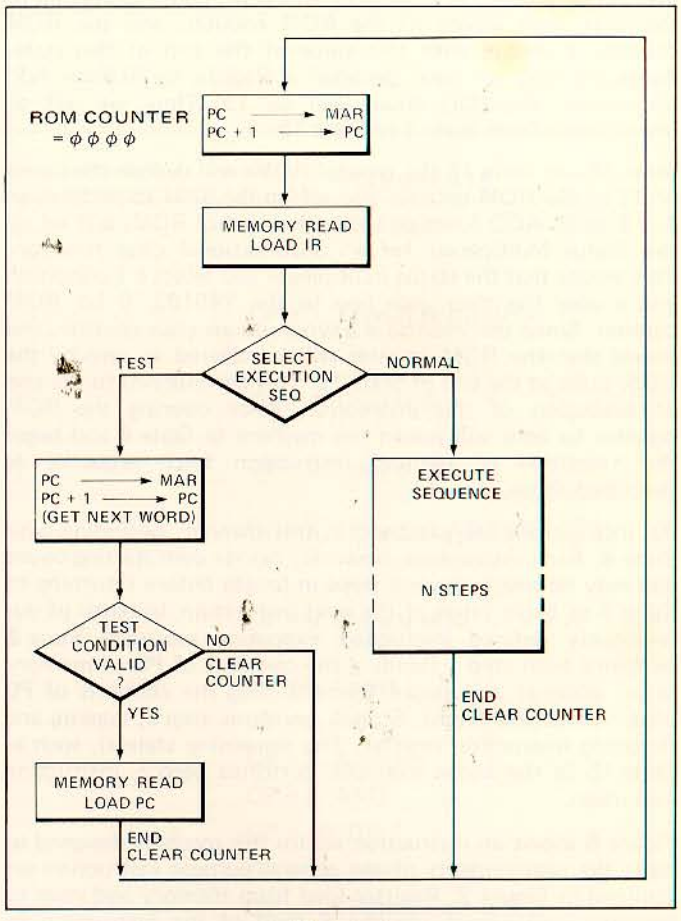


Figure 7
INSTRUCTION EXECUTION FLOW CHART

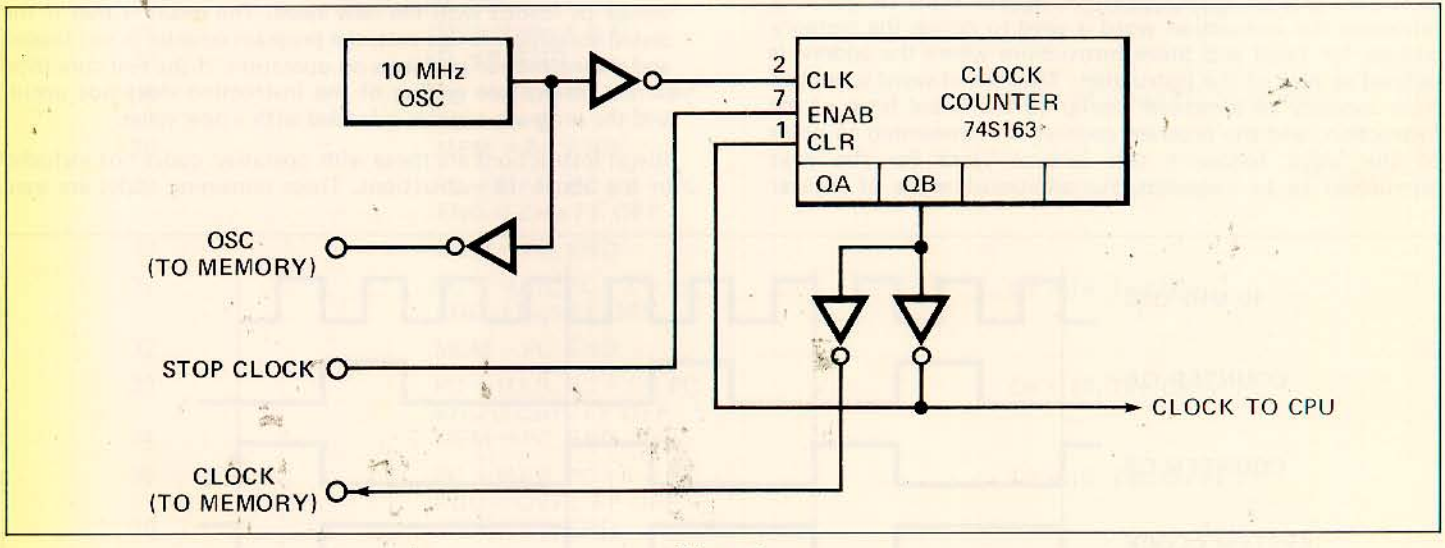


Figure 8
CLOCK CIRCUITRY

State 1—In this state the next instruction to be executed is loaded into the Instruction Register and decoded. The contents of memory at the location defined by MAR are gated onto the bus and the IR load clock is enabled. Instruction decode is done at this time by the 256 x 8 Start Count ROM. One of 256 possible instructions are decoded directly from bits Ø-7 of the data bus, corresponding to the 8 bit operation code field of the instruction. Only 18 of the 256 possible instructions will be decoded with the remainder decoded as illegal instructions. Since this is the only time that the instruction is decoded, no register is required for the operation code field of the instruction and it is decoded directly from the data bus.

The Start Count ROM decodes the operation code field of the instruction to generate an 8 bit start count which selects in

instruction execution sequence. This 8 bit count is applied to the data load inputs of the ROM counter, and the ROM counter is loaded with this value at the end of this state. Assuming that we have decoded a Register-to-Register Add instruction, the start count will be 15. Thus, we will go immediately from State 1 to State 15.

State 15—In State 15 the control ROMs will decode the count of 15 in the ROM counter and set up the 6701 to perform an A + B to B, ADD function. Also, the control ROMs will set up the Status Multiplexer for an unconditional clear function. This means that the status multiplexer will select a 1 condition and enable the clear gate line to the 74S163, 8 bit ROM counter. Since the 74S163 is a synchronous clear counter, this means that the ROM counter will be cleared to zero by the clock pulse at the end of State 15. This corresponds to the end of execution of this instruction, since clearing the ROM counter to zero will return the machine to State Ø and begin the execution of another instruction fetch sequence as described above.

All instructions are executed in this manner, beginning with State Ø. Each instruction, however, has its own starting count and may be one or several steps in length before returning to State Ø to begin fetch of the next instruction. In terms of our previously defined instruction execution sequence, State Ø performs both step 1 (sending the contents of PC to memory as an address) and step 4 (incrementing the contents of PC after instruction fetch). State 1 performs step 2, loading and decoding instruction register. The remaining state(s), such as State 15 in the above example, performs Step 3, instruction execution.

Figure 6 shows an instruction set for this machine designed to meet the requirements of the general purpose instruction set outlined in Figure 2. Register load from memory and store to memory instructions use the B field of the instruction to define the register to be loaded and stored. The A field defines the register to be used as the source of the memory address in calculated address instructions. A second word of memory following the instruction word is used to define the memory address for Load and Store instructions where the address is defined as part of the instruction: The second word is fetched from memory in a manner similar to the initial fetch of the instruction, and the program counter is incremented to point to the word following this address word for the next instruction to be executed. An additional mode of register load has been added: that of direct load of the contents of this second word into a register rather than using the contents of this word as a memory address.

Register combination instructions use the A and B fields of the instruction word to define the source and destination registers for the date, respectively, and includes copy, invert, add, subtract, AND, and OR functions.

Shift instructions make use of internal Q register of the 6701. The A and B fields of shift instructions are used to define two registers to be shifted as a single 32 bit word. This fulfills the initial requirement of being able to shift a single register, plus add some additional convenience and flexibility. This instruction is executed by loading the register specified by the B field into Q, simultaneously shifting the register defined by the A field with Q, and then transferring the contents of Q back to B. The B and Q shifters are connected with a result that a 32 bit shift is performed, with the least significant bit of the B shifter being shifted into the most significant bit of Q, and zeroes being shifted into the least significant bit of Q or most significant bit of B for Shift Left and Shift Right instructions, respectively.

Since the program counter is now one of the 16 registers in the 6701, it may be loaded directly by any of the load or register modify instructions described above. The Load Program Counter and Save Old Value instruction uses the B field to specify the register to receive the old PC value. The new PC value is loaded directly from the word immediately following the instruction word.

The Test and Conditional Load Program Counter instructions make use of the status multiplexer shown in the control section. The execution sequence for these instructions is coded as a load program counter sequence, with the new PC contents following the instruction word in memory. However, the status multiplexer is used at the appropriate point to terminate the sequence early if the tested status condition is not met, with the termination occuring just before the program counter would be loaded with the new value. The result is that if the tested condition is not met, the program counter is not loaded and the instruction performs no operation. If the test condition is met, premature ending of the instruction does not occur, and the program counter is loaded with a new value.

Illegal instructions are those with operation codes not included in the above 19 instructions. These remaining codes are used

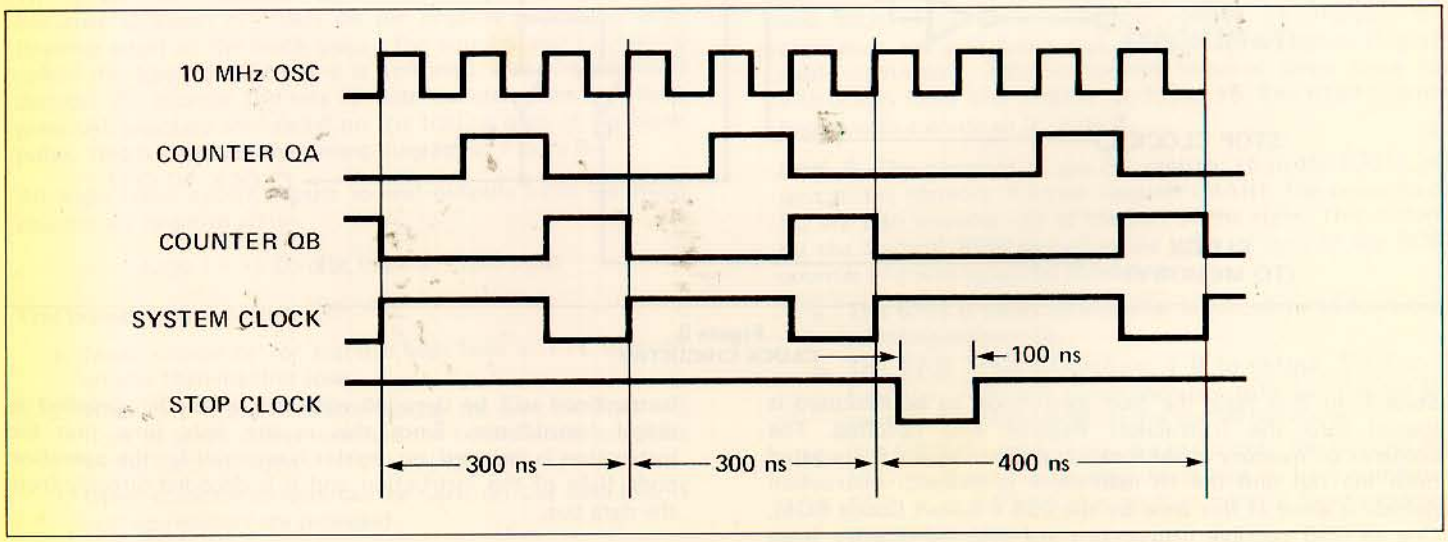


Figure 9
CLOCK TIMING DIAGRAM

ROM ADDRS	OPERATION	COMMENTS
Ø	PC → MAR, PC + 1 → PC	Instruction Fetch
1	MEM → IR Decode and Load Start Count	
2	PC → MAR, PC + 1 → PC	OP = 1, Load B @ ADDRS
3	MEM → MAR	
4	MEM → B, END	
5	A REG → MAR	OP = 2, Load B @ A
6	MEM → B, END	
7	PC → MAR, PC + 1 → PC	OP = 3, Load B Data
8	MEM → B, END	
9	PC → MAR, PC + 1 → PC	OP = 4, Store B @ ADDRS
10	MEM → MAR	
11	B → MEM, END	
12	A → MAR	OP = 5, Store B @ A
13	B → MEM, END	
14	A → B, END	OP = 6, Copy
15	B + A → B, END	OP = 7, Add
16	B – A → B, END	OP = 8, Subtract
17	B∧ A → B, END	OP = 9, AND
18	B∨ A → B, END	OP = 10, OR
19	$\overline{B}$ → B, END	OP = 11, Invert
20	B → Q	OP = 12, Rotate Left A & B
21	Shift Left A & Q	
22	Q → B, END	
23	B → Q	OP = 13, Rotate Right A & B
24	Shift Right A & Q	
25	Q → B, END	
26	PC → MAR, PC + 1 → PC	OP = 14, Load PC, Save Old
27	PC + 1 → B	
28	MEM → PC, END	
29	PC → MAR, PC + 1 → PC END if Zero FF OFF	OP = 15, Test Zero
30	MEM → PC, END	
31	PC → MAR, PC + 1 → PC END if Sign FF OFF	OP = 16, Test Sign
32	MEM → PC, END	
33	PC → MAR, PC + 1 → PC END if Carry FF OFF	OP = 17, Test Carry
34	MEM → PC, END	
35	PC → MAR, PC + 1 → PC END if OVFL FF OFF	OP = 18, Test OVFL
36	MEM → PC, END	
37	Not used	
↓	↓ ↓	
254	Not used	
255	ϕ → PC, END	OP = X, Illegal OP (Restart @ ϕ)

Figure 10
MICROPROGRAM

OP CODE	INSTRUCTION
18	LOAD REGISTER B: ADDRESS = (A) + WORD FOLLOWING INSTR
19	STORE REGISTER B: ADDRESS = (A) + WORD FOLLOWING INSTR
20	BYTE SWAP B
21	ARITHMETIC SHIFT RIGHT B: SHIFT IN SIGN
22	INCREMENT B
23	DECREMENT B
24	ADD CARRY
25	MOVE WORD: FROM ADDRESS IN A TO ADDRESS IN B INCREMENT A & B AFTER
26	PUSH B INTO MEMORY STACK: STACK ADDRESS IN A
27	POP MEMORY STACK INTO B: STACK ADDRESS IN A
28	EXCLUSIVE OR: B ∘ A → B
29	ETC. – – –

Figure 11
INSTRUCTION SET EXPANSION

WORD SIZE	16 BITS
ADDRESSING CAPABILITY	65,536 WORDS
MICROCYCLE TIME	300 NANOSECONDS
ADDRESSABLE I/O REGISTERS	65,536 SHARED WITH MEMORY
INSTRUCTION EXECUTION TIMES:	
Add 2 Registers	0.90 Microseconds
Load/Store Direct	1.50 Microseconds
Load/Store through REG	1.20 Microseconds
Test	1.20 Microseconds
PARTS COUNT (Excluding memory and MAR)	24 CHIPS
POWER REQUIRED	9.4 WATTS TYP (1.88 amps @ 5.0 volts)

Figure 12
PERFORMANCE SUMMARY

to select ROM address 255, the illegal instruction sequence. In this case, location 255 contains a restart sequence which clears the PC to 0000, and restarts the program at zero. If MMI 6301s are used for the Start Count ROM, uncoded locations result in an all highs, 255 output, which gives an automatic illegal instruction decode. It is common to reserve operation codes of all zeroes and all ones as illegal operation codes for:

1. If the PC is set to an address outside of existing memory, an all zeroes instruction will result.
2. If the PC is set to an address corresponding to a data area, it is probable that a small positive or negative number will be fetched with the first eight bits of the number being all zeroes or all ones, respectively.
3. The illegal instruction restart feature can be used as a manual restart by momentarily disabling the Start Count ROM so that an all highs output results.

Figure 8 and 9 show the clock circuitry and timing diagram for each microcycle. A Stop Clock line is provided which allows the clock to be stopped during the first part of any microcycle. This line can be used by a slow memory to force the processor to wait until it has completed its read or write cycle. It can also be used to implement single step operations in conjunction with some external step logic.

This instruction set implements the functions defined by the basic computer/memory controller instruction set of Figure 2, including some useful additions. Speed of execution of this machine will be limited by the settling time of the ROM counter and control ROMs, and the settling time of the 6701 for its various operations. A 300ns clocking interval is practical for this machine, and will result in a 900ns Register-to-Register add time for the first sequence described above, assuming a 150ns access time memory.

Figure 10 gives the full microprogram for this machine. Note that only 37 out of 256 possible steps were used to implement 19 out of a possible 256 instructions. Applying this microprocessor to specific applications will undoubtedly suggest additional instructions in order to do in one instruction an operation which would normally require several of the basic instructions. Some examples of instruction set expansion are shown in Figure 11. For example:

1. Indexed Register Load and Store instructions combine the contents of a specified register with contents of the word following the instruction to define the memory address for the data transfer.
2. A Byte Swap instruction, where a register is shifted on itself eight spaces, can be convenient for handling 8 bit character data, etc.

A performance summary is given in Figure 12.

At this point, it should be clear that instruction set expansion and customization quickly becomes application oriented; and a memory controller, like any other logic controller, eventually becomes oriented to particular application areas as the design becomes more sophisticated. The basic instruction set in Figure 4 provides a general purpose capability for the machine. The capability for instruction set expansion provides the option to add instructions to significantly improve overall controller performance in specific application areas. The straightforward design of this computer/memory controller can be improved in several ways. The general method of improvement is to reduce the number of states that are required to fetch and execute each instruction, and to add more powerful instructions (which may require additional hardware to support them) to decrease the amount of time required to perform a function in a given application.

By defining a computer as a memory controller, much of the mystery surrounding computer technology can be removed. It is simply a program controlled logic design that happens to have its program stored in the same unit that it is controlling. This also helps to clarify performance comparisons among various computers. The basic parameters are the size and speed of the memory being controlled, plus ease and efficiency of programming of the memory controller for a particular memory application. Programming ease and efficiency are related. Efficiency is defined by how few instructions are required to perform common tasks in a specific application. Ease is subjectively defined as how easy it is to remember and use the instruction set. Ease and efficiency are often combined by having a small, efficient instruction set uniquely designed for a specific application. For instance: Byte manipulation instructions for character processing, hardware multiply, divide, and floating point operations for scientific applications which require considerable classical mathematical calculation; etc.

The purpose of this article is to show you that computers, far from being mysterious and esoteric are just another example of a relatively simple digital system that happens to have a great deal of capability. So, if the microprocessor you are currently considering for inclusion in your product design is not a very efficient memory controller for your application, why not consider designing your own?!

mmi

Monolithic Memories, Inc., 1165 East Arques Avenue, Sunnyvale, California 94086, (408) 739-3535/TWX 910-339-9229.

Printed in U.S.A.